UPPSALA
UNIVERSITET

Benchmarking of option price solvers as a service in OpenStack

Applied Cloud computing project(1TD265)
Project Report by Team 4

Melker Forssell, Markus Skogsmo, Isak Borg Shuzhi Dong

October 31, 2019

Abstract

In this report, we present the integration of the existing project BENCHOP which is a collaborative financial computing project initiated by Computational Finance Research Group from Uppsala University to a Cloud Computing environment. By developing a cloud service we aim to support users in running the benchmarks. By using contextualization feature with Docker, this integration enables launching of the application as a public service in a cloud environment.

1 Introduction

The goal of this project is to implement a cloud based system as a service that enables users start different benchmarks of option price solvers provided by the BENCHOP project through a web-interface. BENCHOP provides a common suite of benchmark problems for option pricing [3], which is done to provide the finance community with tools that can be used both for comparisons between existing methods and for evaluation of new methods. This project is a prototype used to explore the possibilities, scalability and requirements of deploying to a cloud environment.

1.1 Project description

Due to licensing issues we can not utilize Matlab in our cloud based solution, so GNU Octave [4], which is an open source project and implementation of the Matlab programming language, has been chosen for this project in order to run the Matlab scripts provided by the BENCHOP project.

The different benchmark problems all utilise the Black-Scholes model. Black-Scholes Model is a pricing model used to determine the fair price or theoretical value for a call or a put option based on six attributes or variables: type of option, volatility, underlying stock price, time, strike price, and risk-free rate. The quantum of speculation more in case of stock market derivatives, and hence proper pricing of options eliminates the opportunity for any arbitrage. This model is used to determine the price of a European call option, which simply represents the option can only be exercised on the expiration date. In mathematical formulation [3], we let S represent the actual (stochastic) asset price realization, s is the asset price in the PDE formulation, t is the time, r is the risk free interest rate, σ is the volatility, W is a Wiener process and u is the option price.

SDE-setting:

$$dS = rSdt + \sigma SdW$$

PDE-setting:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sigma^2 s^2 \frac{\partial^2 u}{\partial s^2} + rs\frac{\partial u}{\partial s} - ru = 0$$

On the whole, with different numerical method, different option types can be solved. In our application, benchmarks are performed by European call option(Problem 1a), American call option(Problem 1b) and Up-and-Out option(Problem 1c). In our app.py file, we can choose which problem to run as an argument when called.

Three numerical methods, the COS method, Finite Difference (FD) method and RBF-FD method have been calculated in our application.

2 Model

Our model consists out of three docker containers. The first container has the Celery [1] tasks and the Benchop files and starts a celery worker once executed, the second container includes the Flask API [2], which allows us to call the service after deployment to start the different tasks, and the last container is a standard Redis image acting as both our broker and backend server. Both the Celery image and the Flask API are built locally and then pushed to Dockerhub. The service is then orchestrated in Docker Swarm through Docker Stack.

When the Docker stack initializes, the three containers are pulled from Dockerhub. The Worker, Redis and the Flask API are each initialized as a service which can be scaled using replicas. Docker Swarm will split the services among the created nodes which mean that this solution can easily be scaled both horizontally and vertically. When a GET request reaches the Flask app it will be passed on to the Redis broker which distributes the work to the Celery service and the Swarm will distribute the work to an available Celery container. Then, when the result is received, it is sent back through the Redis backend to the Flask app as can be seen in Figure 1. By using the network driver overlay the Docker Swarm will handle all the communication within the swarm, so the only way to communicate with our swarm is by using port 5000 which sends requests to the Flask app. We choose to only scale the worker since we saw no point of scaling the other services.

3 Results

We have created a service which can be launched and deployed in a cloud environment. It has so far been test-run in both the SNIC science cloud and AWS and here it can be scaled to as many machines as needed or required. We have created a simple interface to allow users to start the different benchmark problems and receive the calculated results.

4 Conclusion and Discussion

We have experimented and shown that the BENCHOP project can be made readily available to a lot more users by integrating it on a Cloud Computing environment. If our project were to be further developed and adapted it could lead to an even greater availability to the different benchmarks, which in turn hopefully would lead to easier development and assessment of both old and new finance methods.

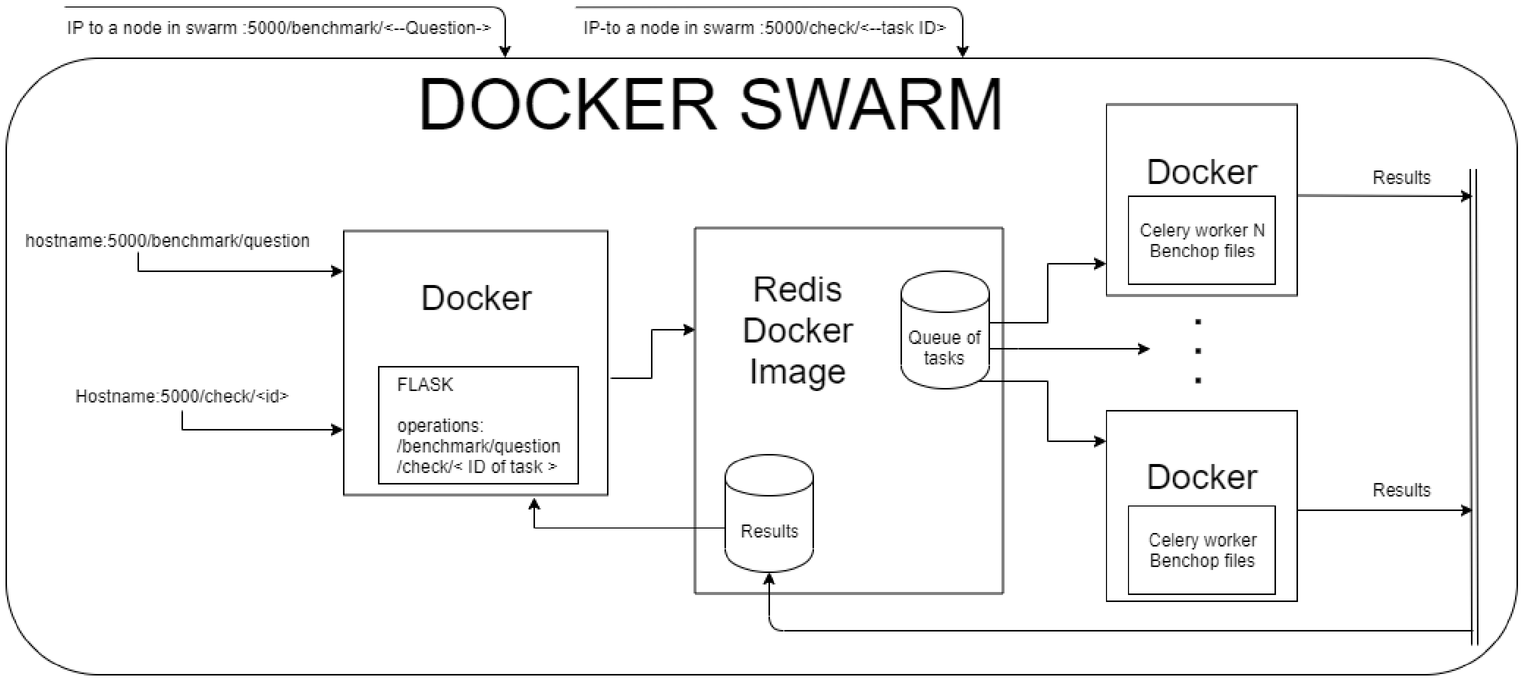

Figure 1: Overview of our project model.

4.1 Problems

During the work on this project we faced only a few problems. The major and most hindering of these problems were the fact that the SNIC science cloud became very hard, to nearly impossible, to connect to around the same time that we were given the project description. This made it very difficult for us to deploy and test our project. In the end we managed to verify that the project does indeed work on both SNIC science cloud and AWS EC2, and that it can scale the solution to as many different machines as needed, but this hurdle with the SNIC science cloud complicated and slowed down the work progress.

Another problem that caused a significant delay was the fact the were unable to install, and therefore utilize, oct2py in order to run the BENCHOP problems. This was solved by using python to open a shell on the workers, and in this shell Octave can be run to execute the BENCHOP code. This also means that the data is returned as a string instead of a table.

4.2 Future work

To further improve this model we could use Cloud Init to initialize the Swarm Manager and start the service autonomously. Heat could be utilized to join more nodes to the swarm, and once the nodes have joined the swarm the services could then be scaled and split among the nodes to decrease the run time and load on each individual node.

The interface could also be further developed to give the user a smoother user experience. The tasks themselves could be adapted to allow the user to pass their on variables in order to setup different scenarios for comparisons between methods.

References

[1] Celery documentation. Celery. Accessed 2019-10-20. [Online]. Available: http://docs.celeryproject.org/en/latest/reference/

[2] Flask API documentation. Flask. Accessed 2019-10-27. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/

[3] L. Höök, E. Larsson *et al.*, "BENCHOP—The BENCHmarking project in Option Pricing," *International Journal of Computer Mathematics*, 2015. [Online]. Available: http://uu.diva-portal.org/smash/get/diva2:848689/FULLTEXT01.pdf?fbclid=IwAR268VCJxioxza8ARPOO9YAnm_OBjDHd5z9dnKdT1jJ3zEap1TE6oPc0ezQ

[4] GNU Octave. Octave. Accessed 2019-10-20. [Online]. Available: https://www.gnu.org/software/octave/