BENCHOP as a Service

Calculating benchmark problems providing useable comparison and evaluation of different methods in option pricing

Delsante, Arianna[*], Eklund, Joakim[*], Engberg, Alexander[*] and Nhat Pham, Minh[*]

[*]Applied Cloud Computing

October 27, 2018

Abstract

The aim of this project was to design and create a scalable cloud architecture running the benchmark as a service. With the implementation of flask, celery and docker we created a scalable benchmark as a service application for a user who wants to calculate different benchmarking problems with the help of different methods. The service created can easily be modified and enlarged depending on the needs of the user.

1 Introduction

The BENCHmarking project aim is to serve as a tool for the development of methods in option pricing, which involves trying to predict what will happen to the price of one or several options, the value is based on variables relating to underlying asset, option and level of interest rates. In this context BENCHOP provides a set of benchmark problems that can be used for comparison and evaluation of methods[1].

The aim of this project is to provide users with a cloud service that can be easily used to run those benchmarks. A cloud solution would speed up the evaluation of a benchmark due to its capability of running different solvers in parallel. The project aims to find a cloud solution able to provide scalability and contextualization when more nodes are added.

1.1 The Black–Scholes–Merton model

The Black-Scholes-Merton model was initially formulated in the Black-Scholes formula where the partial differential equation, which came to be called, the Black-Scholes equation was published which evaluates option price over time [2]. Robert Merton later published an understanding of Fischer Black and Myron Scholes work using mathematical stochastic processes that helped to formulate the BSM model [3]. Originally the Black-Scholes formula was formulated to price European call and put options. Since then the Black-Scholes-Merton model has been extended and developed to include other financial derivative markets and to take into account real-world features of asset price dynamics[1]. The model is widely used in the world of finance, and modified in ways that is adequate for its users.

The benchmark problems in the BENCHOP project, that have been chosen to be supported of the cloud service, is provided by the BSM model. The financial derivative markets that the benchmark problems are provided for are the European call, American put and Barrier call up-and-out. The benchmark problems that the cloud service provide, allow users to calculate the BSM model for one underlying asset, with discrete dividends and with local volatility for the European call option. The parameter that the user can modify is the r parameter which is the risk free interest rate. Users of the service then have the possibility to either calculate one of the benchmark problem, returning the time to execute and the real error, or calculate all of three benchmark problems, returning a comparison of the time to execute and the real error between the problems.

1.2 Technologies used

Various types of technology have been used throughout the development: Flask is used to establish a connection between web application and virtual machine, Celery is used to develop distributed task queue making use of the message broker RabbitMQ, an intermediary that allows message passing, Docker containers are implemented to perform different tasks independently. We will focus on the use of the different technologies in the following subsections.

The goal for the project is to use these technologies to develop a cloud solution for solving a problem that is not cloud related. The reason behind this is that, with a cloud implementation, we think is possible to achieve a great speed up in the evaluation of the benchmark since multiple solvers will run in parallel. Furthermore, a cloud solution will be scalable, making it possible for the system to handle increasing loads. At the same time the proposed solution enables different users to perform tasks simultaneously.

Another improvement brought by the solution proposed here is the possibility of evaluating and developing benchmarks without the need to install MATLAB in the machine.

1.2.1 Flask

Flask is a microframework, it has little or no dependencies from external libraries but it provides tools and libraries to build a web application. Its only dependencies are: Werkzeug, a WSGI utility library which provides a protocol for Python to communicate with a web-application that is not in the common gateway interface and jinja2 which is a template engine used to set a template engine for each page in a website [4].

Our aim is to use the Flask Rest API to connect to the virtual machine. This API allows to request specific services through HTTP, it is a web service based on the REST architectural style, where no new standards or frameworks need to be created because specific HTTP features are used. The API divides a transaction into small modules, each addressing part of the underlying transaction, this implies a lot of flexibility from the developer point of view along with challenges when designing from scratch. To restrain the complexity, the project uses a model provided by OpenStack Swift, this architecture includes a proxy server and storage nodes; the proxy server implements a Swift REST-based API that allows communication between clients and storage servers over HTTP making possible to store and retrieve objects and their associated metadata from the cluster . Currently RESTful APIs are the clear choice when a cloud service is exposed to web services. In fact RESTful APIs are used by Amazon, Google, Linkedin and Twitter [5].

1.2.2 Celery

Another technology used in the project is Celery. It is a distributed task queue that operates with distributed message passing. Celery normally works asynchronously, if not differently designed, connecting task objects to an Advance Message Queuing Protocol (AMQP). The advantages of using Celery lie in the possibility of having the task queue hosted either locally or externally. Each Celery workers, in the same way, can be hosted locally or in another or multiple hosts. A listening worker as soon as is available picks one of the tasks in the queue, processes and executes the task returning the result to the celery client which is initialized into the application [6].

In the context of this project Celery is used to implement multiple workers that handle the tasks. The Celery instance AppCelery is created in a dedicated module referred as tasks.py, then imported to be used in the project.

When creating the instance the backend argument specifies the result backend to use, since we want to keep track of task state and results, the RPC result backend is used in the project. The broker argument specifies which broker is in use, in fact Celery requires a separate service to send and receive messages.

1.2.3 RabbitMQ

As mentioned previously, Celery requires a message broker, that is, an intermediary for messaging, in order to send and receive messages. In this project we will be using RabbitMQ as broker.

Through RabbitMQ, applications can connect to each other or to user devices data. RabbitMQ is easy to deploy in the cloud and can support multiple messages protocols. Moreover it is less susceptible to data loss in comparison to other brokers like Redis [7].

The choice of RabbitMQ lies also in his reliability, in fact, it offers different features to independently manage the optimal trade-off between performance and reliability. another important feature for our project is the ability of mirroring queues across several machine in a cluster, keeping the transmitted message safe in case of hardware failure.

In the first step a RabbitMQ user ACC15 is configured along with the virtual host myvhost. Permission are set so that the user can access the virtual host.

1.2.4 Docker

Docker is an engine that has been created to facilitate the creation and deployment of application through the use of containers, those are standard units of software that make it possible for an application to run in different computing environments. This is done by gathering together the code and its dependencies. With docker is possible to manage multiple containers on a single machine. The user has full control on a container thanks to a process-level API that allows to set the container environment and other primitives.

In this project Docker containers are used by a slave virtual machine where each Celery worker in the container can perform tasks independently, those tasks are calculation of the benchmark problem. Calculations are done using GNU Octave and sending back the information to RabbitMQ.

2 Method of approach

In this section, a description of how the project has been formed and how the group has chosen to approach the problem will be presented. The group of four decided to partition the time into two different time slots. The first part of the time was going to be spent getting to know the software and the different technologies that were going to be used and how they interact with each other and the user. In addition to this, a simple working design of the architecture was going to be created. A design that would act upon requests from a user, calculate a benchmark and give it back to the user. The second time slot was completely dedicated for further alterations of the simple model. During this time the group were to focus on scalability, contextualisation and user interface to mention a few things.

2.1 Architecture

In general, the group has decided to visualize the Matlab code and its functions as a black-box where we haven't analyzed the code more than necessary to implement it as a cloud service.

To implement and run the code within the cloud architecture the Matlab file "Table.m", which runs the benchmark, needed to be altered. In this file, we implemented a function so the file and its content can be called using GNU Octave from the cloud environment. In addition, this enables us to choose both ingoing and outgoing parameters which are usable when returning answers to the users and when modifying the service to add more functionality or when the user only wants to calculate the benchmark for one problem. In the Matlab function Table() we have partitioned the different problems and separated them from each other in order for the user to only calculate one at the time. In addition to partitioning the problems, we also deleted the three last problems, problem 4, 5 and 6. This was done due to the long execution time and with our objection to creating a well-functioning cloud architecture, we decided to exclude those parts of the code.

We have enabled the user to alter the parameter r which represents the interest rate. When running the application we noticed the last method, RBF-FD (Radial basis functions generated finite differences with BDF-2 in time), returned nothing but NaN as result from every benchmark problem. This is a result of the GNU Octave calculations because the framework can't handle executions in the same way as Matlab. This made us remove it completely from the execution which only leaves the COS (Fourier method based on Fourier cosine series and the characteristic function) and the FD (Finite differences on uniform grids with Rannacher smoothed CN in time) methods left. However, it is easy for the user to add methods they want to calculate the problems. The only thing to do is to add the method as a new folder in the BENCHOP folder on Github.

The cloud architecture that the group designed is constructed as Figure 1 below. Due to the fact that the application should act as a service to the user, it needs an easy interface in order to request the calculations being done, the user should not be able to have any previous knowledge about cloud computing and the different parts needed. Therefore the user uses the web and a HTML-page to connect with the application and request the calculations.

The connection to the master virtual machine is done with Flask rest API and it provides the master virtual machine with all settings and preferences made by the user. The system will use Rabbitmq to act as a message broker between the different components along with acting as a back-end to save results. The master virtual machine is connected to a slave Virtual machine that has multiple Docker containers running within it. In each Docker container, there is a Celery worker ready to calculate the benchmark problem.

When the master receives tasks to calculate from the user it distributes it to the different workers who calculate the benchmarks. The result is then sent back to the Rabbitmq backend. The result is then sent back to the user and gets displayed in the HTML-window.

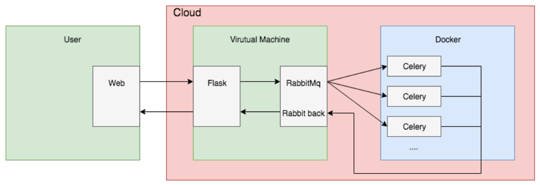

Figure 1: A description of the system architecture

2.1.1 Simple design of the system

The simple system created in the beginning was as simplistic as it could get while still using all the frameworks, except Docker containers, and returning a calculated result to the user. This model enabled us to learn about the different frameworks and how they should be connected in order to work which was of great help when working on the more advanced architecture. This simple design also acted as a base upon where we could develop a more advanced and scalable version of the architecture.

2.1.2 Advanced settings

To enhance and simplify the experience for the user an HTML-interface was created. The interface provides the user with a simple way to choose parameters, problem and execute the calculations without having to comprehend any prior knowledge about the application. The window also displays the benchmarks after it has been calculated.

To further enhance the benchmark calculation the user has the possibility to choose another parameter value of r (interest rate) rather than using the default values provided by the initial problem. This enables the user to in a greater extent modify the calculations to mimic real life situation.

In order to speed up calculation time, the problems in the Matlab code has been separated. This enables the application to partition the workload when calculating the different benchmarks thus reducing the overall calculation time. This is done by assigning the different benchmark problems to different workers who are able to work and calculate simultaneously. When the benchmark calculations are done they all send back their results to the user via the master. This also enables the application to calculate benchmarks for different users at the same time, the effectiveness of the system will then be based on how many workers there are available at the moment and how many incoming requests there are.

Contextualization Contextualization defines as the autonomous process of integrating components in an application into its deployment context and as a result providing the appropriate software and hardware environment for the successful execution of an application [8]. In our application, the start of new containers to hold celery workers is contextualized. This means that whenever the system needs to be extended a new worker can easily and quickly be deployed, making the application adaptable for future alterations and optimization. The contextualization also ensures that all deployed workers work in the same environment and in the same exact way, ensuring reliability of the architecture.

3 Analysis & Conclusions

The present solution to the BENCHOP as a service developed by the group works as the predetermined plan. We have managed to create an architecture that works in the intended way that is based upon the first system design and satisfyingly handling calculations of the benchmark problem.

In the project we divided the workload between multiple Celery workers, this improved performances significantly. First we tried to implement a single worker, subsequently we added up to three nodes achieving a good speedup.

The system is highly responsive, to achieve this we tried to make sure that all the available resources were used in the best possible way, for example if the user wants to run all three methods in parallel then one worker performs the computation for one methods and a second worker will handle the rest, if we assume two active workers. In other words, tasks are dynamically allocated between various workers given the workload.

We therefore think that the proposed solution is suitable for the BENCHOP service, in fact our cloud implementation serves as an extension which increases performances and gives scalability to the system.

3.1 What could have been done different

There are a number of things that the group could have done differently when it comes to the configuration and architecture of the Benchop cloud service. The things holding us back was mainly three things: time, available resources and knowledge in the field of cloud computing. With a little bit extra of everything stated above, we could have built an architecture more adaptable and scalable that could handle user requests in a greater way. We could also, with a bit more time and resources, have implemented more methods used for calculations of the different benchmark problems. This would have provided the user with more choices when comparing the different methods and more data to analyze which method works better on which problem. Also, we could have extended the number of changeable parameters. Giving the user the opportunity to specify the option information to a greater extent to resemble a real-life example.

Concerning the design of architecture, there are always improvements that can be made with regards to contextualisation and orchestration. One example is the automatic booting and deletion of worker containers when there are more/ fewer tasks. To make it as good and scalable as possible the service should be able to sense the incoming tasks and scale the system to the extent that it can handle all requests without significantly increasing the execution time.

References

[1] von Sydow, L., Höök, L J., Larsson, E., Lindström, E., Milovanovic, S. et al. *BENCHOP-The BENCHmarking project in Option Pricing.* Last Access October 26, 2018. Available at: `https://pdfs.semanticscholar.org/38b8/31fde5dd0a218c4479fbc713fc6593449c14.pdf`.

[2] F. Black and M. Scholes *The pricing of options and corporate liabilities*, J. Polit. Econ. 81 (1973), pp. 637

[3] R.C. Merton, *Theory of rational option pricing*, Bell J. Econom. Man. Sci. 4 (1973), pp. 141–183.

[4] Flask. Last Access October 26, 2018. Available at: `http://flask.pocoo.org/`.

[5] Flask-RESTful - Flask-RESTful 0.3.6, Last Access October 26, 2018. Available at: `https://flask-restful.readthedocs.io/`.

[6] Celery - OpenStack wiki. Last Access October, 2018. Available at: `https://wiki.openstack.org/wiki/Celery`.

[7] First Steps with Celery — Celery 4.2.0 documentation. Last Access October 26, 2018 . Available at: `http://docs.celeryproject.org/en/latest/getting-started/first-steps-with-celery.html`

[8] Armstrong D, Djemame K, Nair S, Tordsson J, Ziegler W (2011) *Towards a Contextualization Solution for Cloud Platform Services. In: Cloud Computing Technology and Science (CloudCom)*, 2011 IEEE Third International Conference on. IEEE. pp 328–331