Annales Mathematicae et Informaticae
Accepted manuscript
DOI: https://doi.org/10.33039/ami.2023.08.002
URL: https://ami.uni-eszterhazy.hu

Evaluation of scalability in the Fission serverless framework

Balázs Fonyódi, Norbert Pataki, Ádám Révész

Department of Programming Languages and Compilers,
Faculty of Informatics,
Eötvös Loránd University,
Budapest, Hungary
fonyodi1balazs@gmail.com
patakino@elte.hu
reveszadam@gmail.com

Abstract. The efficient code execution often requires concurrency, so many programming languages, libraries and framework aim at parallelism. Based on the granularity and abstraction level, many approaches of concurrency are available. However, concurrency carries difficulties but modern ways try to make it more convenient.

A rather new solution is cloud computing which enhances the concurrency in a way that standalone virtual machines utilize the shared hardware. Containerization takes advantage of lightweight virtual machines called containers because they use a shared kernel of the operating system. Conteiner orchestration (e.g. Kubernetes) enables containerization among multiple hosts. Serverless programming supports container orchestration for individual function so every trigerred function may run in a different container which is inside a cluster of hosts.

In this paper, we briefly present the modern cloud computing ways of concurrency. This subtle distributed approach requires a comprehensive evaluation. We take advantage of the open source Fission serverless framework and implement some distributed algorithms in this realm using the Python programming language. For a deeper comprehension, we measure and evaluate the scalability of Fission framework and the entire system. We execute the distributed algorithms with different sizes of input and we fine-tune the configuration of the Fission framework.

Keywords: Function-as-a-Service, serverless, Fission, distributed algorithms

AMS Subject Classification: 68W15 Distributed algorithms

Submitted: July 28, 2023
Accepted: August 7, 2023
Published online: August 8, 2023

1. Introduction

Parallelism and concurrency play an important role in high performance computing. Based on the granularity, one can choose multithreaded, multicore or manycore solution for a more efficient application [13]. Grid computing and distributed algorithms are also available for a long time. On the other hand, these approaches inflict many challenges (for instance, race conditions, deadlock, resource guarantees, etc.) [6]. Programming languages, libraries and frameworks have been proposed, but the developers are still eager for a convenient, elegant, safe approach which supports the efficient concurrent execution of the code.

In recent years, cloud native computing became one of the most dominant paradigms for building applications. This was sped up with Google releasing Kubernetes in 2014, and the Cloud native trend does not seem to slow down any time soon. In this rapidly evolving landscape, developers continuously seeking new ways to optimize their infrastructure while reducing the complexity. One of the newer forms of developing in this environment is called *Function-as-a-Service* (FaaS). It is a so called serverless computing model where developers only write and run individual functions in the cloud. One of the advantages is that to run these event driven functions, it is not necessary to manage and understand the underlying infrastructure. Another great thing about FaaS is the scalability and reliability. Most of the FaaS platforms, such as AWS Lambda, Azure Functions or even open source providers such as Fission provide automatic scaling capabilities. They can dynamically allocate and deallocate resources as needed by the number of incoming requests. These functions run in an isolated environment, meaning that each function runs in its own container or pod. This means that the functions cannot interfere with each other and they are not impacted by other processes and so the risk of failures due to conflicts are reduced. This approach results in a very sophisticated concurrency model.

The concurrency has some important questions that belong to the performance. An intriguing one is how efficient to launch a new computation. What is the cost of triggering a new subcomputation? What is the cost of the communication between the subcomputations? A distributed algorithm should perform better, but the algorithm improves the runtime only if these mentioned costs are cheap enough [14].

In this paper, we take advantage of an open source FaaS platform called Fission. We evaluate the concurrency aspect of this FaaS platform with the implementation and the execution of recursive, distributed algorithms. We focus on the scalability aspect of the performance. The cost of a triggered new subcomputation is based on boot of a Docker container and communication over HTTP, thus it is valuable to check.

The rest of this paper is organized as follows. We introduce the cloud-based approaches from Docker to FaaS in Section 2. We present the environment of the evalution in Section 3. We present the implementation details of the distributed algorithms in Section 4. We discuss the result of the evaluation in Section 5. Finally, this paper is concluded in Section 6.

2. Approaches of the cloud computing

Containerization has become an emerging approach in modern software engineering since it enables the shipping of the software artifacts and products with all required dependencies in a platform-independent way [2]. Containerization eliminates the virtualization costs of not used OS services and the kernel itself per container. Moreover, containerization supports isolation effectively since the containers are seem to be separate operating systems but they use a shared kernel [10].

The containers are lightweight and they enable the fast and simple deployment and configurations. However, this approach is limited only one host. Kubernetes is a container orchestration system which manages Docker containers over multiple Docker hosts [8].

Function as a Service (FaaS) is a category of cloud computing services that provides a platform allowing programmers to develop, maintain, operate, scale and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and deploying an application. This new abstraction approach eliminates further configuration and deployment cost. Building an application following this model is one way of achieving a "serverless" architecture [3]. This serverless programming approach provides the deployment of standalone function without launching any virtual machine or container [12].

Serverless programming is a rather new approach, however, there are real-world applications, for instance, Coca-Cola, Santander Bank and Expedia take advantage of this new paradigm [4].

Many frameworks are available for serverless programming, OpenFaaS, Kubeless and Fission to name a few open source tools [7]. Earlier, we defined our functional approach for the Kubeless realm [11]. However, it is still an open source artifact, VMWare has decided to stop driving and updating Kubeless [15]. Moreover, according to many aspects, Fission was evaluated as the most efficient serverless framework [9]. Furthermore, it has a wide language support and provides autoscaling which will be useful for measuring the speed of algorithms with different CPU settings.

3. Environment

3.1. Kubernetes

The environment of this research is created in a Kubernetes cluster. Kubernetes is the de facto standard in container orchestration systems. This paper might not be about Kubernetes, but there are some important terms that should be shortly introduced. Pods are the lowest level abstraction in a Kubernetes cluster. In this environment, a Docker container basically equals to a pod. It describes one or more containers in a shared network.

Our research focuses on the scalability of functions. These functions are running

in pods and for scaling, we need more of these. Kubernetes has a solution for that. A ReplicaSet is responsible for managing a set of identical Pods. When one creates a ReplicaSet, the number of replicas is specified, along with a Pod template. The template describes the specification of each Pod that the ReplicaSet should create and manage. A ReplicaSet ensures that a specified number of identical Pods are running at any given time by creating or deleting Pods as needed.

3.2. Fission

Fission is a Kubernetes native serverless framework. Fission can be deployed to any Kubernetes cluster whether it is on a private cloud or a private computer. Developers can write short lived functions in multiple programming languages, such as Go, Python, Java or NodeJS. These functions can be triggered with HTTP requests or other event triggers. Functions can be easily deployed using one specific command and the fast cold-start time ensures that the pods get ready quickly. Another feature of Fission is automatic scaling by CPU usage. This means that the system can create or destroy instances when needed, so it does not use unnecessary resources. Figure 1 is a flowchart that presents how Fission works inside a Kubernetes cluster.

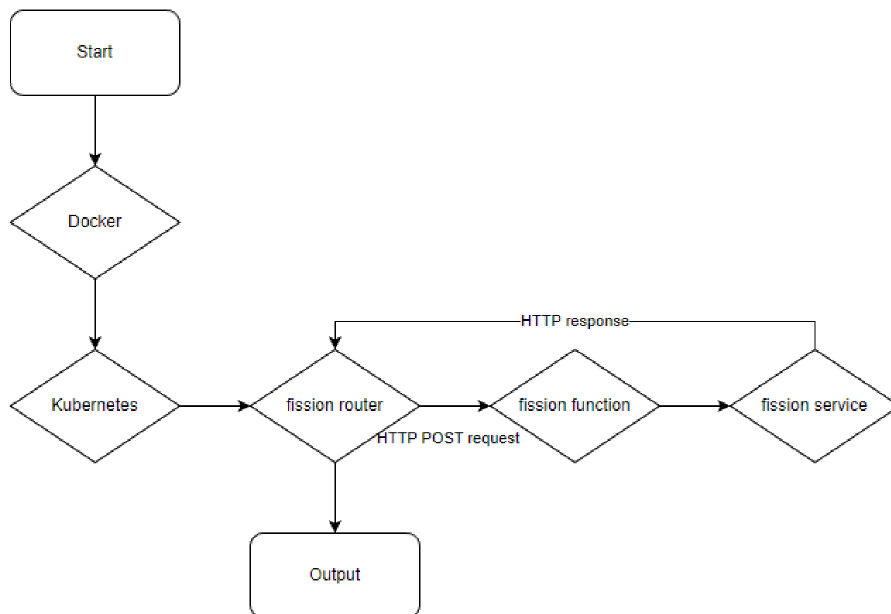

Figure 1. The workflow in our environment.

The router can send a HTTP request to a specific function which forwards to the Fission service. The service sends the HTTP response back to the router and the router creates the output.

4. Implementation and code overview

4.1. Karatsuba function

Karatsuba's algorithm is a fast multiplication algorithm which multiplies two numbers while reducing the number of recursive calls that is needed for the default multiplication [5]. This is achieved by splitting the two numbers into smaller ones, thus reducing them into subproblems in a divide and conquer manner and solving them recursively. The algorithm has a time complexity of $n^{\log_2 3}$ instead of n^2 for the traditional multiplication algorithm. The libraries needed for this version of code are Flask and Requests. The request subclass form flask enables the function to obtain the JSON data containing the two numbers, while the requests library allows the script to send HTTP requests to another instance of the same script to compute subtasks recursively. The function is split into three subfunctions for better readability. The main function handles the processing for the multiplication. This function obtains the two numbers with the `get_json()` from the request object, extracts the two numbers and passes both to the `karatsuba` function. The result from that is returned as a string. The `make_request` function takes the JSON data containing the two numbers and sends them to a URL as a post request. The `karatsuba` function is the part that handles the multiplication, it receives the two numbers and a string that represents the current recursion level. At first, the function checks whether the two numbers are single digit numbers. If they are, their product is returned, otherwise it computes the lengths of the two numbers, finds the maximum length, and splits each number into two parts of roughly equal length. The reason for this is to have three subproblems:

- Compute the product of the two upper halves of the numbers (`ac`)
- Compute the product of the two lower halves of the numbers (`bd`)
- Compute the product of the sum of the two halves of each number minus `ac` and `bd` (`ac_plus_bd`).

The function then returns the sum of the aforementioned subproblems, shifted accordingly to the required number of digits. The function has three recursive calls, but instead of handling these locally, each recursive call makes a HTTP request.

4.2. Merge sort algorithm

Merge sort algorithm developed by John von Neumann is a classic algorithm for sorting. The performance of this algorithm is typically evaluated in different concurrent situations.

This code also starts by importing the required Flask and requests modules. When the client sends a POST request to the '/merge' route, the Flask app receives it and triggers the main function. The application returns the sorted array as a JSON response. The sorting is done by sending POST requests to the same Flask

app to sort each half of the array in a recursive way, and then merging the sorted halves using a standard merging algorithm.

The Python source code of the merge sort algorithm:

```
from flask import Flask, request, jsonify
import requests

app = Flask(__name__)

@app.route('/merge', methods=['POST'])
def main():
    data = request.get_json()
    array = data['array']
    call_id = data['call_id']
    if len(array) > 1:
        mid = len(array)
        left = array[:mid]
        right = array[mid:]
        sorted_left = merge_sort_helper(left, f"{call_id}_left")
        sorted_right = merge_sort_helper(right, f"{call_id}_right")
        return merge(sorted_left, sorted_right)
    else:
        return jsonify(array)

def merge(sorted_left, sorted_right):
    i = j = 0
    merged_array = []
    while i < len(sorted_left) and j < len(sorted_right):
        if sorted_left[i] <= sorted_right[j]:
            merged_array.append(sorted_left[i])
            i += 1
        else:
            merged_array.append(sorted_right[j])
            j += 1
    while i < len(sorted_left):
        merged_array.append(sorted_left[i])
        i += 1
    while j < len(sorted_right):
        merged_array.append(sorted_right[j])
        j += 1
    return jsonify(merged_array)

def merge_sort_helper(array, call_id):
    json_data = {'array': array, "call_id": call_id}
    try:
```

```
        headers = {'Content-type': 'application/json'}
        response = requests.post('http://router.fission.svc/merge',
                                 json=json_data,
                                 headers=headers)
        return response.json()
    except requests.exceptions.RequestException as e:
        print(e)

if __name__ == '__main__':
    app.run(debug=True)
```

5. Evaluation

The test environment was running on a home setup, using Windows with WSL, Docker, Kubernetes and Fission. As of now, we only measured the runtime of each function with different sized inputs with the `time` command. The first number that is going to be printed out is the moment the user hits the Enter key until the moment the function is completed.

5.1. Scalability

We have already discussed the concept of ReplicaSets. Fission by default creates three pods for one function but first we scaled it only to one. The functions were called different sized inputs, Karatsuba's algorithm with a digit number of 10, 32, 64, 128 and 256 length numbers and the merge sort with an array size of 10, 64, 128, 256 and 512.

The code was implemented so that every recursive call starts a new instance, this way, the resources are divided and in theory for very large numbers the runtime should be faster. However, starting these pods have a cost, referred to as a cold start which is about 100ms [1]. When a function is triggered, Fission starts the predefined pods, so that time was not accounted for in the measurements. When a recursive call happened, a new pod was started. Karatsuba's algorithm has three recursive calls, so even for small numbers, at least 4 or 5 pods started running, which equals to about 0.4-0.5 seconds that was wasted. The bigger the number, the more recursive calls happened which slowed down the algorithm significantly. With more Replicas, Fission should share the resources. But because of the nature of the code, the execution still starts more and more instances, as the size of inputs increases.

5.2. Measurements and results

Evaluating the performance of merge sort, our theory that dividing functions this way might not be the best approach on a small scale system, seems to prove us right. Since merge sort only has two recursive calls, the number of instances being

started lowers significantly, thus the runtime of the function does not grow as much as with Karatsuba's algorithm.

Figure 2 and Figure 3 show the runtime of the analized functions and their runtime with different ReplicaSet configurations and with different input sizes.

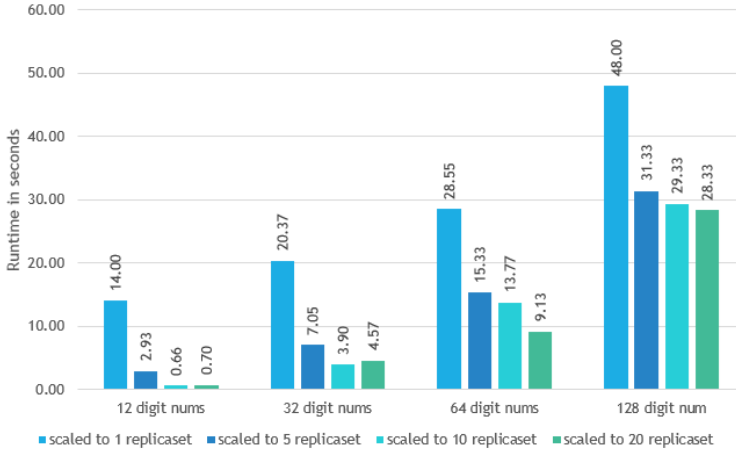

Figure 2. The runtime of Karatsuba's algorithm on multiple instances with different amount of data.

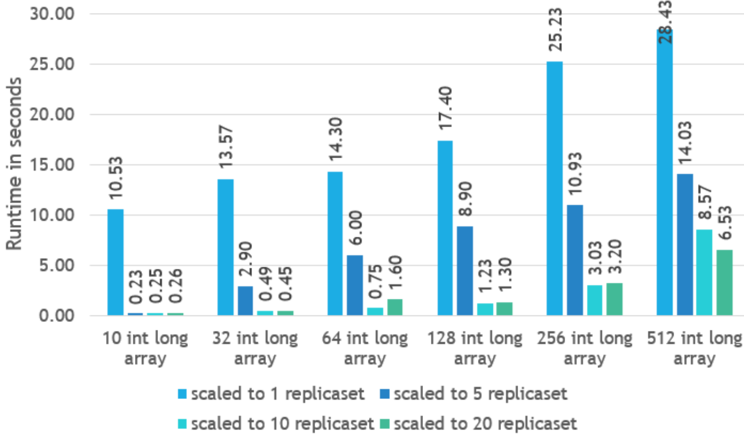

Figure 3. The runtime of merge sort on multiple instances with different amount of data.

An important thing to note is the starting number of the ReplicaSets. One would think that more ReplicaSets equal faster runtime, since Fission can divide the function into more resources. This is true until a certain amount of Replicas. However, from the Figure 2 and Figure 3, it is clearly visible that after a certain

number, the runtime does not decrease or might even increase a little bit. This can occur because it actually takes time to divide the function to these resources. Interesting to note, that sometimes the more is less. Starting more replicas does not solve the runtime issue, it can even slow down the function execution a bit.

6. Conclusion

Cloud native computing provides high-level abstractions for concurrency that is essential for an improved performance. These abstractions assist the developers in a convenient way. FaaS services allow to deploy, maintain and operate separate functions in a cloud using containerization and orchestration.

We utilize the Fission serverless programming framework and started to evaluate how this granularity of concurrency improves the runtime. We implemented two classical algorithms (merge sort, Karatsuba's algorithm) in a recursive manner using the Python programming language. We measured the runtime with different sizes of inputs and with different configurations of Fission. However, the comparison and evaluation are not comprehensive, so our future work focuses on a more detailed analysis. Right now, we found that our cases have a rather high cost of the new subcomputation's start and HTTP communication.

References

[1] D. Balla, M. Maliosz, C. Simon: *Open Source FaaS Performance Aspects*, in: 2020 43rd International Conference on Telecommunications and Signal Processing (TSP), 2020, pp. 358–364, doi: https://doi.org/10.1109/TSP49548.2020.9163456.

[2] D. Bernstein: *Containers and Cloud: From LXC to Docker to Kubernetes*, IEEE Cloud Computing 1.3 (2014), pp. 81–84, doi: https://doi.org/10.1109/MCC.2014.51.

[3] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski: *Serverless Programming (Function as a Service)*, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Los Alamitos, CA, USA: IEEE Computer Society, June 2017, pp. 2658–2659, doi: https://doi.ieeecomputersociety.org/10.1109/ICDCS.2017.305.

[4] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski: *The Rise of Serverless Computing*, Commun. ACM 62.12 (Nov. 2019), pp. 44–54, issn: 0001-0782, doi: https://doi.org/10.1145/3368454.

[5] X. Fang, L. Li: *On Karatsuba Multiplication Algorithm*, in: The First International Symposium on Data, Privacy, and E-Commerce (ISDPE 2007), 2007, pp. 274–276, doi: https://doi.org/10.1109/ISDPE.2007.11.

[6] W.-m. Hwu, K. Keutzer, T. G. Mattson: *The Concurrency Challenge*, IEEE Design & Test of Computers 25.4 (2008), pp. 312–320, doi: https://doi.org/10.1109/MDT.2008.110.

[7] K. Kritikos, P. Skrzypek: *A Review of Serverless Frameworks*, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 161–168, doi: https://doi.org/10.1109/UCC-Companion.2018.00051.

[8] V. Medel, O. Rana, J. Á. Bañares, U. Arronategui: *Modelling Performance & Resource Management in Kubernetes*, in: Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC '16, Shanghai, China: Association for Computing Machinery, 2016, pp. 257–262, isbn: 9781450346160, doi: https://doi.org/10.1145/2996890.3007869.

[9] S. K. Mohanty, G. Premsankar, M. di Francesco: *An Evaluation of Open Source Serverless Computing Frameworks*, in: 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2018, pp. 115–120, doi: https://doi.org/10.1109/CloudCom2018.2018.00033.

[10] Á. Révész, N. Pataki: *Containerized A/B Testing*, in: Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, ed. by Z. Budimac, Belgrade, Serbia: CEUR-WS.org, 2017, 14:1–14:8, url: http://ceur-ws.org/Vol-1938/paper-rev.pdf.

[11] Á. Révész, N. Pataki: *LambdaKube - A Functional Programming Approach in a Distributed Realm*, in: 2021 4th International Conference on Geoinformatics and Data Analysis, ICGDA 2021, Marseille, France: Association for Computing Machinery, 2021, pp. 67–72, isbn: 9781450389341, doi: https://doi.org/10.1145/3465222.3465233.

[12] Á. Révész, N. Pataki: *Stack Traces in Function as a Service Framework*, in: Proceedings of the 11th International Conference on Applied Informatics (ICAI) (Eger, Hungary, Jan. 29–31, 2020), ed. by I. Fazekas, G. Kovásznai, T. Tómács, CEUR Workshop Proceedings 2650, Aachen, 2020, pp. 280–287, url: http://ceur-ws.org/Vol-2650/#paper29.

[13] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, B. Esbaugh: *Parallelism via Multithreaded and Multicore CPUs*, Computer 43.3 (2010), pp. 24–32, doi: https://doi.org/10.1109/MC.2010.75.

[14] M. Tóth, I. Bozó, T. Kozsik: *Pattern Candidate Discovery and Parallelization Techniques*, in: Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages, IFL '17, Bristol, United Kingdom: Association for Computing Machinery, 2017, isbn: 9781450363433, doi: https://doi.org/10.1145/3205368.3205369.

[15] Q. L. Trieu, B. Javadi, J. Basilakis, A. N. Toosi: *Performance Evaluation of Serverless Edge Computing for Machine Learning Applications*, 2022, doi: https://doi.org/10.48550/ARXIV.2210.10331, url: https://arxiv.org/abs/2210.10331.