Implementing Cold-Start Reduction Techniques on Globus Compute

João Gabriel Lembo[1], **Alfredo Goldman**[1]

[1]Department of Computer Science – Universidade de São Paulo (USP)

{joaogabriellembo,gold}@ime.usp.br

Abstract. *This work addresses the cold-start delay issue within Globus Compute, a FaaS solution aimed at scientific workflows. We introduce the Warm-Start Engine, a novel engine for the system which integrates two techniques: an increase in the keepalive time of resources and a new task assignment algorithm. To validate our approach, we conducted a comparative performance analysis of the original system against two new versions, one with only the extended keepalive time and another with both modifications. The results demonstrate that the Warm-Start Engine reduces total workflow execution times by up to 62% and improves individual function execution times by as much as 82%, when compared to the baseline Globus Compute implementation.*

1. Introduction

Serverless computing [Castro et al. 2019] is a rising model in cloud computing for researchers and enterprises. In the model, the server's infrastructure is fully abstracted from developers, so that their only responsibility is the application to be executed.

The main implementation of this model is Function-as-a-Service (FaaS), which consists of the remote execution of functions on demand. In FaaS, developers submit their functions to a provider, which stores these functions and executes them upon user request. The provider also offers and manages all infrastructure and resources needed for function execution. When a function is invoked, the provider initializes the necessary components before it can execute it, and the time taken for this process to complete is named cold-start delay. This cold-start delay is a recurrent issue in serverless literature, with several articles identifying its great negative impact on performance and studying methods and techniques to reduce it [Ebrahimi et al. 2024, Basu Roy et al. 2023, Fireman et al. 2024, Liu et al. 2023, Bermbach et al. 2020, Bauer et al. 2024a].

Globus Compute [Chard et al. 2020], an open-source FaaS tool, is not exempt from this issue. Globus Compute is a FaaS tool designed to facilitate the execution of scientific workflows, by allowing any machine, from laptops to supercomputers, to be used as an endpoint for remote function execution, enabling users to run their processes on the most adequate machines. This requires the service to allocate resources and prepare the infrastructure to execute the functions, meaning that the cold-start delay is also present.

In this work, a new engine to Globus Compute is proposed, the Warm-Start Engine, which aims to reduce cold-start delay by trying to enforce that functions are executed on resources where they were previously processed, avoiding the need to install their dependencies, thus reducing initialization times. This work details all the steps for

the engine's implementation, as well as an experiment conducted to verify its effectiveness. The results shows that the Warm-Start Engine can decrease workflow execution times by up to 62% and can also reduce function execution and task completion times.

The paper is organized as follows. Section 2 outlines various approaches to the cold-start issue found in literature. Section 3 presents the architecture and operation of Globus Compute. Section 4 details the core of this work, the Warm-Start Engine. The experiment conducted to evaluate the performance of the new engine is exhibited in section 5 with its results analyzed in section 6. Finally, section 7 summarizes the work.

2. Related Work

Various studies have tackled the cold-start problem, either by analyzing the causes for it as well as proposing techniques and mechanisms to prevent it or, at least, reduce it.

[Fireman et al. 2024] devised a new technique, *Prebaking*, which reduces startup time of containers by capturing snapshots of function processes and containers, after they have executed required steps for function invocation, such as setting up the runtime environment. These snapshots can then be restored faster than executing all these steps again, reducing cold-start delay and allowing functions to start their executions sooner.

[Liu et al. 2023] proposed another approach, FaaSLight, which has the objective of minimizing cold-start delay by reducing the size of the code to be executed. By pre-analyzing the code to be sent to a FaaS service, FaaSLight distinguishes between necessary and optional code, sending only the former to the service, decreasing the time needed to load the code.

[Bermbach et al. 2020] introduced a new approach, that aims at decreasing the number of cold-starts that happen, instead of reducing their duration. Their idea is based on using application knowledge about the composition of functions to anticipate occurrences of cold-starts and initiate the infrastructure ahead of time to prevent them.

[Basu Roy et al. 2023] presented ProPack, a technique that packs multiple instances of a function into a single invocation, reducing the number of invocations needed, ultimately reducing the number of cold-starts needed.

[Bauer et al. 2024a] evaluated four container-building strategies to reduce cold-start delay in their initialization, which consist of different ways to install packages in the containers before function execution. The strategies include installing dependencies during container building, using Docker images with common packages already installed, and installing all packages after the container is ready.

Finally, in a more general approach, [Ebrahimi et al. 2024] conducted a systematic review on mechanisms for mitigating cold-start delay. In their work, they classified these techniques in four categories: application-based, which are modifications done to functions and dependencies; checkpoint-based, that create snapshots of containers to skip steps of initialization; invocation time prediction-based, where methods are used to prepare the infrastructure in advance; and cache-based, which preload libraries and files into containers. This work provides a extensive overview of the mechanisms available in literature, which can enable researchers to devise new methods and improvements for the reduction of cold-start delay.

These studies demonstrate that, although the cold-start issue is frequently addressed, research rarely focus on improving a service directly, but rather in system-agnostic techniques that function at application-level. While it is definitely a valuable and interesting approach, rendering promising results, it shows that there are opportunities for service-level mechanisms. Even though these techniques can be used to a lesser extent, being tied to a specific system, they also have the capability to be more efficient, since they can fully manage the internal workings of the chosen service. Therefore, studying cold-start techniques with focus on Globus Compute is not only a new opportunity of research, but can also produce valuable results and insights, which can further research on both the service and the cold-start issue.

3. Globus Compute

Globus Compute is a FaaS platform created by Globus, a cyberinfrastructure service aimed at research, from the University of Chicago and Argonne National Laboratory. With the emergence of FaaS, a service was needed to let researchers focus on their work, by abstracting responsibilities such as data management and software configuration. Consequently, in 2019, Globus Compute was created, with the objective of enabling remote function execution in scientific workflows, granting more versatility to researchers.

3.1. Overview

The basic premise of Globus Compute is to execute functions on the most adequate machines, distributed across a heterogeneous infrastructure. Once the software is installed, the machine becomes an endpoint, which means that it can receive functions to be executed, provided that the user is authorized to use that endpoint.

The functions to be executed are snippets of Python code that perform a certain task. They can be registered on the system, enabling them to be invoked using only their unique identifier, or sent directly as a parameter at the time of execution. Furthermore, all the dependencies have to be included in the function body to be imported upon execution.

Besides the software installed on the machine, Globus Compute also has a cloud service responsible for receiving the requests, storing registered functions, routing them to the correct endpoints and returning function results.

3.2. Software Architecture

In order to demonstrate the cold-start techniques implemented on the Globus Compute software installed in machines, it is necessary to explain its operation, detailing the process from the moment a task is received until its result is returned. To that end, it is important to first explain the architecture of the Globus Compute software, exhibiting the elements that constitute it and their responsibilities.

Upon installing the software on a machine, configuring it as an endpoint, and then starting it, a persistent process is executed, the Globus Compute agent. It is responsible for communicating with the cloud service, receiving tasks, adding them to queues and directing them for execution. This process also handles load balancing and resource allocation and distribution.

To manage these resources, the agent uses an engine which determines the rules for acquiring and releasing resources and for task distribution. Users can select the engine via system configuration, thus deciding resource allocation and task dispatch policies.

Another component of the Globus Compute system are managers, each handling resources of a single node of the endpoint, distributing them to multiple workers, which are the processes that execute the functions sent by the user. Managers also handle task dispatch to workers and act as a communication bridge between them and the agent.

3.3. Software Operation

With the system architecture and its three main components detailed, the flow of a task inside a Globus Compute endpoint is presented. Firstly, when the endpoint is started, the agent establishes the connection with the cloud service in order to receive tasks and initialize managers, which then establish the communication with the agent and may also start workers, concluding startup.

Then, when a function is sent from the cloud service to the agent, it is stored in a queue of received tasks. At frequent time intervals, the agent traverses the task queue, distributing them randomly managers with available capacity. Similarly to the agent, these managers also traverse their tasks queues, assigning tasks to available workers. At this point, managers might also start more workers to handle workload, or kill idle workers after a certain amount of time has passed.

The worker, in turn, executes the function immediately upon receiving it, returning the result after its completion. Each worker receives one task at a time, so there are no queues for it to manage. When it returns the result to the manager, it is also informing that it is available to receive a new task. The task result them undergoes a reverse process of communication: the manager sends it to the agent, which forwards it to the cloud service for user retrieval.

With the communication between the agent (and its engine), managers and workers established, the techniques devised in this work can be introduced.

4. The Warm-Start Engine

The objective of this work was defined as the implementation of cold-start reduction techniques on Globus Compute. This was done for functions without containers, since the service already has certain ways to avoid cold-start on containerized executions.

From the architecture of Globus Compute, the best way to create these techniques would be through a new engine, since they are responsible for execution resources (managers and workers). Furthermore, the original code would be preserved, since all new functionalities would be programmed in the new engine. The two techniques implemented to avoid cold-start are explained below. The repository containing these modifications is open-source and available at the *warm-start-engine* branch at `https://github.com/JG-Lembo/globus-compute`.

4.1. Increase in Keepalive Time

One of the most common techniques to avoid cold-start delay consists in keeping the infrastructure activated for a certain period of time after the execution of a function. Thus, any function invoked during this period can be executed without the wait for the initialization of the infrastructure. The period of time when the infrastructure stays activated is called keepalive time.

Globus Compute already employs this technique, keeping managers and workers activated for one minute after they were last used. This interval, though short, already brings benefits to the system, since the intervals between function invocations, as well as their execution times, are, in many workflows, shorter. However, an analysis on data of Globus Compute utilization [Bauer et al. 2024b] shows that a keepalive time of five minutes could avoid cold-start in 93% of all functions invoked on the service. As a result, the first step was to implement a keepalive time of five minutes, to assess its impact and verify these findings.

The keepalive time is managed in two parts of the code, separately. The first one is in the engine, through a specific strategy, an element of the engine responsible for verifying the workload of the service, and using this information to determine if new resources can be released or should be allocated. The other part where there is keepalive time management is inside managers, which store a value for maximum idle time of workers, so that workers which have not executed any functions for that amount receive a message from the manager to shut themselves down.

The presence of two types of keepalive time management requires extra caution, since altering only one of them would not bring the desired results. For example, if only the keepalive time of the strategy is increased, even though it would wait five minutes to release resources, it would not stop workers from shutting down after one minute, which would remove the benefit of warm-start for the last four minutes.

Combining this need with the possibility of allowing users to define their own keepalive time, rather than simply modifying both default values to five minutes, a new configuration variable was created, *keepalive_time*, with 300 seconds as a default. Then, both sections of the code where keepalive time is managed are started with the configured time value as one of their parameters, guaranteeing that both operate with the same time interval, thereby allowing the full benefit of the keepalive time chosen.

The version of the software that includes only this increase in keepalive time, without other cold-start reduction techniques, will hereafter be referred to as the *keepalive version*. Its goal is to measure the impact of the change proposed by [Bauer et al. 2024b] and to enable the assessment of whether new warm-start techniques are necessary and would have positive impact, or if simply increasing the keepalive time is enough to decrease the execution time of the system. In section 5, the performance of the keepalive version will be analyzed to evaluate the impact of increasing keepalive time to five minutes.

4.2. New Task Assignment Operation

After implementing the increase on keepalive time, the next decision was to add a new technique to avoid cold-start delay. In Globus Compute, if managers and workers are already active, the initialization time is reduced, since necessary resources are already allocated. However, this time is still influenced by the import of libraries required for a function to run.

In Python processes, which is the language used in Globus Compute, importing a library makes its contents be available as long as the process is running. Since this work discusses executions without containers, workers are pure Python processes that run on the machine used as the endpoint, and therefore maintain the contents of a library imported for a function execution. Hence, if a function is executed in a worker that

previously executed another one, and both share a certain library, it will already be loaded for execution. This means that the import time of the library will be eliminated from the initialization time, decreasing cold-start delay.

Given this Python behavior, it could be interesting to ensure that functions that need certain libraries are executed on workers that already imported them in a prior execution. The simplest way to achieve that, and thus ideal for a first evaluation of this type of technique, is to guarantee that a function, starting from its second invocation, is dispatched to a worker that already executed it.

Hence, the next step was to alter the method with which Globus Compute dispatches its tasks, both from endpoint to managers and from managers to workers. Since this two processes of task assignment are located in different parts of the code, and use distinct techniques, they were analyzed and modified separately, as outlined below.

4.2.1. Propagation of Function UUIDs

In order to ensure that the system knows when a function was already executed on a certain manager or worker, it is necessary that the task contains an identification for the function. Every function executed in Globus Compute has a unique identifier, which allows the service to retrieve the serialized body of the function, as well as verify its execution permissions. This identifier can be created at the time of function registration or during its execution, for functions without prior registration.

When the cloud service sends a task to the endpoint, it also provides the identifier of the function, enabling the endpoint to verify if it has the permission to be executed on this endpoint. Since this verification is done at the start of the execution flow and does not depend on any specific engine, this identifier is not propagated to the chosen engine, and consequently, it is not available to managers and workers.

Therefore, these processes cannot see the identifier of the function contained in a task, thus rendering it impossible to ensure that the function is sent to a resource that already executed it. As a result, it is required to propagate the identifier, enabling these processes to check for previous executions. In order to achieve that, it is sufficient to add this identifier as a parameter of the task submission to the engine.

To avoid changing code outside the Warm-Start Engine, a new method was created on the engine interface which simply calls the original submission, overriding it on the new engine to add this new parameter. Therefore, the engine has access to the identifier when assigning tasks to managers, as well as these have access to dispatch them to workers. As a result, it was possible to develop the new methods of task assignment.

4.2.2. Task Assignment To Workers

Since workers are the Python processes where functions will be executed, and are therefore responsible for library imports, the task assignment from managers to workers was modified first.

In Globus Compute, managers assign tasks to workers from a queue. When the

system is started, a manager activates a number of workers, which are added to a queue of available workers. As it iterates over the tasks, the manager assigns them to workers in a first-in, first-out (FIFO) manner, removing those which receive tasks. The workers, in turn, are inserted back in the queue when they return their results, indicating to the manager that they are available for a new execution.

In order to modify this behavior, enabling a function to be run on a worker where they were already executed, three new elements were required: a mapping from tasks to functions, so that the function UUID can be retrieved using the task UUID; a mapping from functions to workers where they were already executed, so the system identifies which workers have the necessary dependencies loaded; and a set of busy workers, which keeps track of workers that are actively executing functions.

The new operation of the system, using these three elements, is as follows. When the manager receives a task, it uses the function identifier to search for previous executions, through the function to worker mapping. If there is a prior execution, the manager iterates through the list of workers retrieved from the mapping, checking for an available one using the set of busy workers. When an available worker is found, the manager assigns them the task, inserting them into the busy set. However, if all workers from the mapping are unavailable, or if the function is being invoked for the first time, the manager resorts to the original behavior, getting the next worker of the worker queue, using the busy set to ensure it is available.

Hence, the approach used by the manager to attribute tasks to workers, prioritizing the ones which already executed their functions, is described. However, to ensure its effectiveness, it is necessary that the engine also prioritizes sending a function to a manager which controls a worker where it has already been executed. Thus, the task assignment from the engine to the managers was also altered, as presented below. In order to simplify the explanation, the expression ”manager already executed a function” will be used to indicate that a manager is responsible for a worker which already executed that function.

4.2.3. Task Assignment to Managers

The task assignment from the engine to managers is more complex than the previous one, since they can receive multiple tasks to distribute among their workers. In the original version of the system, in order to distribute tasks, the engine iterates over its managers, verifying their capacity to receive new tasks. If the engine detects that a manager is available to receive at least one task, it utilizes the task queue matching the manager's type to determine the tasks to be sent. The engine them assigns them one by one until the maximum capacity has been reached or the task queue is empty.

After the assignments, the engine composes, to each manager, a message with all tasks it has been assigned. From this moment onward, tasks are stored at each manager's queue, and are now their responsibility through the process described earlier.

This stage of the work consisted, basically, in creating a new dispatch algorithm, which considers not only manager capacity, but also the functions it has previously executed. The algorithm achieves that through a mapping between functions and a list of managers that already executed them, similar to the mapping from functions to workers

in the previous step.

The main difference of the new dispatch algorithm is that its core is the task queue, and not the manager list, with new version iterating over tasks, attributing them to the most adequate managers. After calculating the available capacity of all managers, the algorithm traverses the task queue of a type, searching for an adequate manager to receive it.

Firstly, the algorithm verifies the list of managers that already executed the function, choosing the first one that has available capacity. If the task cannot be assigned to any of them, the algorithm analyzes the remaining managers, until it finds one available. After these procedures, a task is guaranteed to have been assigned to a manager, since it is processed only if there was available capacity in at least one of the managers.

Through this algorithm, the engine can assign tasks to managers that already executed their functions. These managers, in turn, use the new dispatch method to assign them to workers which already executed them. Combining these techniques increases the likelihood that a function will be sent to a worker which already loaded its required libraries, reducing initialization times and, consequently, guaranteeing warm-start.

This version of the Globus Compute system, with increased keepalive time and new task assignment techniques, will hereafter be named the *warm-start version*. Thus, the two versions of Globus Compute developed during this work have been presented and detailed, the keepalive version and the warm-start version. The next step, therefore, was to design and conduct an experiment in order to compare the performance of these two versions with one another and with the original one.

5. Experiments with Warm-Start

After the implementation of the techniques for cold-start reduction, it was essential to analyze whether they reduced initialization times and, more importantly, total execution times. The reasoning is that it would not be advantageous for the system if initialization times were lowered, but the total time of a function execution was increased, due to overhead introduced by the new assignment methods.

It was also interesting to compare the performance of the keepalive version with the original, to measure the impact of the new time configuration, as well as with the warm-start version, to verify if the additional techniques also had an impact. This section presents how this experimentation was conducted, detailing its development and all tools required for its execution.

To design an experiment which adequately tests the two new versions of the system, there are certain requirements. Firstly, the experiment should include functions which import different libraries, since otherwise a function could show lower initialization times simply because another function previously executed shared the same dependencies. Some sharing, however, is also desirable, as real workflows often include functions that rely on common libraries such as *numpy*, *pandas*, or *scipy*. Then, the interval between function invocations should be short, given that after five minutes (one minute, in the original version), managers and workers are deactivated. Lastly, there should be a high volume of function invocation in the workflow, which could result in functions not being sent to the most adequate workers, due to lack of capacity.

To meet these requirements and also simulate realistic workflows, the Globus

team suggested the use of TaPS, a framework for evaluating task-based workflow execution. TaPS meets the three requirements for the experiment, since it offers benchmarks of distinct domains, guaranteeing that different libraries are imported, featuring high task volume and short invocation intervals, as desired.

5.1. Experiment Design

The proposal of the experiment was to simultaneously execute twp benchmarks, *Cholesky Factorization* and *Molecular Design*, choosing parameters that would result in approximately equal durations, so that they would run concurrently for the majority of the experiment. These simultaneous executions would be performed several times, allowing for a statistic analysis.

The objective of the experiment was to measure, mainly, the execution times of functions (and consequently, the impact on initialization times), but task assignment time was also recorded, as well as total task times, which is defined as the time between the task being received by the endpoint and the delivery of its result.

As for the configuration of the Globus Compute endpoint, it was decided to use two managers, each with two workers, enabling the verification of the operation of the new task assignment methods without consuming a large amount of resources.

Finally, two AWS EC2 instances were used for the experiment. The endpoint was a *t2.xlarge* instance, with 4 CPUs and 16 GiB of memory, ensuring that the two managers and four workers had the necessary resources for adequate operation of the system. The other one was a *t2.micro* instance, with one CPU and 1 GiB of memory, with the sole objective of triggering the execution of both benchmarks.

Both the number of managers and workers, as well as the size of the instances, were determined by financial limitations associated with using powerful instances on AWS. Future experiments can be performed on machines with more resources and larger quantity of managers and workers.

5.2. Experiment Execution

The execution of each round of the experiment was done through a Python script which created two threads, each one invoking one of the benchmarks. After each round, which comprised the full execution of both benchmarks, the timestamps for relevant events were extracted, which were then used to calculate important time intervals for each function.

To achieve that, time intervals were calculated for each task, including the durations of the task arrival, dispatch to the manager, dispatch to the worker and function execution. Moreover, the total time of the task execution was also derived, by calculating the time between the task arrival and the delivery of its result. Finally, the averages for each time interval were calculated for each function. These time intervals were then used to conduct a statistical analysis on the performance of the system, as detailed in the next section.

6. Results

The experiment resulted in 25 measurements for the original version of Globus Compute and 27 measurements for both the keepalive and warm-start versions. This relatively

small sample size is due both to the time required for each execution of the experiment and financial constraints that prevent that AWS resources are used indefinitely for multiple repetitions of the experiment.

Due to the small sample for each version, it is not possible to identify normality in the data. Therefore, in order to compare samples more adequately, a non-parametric test, the Mann-Whitney test [Conover 1999], was used, with a significance level of 5%.

Each function was analyzed separately, due to their different imports, execution times, number of invocations and results. For each of them, versions were compared in pairs, to verify which of them produced statistically smaller values or if there was no significant difference. If the test detects that one version yields smaller values, the version is considered to have performed better.

In this section, a brief discussion of the different behaviors and results obtained by the Mann-Whitney for each time-interval analyzed is presented.

6.1. Total Task Time

The total task time represents the full processing time of a task, from the arrival at the endpoint to the retrieval of the result. As a result, it is the most important aspect to be studied, since it represents how fast users receive results after invoking a function. Hence, it is the first metric analyzed.

For seven of the eight functions analyzed, the test detected that the warm-start version had the smallest times (tied with the keepalive version in 3 of those functions), while the keepalive version was also better than the original in six of them. For the remaining function, no statistically relevant difference was found between any of the versions.

Thus, it is possible to conclude that the keepalive version improved on the original, while warm-start enhanced it further, demonstrating the effectiveness of redirecting functions to managers and workers which previously executed them.

6.2. Execution Time

After the analysis of the total task times, it is interesting to observe function execution times, which should theoretically be most affected by warm-start. For this metric, the warm-start version performed best for all functions, followed by keepalive and then the original. The only exception was the *run_model* function from the *Molecular Design* benchmark, for which no difference was observed between the warm-start and keepalive versions.

This analysis indicates that the logic implemented by the warm-start version had the intended outcome, which was to lessen function execution times, by allowing functions to run on workers which already had their dependencies loaded.

6.3. Task Assignment Time

Lastly, the task assignment time is also a relevant metric, since it can demonstrate how the new task dispatch algorithm impacts the time taken to send a task to a certain manager.

For this metric, the keepalive version was better than the original for five functions, while the warm-start version was detected to be better than the original one for six of them. For the remaining two, no difference was found between any of the versions.

The analysis of the task assignment times presented numerous similarities with the analysis of total task times, which could possibly explain why the warm-start and keepalive versions showed no statistical difference between them in the analysis of total task times.

6.4. Total Experiment Time

To complete the analysis, then, it is essential to look at the total execution time of the experiment, to verify the impact of the modifications on the time required for full execution of both benchmarks. Since the experiment simulates a real workflow, the total execution time provides an example of the time it would take for a user to have their flow finished. The analysis was made in a similar manner, extracting the times for each execution of the experiment and then applying the Mann-Whitney test between all versions.

The test detected that the experiment was completed in less time in the warm-start version, followed by the keepalive version and then the original version, with the same significance level of 5%. The average execution times of the experiment for each version were 3236 seconds (original), 2254 seconds (keepalive) and 1254 seconds (warm-start), demonstrating an improvement of 62% of the warm-start version over the original one. A box plot of the results is shown in figure 1.

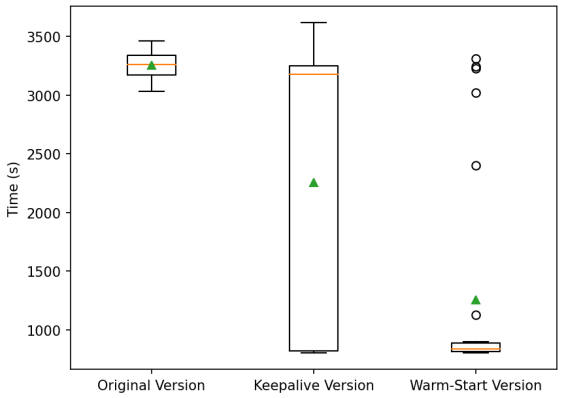

Figure 1. Box plot of the total execution times for the experiments.

The total execution time of the experiments revealed that the warm-start version was capable of outperforming both the original version and the keepalive version, suggesting that the extra work of creating a new task dispatch algorithm was indeed valuable.

It is also possible to observe that the keepalive version had better performance than the original, but a great variance is verified in figure 1, presenting values that range from 1000 to more than 3000 seconds. Meanwhile, both the original and warm-start versions exhibited low-variance results, centered around 3000 and 1000 seconds, respectively.

This reduction in the execution time of the experiment, combined with the results from the time interval measurements, highlights a dependence among the time metrics analyzed, which warrants further discussion. Time improvements in any stage of the flow of a task might lead it to being completed faster, freeing up resources for new tasks to be processed. As a result, the execution time of the workflow is reduced, as supported by the analysis of the experiment.

Therefore, the results presented in this analysis demonstrate that increasing keepalive time might actually improve performance, as proposed by [Bauer et al. 2024b], but also that allowing functions to run on resources where they were already executed is even more valuable. By running on workers which already have their libraries loaded, functions experience shorter execution times, which ultimately leads to reduced execution times of entire workflows.

Further experimentation should be done with more complex workflows, higher function volumes, additional benchmarks, variable execution times, and different machines, to confirm whether the alterations could effectively improve Globus Compute.

7. Conclusion

Globus Compute is a Function-as-a-Service tool of great value to the scientific community. Allowing any machine to become an endpoint provides great flexibility to researchers in the creation and execution of their workflows. Hence, studying ways to improve it is a task that not only benefits the service itself, but also all users who utilize it for their workflows. Thus, the reduction of cold-start delay presents itself as an option of improvement, shortening function initialization times and, consequently, decreasing workflows execution times.

To achieve this objective, the first development was the increase on the keepalive time of managers and workers, improving the likelihood that they are active when a function is invoked. The experiment demonstrated that this change reduced workflow execution times, with several keepalive runs completing in approximately 1000 seconds, while the original version always took over 3000 seconds.

Additionally, another technique was implemented to reduce cold-start, which consisted in adding system functionalities that attempted to assign functions to managers and workers which previously executed them, so that their required libraries were already loaded. The final result of this work is the new version of the Globus Compute system, which combines increased keepalive times with the new task assignment algorithm, the warm-start version.

The experiment demonstrated that this new version greatly improved performance, reducing function and workflow executions times. It follows, then, that the reduction in cold-start delay through the implemented techniques had positive result, contributing to the service and, consequently, to the whole scientific community.

However, further experiments must be devised and conducted, to better evaluate the impact of the alterations proposed in this work. Moreover, new techniques can be studied and implemented, in order to further avoid the cold-start delay and accelerate workflow execution.

Acknowledgments

This research is funded by FAPESP grant #19/26702-8. The opinions, hypotheses, and conclusions or recommendations expressed in this material are the sole responsibility of the authors and do not necessarily reflect the views of FAPESP.

The authors thank Kyle Chard and Daniel S. Katz, as well as all the Globus Compute team, for their advice and technical help.

References

Basu Roy, R., Patel, T., Liew, R., Babuji, Y. N., Chard, R., and Tiwari, D. (2023). ProPack: Executing Concurrent Serverless Functions Faster and Cheaper. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, page 211–224, New York, NY, USA. Association for Computing Machinery.

Bauer, A., Gonthier, M., Pan, H., Chard, R., Grzenda, D., Straesser, M., Pauloski, J. G., Kamatar, A., Baughman, M., Hudson, N., Foster, I., and Chard, K. (2024a). An Empirical Investigation of Container Building Strategies and Warm Times to Reduce Cold Starts in Scientific Computing Serverless Functions. In *2024 IEEE 20th International Conference on e-Science (e-Science)*, pages 1–10.

Bauer, A., Pan, H., Chard, R., Babuji, Y., Bryan, J., Tiwari, D., Foster, I., and Chard, K. (2024b). The globus compute dataset: An open function-as-a-service dataset from the edge to the cloud. *Future Generation Computer Systems*, 153:558–574.

Bermbach, D., Karakaya, A.-S., and Buchholz, S. (2020). Using application knowledge to reduce cold starts in FaaS services. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 134–143, New York, NY, USA. Association for Computing Machinery.

Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. (2019). The rise of serverless computing. *Commun. ACM*, 62(12):44–54.

Chard, R., Babuji, Y., Li, Z., Skluzacek, T., Woodard, A., Blaiszik, B., Foster, I., and Chard, K. (2020). FuncX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 65–76. Association for Computing Machinery.

Conover, W. J. (1999). *Practical Nonparametric Statistics*. Wiley.

Ebrahimi, A., Ghobaei-Arani, M., and Saboohi, H. (2024). Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions. *Journal of Systems Architecture*, 151:103115.

Fireman, D., Silva, P., Pereira, T. E., Mafra, L., and Valadares, D. (2024). Prebaking runtime environments to improve the FaaS cold start latency. *Future Generation Computer Systems*, 155:287–299.

Liu, X., Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., Wang, H., and Jin, X. (2023). FaaSLight: General Application-level Cold-start Latency Optimization for Function-as-a-Service in Serverless Computing. *ACM Trans. Softw. Eng. Methodol.*, 32(5).