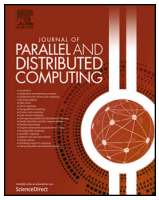
Journal of Parallel and Distributed Computing 190 (2024) 104891

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



Exploiting inherent elasticity of serverless in algorithms with unbalanced and irregular workloads ☆

Gerard Finol *, Gerard París, Pedro García-López, Marc Sánchez-Artigas

Universitat Rovira i Virgili, Tarragona, Spain

ARTICLE INFO

MSC:
68M14

Keywords:
Cloud computing
Serverless computing
FaaS
Elasticity
Unbalanced

ABSTRACT

Function-as-a-Service execution model in serverless computing has been successful in running large-scale computations like MapReduce, linear algebra, and machine learning. However, little attention has been given to executing highly-dynamic parallel applications with *unbalanced* and *irregular* workloads. These algorithms are difficult to execute with good parallel efficiency due to the challenge of provisioning the required computing resources in time, leading to resource over- and under-provisioning in clusters of static size. We propose that the elasticity and fine-grained "pay-as-you-go model" of the FaaS model can be a key enabler for effectively running these algorithms in the cloud. We use a simple serverless executor pool abstraction, and evaluate it using three algorithms with *unbalanced* and *irregular* workloads. Results show that their serverless implementation can outperform a static Spark cluster of large virtual machines by up to 55% with the same cost, and can even outperform a single large virtual machine running locally.

1. Introduction

Serverless computing has been considered the next natural step in the evolution of cloud computing. Features like elasticity, no need to provision servers, and pay-as-you-go billing have attracted a lot of interest both in academy and industry. Since mid-2010s, all public cloud providers have presented serverless computing offerings under the Function-as-a-Service (FaaS) execution model. Services such as AWS Lambda, Google Cloud Functions, IBM Cloud Functions or Azure Functions, allow developers to write modular pieces of code that can be executed in response to certain events. The FaaS model was initially conceived to execute short and event-driven stateless computations, so it imposes a set of architectural constraints and operational limits on memory and CPU resources, network connectivity and function concurrency.

Despite these constraints, researchers have stepped beyond the limits of serverless functions to build more complex applications [24]. Simplicity and fine-grained accounting have attracted researchers to this new compute abstraction that brings new opportunities to better support changing workloads. So, serverless functions have been used to run data-parallel algorithms [23,25,36], video encoding [18], large-scale linear algebra operations [40], stateful computations [8,10,11,41] or real-time data processing [42,47], among other applications.

Although problems that exhibit high levels of parallelism seem a perfect fit for serverless computing, the high elasticity of functions may also be useful to address problems exhibiting irregular, and unbalanced workloads. With a fixed set of compute resources, running these algorithms with high parallel efficiency remains challenging due to the number of decisions to make ahead of time, ranging from provisioning and cluster management to deep understanding of system-level parameters. For instance, an efficient execution of the Unbalanced Tree Search (UTS) benchmark [33] requires fine-tuning task load-balancing [39], especially at large scales [43], compelling users to deal with complex programming models and optimizations. Indeed, the goal of most of these optimizations is to mask the overheads of dynamic thread creation in static multithreaded systems. In any case, it is clear that a cluster of static size will always lead to a certain degree of either: 1. Increased job runtime due to lack of compute resources; or 2. A lower resource utilization if the cluster is provisioned to absorb the peak demand. With a serverless solution, however, the number of compute resources

☆ This article is an extension of a work published in IEEE CLOUD'20 (París et al., 2020) [32]. The major new contributions of this article are: 1. Performance study of the Betweenness Centrality algorithm; 2. Algorithms characterization; and 3. Cost analysis of UTS optimizations.

* Corresponding author.
E-mail addresses: gerard.finol@urv.cat (G. Finol), gerard.paris@alumni.urv.cat (G. París), pedro.garcia@urv.cat (P. García-López), marc.sanchez@urv.cat (M. Sánchez-Artigas).

https://doi.org/10.1016/j.jpdc.2024.104891
Received 28 February 2023; Received in revised form 29 February 2024; Accepted 4 April 2024
Available online 10 April 2024

can be tightly adjusted to the workload, dynamically adding the right number of vCPUs at each stage of the workload without user intervention [23,24,36,40]. This clearly opens the door to run highly-dynamic and unbalanced workloads with an improved parallel efficiency.

As usual, however, there is no free lunch. And a key question is to determine whether using a serverless solution for irregular workloads does also pay off in terms of cost. Prior literature has proven that the FaaS execution model can deliver high performance in many tasks [3,7,18,23,36,40]. Nevertheless, superior performance of serverless computing has been generally achieved at the expense of greater monetary costs, sometimes leading to a poor cost-performance trade-off. To wit, [24] reports serverless solutions with up to 7x increase in cost compared with a traditional solution with VMs. Other papers such, as MLLess [21,38], have proven the other way around. Hence, it is interesting to start drawing out the cost-efficiency of serverless solutions for this kind of problems.

1.1. Contributions

Our goal in this article is to evaluate if the high elasticity of serverless is ideally suited to run algorithms with unbalanced and irregular workloads. Concretely, our focus is to ascertain whether a better (or similar) cost-performance trade-off than cluster-based solutions for these workloads can be delivered without complex mechanisms for dynamic load balancing. The rationale behind a serverless approach is that when the task generation rate is high, spinning up a new function to process a dynamic task can be competitive to waiting for an existing thread to be free and steal the task. Cluster-based solutions lack of such elasticity, and require of complex dynamic load balancing mechanisms to properly distribute load among the compute resources.

This article shows that serverless computing is a compelling approach to efficiently run unbalanced algorithms at large scale. Actually, we demonstrate in the article that a basic serverless executor abstraction [8] mimicking the Java concurrency library, but leveraging functions as Java threads, is by far enough to deliver a better cost-performance ratio than cluster-based solutions. This is good news, because a serverless executor construct based on the thread pool pattern [45] allows developers to preserve the simplicity of the original algorithms as well, with no need to deal with non-obvious system-level components such as task schedulers to achieve good performance. Traditionally, algorithms such as UTS require significant coding effort to scale out [43]. Thus, providing a serverless solution that can rival with HPC-like solutions with close to zero configuration complexity, yet keeping the original simplicity of the algorithms is highly desirable. We believe that this work is a good starting point for more sophisticated implementations yet to come. Also, we present a serverless hybrid executor middleware, implementing also the Java concurrency interface, which combines local CPUs and serverless functions to create a hybrid thread pool. As illustrated in Section 6, our serverless hybrid executor can improve further the cost-performance ratio of the serverless executor.

To evaluate our proposal, we use three challenging algorithms that exhibit unbalanced workloads and irregular task-parallelism: the Unbalanced Tree Search (UTS) [33], the Mariani-Silver algorithm [29], and the Betweenness Centrality (BC) benchmark [6]. Particularly, we compare our serverless implementation of UTS with an existing Spark implementation [22]. Also, we compare the cost-performance ratio of the serverless implementations of the algorithms against large VM instances. The comparison between cloud functions and VM instances provides novel insights into the parallel efficiency of services like AWS Lambda and large EC2 VMs, including low-level issues such as the effect of Intel's Hyper-Threading on the efficiency of compute-intensive, unbalanced tasks.

In general, the use of simpler abstractions favors the more transparent and elastic execution of algorithms at large scale. Coulouris's textbook [15] defines transparency as "*the concealment from the user and the application programmer of the separation of components in a distributed system*", and also defines a specific form of transparency, known as scaling transparency, as the property that "*allows the system and applications to expand in scale without change to the system structure or the application algorithms*". In this sense, this work is an example that an elastic programming model that maintains the simplicity of development of single-machine applications when scaling up to thousands of cloud cores [20] is possible.

In summary, our contributions are:

1. We present a serverless hybrid executor middleware, based on the Java concurrency library, that can execute Java threads over serverless functions and local CPUs. Our elastic programming model provides scaling transparency by efficiently combining local and remote computing entities.
2. We have implemented[1] three unbalanced and irregular task-parallel applications using the serverless executor: UTS, Mariani-Silver and Betweenness Centrality. We provide the first evidence that the FaaS execution model can be leveraged to efficiently run this kind of algorithms.
3. We analyze the cost of running such algorithms in serverless environments and identify those situations where a serverless model can be cost-efficient. As shown in our evaluation, a performance benefit between 20% to 55% can be attained without increasing monetary costs compared to a Spark cluster.
4. We present a characterization of these algorithms using the Coefficient of Variation, the task generation rate and the CDF of the execution time. We use this characterization to optimize the configuration and deployment of these algorithms to the serverless environment.

This article has the following structure. Section 2 contains related work and background for this article. In Section 3 we describe the serverless executor used as well as our novel hybrid executor. Methodology of the validation is presented in the Section 4, in particular, the metrics used, and the algorithms studied are explained along with their characterization. Section 5 contains a performance analysis where we evaluate, inter alia, the effects of Hyper-Threading and shared resources, the UTS optimizations and the performances of the hybrid executor. Section 6 contains a cost-performance study where we compare the serverless and hybrid executors with Spark clusters and large VMs. Finally, conclusions and future work can be found in Section 7.

2. Background / related work

Traditionally, VM clusters and HPC systems have been used to deal with highly unbalanced algorithms. Having a fixed amount of allocated resources generates the need to maximize their utilization, so users do not end up paying for a large number of idle CPU cycles. This need has typically been met with the implementation of complex dynamic load balancing and job stealing techniques.

One example of an unbalanced algorithm is the Unbalanced Tree Search. The UTS benchmark was initially proposed by Prins et al. [33] in 2003 and has since been used as an unbalanced application for evaluating load-balancing algorithms in several parallel computing architectures and programming models, including MPI [16], Unified Parallel C [30] or X10's APGAS [39]. However, all these programming models have been specifically designed to perform load balancing with a static resource provisioning.

On the other hand, serverless platforms (e.g., AWS Lambda, Google Cloud Functions) were initially designed to run short-running, event-driven and stateless computations. In recent years, several execution frameworks [7,12,13,17–19,23,34,37,40,46,49] have been proposed to

[1] Source code available at https://github.com/gfinol/elastic-exploration.

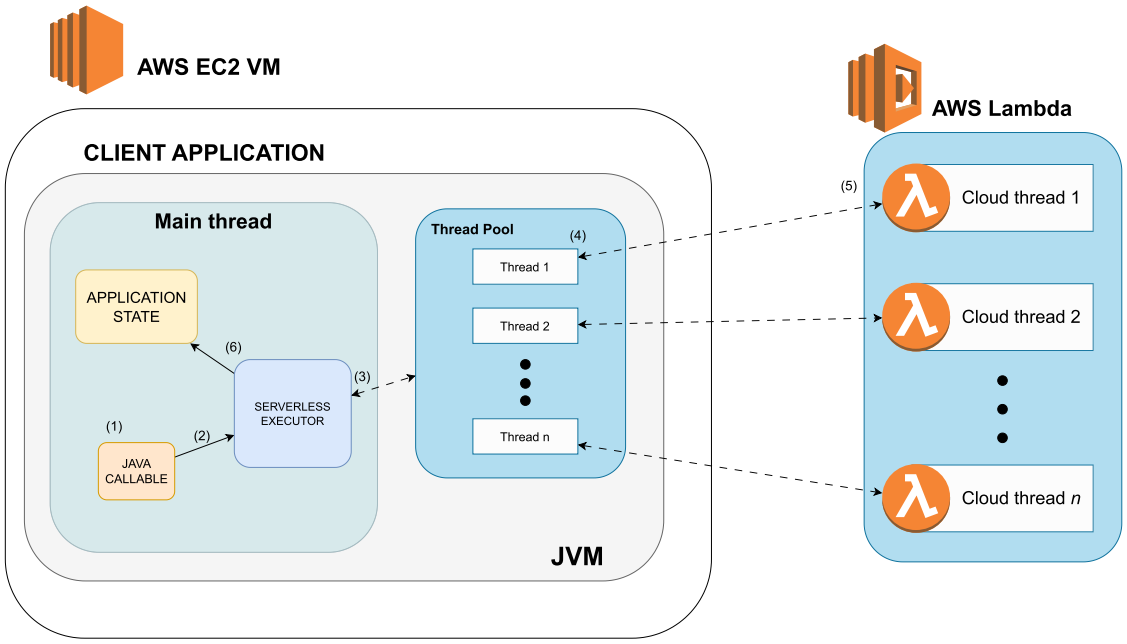


Fig. 1. ServerlessExecutor's architecture.

leverage the scalability of the FaaS model to build more complex parallel applications on top of serverless platforms.

Other frameworks, like Mashup [35] propose a hybrid environment combining VM clusters with serverless computing. Mashup targets HPC and scientific workflows and focuses on the task scheduling. For a given directed acyclic graph (DAG) of tasks to perform, it optimizes whether the task should be run in a VM cluster or in cloud functions. In order to optimize the execution time, it automatically measures time duration of the task for both VM cluster and cloud functions. These experimental measures are ill-suited for dynamic algorithms, where task duration cannot be anticipated at all.

All these available frameworks have mainly been applied to embarrassingly parallel problems. Instead, algorithms with unbalanced and irregular workloads have received little attention, and are only considered in frameworks designed for specific applications, like graph processing (Graphless [46]) or large-scale linear algebra (numpywren [40]). Both of them use a data driven approach. Graphless partitions the set of nodes of a graph and distributes them to cloud functions. This approach can be beneficial for an algorithm like BC, where the computation of the centrality of a node is independent of the others nodes' centrality. Numpywren implements common algebra algorithms (e.g., matrix factorization) as a set of independent DAG tasks, where each task operates over a particular subset of the entire dataset. Unbalanced algorithms like UTS are not suitable to this kind of frameworks because we can not know beforehand the structure of the tree. Therefore, there is no way to find an optimal partition or correctly represent computations into a DAG of tasks.

To the best of our knowledge, there is no previous work studying the feasibility of serverless elasticity for CPU-intensive algorithms with unbalanced and irregular workloads.

3. Serverless executor middleware

For the validation in this work, we have borrowed the `ServerlessExecutor` from Crucial [8]. In addition, from this "pure" serverless executor, we have crafted a hybrid executor, where serverless functions are invoked when the number of local threads in a VM achieves the number of local vCPUs.

To compare performance and cost of serverless functions against a local multithread version, we have implemented parallel multithread versions of the algorithms using the same `Executor` abstraction. In fact, serverless and multithread implementations are very similar. Because we use Java Concurrency API, an existing Java multithread implementation is easily adaptable to serverless.

3.1. Serverless executor

The `ServerlessExecutor` from Crucial is a general-purpose serverless executor based on thread pool pattern from Java Concurrency library [45]. The fragment of code that runs as a serverless function is written as a typical Java `Callable` task, so the developer can use remote cloud functions as if they were local threads.

This serverless implementation follows a master-worker model to support dynamic parallelism of nested algorithms. The client application is responsible to invoke remote tasks as cloud threads and asynchronously retrieve results with a *Future* abstraction. For our work, we did not use the Distributed Shared Object layer that Crucial provides. Therefore, the `Callable`s used are stateless, and cloud functions do not share any data between them. The state of the application is kept in the master.

Fig. 1 shows a description of the architecture of the `ServerlessExecutor` with an example of its usage in this work:

(1) A stateless Java `Callable` is created. It contains the method that will be executed in an AWS Lambda function.
(2) The Java `Callable` is submitted to the `ServerlessExecutor` instance using the `ExecutorService` interface from the Java standard library.
(3) The `ServerlessExecutor` uses an internal thread pool to invoke AWS Lambdas. It assigns one internal thread to one AWS Lambda invocation.
(4) This thread will be responsible to serialize the `Callable`, send it to the cloud function as a parameter in the invocation request and retrieve the eventual result of the invocation.
(5) The `Callable` is executed in an AWS lambda function. The resulting value (or `Exception`) is returned to the corresponding thread.
(6) The `ServerlessExecutor` returns a `Future` object. When the execution of the `Callable` finishes in the cloud function, the produced result will be contained in this returned `Future`. Eventually,

```
1  public abstract class ServerlessHybridExecutorService
       implements ExecutorService {
2    private ExecutorService localExecutorService;
3    private ExecutorService serverlessExecutorService;
4
5    ...
6
7    public <T> Future<T> submit(Callable<T> task) {
8      Future<T> future = null;
9      if (isLocalExecutorIdle()){
10       future = localExecutorService.submit(task);
11     } else {
12       future = serverlessExecutorService.submit(task);
13     }
14     return future;
15   }
16
17   ...
18 }
```

Listing 1: Implementation of the `submit` method for the HybridExecutor.

the main thread will retrieve the resulting value from the `Future` and update the application's state.

FaaS platforms impose some limits on function concurrency that the client application must be aware of. For example, AWS Lambda has a default limit of 1,000 concurrent function executions (a limit that can be increased upon request). Therefore, the `ServerlessExecutor` takes this limitation into account to avoid function throttling exceptions. The invocation frequency is also limited, with great differences in default values between providers: 5,000 per minute in IBM Cloud Functions and 10,000 per second in AWS Lambda. In this case, the invocation frequency limit of AWS Lambda is high enough to run our experiments.

3.2. Hybrid executor

We have implemented a `HybridExecutor` that combines local threads from a VM and remote serverless functions. The idea behind the `HybridExecutor` is to have one VM capable of managing a baseline number of simultaneous tasks, but, at the same time, be able to scale it vertically with cloud functions automatically and without the need for prior provisioning. This allows us to have, at the same time, the benefits of a monolithic system together with the best of the serverless paradigm: the cost of the VM is constant and low, but we have the elasticity and immediate scalability of serverless.

This hybrid version is also transparent from the application point-of-view: `Callable` tasks are submitted to the `HybridExecutor`, which decides whether to run them as local threads or remote serverless functions. Note that `HybridExecutor` also implements the Java's `ExecutorService` interface. Internally, it manages two executors: one backed by a local thread pool and the `ServerlessExecutor`. Listing 1 shows the implementation of the `submit` method, that receives a `Callable` and executes it in one of the two internal executors. As it can be seen, we implemented a naive scheduling policy for the hybrid executor. It will schedule the task depending on the current local load at the time of the submition of the task. If there are no free threads in the local thread pool (line 9), new tasks are scheduled to be run as serverless functions (line 12). But, in the case that the current local load is capable of executing another task, it will be executed locally (line 10).

3.3. Limitations of our approach

As of today, our serverless executor model presents some limitations. As every serverless approach, it has to deal with various constraints of serverless computing, namely limited duration of functions, limit on concurrency and invocation frequency, cold starts, lack of direct network connectivity between functions, and the stateless nature of functions.

Limitation #1 The evolution of the industry in cloud services allows us to be optimistic when it comes to the duration of the functions, limit of concurrency and frequency of invocation. We can observe that these limits are getting higher and, over the time, will have less and less relevance. In our case, these limits have been taken into account and several adjustments have been made on per-algorithm basis to overcome some of them. For example, we need to control the concurrency of cloud functions to avoid exceeding concurrency limits. We also have to tune some parameters of the implemented algorithms to ensure a proper duration of tasks: too short tasks may not be able to mask cloud functions overheads. Some of these adaptations could be avoided using more sophisticated scheduler policies for both the serverless executor and the hybrid executor, but that is out of the scope of this article and will remain as an open challenge for future work.

Limitation #2 When a cloud function is invoked for the first time or after an extended period of inactivity, the underlying cloud's infrastructure needs to allocate the necessary resources to initialize and execute the function. This initialization causes a delay in the execution of the cloud function known as cold start. Cold start can impact the performance of the system, specially when cloud functions run short-lived tasks and the cold start represents a significant portion of the execution time. The mitigation of the cold start problem is an active research field, that we considered beyond the scope of this work. Recently presented solutions, such as SnapStart [2], indicate that the industry is making big efforts to reduce cold start. We have executed all our validations using warm cloud functions to avoid cold start.

Limitation #3 As of today, our approach uses stateless cloud functions. That is, the tasks launched as cloud functions do not share data with one another. Instead, all data will be received and returned as parameters, and the application state is kept at the master. This introduces a single point of failure, so if anything fails inside the master, the state may be lost. However, as a first step, we decided to start with a simple, purely serverless approach with no stateful share components such as databases. As the results in Section 5 and 6 show, this approach is more than enough for the purpose of this work. We leave for future work the adoption of a stateful serverless architecture that will be needed to support other unbalanced workloads with state requirements. Note that Crucial [7] already offers the possibility to have a shared state using a distributed shared object (DSO) layer. So transition to other kind of algorithms should be relatively simple. Note that Crucial [7] already offers the possibility of maintaining a shared state using a distributed shared object (DSO) layer. Therefore, transitioning to other types of algorithms should be relatively straightforward. In addition, the DSO layer could be combined with the orchestration technique presented in Unum [26], which would eliminate the need for a master to orchestrate the functions.

4. Methodology

4.1. Algorithms

In this work we target three of these algorithms with unbalanced workloads and irregular concurrency: UTS, Mariani-Silver and Betweenness Centrality (BC). The first two present nested parallelism (i.e., parallel tasks that generate new parallel tasks) while the third can be statically partitioned. UTS is a well-known benchmark that has been widely used to evaluate task parallelism and load balancing techniques in several parallel computing architectures and different programming models [22,31,43]. The Mariani-Silver algorithm is a recursive optimization technique to compute the Mandelbrot set. This algorithm has been used as an interesting case study of dynamic parallelism in

Table 1
UTS tree size for seed = 19 and $b_0 = 4$.

Depth d	Tree size (# of nodes)
14	1,057,675,516
15	4,230,646,601
16	16,922,208,327
17	67,688,164,184
18	270,751,679,750

CUDA [1]. And Betweenness Centrality is a graph algorithm that computes a centrality metric for each vertex of a graph. The implementation used in this paper is taken from the fourth kernel in the SSCA2 (Scalable Synthetic Compact Application 2) v2.2 benchmark [6].

4.1.1. UTS

The Unbalanced Tree Search (UTS) benchmark [31] was initially proposed by Prins et al. [33] in 2003 and since then, it has been used as an unbalanced application for evaluating load-balancing algorithms in several parallel computing architectures and different programming models like MPI [16], Unified Parallel C [30] or X10's APGAS [39]. However, the underlying executor model of the above parallel computing architectures is not elastic. The implementation of UTS presented in this work is the first that tackles an elastic resource provisioning.

UTS counts the number of nodes in a highly unbalanced task tree that is dynamically generated using SHA-1 cryptographic hashes. The number of children of a node is a random variable with a given distribution. In this paper, we have used a geometric distribution with an expected branching factor $b_0 = 4$ and a depth cut-off d between 14 and 18. Table 1 shows how the number of nodes increases exponentially as we increase the depth cut-off.

The algorithm is constantly generating parallel tasks that explore the unbalanced tree. Each parallel task iterates through a maximum of n nodes of the assigned subtree. Due to the imbalance of UTS, some tasks will traverse n nodes, while many others will traverse significantly less. This algorithm also allows configuring a parameter named split factor: the maximum amount of partitions a subtree is divided into before launching new parallel tasks. This algorithm presents a complexity of $\mathcal{O}(n)$ for a given tree with n nodes, and it has a scalability that varies during its execution. It depends on the number of known nodes that have not been explored yet. Therefore, the scalability at any point of the execution can not be predicted, but it will be always lower than the split factor.

In Listing 2 we present the serverless implementation of the UTS. In this implementation, each serverless function receives a `bag` parameter that encapsulates the subtree assigned to the task (line 25). Once the `bag` is processed, i.e. up to n nodes of the subtree have been explored, it is returned (line 30) to the master program, where it is added to a queue (line 21). Inside the client application, another thread is responsible to continually retrieve bags from the queue (line 6), resizing these `bags` according to the split factor (line 9), and subsequently launch new parallel tasks (line 10).

4.1.2. Mariani-Silver

The Mandelbrot set [27] is a subset of points $c \in \mathbb{C}$ such

$$\lim_{n\to\infty} \|z_{n+1} = z_n^2 + c\| < \infty,$$

where $z_0 = 0$. The Mandelbrot set problem used here consists of drawing the Mandelbrot set, coloring each point in function of how quick z_n converges to ∞. It is known that if $\|z_n\| > 2$ then, z_n converges to ∞. Therefore, we can use the smallest n such that $\|z_n\| > 2$ as a measure of how quick z_n converges to ∞, and we will refer to this value as dwell value. The simplest algorithm to render the Mandelbrot set is the naive Escape Time algorithm, where the z_n iterative calculation is performed for every point in the plot area. However, there are several optimization

```
1  private void uts(List<Bag> bags) {
2    parallelize(bags, iters);
3
4    while(still active threads || queue not empty) {
5      // Get a bag from the queue.
6      Bag bag = queue.poll();
7      if (bag != null) {
8        // Split the current bag into a list of bags
           with #splitFactor elements.
9        bags = resizeBag(bag, splitFactor);
10       parallelize(bags, iters);
11     }
12   }
13 }
14
15 private void parallelize(List<Bag> bags, int iters) {
16   // For each bag of the list, invoke a serverless
       function
17   for (Bag bag : bags) {
18     Future<Result> future = serverlessExecutor.submit(
           new RemoteUTSCallable(bag, iters));
19     // Get the result from the Java Callable and added
       to the queue.
20     Result result = future.get();
21     queue.offer(result.getBag());
22   }
23 }
24
25 class RemoteUTSCallable implements Callable<Result>,
       Serializable {
26   public Result call() throws Exception {
27     // Explore up to n nodes of the subtree.
28     for (int n = numberOfIterations; n > 0 && bag.size
           > 0; --n)
29       exploreNextNode();
30     return new Result(bag);
31   }
32 }
```

Listing 2: Simplified version of UTS serverless implementation.

techniques that can be applied to speed up the rendering without having to compute all the points. One of these techniques is the Mariani-Silver algorithm, an adjacency optimization technique. This algorithm relies on the fact that the Mandelbrot set is connected: there is a path between any two points belonging to the set. Therefore, if the border of a rectangle has the same dwell value, all the inner points will either converge or not. Listing 3 shows the Mariani-Silver `Callable` code, i.e. the code that is executed in the AWS Lambda function. In this algorithm, a rectangular grid is recursively subdivided into subrectangles. The pixels on the boundary of each subrectangle are evaluated, and if they all evaluate to the same result the subrectangle is filled with the same solid color (line 8); otherwise the subrectangle is divided again into two or more smaller rectangles (line 20). The algorithm is recursively applied to each piece until the maximum nesting depth is reached. In this case, all pixels of the rectangle are evaluated (line 14).

When the rectangle has to be split (line 20), new recursive tasks are generated. This kind of nested loops is commonly found in algorithms using hierarchical data structures, such as adaptive meshes, graphs and trees, and also in algorithms like this that uses recursion, and that parallelism can be exploited at each level of recursion. It is difficult to avoid the nested loops, and therefore these algorithms exhibit irregular workloads.

Each parallel task evaluates a single subrectangle and returns the action to take after the evaluation. If the subrectangle has to be subsequently split, the master will be responsible for creating and invoking the new tasks. Otherwise, the evaluation returns the dwell values to assign to the subrectangle (either the boundary value or all individual values of the subrectangle). The maximum dwell, the number of initial

```
1  class MSCallable implements Callable<Result>,
       Serializable {
2    public Result call() {
3      Result result = new Result(rectangle);
4
5      // If the border of the rectangle has the same
       Dwell
6      if (borderHasCommonDwell(rectangle)) {
7          // We perform the Mariani-Silver optimization
       and fill the whole rectangle with the common
       Dwell value.
8          result.setNextAction(Result.Action.FILL);
9
       result.setDwellToFill(rectangle.getBorderDwell());
10
11     // In case we have reached the max depth
       parameter. We can not divide the rectangle
12     } else if (rectangle.getDepth() >= MAX_DEPTH) {
13         // We evaluate each pixel of the rectangle and
       return its value in a matrix.
14         int[][] dwellArray = evaluate(rectangle);
15
       result.setNextAction(Result.Action.SET_DWELL_ARRAY);
16         result.setDwellArray(dwellArray);
17
18     } else {
19         // Split \xch{current}{currect} rectangle into
       smaller subrectangles.
20         result.setNextAction(Result.Action.SPLIT);
21     }
22     return result;
23   }
24 }
```

Listing 3: Mariani-Silver Callable code executed in the AWS Lambda functions.

subdivisions, the split factor, and the maximum depth are configurable. This algorithm presents a complexity of $\mathcal{O}(n \cdot m \cdot d)$ to generate an image of $n \times m$ pixels using d as the maximum dwell value, and it scales with the number of subdivisions of each rectangle.

4.1.3. Betweenness centrality

Betweenness Centrality (BC) is a graph algorithm that computes a centrality metric for each vertex of a graph. In particular, the version used in this work is taken from SSCA2 (Scalable Synthetic Compact Application 2) v2.2 benchmark [6]. Our Java implementation is based on the X10 code used to validate the GLB programming model for large-scale distributed systems [48].

Let σ_{st} denote the number of shortest paths between vertices s and t, and $\sigma_{st}(v)$ the number of those paths passing through v. Betweenness Centrality of a vertex v is defined as

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}. \tag{1}$$

The output of the algorithm is a betweenness centrality score for each vertex in the graph.

The implementation follows the Brandes' algorithm described in the benchmark, augmenting Dijkstra's single-source shortest paths (SSSP) algorithm for unweighted graphs. Listing 4 shows our serverless implementation, which assumes that the whole graph fits in the memory of each function, allowing the same graph to be generated across all functions (line 34). The set of N vertices is statically partitioned among T tasks (line 18), so each function is responsible for performing the computation for the source vertices assigned to one of the tasks (for all N target vertices) and computes its task-local `betweennessMap` (line 37). Then, all local `betweennessMaps` are collected as results of the serverless functions and are finally aggregated in a `globalBetweennessMap` (line 27). Note that the `betweennessMap` and `globalBetweennessMap` are just a mapping between the index of the vertex and its Betweenness Centrality.

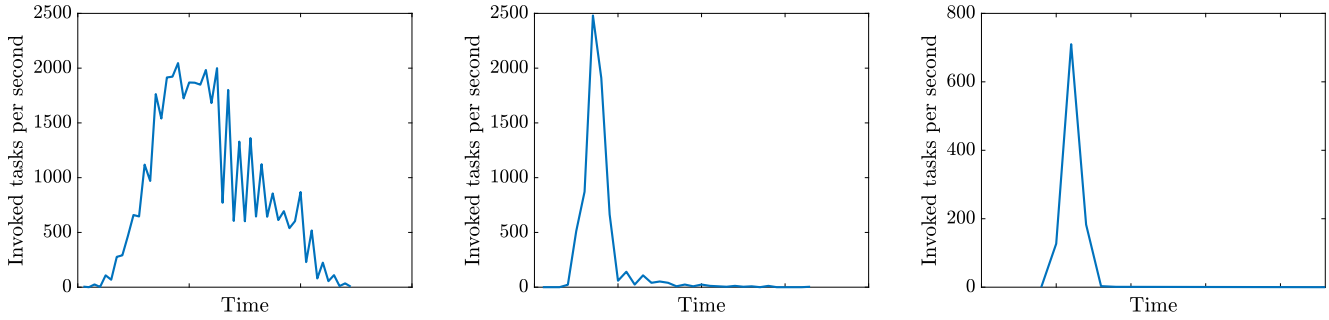
```
1  private void MainBetweennessCentrality(){
2   // Generate the R-MAT that represents the graph
3   Rmat rmat = new RMat(...);
4   setup(rmat);
5   runServerless();
6  }
7
8  // Generate the graph from the R-MAT and permute its
       vertices.
9  private void setup(RMat rmat) {
10  graph = rmat.generate();
11  permuteVertices();
12 }
13
14 private void runServerless() {
15  List<Future<Object>> futures = new ArrayList<>();
16  // Invoke the FaaS targeting the ServerlessCallable
       and add the futures to an array.
17  for (int startIndex = 0; i < N; startIndex +=
       taskSize) {
18   Future<Result> f = serverlessExecutor.submit(new
       ServerlessCallable(startIndex, startIndex +
       taskSize - 1, rmat));
19   futures.add(f);
20  }
21
22  // Retrieve the result from the futures and merge
       the results.
23  for(Future f: futures){
24   Result result = f.get();
25   double[] bMapLocal = result.getBetweennessMap();
26   for(int i = 0; i < N; i++)
27    if (bMapLocal[i] != 0) betweennessMap[i] =
       bMapLocal[i];
28  }
29 }
30
31 public class ServerlessCallable implements
       Callable<Result>, Serializable {
32  public Result call() throws Exception {
33    // Each AWS Lambda function generates its own copy
       of the graph.
34    generateGraph();
35    setArrays();
36    // Compute the \xch{Betweenness}{Betweennes}
       Centrality for the nodes in the partition.
37    for(int i = startIndex; i <= endIndex; i++)
       computeBC(i);
38    // Return the \xch{Betweenness}{Betweennes}
       Centrality
39    Result result = new Result(betweennessMapLocal);
40    return result;
41  }
42 }
```

Listing 4: Betweenness Centrality serverless implementation.

As the graph is generated (line 10) following a "recursive matrix" (R-MAT) model [14], the amount of work to compute each source vertex is different. The vertices are permuted (line 11) before partitioning the graph in order to create more homogeneous tasks, but the resulting tasks are still not perfectly balanced. This algorithm presents a complexity of $\mathcal{O}(v \cdot e)$ for a graph with v vertices and e edges [9]. Note that BC can scale with the number of vertices of the graph, because the centrality of one vertex can be computed independently of the other vertices.

Table 2
Characterization of the tested algorithms.

Algorithm	Task dependencies	C_L	Parameters	Input	Output
UTS	Yes	1.20	Seed = 19 $b_0 = 4$ $d = 18$	Tree parameters	Node counter
Mariani-Silver	Yes	4.06	Width = 4096 Height = 4096	Rectangle parameters	Color matrix
Betweenness Centrality	No	0.23	Seed = 2 $T = 128$ R-MAT = $\begin{pmatrix} 0.55 & 0.1 \\ 0.1 & 0.25 \end{pmatrix}$	Unweighted Graph	Centrality array



(a) UTS (b) Mariani-Silver (c) Betweenness Centrality

Fig. 2. Task generation rate.

4.2. Algorithms characterization

All the studied algorithms have unbalanced and irregular workloads, but each one has specific characteristics. Table 2 summarizes some of these differences. For example, UTS and Mariani-Silver are recursive algorithms that generate data dependencies between tasks, whereas BC workload can be partitioned in independent tasks beforehand.

A metric for imbalance is crucial for characterizing these types of algorithms. A common metric is the coefficient of variation. If we consider L as the set of execution times for all subtasks of an algorithm, we can define an imbalance metric in terms of the coefficient of variation of L:

$$C_L = \frac{\sigma_L}{\mu_L} \tag{2}$$

As shown, C_L is defined as the ratio of the standard deviation σ_L over the mean μ_L. C_L expresses the extent of variability in relation to the mean of the measured values without depending on the measurement unit. This metric has been used to quantify load imbalance on virtualized enterprise servers [4] as well as in many other areas of computer science. The higher the C_L, the greater the variation, and thus the greater the skewness between task durations. Table 2 also contains the coefficient of variation C_L for each of the studied algorithms. Mariani-Silver has the highest C_L coefficient, which implies that it has the highest task duration variability, and the BC algorithm has the lowest C_L.

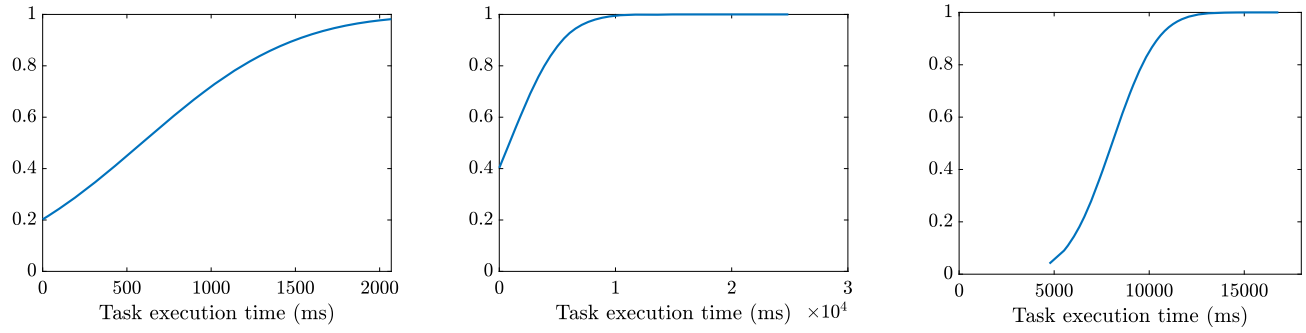
The management of resources is, in part, influenced by the number of task generated during the execution of the algorithms. Therefore, another important factor to take into account is the task generation rate. Fig. 2 shows how many tasks per second are generated by each algorithm. We can see how UTS has a very irregular high task generation rate during practically the entire execution. This is the opposite of the case of the BC, where all the tasks are generated at once at the beginning of the execution. Mariani-Silver is an intermediate term, it generates a large part of the tasks at the beginning, but during the rest of the execution it maintains an irregular generation rate on a much smaller scale.

In order to complement the characterization provided by the C_L coefficient, the Cumulative Distributive Function (CDF) of the execution time of the tasks of each algorithm has been studied. Fig. 3 contains the graphical representation of the CDF for each algorithm. We can see that, in UTS, 20% of the tasks have a duration of less than 1 ms and from the duration of the tasks it is uniformly distributed until reaching 2000 ms. In the case of Mariani-Silver we can observe a very wide task duration interval, where 40% of the tasks are under 1 s and 99% are under 10 s. The large percentage of tasks with a very short duration is due to the Mariani-Silver adjacency optimization technique. This technique allows to safely fill in the entire region without computing the interior when all the boundary pixels of the region are evaluated to the same result. In the Mariani-Silver's CDF, the long tail of the graph on the right side is particularly striking, indicating the existence of some (very few) tasks with a very high execution time (up to 25 s) compared to the rest of the tasks. The BC's CDF shows a more homogeneous task duration than the other two algorithms, matching the C_L coefficient. Even so, execution times are evenly distributed between 4 s and 12 s for most tasks, which is still a wide range despite the random distribution of the vertex assigned to each task.

4.3. Performance and cost metrics

When considering performance of a serverless environment, a basic performance indicator is the total execution time. Also, as the selected algorithms can be quantitatively characterized by the number of traversed nodes (UTS) or the number of computed points (Mandelbrot), we can define a throughput indicator dividing the number of nodes or points by the total execution time. Using this throughput indicator and the total cost of the execution, we can derive a price to performance ratio ($R_{\text{Price-Performance}}$) for each algorithm.

The price to performance ratio has been used to discuss trade-offs between cost and performance, as it shows the throughput that can be achieved per dollar spent. It has been computed using the following formula:



(a) UTS (b) Mariani-Silver (c) Betweenness Centrality

Fig. 3. Cumulative Distributive Function (CDF) of the execution time of each task.

$$R_{\text{Price-Performance}} = \frac{\text{Throughput}}{\text{Cost}}. \tag{3}$$

As mentioned before, we can define a throughput metric for each algorithm. To compute the throughput in the UTS we use the number of nodes, so it is computed as:

$$\text{Throughput}_{\text{UTS}} = \frac{\#\text{ of nodes}}{\text{Execution time}}, \tag{4}$$

whereas, in the Mandelbrot, we use the number of points, so it is computed as:

$$\text{Throughput}_{\text{Mandelbrot}} = \frac{\#\text{ of points}}{\text{Execution time}}. \tag{5}$$

The cost metric depends on the used resources, since the serverless cost ($\text{Cost}_{\text{Serverless}}$) is calculated differently than the VM cost (Cost_{VM}). In both cases, we use the provider listed prices. The cost of running an algorithm in a VM can be computed as the VM price per hour divided by 3600 (seconds per hour)[2] and multiplied by the total execution time (t_{total}) in seconds:

$$\text{Cost}_{\text{VM}} = \frac{\text{VM Price \$}}{3600} \cdot t_{\text{total}}. \tag{6}$$

The total cost of running an algorithm with serverless functions is calculated as: the cost of the invocations, plus the cost of the execution time, plus the cost of the VM where the client application is executed (Eq. (7)):

$$\text{Cost}_{\text{Serverless}} = \text{Cost}_{\text{Invocations}} + \text{Cost}_{\text{Execution}} + \text{Cost}_{\text{Client}}. \tag{7}$$

Considering the current AWS pricing model, the total cost of n invocations is calculated as:

$$\text{Cost}_{\text{Invocations}} = \lambda_i \cdot n, \tag{8}$$

where λ_i is the AWS Lambda invocation cost per serverless function. The cost of the execution time is the gigabyte-second price multiplied by the gigabytes of memory assigned to functions and the billed duration of each execution in seconds (t_i):

$$\text{Cost}_{\text{Execution}} = \lambda_e \cdot \frac{\text{memory in MB}}{1024\text{ MB}} \cdot \sum_{i=0}^{n} t_i, \tag{9}$$

where λ_e is the AWS Lambda execution cost per Gigabyte per second. The cost of the client VM is computed similar to Cost_{VM}:

$$\text{Cost}_{\text{Client}} = \frac{\text{VM Price \$}}{3600} \cdot t_{\text{total}}. \tag{10}$$

Table 3 shows the specific AWS prices from previous formulas at the time of performing the experiments.

[2] AWS does a per-second billing, so we only pay for the number of seconds of VM used.

Table 3
AWS Lambda prices.

Constant	Value
λ_i	\$0.0000002
λ_e	\$0.0000166667
VM Price (m5.xlarge)	\$0.192
VM Price (c5.2xlarge)	\$0.34

Table 4
Average invocation overheads.

	Overhead
Serverless Functions	13 ms
Local threads	18 µs

4.4. Experimental setup

All experiments in this paper are conducted in Amazon Web Services (AWS). We use AWS Lambda as serverless functions, with each function configured with the amount of memory to have the equivalent to one full vCPU of compute time, according to AWS documentation [5]. If not stated otherwise, the client application is executed in a `m5.xlarge` instance in the same region assigned to AWS Lambda. All execution times are the mean of at least 10 runs. In order to make comparisons as fair as possible, all the serverless functions used have been pre-warmed, avoiding cold start. Similarly, the launch and startup time of VMs have not been considered in our evaluations.

5. Performance analysis

In this section, we evaluate the performance that can be obtained using the serveless and hybrid executors in algorithms with unbalanced and irregular workloads. We compare different aspects of performance, such as invocation overheads or execution time, between the serverless, the hybrid and the multithreaded implementations. Finally, we analyze an optimization technique in the UTS, which can be easily implemented with the executor abstraction.

5.1. Overheads analysis

In this section, we analyze different types of overheads that ServerlessExecutor and HybridExecutor may experience. The overheads are measured by running micro-benchmarks, and we compare the results obtained with Java's multithreaded Executor.

5.1.1. Invocation overhead

For task-parallel workloads in FaaS, one of the main factors that affect final performance is the overhead of running a remote task. Table 4 compares the overheads to invoke a dummy task as a serverless

Table 5
Performance and parallel efficiency of UTS.

	Logic CPUs	Depth	Time (s)	Throughput (M nodes/s)	Parallel efficiency
Sequential	1	14	75.86	13.94	
AWS Lambda	96	17	74.55	907.94	67.84%
AWS Lambda	48	17	141.67	477,76	71.40%
`c5.24xlarge`	96	17	113.90	594.22	44.40%
`c5.24xlarge` (HT disabled)	48	17	131.73	513.82	76.78%
`c5.12xlarge`	48	17	214.83	315.06	47.09%

Table 6
`HybridExecutor`'s scheduler overhead comparison.

	Time to submit all tasks (ms)		Total execution time (ms)	
	Average	σ	Average	σ
Multithread	4.47	0.915	5005.26	2.34
Serverless (`c5.12xlarge`)	4.53	0.97	5054.60	22.58
Serverless (`m5.xlarge`)	49.33	12.37	5096.66	13.11
Hybrid (48 local threads)	3.36	0.57	5004.69	3.89
Hybrid (24 local threads & 24 Lambda)	3.15	0.60	5060.93	19.18

function and as a local thread, using the executors presented in this paper. The task simply receives an input parameter and returns an output parameter without performing intensive computations. Both parameters are short strings. To obtain an average overhead, we measure the time to sequentially submit and obtain results of 1k serverless functions and 1M local threads. Both executors are previously warmed up to discard overheads attributed to thread pool initialization, connection set up and cold starts. The experiments are carried out in an EC2 `c5.2xlarge` instance, and we use AWS Lambda to run serverless functions. Even though the overhead of invoking a serverless function ($\sim$ 13 ms) is three orders of magnitude greater than the overhead of creating a local thread, it is still low enough to be masked by the higher concurrency provided by the serverless functions. As can be seen in the results of the performance analysis of the UTS algorithm.

5.2. HybridExecutor overheads

As we explained in Section 3, the `ServerlessExecutor` runs all the Callables in cloud functions, but the `HybridExecutor` must decide where to execute the tasks: either as local threads in the client VM or in cloud functions. Although our scheduling implementation in the `HybridExecutor` is straightforward, it executes tasks in FaaS only when all threads of the local executor are occupied, we are introducing a potential overhead.

To measure this possible overhead, we have performed a micro-benchmark. We run 48 Callables that will sleep for 5 seconds and return. We measured two metrics, the time needed to submit all the 48 Callables to the Executor and the total execution time. Note that, as can be seen in Listing 1, the scheduling of the `HybridExecutor` is done in the submit method. We refer, as total execution time, to the time elapsed from the submission of the first callable until the completion of the last callable, including the 5 seconds sleep.

Setup We compare the `HybridExecutor` with both the multi-threaded Javas' executor and the ServerlessExecutor. For the multi-threaded and hybrid executor we use the `c5.12xlarge` VM from AWS EC2 (48 vCPUs). The `HybridExecutor` has been tested with two different configurations, the first one with a local thread pool of 48 vCPUs (therefore it does not need to invoke cloud functions) and the second one with 24 local threads and 24 cloud functions. Note that, even though no cloud functions are called, the scheduling condition must still be evaluated. For the ServerlessExecutor we also used two configurations, the first one is our default configuration explain in Section 4.4 and the second one uses the same `c5.12xlarge` VM to invoke the AWS Lambdas.

Results Table 6 shows the average result and the standard deviation for both metrics measured. We observe an almost identical time to submit all task in all the configurations, except for the ServerlessExecutor using the `m5.xlarge` VM with only 4 vCPUs, which was an expected result given the big differences in vCPUs. When looking at the total execution time, we observe two interesting facts. First, that the configuration of the hybrid executor using only local threads performs as good as the multithreaded executor. Second, the hybrid executor that uses both local threads and AWS Lambda functions performs as good as the Serverless executor using the same VM and slightly better than the one using the small VM (again, an expected result because of the difference in vCPUs). The results of this benchmark show that the overheads of the `HybridExecutor` are negligible for the type of workloads that we focus on this work. At the same time, results also show a slight disadvantage for the ServerlessExecutor when used with our default configuration.

5.3. Effects of hyper-threading on compute-intensive tasks

Running a parallel algorithm in a distributed environment, either serverless functions or a cluster, obviously adds some overheads to the final execution time, mainly related to serialization and network latency. If permitted by the scale of the problem, a viable alternative is to run a parallel version of the code in one of the largest virtual machine instances offered by cloud providers. In Table 5 we show a comparison of the performance achieved with the serverless and parallel versions of UTS. The parallel version is run in a `c5.24xlarge` VM with 96 vCPUs and in a `c5.12xlarge` VM with 48 vCPUs. For comparison purposes, the serverless version has been limited here to launch a maximum of 96 or 48 concurrent functions. In order to have a reference of the throughput achieved, we also run a single-threaded version in the same VM using a lower depth parameter of 14. The selected depth does not affect the throughput of the sequential code.

Despite the inherent overheads, the serverless version appears to be surprisingly faster than the parallel version: traversing a 17-depth tree takes 74.55s in average with a maximum of 96 concurrent AWS Lambda functions, while it takes 1.5x more time with the `c5.24xlarge` VM. Comparing the throughput of the parallel version with the sequential version, we see that the parallel efficiency is very low (44.4%). This can

be attributed to the effect of Intel's Hyper-threading (HT) to a compute-intensive task like this. Hyper-Threading technology makes a single physical core appeared as two logical cores. The physical core resources are shared, and the architectural state is duplicated for the two logical cores [28]. It is known that depending on the characteristics of the application run, Hyper-Threading may help or hinder performance [44]. In particular, compute-intensive applications have less chance to be improved in performance from Hyper-Threading because the CPU resources could already be highly utilized [44]. When running the parallel version in the VM, each pair of vCPUs or logical CPUs share the same physical core. In effect, we verified that disabling Hyper-Threading, and thus reducing in half the number of logical CPUs, the throughput of the parallel version only decreased slightly, and the parallel efficiency reached a more acceptable rate of 76.78%.

To finish the study of the parallel efficiency, we make the comparison in the case of a parallelism of 48. In this case we have a VM with Hyper-Threading (c5.12xlarge), a VM without Hyper-Threading (c5.24xlarge) and the AWS Lambda limited to the maximum use of 48 concurrent functions. In this scenario, the comparison between the two VMs is not interesting, since c5.24xlarge has twice as many physical cores as c5.12xlarge. However, both have been configured to have the same number of logical cores, just like the serverless based system. This allows us to contextualize the parallel efficiencies of the serverless functions and measure the effects of the existing overheads.

Results show that the `c5.24xlarge` VM with Hyper-Threading disabled gets the best performance, with an execution time of 131.73 s. However, the execution time of the serverless functions is only 7% higher (141.67 s), in contrast with the c5.12xlarge VM execution time, which is 51% higher (214.83 s) than the serverless functions. Therefore, the functions get a parallel efficiency much closer to the parallel efficiency of a VM without Hyper-Threading.

This is an interesting insight that must be considered when comparing performance in virtual machines versus serverless functions. Although, according to AWS documentation, each Lambda function has been configured to an equivalent of one vCPU [5], a direct comparison to a VM with the same theoretical resources may lead to misleading results, especially for compute-intensive tasks. While all physical cores of the VM are at full utilization, the serverless functions are run in AWS Lambda infrastructure, a setting that is out of the control of the user and its core real utilization can vary and is not known by the user either. However, this experiment suggests that, at least at AWS Lambda, compute intensive parallel tasks may exhibit better performance than the same tasks at a VM with the same resources in terms of vCPUs.

5.4. UTS optimizations

Our serverless implementation of UTS can be optimized to further reduce total execution time. If the application can sustainably maintain a high task concurrency, UTS tree traversal finishes earlier. The concurrency greatly depends on the split factor, the number of parts that a task is split into. A high split factor generates more tasks, but an excessive number of tasks is counterproductive because it adds overheads. Another parameter that can be tuned is the number of nodes processed at each task.

The basic serverless implementation maintains these two parameters static during the execution. Here, we apply an optimization to dynamically modify the split factor and the number of nodes processed depending on the current level of concurrency.

In order to achieve a significant improvement, the parameters updates do not need to be done with any complex parameter curve. Listing 5 shows an example for the UTS optimization implementation, where simple parameters updates are performed in four different stages of the UTS execution. We assign a large split factor (line 4) until a high concurrency is achieved, and then slowly start to decrease the split factor (lines 13 and 18) as the measured concurrency also goes down defined levels. Likewise, the number of processed nodes

```
1  private void uts(List<Bag> bags) {
2   parallelize(bags, iters);
3   iters = 50_000;
4   splitFactor = 200;
5
6   while(<still active threads || queue not empty>) {
7    Bag bag = queue.poll();
8    if (bag != null) {
9     activeThreads.addAndGet(-1);
10
11    if (step==0 && active threads > 800) {
12     step++;
13     splitFactor = 50;
14     iters = 2_500_000;
15    }
16    if (step==1 && active threads > 1300) {
17     step++;
18     splitFactor = 5;
19     iters = 5_000_000;
20    }
21    if (step==2 && active threads < 1100) {
22     step++;
23     iters = 2_500_000;
24    }
25    if (step==3 && active threads < 100) {
26     step++;
27     iters = 1_000_000;
28    }
29
30    bags = resizeBag(bag, splitFactor);
31    parallelize(bags, iters);
32   }
33  }
34 }
```

Listing 5: UTS Optimization example.

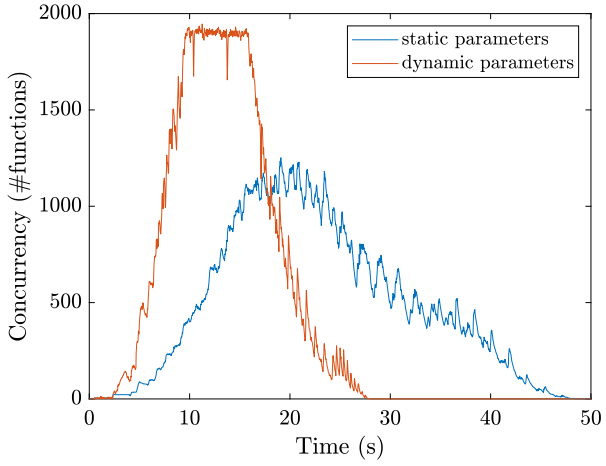
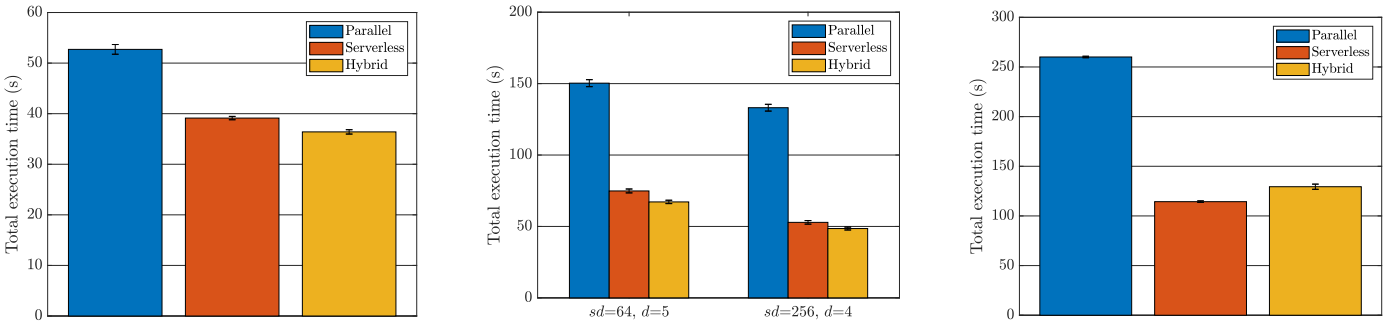


Fig. 4. Concurrency and total execution time for serverless version of UTS ($d = 18$).

is kept lower at the beginning (line 3) and the end of the execution (line 27), and is increased when high concurrency levels are achieved (line 19).

Fig. 4 shows the concurrency and total execution time improvement after this optimization is applied. The maximum concurrency allowed by AWS Lambda in this environment is 2,000 functions, so we configure the serverless thread pool with this maximum. In the dynamic version, we observe that the effective concurrency saturates near this maximum, achieving higher concurrency than the static version. As a result, this version finishes in 27.9 s.



(a) Unbalanced Tree Search ($d = 16$). (b) Mariani-Silver in two configurations. (c) Betweenness Centrality (2^{17} vertices).

Fig. 5. Performance comparison of the Hybrid executor. The error bars represent the standard deviation of all measurements.

5.5. Preliminary hybrid executor evaluation

To assess the performance implications of the hybrid executor, we conduct some preliminary evaluations using the three algorithms presented previously. We used the UTS algorithm with a depth of $d = 16$, the Mariani-Silver algorithm to generate a 4096×4096 pixels image of the Mandelbrot Set, and the BS algorithm with a graph size of 2^{17} vertices. To maximize the performance benefits of locality, the hybrid executor followed a simple policy rule: dispatch tasks to serverless functions only if there are no local threads available in the pool.

Setup We used UTS algorithm with static parameters, in particular we used a split factor of 5 and 5 million iterations per worker. In the Mariani-Silver, the maximum dwell of a point was configured to 5 million iterations and each rectangle was subsequently divided in 4 parts. We run the Mariani-Silver using two different configurations: the first one with an initial subdivision (sd) of 64 and a maximum recursion depth (d) of 5, and a second one with $sd = 256$ and $d = 4$. Note that the second configuration implies an initial steeper demand for concurrency, leading to an earlier use of serverless functions. Finally, the BC algorithm has been configured using a task size of $T = 256$ vertex and the random seed and R-MAT of Table 2. Both parallel and hybrid implementations were executed in a `c5.12xlarge` EC2 instance (48 vCPUs). We run the parallel version with 48 concurrent threads. The hybrid version was restricted to 24 local threads to compel the participation of cloud threads.

Results Fig. 5 shows the total execution time of the algorithms under our three different implementations: multithreaded, serverless and hybrid. As can be seen in the figure, our serverless and hybrid executors obtain better performance than a large VM, achieving a speedup factor > 2 in the Mariani-Silver and the BC algorithms. Non-surprisingly, the performance of the hybrid executor over the pure serverless executor is not so high, as can be seen in Figs. 5a and 5b. The main reason is that these algorithms are highly CPU-bound, with a very small I/O demand. So, the small improvement of the hybrid executor mainly comes from the hiding of the invocation latencies of the remote cloud threads. We expect significant improvements for dynamic parallel algorithms, where data locality is a key factor for their scalability. Note that, contrary to the other two algorithms, the BC results in Fig. 5c show a slightly worse performance of the hybrid implementation with respect to the serverless implementation. These differences are explained in Section 5.6. Either way, the combination of local and remote cloud resources makes a case worth exploring in the future.

5.6. Effects of shared physical resources

The Betweenness Centrality algorithm is a CPU-intensive algorithm that also makes a heavy use of data read from memory. In this section, we study the performance of this algorithm in a serverless and multithreaded environment. As explained in Section 4.1.3, and as can be seen in the line 34 of the Listing 4, the serverless implementation of the BC algorithm computes, in each cloud function, the entire graph. Note that due to the random nature of the graph it is impossible to partition it without computing the shortest paths, so to compute the centrality of a node it is necessary to have the entire graph, which is too large to be passed as a parameter to a cloud function. In the multithreaded implementation, however, the graph can be shared by each task, so there is no need to replicate it. This is the only difference between the serverless and multithreaded versions, the rest of the BC code is exactly the same.

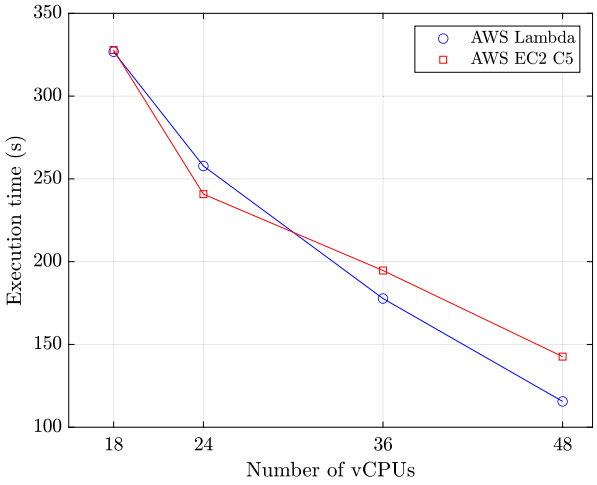


Fig. 6. Performance comparison of Betweenness Centrality ($N = 17$).

To rule out the effects caused by Hyper-Threading explained in Section 5.3, VMs running multithreaded BC code will have Hyper-Threading disabled for this comparison. These VMs are compared to cloud functions by limiting the maximum number of concurrent functions to the number of CPUs in each VM. In particular, the following AWS specialized compute VMs have been used: `c5.9xlarge` (18 CPUs), `c5.12xlarge` (24 CPUs), `c5.18xlarge` (36 CPUs) and `c5.24xlarge` (48 CPUs). Client application for the serverless implementation has been executed in a `c5.2xlarge`.

In Fig. 6 we show a comparison of the performance of the two implementations. Despite the fact that each serverless function must recompute the graph and the overheads intrinsic to distributed systems, the serverless version reduces up to 10% the execution time of the parallel version using the same number of vCPUs. In the plot, we can also see that the performance of the multithreaded version degrades as the size of the VMs increases. However, the serverless version is able to scale while maintaining performance.

In this experiment, it could be assumed that VMs had some advantage. Since they have the same number of CPUs, Hyper-Threading is disabled, and they avoid the overhead of Cloud Function virtualization.

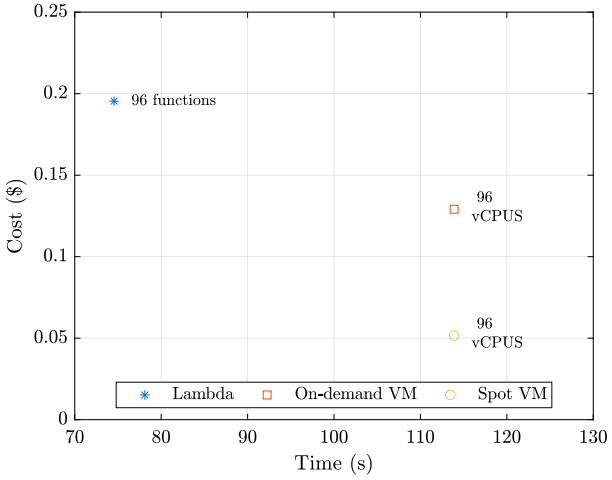


Fig. 7. Cost-performance comparison of the parallel and serverless version of UTS at depth 17 with comparable CPU resources.

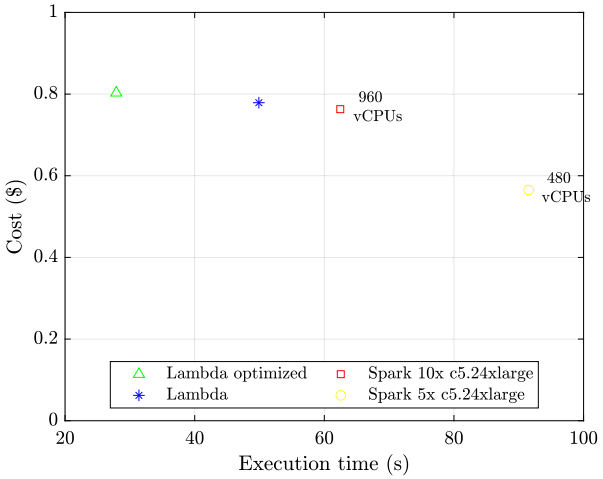


Fig. 8. Cost-performance comparison of the Spark and serverless version of UTS at depth 18.

However, the serverless implementation is able to perform 10% better while doing the same exact computations. This is because the performance of the JVM degrades with a high number of threads. Degradation is mainly caused by two factors, the increased Garbage Collector's response time and the usage of shared resources. For this case, shared resources cause drawbacks like memory access contention and inefficient use of the shared L3 cache, which degrade the total performance of the system.

6. Cost analysis

Alongside performance, cost is the other main characteristic that must be analyzed in every serverless computing application. Although the promise of cost savings (through a true pay-as-you-go model) is one of the features that attract new applications to FaaS, an accurate cost assessment must be carried out when comparing FaaS with other execution environments. In fact, several FaaS applications prototypes show a higher cost than comparable traditional IaaS-based applications [24].

Compared to other cloud computing offerings, FaaS price includes much more than the specific resources: scaling, redundancy for availability, monitoring, and logging. But if we only take into account the computing power, cheaper alternatives exist also in the cloud. Transient servers like Amazon EC2 Spot offer spare compute capacity at steep discounts. After recent changes in Amazon Spot pricing model, there are fewer instance interruptions and the prices are more stable and predictable, making transient servers a good option for running exploratory tasks.

A clear scenario where it is worth paying FaaS cost is when the nature of workload causes an underutilization of virtual machine or cluster computing resources. It is common to overprovisioning a cluster to cope with peak demand of irregular workloads. An alternative, when using cloud computing, is to autoscale resources according to computing demands. But startup latency of VM instances is still higher than lighter serverless containers. Moreover, application programmer has to deal with the complexities of elastic scaling, while serverless offers this out-of-the-box.

Determining the appropriate amount of resources to devote to an application is still a matter of research, specially for irregular workloads. We believe that due to its inherent elasticity, FaaS may be a cost-efficient approach to explore the solution space of a problem, even if cost is a bit higher. Serverless functions could be useful to find the right provisioning for a problem before moving to less expensive, but more complex to manage cloud resources, like transient servers.

Fig. 7 shows the charged costs for the UTS at depth 17 both in AWS Lambda (with the limit set to 96 concurrent functions) and using the parallel version in a `c5.24xlarge` VM. Although the performance of the serverless version is better, the cost is higher, specially if compared to the spot instance pricing model.

Highly parallelizable algorithms can benefit from the high concurrency and elasticity that FaaS platforms can provide. Fig. 8 shows a cost-performance comparison of the serverless version of UTS with a Spark implementation [22] that uses a Map/Reduce strategy, with tree traversal divided into rounds. Both versions are parameterized to achieve the best performance possible. The serverless version is limited to 2,000 concurrent functions, the maximum concurrency supported by the cloud provider in the region we run the experiment. We run the Spark versions with two different settings: an Elastic Map Reduce (EMR) cluster of 10x `c5.24xlarge` workers (totaling 960 vCPUs), and a smaller cluster of 5x `c5.24xlarge` workers (480 vCPUs). The master node is a smaller `m5.2xlarge` instance. Costs are calculated on the basis of the on-demand pricing of EMR. For example, for the 10-node cluster and considering t is the total execution time in seconds:

$$Cost_{EMR} = \frac{t}{3600}(10 \cdot 4.35\$ \text{ per hour} + 0.48\$ \text{ per hour}) \tag{11}$$

We see that the unoptimized serverless version depicted in Fig. 8 is able to outperform Spark by up to 20%, for a similar cost. If we calculate the price to performance ratio, dividing the performance in a million nodes per second by the cost in dollars, we find that the serverless version has a ratio of 6,966 M nodes/s/$, while the Spark version only achieves 5,689 M nodes/s/$. If we apply the dynamic optimizations described in section 5, the serverless version is able to outperform Spark by up to 55%. The dynamic optimizations led to a price to performance ratio of 11,366 M nodes/s/$, achieved through significant time reduction and minimal cost increase.

In fact, the cost of serverless executions hardly depends on the total execution time. In the case of the UTS, the main challenge is to be able to have the maximum number of workers exploring the tree, so that the average amount of work they perform is large enough to compensate the system overhead. As explained in the previous section, the optimization of dynamic parameters can be used during the exploration. At times when you have fewer workers, you can try to divide the tasks more and when you have the pool of workers at full capacity, divide them into fewer tasks. Similarly, when you have few workers it is interesting that workers do not explore too much in order to create new tasks soon, instead when you have many workers you can allow them to explore more in order to try to reduce overheads.

With this in mind, if we look at Eq. (7) that makes explicit the cost of serverless executions, the sums of invocations and the cost of the VM are orders of magnitude below the price of serverless functions'

Table 7
Cost/Performance of UTS ($d = 18$) for serverless and Spark cluster with resource utilization metrics.

	vCPUs	Execution time (s)	Cost ($)	Price-Performance Ratio (M Nodes/s/$)	CPU usage (%)	
					Average	Max
Spark (10x m5.2xlarge)	80	489.65	0.71	789.90	79.3%	87.9%
Serverless (80 AWS Lambda)	80	313.19	0.70	1,234.98	N/A	
Serverless (800 AWS Lambda)	800	54.39	0.70	7,111.00	N/A	

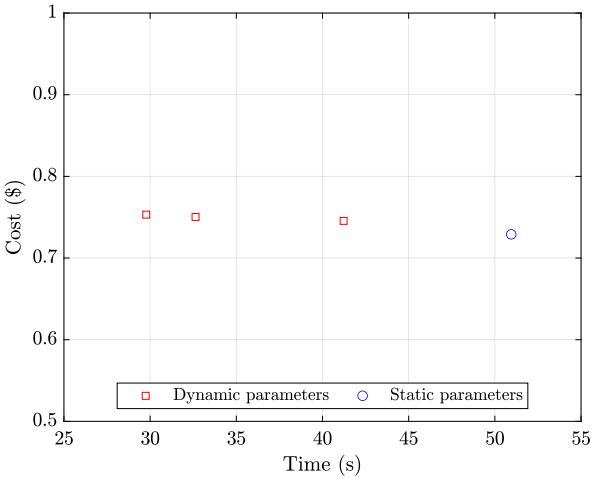


Fig. 9. Cost-performance comparison of UTS at depth 18 using static and dynamic parameters.

execution time. As the computation time used by serverless functions is practically constant and given the pay-as-you-go billing model, the most aggressive optimization strategies have approximately the same cost as the execution with static parameters. This fact, combined with the UTS optimizations using dynamic parameters, allows for an improvement in performance at a very low extra cost.

Fig. 9 shows the cost and wall time of the execution of the UTS using static parameters, together with three different executions that use dynamic parameters. Note that all executions using dynamic parameters implement the techniques described in Section 5.4, but with a different level of fine-tuned parameter curve. It can be seen that the cost difference does not exceed $0.024, which represents 3.31% of the total cost of the static execution. However, an execution time improvement of 41.56% is achieved compared with the static parameters.

In Table 7 we can see a comparison of the UTS between a Spark cluster and the serverless implementation where we measure the resource utilization in Spark during the execution of the UTS, particularly focusing on CPU utilization because UTS is a CPU-intensive algorithm. Note that AWS Lambda does not allow us to obtain CPU utilization data. Given the pay-as-you-go model, one could speculate that the resource utilization of AWS Lambda should be close to 100%. However, the truth is that we cannot precisely determine the portion of the execution time of the cloud function (which is paid for) that corresponds to CPU usage. This comparison was made using an EMR Spark cluster with 10 m5.2xlarge VMs, which in total add up to 80 vCPUs. On the serverless side, we used two different configurations: one limiting the concurrency of AWS Lambda to 80 functions, and another limiting it to 800 concurrent functions.

We can observe in Table 7 that both the cost of the two serverless configurations and the cost of Spark cluster are the same. However, the Price-Performance ratio of the Spark cluster is only the 64% of the Serverless execution with 80 AWS Lambdas, and 11.1% of the configuration with 800 AWS Lambdas. These poor results from the Spark cluster are due to a low average CPU utilization along with the effects of Hyperthreading explained in Section 5.3. In addition, the results in this experiment also show a similar insight to the previously presented in Fig. 9. The cost of the UTS using up to 80 or 800 concurrent cloud functions is the same, but the execution time is much shorter, as a direct consequence this last configuration achieves an almost 7× better Price-Performance ratio.

Table 8
Cost/Performance of Mariani-Silver serverless and parallel implementations.

	Time(s)	Cost ($)	Price-Performance Ratio (MP/s/$)
Parallel (c5.12xlarge)	133.13	0.0852	1.47
Serverless	52.85	0.2360	1.34
Hybrid	48.53	0.2041	1.69

In section 5 we already saw that our serverless and hybrid executors can obtain better performance than a multithreaded executor in a large virtual machine for the Mariani-Silver algorithm. In Table 8, we show the costs of the Mariani-Silver with the configurations $sd = 256$ and $d = 4$. VM costs are accounted for according to on-demand instance pricing without considering VM initialization times. The minimum billing period for VM is 1s. Although the cost of the parallel multithreaded version is lower than the versions that use serverless functions, our hybrid serverless executor is the most cost-effective implementation, achieving the highest price to performance ratio. This result is a consequence of the slightly better performance already seen in Fig. 5b combined with the economic cost reduction provided by the hybrid executor.

7. Conclusions and future work

In this work, we have validated the hypothesis that the high elasticity of the FaaS execution model can be a key enabler for effectively running applications with unbalanced workloads and irregular concurrency in the cloud. Through an evaluation focused on performance and cost, we see that FaaS can achieve a good performance for high-concurrency algorithms like UTS, thus masking the latency of invoking remote functions. We demonstrate better performance than a Spark cluster for similar cost. Further, we prove that a hybrid implementation combining local threads and serverless functions can be more cost-efficient than a pure serverless implementation for recursive algorithms that cannot sustainably reach high levels of concurrency. Finally, we show that a pure serverless solution can outperform a large EC2 VM in the Betweenness Centrality workload when using the same amount of virtual CPUs. Consequently, we are safe to conclude that a simple executor similar to those available in the Java Concurrency API can be sufficient to run high concurrency algorithms on serverless platforms, thus paving the way towards a more transparent and elastic execution of those problematic algorithms at large scale.

In the future, we would like to extend and further validate our work with other algorithms that involve stateful computations, like complex graph algorithms, that usually operate on I/O-bound irregular and unbalanced workloads. We also plan to study optimal hybrid deployment strategies to adapt executors to different application workloads.

CRediT authorship contribution statement

Gerard Finol: Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Gerard París:** Formal analysis, Investigation, Software, Writing – original draft, Methodology, Data curation. **Pedro García-López:**

Conceptualization, Formal analysis, Funding acquisition, Investigation, Project administration, Resources, Supervision. **Marc Sánchez-Artigas:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Project administration, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Code and data is open source and available, link can be found in the article.

Acknowledgments

This work has been partially funded by the EU Horizon program under grant agreements No. 825184 and No. 101092644, by the Spanish Government through project PID2019-106774RB-C22, and by the Ministry of Economic Affairs and Digital Transformation, in conjunction with the European Union-NextGenerationEU (within the framework of the PRTR and the MRR), through the CLOUDLESS UNICO I+D CLOUD 2022. Marc Sánchez-Artigas is a Serra Húnter Fellow. Gerard Finol is a URV Martí Franquès grant fellow.

References

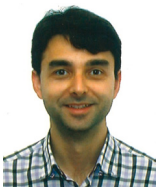
[1] A. Adinets, Nvidia developer blog – adaptive parallel computation with cuda dynamic parallelism, https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/, 2014.
[2] Amazon Web Services, retrieved, June 2023.
[3] A. Arjona, G. Finol, P.G. López, Transparent serverless execution of python multiprocessing applications, Future Gener. Comput. Syst. 140 (2023) 436–449, https://doi.org/10.1016/j.future.2022.10.038, https://www.sciencedirect.com/science/article/pii/S0167739X22003612.
[4] E. Arzuaga, D.R. Kaeli, Quantifying load imbalance on virtualized enterprise servers, in: Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering, WOSP/SIPEW'10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 235–242.
[5] AWS, Configuring lambda function memory, https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html.
[6] D.A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, T. Meuse, Hpcs scalable synthetic compact applications #2 graph analysis, http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.2.pdf, 2007.
[7] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, P. García-López, On the FaaS track: building stateful distributed applications with serverless architectures, in: Proceedings of the 20th International Middleware Conference, Middleware'19, 2019, pp. 41–54.
[8] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, P. García-López, Stateful serverless computing with crucial, ACM Trans. Softw. Eng. Methodol. 31 (3) (2022), https://doi.org/10.1145/3490386.
[9] U. Brandes, A faster algorithm for betweenness centrality, J. Math. Sociol. 25 (2) (2001) 163–177, https://doi.org/10.1080/0022250X.2001.9990249.
[10] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, C.S. Meiklejohn, Durable functions: semantics for stateful serverless, Proc. ACM Program. Lang. 5 (2021), https://doi.org/10.1145/3485510 (OOPSLA).
[11] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C.S. Meiklejohn, X. Zhu, Netherite: efficient execution of serverless workflows, Proc. VLDB Endow. 15 (8) (2022) 1591–1604, https://doi.org/10.14778/3529337.3529344.
[12] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, R. Katz, Cirrus: a serverless framework for end-to-end ml workflows, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC'19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 13–24.
[13] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, Y. Cheng Wukong, A scalable and locality-enhanced framework for serverless parallel computing, in: Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC'20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–15.
[14] D. Chakrabarti, Y. Zhan, C. Faloutsos R-MAT, A recursive model for graph mining, in: M.W. Berry, U. Dayal, C. Kamath, D.B. Skillicorn (Eds.), Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004, SIAM, 2004, pp. 442–446.
[15] G. Coulouris, J. Dollimore, T. Kindberg, G. Blair, Distributed Systems: Concepts and Design, 5th edition, Addison-Wesley Publishing Company, USA, 2011.
[16] J. Dinan, S. Olivier, G. Sabin, J.F. Prins, P. Sadayappan, C. Tseng, Dynamic load balancing of unbalanced computations using message passing, in: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007), IEEE, 2007, pp. 1–8.
[17] G.T. Eizaguirre, M. Sánchez-Artigas, A seer knows best: auto-tuned object storage shuffling for serverless analytics, J. Parallel Distrib. Comput. 183 (2024) 104763, https://doi.org/10.1016/j.jpdc.2023.104763, https://www.sciencedirect.com/science/article/pii/S0743731523001338.
[18] S. Fouladi, R.S. Wahby, B. Shacklett, K.V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, K. Winstein, Encoding, fast and slow: low-latency video processing using thousands of tiny threads, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17), 2017.
[19] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, K. Winstein, From laptop to lambda: outsourcing everyday jobs to thousands of transient functional containers, in: 2019 USENIX Annual Technical Conference (ATC'19), 2019, pp. 475–488.
[20] P. García-López, A. Slominski, S. Shillaker, M. Behrendt, B. Metzler, Serverless end game: disaggregation enabling transparency, CoRR, arXiv:2006.01251 [abs].
[21] P. Gimeno Sarroca, M. Sánchez-Artigas, Mlless: achieving cost efficiency in serverless machine learning training, J. Parallel Distrib. Comput. 183 (2024) 104764, https://doi.org/10.1016/j.jpdc.2023.104764, https://www.sciencedirect.com/science/article/pii/S074373152300134X.
[22] D. Grove, S.S. Hamouda, B. Herta, A. Iyengar, K. Kawachiya, J. Milthorpe, V. Saraswat, A. Shinnar, M. Takeuchi, O. Tardieu, Failure recovery in resilient x10, ACM Trans. Program. Lang. Syst. 41 (3) (2019), https://doi.org/10.1145/3332372.
[23] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht, Occupy the cloud: distributed computing for the 99%, in: Proceedings of the 2017 Symposium on Cloud Computing, SoCC'17, 2017.
[24] E. Jonas, et al., Cloud programming simplified: a Berkeley view on serverless computing, Tech. Rep. UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb. 2019.
[25] S. Joyner, M. MacCoss, C. Delimitrou, H. Weatherspoon Ripple, A practical declarative programming framework for serverless compute, CoRR, arXiv:2001.00222 [abs], arXiv:2001.00222.
[26] D.H. Liu, A. Levy, S. Noghabi, S. Burckhardt, Doing more with less: orchestrating serverless applications without an orchestrator, in: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), USENIX Association, Boston, MA, 2023, pp. 1505–1519, https://www.usenix.org/conference/nsdi23/presentation/liu-david.
[27] B.B. Mandelbrot, The Fractal Geometry of Nature, 3rd edition, W. H. Freeman and Comp., New York, 1983.
[28] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, M. Upton, Hyper-threading technology architecture and microarchitecture, Intel Technol. J. 6 (1) (2002).
[29] R. Munafo, Mariani/silver algorithm at mu-ency – the encyclopedia of the Mandelbrot set, https://mrob.com/pub/muency/marianisilveralgorithm.html, 2010.
[30] S. Olivier, J.F. Prins, Scalable dynamic load balancing using UPC, in: 2008 International Conference on Parallel Processing, ICPP 2008, IEEE Computer Society, 2008, pp. 123–131.
[31] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, C.-W. Tseng, Uts: an unbalanced tree search benchmark, in: Languages and Compilers for Parallel Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 235–250.
[32] G. París, P. García-López, M. Sánchez-Artigas, Serverless elastic exploration of unbalanced algorithms, in: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), 2020, pp. 149–157.
[33] J. Prins, J. Huan, B. Pugh, T. Chau-Wen, P. Sadayappan, Upc implementation of an unbalanced tree search benchmark, Tech. Rep. TR03-034, Univ. North Carolina at Chapel Hill, October 2003.
[34] Q. Pu, S. Venkataraman, I. Stoica, Shuffling, fast and slow: scalable analytics on serverless infrastructure, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19), 2019, pp. 193–206.
[35] R.B. Roy, T. Patel, V. Gadepally, D. Tiwari, Mashup: making serverless computing useful for hpc workflows via hybrid execution, in: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 46–60.
[36] J. Sampé, G. Vernik, M. Sánchez-Artigas, P. García-López, Serverless data analytics in the ibm cloud, in: Proceedings of the 19th International Middleware Conference Industry, Middleware'18, 2018, pp. 1–8.
[37] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, P. Garcia-Lopez, Outsourcing data processing jobs with lithops, IEEE Trans. Cloud Comput. (2021) 1, https://doi.org/10.1109/TCC.2021.3129000.
[38] M. Sánchez-Artigas, P.G. Sarroca, Experience paper: towards enhancing cost efficiency in serverless machine learning training, in: 22nd International Middleware Conference (Middleware'21), ACM, 2021, pp. 210–222.
[39] V.A. Saraswat, P. Kambadur, S. Kodali, D. Grove, S. Krishnamoorthy, Lifeline-based global load balancing, SIGPLAN Not. 46 (8) (2011) 201–212.
[40] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, J. Ragan-Kelley, numpywren: serverless linear algebra, CoRR, http://arxiv.org/abs/1810.09679, arXiv:1810.09679.

[41] V. Sreekanti, C. Wu, X.C. Lin, J. Schleier-Smith, J.M. Faleiro, J.E. Gonzalez, J.M. Hellerstein, A. Tumanov, Cloudburst: stateful functions-as-a-service, CoRR, http://arxiv.org/abs/2001.04592, arXiv:2001.04592.
[42] M. Szalay, P. Mátray, L. Toka, Real-time faas: towards a latency bounded serverless cloud, IEEE Trans. Cloud Comput. 11 (2) (2023) 1636–1650, https://doi.org/10.1109/TCC.2022.3151469.
[43] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V.A. Saraswat, A. Shinnar, M. Takeuchi, M. Vaziri, X10 and APGAS at petascale, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'14, Orlando, FL, USA, February 15-19, 2014, ACM, 2014, pp. 53–66.
[44] R.A. Tau Leng, J. Hsieh, V. Mashayekhi, R. Rooholamini, An empirical study of hyper-threading in high performance computing clusters, in: Linux HPC Revolution, vol. 45, 2002.
[45] The Java™ Tutorials, Thread pools oracle documentation, https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html, 2022, retrieved May 2022.
[46] L. Toader, A. Uta, A. Musaafir, A. Iosup, Graphless: toward serverless graph processing, in: 2019 18th International Symposium on Parallel and Distributed Computing (ISPDC), 2019, pp. 66–73.
[47] L. Xu, D. Saxena, N.J. Yadwadkar, A. Akella, I. Gupta Dirigo, Self-scaling stateful actors for serverless real-time data processing, arXiv:2308.03615 [cs], http://arxiv.org/abs/2308.03615.
[48] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat, M. Takeuchi, Glb: lifeline-based global load balancing library in x10, in: Proceedings of the First Workshop on Parallel Programming for Analytics Applications, PPAA'14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 31–40.
[49] W. Zhang, V. Fang, A. Panda, S. Shenker Kappa, A programming framework for serverless computing, in: Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC'20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 328–343.

Gerard Finol received his Bachelor's degrees in Computer Science and Mathematics from the University of Barcelona. During his studies, he participated in an exchange program at Ecole Polytechnique Fédérale de Lausanne (EPFL) as part of the Swiss-European Mobility Programme. He is currently a predoctoral researcher at the "Cloud and Distributed Systems Lab" research group at the University Rovira i Virgili. His research interests primarily lie in serverless computing, cloud computing, and distributed systems. Gerard actively contributes to European research projects such as NEARDATA and CLOUDSKIN.

Gerard París is a computer engineer with the Cloudlab Research Group at Universitat Rovira i Virgili. His research interests include cloud computing, distributed systems and software architectures. He received his BSc and MSc degrees in Computer Engineering from the Universitat Rovira i Virgili, Spain, in 2004 and 2006, respectively. He has actively participated in several EU funded research projects.

Pedro Garcia is professor of the Computer Engineering and Mathematics Department at the University Rovira i Virgili (Spain). He leads he "Cloud and Distributed Systems Lab" research group and he has coordinated large european research projects. In particular, he leads CloudStars (2023-2027), NearData (2023-2025), CloudSkin (2023-2025), and he participates as partner in EXTRACT (2023-2025). He also coordinated FP7 CloudSpaces (2013-1015), H2020 IOStack (2015-2017) and H2020 CloudButton (2019-2022).



Marc Sanchez-Artigas received his Ph.D. degree in Computer Science in 2009 from the Universitat Pompeu Fabra (UPF), Spain. During his Ph.D. studies, he worked at Ecole Polytechnique Fédérale de Lausanne (EPFL). In the same year, he joined the Universitat Rovira i Virgili, where he currently works as a Serra-Hunter Associate Professor. He received the Best Paper Award from IEEE LCN'07 and the Best Dataset Award from ACM IMC'15. He has published 80+ articles in important venues such as IEEE P2P, ACM/IFIP Middleware, IEEE ICDCS, IEEE ICDE, IEEE INFOCOM and USENIX FAST, among others. During the years, he has participated in multiple research European projects (CloudSpaces, IOStack, CloudButton, EXTRACT, NearData) and European mobility networks (CloudStars), and coordinated the Horizon Europe project CloudSkin (grant no. 101092646). He also has actively been involved in the coordination of several Spanish projects in Cloud and Edge Computing. He has been a technical program committee of several conferences and workshops on systems such as IEEE P2P, ACM SYSTOR, IPDPS, and Euro-Par, among others, and served as Guest Editor in the Computer Networks Journal, Elsevier. He has also organized various scientific national and international conferences and workshops.