Serverless Elastic Exploration of Unbalanced Algorithms

Gerard París
Universitat Rovira i Virgili
Tarragona, Spain
Email: gerard.paris@urv.cat

Pedro García-López
Universitat Rovira i Virgili, Tarragona, Spain
IBM Watson Research, Yorktown Heights, NY, USA
Email: pedro.garcia@urv.cat

Marc Sánchez-Artigas
Universitat Rovira i Virgili
Tarragona, Spain
Email: marc.sanchez@urv.cat

***Abstract*—In recent years, serverless computing and, in particular the Function-as-a-Service (Faas) execution model, has proven to be efficient for running parallel computing tasks. However, little attention has been paid to highly-parallel applications with unbalanced and irregular workloads. The main challenge of executing this type of algorithms in the cloud is the difficulty to account for the computing requirements beforehand. This places a burden on scientific users who very often make bad decisions by either overprovisioning resources or inadvertently limiting the parallelism of these algorithms due to resource contention. Our hypothesis is that the elasticity and ease of management of serverless computing can help users avoid such decisions, which may lead to undesirable cost-performance consequences for unbalanced problem spaces.**

In this work, we show that with a simple serverless executor pool abstraction one can achieve a better cost-performance trade-off than a Spark cluster of static size and large EC2 VMs. To support this conclusion, we evaluate two unbalanced algorithms: the Unbalanced Tree Search (UTS) and the Mandelbrot Set using the Mariani-Silver algorithm. For instance, our serverless implementation of UTS is able to outperform Spark by up to 55% with the same cost. This provides the first concrete evidence that highly-parallel, irregular workloads can be efficiently executed using purely stateless functions with almost zero burden on users — i.e., no need for users to understand non-obvious system-level parameters and optimizations.

***Keywords*-Serverless computing; FaaS; elasticity; UTS; Mariani-Silver;**

I. 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 like 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 stretched the limits and constraints of serverless functions to build more complex applications [1]. Simplicity and fine-grained accounting have attracted researchers to a new compute abstraction that offers opportunities to overcome its current limitations. So, serverless functions have been used to run data-parallel algorithms [2], [3], [4], video encoding [5], large-scale linear algebra operations [6] or stateful computations [7], [8], among other applications.

Unfortunately, no attention has been paid to a class of problems that exhibit high levels of parallelism, but with unbalanced and irregular workloads. Running these algorithms remains challenging for many users due to the number of decisions to be ahead of time to efficiently execute them, 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 [9] requires fine-tuning task load-balancing [10], especially at large scales [11], which leads the user 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 over the course of execution in static multithreaded systems. Provisioning a cluster of any static size will either slow down the job or leave the resources under-utilized. With a serverless solution, however, users may obviate the need to explicitly configure and manage on-demand compute units (e.g., VMs) [1], [2], [3], [6].

While simplicity is a key feature of serverless computing, an active research question is to determine which workloads are a natural fit for serverless computing. Prior literature has proven that a FaaS execution model can deliver high performance in many tasks [2], [3], [6], [5], [7]. 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, [1] reports serverless solutions with up to 7x increase in cost compared with a traditional solution with VMs.

Our focus in this work is on unbalanced algorithms such as UTS. More concretely, our goal is to ascertain whether the elasticity of serverless computing can achieve a better cost-performance trade-off than traditional VM-based solutions that do not have the flexibility required for

such workloads. This paper provides a positive answer to this question, making serverless computing a compelling approach to efficiently run unbalanced algorithms at large scale. Actually, we demonstrate in the paper that a basic serverless executor abstraction, mimicking the Java concurrency library, but launching Java threads as cloud functions, is enough to deliver a better cost-performance ratio than VM-based solutions. Simply put, with our novel executor, there is no need for users and developers to deal with non-obvious system-level components such as task schedulers to achieve good performance. It is enough for them to abide by the well-known, thread pool pattern to code their applications. For instance, our serverless implementation of UTS is able to outperform Spark by up to 55% with the same cost. We are the first to provide concrete evidence that highly-parallel, irregular workloads can be efficiently executed using purely stateless functions with almost zero burden on users. Another interesting feature of our executor is that can seamlessly combine local computing resources with remote cloud functions to achieve even better gains.

These excellent results start to pave the way towards a more transparent and elastic execution of algorithms at large scale. Coulouris's textbook [12] 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 – 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*. Our ultimate aim is thus to provide an elastic programming model that maintains the simplicity of development of single-machine applications when scaling up to thousands of cloud cores [13].

A. Contributions

In order to assess if FaaS is an appropriate execution environment for unbalanced algorithms, we implement and evaluate two challenging algorithms that exhibit nested parallelism: the Unbalanced Tree Search (UTS) [9] and the Mariani-Silver algorithm [14]. 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 [15], [11], [16]. 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 CUDA [17].

Remarkably, our novel serverless executor construct allows users to preserve the simplicity of the original algorithms, while achieving an optimal trade-off between cost and performance. Traditionally, algorithms such as UTS require significant effort to scale out [11]. Consequently, 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.

To verify the suitability of serverless computing to efficiently run unbalanced algorithms at large scale, we compare our serverless implementation of UTS with an existing Spark implementation [16]. Also, we compare the cost-performance ratio of the serverless implementations of both 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 HyperThreading on the efficiency of compute-intensive, unbalanced tasks.

In summary, our contributions are:

1) A serverless executor middleware, based on the Java concurrency library that executes Java threads over serverless functions. Our elastic programming model provides scaling transparency by efficiently combining local and remote computing entities.
2) Through the first implementation of UTS and Mariani-Silver algorithms with our serverless executor [1], we provide the first evidence that the FaaS execution model can be leveraged to efficiently run unbalanced and irregular task-parallel applications.
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.

II. Serverless Executor Middleware

We propose a general-purpose serverless executor based on the cloud thread abstraction [7] and the Java Concurrency library. 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. We have implemented a `ServerlessExecutor` that processes submitted Callable tasks and runs them as serverless functions.

Our 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. The client application has also the logic that decides to split tasks. Therefore, the client application collects return values from remote tasks through a local thread pool. These return values are inserted to a queue. A local thread keeps polling the queue and eventually splits pending tasks before invoking new remote threads. Serverless tasks are completely stateless.

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

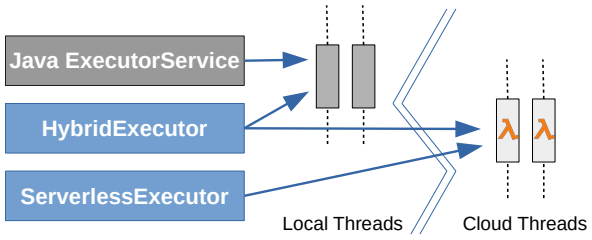

Figure 1: Serverless and Hybrid executors.

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 local thread pool must be limited 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.

To compare performance and cost of serverless functions against a local multithread version, we have implemented parallel multithread versions of the algorithms following the same master-worker approach. 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.

We have also implemented a `HybridExecutor` that combines local threads and remote serverless functions (as depicted in Figure 1). This hybrid version is 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, depending on the current local load: if there are pending tasks in the local thread pool queue, new tasks are scheduled to be run as serverless functions.

A. Limitations

The serverless executor model used has some limitations. First of all, every serverless approach has to deal with various constraints and limitations of serverless computing, namely limited duration of functions, limitations on concurrency and invocation frequency, lack of network connectivity between functions, and stateless nature of functions. Second, an adaptation effort is needed to overcome some of these limitations and, 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. And finally, considering the mentioned limitations and the lack of an efficient shared memory model for serverless, our first approach is stateless. The tasks launched as cloud functions will not share data with each other. Instead, all data will be received and returned as parameters. This limits the range of algorithms that can be analyzed. However, it is an intentional decision to start with a simpler and pure-serverless approach, and we left for future work the adoption of a distributed shared memory approach that will be needed to support other algorithms like graph and data analytics applications.

Table I: 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	$270,751,679,750$

III. Methodology

A. Algorithms

In this work we target two of these irregular and unbalanced algorithms: UTS and Mariani-Silver.

1) UTS: The Unbalanced Tree Search (UTS) benchmark [15] is a challenging algorithm that is commonly used to evaluate the performance of different parallel frameworks under an irregular workload. 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 used a geometric distribution with an expected branching factor $b_0 = 4$ and a depth cut-off d between 14 and 18. Table I shows how the number of nodes increases exponentially as we increase the depth cut-off.

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. Another parameter that can be configured is the split factor: the maximum amount of partitions a subtree is divided into before launching new parallel tasks. In listing 1 we see that each remote task (line 33) receives a `bag` parameter that encapsulates the subtree assigned to the task. Once the `bag` is processed, it is returned (line 42) to the master program where it is added to a queue (line 29). Another thread is responsible to retrieve bags from the queue (line 5), resizing subtress (line 8), and subsequently launch new parallel tasks (line 9).

```
1 private void uts(List<Bag> bags) {
2   parallelize(bags);
3
4   while(<still active threads || queue not empty>) {
5     Bag bag = queue.poll();
6     if (bag != null) {
7       activeThreads.addAndGet(-1);
8       bags = resizeBag(bag, splitFactor);
9       parallelize(bags);
```

```
10      }
11    }
12  }
13
14  private void parallelize(List<Bag> bags) {
15    activeThreads.addAndGet(bgas.size());
16
17    for (Bag bag : bags) {
18      Future<Object> future = localExecutor
19        .submit(new LocalUTSCallable(bag, iters));
20    }
21  }
22
23  class LocalUTSCallable implements Callable<Object> {
24    ...
25    public Object call() throws Exception {
26      Future<Result> future = serverlessExecutor.submit(
27        new RemoteUTSCallable(bag, iters));
28      Result result = future.get();
29      queue.offer(result.getBag());
30    }
31  }
32
33  class RemoteUTSCallable implements
        Callable<TMResult>, Serializable {
34    ...
35    public Result call() throws Exception {
36      // initialization of MessageDigest
37      ...
38      for (int n = numberOfIterations; n > 0 && bag.size
            > 0; --n) {
39        // traverse one node
40        ...
41      }
42      return new Result(bag);
43    }
44  }
```

Listing 1: UTS serverless implementation

2) Mariani-Silver: The simplest algorithm to render the Mandelbrot Set is the naive Escape Time algorithm, where a repeating calculation is performed for every point in the plot area. However, there are several optimization 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. 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; otherwise the subrectangle is divided again into two or more smaller rectangles. 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.

Listing 2 shows the main code of the Mariani-Silver algorithm. When the rectangle has to be split (line 12), new recursive tasks are generated. This kind of nested loops are 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.

```
1  public Result call() {
2    Result result = new Result(rectangle);
3    if (borderHasCommonDwell(rectangle)) {
4      result.setNextAction(Result.Action.FILL);
5      result.setDwellToFill(rectangle.getBorderDwell());
6    } else if (rectangle.getDepth() >= MAX_DEPTH) {
7      // per-pixel evaluation of the rectangle
8      int[][] dwellArray = evaluate(rectangle);
9
         result.setNextAction(Result.Action.SET_DWELL_ARRAY);
10     result.setDwellArray(dwellArray);
11   } else {
12     result.setNextAction(Result.Action.SPLIT);
13     // New tasks will be generated
14   }
15   return result;
16 }
```

Listing 2: Mariani-Silver Callable code

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 of 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 subdivisions, the split factor, and the maximum depth is configurable.

B. 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.

Regarding cost, we calculate the cost metric using the serverless provider listed prices. 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 (Eq. 1):

$$Cost_{Serverless} = Cost_{Invocations} + Cost_{Execution} \quad (1)$$

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

$$Cost_{Invocations} = 0.0000002\$ * n \quad (2)$$

And 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):

$$Cost_{Execution} = 0.0000166667\$ * \frac{\text{memory in MB}}{1024\text{MB}} * \sum_{i=0}^{n} t_i \quad (3)$$

To discuss trade-offs between cost and performance, we also refer to the price to performance ratio, a useful metric to show the throughput that can be achieved per dollar spent.

C. 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 $1,792$MB of memory. According to AWS documentation, at this amount of memory, each function has the equivalent to one full vCPU of compute time. 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.

IV. Performance analysis

In terms of performance, current serverless computing platforms have some limitations that must be considered beforehand. One of the main problems of serverless applications is the predictability of performance [1]. One source of unpredictability is the known problem of cold starts of cloud functions: the time it takes to start a new function container, to initialize the software environment, and to initialize the specific user code. Depending on the application, this startup latency may be high. In this sense, the advantage of task-parallel applications is that these cold starts only affect the first batch of functions, and successive tasks benefit from having already warm containers.

Obviously, the predictability of performance in FaaS is lower than in traditional IaaS VM instances, and even lower than in HPC dedicated clusters in computing centers. However, there is an increasing interest in moving HPC workloads to the cloud, and particularly to FaaS platforms, due to its elasticity, availability, and pay-as-you-go model.[18]

For task-parallel workloads in FaaS, one of the main factors that affect final performance is the overhead of running a remote task. Table II compares the overheads to invoke a dummy task as a serverless 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 a serverless function (~13ms) is three orders of magnitude greater than the overhead of a local thread, it is still low enough to permit efficient execution of parallel workloads.

Table II: Average invocation overheads

	Overhead
Serverless Functions	13ms
Local threads	18μs

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 III 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. For comparison purposes, the serverless version is limited here to launch a maximum of 96 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. Despite the inherent overheads, the serverless version appears to be surprisingly faster than the parallel version. This can be attributed to the effect of Intel's Hyper-threading (HT) to a compute-intensive task like this. When running the parallel version in the VM, each pair of vCPUs or logical CPUs share the same physical core. In effect, with HT disabled, and thus reducing in half the number of logical CPUs, the throughput only decreases slightly and the parallel efficiency reaches a more acceptable rate of 76.78%.

This is an interesting insight that must be considered when comparing efficiency in virtual machines versus serverless functions. Although, according to AWS Lambda documentation, each function with $1,792$ MB is equivalent to one vCPU, a 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.

A. 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 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

Table III: Performance and parallel efficiency of UTS.

	Logic CPUs	Depth	Time(s)	Throughput (M nodes/s)	Parallel efficiency
Sequential	1	14	75.863	13.94	
AWS Lambda	96	17	74.55	907.94	67.84%
`c5.24xlarge`	96	17	113.9	594.22	44.40%
`c5.24xlarge` (HT disabled)	48	17	131.73	513.82	76.78%

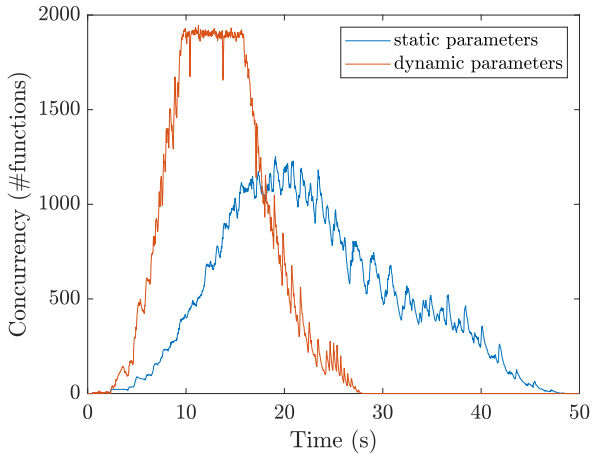

Figure 2: Concurrency and total execution time for serverless version of UTS ($d = 18$).

optimization to dynamically modify the split factor and the number of nodes processed depending on the current level of concurrency. By defining some ad-hoc concurrency levels, we assign a large split factor until the maximum concurrency is achieved, and then slowly start to decrease the split factor as the measured concurrency also goes down defined levels. Likewise, the number of processed nodes is kept lower at the beginning and the end of the execution, and is increased when high concurrency levels are achieved. Figure 2 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. Thanks to this, this version finishes in 27.9s. It must be noted that the dynamic version launches around $84,000$ serverless functions, while the static version only uses around $41,000$. Despite that, the cost of the dynamic version is only 9% expensive than the static version ($0.86 versus $0.79).

B. Preliminary hybrid executor evaluation

To assess the performance implications of the hybrid executor, we conduct some preliminary evaluations using the Mariani-Silver algorithm to generate a 4096x4096 pixels image of the Mandelbrot Set, where the fractal space is divided adaptively using dynamic parallelism. To maximize the performance benefits of locality, the hybrid executor followed a simple policy rule: dispatch tasks to serverless functions only if there are not local threads available in the pool.

Setup: The maximum dwell of a point was configured to 5 million iterations. Each rectangle was subsequently divided in 4 parts. We run 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. 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: Figure 3 shows the total execution time of the algorithm under our three different implementations. As can be seen in the figure, our serverless and hybrid executors obtain better performance than a large VM, achieving a speedup factor $>$ 2X. Non-surprisingly, the performance of the hybrid executor over the pure serverless executor is not so high for this algorithm. The main reason is that the Mariani-Silver algorithm is 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. Either way, the combination of local and remote cloud resources makes a case worth exploring in the future.

V. 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 [1].

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

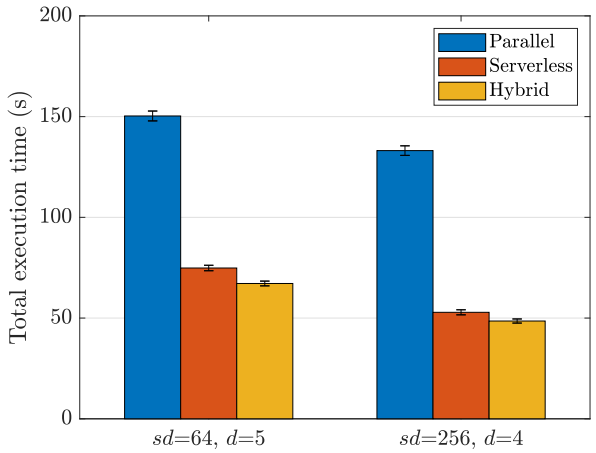

Figure 3: Total execution time of Mariani-Silver for different implementations and configurations. The error bars represent the standard deviation of all measurements.

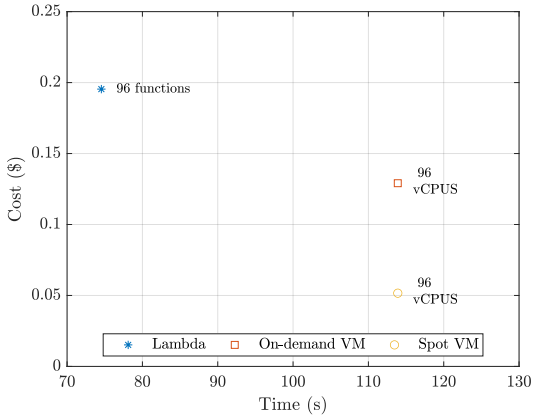

Figure 4: Cost-performance comparison of the parallel and serverless version of UTS at depth 17 with comparable CPU resources.

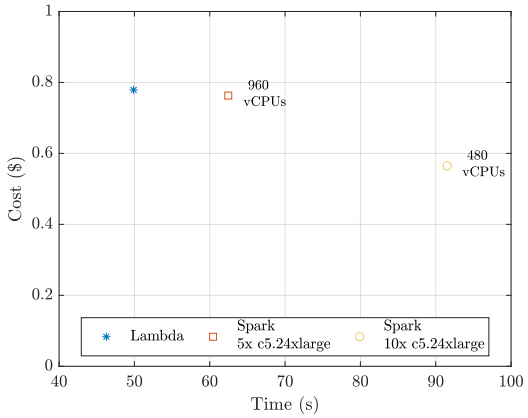

Figure 5: Cost-performance comparison of the Spark and serverless version of UTS at depth 18.

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 more lighter serverless containers. Moreover, application programmer have 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.

Figure 4 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. Figure 5 shows a cost-performance comparison of the serverless version of UTS with a Spark implementation [16] 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 (totalling 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:

$$Cost_{EMR} = \frac{\text{t}}{3600} * (10 * 4.35\$ \text{ per hour} + 0.48\$ \text{ per hour}) \quad (4)$$

We see that the unoptimized serverless version depicted in Figure 5 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 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 IV, the serverless version is able to outperform Spark by up to 55%.

In section IV we already saw that our serverless and hybrid executors can obtain better performance than a multi-threaded executor in a large virtual machine for the Mariani-Silver algorithm. In Table IV, 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.

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

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

VI. Related work

UTS benchmark was initially proposed by Prins et al. [9] in 2003 and has since been used as an irregular application for evaluating load-balancing algorithms in several parallel computing architectures and using different programming models like MPI [19], Unified Parallel C [20] or X10's APGAS [10]. However, all these programming models are not elastic. The implementation of UTS presented in this work is the first that tackles an elastic environment.

In the field of serverless computing, several execution frameworks have been proposed to leverage scalability. PyWren [2] introduces a BSP-style serverless execution framework with a map abstraction to run Python functions in parallel. Others like ExCamera [5] or Crucial [7] adopt the thread abstraction to map serverless functions. All these frameworks have mainly been applied to embarrassingly parallel problems. Instead, irregular algorithms have received little attention, and are only considered in frameworks designed for specific applications, like graph processing (Graphless [21]) or large-scale linear algebra (numpywren [6]).

VII. Conclusion

In this paper, we have validated the hypothesis that inherent elasticity of the FaaS execution model can benefit unbalanced and irregular task-parallel applications. 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. We also see that a hybrid implementation combining local threads and serverless functions, can be more cost-efficient than completely serverless versions for a recursive algorithm that cannot sustainably reach high levels of concurrency. We conclude that a general-purpose programming model that mimicks Java Concurrency API can be efficiently used to run high concurrency algorithms in serverless, thus paving the way towards a more transparent and elastic execution of 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 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.

Acknowledgment

This work has been partially supported by the EU Horizon 2020 programme under grant agreement No 825184, and by the Spanish Government (PID2019-106774RB-C22). Marc Sánchez-Artigas is a Serra Húnter Fellow.

References

[1] E. Jonas et al., "Cloud programming simplified: A berkeley view on serverless computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, Feb 2019.

[2] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in Proceedings of the 2017 Symposium on Cloud Computing, ser. SoCC'17, 2017.

[3] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the ibm cloud," in Proceedings of the 19th International Middleware Conference Industry, ser. Middleware '18, 2018, pp. 1–8.

[4] S. Joyner, M. MacCoss, C. Delimitrou, and H. Weatherspoon, "Ripple: A practical declarative programming framework for serverless compute," CoRR, vol. abs/2001.00222, 2020.

[5] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and 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.

[6] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," CoRR, vol. abs/1810.09679, 2018.

[7] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures," in Proceedings of the 20th International Middleware Conference, ser. Middleware '19, 2019, p. 41–54.

[8] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," CoRR, vol. abs/2001.04592, 2020.

[9] J. Prins, J. Huan, B. Pugh, T. Chau-Wen, and P. Sadayappan, "Upc implementation of an unbalanced tree search benchmark," Univ. North Carolina at Chapel Hill, Tech. Rep. TR03-034, October 2003.

[10] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," SIGPLAN Not., vol. 46, no. 8, p. 201–212, Feb. 2011.

[11] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. A. Saraswat, A. Shinnar, M. Takeuchi, and 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.

[12] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, Distributed Systems: Concepts and Design, 5th ed. USA: Addison-Wesley Publishing Company, 2011.

[13] P. García-López, A. Slominski, S. Shillaker, M. Behrendt, and B. Metzler, "Serverless end game: Disaggregation enabling transparency," CoRR, vol. abs/2006.01251, 2020.

[14] R. Munafo, "Mariani/silver algorithm at mu-ency – the encyclopedia of the mandelbrot set," 2010. [Online]. Available: https://mrob.com/pub/muency/marianisilveralgorithm.html

[15] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "Uts: An unbalanced tree search benchmark," in Languages and Compilers for Parallel Computing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 235–250.

[16] D. Grove, S. S. Hamouda, B. Herta, A. Iyengar, K. Kawachiya, J. Milthorpe, V. Saraswat, A. Shinnar, M. Takeuchi, and O. Tardieu, "Failure recovery in resilient x10," ACM Trans. Program. Lang. Syst., vol. 41, no. 3, Jul. 2019.

[17] A. Adinets, "Nvidia developer blog – adaptive parallel computation with cuda dynamic parallelism," 2014. [Online]. Available: https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/

[18] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "Funcx: A federated function serving fabric for science," in Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, ser. HPDC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 65–76.

[19] J. Dinan, S. Olivier, G. Sabin, J. F. Prins, P. Sadayappan, and C. Tseng, "Dynamic load balancing of unbalanced computations using message passing," in 21th International Parallel and Distributed Processing Symposium (IPDPS 2007). IEEE, 2007, pp. 1–8.

[20] S. Olivier and 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.

[21] L. Toader, A. Uta, A. Musaafir, and A. Iosup, "Graphless: Toward serverless graph processing," in 2019 18th International Symposium on Parallel and Distributed Computing (ISPDC), 2019, pp. 66–73.