WUKONG: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing

Benjamin Carver
George Mason University
bcarver2@gmu.edu

Jingyuan Zhang
George Mason University
jzhang33@gmu.edu

Ao Wang
George Mason University
awang24@gmu.edu

Ali Anwar
IBM Research–Almaden
Ali.Anwar2@ibm.com

Panruo Wu
University of Houston
pwu7@uh.edu

Yue Cheng
George Mason University
yuecheng@gmu.edu

ABSTRACT

Executing complex, burst-parallel, directed acyclic graph (DAG) jobs poses a major challenge for serverless execution frameworks, which will need to rapidly scale and schedule tasks at high throughput, while minimizing data movement across tasks. We demonstrate that, for serverless parallel computations, decentralized scheduling enables scheduling to be distributed across Lambda executors that can schedule tasks in parallel, and brings multiple benefits, including enhanced data locality, reduced network I/Os, automatic resource elasticity, and improved cost effectiveness. We describe the implementation and deployment of our new serverless parallel framework, called WUKONG, on AWS Lambda. We show that WUKONG achieves near-ideal scalability, executes parallel computation jobs up to 68.17× faster, reduces network I/O by multiple orders of magnitude, and achieves 92.96% tenant-side cost savings compared to numpywren.

ACM Reference Format:
Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. WUKONG: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing . In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421286

1 INTRODUCTION

In recent years, a new cloud computing model called serverless computing or Function as a Service (FaaS) [21] has emerged. Serverless computing enables a new way of building and scaling applications and services by allowing developers to break traditionally monolithic server-based applications into finer-grained cloud functions. Developers write function logic while the service provider performs the notoriously tedious tasks of provisioning, scaling, and managing the backend servers that the functions run on [38].

Serverless computing solutions are growing in popularity and finding their way into both commercial clouds (e.g., AWS Lambda, Google Cloud Functions, and Azure Functions) and open source projects (e.g., OpenWhisk). While serverless platforms were originally intended for event-driven, stateless applications [1], a recent trend is the use of serverless computing for more complex, stateful, parallel applications.

Some types of compute- and data-intensive applications are inherently parallelizable and can be structured as a directed acyclic graph (DAG) of short, fine-grained tasks [2, 36, 39, 49]. The large-scale parallelism and auto-scaling services provided by serverless platforms makes them well-suited for such kinds of burst-parallel fine-grained tasks that characterize DAG-based parallel computation workflows. Burst-parallel applications include data analytics [39], optimization algorithms [13], and real-time machine learning classifications such as support vector machines (SVM) [20, 33, 55]; these applications typically demand low-latency scheduling [49] with large-scale parallelism [26].

FaaS providers charge function execution time at a fine granularity – AWS Lambda bills on a per-invocation basis. Workloads with short tasks can take advantage of this fine-grained pay-per-use pricing model to keep monetary costs low. Consequently, serverless computing can be leveraged by next-generation, burst-parallel workloads in high-performance computing (HPC) and data analytics.

Migrating such applications from a traditional serverful deployment to a serverless platform presents unique opportunities. Traditional serverful deployments rely on existing workflow management frameworks such as MapReduce [34], Apache Spark [54], Sparrow [49], and Dask [9] to provide

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10.
https://doi.org/10.1145/3419111.3421286

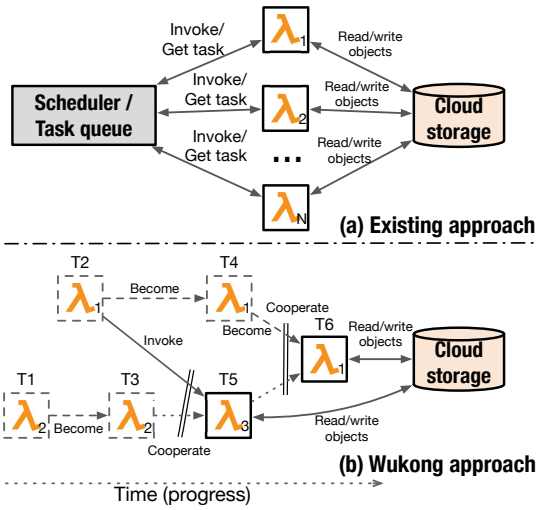

Figure 1: In (a), the central scheduler tracks all task completions, updates all task dependencies, and identifies all ready tasks. The scheduler dispatches ready tasks to Lambda executors; or the scheduler deposits ready tasks into a shared work queue, and a pool of Lambda executors contend for the queued tasks. Intermediate task inputs and outputs are stored outside of the executors, which reduces data locality. In (b), task scheduling is performed by a fleet of Lambda executors that schedule and execute their assigned tasks in parallel and cooperate to ensure that task dependencies are satisfied. Intermediate task inputs and outputs may be stored inside the executors, which increases data locality. This approach also enables fine-grained and automatic Lambda resource elasticity, as Lambda executors finish assigned tasks and return (e.g., Lambda 2).

a logically centralized scheduler for managing task assignments and resource allocation. The scheduler traditionally has various objectives, including load balancing, maximizing cluster utilization, ensuring task fairness, and so on. However, a traditional serverful scheduler is not required by serverless computing. This is because: (1) FaaS providers are responsible for managing the "servers" (i.e., where the task executors are hosted); and (2) serverless platforms typically provide a *nearly unbounded* amount of *ephemeral* resources. As a result, a hypothetical serverless parallel computing framework may not necessarily care about traditional "scheduling"-related metrics (such as load balancing and cluster utilization), since the framework has no control over where tasks are executed. (The service provider, of course, cares about these metrics.)

Yet, designing an efficient serverless-oriented parallel computing framework introduces unique challenges. First, while serverless platforms (e.g., AWS Lambda) promise to offer superior elasticity and auto-scaling properties, the serverless invocation model imposes non-trivial scheduling overhead. Unlike a typical serverful parallel framework where the central scheduler directly communicates with each worker process using TCP [3, 34], in a serverless setup, the scheduler can dispatch tasks to serverless workers in one of three ways.

Figure 1(a) depicts a high-level overview of all three methods. In method #1, the scheduler invokes a Lambda function (using the HTTP protocol) to dispatch the task code and execute the task. Note that with this method, there is a one-to-one association between tasks and Lambda functions. Given an average invocation overhead of 50 ms (typical for AWS Lambda functions), the scheduler could quickly become a performance bottleneck, especially for large and complex jobs with thousands of tasks. These observations indicate that a naive attempt to simply port an existing serverful DAG framework to serverless computing will be unsuccessful. In order to create a performant, cost-effective serverless DAG engine, new techniques must be developed to fully take advantage of the characteristics of the serverless platform.

In method #2, the scheduler launches short-lived[1] Lambda executors as workers that establish TCP connections with the scheduler and receive RPC requests for task processing. Example frameworks include ExCamera [37] and PyWren [42]. Task executors within these frameworks may execute several tasks as opposed to just one as with the first method.

Similarly, in method #3, the scheduler places tasks in a shared work queue. These tasks are retrieved from the queue by serverless executors; state-of-the-art systems such as numpywren [52] launch stateless Lambda executors that connect to a centralized shared queue and constantly retrieve tasks from the queue. Frameworks using this method may have a component separate from the central scheduler that is responsible for invoking the AWS Lambda executors. This is sometimes referred to as a "provisioner". In the latter two approaches, as shown in Figure 1(a), a tightly synchronized central scheduler tracks all task completions, updates all task dependencies, and identifies any ready tasks. The scheduler dispatches ready tasks to Lambda executors, or the scheduler deposits ready tasks into a shared work queue, and a pool of Lambda executors contend for the queued tasks. Intermediate task inputs and outputs are stored externally, which reduces data locality.

The second challenge is that serverless platforms come with inherent constraints, including bandwidth-limited, outbound only network connectivity; therefore, serverless workflows must rely on external cloud store for intermediate data storage and exchange, which creates excessive data movement overhead. Data locality enhancement is thus critical for minimizing communication costs.

Researchers have developed serverless parallel computing frameworks that support parallel job processing [37, 42, 52];

[1] Lambda functions may run up to 900 seconds in AWS cloud.

however, these solutions do not fully address the aforementioned performance issues of efficient scheduling and data locality, which leads to long scale-out delays, sub-optimal performance, and higher monetary cost. To this end, we design and build a new serverless parallel computing framework called WUKONG[2]. WUKONG is a serverless-oriented, decentralized, locality-aware, and cost-effective parallel computing framework. *The key insight of WUKONG is that partitioning the work of a centralized scheduler (i.e., tracking task completions, identifying and dispatching ready tasks, etc.) across a large number of Lambda executors, can greatly improve performance by permitting tasks to be scheduled in parallel, reducing resource contention during scheduling, and making task scheduling data locality-aware, with automatic resource elasticity and improved cost effectiveness.*

Scheduling is decentralized by partitioning a DAG into multiple, possibly overlapping, subgraphs. Each subgraph is assigned to a task Executor (implemented as an AWS Lambda function runtime) that is responsible for scheduling and executing tasks in its assigned subgraph. This decentralization brings multiple key benefits:

Enhanced data locality and reduced resource contention: Decentralization improves the data locality of scheduling. Unlike PyWren [42] and numpywren [52], which require executors to perform network I/Os to obtain each task they execute (since numpywren's task executor is completely stateless), WUKONG preserves task dependency information on the Lambda side. This allows Lambda executors to cache intermediate data and schedule the downstream tasks in their subgraph locally, i.e., without constant remote interaction with a centralized scheduler.

Harnessing scale and local optimization opportunities: Decentralizing scheduling allows an Executor to make local data-aware scheduling decisions about the level of task granularity (or parallelism) appropriate for its subgraph. Agile executors can scale out compute resources in the face of burst-parallel workloads by partitioning their subgraphs into smaller graphs that are assigned to other executors for an even higher level of parallel task scheduling and execution. Alternately, an executor can execute tasks locally, when the cost of data communication between the tasks outweighs the benefit of parallel execution.

Automatic resource elasticity and improved cost effectiveness: Decentralization does not require users to explicitly tune the number of active Lambdas running as workers and thus is easier to use, more cost effective, and more resource efficient.

We make the following contributions in this paper.

- We thoroughly explore the problem space of serverless parallel computing framework design. For a range of parallel computation applications, we identify issues of the state-of-the-art serverless frameworks—task scheduling, data locality, and resource efficiency (monetary cost effectiveness).
- We present the design and implementation of WUKONG, a new serverless parallel computing framework that solves the identified issues. WUKONG synergizes a set of optimization techniques, including decentralized scheduling, task clustering, and delayed I/O. These techniques together achieve near-ideal scalability, reduce data movement over the network, enhance data locality, and improve cost effectiveness.
- We evaluated WUKONG extensively on AWS. Our results show that WUKONG reduces network I/O by many orders of magnitude and achieves up to 68.17× higher performance than numpywren, while reducing the monetary cost by as much as 92.96%.

[2] https://mason-leap-lab.github.io/Wukong

2 BACKGROUND AND MOTIVATION

2.1 Why Serverless?

Serverless Computing handles virtually all system administration tasks, making it easier for developers to use a near-infinite amount of cloud resources, including bundled CPUs and memory, object stores, and a lot more [43]. Service providers provide a flexible interface for defining serverless functions, which allows developers to focus on core application logic. Service providers in turn auto-scale function executions in a demand-driven fashion, hiding tedious server configuration and management tasks from the users.

General Constraints and Limitations. Service providers place limits on the use of cloud resources to simplify resource management. Take AWS Lambda for example: users configure Lambda's memory and CPU resources in a bundle. Users can choose a memory capacity between 128MB–3008MB in 64MB increments. Lambda will then allocate CPU power linearly in proportion to the amount of memory configured. Each Lambda function can run at most 900 seconds and will be forcibly stopped when the time limit is reached. In addition, Lambda only allows outbound TCP network connections and bans inbound connections and the UDP protocol.

Opportunities. Running large-scale, burst-parallel computation jobs has long been challenging for domain scientists due to the complexity of configuring, provisioning, and managing compute clusters [17, 18]. By taking over system administration and automatically providing capability to launch thousands of processes with no advance notice, the emerging serverless computing model *seems* to provide a foundation that will attract domain scientists and data analysts. *However, to fully unleash the potential of serverless computing, an efficient serverless-optimized parallel computing framework is needed.*

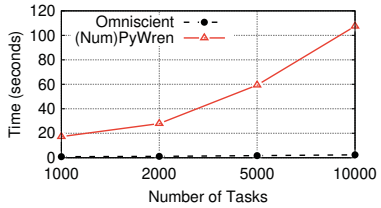

Figure 2: (Num)PyWren scaling tasks on AWS Lambda.

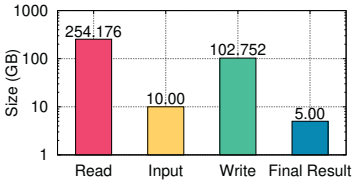

Figure 3: Numpywren GEMM read and write amplification.

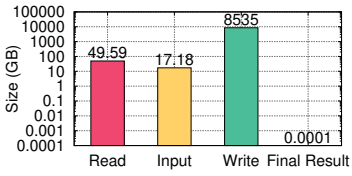

Figure 4: Numpywren TSQR read and write amplification.

2.2 Challenges

We build on our experience with serverless frameworks to synthesize the performance requirement for an ideal, serverless, parallel computing framework, and discuss why current solutions are not able to meet these performance requirements of burst-parallel applications at both the task scheduling and the data locality level.

Challenge to Rapidly Scale Out. A family of burst-parallel computation jobs (e.g., data analytics [49], machine learning classifications [20], etc.) are dominated by short-lived tasks with a span ranging from hundreds of milliseconds (ms) to tens of seconds [39, 42, 49, 55]. Such applications pose a difficult scheduling challenge to the serverless computing platforms. This is because, while serverless computing promises to deliver elastic auto-scaling feature in response to bursts of concurrent workloads, serverless function invocations incur non-negligible overhead, thus creating a scheduling bottleneck with slow scaling out.

PyWren is a state-of-the-art serverless execution engine [42]. PyWren enables users to program MapReduce-like applications on serverless platforms such as AWS Lambda. Numpywren [52] is a system for linear algebra built on top of PyWren. Numpywren uses PyWren's existing infrastructure to deploy their own serverless task executors, which run as a user-defined function within PyWren's own Lambda executors. In order for numpywren to scale the size of their Lambda cluster, numpywren invokes additional PyWren executors (using PyWren's own API) from their central scheduler. Based on this design, numpywren relies heavily on PyWren for Lambda scaling and management.

Figure 2 shows PyWren's ability to schedule large numbers of no-op tasks on AWS Lambda-based executors. Ideally, an omniscient serverless scheduler should be able to take full advantage of the massive parallelism offered by serverless computing and rapidly scale to thousands of Lambda executors in seconds in response to bursty, highly parallel workloads. PyWren uses a centralized scheduling approach, where it employs 64 threads for task scheduling and invocation; it takes almost 2 minutes to scale out to 10,000 Lambda executors[3]. To make it worse, a serverless framework like PyWren cannot always keep thousands of Lambda executors actively running (unlike the worker servers to a typical serverful parallel framework such as MapReduce [34]), so it has to constantly invoke many Lambdas in response to bursts of job tasks.

This serves to illustrate the failure of existing serverless execution frameworks to fully utilize serverless computing's elastic auto-scaling property.

Challenge of Excessive Data Movement. Parallel applications require intermediate data exchange among tasks. Direct task-to-task data communication is naturally supported in traditional serverful parallel computing frameworks such as MPI [18], MapReduce [34], and Dask [9]. However, data exchange in serverless applications may be supported only indirectly, through the use of remote cloud storage systems.

Consider the data exchanged during the execution of $25k \times 25k$ GEMM (general matrix multiplication) and $8,192k \times 128$ TSQR (tall skinny QR) on numpywren. Figure 3 and Figure 4 present a comparison between pure input and output sizes and the amount of data transferred during the two workloads respectively. For GEMM, the total quantity of data read is more than 25× the size of the input data while the total quantity of data written is more than 20× the size of the output. This trend is further exemplified by TSQR. While the amount of data read is only 2.88× greater than the input data size, the amount of data written over $65M\times$ greater than the output size. This is because numpywren and PyWren adopt a stateless Lambda executor design where a task can be dispatched to any Lambda executor; once dispatched to a Lambda, the task simply performs the following four steps: 1) reads its input data (the intermediate data generated by one or multiple upstream tasks) from cloud storage (numpywren uses S3), 2) performs computation, 3) writes the intermediate results (as output) to the cloud storage, and 4) returns. While the stateless design seems to be a good fit for serverless platforms, it does not preserve data locality, which results in excessive data movement.

This stresses a strong need for reducing data movement and increasing data locality in serverless frameworks.

3 WUKONG DESIGN

In this section, we present the system design of Wukong. This design is motivated by the observations that existing

[3] As did in [37], we also performed warmup operations to make sure each Lambda invocation does not incur a cold start [22].

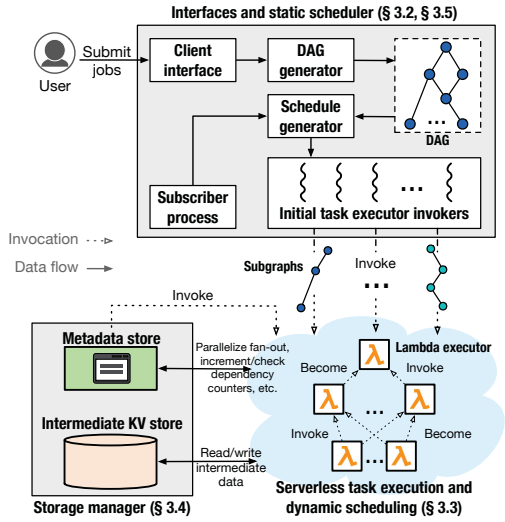

Figure 5: Overview of WUKONG architecture.

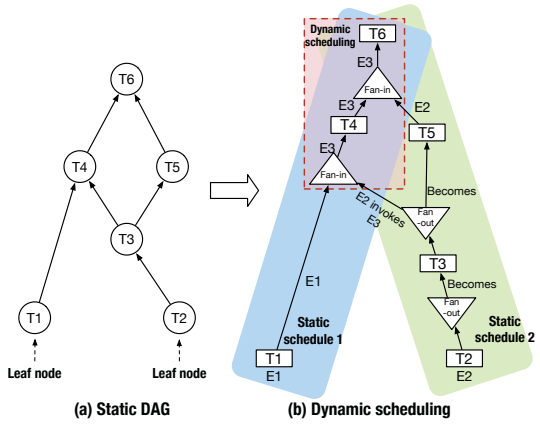

Figure 6: Static DAG (a) and dynamic scheduling (b). WUKONG's Executors coordinate in the area inside the dashed box in (b) using dynamic scheduling. "T1" denotes Task 1. "E1" denotes Lambda Executor 1.

serverless parallel frameworks are slow to scale out and are bottlenecked by excessive data movement (§2).

3.1 High-Level Design

Figure 5 shows the high-level design of WUKONG. The design consists of three major components: a static schedule generator which runs on Amazon EC2, a pool of Lambda Executors, and a storage cluster.

Scheduling in WUKONG is decentralized and uses a combination of static and dynamic scheduling. A static schedule is a subgraph of the DAG. Each static schedule is assigned to a separate Executor. An Executor uses dynamic scheduling to enforce the data dependencies of the tasks in its static schedule. An Executor can invoke additional Executors to increase task parallelism, or cluster tasks to eliminate any communication delay between them. Executors store intermediate task results in an elastic in-memory key-value storage (KVS) cluster (hosted using AWS Fargate [5]; see §3.4) and job-related metadata (e.g., counters) in a separate KVS that we call metadata store (MDS).

3.2 Static Scheduling

WUKONG users submit a Python computing job to the DAG generator, which uses the Dask library to convert the job into a DAG. The Static-Schedule Generator generates *static schedules* from the DAG. For a DAG with n leaf nodes, n static schedules are generated. A static schedule for leaf node `L` contains all of the task nodes that are reachable from `L` and all of the edges into and out of these nodes. The data for a task node includes the task's code and the KVS keys for the task's input data. The schedule for `L` is easily computed using a depth-first search (DFS) that starts at `L`. Lambda Executors notify the static scheduler when a final result has been stored in Redis by sending a message to the static scheduler's subscribe process. Upon receiving a message, the static scheduler will download final results and return them to the user automatically.

Figure 6(a) shows a DAG with two leaf nodes. Figure 6(b) shows the two static schedules that are generated from the DAG: `Schedule 1` (blue) and `Schedule 2` (green).

A static schedule contains three types of operations: task execution, fan-in and fan-out. To simplify our description, when DAG task `Tx` is followed immediately by task `Ty`, and `Tx (Ty)` has no fan-out (fan-in), we add a trivial fan-out operation between `Tx` and `Ty` in the static schedule. This fan-out operation has one incoming edge from `Tx` and one outgoing edge to `Ty`, i.e., there is no actual fan-out. In Figures 6(a) and (b), this is the case for DAG tasks `T2` and `T3`.

A fan-in task `T` may depend on tasks that will be executed by different Executors, e.g., task `T4` in Figure 6. The dynamic scheduling technique described below ensures that `T`'s data dependencies are satisfied and that `T` is executed by only one Executor. Note also that a static schedule does not map its tasks to processors; this mapping is done automatically by the AWS Lambda platform when an Executor function instance is invoked with the static schedule and placed on a VM by AWS Lambda.

3.3 Task Execution & Dynamic Scheduling

Task Execution. WUKONG workflow execution starts when the static scheduler's Initial-Executor Invokers assign each static schedule produced by the Static-Schedule Generator to a separate Executor. Recall that each static schedule begins with one of the leaf node tasks in the DAG. The Initial-Executor invokes these "leaf node" Executors in parallel. After executing its leaf node task, each Executor then executes

the tasks along a single path through its schedule. An Executor may execute a sequence of tasks before it reaches a fan-out operation with more than one out edge or it reaches a fan-in operation. *For such a sequence of tasks, there is no communication required for making the output of the earlier tasks available to later tasks for input.* All intermediate task outputs are cached in the Executor's local memory with inherently enhanced data locality. Executors also look ahead to see what data their tasks may need, which allows them to discard data in their local memory that is no longer needed.

Furthermore, Executors can increase parallelism by scheduling and invoking new Executors to execute the task targets of a fan-out. A group of Executors that reach a common fan-in node decrease parallelism by scheduling one of them to execute the fan-in task and the rest to stop. Wukong uses dynamic scheduling to resolve conflict on the fly. More importantly, *this dynamic scheduling of the tasks in an Executor's static schedule leads to a decentralized scheduling model that naturally fits serverless computing, eliminates the need for a centralized scheduler processes that check data dependencies and invoke ready tasks with improved scalability.*

Dynamic Scheduling for Fan-Out Operations. For fan-out operations there are two cases:

Case 1: none of the n (where $n > 1$) fan-out edges is a fan-in edge. Then E "becomes" the Executor for one of the fan-out's tasks, say T, by executing T, and E "invokes" an Executor for the other fan-out tasks.

Case 2: one or more of the fan-out edges is also a fan-in edge. For example, fan-out node 3 in Figure 6(a) has a fan-out edge that is a fan-in edge to node 4. The selection of a "becomes" edge for E is based on the immediate availability of the tasks targeted by the fan-out edges. If no task target's dependencies are satisfied then no task target is immediately available for execution and none of the fan-out edges can be selected as E's "becomes" edge (the fan-in edges have fan-in operations that will be executed next); otherwise, one of the fan-out edges for the available target tasks is selected as the "becomes" edge.

An intermediate objects needed for input by an invoked Executor is passed to the Executor as an argument if the size of the object is less than the maximum allowed argument size (256K); otherwise, the object is sent to the Storage Manager, and the associated KVS keys are passed to the invoked Executors as arguments.

Each of the $n-1$ Executors invoked by E is assigned a static schedule that begins with one of the $n-1$ fan-out edges. Each of these (possibly overlapping) static schedules corresponds to a sub-graph of E's static schedule. Executor E continues task execution and scheduling along the remaining fan-out edge and executes the operation encountered on this edge.

In Figure 6(b), fan-out edges are labeled either "invokes" or "becomes" to indicate whether the Executor invokes a new Lambda Executor to execute a fan-out task or executes the fan-out task itself.

Since Executor invocations, which are in the form of AWS Lambda function invocations, incur a high overhead (e.g., invoking an AWS Lambda function takes about 50 milliseconds with the Boto3 AWS Python API), we use a number of dedicated Executor-Invoker processes that are co-located with the Static Scheduler (Figure 5). When an Executor performs a fan-out operation, and a large number of new Executors must be invoked, the Executor delegates the invocations to the Static Scheduler. The Static Scheduler evenly distributes task invocation responsibilities among the Invoker processes, enabling (near-)linear speedup over sequential invocations.

Dynamic Scheduling for Fan-in Operations. If Task Executor E executes a fan-in operation with n (where $n > 1$) in-edges, then E and the $n-1$ other Executors involved in this fan-in operation cooperate to see which one of them will continue their static schedules on the out edge of the fan-in (e.g., node 4 in Figure 6).

For a fan-in operation executed by E for fan-in task T, E atomically gets and updates a value in the KVS that tracks the number of T's input dependencies that have been satisfied during execution There are two cases:

Case 1: all of the input dependencies of T have been satisfied. Then E continues its static schedule by executing T

Case 2: all of the input dependencies of T have not been satisfied. Then E sends the intermediate object needed by T to the Storage Manager.

In Figure 6(b), each fan-in task is labeled with the Executor that executed the task.

Task Clustering. Storing and retrieving large intermediate objects can be very costly. Wukong implements task clustering to avoid large object storage.

Task Clustering for Fan-Out Operations. If the output object of some fan-out task T executed by Executor E is larger than a user-defined threshold t (e.g., 200 MB), E will try to cluster the target tasks of T's fan-out edges, i.e., E will execute the target tasks whose dependencies are satisfied instead of invoking new Executors for these tasks. For example, the Executor that executes task C can also execute tasks F and G to avoid the time and cost of communicating task C's large object output to tasks F and G. In cases like this, the fan-out edges will have multiple edges labeled "becomes".

Task Clustering for Fan-In Operations. If E executes a fan-in operation for task T and the input dependencies of a single task T have not been satisfied, then E sends the intermediate object needed by T to the Storage Manager. If this object is large, E then rechecks T's input dependencies. If T's input dependencies became satisfied while the large intermediate object was being stored, E becomes the Executor for T, This avoids the communication delay that would have occurred when T retrieved the large object from storage, but

not the delay that occurs when R stored the object. The delay that occurred when the large object was stored by E can be avoided if the storage operation can be delayed until after T's input dependencies become satisfied.

Suppose that when Executor E executes the fan-out operation it identifies multiple fan-out tasks that are also fan-in tasks and that have input dependencies that are not satisfied. E will then execute any ready fan-out tasks it finds and delay the decision about how to handle the fan-out tasks that are not ready. (There may be many fan-out tasks that are not also fan-in tasks or that are fan-in tasks with satisfied input dependencies.)

Delayed I/O. After Executor E executes the ready tasks, E will recheck whether the input dependencies of the unready tasks have since become satisfied. If so, the newly ready tasks can be executed. If at least one unready task becomes ready, the ready task can be executed and the unready tasks can be rechecked again, and so on, for a configurable number of times. Our profiling indicates that it is almost always better to wait until all of the unready tasks become ready to avoid the very large communication delay the occurs when many large objects are stored and retrieved potentially at the same time. A pattern of simultaneous large object writes and reads occurred often in the DAG workloads that we executed. If unready tasks remain when this process stops, E must send the intermediate objects needed by the unready tasks to storage; however, as described above, E can possibly avoid the communication delay associated with retrieving the objects from storage by checking the unready objects one more time.

3.4 Storage Management

WUKONG's Storage Cluster is built atop AWS Fargate [5], a serverless container engine that can be elastically scaled out-/in. The Storage Cluster includes a number of AWS Fargate tasks, each of which hosts a Redis instance, and a KV Store Proxy Service. The KV Store Proxy is executed within an EC2 VM along with an additional instance of Redis used exclusively for storing static schedules and dependency counters. The user can configure the size of the Fargate cluster to dynamically accommodate workloads of different sizes. The user simply specifies how many nodes they would like, and WUKONG ensures that these nodes are created and available. The Fargate nodes are used for the storage of intermediate data generated during a workload's execution. We opt to use Redis instead of S3 as Redis can provide both high bandwidth for large object workloads and high IOPS for small object workloads [44, 50], whereas S3's IOPS is throttled. However, modifying WUKONG to use S3 is trivial.

The KV Store Proxy performs various storage operations on behalf of the Task Executors and the Scheduler. At the start of a workflow, the Storage Manager receives the workflow DAG and the static schedules derived from the DAG from the Scheduler.

```
1  def add(x,y):
2    time.sleep(0.5)
3    return x + y
4
5  L = range(8)
6  while len(L) > 1:
7    L = list(map(delayed(add),\
8             L[0::2], L[1::2]))
9
10 L[0].compute()
```

Figure 7: TR code.

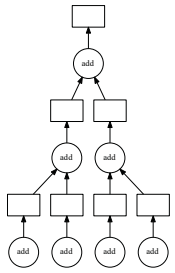

Figure 8: TR DAG.

Intermediate and Final Result Storage. Task Executors publish their intermediate and final task output objects to the KV Store. Final outputs are relayed to a Subscriber process in the Scheduler for presentation to the Client.

Small Fan-out Task Invocations. When a Task Executor performs a fan-out operation that has a small number of out edges, the Task Executor makes the necessary Executor invocations itself.

Large Fan-out Task Invocations. When a fan-out has a number of out edges that is larger than a user-specified threshold, the Task Executor publishes a message that is relayed to a Subscriber process in the Storage Manager that then passes the message on to the Proxy. This message contains an ID that identifies the fan-out's location in the DAG. The Proxy uses the DAG and the fan-out ID to identify the fan-out's out edges in the DAG. This allows the Proxy, with the assistance of the Fan-out Invokers in the Storage Manager, to make the necessary Task Executor invocations, in parallel. The Proxy passes to each Executor its intermediate inputs (or their key values in the KV Store) and the Executor's static schedule.

3.5 Programming Model

WUKONG reuses Dask's Python programming interfaces [12] (including high-level APIs, such as Dask libraries and data structures, and low-level APIs, such as `Dask.delayed`) for implementing parallel programs. In general, any code written for Dask should execute on WUKONG. Figure 7 shows an example code snippet that implements tree reduction (TR) (a detailed description of TR is in §4.1). WUKONG also reuses Dask's DAG generator which translates high-level Python code into a DAG [11]. Figure 8 depicts the DAG generated for TR with an 8-element array.

3.6 Fault Tolerance

For fault tolerance, we relied on the automatic retry mechanism of AWS Lambda, which allows for up to two automatic retries of failed function executions. Investigating better fault tolerance scheme is part of our future work.

4 EVALUATION

Implementation. We have implemented Wukong using roughly 12*k* lines of Python code (5,349 LoC for the AWS Lambda Executor Runtime, 3,057 LoC for the Storage Manager, and 3,577 LoC for the Static Scheduler). We use the Dask library [9] to generate DAGs to use as the input computation graph for Wukong.

4.1 Experimental Goals and Methodology

Our evaluation was performed on AWS. The static scheduler ran in an `r5n.16xlarge` EC2 VM. We used a scale-out Redis cluster (Multi-Redis) as Wukong's intermediate storage. The Fargate nodes within the storage cluster were each configured to have 30 GB of memory and 4 vCPUs. For the Multi-Redis setup, Wukong used 75 nodes for the storage cluster, as we've determined this value to be both performant and cost-effective for many workloads. The MDS proxy was co-located on the same `r5n.16xlarge` VM as a Redis instance that was used for storing static schedules and dependency counters. Each Lambda function was allocated 3 GB of memory and a maximum execution time of seven minutes.

We compared the performance of Wukong against both Dask distributed (which we refer to simply as "Dask") and numpywren [52] / PyWren [42]. We chose to compare our performance against Dask as both Wukong and Dask use the exact same input DAG and the same algorithms for their computations. We compared the end-to-end performance of Wukong against numpywren, and we compared the scalability of Wukong against PyWren, which is numpywren's underlying Lambda execution framework.

We chose to compare Wukong against numpywren because both are serverless DAG execution frameworks; however, there are significant differences between Wukong and numpywren. One difference is that Wukong uses Dask DAGs, which explicitly encode tasks and their dependencies. Numpywren, on the other hand, uses an implicit DAG representation for its programs, which are all implemented in the LAmbdaPACK language for linear algebra [52]. The nodes of the DAG that represent a LAmbdaPACK program are generated on-demand (at runtime).

Another difference is that numpywren uses AWS S3 as its intermediate data store. In order to make the comparison between Wukong and numpywren more fair, we modified numpywren to use a single instance of Redis as its intermediate object store. We compared this version of numpywren, which we refer to as "Numpywren Single Redis", with a modified version of Wukong that also uses a single instance of Redis for intermediate data storage. In addition, we compared numpywren S3 against "Wukong Multi-Redis", which uses a Fargate Redis cluster for its intermediate object store. This second comparison was performed in order to show the optimal performance achieved by each system.

Finally, numpywren only supports linear algebra algorithms and only several of these algorithms are also available in Dask. As a result, we are limited in which workloads we can run on both Wukong and numpywren.

To better understand Wukong's performance, we compared Wukong against 2 different Dask configurations. Both configurations used the same amount of CPU (2,000 cores) and memory (3,000 GB memory). This was the largest VM cluster that we could configure. The first configuration consisted of 1,000 2-core 3GB workers running on 125 `c5.4xlarge` 16-core 32 GB VMs. Each VM had eight workers running on it. The second configuration consisted of 125 workers; each worker exclusively ran on a `c5.4xlarge` VM and was allocated 16 cores and 24 GB memory. The first configuration was selected so that each worker was approximately as powerful as the AWS Lambda functions used by Wukong; it represents a *worst-case scenario* that stresses the Dask scheduler with many workers and incurs high communications due to the lack of data locality, emulating a serverless environment with centralized scheduling. The second configuration represents a *best-case scenario* where the relatively more powerful workers could process larger data with improved data locality and significantly reduced communications.

Additionally, we used the exact same input data partitions for Dask and Wukong. We scaled the problem size for each benchmark by having each (Dask, Wukong, or numpywren) worker assigned with (at least) one partition of the input data. As such, a test used only a fraction of the 2,000 cores of the Dask cluster, until the problem size was large enough to occupy all the resources. The largest number of partitions (of input data) used during our evaluation was 4,096 for 16 *M* TSQR (which will be described shortly). This number of partitions does not overwhelm either Dask configurations as TSQR's DAG features large fan-ins in the middle of the job. The large fan-ins result in the number of tasks allocated to each worker decreasing as the workload progresses.

In our evaluation results, each data point is the average of *ten* runs. The error bars on the graphs of the results indicate the biggest and smallest results obtained. Wukong is easy-to-use as it exposes only two configuration knobs to the end users—the input partition size and the number of Fargate nodes. (A sensitivity analysis of these two configuration knobs is omitted due to space constraint.)

We evaluated the following parallel applications.

Tree Reduction (TR): TR sums the N elements of an array using a total of $N - 1$ operations performed over multiple passes. Each pass adds adjacent elements, in parallel, until after $log(N)$ passes only a single element remains. TR serves as a microbenchmark for evaluating the effect of task granularity and parallelism on performance. See Figure 7 and Figure 8 for an example of the Python code snippet and generated DAG.

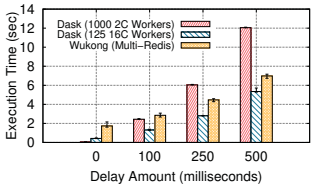

Figure 9: TR.

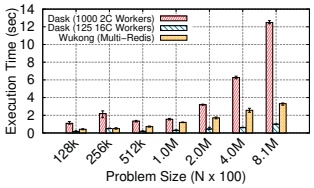

Figure 10: SVD1.

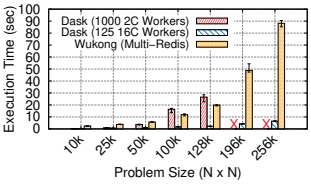

Figure 11: SVD2.

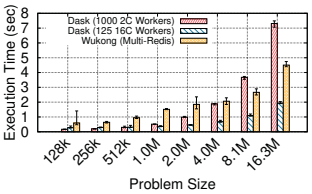

Figure 12: SVC.

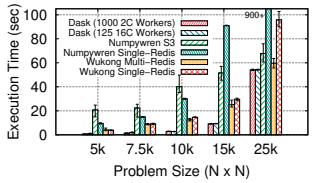

Figure 13: GEMM.

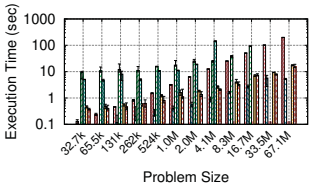

Figure 14: TSQR (log-scale).

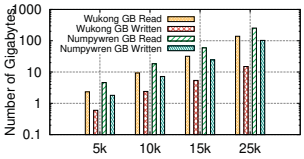

Figure 15: GEMM I/O (log).

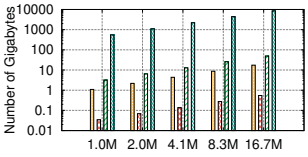

Figure 16: TSQR I/O (log).

Singular Value Decomposition (SVD): We evaluated two variants of SVD. The first variant (SVD1) computes the SVD of a tall skinny matrix. The second variant (SVD2) computes the SVD of a square (i.e., $n \times n$) matrix using an approximation algorithm provided by [40]. Note that Dask's SVD algorithm differs considerably from numpywren's, and thus a direct comparison between WUKONG and numpywren for SVD is impossible. For reference, WUKONG can execute SVD $256k \times 256k$ in 88 seconds on average while [52] reports an average of 77,828 seconds for the same problem size for SVD.

Support Vector Classification (SVC): SVC is a machine learning workload. The benchmark we used is publicly available in the Dask-ML documentation [10].

General Matrix Multiplication (GEMM): GEMM performs matrix multiplication, an important component of many linear algebra algorithms.

Tall-and-Skinny QR Factorization (TSQR): This workload performs a QR factorization of a tall skinny matrix.

4.2 End-to-End Performance Comparison

TR. The size of the array used for TR was 1024. We also intentionally added a delay to each task of TR. This delay simulates an increased amount of work performed per task. We varied the amount of delay between 0–500 ms. Figure 9 shows that both configurations of Dask outperforms WUKONG by a large margin for the base case. This is because Dask uses TCP to dispatch the $1024/2 = 512$ `addition` tasks to workers, whereas for WUKONG the overhead of scaling out to 512 Lambda executors dominates. As we add increasing amounts of delay to each task, the performance gap between Dask and WUKONG decrease. Once the delay is 250 ms or more, WUKONG executes the workload faster than the 1,000-worker Dask cluster, as WUKONG uses decentralized scheduling to rapidly scale out to 512 workers. This experiment shows an interesting tradeoff between per-task execution time and serverless scaling cost – WUKONG performs best when each task performs a non-trivial amount of work, as this compensates for the overhead of spinning up additional Lambdas.

SVD. We tested seven problem sizes for both SVD1 and SVD2 (see Figure 10 and Figure 11). For nearly all problem sizes, WUKONG out-performed the 1,000-worker Dask cluster, completing the job anywhere from 62.02% to 69.09% faster than Dask. This performance difference results from the benefits of WUKONG's decentralized scheduling techniques, which greatly reduce the overhead of executing tasks on a large number of workers. The 1,000-worker Dask was bottlenecked by its central scheduler due to the increasing load from the one thousand workers.

The 125-worker Dask cluster consistently outperformed WUKONG. This is because each Dask worker is provisioned with more resources (i.e., more CPUs, greater network bandwidth, a larger amount of RAM per worker, etc.), which significantly increases data locality and reduces cross-worker communications. More importantly, for SVD2, WUKONG was able to scale to considerably larger problem sizes than what the 1,000-worker Dask cluster was capable of handling (crosses in Figure 11). The results demonstrate WUKONG's ability to provide competitive performance with traditional serverful frameworks, while also scaling effectively for increasingly large problem sizes. WUKONG's ability to scale effectively here is largely because of its use of task clustering and delayed I/O. These techniques dramatically reduce data movement and ensure all downstream tasks which depend on large data are executed on the Task Executor that already has the data in-memory. A quantitative analysis of the benefits of these techniques is given later in §4.5.

SVC. Figure 12 shows SVC's performance. As with the previous benchmark, Dask was able to perform better for smaller problem sizes; however, when we increased the problem size, the performance gap between the two frameworks decreased. As the scale of the problems increased to $4.0M$ samples and beyond, WUKONG began to scale more effectively than the 1,000-worker Dask cluster. The large parallelism of WUKONG

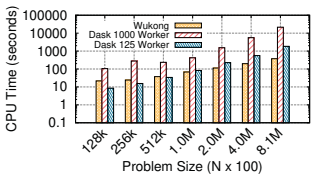

Figure 17: SVD1 CPU time.

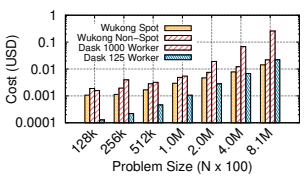

Figure 18: SVD1 cost.

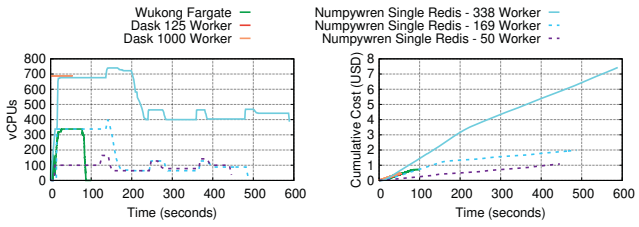

Figure 19: GEMM CPU usage and cost timeline.

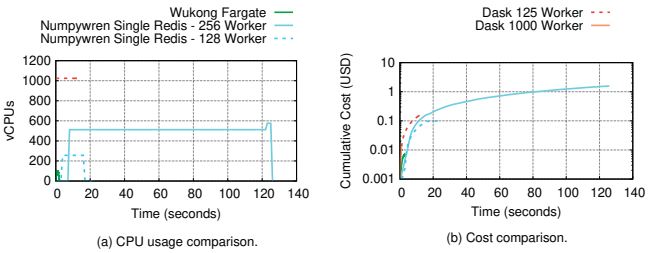

Figure 20: TSQR CPU usage and cost timeline.

enabled the framework to complete the workload in 46.45% less time; however, the 125-worker Dask cluster executed the workload roughly 50.56% faster than Wukong. This is likely due to both the higher network bandwidth and increased data locality of the Dask workers.

GEMM. Like TR, the results of our GEMM experiments identify a workload that is difficult to execute in a serverless environment. As shown in Figure 13, Wukong achieved worse performance than Dask. This is because GEMM natively requires a number of large data objects to be communicated between tasks before the computation can begin. Due to the limited bandwidth available to the intermediate data storage, both Wukong and numpywren experience long delays during this phase of the workload.

Wukong's performance greatly exceeded that of both numpywren configurations for all problem sizes. For the largest problem size, Wukong (single Redis shard) executed the workload 89.76% faster than numpywren (single Redis shard). Wukong (with Fargate) was 51.51% faster than numpywren (with S3) for $15k \times 15k$ matrices because Wukong reduced the amount of data read from and written to the intermediate KVS, and this reduction on I/Os directly translates to performance improvement. As shown in Figure 15, Wukong read 49.39% less data than numpywren for the smallest problem size and 45.24% for the largest; Wukong reduced the amount of data written by as much as 85% for the largest problem size.

TSQR. Figure 14 shows that Wukong outperformed both numpywren (with S3) and numpywren (single Redis shard) for all problem sizes. For the $4.1M \times 128$ matrix, Wukong (single Redis shard) was executing the workload 68.17× faster, or in 98.53% less time, than numpywren (single Redis shard); and Wukong (Fargate) is 9.19× faster, or in 89.11% less time, than numpywren (S3). Numpywren (single Redis shard) failed to execute the next larger problem size, and the largest workload numpywren (S3) was able to execute was $16.7M \times 128$. For this workload, Wukong was 13.36× faster, executing the workload in 92.51% less time. This again, is because of the significantly reduced reads and writes. Wukong's "become" functionality allowed serial tasks along a single subgraph path to execute locally on the same Lambda Executor; whereas numpywren randomly assigned high priority tasks (which were ready to execute) to any stateless Lambda executor. As a result, numpywren wrote 16,027× more data for the smallest problem size and 15,631× more data for the largest problem size, which resulted in dramatic performance degradation (Figure 16).

4.3 CPU Time and Monetary Cost

Figure 17 presents a comparison between Wukong (Multi-Redis) and both Dask clusters on their total CPU time (core seconds) for SVD1. Note that this comparison only considers cores actively used by Dask for each problem size. Wukong uses more core seconds than Dask-125 for the first three problem sizes, as Dask-125 finishes the job significantly faster than Wukong. For $1.0M$ and above, Wukong uses less core seconds than Dask-125. Dask-1000 incurs the longest CPU time as it is the slowest of the three. The disparity between the two frameworks increases as the problem size grows.

Figure 18 presents a comparison of the monetary cost to execute SVD1 for varying problem sizes. This cost analysis only considers the Dask VMs actively used for each given workload size. At first, Wukong is more expensive than Dask-125. As the problem size increases, the cost of executing the workload on Wukong increases at a much slower rate than the cost of running the workload on Dask-125. By the largest problem size, the price of executing the workload on Wukong is equal to that of executing the workload on Dask-125. Additionally, Wukong achieves faster performance, lower cost, and more efficient CPU usage than Dask-1000 using non-spot pricing for all except the smallest problem size. These results demonstrate Wukong's pay-per-use property.

Figure 19 shows a comparison between various configurations of Wukong Single Redis, Dask, and numpywren Single Redis on the number of vCPUs and cumulative cost used during GEMM $25k \times 25k$. These configurations include both

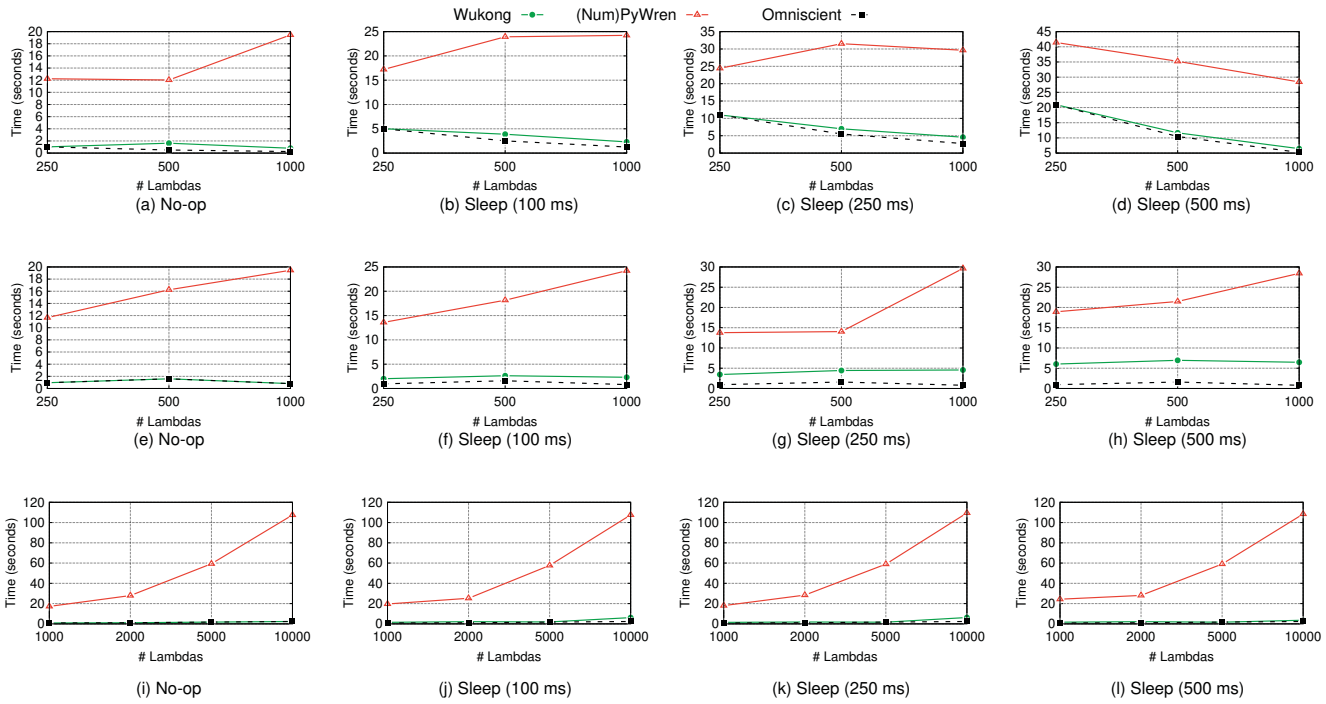

Figure 21: Figure 21(a)-(d) – strong scaling: time (Y-axis) to execute 10,000 tasks over N Lambda executors (X-axis). Figure 21(e)-(h) – weak scaling: time to execute 10 tasks per Lambda. Figure 21(i)-(l) – serverless scaling: time to execute N task on N Lambda. For each row, plots are for (from left to right) tasks of 0, 100, 250, and 500 ms.

Dask configurations and three configurations of numpywren Single Redis. By design, numpywren allows users to specify the initial[4] number of Lambda executors (workers) for a job. We configured numpywren to use 50, 169, and 338 workers. Numpywren-169 was selected because the maximum concurrency achieved by WUKONG during this workload was 169 Lambdas. We tested a scaled-out version of numpywren that used twice as many starting workers (338) as well as a scaled-down version using 50 workers. Notably, numpywren-50 was the fastest configuration, followed by numpywren-169 and finally numpywren-338. This suggests that increasing the number of numpywren's parallelism significantly increases contention, possibly at the framework's central queue or scheduler, leading to performance degradation.

WUKONG was both cheaper and used less vCPUs during the execution of the workload than all three numpywren configurations. Specifically, WUKONG was 33.47% cheaper and 77.57% faster than the best-performing numpywren configuration. More importantly, WUKONG is autonomous and does not require users to explicitly tune the parallelism, which improves the usability.

[4]One can configure the starting number of executors, a maximum number of executors, and the policy used to scale the number of executors dynamically during execution. We opted to select the default scaling policy for all numpywren runs.

Similarly, Figure 20 shows a comparison for TSQR 4.0M. The first configuration of numpywren used 128 workers while the second used 256 workers. These are based on the number of leaf tasks in WUKONG's workload (512), though we found that using 512 numpywren workers resulted in worse performance. WUKONG used less CPU resources and was significantly cheaper than all other frameworks. It also out-performed all other frameworks except for Dask-125. Specifically, WUKONG completed the workload in 87.41% less time and 92.96% more cheaply than the best-performing numpywren configuration (i.e., 14.22× cheaper and 7.94× faster). Lastly, while WUKONG did not out-perform Dask-125, WUKONG was 95.67% cheaper than Dask-1000 and 45.70% cheaper than Dask-125 for this workload. WUKONG did not reach more than 106 vCPUs during the workload's execution as many of the leaf tasks performed only a small amount of work before writing their data to the KVS and exiting. That is, by the time additional leaf tasks are scheduled, previously-invoked leaf tasks were already finishing their execution, forming a scheduling pipeline.

4.4 Elastic Scaling

We next evaluate WUKONG's scalability on traditional strong / weaking scaling and serverless scaling metrics, and compare it against (Num)-PyWren. The maximum concurrency we got from Amazon was 5,000 concurrent Lambdas. In this experiment, for both (Num)-PyWren and WUKONG, all the

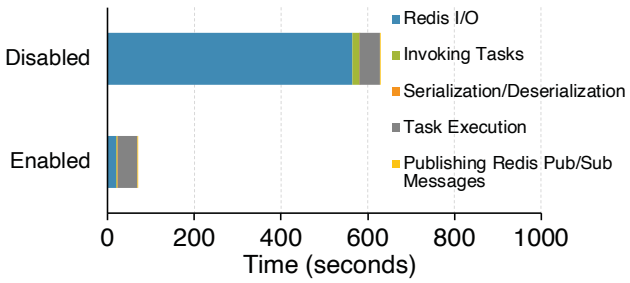

Figure 22: SVD2 50k x 50k aggregated execution time breakdown with and without task clustering and delay I/O.

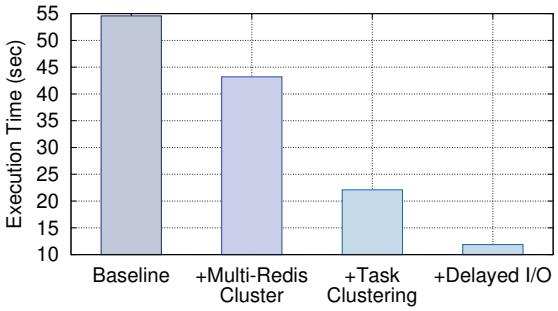

Figure 23: Contributions of optimizations to Wukong's performance of SVD2.

executors in the Lambda pool got warmed up before accepting task requests, eliminating the cold start concerns. The Lambda pool scaled from zero.

Strong Scaling. For strong scaling, we varied the number of Lambdas to be launched to execute 10,000 tasks. In order to simulate workloads with various computation loads, we added a delay of 100 ms, 250 ms, and finally 500 ms to each task. Each test was repeated three times.

Figure 21(a)–(d) show that, in all cases, Wukong exhibited near-ideal strong scaling behavior, scaling significantly better than (Num)-PyWren in all cases, thereby demonstrating the effectiveness of our decentralized scheduling mechanism. It was not until the 500 ms delay case that (Num)-PyWren exhibited a downward trend in execution time as the number of Lambda executors scaled. This is because: 1) 500-ms tasks tend to run longer, and 2) with more Lambda executors and a fixed total amount of tasks, each Lambda gets assigned less number of tasks.

Weak Scaling. For weak scaling, we executed ten tasks per worker and varied the number of Lambda executors from 250 to 1,000. As shown in Figure 21(e)–(h), Wukong exhibited near-ideal weak scaling behavior for all sleep delays and worker configurations. Notably, thanks to the decentralized scheduling, Wukong was able to scale to 1,000 Lambda executors much more effectively than (Num)PyWren, which experienced increasingly large delays as the number of Lambdas increased.

Serverless Scaling. Finally, we test serverless scaling – executing N tasks on N Lambda executors, with each Lambda effectively executing one single task. Figure 21(i)–(l) plots the results. We observe that (Num)-PyWren took an increasing amount of time to finish executing N tasks on N Lambdas, and executing 10,000 tasks took almost two minutes. In contrast, Wukong rapidly scales out to 10,000 tasks in few seconds, almost approaching the behavior of an omniscient serverless execution framework. This again demonstrates the efficacy of Wukong's decentralized scheduling.

4.5 Factor Analysis

Task Clustering and Delay I/O. Finally, we look at the impact of task clustering and delaying I/O on Wukong's performance. Clustering and delay I/O prevent tasks with large intermediate data from writing their data to Redis. Instead, these tasks attempt to execute downstream tasks locally. Any downstream tasks that cannot immediately be executed because their dependencies are not satisfied are put into a queue. The dependencies of the queued tasks are routinely rechecked until the dependencies are satisfied or a maximum delay time is reached. By delaying I/O, more tasks can be clustered and expensive network I/Os data can be avoided, decreasing the workload runtime dramatically.

Figure 22 displays two aggregations of time for the activities performed for SVD2. In both cases, "publishing messages", "task execution", and "serialization/deserialization" each took roughly the same amount of time (in aggregate); however, the difference between the times for task invocation and especially Redis I/O is significant. With the two optimizations disabled, task invocations and Redis I/O made up an aggregate 14.80 and 565.21 seconds, respectively. When the optimizations were enabled, task invocation took an aggregate 2.05 seconds while Redis I/O took just 20.36 seconds. There is 7.21× more aggregate time spent invoking tasks and 27.76× more aggregate network I/O performed with clustering disabled.

We analyze Wukong's performance by breaking down the performance gap between a baseline and Wukong with all optimizations enabled (Figure 23). The use of the Fargate multi-Redis storage cluster results in a 20.85% performance improvement over using AWS ElastiCache for intermediate data storage. When using Fargate, the I/O performed during the workload is spread across a large number of Redis instances, resulting in reduced network contention and consequently reduced I/O latency. (Using a large number of ElastiCache instances is cost prohibitive.) When clustering (without delayed I/O) is enabled, performance improves by another 48.82% as a significant amount of large object I/Os is eliminated. Finally, enabling delayed I/O results in

a 46.21% improvement relative to the use of clustering and Fargate alone. Overall, WUKONG is 4.6× faster when all optimizations are used, demonstrating the effectiveness of these techniques.

5 RELATED WORK

Researchers have identified new stateful parallel applications for serverless computing. These efforts have lead to serverless parallel computing frameworks [6, 16, 25, 28, 30, 36, 37, 41, 42, 50, 52], which have been built using methods 2 and 3 describes in §1. However, none of them explicitly addresses the data locality issue of stateful serverless parallel applications. WUKONG goes beyond existing work with a novel *locality-aware* decentralized scheduling approach for complex DAG jobs.

funcX [32] is an open FaaS platform that enables high performance "serverless supercomputing" over existing HPC infrastructures. WUKONG is orthagonal to funcX and should be portable to existing commercial and open-source FaaS platforms [4, 8, 15, 19, 35]. Porting WUKONG to other serverless platforms is part of our future work.

SAND [24] is a serverless platform that increases data locality by running some or all of the functions/tasks for a given workload within the same container on the same server, but as separate processes. WUKONG is a serverless application framework that increases data locality by executing multiple dependent tasks in the same WUKONG Executor function, and hence in the same process, container, and server. Thus, SAND improves data locality in the serverless platform layer, while WUKONG improves locality in the application framework layer running atop the serverless platform.

INFINICACHE [53] is a distributed memory cache that exploits the memory of serverless functions for object caching. WUKONG complements INFINICACHE in that WUKONG can use INFINICACHE to cache intermediate data.

General-purpose serverless orchestration frameworks include AWS Step Functions [6], Azure Durable Functions [7], Fission Workflows [14], and HyperFlow [16, 47]. But these frameworks are not well-suited for supporting large, complex jobs, because they require manual workflow configurations (e.g., JSON) [46].

A large body of research has explored distributed scheduling [27, 29, 48, 49, 51]. However, these solutions all target serverful scheduling with serverful deployment specific optimization objectives. WUKONG targets serverless computing and takes advantage of the massive parallelism of serverless computing for high efficiency.

6 DISCUSSION AND LESSONS

WUKONG is not intended to replace established, serverful processing frameworks such as Spark [54] and TensorFlow [23]. Rather, because WUKONG is less powerful but easier to use than these serverful frameworks, WUKONG targets users who lack a strong CS background, but who work on lighter-weight and smaller computing-related problems, in areas such as data analytics and machine learning, particularly those whose solutions can be implemented using numerical Python libraries.

Our initial attempt to port Dask to a serverless platform was unsuccessful due to the poor performance of the resulting framework [31]. There were several major bottlenecks in this original design. For one, the use of a centralized scheduler to assign tasks to Lambda executors was too slow for all but the smallest workloads. Specifically, the centralized scheduler was unable to process thousands of concurrent network connections with Lambda executors. Additionally, the centralized scheduler struggled to rapidly invoke Lambda functions when scaling out for large workloads. Finally, Lambda executors spent an overwhelming majority of their time reading and writing intermediate data. [31] presents a preliminary study about these performance bottlenecks.

To address these performance issues, we developed our decentralized scheduling technique. While this technique significantly improved performance, there were still bottlenecks due to a lack of data locality. Specifically, reading and writing very large intermediate objects dominated end-to-end execution time, particularly for workloads such as SVD. To address these bottlenecks, we developed task clustering and delayed I/O. Though these techniques have been used in serverful contexts [29, 45], they had never been utilized in a serverless context. With the addition of these two techniques, large object reads and writes were eliminated, which greatly improved performance and resource utilization.

7 CONCLUSION

We have presented WUKONG, a new serverless parallel computing framework that uses locality-enhanced, decentralized scheduling (atop AWS Lambda), task clustering, and delayed I/O to achieve high performance, near-ideal scalability, and data locality while being cost-effective. Our evaluation demonstrates the effectiveness of decentralized scheduling, task clustering, and delayed I/O in reducing both the execution time and cost of executing workloads. Further, we have shown that WUKONG exhibits near-ideal scaling behavior, reduces the execution time by as much as 98.53% compared to numpywren, and reduces network I/O by multiple orders of magnitude. Finally, WUKONG can reduce costs by upwards of 92.96% and 95.67% compared to numpywren and Dask, respectively.

WUKONG's source code is available at:

https://mason-leap-lab.github.io/Wukong.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable comments and suggestions that improved the paper. This work is sponsored in part by NSF grants CCF-1919075, CCF-1919113, OAC-2007976, George Mason University, an AWS Cloud Research Grant, and Google Cloud Platform Research Credits.

REFERENCES

[1] [n.d.]. 2018 Serverless Community Survey: huge growth in serverless usage. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/.

[2] [n.d.]. Alibaba Cluster Trace Program (New 2018 Version. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.

[3] [n.d.]. Apache Hadoop. https://hadoop.apache.org/.

[4] [n.d.]. Apache OpenWhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[5] [n.d.]. AWS Fargate: Serverless compute for containers. https://aws.amazon.com/fargate/.

[6] [n.d.]. AWS Step Functions. https://aws.amazon.com/step-functions/.

[7] [n.d.]. Azure Durable Functions Overview. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

[8] [n.d.]. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[9] [n.d.]. Dask: Scalable Analytics in Python. https://dask.org/.

[10] [n.d.]. Dask: Scalable Machine Learning in Python. https://dask-ml.readthedocs.io/en/latest/#.

[11] [n.d.]. Dask Task Graphs. https://docs.dask.org/en/latest/graphs.html.

[12] [n.d.]. Dask User Interfaces. https://docs.dask.org/en/latest/user-interfaces.html.

[13] [n.d.]. Developing Convex Optimization Algorithms in Dask parallel: math is fun. https://matthewrocklin.com/blog/work/2017/03/22/dask-glm-1.

[14] [n.d.]. Fission Workflows. https://github.com/fission/fission-workflows.

[15] [n.d.]. Google Cloud Functions. https://cloud.google.com/functions/.

[16] [n.d.]. HyperFlow: a scientific workflow execution engine. https://github.com/hyperflow-wms/hyperflow.

[17] [n.d.]. kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.

[18] [n.d.]. MPI Forum. https://www.mpi-forum.org/.

[19] [n.d.]. OpenFaaS: Serverless Functions, Made Simple. https://www.openfaas.com/.

[20] [n.d.]. Scikit-learn Support Vector Machines. https://scikit-learn.org/stable/modules/svm.html#svm-classification.

[21] [n.d.]. Serverless: Build and run applications without thinking about servers. https://aws.amazon.com/serverless/.

[22] [n.d.]. Serverless: Cold Start War. https://mikhail.io/2018/08/serverless-cold-start-war/.

[23] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[24] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. {SAND}: Towards High-Performance Serverless Computing. (2018), 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[25] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. [n.d.]. Sprocket: A Serverless Video Processing Framework. In *ACM SoCC '18*.

[26] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[27] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. [n.d.]. Parsl: Pervasive Parallel Programming in Python. In *ACM HPDC '19*.

[28] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. [n.d.]. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *ACM Middleware '19*.

[29] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. 2012. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. *Parallel Comput.* 38, 1-2 (Jan. 2012), 37–51. https://doi.org/10.1016/j.parco.2011.10.003

[30] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/3357223.3362711

[31] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. 2019. In Search of a Fast and Efficient Serverless DAG Engine. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. 1–10.

[32] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. FuncX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/3369583.3392683

[33] Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. *Mach. Learn.* 20, 3 (Sept. 1995), 273–297. https://doi.org/10.1023/A:1022627411411

[34] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[35] Fission. [n.d.]. Serverless Functions for Kubernetes. https://fission.io/.

[36] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. [n.d.]. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *USENIX ATC 19*.

[37] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. [n.d.]. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *USENIX NSDI 17*.

[38] Jim Gray. 1985. Why Do Computers Stop And What Can Be Done About It?

[39] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. [n.d.]. Who Limits the Resource Efficiency of

My Datacenter: An Analysis of Alibaba Datacenter Traces. In *ACm IWQoS '19*.

[40] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. [n.d.]. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. ([n. d.]). arXiv:0909.4061 http://arxiv.org/abs/0909.4061

[41] V. Ishakian, V. Muthusamy, and A. Slominski. 2018. Serving Deep Learning Models in a Serverless Platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 257–262. https://doi.org/10.1109/IC2E.2018.00052

[42] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. [n.d.]. Occupy the Cloud: Distributed Computing for the 99%. In *ACM SoCC '17*.

[43] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report.

[44] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[45] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.* (1999).

[46] Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. [n.d.]. Comparison of FaaS Orchestration Systems. ([n. d.]), 148–153. https://doi.org/10.1109/UCC-Companion.2018.00049 arXiv:1807.11248

[47] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2017. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems* (Nov. 2017). https://doi.org/10.1016/j.future.2017.10.029

[48] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[49] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. [n.d.]. Sparrow: Distributed, Low Latency Scheduling. In *ACM SOSP '13*.

[50] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. [n.d.]. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *USENIX NSDI 19*.

[51] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. ACM, New York, NY, USA, 351–364. https://doi.org/10.1145/2465351.2465386

[52] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

[53] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[54] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. [n.d.]. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX NSDI 12*.

[55] Kaihua Zhu, Hao Wang, Hongjie Bai, Jian Li, Zhihuan Qiu, Hang Cui, and Edward Y. Chang. 2008. Parallelizing Support Vector Machines on Distributed Computers. In *Advances in Neural Information Processing Systems 20*, J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis (Eds.). Curran Associates, Inc., 257–264. http://papers.nips.cc/paper/3202-parallelizing-support-vector-machines-on-distributed-computers.pdf