In Search of a Fast and Efficient Serverless DAG Engine

Benjamin Carver, Jingyuan Zhang, Ao Wang, Yue Cheng
Serverless Computing

- Emerging cloud computing platform based on the composition of fine-grained user-defined functions
- Service provider is responsible for provisioning, scaling, and managing resources
- Pay-per-use pricing model with fine granularity
Background

- Data analytics applications can be modeled as a directed acyclic graph (DAG) based workflow
  - Nodes: fine-grained tasks
  - Edges: dependencies between tasks, often large fan-outs

- DAG workflows well-suited for serverless computing (or Functions-as-a-Service)
  - Auto-scaling accommodates short tasks and bursty workloads
  - Pay-per-use keeps the cost of short tasks low
From Serverful to Serverless

- **Serverful** focuses on load balancing and cluster utilization
  - Bounded resources, unlimited time
  - User explicitly allocates tasks to processors
  - Servers managed by the user

- **Serverless platforms** provide a nearly unbounded amount of ephemeral resources
  - Bounded time, unlimited resources
  - Cloud provider automatically allocates serverless functions to VMs
  - Servers managed by the service provider
AWS Lambda Constraints

- Lambda function invocation currently take 50ms on average
- Outbound-only network connectivity
- Relatively low network bandwidth
- Execution time limits (900 seconds)
- Lack of quality-of-service (QoS) control, leading to stragglers
  - e.g., cold starts
Existing Parallel Frameworks Using Serverless Computing

- **PyWren [SoCC’17]**
  - Parallelize existing Python code with AWS Lambda

- **Numpywren**
  - System for linear algebra built atop PyWren

- **ExCamera [NSDI’17]**
  - System which allows users to edit, transform, and encode videos using fine-grained serverless functions

- **gg [ATC’19]**
  - Framework and command-line tools to execute “everyday applications” within cloud functions
Typical Approaches

- **Approach 1: Queue-based Master-Worker**
  - Master submits ready tasks to a queue
  - Workers are cloud functions that process tasks in parallel, e.g., Numpywren
  - **Drawbacks**: cannot exploit data locality as easily; reading from queue could become a bottleneck

- **Approach 2: Centralized scheduler directly invokes cloud functions to process ready tasks**, e.g., ExCamera
  - **Drawback**: centralized scheduler could become a bottleneck for system
Typical Approaches

- **Approach 1: Queue-based Master-Worker**
  - Master submits ready tasks to a queue
  - Workers are cloud functions that process tasks in parallel, e.g., Numpywren
  - **Drawbacks**: cannot exploit data locality as easily; reading from queue could become a bottleneck

- **Approach 2: Centralized scheduler directly invokes cloud functions to process ready tasks**, e.g., ExCamera
  - **Drawback**: centralized scheduler could become a bottleneck for system

Wukong solves these drawbacks.
Wukong

- **Approach**

- **Architecture**
  - Static Scheduler
  - Task Executors
  - Storage Manager

- **Evaluation**
Our Approach - Wukong

- **Static Scheduling**
  - Decentralized, cooperative scheduling
    - Lambda functions coordinate with each other to execute overlapping sections of assigned sub-DAGs
  - Statically partition DAG into sub-DAGs
    - Assign each partition to a Lambda function

- **Dynamic Scheduling**
  - Task executors cooperate here!
Wukong

- Approach

- **Architecture**
  - Static Scheduler
  - Task Executors
  - Storage Manager

- Evaluation
Static Scheduler

- Partitions DAG into sub-DAG using a depth-first search (DFS) from each leaf node.
- Assigns sub-DAGs to executors
Executors

- Decentralized, cooperating schedulers
- Schedule and execute tasks in assigned sub-DAGs
- Cooperate on scheduling tasks contained in two or more sub-DAGs
Storage Manager

- Performs storage operations on behalf of Executors and Static Scheduler
- Using KV Store for intermediate data storage
Wukong

- Approach

- Architecture
  - Static Scheduler
  - Task Executors
  - Storage Manager

- Evaluation
Experimental Goals

● Identify and describe the factors influencing performance and scalability

● Compare WUKONG against Dask
  ○ Can WUKONG achieve performance comparable to Dask distributed executing on general-purpose VMs, given the inherent limitations of AWS Lambda?
Experimental Setup

● **Compare against Dask** distributed running on two different setups.
  ○ 5-node EC2 cluster of t2.2xlarge VMs
  ○ Laptop
    ■ Windows 7 64-bit
    ■ Intel Core i5-6200U CPU @ 2.30GHz
    ■ 8GB RAM

● Wukong Static Scheduler, KV Store, and KV Store Proxy running on c5.18xlarge EC2 VMs.

● Task Executor allocated 3GB memory with timeout set to two minutes.
Four DAG Applications

● Microbenchmark
  ○ **Tree Reduction**: repeatedly add adjacent elements of an array until a single value remains

● Linear Algebra
  ○ **General Matrix Multiplication (GEMM)**
    ■ 10,000 × 10,000 and 25,000 × 25,000
  ○ **Singular Value Decomposition (SVD)**
    ■ $n \times n$ matrix and a tall-and-skinny matrix, varying sizes

● Machine Learning
  ○ **Support Vector Classification (SVC)**
    ■ 100,000 - 800,000 samples
Tree Reduction

![Bar chart showing average execution time for different tree reduction methods]

- **Strawman**
- **PubSub**
- **Parallel-Invoker**
- **Wukong**
- **Dask (EC2)**
- **Dask (Laptop)**
Tree Reduction with Delays

![Bar chart showing average execution time with different sleep amounts and delays for Strawman, PubSub, Wukong, and Parallel-Invoker. The x-axis represents sleep amounts (0ms, 100ms, 250ms, 500ms), and the y-axis represents average execution time (seconds). The chart shows the performance comparison under varying delays.]
General Matrix Multiplication (GEMM) and Support Vector Classification (SVC)
Singular Value Decomposition (SVD) - “Tall and Skinny”

\[ X = \text{da.random.random}((200000, 100), \text{chunks}=(10000, 100)) \]

\[ u, s, v = \text{da.linalg.svd}(X) \]

\[ v\text{.compute()} \quad \# \text{Begin execution} \]
Singular Value Decomposition - “$n \times n$”

**SVD-Compressed (rank 5) $n \times n$**

$$X = da.random.random((10000, 10000), \text{chunks}=(2000, 2000))$$
$$u, s, v = da.linalg.svd_compressed(X, k=5)$$
$$v.compute() \# \text{Begin execution}$$

![Graph showing execution times for different problem sizes](image)
Factors Influencing Performance

- PubSub
- Decentralization of Task Executors
- Using KV Store Proxy

KV Store Proxy Using PubSub

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%

Tree Reduction

SVD 1

SVD 2

- Parallel-Invoker
- Task Invokers Using Scheduling Domains
- Multiple KV Store Shards
- KV Shards Have Their Own VM (Wukong)
Conclusion

- Serverless platform introduces unique challenges and opportunities

- Decentralization provides a large performance increase
  - Data locality and minimizing network overhead are also important to performance

- WUKONG achieves performance comparable to serverful Dask distributed running on general-purpose EC2 VMs
  - Improves performance by as much as 3.1x as problem size increases
Thank you!

Questions?

Contact: Benjamin Carver - bcarver2@gmu.edu

GitHub: https://github.com/mason-leap-lab/Wukong
SVD 50,000 × 50,000 CDF Plot

- redis_read_time
- redis_write_time
- dependency_processing
- invoking_downstream_tasks
- task_execution
- deserialize_time
SVD $n \times n$ with “ideal storage”
<table>
<thead>
<tr>
<th>SVD Phase #2</th>
<th>10k x 10k [2k x 2k]</th>
<th>25k x 25k [2k x 2k]</th>
<th>50k x 50k [5k x 5k]</th>
<th>100k x 100k [5k x 5k]</th>
<th>256k x 256k [5k x 5k]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPaths</td>
<td>95</td>
<td>565</td>
<td>345</td>
<td>1309</td>
<td>8376</td>
</tr>
<tr>
<td>NumTasks</td>
<td>172</td>
<td>800</td>
<td>507</td>
<td>1727</td>
<td>10509</td>
</tr>
<tr>
<td>NumLambdas</td>
<td>~84</td>
<td>~480</td>
<td>~295</td>
<td>~1082</td>
<td>8267 to 10511</td>
</tr>
<tr>
<td>LeafTasks</td>
<td>30</td>
<td>182</td>
<td>110</td>
<td>420</td>
<td>2756</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SVD Phase #1</th>
<th>200k x 100 [10k x 100]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPaths</td>
<td>20</td>
</tr>
<tr>
<td>NumTasks</td>
<td>42</td>
</tr>
<tr>
<td>NumLambdas</td>
<td>~20</td>
</tr>
<tr>
<td>LeafTasks</td>
<td>20</td>
</tr>
<tr>
<td>Algorithm</td>
<td>ScaLAPACK (sec)</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>SVD</td>
<td>57,919</td>
</tr>
<tr>
<td>QR</td>
<td>3,486</td>
</tr>
<tr>
<td>GEMM</td>
<td>2,010</td>
</tr>
<tr>
<td>Cholesky</td>
<td>2,417</td>
</tr>
</tbody>
</table>

Table 1: A comparison of ScaLAPACK vs numpywren execution time across algorithms when run on a square matrix with N=256K

Rank 5 Compressed Singular Value Decomposition of a Square Matrix

![Graph](image-url)