

# WUKONG

## 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



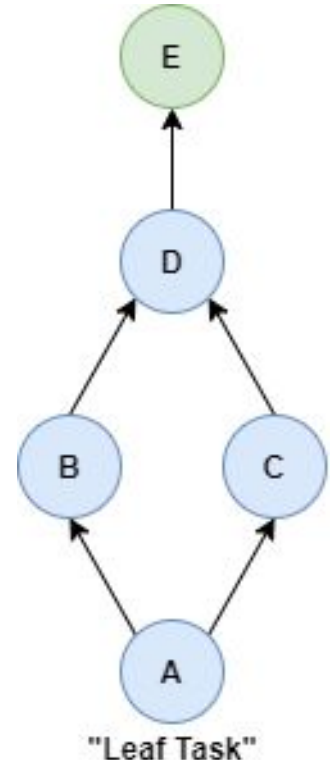
# AWS Lambda



- Amazon Web Service's FaaS offering
- Can vary serverless function memory size between 128 - 3008MB
- Can vary execution time between 3 seconds - 15 minutes
- Cost:
  - \$0.0000002 per request
  - \$0.00001667 per GB second
- Supports JS, Go, Python, Ruby, Java, C#, and Powersh

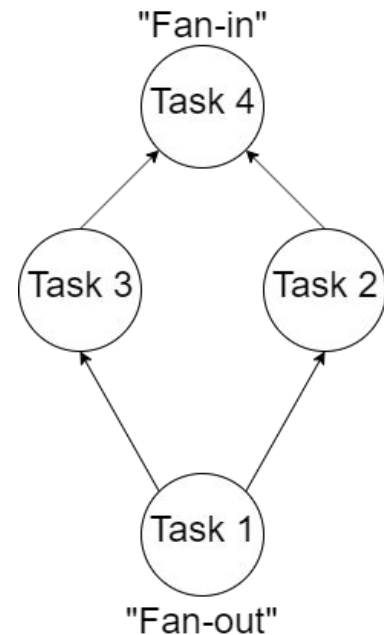
# Background - DAG Scheduling

- Data analytics applications can be modeled as a **directed acyclic graph (DAG)** based workflow
- **Nodes** represent **computations** or “**tasks**”
  - Computations can be executed by a processor.
  - May require reading/writing shared memory.
- **Edges** represent **dependencies between tasks**.
  - Nodes can only be executed after their immediate predecessors have been executed.



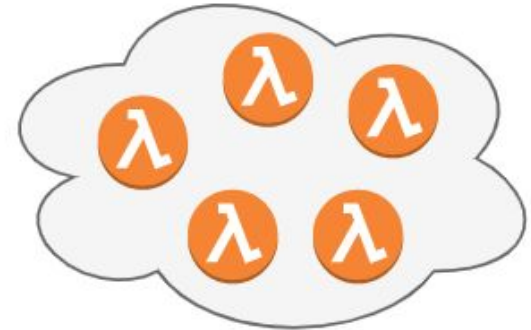
# Background - DAG Scheduling

- DAG workflows well-suited for serverless computing (or Functions-as-a-Service)
  - **Auto-scaling** accommodates **short** tasks and **bursty** workloads
  - Workloads with short tasks can take advantage of **fine-grained pricing** used by FaaS providers.
    - **In general, pay-per-use pricing keeps cost of short tasks low.**



# From Serverful to Serverless (Pt. 1)

- 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



# From Serverful to Serverless (Pt. 2)

- Assumptions of traditional serverful schedulers do not necessarily hold.
- A hypothetical serverless DAG scheduler may not necessarily care about traditional “scheduling”-related metrics and constraints (e.g., load balancing, cluster utilization).
- Individual tasks can be executed anywhere in the serverless data center (which is essentially managed by the serverless provider).

# AWS Lambda Constraints

- Lambda function invocation currently takes 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]
  - 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

**Wukong solves these  
drawbacks.**

# Wukong

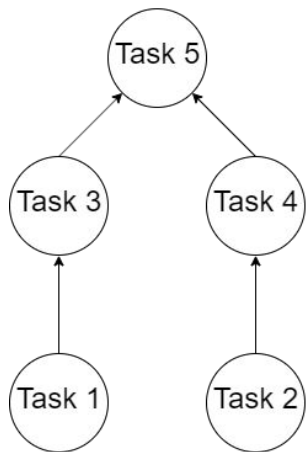
- **Approach**
- Architecture
  - Static Scheduler
  - Task Executors
  - Storage Cluster
- Evaluation

**Task executors cooperate here!**

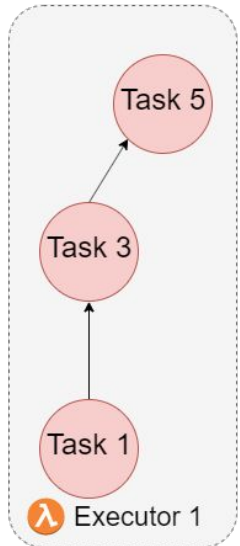
# Our Approach - Wukong

## Static Scheduling

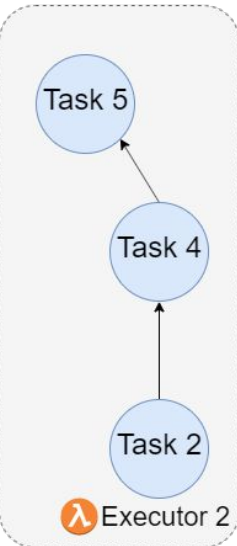
Original DAG



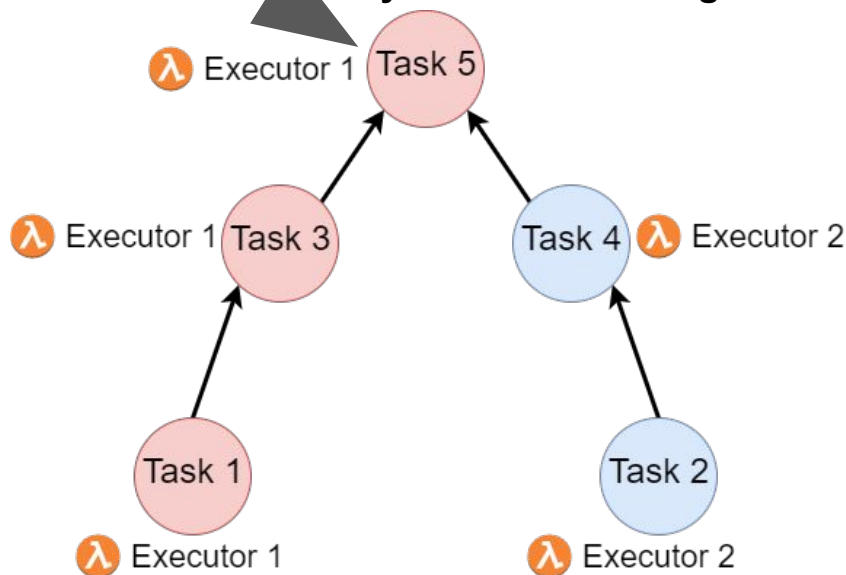
Static Schedule 1



Static Schedule 2



## Dynamic Scheduling

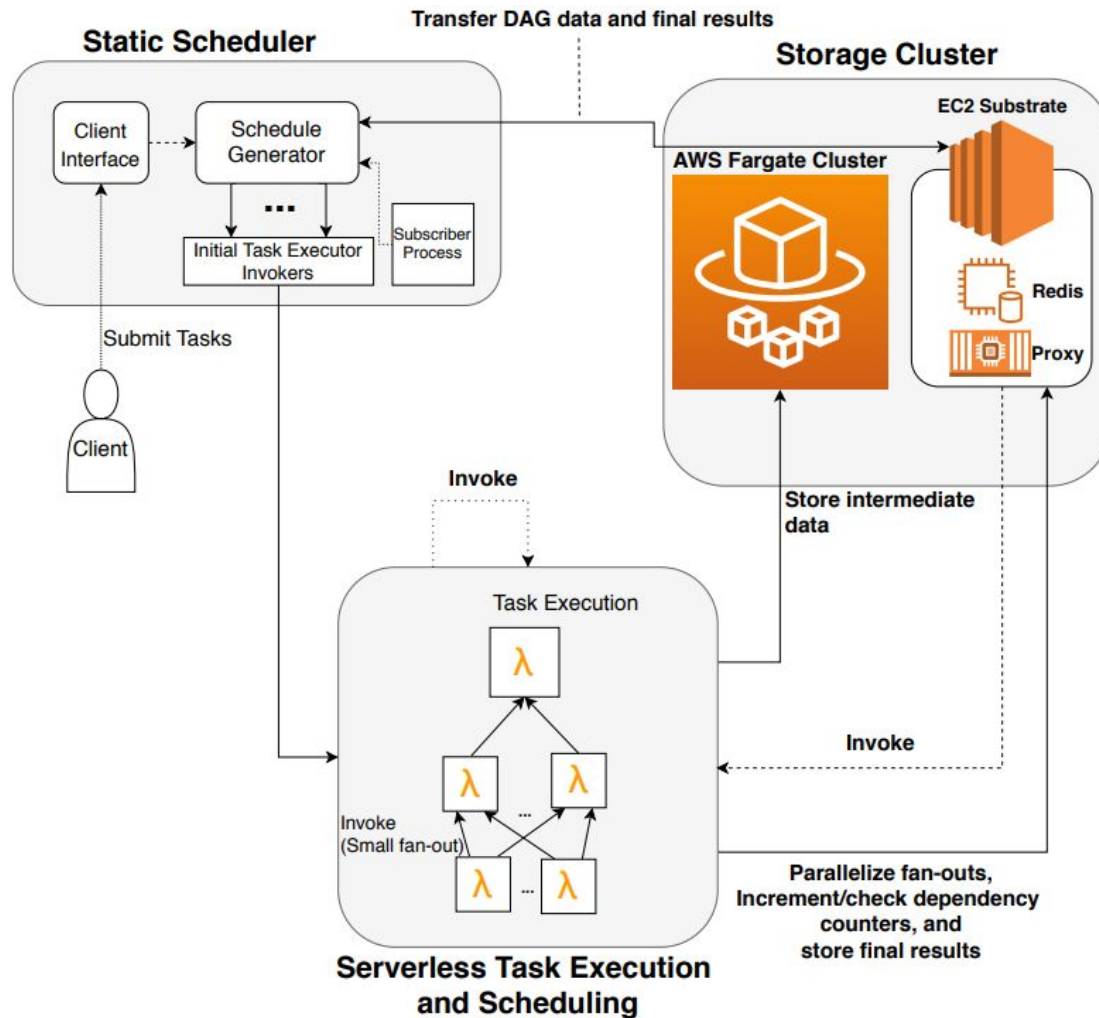


- Statically partition DAG into sub-DAGs
  - Assign each partition to a Lambda function

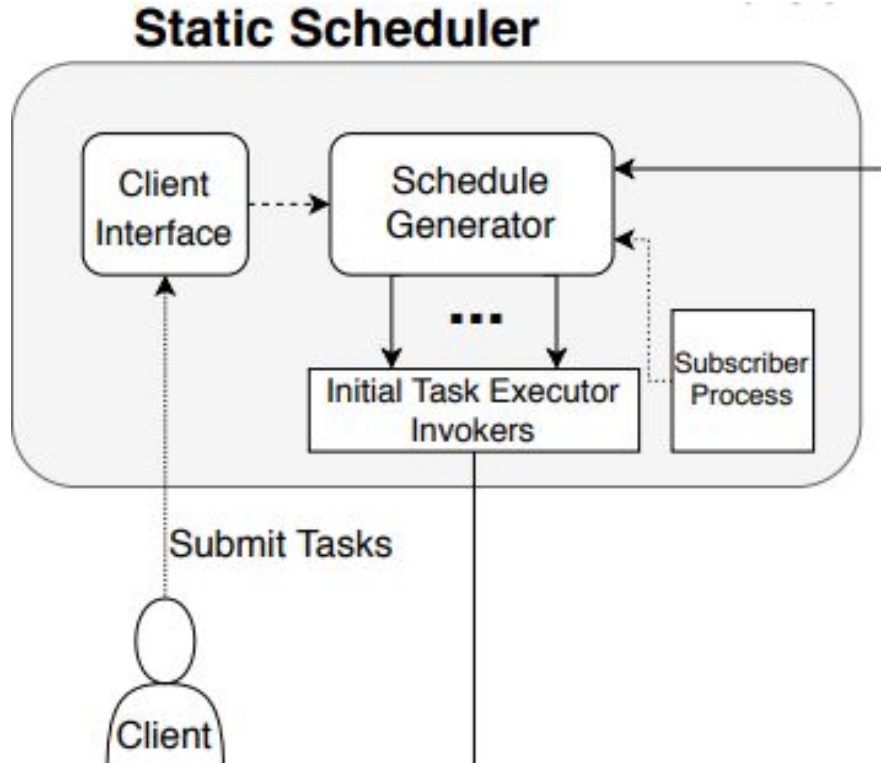
- Decentralized, cooperative scheduling
  - Lambda functions coordinate with each other to execute overlapping sections of assigned sub-DAGs

# Wukong

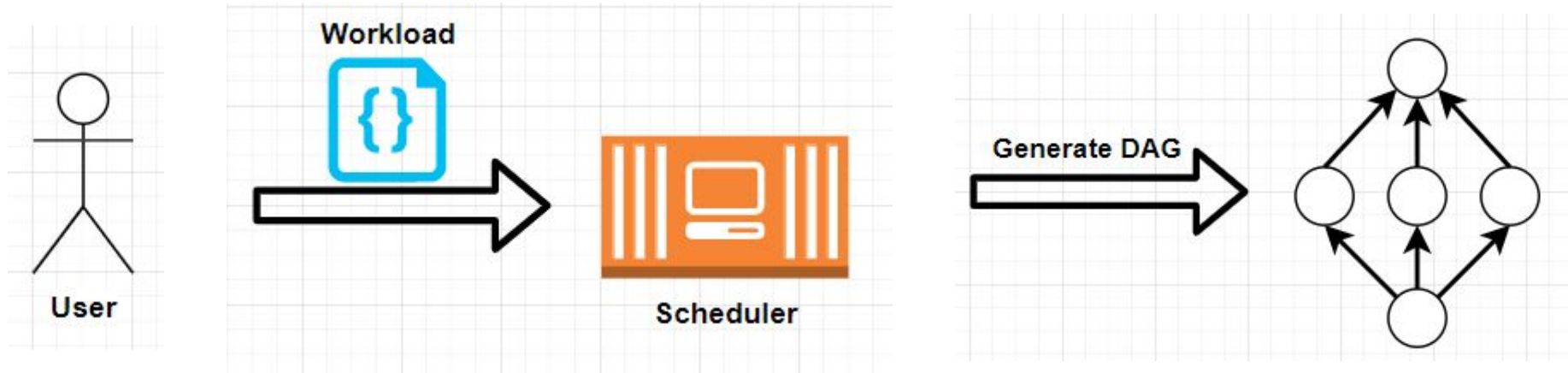
- Approach
- **Architecture**
  - **Static Scheduler**
  - **Task Executors**
  - **Storage Cluster**
- Evaluation



# Static Scheduler



- Partitions DAG into sub-DAG using a depth-first search (DFS) from each leaf node.
- Assigns sub-DAGs to executors
- Returns final results back to client.

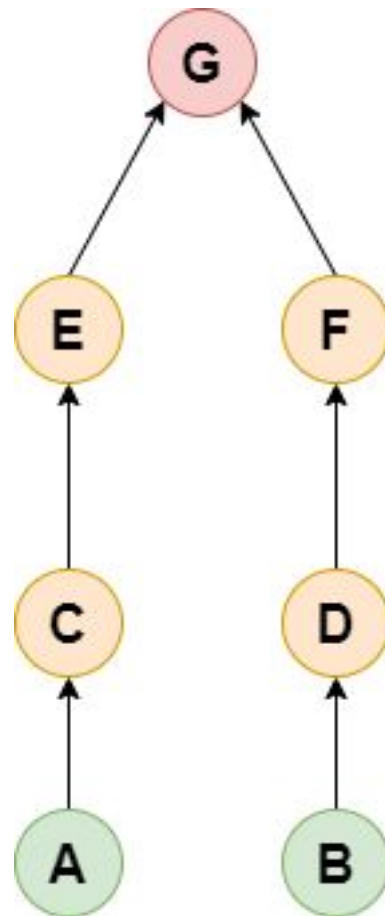


- (1) User submits a workload to the Scheduler**
- (2) Scheduler uses Dask to generate a DAG from the workload.**
- (3) Scheduler performs pre-processing on the newly-generated DAG.**
- (4) Scheduler assigns static schedules to serverless task executors.**

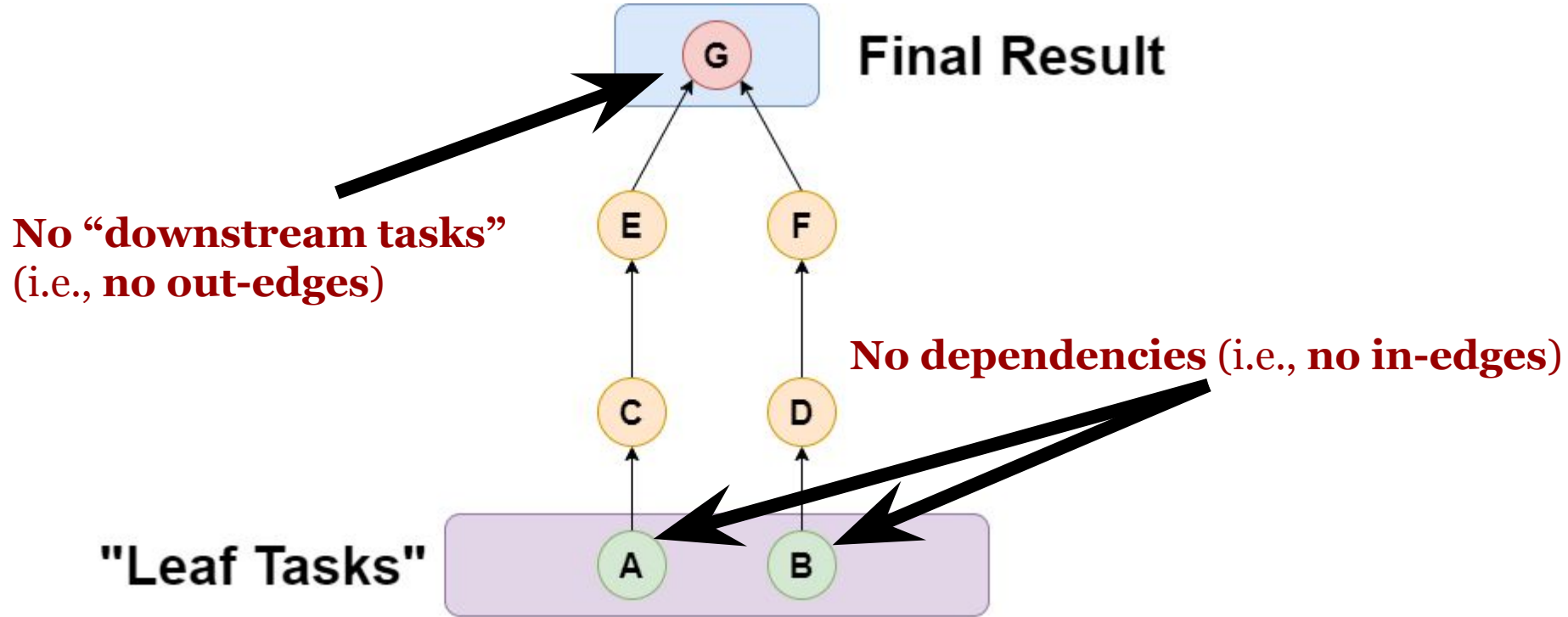


# DAG Pre-processing Stage

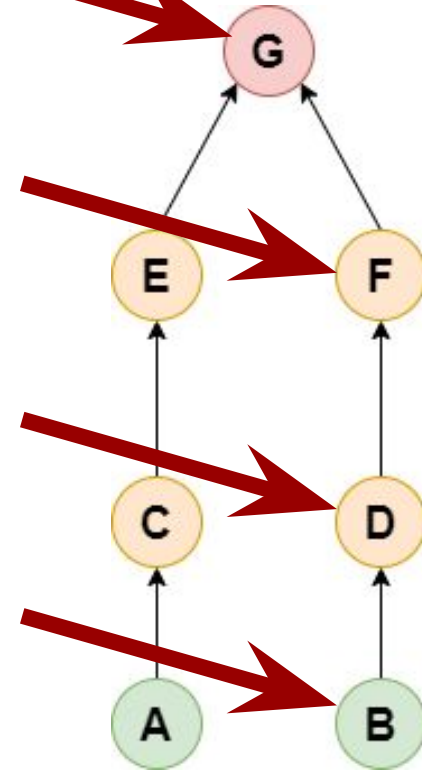
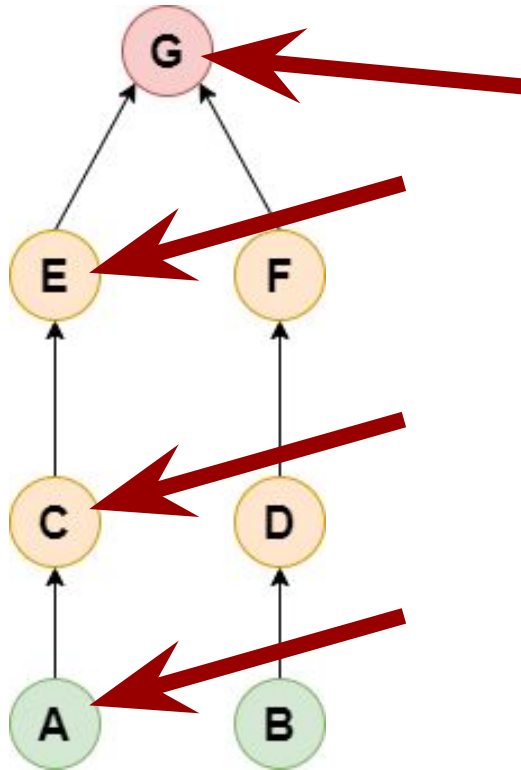
1. Generate DAG from user-submitted workload.
2. Iterate over generated tasks to find “**leaf tasks**”, or the tasks with **no dependencies** (i.e., in-edges), and “**final results**”, or tasks with no “**downstream tasks**” (i.e., out-edges).
3. Perform a series of **depth-first searches** beginning at each “leaf task” to partition DAG into a series of “**static schedules**”.



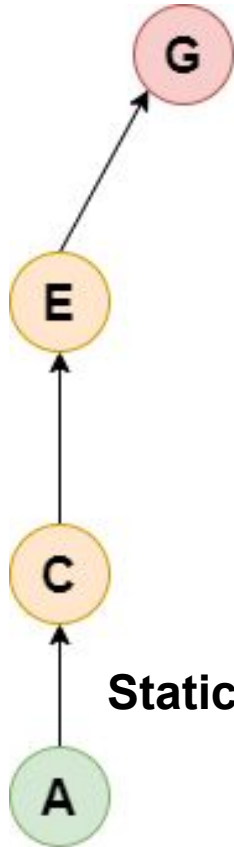
# Identifying “Leaf Tasks” and Final Results



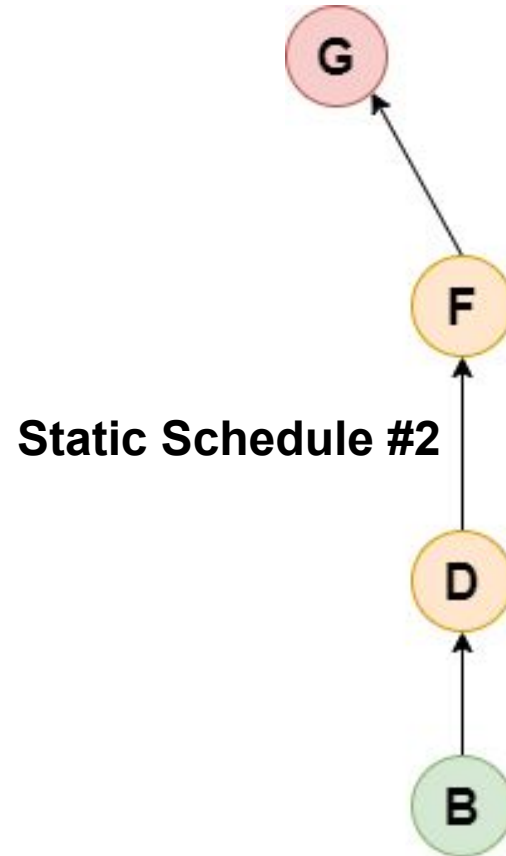
# Depth-First Search (DFS)



# Two Static Schedules



**Static Schedule #1**

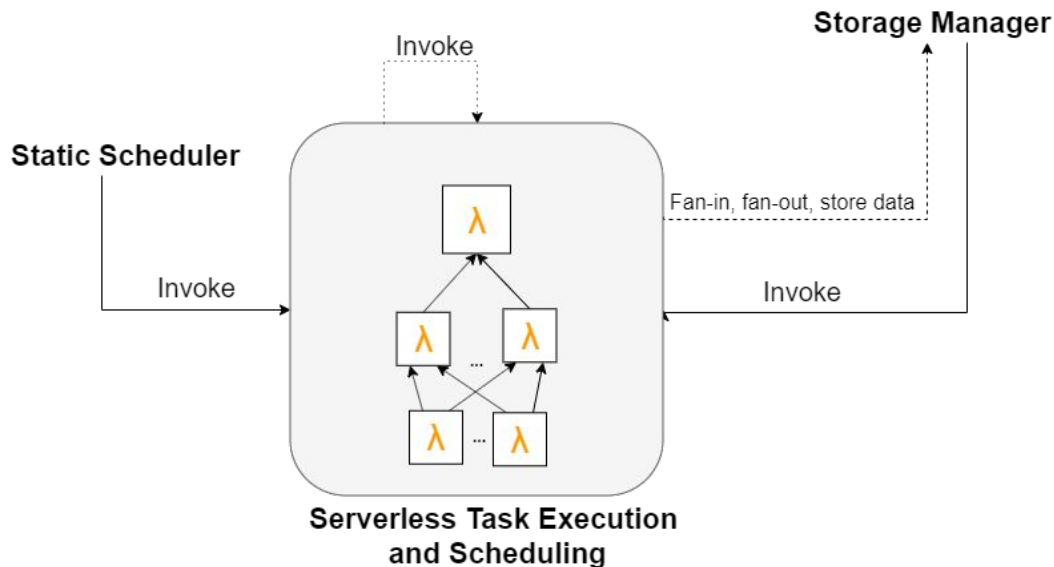


**Static Schedule #2**

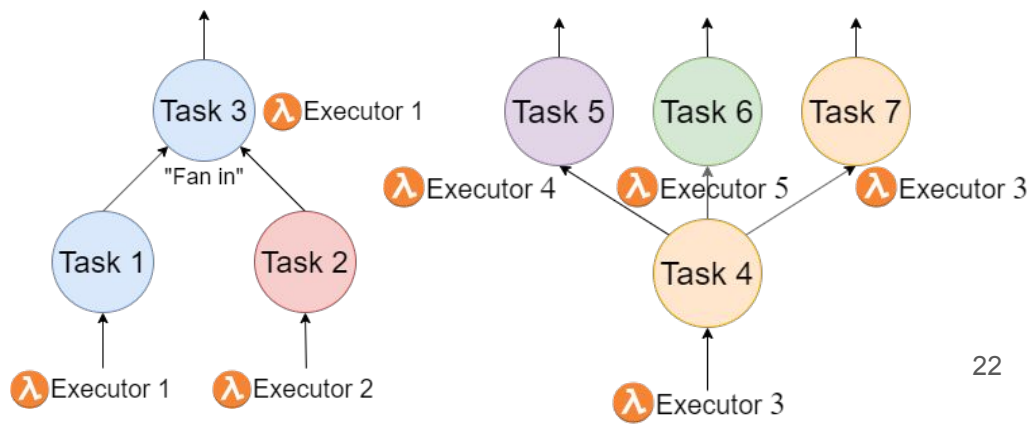
# Post-DAG Preprocessing Steps

- Once the DFS has been completed, the Static Scheduler will serialize each of the static schedules and store them in the Key-Value Store (Redis).
- After storing the static schedules, the Static Scheduler will invoke a AWS Lambda Task Executor for each leaf task.
  - If the static schedule is  $< 256$  kB in size, then the Scheduler will send the static Schedule to the Task Executor via the invocation payload.
  - If the static schedule is  $> 256$  kB in size, then the Scheduler will simply pass the Redis key of the static schedule. The Task Executor will retrieve the static schedule once it begins execution.

# Executors



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



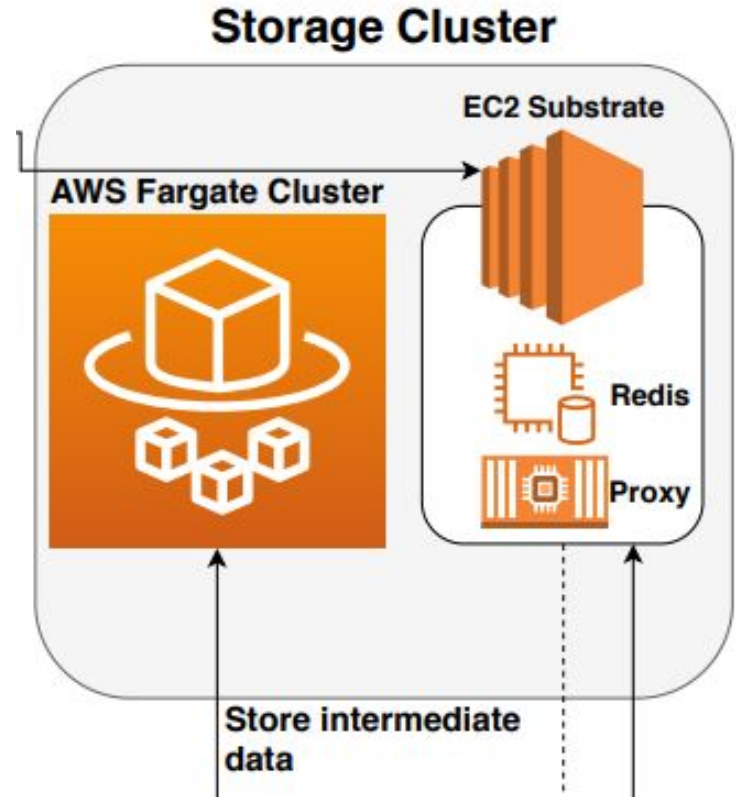


# AWS Fargate

- “Serverless compute engine” for containers
- Works with Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Servers (EKS)
- AWS Fargate is responsible for provisioning and managing the servers; user just specifies resource allocation.
- Relatively inexpensive

# Storage Cluster

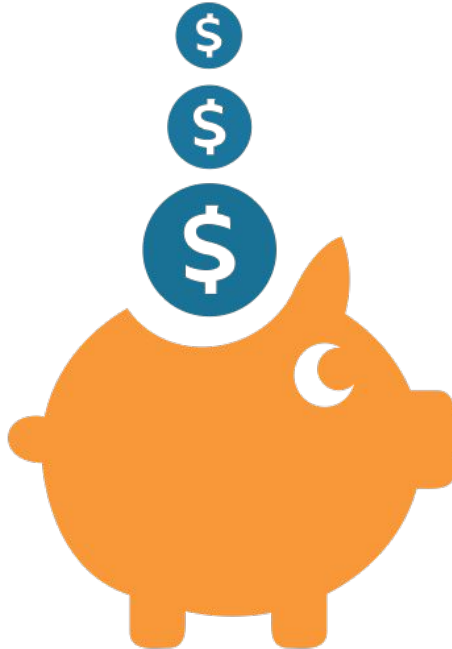
- Performs storage operations on behalf of Executors and Static Scheduler
- Uses AWS Fargate cluster for intermediate data storage.
- Use additional, separate Redis instance running on EC2 for dependency counters and static schedule storage.



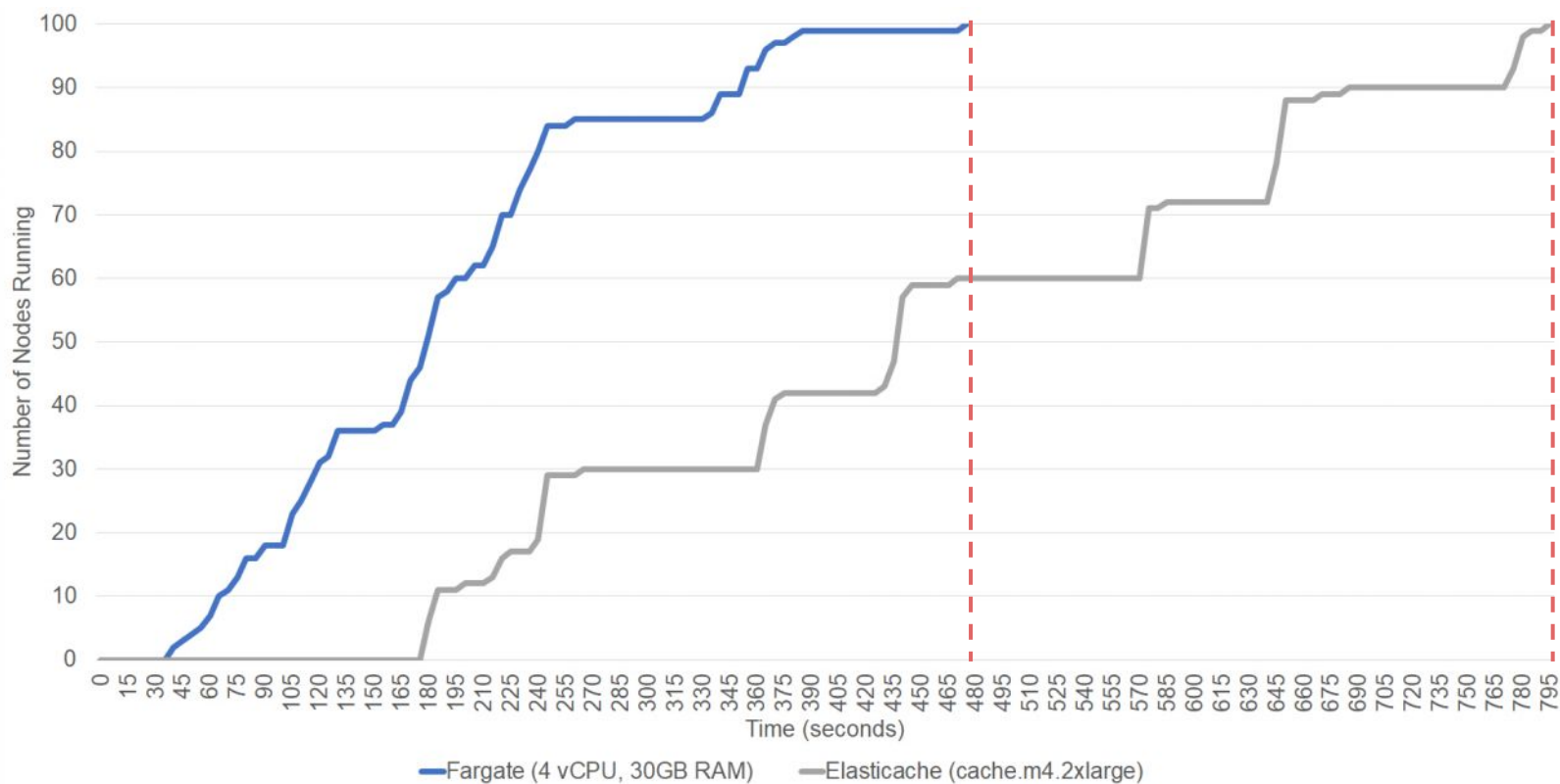


# Storage Cluster - Cost Effectiveness

- Fargate Storage Cluster is **64.8%** cheaper than hosting an equivalent cluster on EC2 and **79.4%** cheaper than hosting the same number of Redis instances on AWS ElastiCache.



# Storage Cluster Elasticity



# 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?

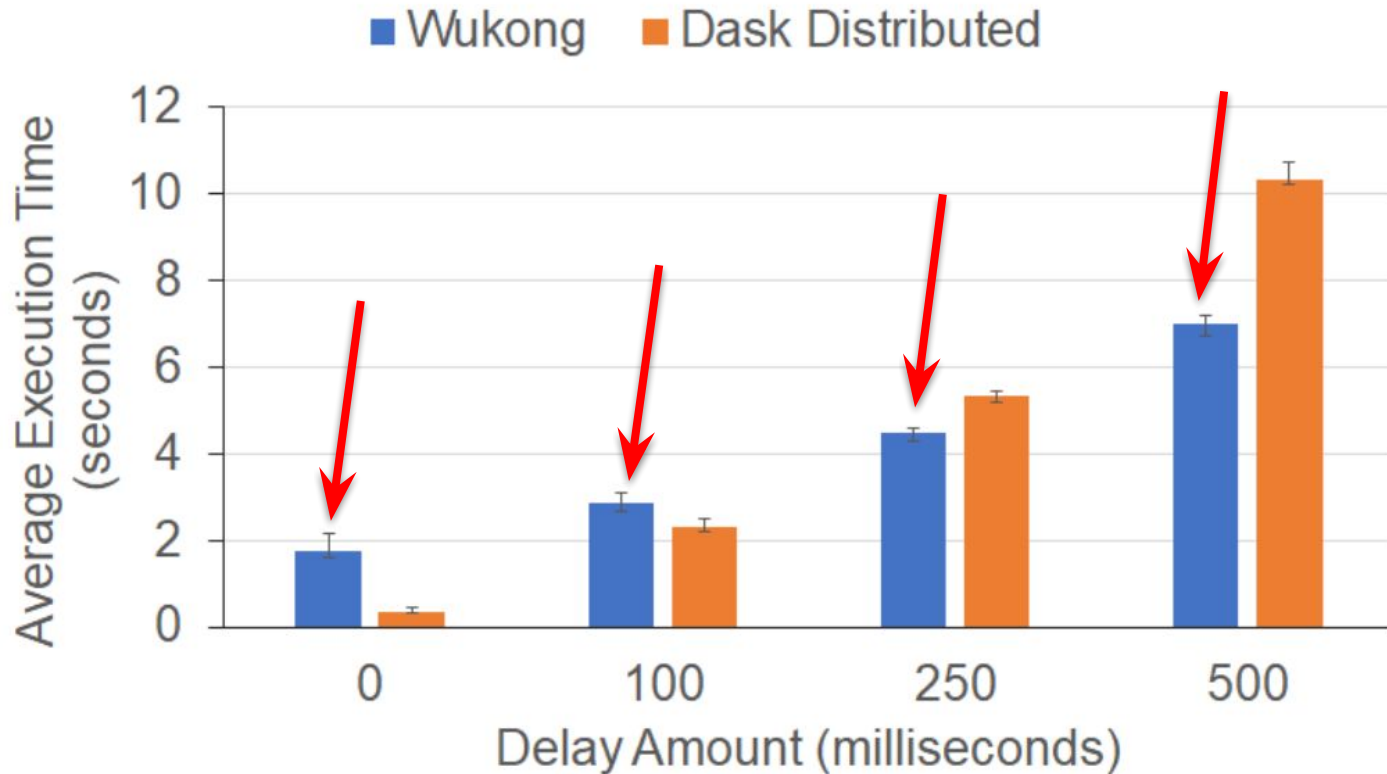
# Experimental Setup

- Compare against Dask `distributed`.
  - 11-node EC2 cluster of `r5.4xlarge` VMs
- Wukong Static Scheduler, KV Store, and KV Store Proxy running on `r5n.16xlarge` EC2 VMs.
- Task Executor allocated 3GB memory with timeout set to two minutes.

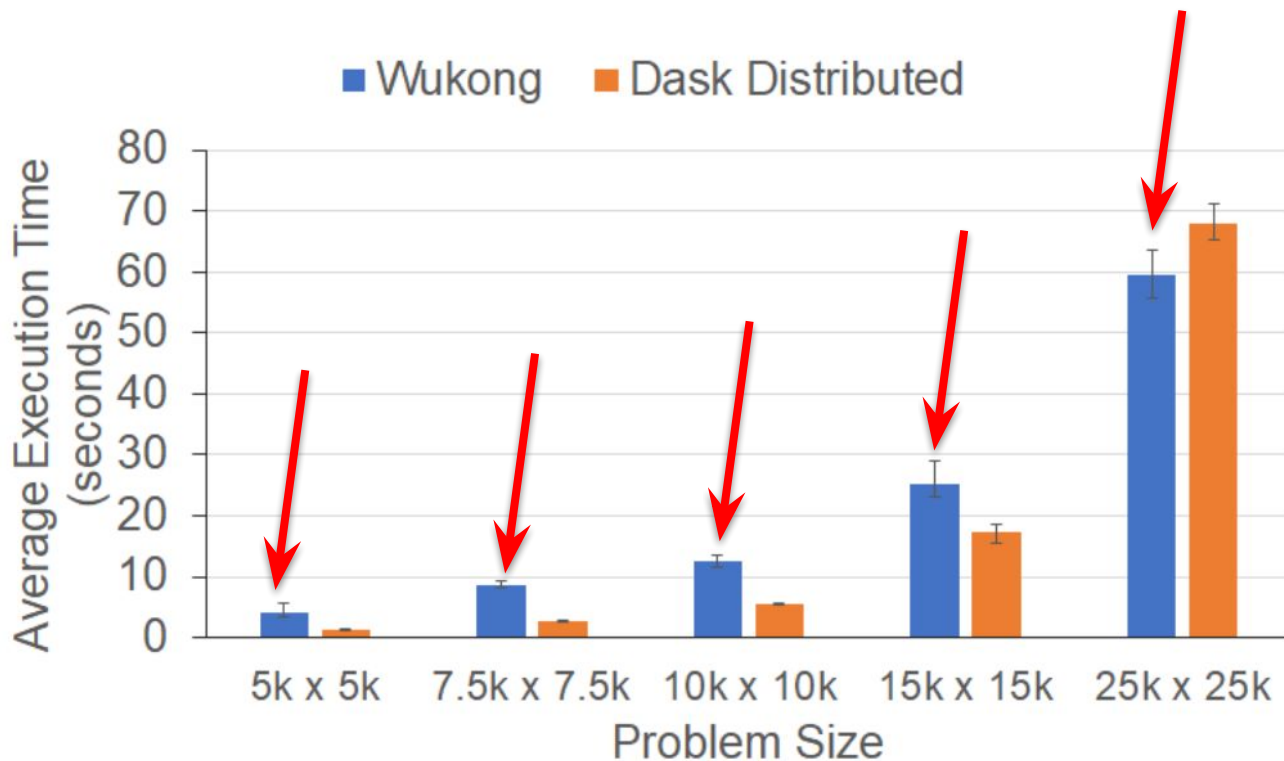
# Four DAG Applications

- Microbenchmark
  - **Tree Reduction:** repeatedly add adjacent elements of an array until a single value remains
- Linear Algebra
  - **General Matrix Multiplication (GEMM)**
    - $5,000 \times 5,000$  through  $25,000 \times 25,000$
  - **Singular Value Decomposition (SVD)**
    - $n \times n$  matrix and a tall-and-skinny matrix, varying sizes
- Machine Learning
  - **Support Vector Classification (SVC)**
    - 100,000 - 8,192,000 samples

# Microbenchmark - Tree Reduction with Delays

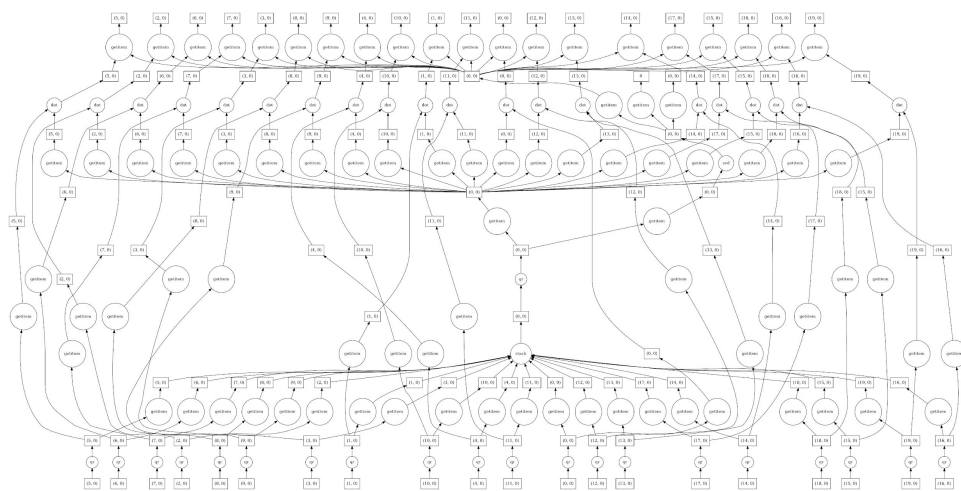
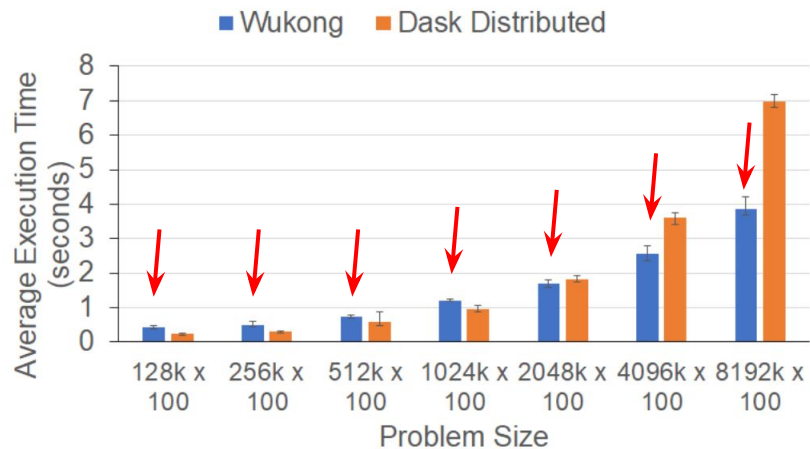


# General Matrix Multiplication (GEMM) -- Dask





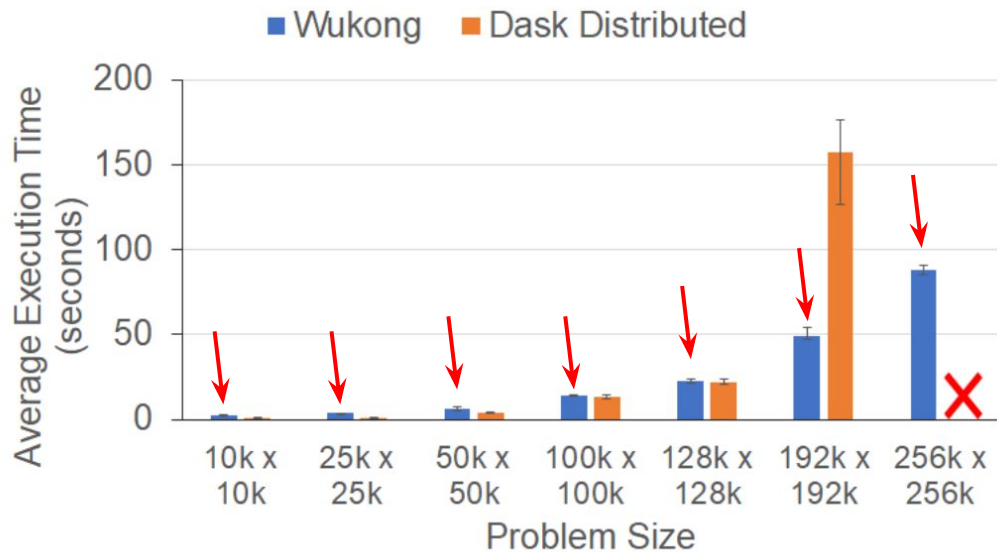
# Singular Value Decomposition (SVD) - “Tall and Skinny” SVD tall-and-skinny



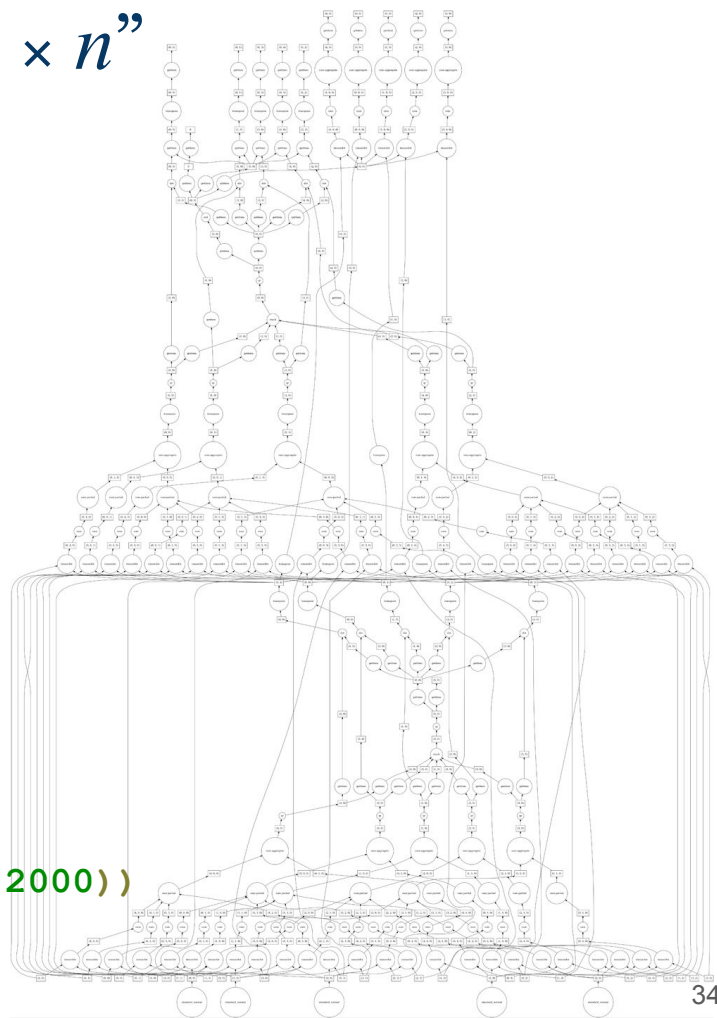
```
X = da.random.random((200000, 100), chunks=(10000, 100))
u, s, v = da.linalg.svd(X)
v.compute() # Begin execution
```

# Singular Value Decomposition - “ $n \times n$ ”

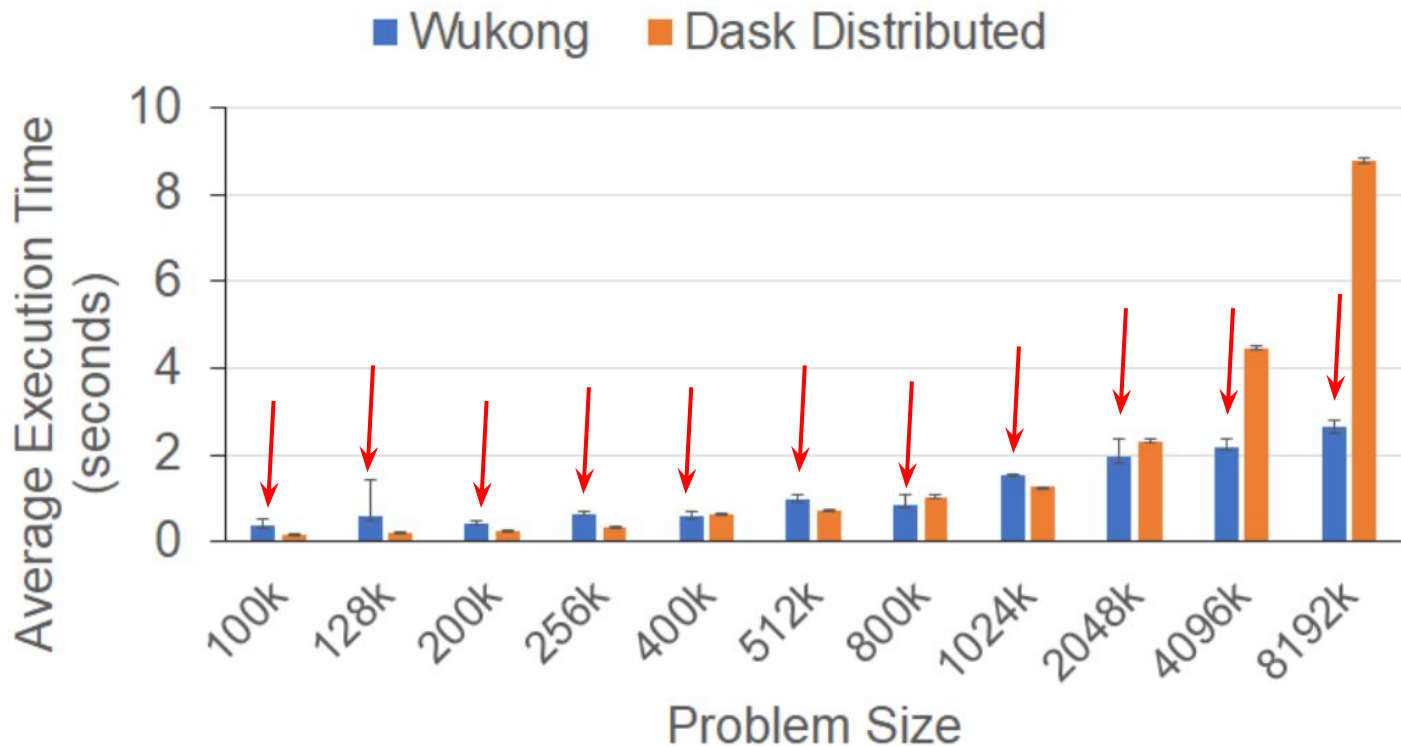
## SVD-Compressed (rank 5) $n \times n$



```
X = da.random.random((10000, 10000), chunks=(2000, 2000))
u, s, v = da.linalg.svd_compressed(X, k=5)
v.compute() # Begin execution
```



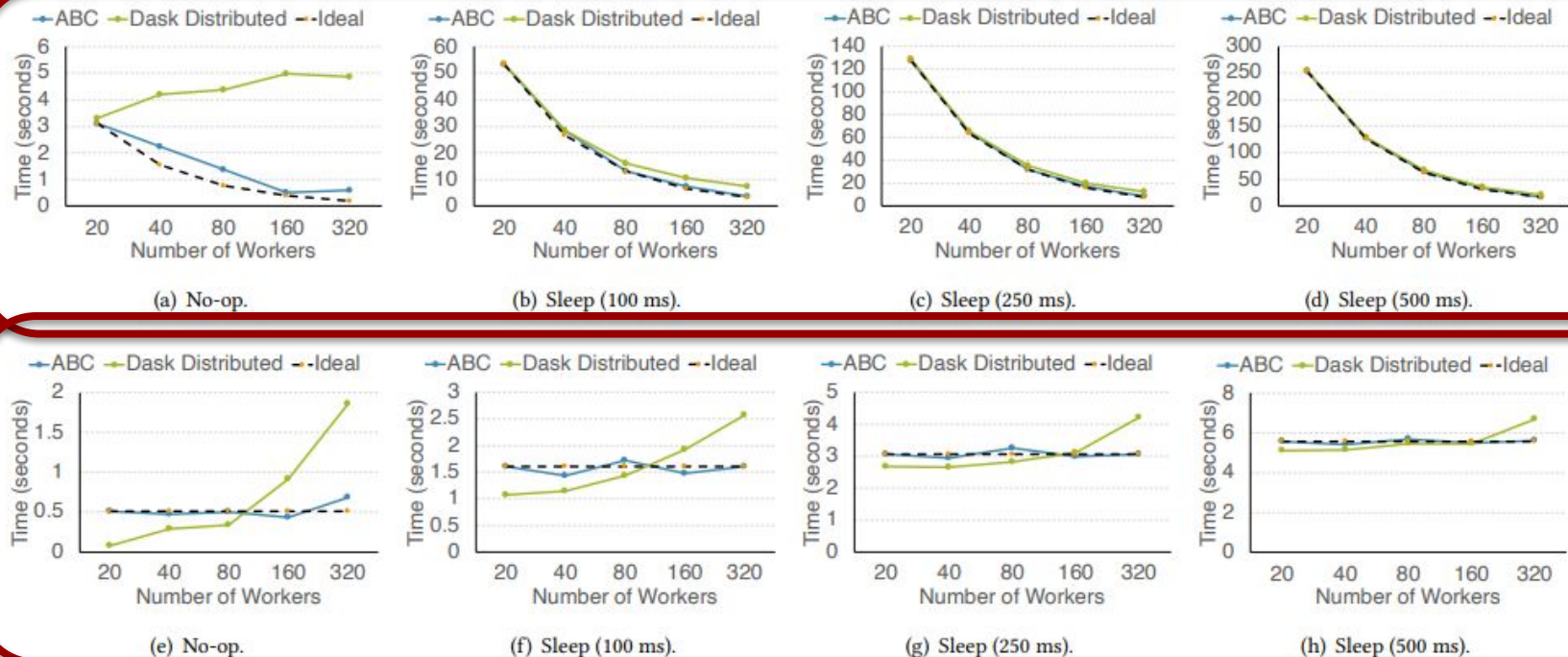
# Support Vector Classification (SVC)



# Scalability - Strong & Weak Scaling

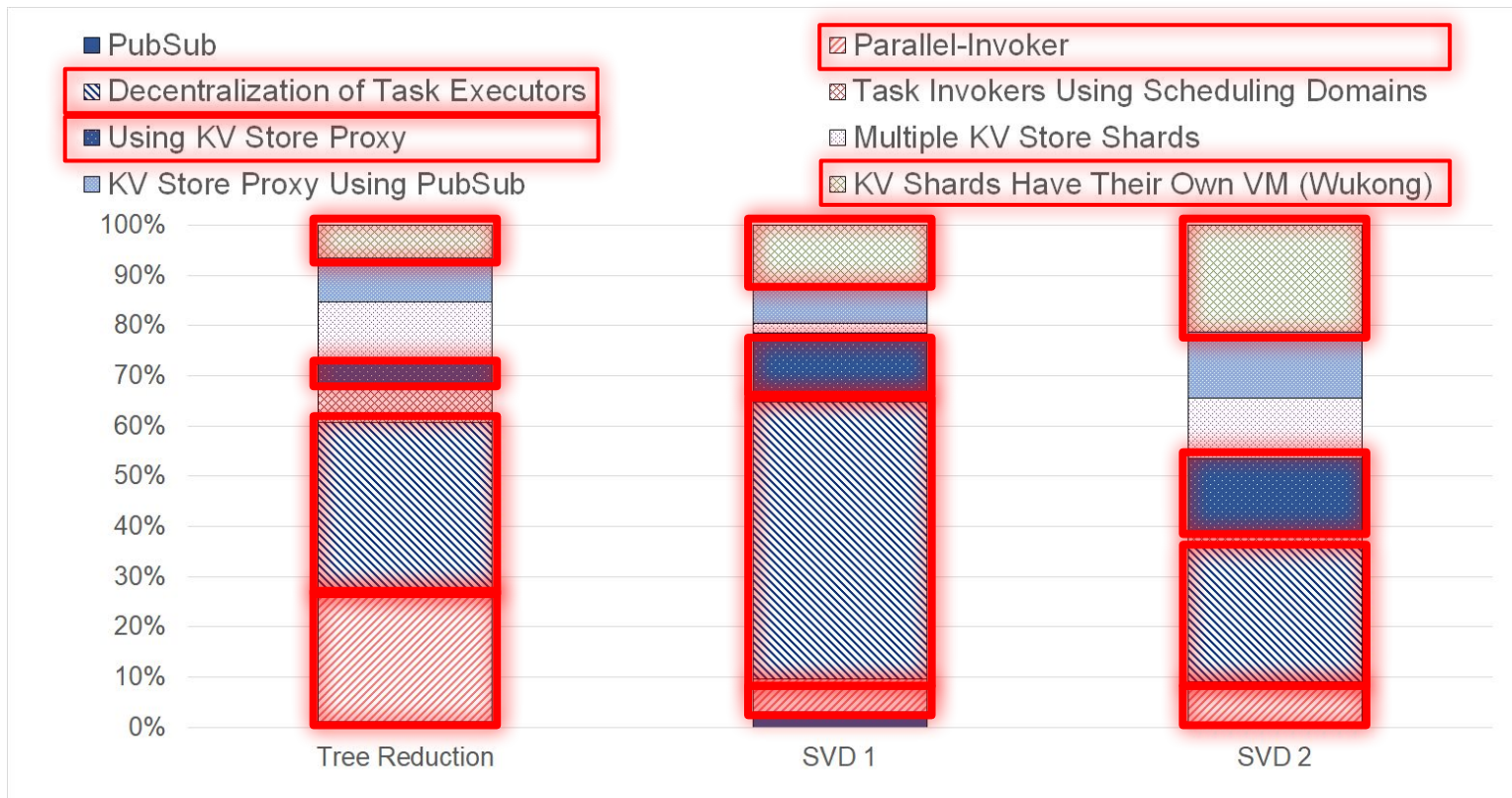
- **Weak Scaling:**
  - **Both** the **number of workers** and the **problem size** are **increased**.
    - **Keep amount of work per worker fixed, add more workers.**  
**Does performance improve?**
  - Usually relevant for **memory-bound** tasks.
- **Strong Scaling:**
  - The number of workers is **increased** while the problem size **remains constant**.
    - **Keep problem size fixed, but add more workers. Does performance improve?**
  - Usually relevant for **CPU-bound** tasks.

# Scalability of Wukong (ABC)



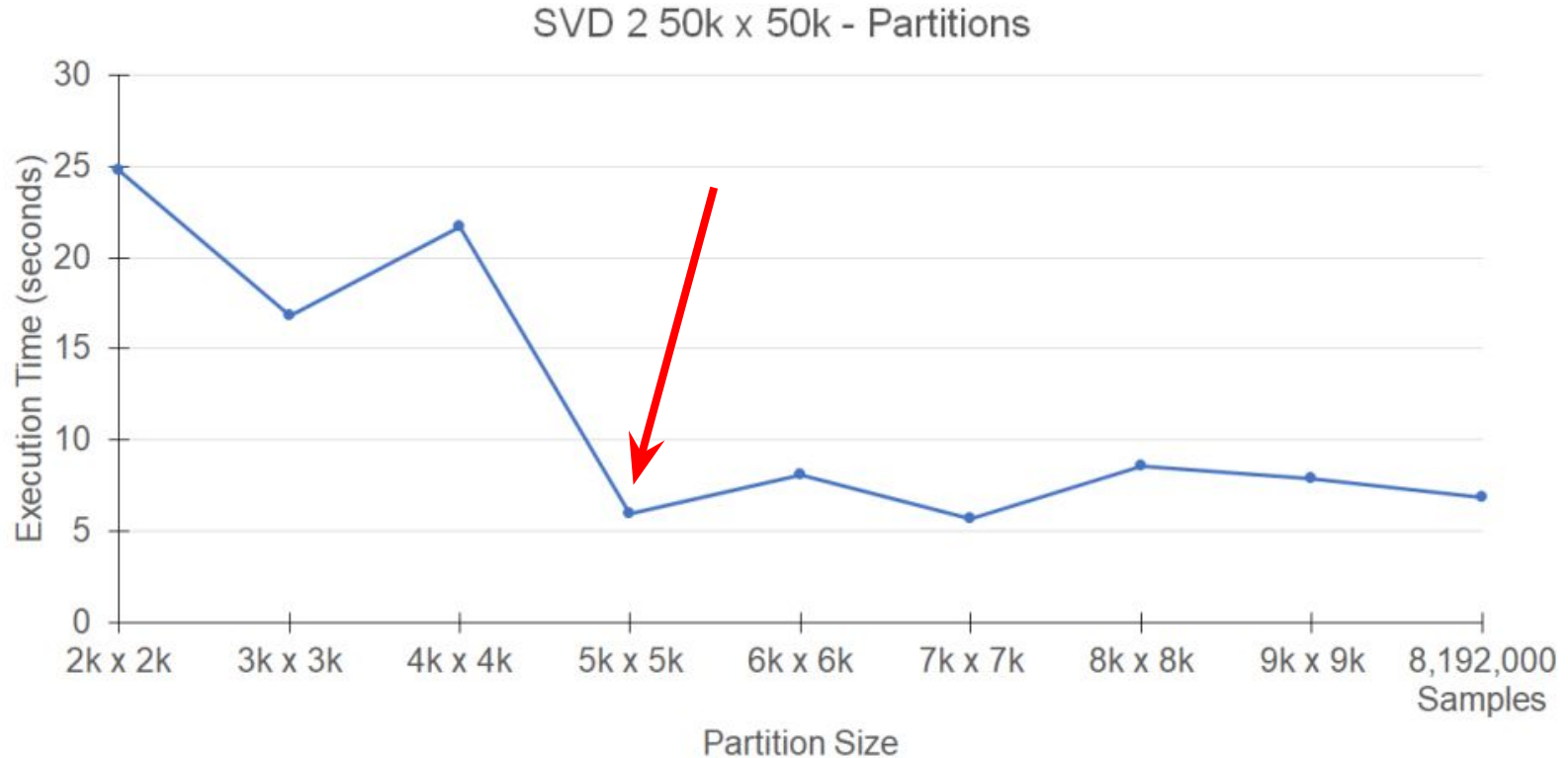
**Figure 13:** Strong scaling vs. weak scaling. Figure 13(a)–13(d) – strong scaling: time (Y-axis) to execute 10,000 tasks over  $N$  workers (X-axis). Figure 13(e)–13(h) – weak scaling: time to execute 10 tasks per worker. For each row, plots are for (from left to right) tasks of 0, 100, 250, and 500 ms.

# Factors Influencing Performance of Wukong

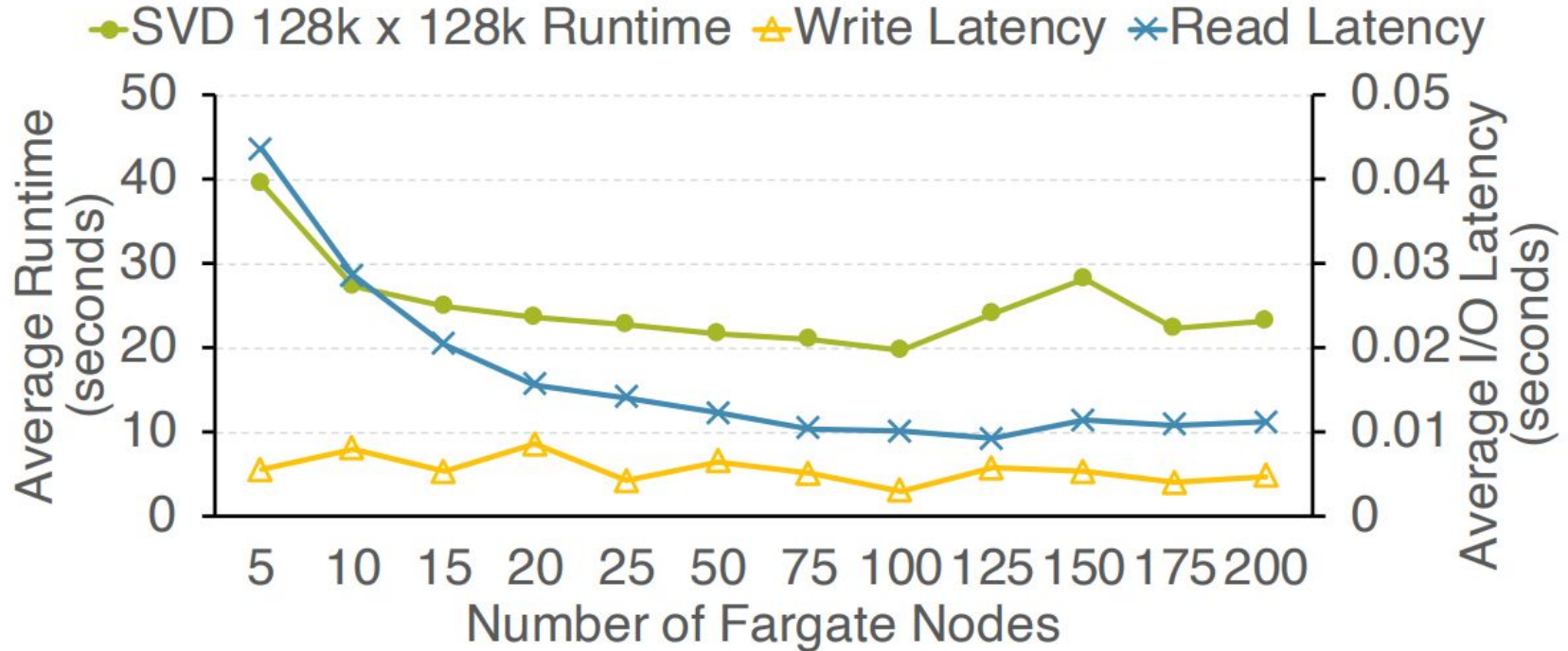




# Workload Parameters - Partition Size



# Workload Parameters - Fargate Cluster Size

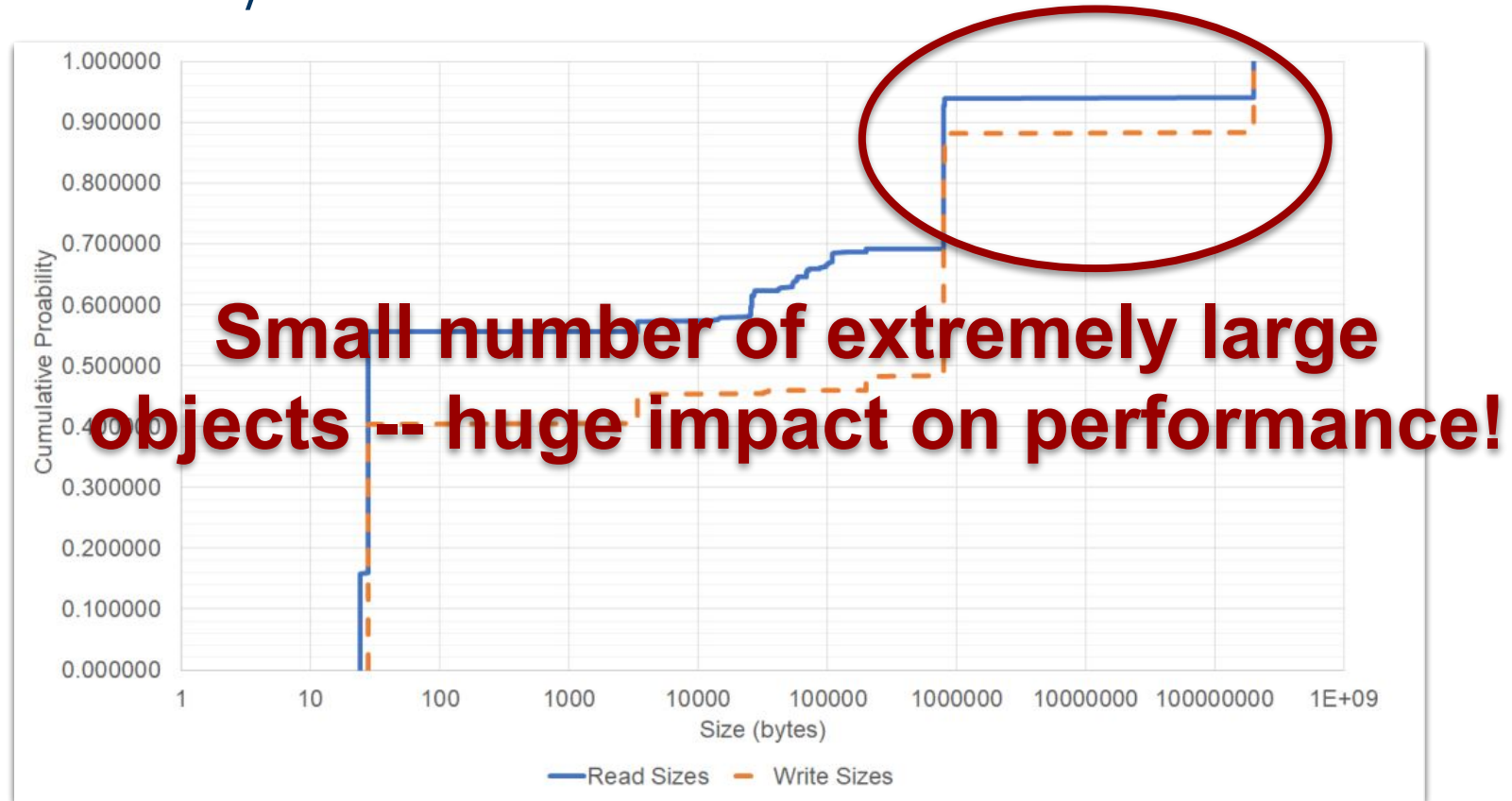




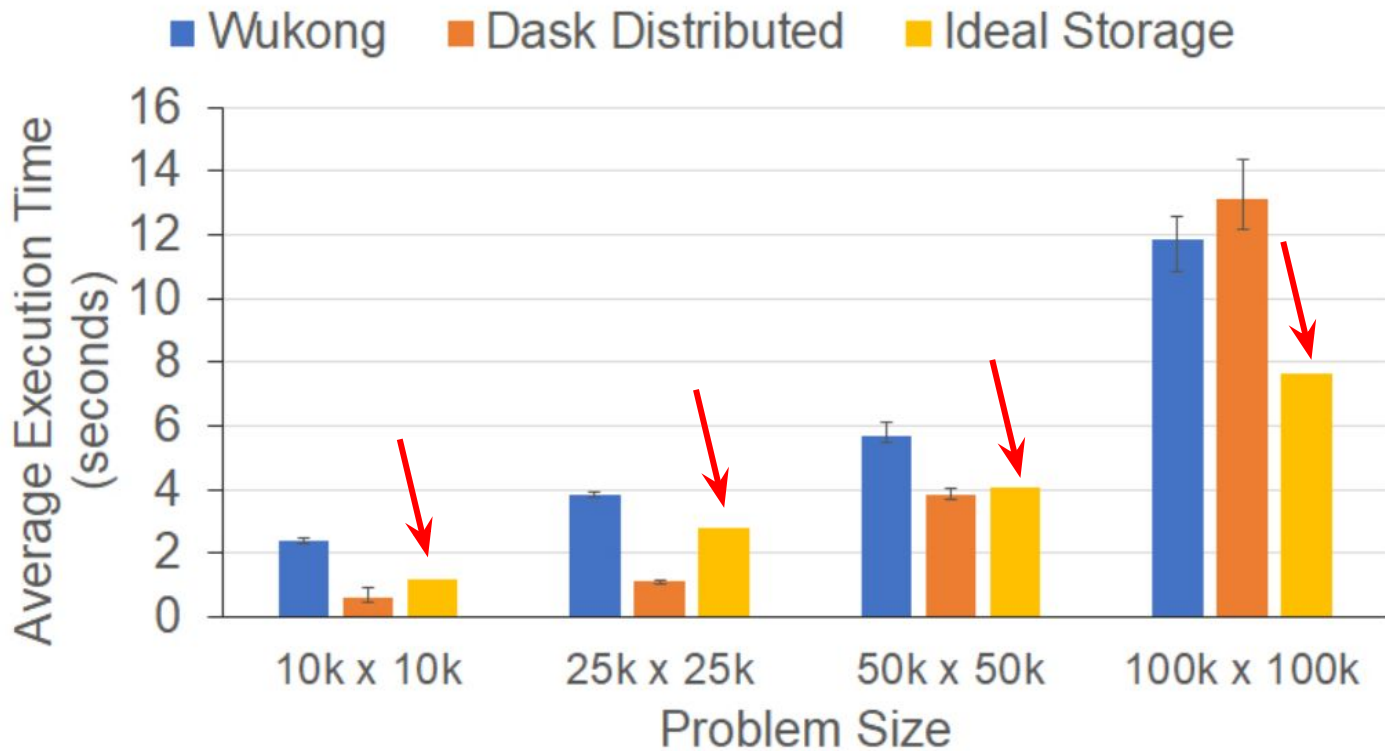
# Diagnosing Performance - Workload Characteristics

- How is the overall execution time divided up?
- What activities are being performed? Taking the longest?
- How can we optimize the common case?

# Workload I/O Characteristics



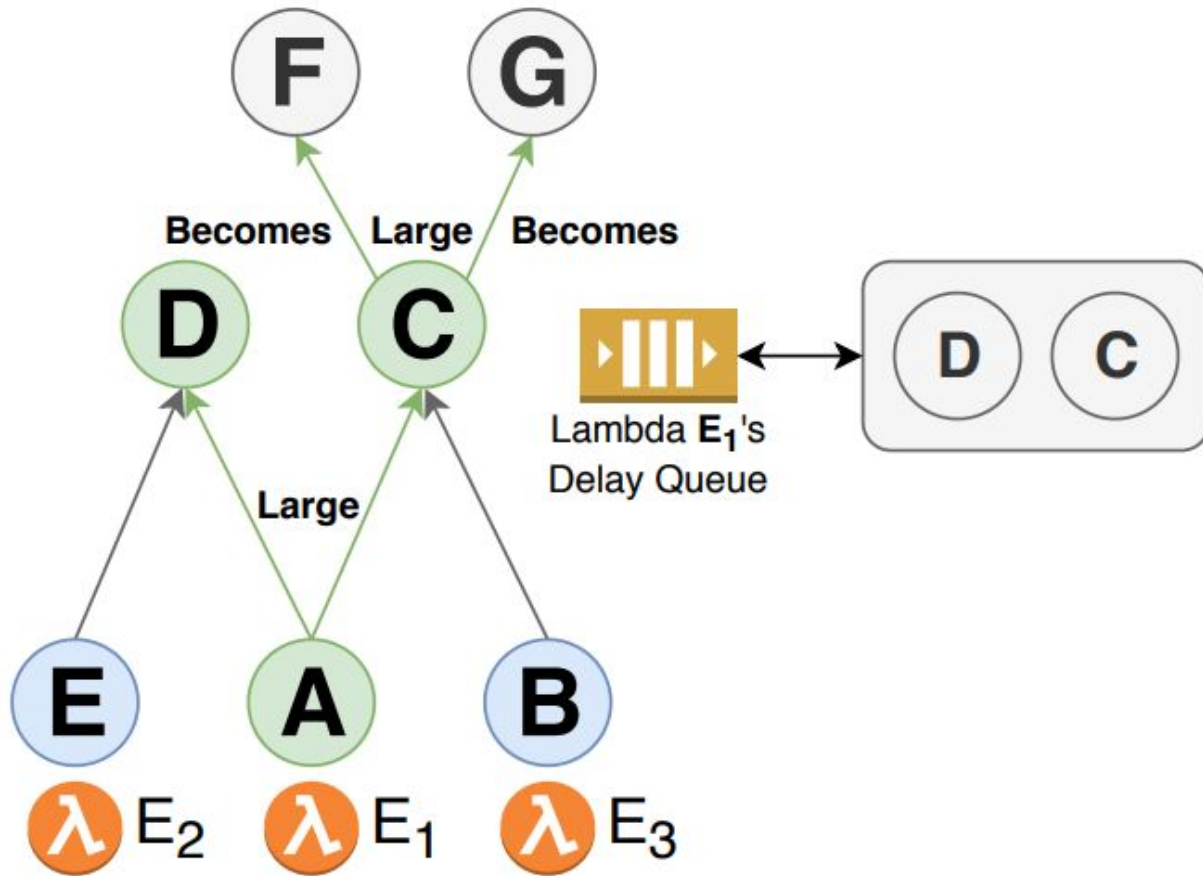
# SVD 50k x 50k Performance with Ideal Storage





# Task Clustering and Delayed I/O

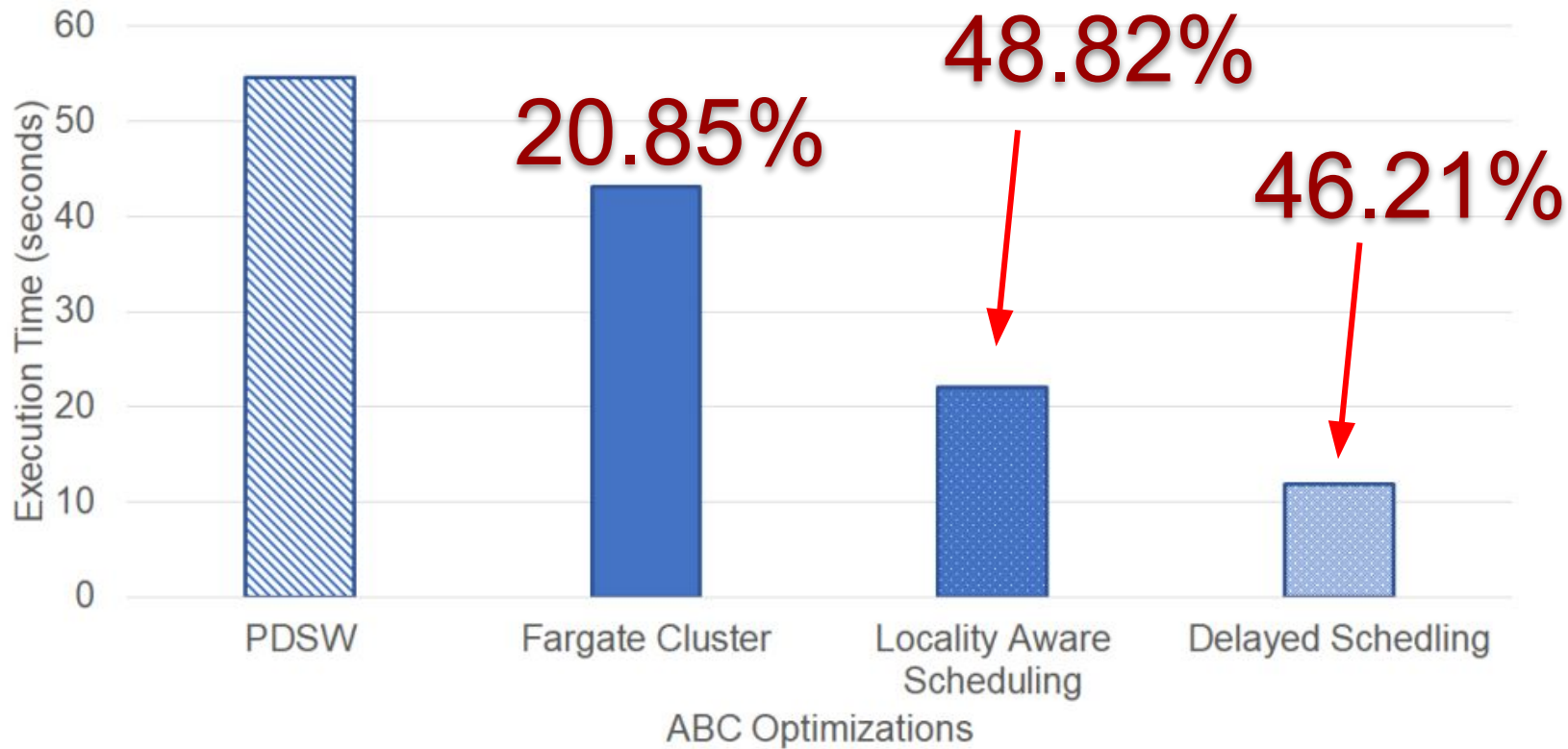
- We developed new techniques to eliminate large-object I/O during execution.
- Task Clustering
  - After executing a task that produces “large” intermediate data, execute all immediate downstream tasks locally on the same Task Executor.
- Delayed I/O
  - If there are some downstream tasks which depend on the large intermediate data *but are not yet ready to execute due to some other missing data dependency*, place these tasks in a queue in keep retrying them until they’re ready.



**Figure 6:** Task clustering and delay I/O.

# How do task clustering and delayed I/O impact performance?





**Overall: 4.6x faster compared to baseline.**<sub>48</sub>



# Do task clustering and delayed I/O impact cost?

- YES
- For SVD 50,000 x 50,000:
  - Task clustering + delayed I/O reduce workload cost by **65.23%** (from \$0.04556 to \$0.01584).
- For SVD 100,000 x 100,000:
  - Task clustering + delayed I/O reduce workload cost by **73.50%** (from \$1.17 to \$0.31).

# 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.

# Thank you!

## Questions?

**Contact:** Benjamin Carver - bcarver2@gmu.edu

**GitHub:** <https://github.com/mason-leap-lab/Wukong> 

