Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads

John Thorpe*, Yifan Qiao*, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, Harry Xu

UCLA University of Wisconsin CMU Google Brain Simon Fraser Princeton University
In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity. The main forms of precipitation include drizzle, rain, sleet, snow, graupel and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals within a cloud. Short, intense periods of rain in scattered locations are called “showers”.

What causes precipitation to fall?

gravity

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?

graupe

Where do water droplets collide with ice crystals to form precipitation?

within a cloud
Graph Neural Networks

Goals:
- Affordability
- Scalability
- Performance
Stages of a Graph Neural Network

Stages of a Graph Neural Network

- **Scatter**
  - Edge-cut partition
- **2-hop neighbors**
- **Message passing along edges**
- **1-hop neighbors**
Stages of a Graph Neural Network

- Scatter
- Apply Edge

Edge-cut partition

Apply neural network on edge data

1-hop neighbors

2-hop neighbors
Stages of a Graph Neural Network

1. Scatter
2. Apply Edge
3. Gather

Edge-cut partition

2-hop neighbors

1-hop neighbors

Aggregate messages from neighbors

0

1

2

3

4

5
Stages of a Graph Neural Network

1. Scattering
2. Apply Edge
3. Collecting
4. Apply Vertex

Edge-cut partition

2-hop neighbors

Apply neural network on aggregated data

1-hop neighbors

---

Stages of a Graph Neural Network

SAGA-NN

1-hop neighbors

2-hop neighbors

Scatter → Apply Edge → Gather → Apply Vertex

Edge-cut partition

0 1 2 3 4 5

Stages of a Graph Neural Network

1-hop neighbors

2-hop neighbors

SAGA-NN

Scatter → Apply Edge → Gather → Apply Vertex

Edge-cut partition

Stages of a Graph Neural Network

Stages of a Graph Neural Network

- Scatter
- Apply Edge
- Gather
- Apply Vertex

Goals:
- Affordability
- Scalability
- Performance

1-hop neighbors

2-hop neighbors

GNNs Comprise Very Different Workloads

Scatter

Gather

Memory intensive!

Apply

Compute intensive!

Input Layer

Hidden Layer 1

Hidden Layer 2

Predictions
GPUs Are Not a Good Fit for Graph Operations

Limited device memory + large adjacency matrix = poor scalability!
GPUs Are Not a Good Fit for Graph Operations

- GPUs work very well for tensor computation
  - Less efficient for Gather
  - Idle for Scatter across partitions
CPUs Are Not Efficient for Tensor Workloads

- CPUs provide scalability for graph operations
  - Not optimized for highly parallel computation
Combining CPUs and GPUs is Cost-Ineffective

Get the scalability of CPUs with performance of GPUs
- GPUs under-utilized during graph operations
Using Many CPU Servers Can Still Be Expensive

Allocating many CPU servers increases parallelism at the expense of cost
- Many unnecessary resources allocated along with CPU machines
Key Insight: Serverless Fits Our Goals

Serverless: cloud execution model that provisions resources on demand

Highly scalable interface fits needs of tensor computation
- Invoke thousands of threads in parallel

Low-cost, flexible pricing model

Fine grained: Only pay for compute resources on millisecond basis

Provide high performance-per-dollar (value)
Serverless Achieves Low-Cost, Scalable Efficiency

![Diagram showing the process of Scatter, Apply Edge, Gather, and Apply Vertex with a Serverless Pool in the center.](image)

21
Challenges with Using Serverless

- Each thread has limited resources
  - Weak CPU, limited memory

- Limited network
  - Design to handle light asynchronous tasks
Challenge 1: Limited Resources

Each serverless thread has limited memory and compute
- Better for highly parallel computation without dependencies
Solution: Computation Separation

Separation of graph and tensor computation
- Scale graph operations on CPU servers
Dorylus Architecture

Graph Servers

Serverless Pool

Weight Servers
Dorylus Architecture

Scatter

Graph Servers

Serverless Pool

Weight Servers
Dorylus Architecture

- Scatter
- Apply
- Edge
- Graph Servers
- Serverless Pool
- Weight Servers
Dorylus Architecture

Graph Servers

Serverless Pool

Weight Servers

Scatter

Apply

Edge

Gather
Dorylus Architecture

Serverless Pool

Graph Servers

Weight Servers

Scatter

Apply Edge

Gather

Apply Vertex
Flow of Decomposed Tasks

Layer 1 forward

Next Epoch

Layer 1 backward

Layer 2, …, Layer L forward

Layer 2, …, Layer L backward

\[\text{Scatter} \rightarrow \text{Apply Edge} \rightarrow \text{Gather} \rightarrow \text{Apply Vertex}\]

\[\text{\nabla Gather} \rightarrow \text{\nabla Apply Edge} \rightarrow \text{\nabla Scatter} \rightarrow \text{\nabla Apply Vertex}\]

\[\text{Layer 1 backward}\]

\[\text{Layer 2, …, Layer L backward}\]

Start Backprop

Serverless thread

Weight Server

Graph Server
Challenge 2: Limited Network

Network latency has high overhead
- Significantly hinders performance

Running sequentially leads to stalls
Solution: Create Pipeline of Decomposed Tasks

Time

<table>
<thead>
<tr>
<th>Scatter</th>
<th>Apply Edge</th>
<th>Gather</th>
<th>Apply Vertex</th>
</tr>
</thead>
</table>

Serverless thread

Graph Server

Time

<table>
<thead>
<tr>
<th>Scatter</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply Edge</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Gather</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply Vertex</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Chunks Moving Through Layer of Pipeline

CPU Graph

Serverless Thread Pool

Gather\(_i\)

0 - 99

Scatter\(_i\)

Gather\(_{i+1}\)

Apply Edge\(_{i-1}\)
Data Chunks Moving Through Layer of Pipeline

CPU Graph

Serverless Thread Pool

... 100 - 199
Gather$_i$

Scatter$_i$

Gather$_{i+1}$

Apply Edge$_{i-1}$

Apply Vertex$_i$ 0 - 99

Data Chunks Moving Through Layer of Pipeline
Data Chunks Moving Through Layer of Pipeline

CPU Graph Server

... 200 - 299 Gather_i 0 - 99 Scatter_i Gather_{i+1}

Serverless Thread Pool

Apply Edge_{i-1} Apply Vertex_i 100 - 199
Data Chunks Moving Through Layer of Pipeline

CPU Graph

Serverless

Thread Pool

... 300 - 399 Gather_i 100 - 199 Scatter_i Gather_{i+1}

Apply Edge_{i-1} 200 - 299 Apply Vertex_i 0 - 99 Apply Edge_i

Gather_i Scatter_i Gather_{i+1}

Apply Edge_i 200 - 299 Apply Vertex_i 0 - 99 Apply Edge_i

Data Chunks Moving Through Layer of Pipeline
Data Chunks Moving Through Layer of Pipeline
Data Chunks Moving Through Layer of Pipeline

Dependencies on neighbor data

Synchronize!
Synchronize after Scatter Hinders Pipeline

Pipeline not fully utilized
  ● Network latency challenge not resolved!

Modified Solution: Introduce asynchrony to pipeline
  ● Allow pipeline to saturate fully
Two Sync Points Makes Asynchrony Difficult

1. Sync before new epoch
   - Dependency on updated parameters

2. Sync after every Scatter
   - Dependency on neighbors' features

---

Gather → Apply Vertex → Scatter → Apply Edge → ... → Weight Update

\( \nabla \) Scatter → \( \nabla \) Apply Vertex → \( \nabla \) Gather → \( \nabla \) Apply Edge → \( \nabla \) Scatter

Start Backprop
Minimizing Effects of Asynchrony on Convergence

Bounded staleness (graph-parallel path)
- No chunk in the system can get $S$ epochs ahead of others
  - $S$ is some staleness bound

Weight stashing at weight servers\(^2\) (tensor-parallel path)
- Cache parameters used in forward to use same version in backward

We have formally proved the convergence of our system

Serverless Optimizations

- Task fusion
- Tensor rematerialization
- Lambda internal streaming

Details in the paper
## Data Graphs

| Graph       | Size (|V|, |E|)         | # features | # labels | Avg. Degree |
|-------------|--------------|------------|----------|------------|
| Reddit-small| (232.9K, 114.8M) | 602        | 41       | 492.9      |
| Reddit-large| (1.1M, 1.3B)  | 301        | 50       | 645.4      |
| Amazon      | (9.2M, 313.9M) | 300        | 25       | 35.1       |
| Friendster  | (65.6M, 3.6B)  | 32         | 50       | 27.5       |

### Target metrics:
- Performance
- Cost
- **Value:** Performance-per-dollar
We Evaluated Several Aspects of Dorylus

Compared staleness bounds to determine optimal asynchrony

Evaluated Dorylus variants without serverless

- CPU-only: All stages run on CPUs
- GPU-only: All stages run on GPUs

Compared against existing systems

Effects of scaling out

Breakdown of time/costs per stage
We Evaluated Several Aspects of Dorylus

Compared staleness bounds to determine optimal asynchrony

Evaluated Dorylus variants without serverless

- CPU-only: All stages run on CPUs
- GPU-only: All stages run on GPUs

Compared against existing systems

Effects of scaling out

Breakdown of time/costs per stage
High Value on Large-Sparse Graphs

Dorylus provides better value than CPU and GPU-based backends on large sparse graphs.

Dorylus outperforms GPU based implementations on very large graphs.
Dorylus Outperforms Existing Systems

- Dorylus outperforms sampling based methods
  - **3.25x** faster than DGL (sampling)
- Slower than GPU-based non-sampling systems
  - Whole graph can fit in GPU memory
Dorylus Scales Full Graph Training

On a large, sparse graph

- Dorylus 1.99x faster than DGL (sampling)
- Only 1.37x slower than Dorylus (GPU only)

Value comparison:

- 17.7x value of DGL (sampling)
- 8.6x value of AliGraph
Conclusion: Dorylus Provides Value

Dorylus: Affordably scaling Graph Neural Network training to billion-edge graphs

- Utilize computation separation to specialize resources
- Implement bounded asynchronous pipeline
- Up to 2.75x more performance-per-dollar than CPU-only, 4.83x GPU-only
- Opens possibility to apply our techniques to other models

Thank you! Code at https://github.com/uclasystem/dorylus. For questions email jothor@cs.ucla.edu