marius

Learning Massive Graph Embeddings on a Single Machine

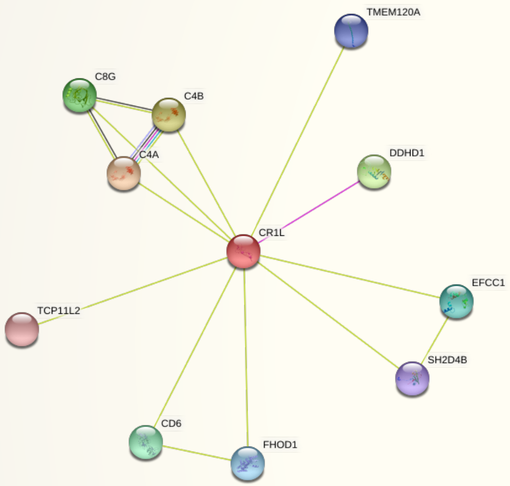
Jason Mohoney, Roger Waleffe, Yiheng Xu,
Theodoros Rekatsinas, Shivaram Venkataraman

University of Wisconsin-Madison

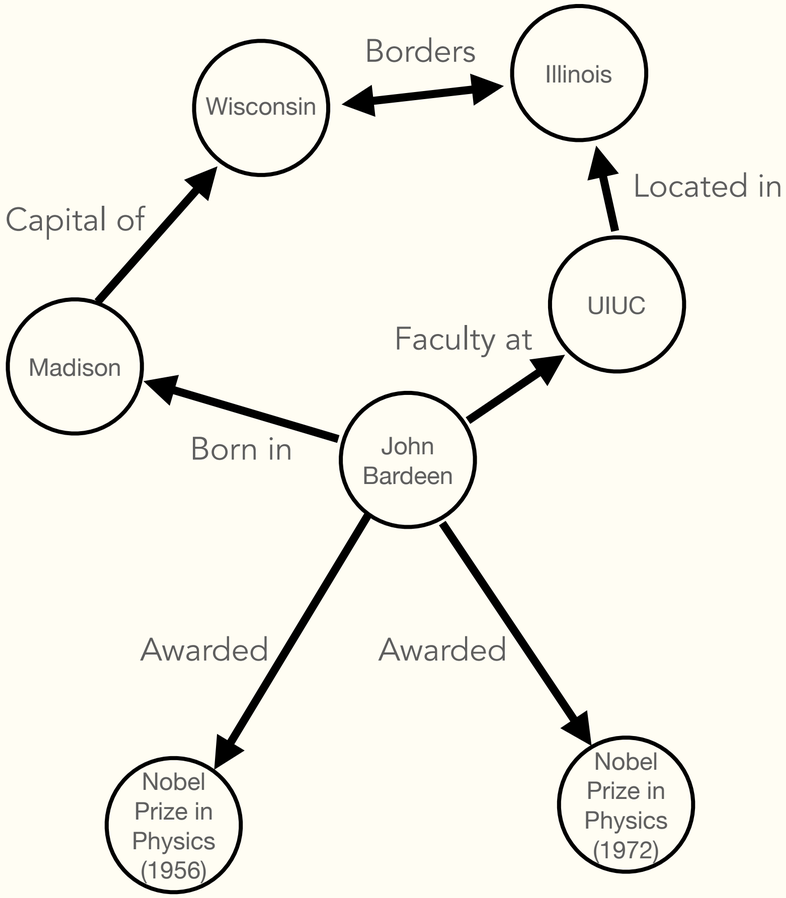
marius

marius-project.org

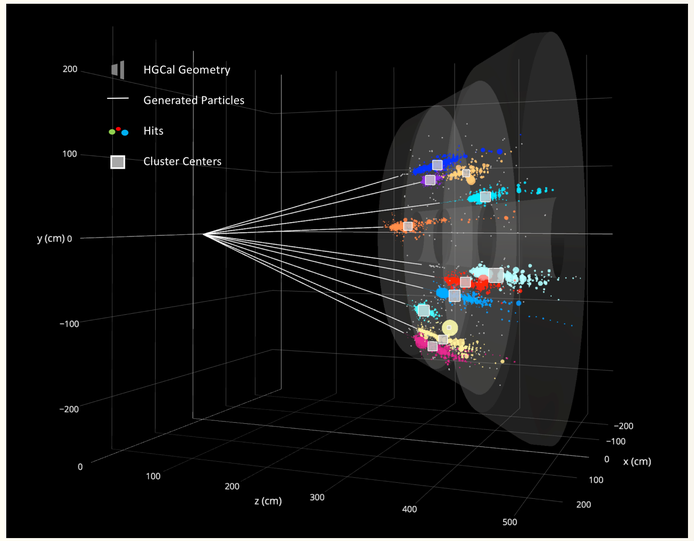
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON



Biochemistry



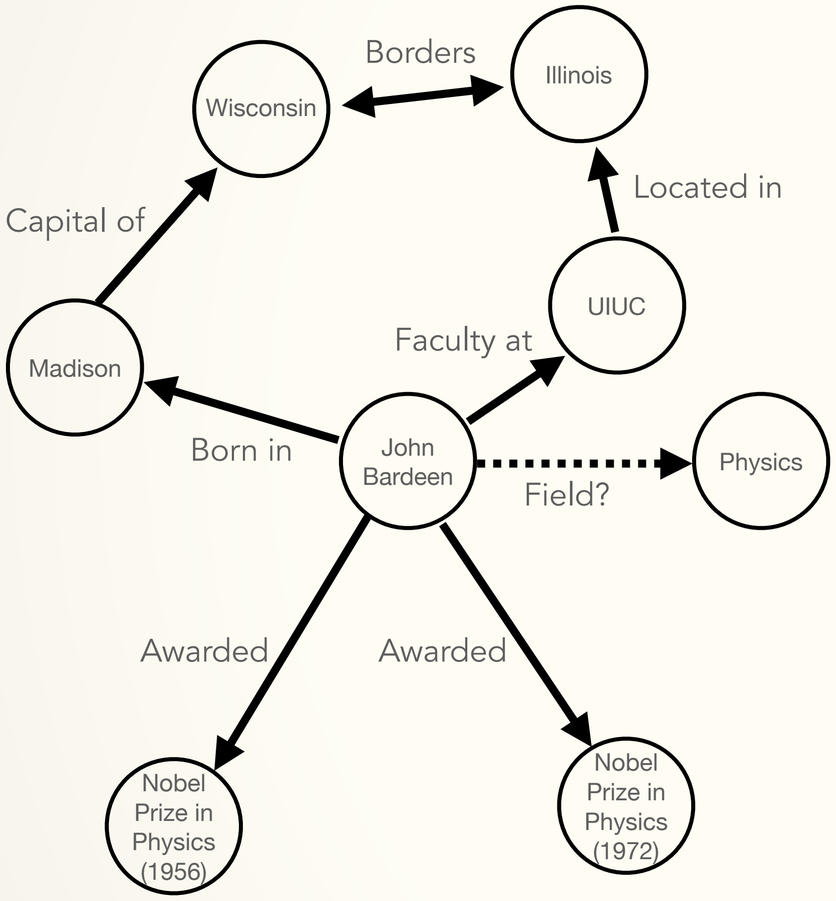
Knowledge Graphs



High-Energy Physics

Graphs are **universal representations** of rich semantics about entities (nodes) and their relationships (edges)

Graph Embeddings



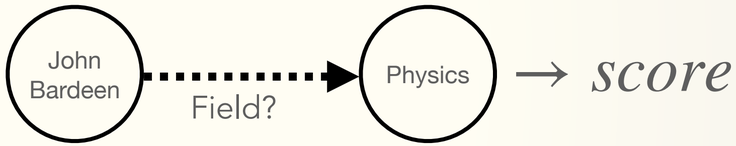
Subgraph of the Freebase knowledge graph

Objective: Apply modern ML on graphs

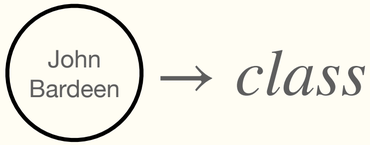
Transform node and edge-types into embeddings (vectors)

<u>Example Tasks:</u>

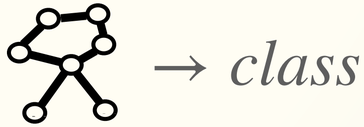
- Link Prediction (focus of this work)



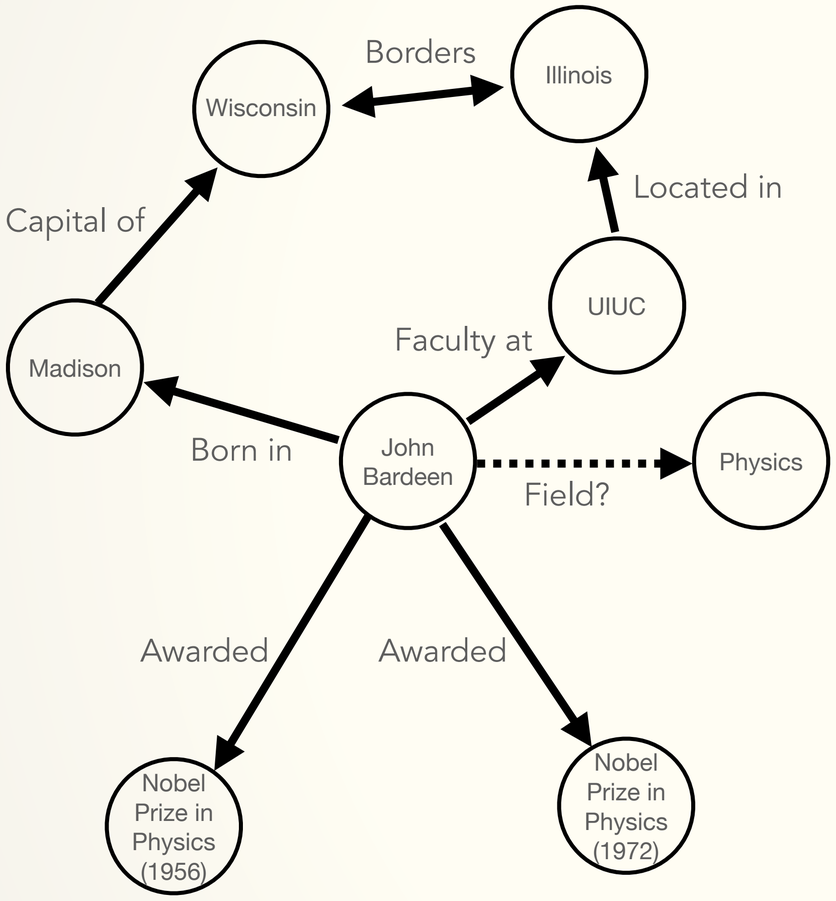
- Node classification



- Graph classification



Graph Embeddings



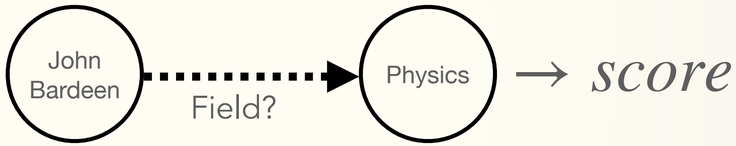
Subgraph of the Freebase knowledge graph

Objective: Apply modern ML on graphs

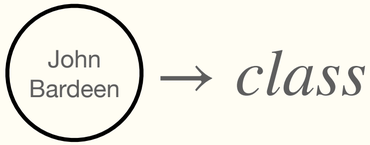
Transform node and edge-types into embeddings (vectors)

<u>Example Tasks:</u>

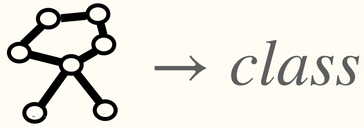
- Link Prediction (focus of this work)



- Node classification



- Graph classification

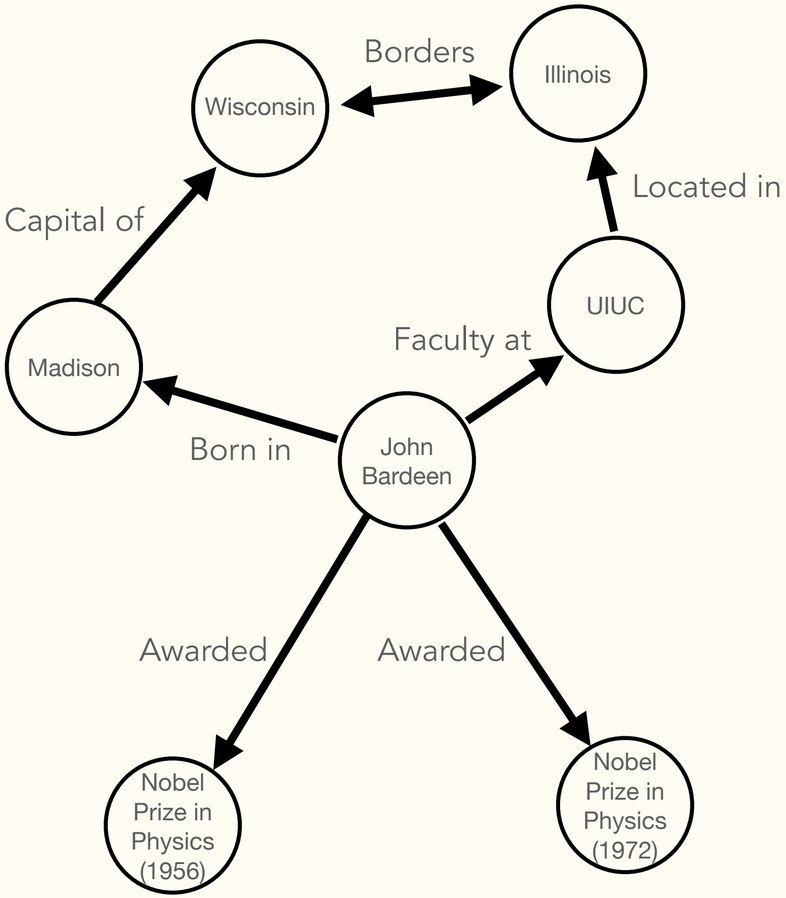


Learning Graph Embeddings

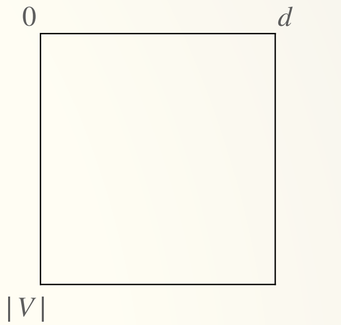
Training requires iterating over all edges and retrieving/updating embedding vectors

Training Process

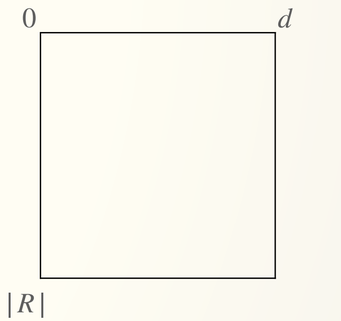
```
// E ordered randomly
for (s, r, d) in E:

    // compute loss of model for an edge
    computeLoss(s, r, d)

    // apply updates to embeddings of edge
    update(s, r, d)
```



$$G = (V, R, E)$$



Node embeddings Table



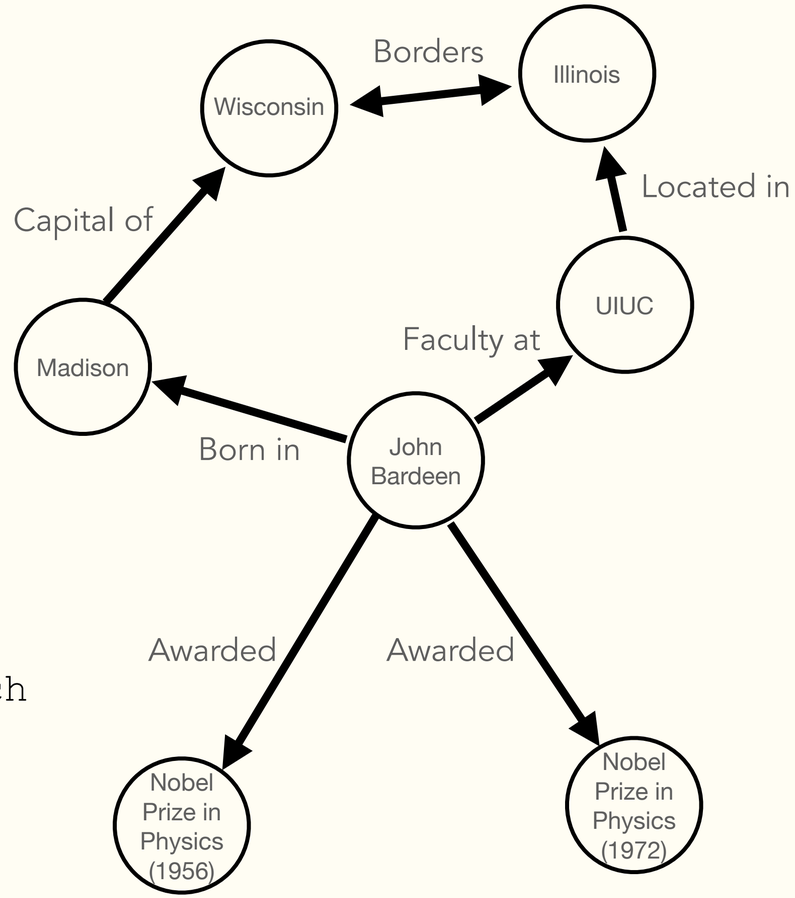
Edge-type embeddings

Learning Graph Embeddings

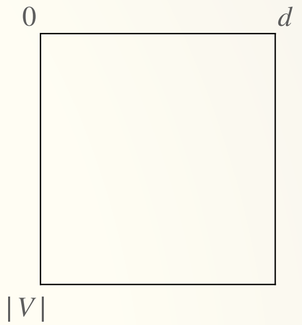
Training requires iterating over all edges and retrieving/updating embedding vectors

Batched Training

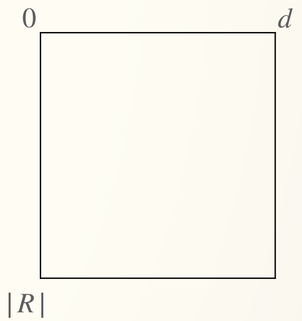
```
// E randomly grouped into batches
for batch in E:

    // compute loss of model for a batch
    computeLoss(batch)

    // apply updates to embeddings in a batch
    update(batch)
```



$G = (V, R, E)$



Node embeddings Table



Edge-type embeddings

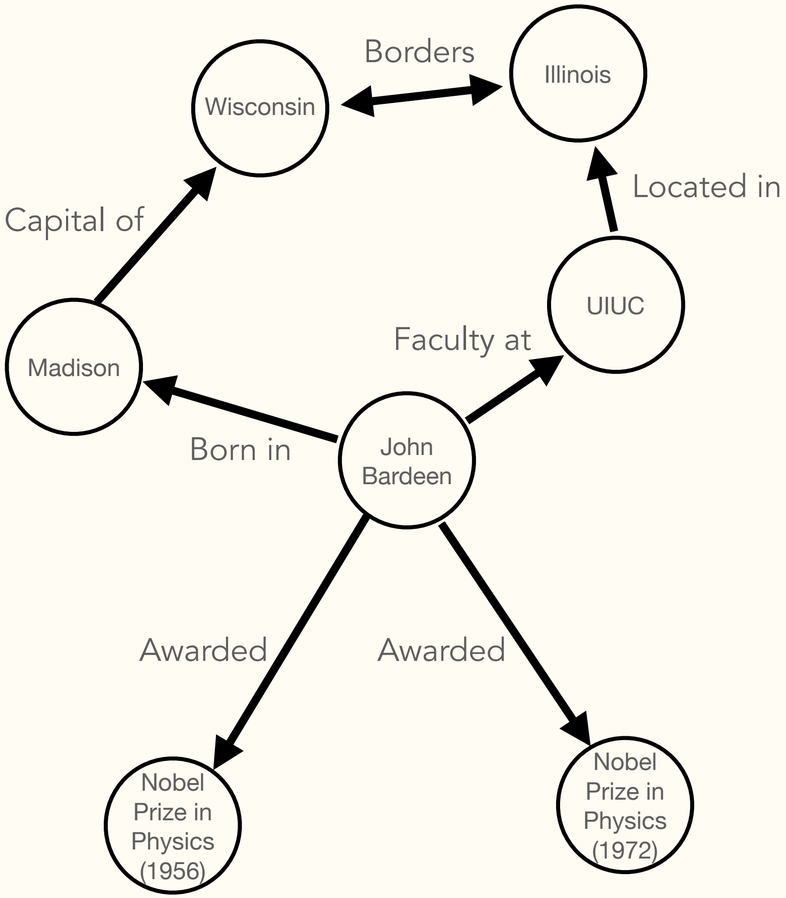
Learning Graph Embeddings

Training requires iterating over all edges and retrieving/updating embedding vectors

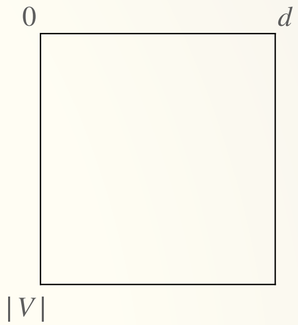
Batched Training: single iteration

```
batch = [(JB, Born, M),(M, Capital, W)]
```

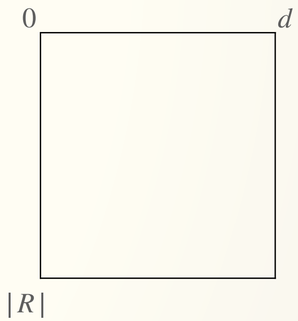
```
// load embeddings
computeLoss(batch)

// update embeddings
update(batch)
```



$$G = (V, R, E)$$



Node embeddings Table



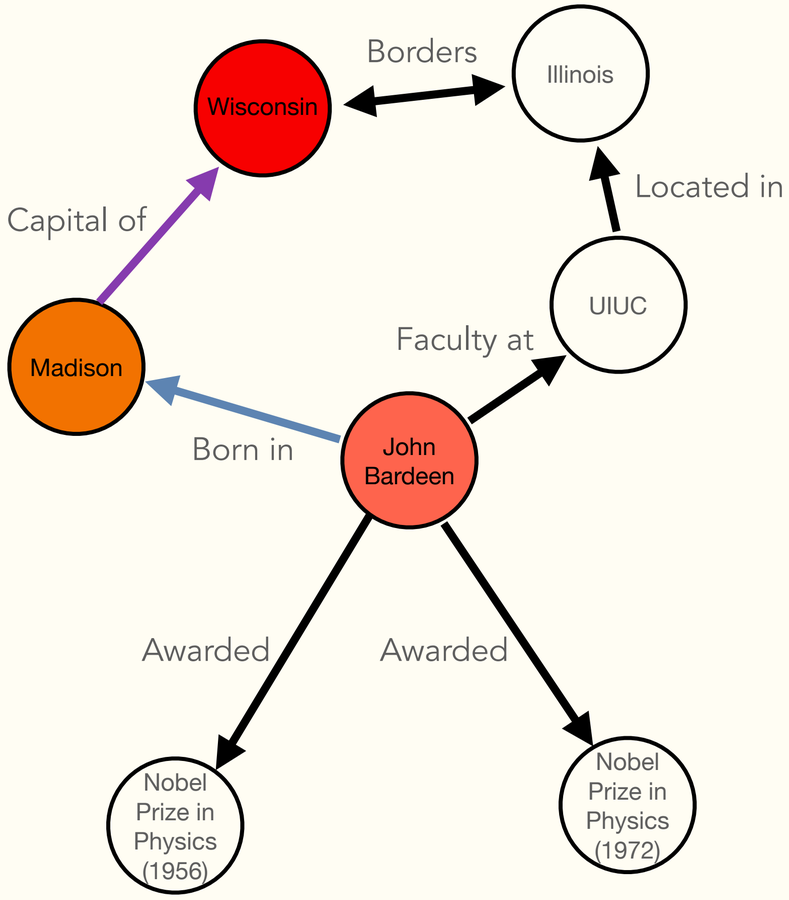
Edge-type embeddings

Learning Graph Embeddings

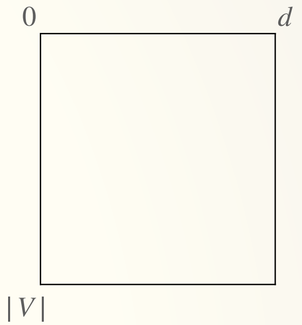
Training requires iterating over all edges and retrieving/updating embedding vectors

Batched Training: single iteration

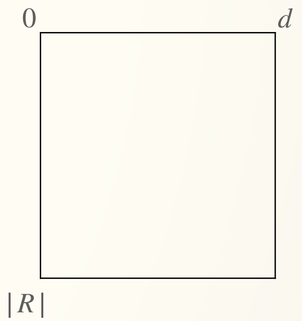
```
batch = [(JB, Born, M),(M, Capital, W)]

// load embeddings and compute loss
computeLoss(batch)

// update embeddings
update(batch)
```



Graph with batch highlighted



Node embeddings Table



Edge-type embeddings

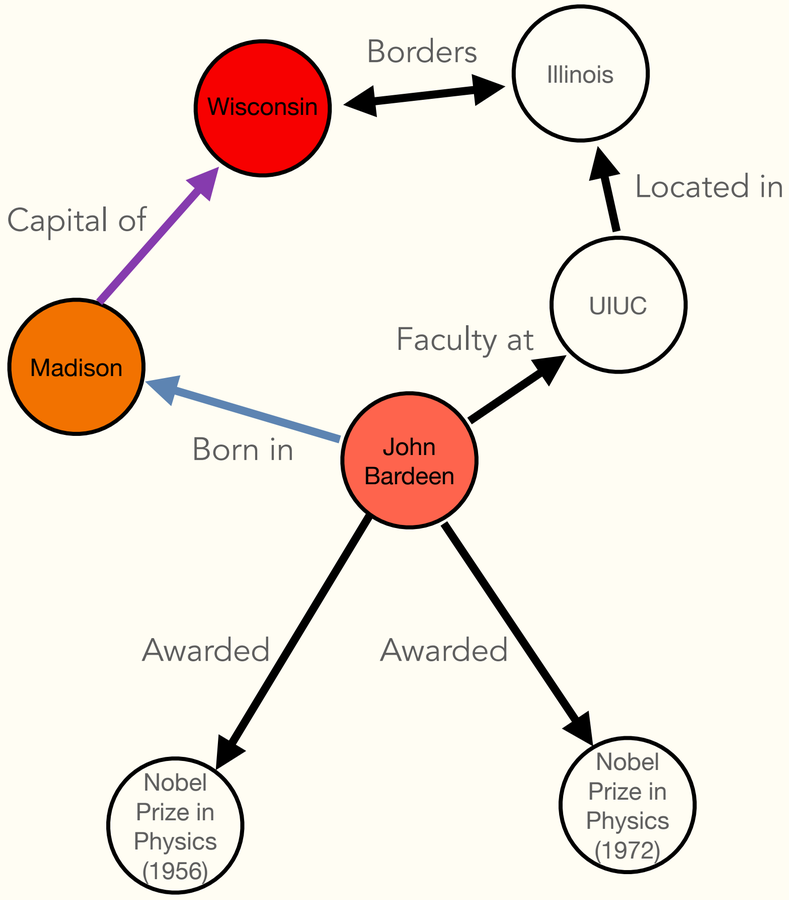
Learning Graph Embeddings

Training requires iterating over all edges and retrieving/updating embedding vectors

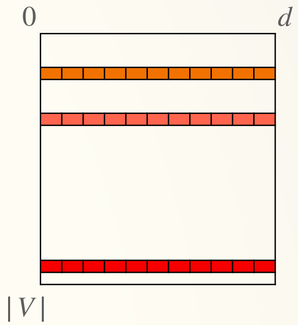
Batched Training: single iteration

```
batch = [(JB, Born, M),(M, Capital, W)]
```

```
// load embeddings and compute loss
computeLoss(batch)

// update embeddings
update(batch)
```

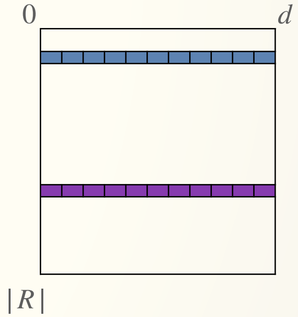


Graph with batch highlighted



Read

Node embeddings Table



Read

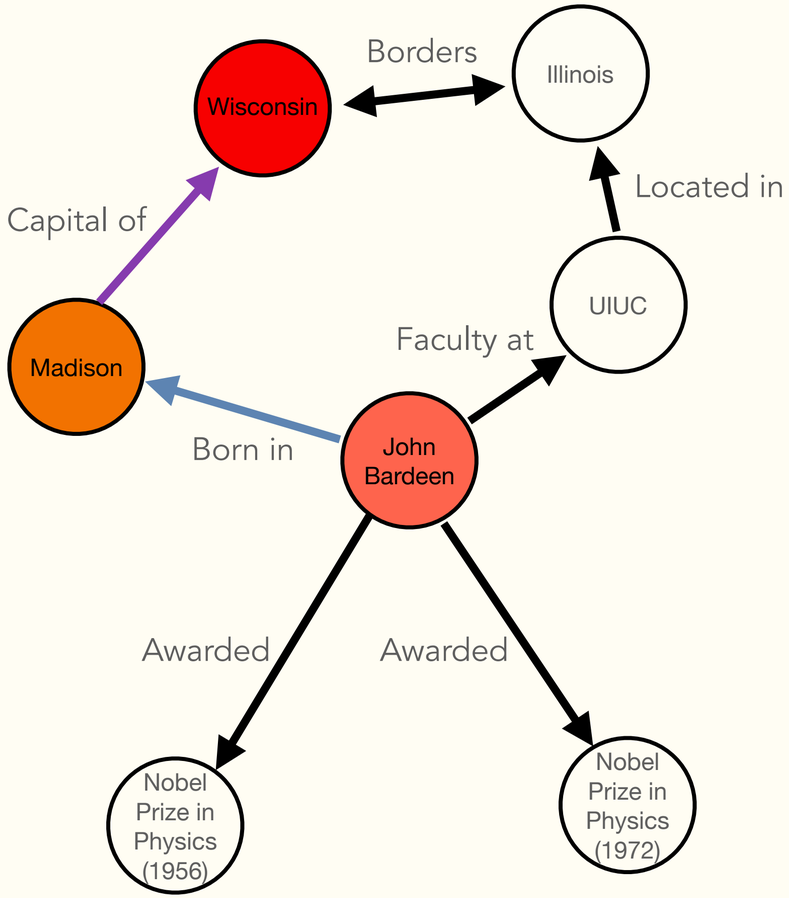
Edge-type embeddings

Learning Graph Embeddings

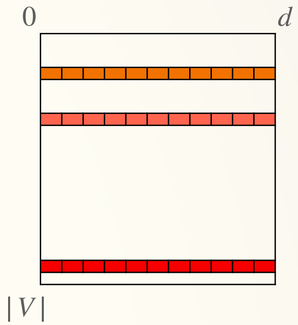
Training requires iterating over all edges and retrieving/updating embedding vectors

Batched Training: single iteration

```
batch = [(JB, Born, M),(M, Capital, W)]

// load embeddings and compute loss
computeLoss(batch)

// update embeddings
update(batch)
```

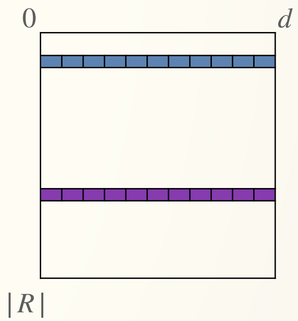


Graph with batch highlighted



Node embeddings Table

Write



Edge-type embeddings

Write

Training requires efficient access to embedding parameters

Irregular Access

Key Challenge: Data Movement

Large Datasets

Freebase86m:

- 338 million edges, 86 million nodes, 15,000 edge types
- Size of node embedding table for d = 400:

86 million x 400 x 4 bytes = 138 GB

AWS P3.2xLarge instance:

- 16 GB GPU Memory
- 61 GB CPU Memory

Embedding table unable to fit in GPU memory!

Moving embeddings to compute

How to scale?

1. Store embeddings in CPU memory and transfer to GPU(s)
   - Bottlenecked by transfer overheads
   - Limited scalability

   DGL-KE

2. Partition node embeddings and store on disk
   - Limited by disk throughput

   PyTorch Big-Graph (PBG)

3. Distribute embeddings across multiple machines
   - Bottlenecked by transfer overheads
   - Expensive

   PBG & DGL-KE

Can the data movement bottlenecks be mitigated?

Scaling to Large Graphs: marius

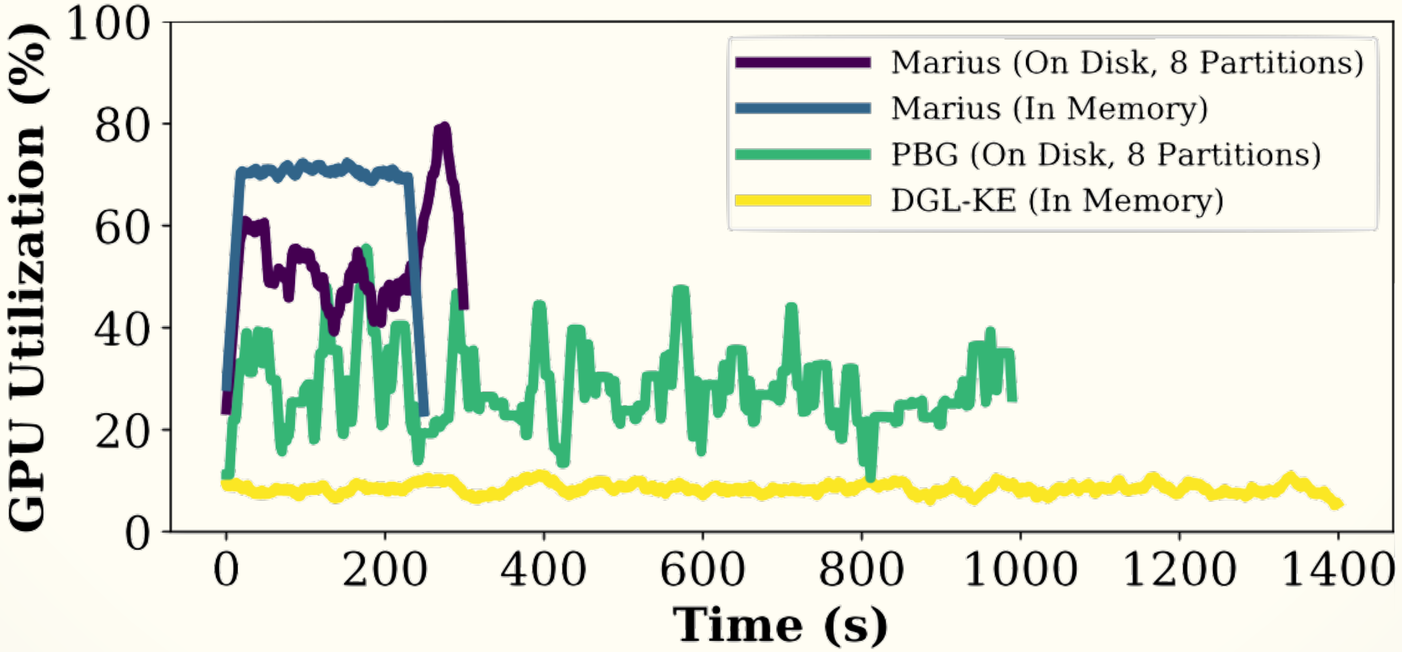
Design Goal: Eliminate data movement overheads inherent in graph embedding training

Method

- Use pipelining and async IO to hide data movement
- Utilize the full memory hierarchy with a partition buffer
- **Minimize IO with Buffer-aware Edge Traversal Algorithm (BETA)**

Results

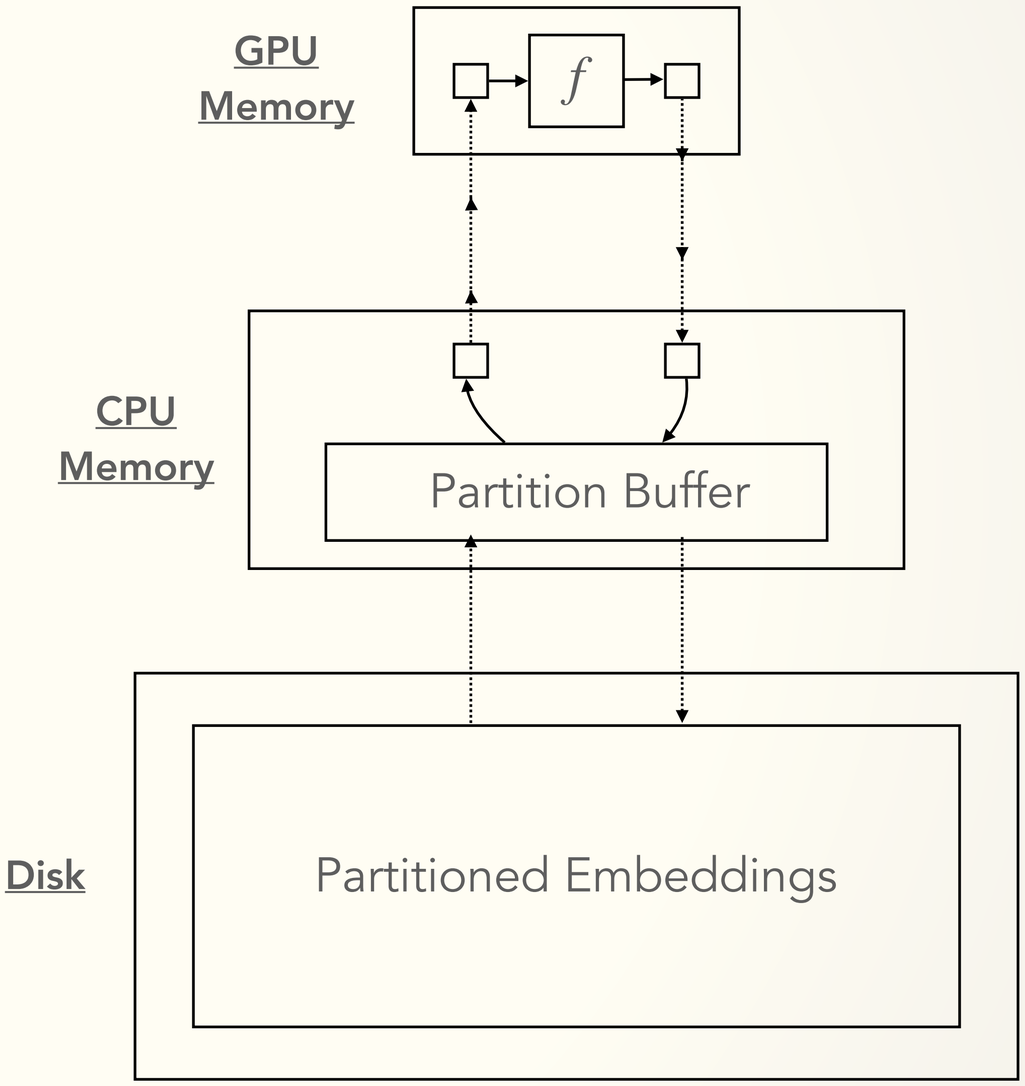
- 10x reduction in runtime vs. DGL-KE on Twitter
- 3.7x runtime reduction vs. PBG on Freebase86m
- 2x higher utilization than PBG, 6-8x higher utilization than DGL-KE



Scaling to Large Graphs: marius

Method

- Use pipelining and async IO hide data movement
- Utilize the full memory hierarchy with a partition buffer
- Minimize IO with Buffer-aware Edge Traversal Algorithm (BETA)



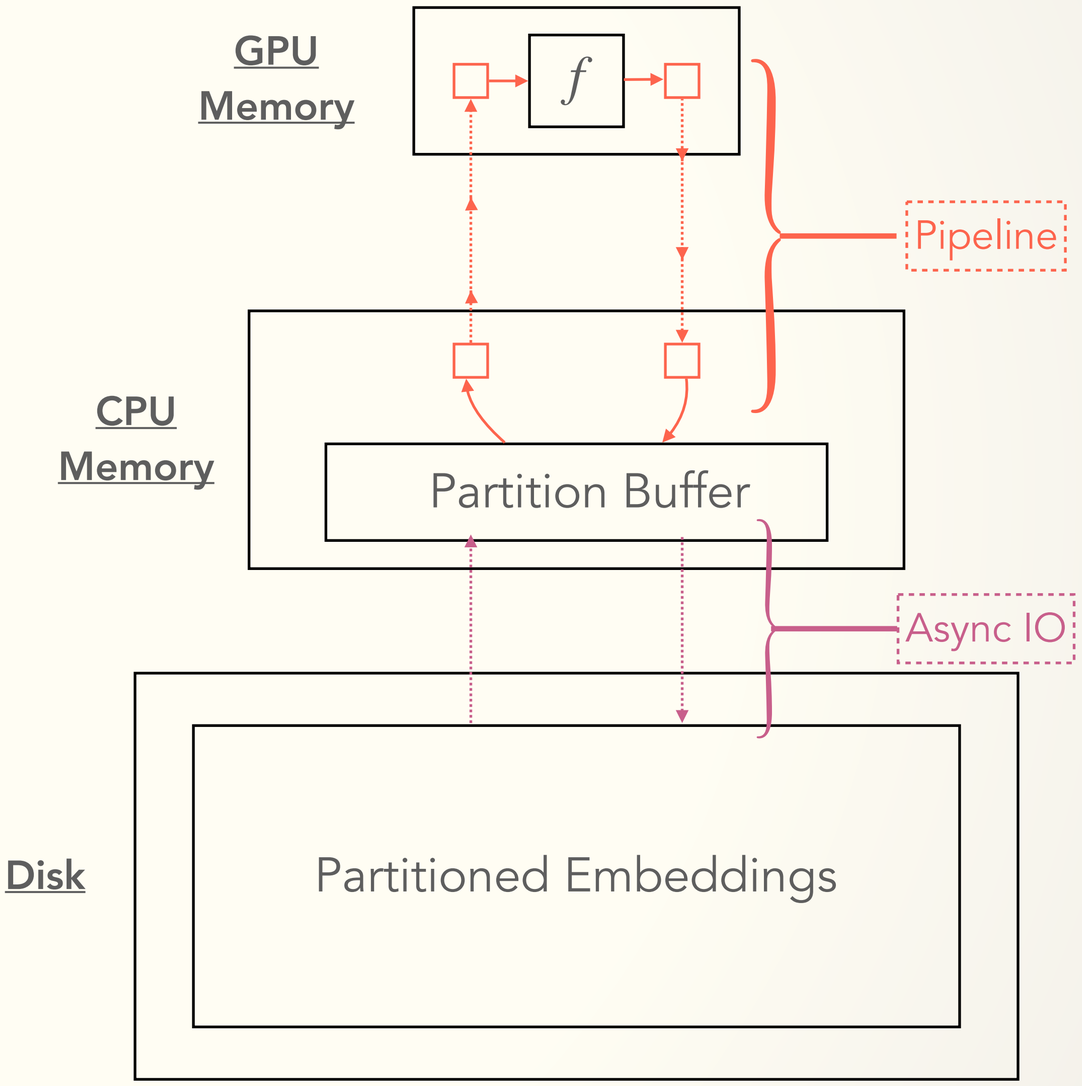
Marius Architecture

Scaling to Large Graphs: marius

Method

- **Use pipelining and async IO hide data movement**
- Utilize the full memory hierarchy with a partition buffer
- Minimize IO with Buffer-aware Edge Traversal Algorithm (BETA)

Maximize GPU utilization



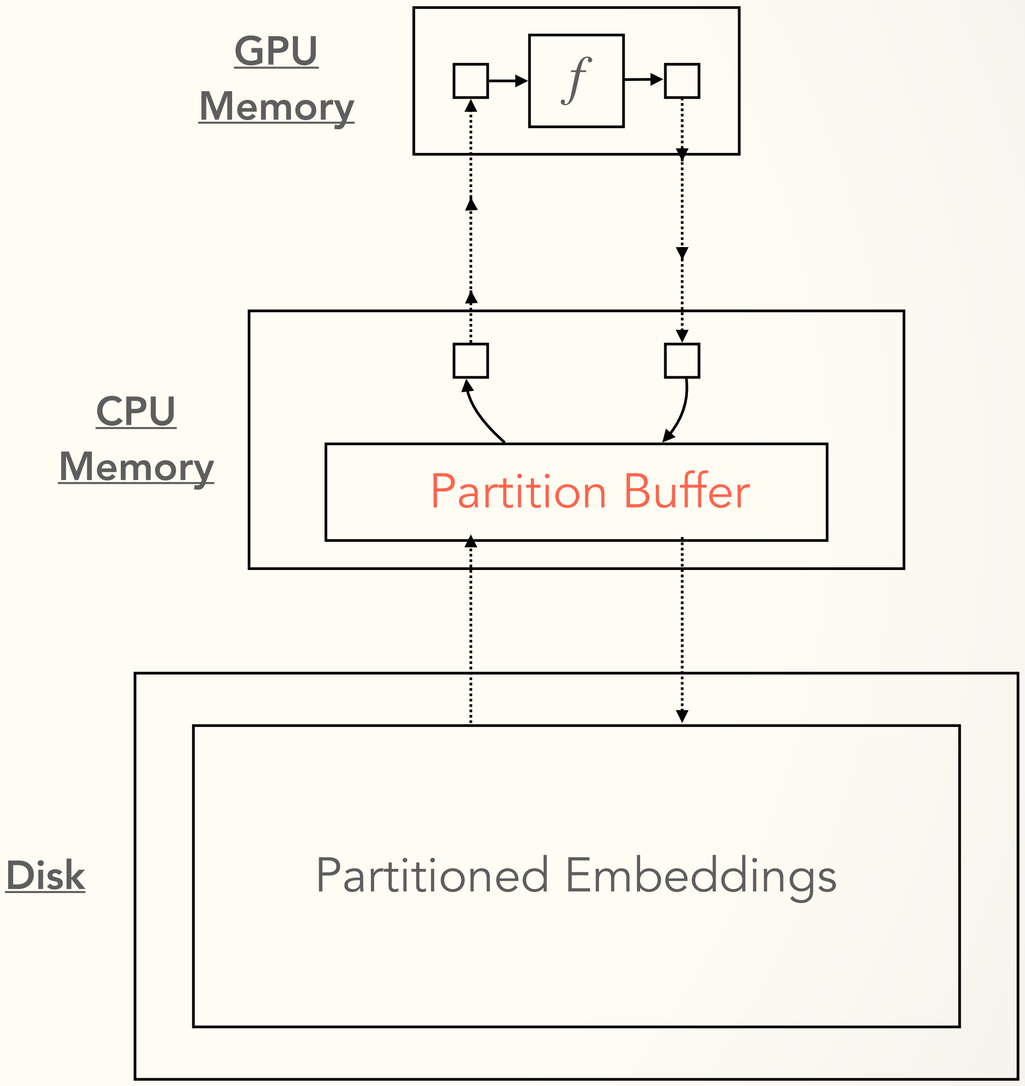
Marius Architecture

Scaling to Large Graphs: marius

Method

- Use pipelining and async IO hide data movement
- **Utilize the full memory hierarchy with a partition buffer**
- Minimize IO with Buffer-aware Edge Traversal Algorithm (BETA)

Minimize IO through partition caching



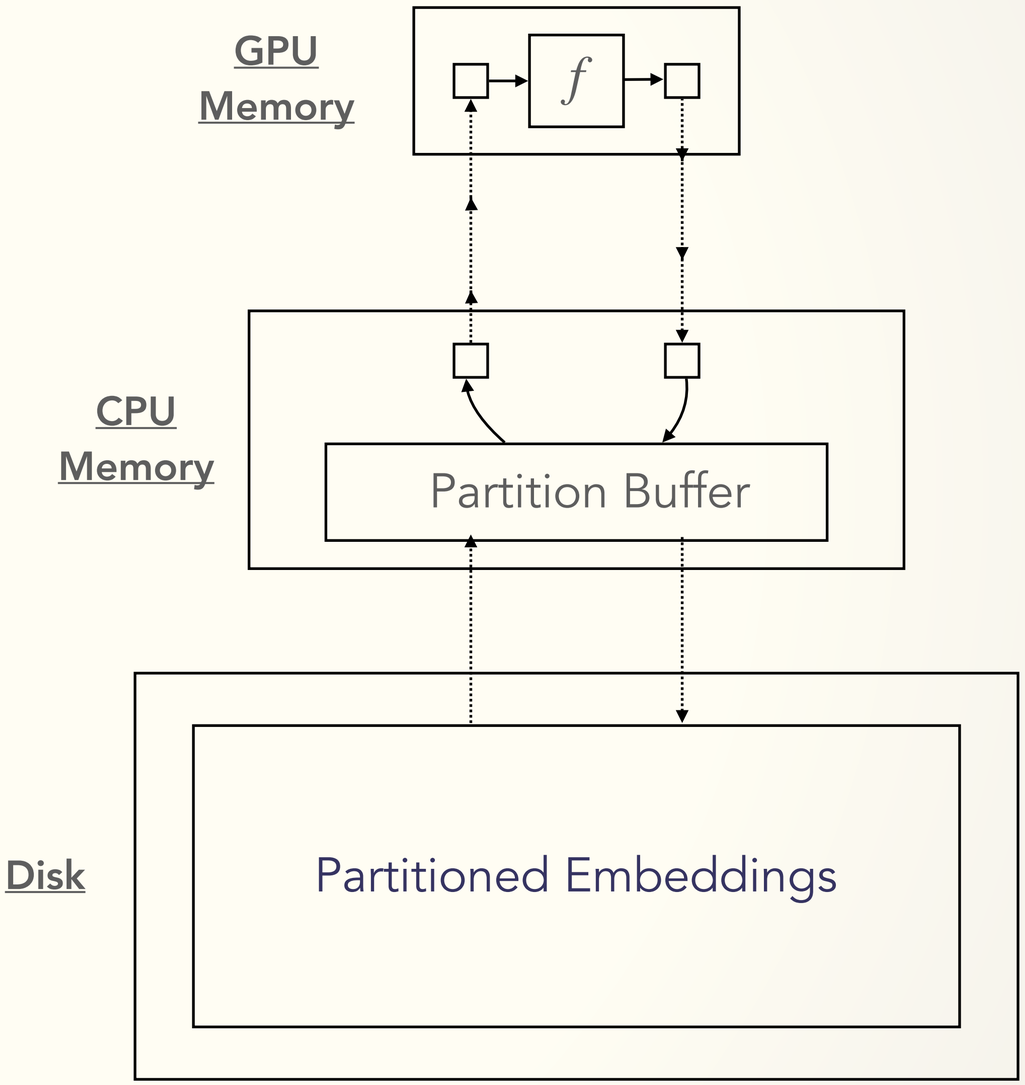
Marius Architecture

Scaling to Large Graphs: marius

Method

- Use pipelining and async IO hide data movement
- Utilize the full memory hierarchy with a partition buffer
- **Minimize IO with Buffer-aware Edge Traversal Algorithm (BETA)**

Minimize IO to lower bound

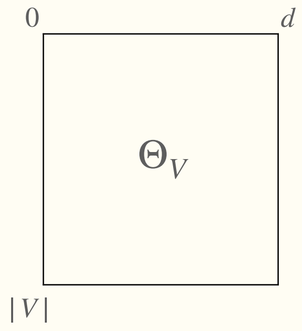


Marius Architecture

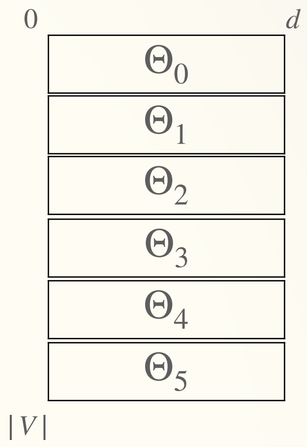
Processing Partitions

Node Embedding Partitions

Node embeddings are partitioned uniformly into p disjoint partitions.



Node embedding table

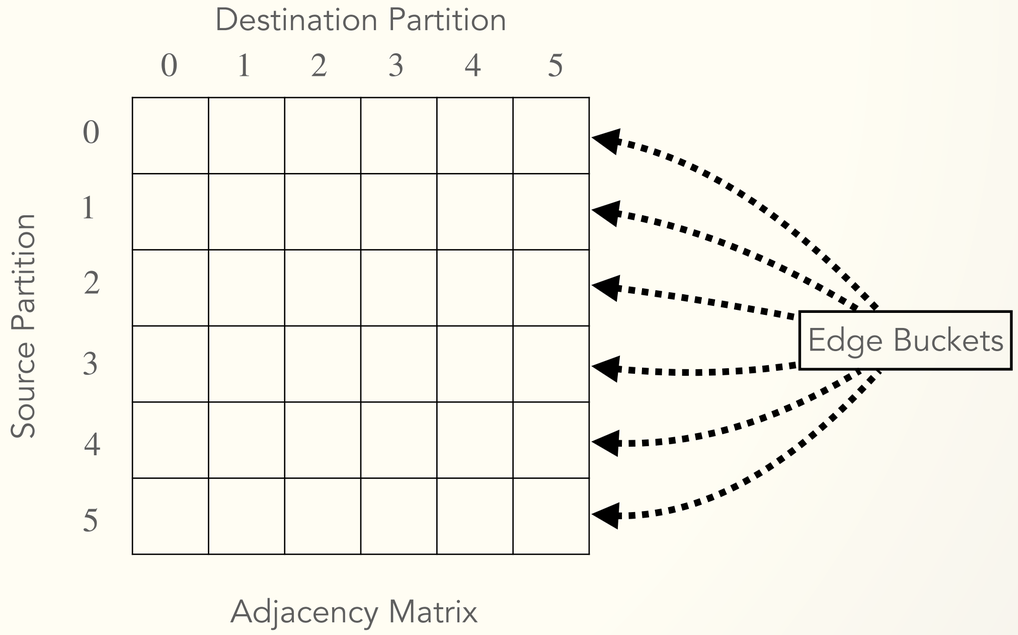


Partitioned node embedding table (p = 6)

Edge Buckets

Edge bucket (i,j) contains all edges with a source in partition i and a destination in partition j

To iterate over all edges, we need to iterate over all edge buckets



Edge bucket orderings and IO

The order in which edge buckets are processed has an impact on IO

Size of partitions: 138 GB / 6 = 23 GB

23 GB / 400 MBps = ~57 seconds

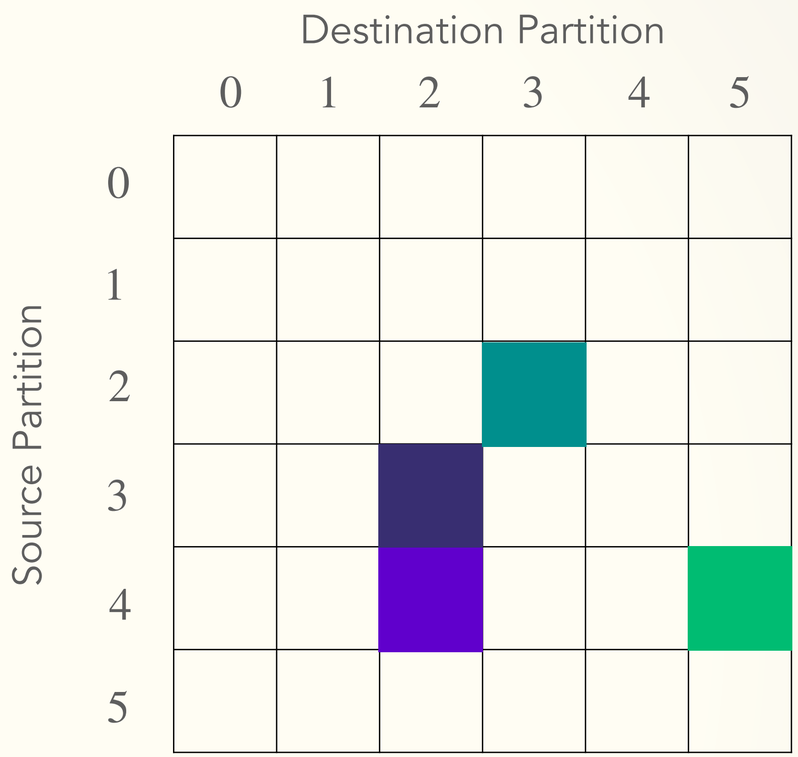
Costly swaps!

Example: After processing edge bucket (3, 2)

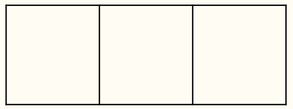
Processing (2, 3): Requires no extra swaps

Processing (2, 4): Requires one swap

Processing (4, 5): Requires two swaps



Partitions in Buffer



$c = 3$

Partitions on disk

$p = 6$

Edge bucket orderings and IO

A Lower Bound

Can never process more than 2c - 1 edge buckets per swap

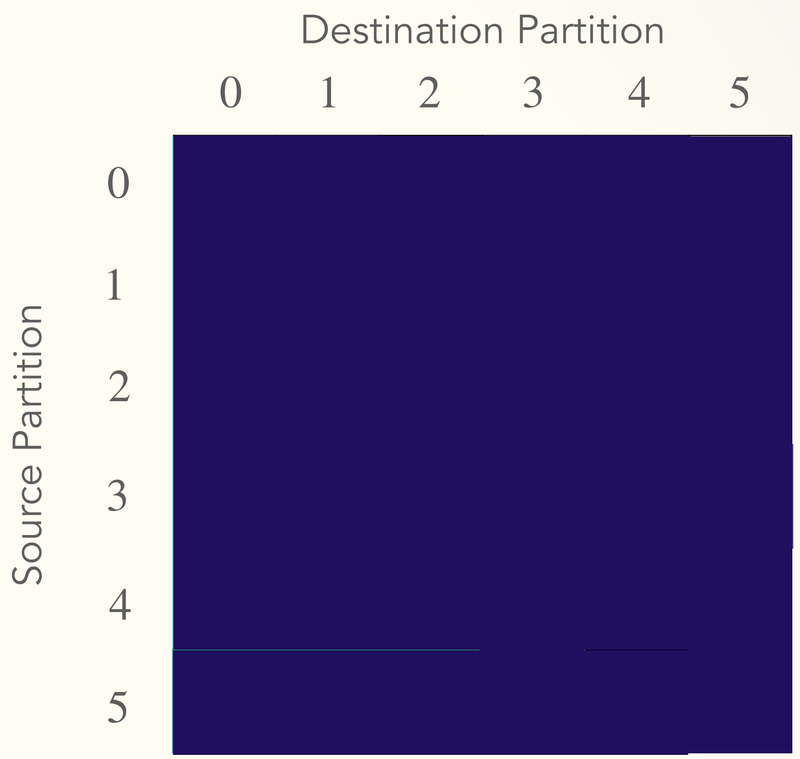
$$\lceil \frac{p^2 - c^2}{2c - 1} \rceil = \lceil \frac{6^2 - 3^2}{2 * 3 - 1} \rceil = 6$$

6 swaps

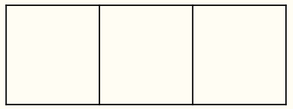
Random Ordering ~23 swaps

Hilbert Curve Ordering 12 swaps

BETA Ordering 7 swaps



Partitions in Buffer



$c = 3$

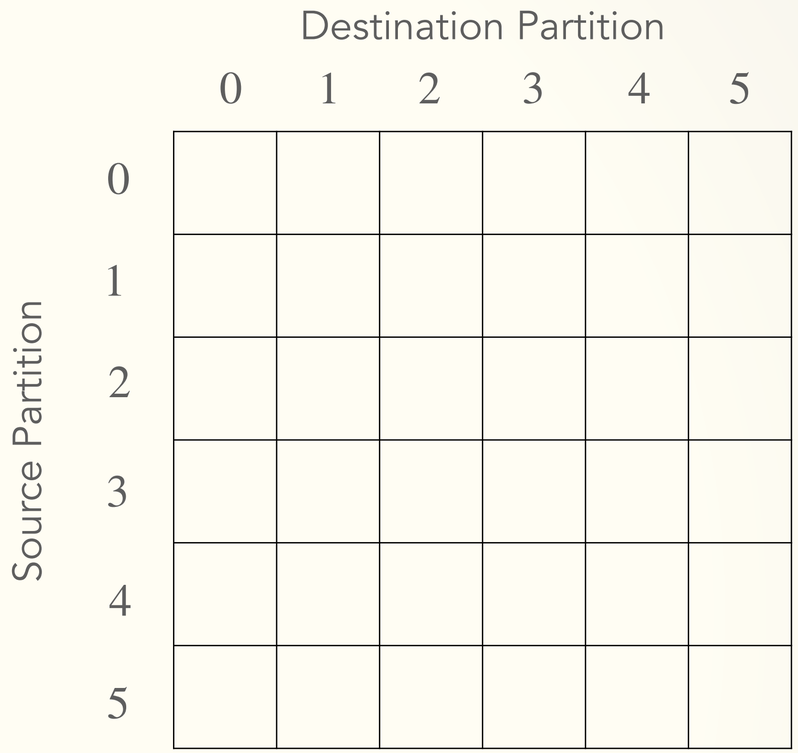
Partitions on disk

$p = 6$

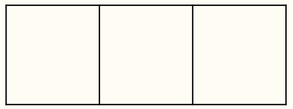
Buffer-aware Edge Traversal Algorithm (BETA)

BETA Ordering

1. Randomly initialize buffer
2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets
3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed



Partitions in Buffer



$c = 3$

Partitions on disk

Θ_0	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5

$p = 6$

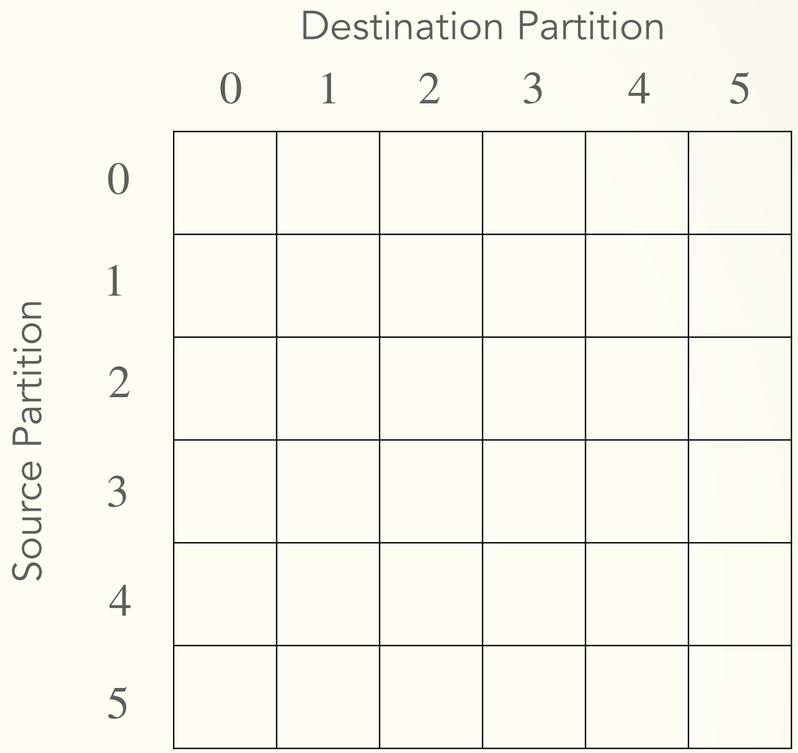
Buffer-aware Edge Traversal Algorithm (BETA)

BETA Ordering

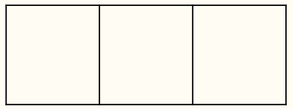
1. **Randomly initialize buffer**

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed



Partitions in Buffer



$c = 3$

Partitions on disk

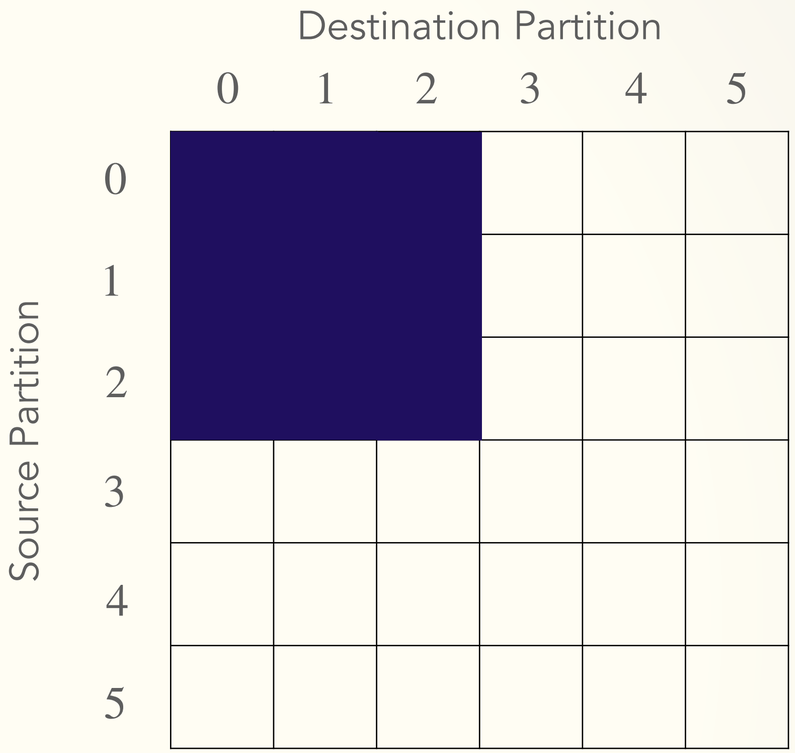
$p = 6$

Buffer-aware Edge Traversal Algorithm (BETA)

BETA Ordering

1. Randomly initialize buffer
2. **Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets**
3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed

0 swaps*



Partitions in Buffer

Θ_0	Θ_1	Θ_2

$c = 3$

Partitions on disk

Θ_0	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5

$p = 6$

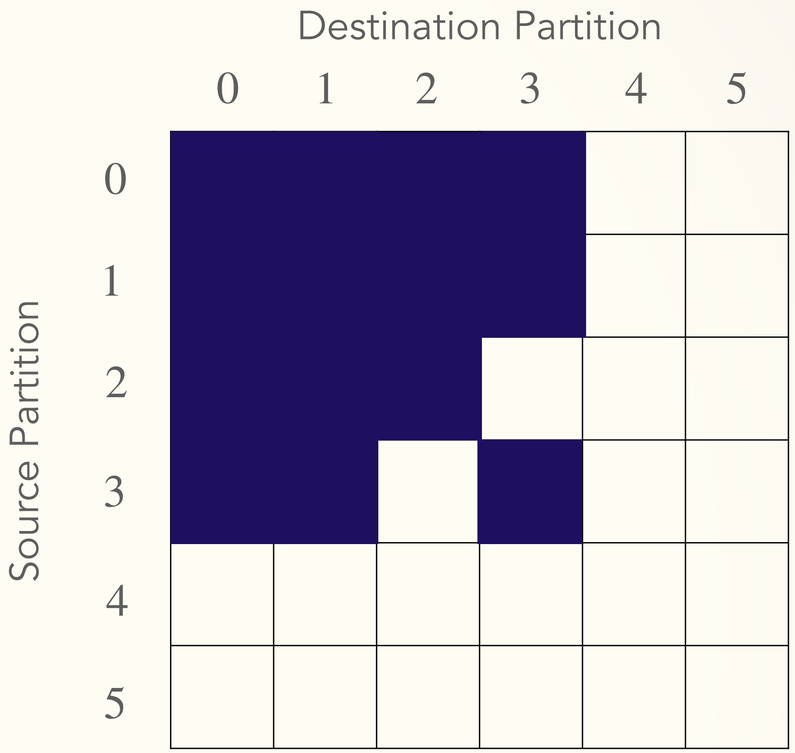
* Not counting initialized buffer, as with the previous orderings

Buffer-aware Edge Traversal Algorithm (BETA)

BETA Ordering

1. Randomly initialize buffer
2. **Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets**
3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed

0 swaps*



Partitions in Buffer

Θ_0	Θ_1	Θ_2

$c = 3$

Partitions on disk

Θ_0	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5

$p = 6$

* Not counting initialized buffer, as with the previous orderings

Buffer-aware Edge Traversal Algorithm (BETA)

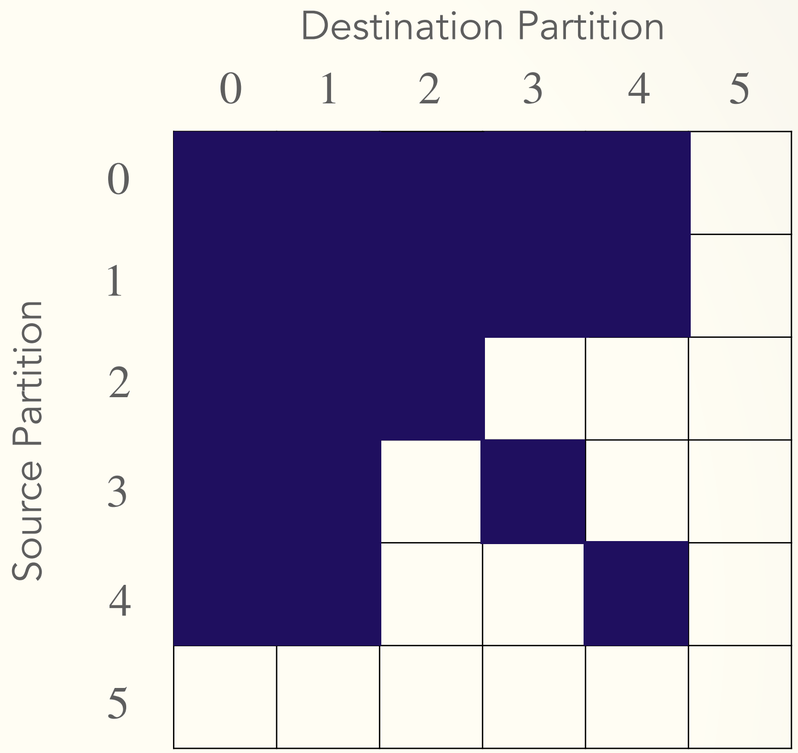
BETA Ordering

1. Randomly initialize buffer

2. **Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets**

3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed

2 swaps



Partitions in Buffer

Θ_0	Θ_1	Θ_3

$c = 3$

Partitions on disk

Θ_0	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5

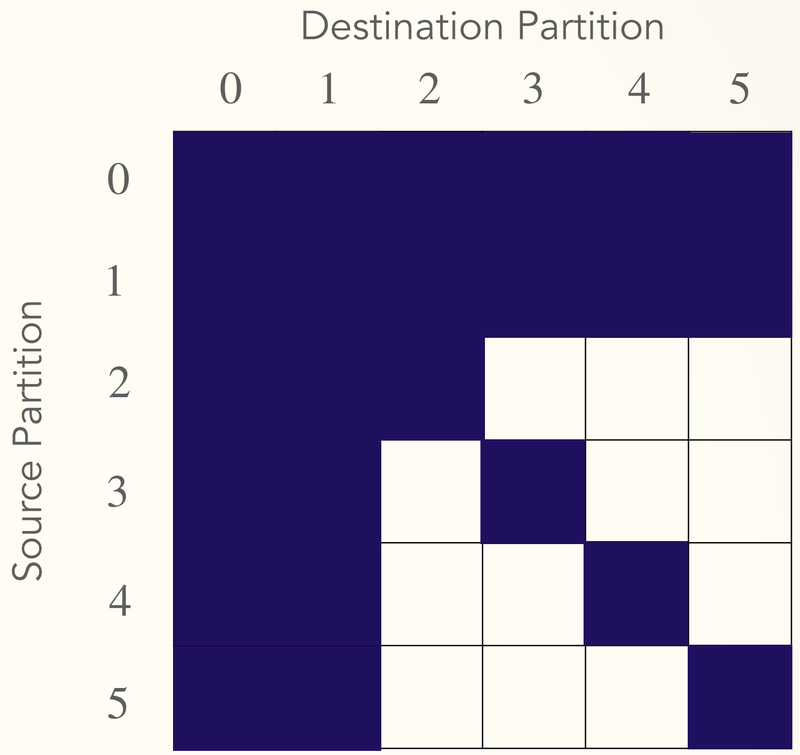
$p = 6$

Buffer-aware Edge Traversal Algorithm (BETA)

BETA Ordering

1. Randomly initialize buffer
2. **Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets**
3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed

3 swaps



Partitions in Buffer

Θ_0	Θ_1	Θ_4

$c = 3$

Partitions on disk

Θ_0	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5

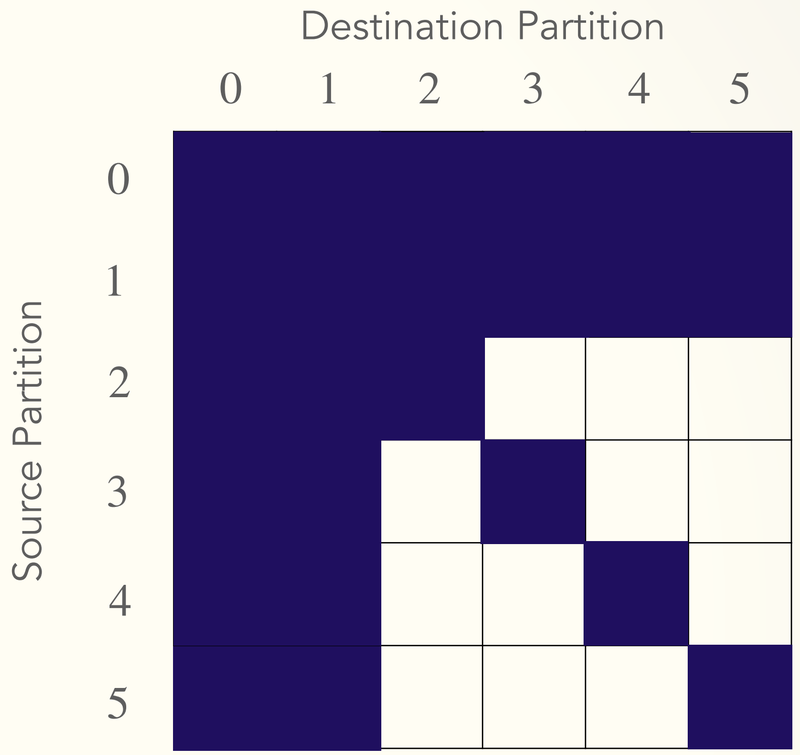
$p = 6$

Buffer-aware Edge Traversal Algorithm (BETA)

BETA Ordering

1. Randomly initialize buffer
2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets
3. **Fix a new c - 1 partitions and repeat until all edge buckets have been processed**

3 swaps



Partitions in Buffer

Θ_0	Θ_1	Θ_5

$c = 3$

Partitions on disk

Θ_0	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5

$p = 6$

Buffer-aware Edge Traversal Algorithm (BETA)

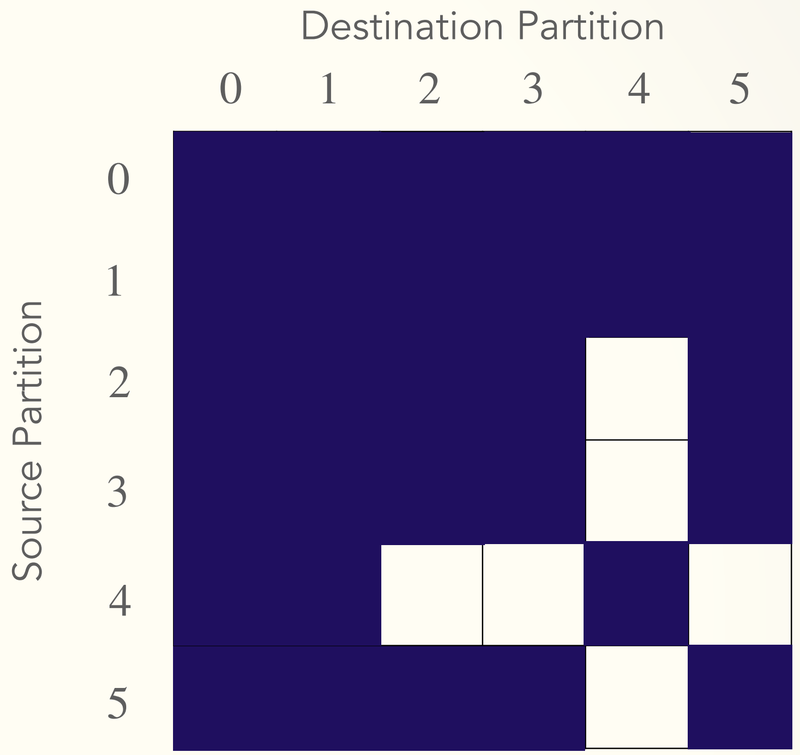
BETA Ordering

1. Randomly initialize buffer

2. **Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets**

3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed

5 swaps



Partitions in Buffer

Θ_2	Θ_3	Θ_5

$c = 3$

Partitions on disk

Θ_0	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5

$p = 6$

Buffer-aware Edge Traversal Algorithm (BETA)

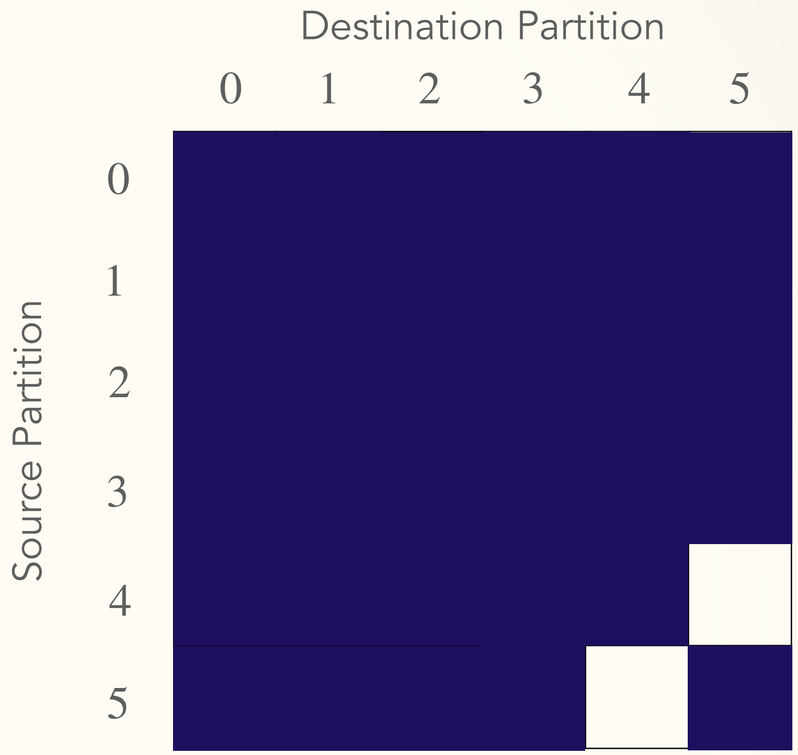
BETA Ordering

1. Randomly initialize buffer

2. **Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets**

3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed

5 swaps



Partitions in Buffer

Θ_2	Θ_3	Θ_5

$c = 3$

Partitions on disk

Θ_0	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5

$p = 6$

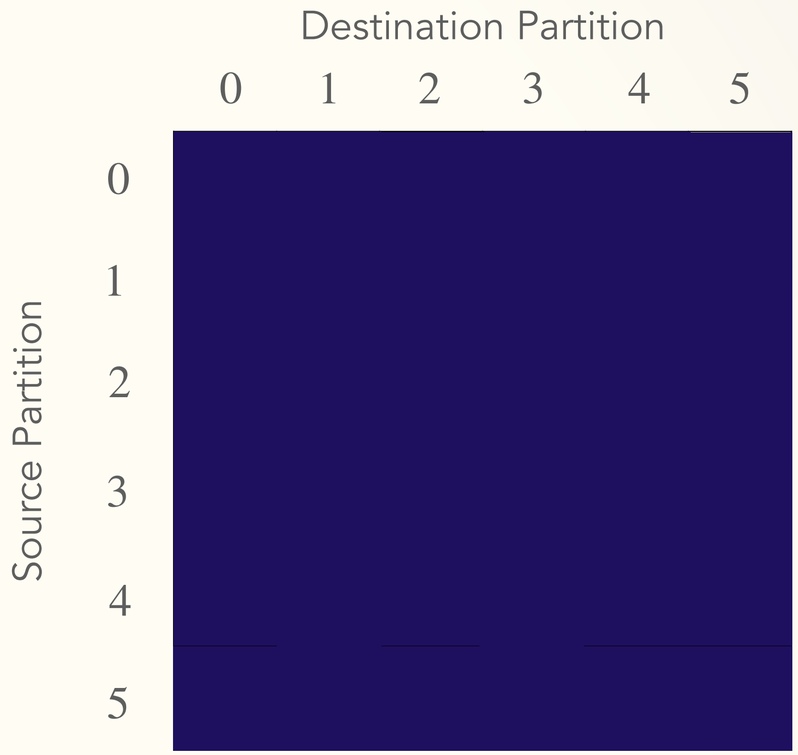
Buffer-aware Edge Traversal Algorithm (BETA)

BETA Ordering

1. Randomly initialize buffer
2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets
3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed

~~6~~ 7 swaps

Close to the 6 swap lower bound!



Partitions in Buffer

Θ_2	Θ_3	Θ_4

$c = 3$

Partitions on disk

Θ_0	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5

$p = 6$

Open sourced system: marius-project.org

Built on PyTorch

~15,000 lines of C++ and growing

Python API

PyTorch Compatible

Installation from source with Pip

1. Install latest version of PyTorch for your CUDA version:

   Linux:

   - CUDA 10.1: `python3 -m pip install torch==1.7.1+cu101 -f https://download.pytorch.org/whl/torch_stable.html`
   - CUDA 10.2: `python3 -m pip install torch==1.7.1`
   - CPU Only: `python3 -m pip install torch==1.7.1+cpu -f https://download.pytorch.org/whl/torch_stable.html`

   MacOS:

   - CPU Only: `python3 -m pip install torch==1.7.1`
2. Clone the repository `git clone https://github.com/marius-team/marius.git`
3. Build and install Marius `cd marius; python3 -m pip install .`

docker

Marius in Docker

Marius can be deployed within a docker container. Here is a sample ubuntu dockerfile (loca
`examples/docker/dockerfile`) which contains the necessary dependencies preinstalled f

Building and running the container

Build an image with the name `marius` and the tag `example` :

```
docker build -t marius:example -f examples/docker/dockerfile examples/docker
```

Marius

Search docs

CONTENTS

- Introduction
- Quick Start
- Build
- System Overview
- Configuration
- IO Format
- Training
- Models
- Loss Functions
- Evaluation
- Storage Backends

API

- Batch
- Buffer
- Config
- DataSet
- Datatypes
- Decoder
- Encoder
- Evaluator
- IO
- Logger
- Marius
- Model
- Ordering
- Pipeline
- Storage
- Trainer
- Util

» Batch View page source

Batch

class **Batch**

Contains metadata, edges and embeddings for a single batch.

Subclassed by PartitionBatch

Public Functions

Batch(bool *train*)

Constructor

~Batch()

void **localSample**()

Destructor Construct additional negative samples and neighborhood information from the batch

void **accumulateUniqueIndices**()

Populates the unique_<>_indices tensors

void **embeddingsToDevice**(int *device_id*)

Transfers embeddings, optimizer state, and indices to specified device

void **prepareBatch**()

Populates the src_pos_embeddings, dst_post_embeddings, relation_embeddings, src_neg_embeddings, and dst_neg_embeddings tensors for model computation

void **accumulateGradients**()

Accumulates gradients into the unique_node_gradients and unique_relation_gradients tensors, and applies optimizer update rule to create the unique_node_gradients2 and unique_relation_gradients2 tensors

void **embeddingsToHost**()

Transfers gradients and embedding updates to host

Experimental evaluation

Datasets
- Freebase86m knowledge graph
- Twitter social graph
- LiveJournal
- Freebase15k

Models
- Dot
- ComplEx
- DistMult

Hardware
- Amazon EC2 p3.2xlarge
- V100 GPU, 61GB DRAM

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Presented here
- Large scale single-GPU comparison with PBG (Facebook) and DGL-KE (Amazon)
- BETA ordering runtime and IO reduction vs. existing orderings and lower bound

More in the paper
- System comparisons on two small/medium sized benchmark datasets
- Cost comparisons with multi-GPU and distributed configurations of DGL-KE and PBG
- The impact of asynchronous training and IO
- Scaling to configurations that are order(s) of magnitude larger than GPU and CPU capacity

Accuracy and Runtime Comparisons

Twitter

System	Model	MRR	Runtime
PBG	Dot Product	0.313	5h15m
DGL-KE	Dot Product	0.220	35h3m
Marius	Dot Product	0.310	**3h28m**

Marius up to **10x** faster than DGL-KE on large social graphs

Twitter
1.46 billion edges
41.6 million nodes
1 edge-type
d = 50

Freebase86m

System	Model	MRR	Runtime
PBG	ComplEx	0.725	7h27m
Marius	ComplEx	0.726	**2h1m**

Marius up to **3.7x** faster than PBG on large knowledge graphs

Freebase86m
338 million edges
86 million nodes
15,000 edge-types
d = 100

All systems are trained to 10 epochs, reaching convergence at near the same time

Compared Orderings

Lower bound
- Minimum number of swaps possible for a configuration

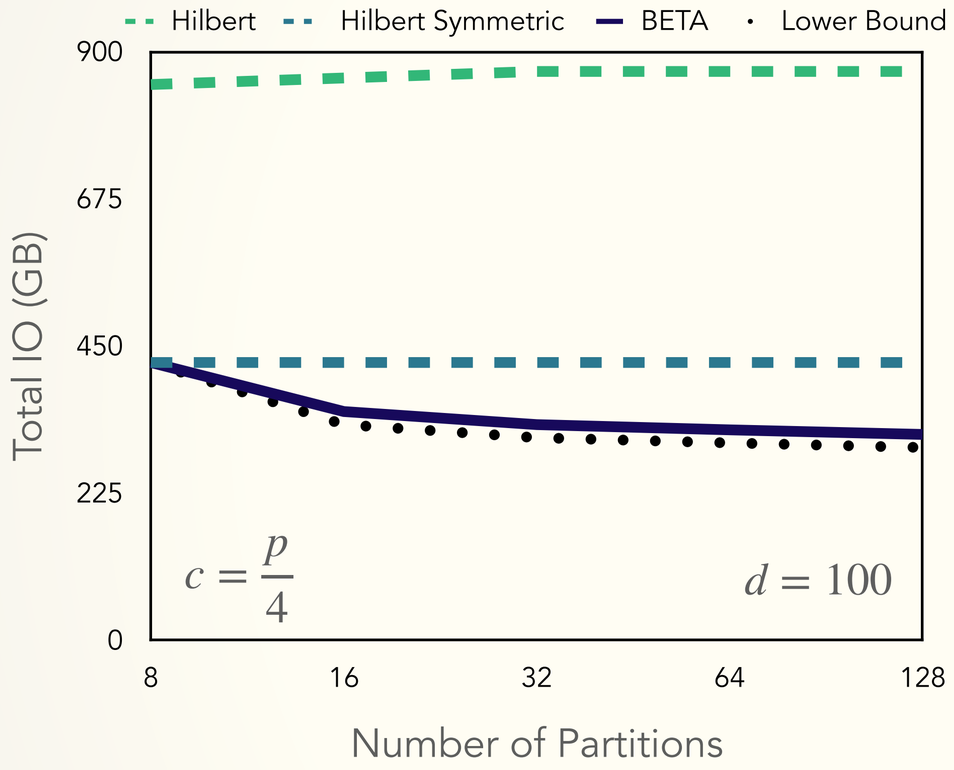
Hilbert
- Uses a Hilbert space filling curve to generate an ordering of the edge buckets

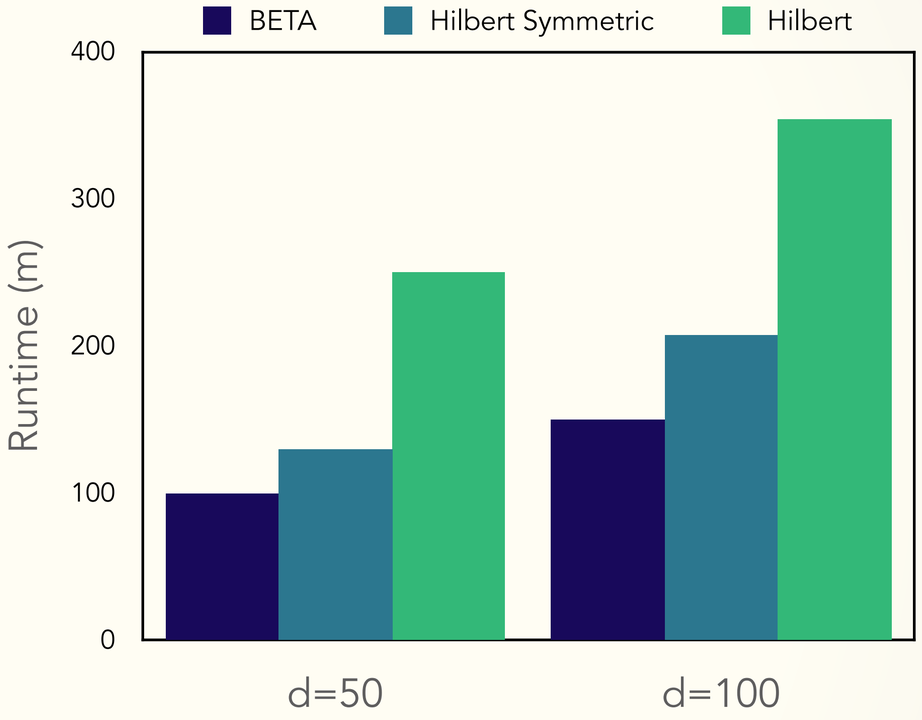
Hilbert Symmetric
- Modified Hilbert ordering which reduces swaps by 2x
- Processes edge buckets (j,i) and (i,j) together

Random
- Not evaluated, impractical to run as swaps scale quadratically with increasing partitions

BETA
- Our approach

Buffer-aware Edge Traversal Algorithm (BETA)





BETA ordering leads to 33% reduction in IO over locality based orderings

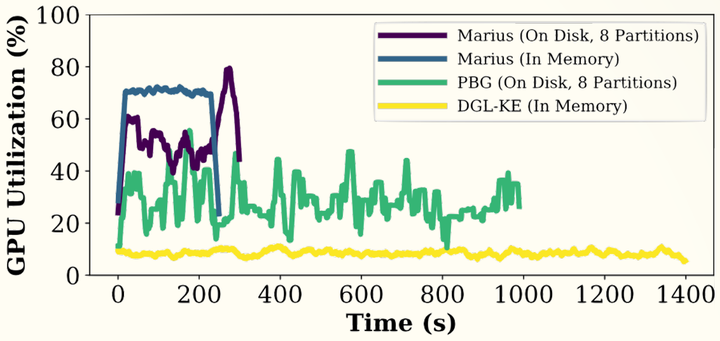
Reduction in IO corresponds directly with ~33% reduction in runtime

Freebase86m

Near the lower bound

c: buffer capacity, p: num partitions, d: embedding size

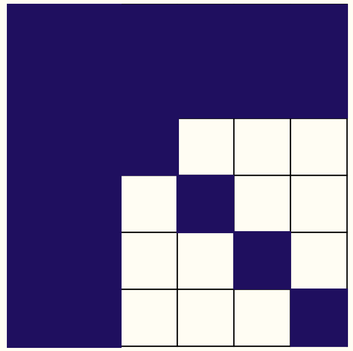
Conclusion & Future Work



Existing systems bottlenecked by data movement

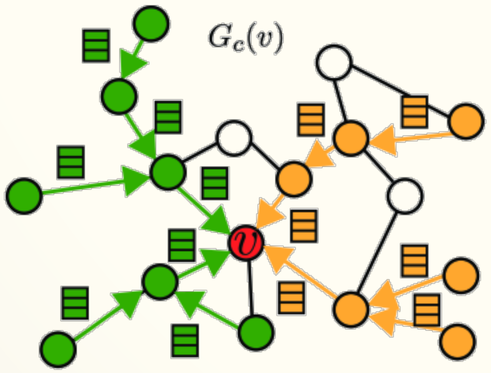
Marius alleviates data movement bottlenecks

- Pipelining/Async IO
- Partition Buffer
- BETA Ordering

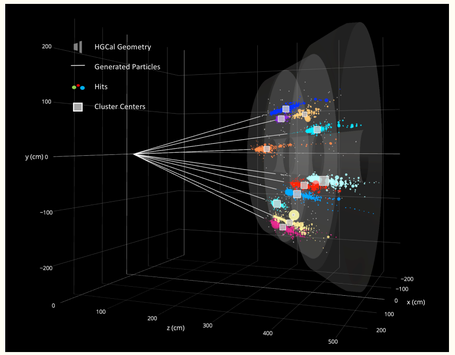


Future work

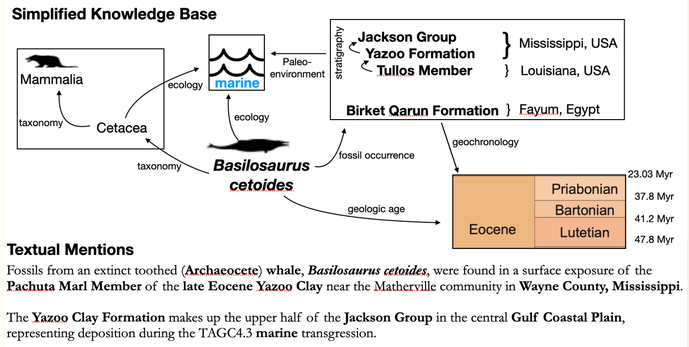
Scaling GNNs



High Energy Physics



Paleobiology (VLDB Demo 2021)

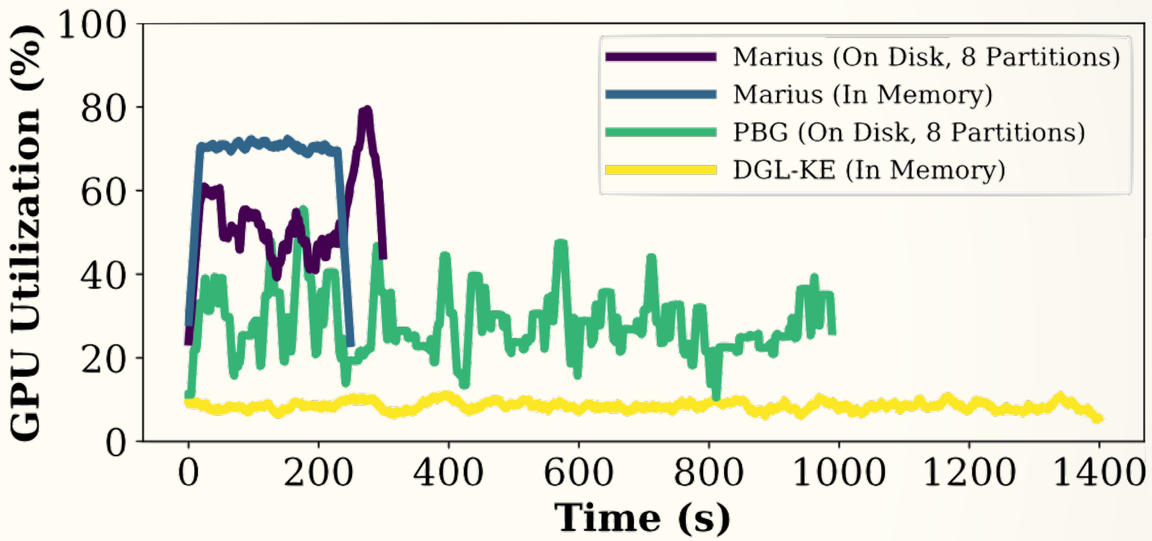


marius

Learning Massive Graph Embeddings on a Single Machine

Jason Mohoney*, Roger Waleffe, Yiheng Xu,
Theodoros Rekatsinas, Shivaram Venkataraman

* Email: mohoney2@wisc.edu



Open-source at marius-project.org

Thank you!

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON