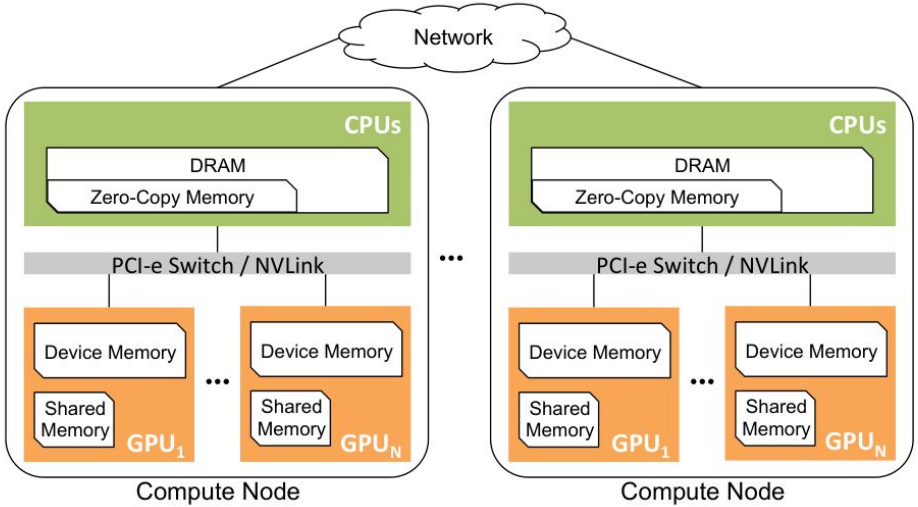
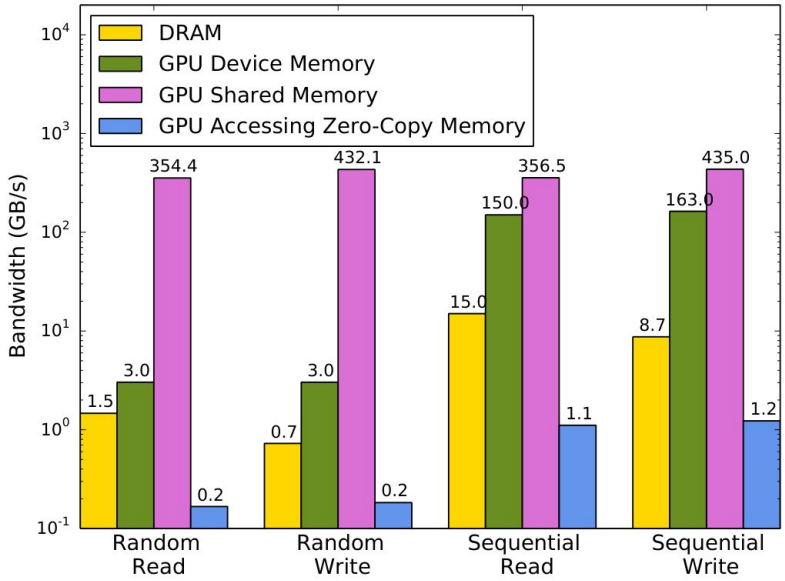
A Distributed Multi-GPU System for Fast Graph Processing

Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken

Background

- Graph processing limited by memory bandwidth
- → GPUs provide much higher memory bandwidth, but the memory hierarchy is more complex
- We cannot reuse CPU systems because of this difference





Background

CPU-based approach:

Store graph in DRAM, and optimise data transfer between DRAM.

Current systems used push-based operations for updates (message passing in Pregel, VertexMaps in Ligra).

GPU-based approach:

Distribute graph among GPU device memory, GPU shared memory, zero-copy memory, and DRAM. Take advantage of locality in multi-GPU settings.

Push-based operations interfere with GPU optimisations (e.g. aggregating vertex updates).

Contributions

- Design and implementation of Lux, a distributed multi-GPU system
- Two execution models: Push for algorithmic efficiency, Pull for GPU optimisation
- Novel dynamic repartitioning strategy with minimal overhead
- Performance models to aid with selecting best configurations

Overview of Lux

- Built for iterative computations on graphs
- Vertex properties mutable, but edge properties immutable
- Interface: init, compute, update

Pull model

Algorithm 1 Pseudocode for generic pull-based execution.

```
1: while not halt do
2:     halt = true                          ▷ halt is a global variable
3:     for all v ∈ V do in parallel
4:         init(v, v^old)
5:         for all u ∈ N⁻(v) do in parallel
6:             compute(v, u^old, (u, v))
7:         end for
8:         if update(v, v^old) then
9:             halt = false
10:        end if
11:    end for
12: end while
```

Pull updates from all in-neighbors

Order is non-deterministic

Compute must be able to run concurrently

-> Grouping updates from all in-neighbors allows for GPU-specific optimisations *(e.g. locally aggregating updates in shared memory)*

Push model

Algorithm 2 Pseudocode for generic push-based execution.

```
1: while F ≠ {} do
2:     for all v ∈ V do in parallel
3:         init(v, v^old)
4:     end for
5:                                          ▷ synchronize(V)
6:     for all u ∈ F do in parallel
7:         for all v ∈ N+(u) do in parallel
8:             compute(v, u^old, (u, v))
9:         end for
10:    end for
11:                                         ▷ synchronize(V)
12:    F = {}
13:    for all v ∈ V do in parallel
14:        if update(v, v^old) then
15:            F = F ∪ {v}
16:        end if
17:    end for
18: end while
```

Each vertex pushes updates to neighbors

Same calls to init and update, but only compute on edges from updated vertices

→ Algorithmic optimisations, but requires synchronisation!

The Lux system

- Core idea: exploit locality in memory hierarchy:
  - Zero-copy memory is visible to all GPUs and CPUs on a node
    -> Use zero-copy memory for mutable data (vertex updates) to make them easily accessible
  - Use shared memory to aggregate updatesk

- Use edge partitioning (divide edges equally)
  - Each partition stores a set of contiguous vertices and all incoming edges
    -> All edges that update a vertex are stored in the same partition
  - Seek to balance edges across partitions - roughly $|E|/x$ edges for each of the x GPUs
  - Because of the ordering, GPUs only need to know first and last vertex of the partition!

- Maintain set of all remote vertices that publish updates to local vertices
  - Send all updates to other compute nodes at the end of an iteration

Load balancing

- Moving partitions between GPUs is expensive compared to CPU systems
- We only need to know the first and the last vertex to describe the partition

- Core idea: Assume compute time is proportional to in-degree.
- After each iteration, we know the load of each partition
- Use this updated information to calculate new partitions
- Only modify partitions when gains are significant compared to the cost of partitioning
- Use global partitioning first, local partitioning afterwards

Performance model

- Predict computation time to select best execution model and configuration for a given application and graph
- Estimate time to load, compute, update, and transfer between nodes
- Core idea: Assume that all vertices take up the same memory, all edges require the same computation time, etc.

Evaluation

Benchmarked on:

- PageRank (PR)
- Connected components (CC)
- Single source shortest path (SSSP)
- Betweenness centrality (BC)
- Collaborative filtering (CF)

Experimented with many parameters to get best performance in the compared models

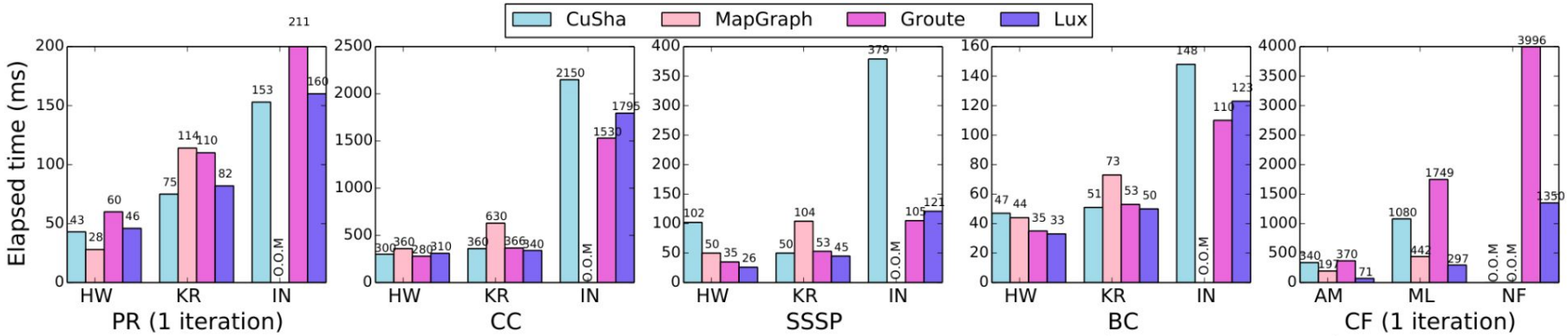


Figure 15: Performance comparison on a single GPU (lower is better).

- Only done on graphs that fit into memory on 1 GPU
- Although Lux uses zero-copy memory, it is still very fast

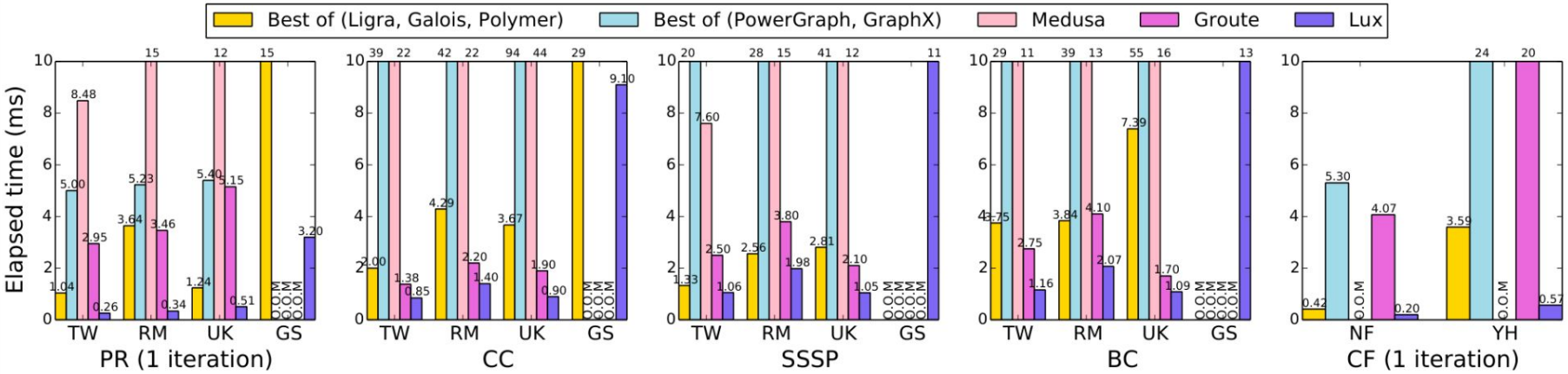


Figure 16: The execution time for different graph processing frameworks (lower is better).

- Recall that the shared memory systems have no overhead in terms of data transfer and partitioning. Thus, to compete, Lux must speed up compute dramatically. It does so by up to 30x

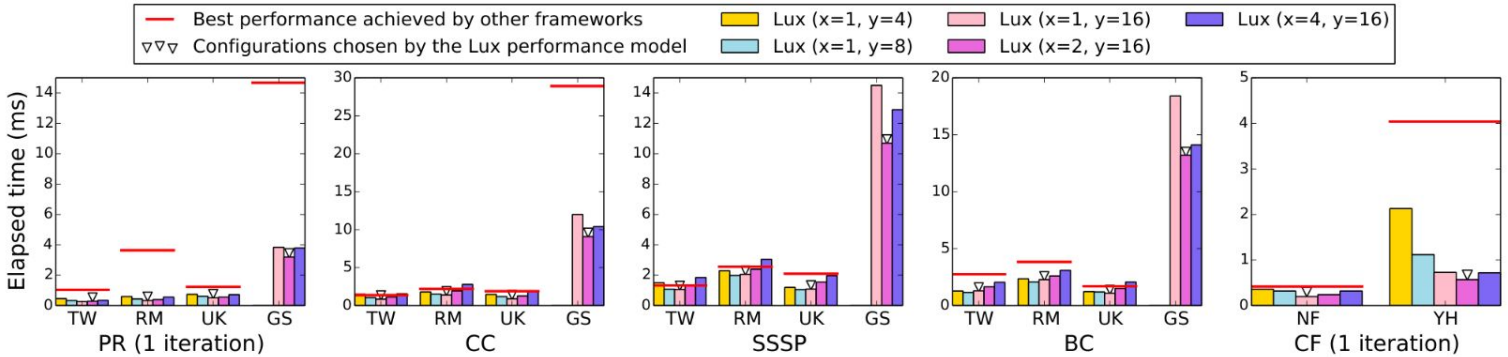


Figure 17: The execution time for different Lux configurations (lower is better). x and y indicate the number of nodes and the number of GPUs on each node.

Load balancing

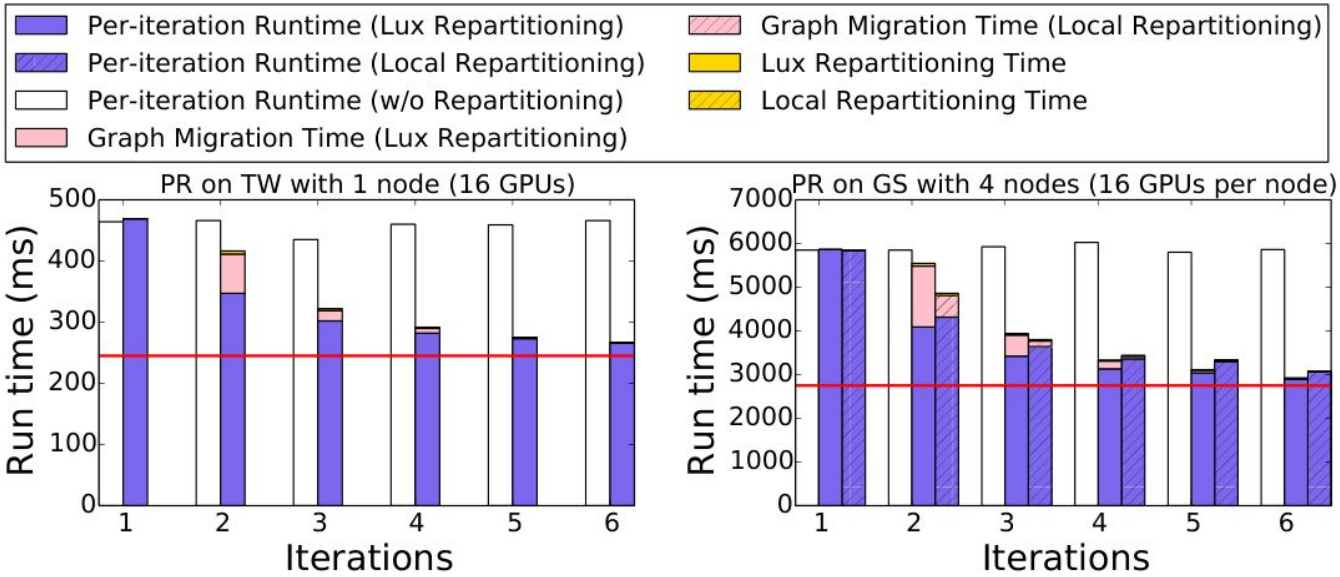


Figure 18: Performance comparison for different dynamic repartitioning approaches. The horizontal line shows the expected per-iteration run time with perfect load balancing.

Price comparison

Machines	Lonestar5	XStream (4GPUs)	XStream (8GPUs)	XStream (16GPUs)
Machine Prices (as of May 2017)				
CPUs [4, 3]	15352	3446	3446	3446
DRAM [8]	12784	2552	2552	2552
GPUs [7]	0	20000	40000	80000
Total	28136	25998	45998	85998
Cost Efficiency (higher is better)				
PR (TW)	0.20	0.84	0.64	0.45
CC (TW)	0.18	0.26	0.21	0.14
SSSP(TW)	0.14	0.25	0.20	0.10
BC(TW)	0.14	0.30	0.18	0.10
CF (NF)	0.85	1.07	0.68	0.58

Criticism

- Requires immutable edges
- Focuses on static graphs - no support for dynamic graphs?
- Evaluation does not examine graphs exceeding GPU memory (spillover into DRAM)
- Not radically different from gather-apply-scatter (Pregel and PowerGraph)
- Scalability could probably be improved, especially with multiple compute nodes

Questions