PIM-tree: A Skew-resistant Index for Processing-in-Memory

Hongbo Kang
Tsinghua Univeristy

Yiwei Zhao
CMU

Guy E. Blelloch
CMU

Laxman Dhulipala
University of Maryland

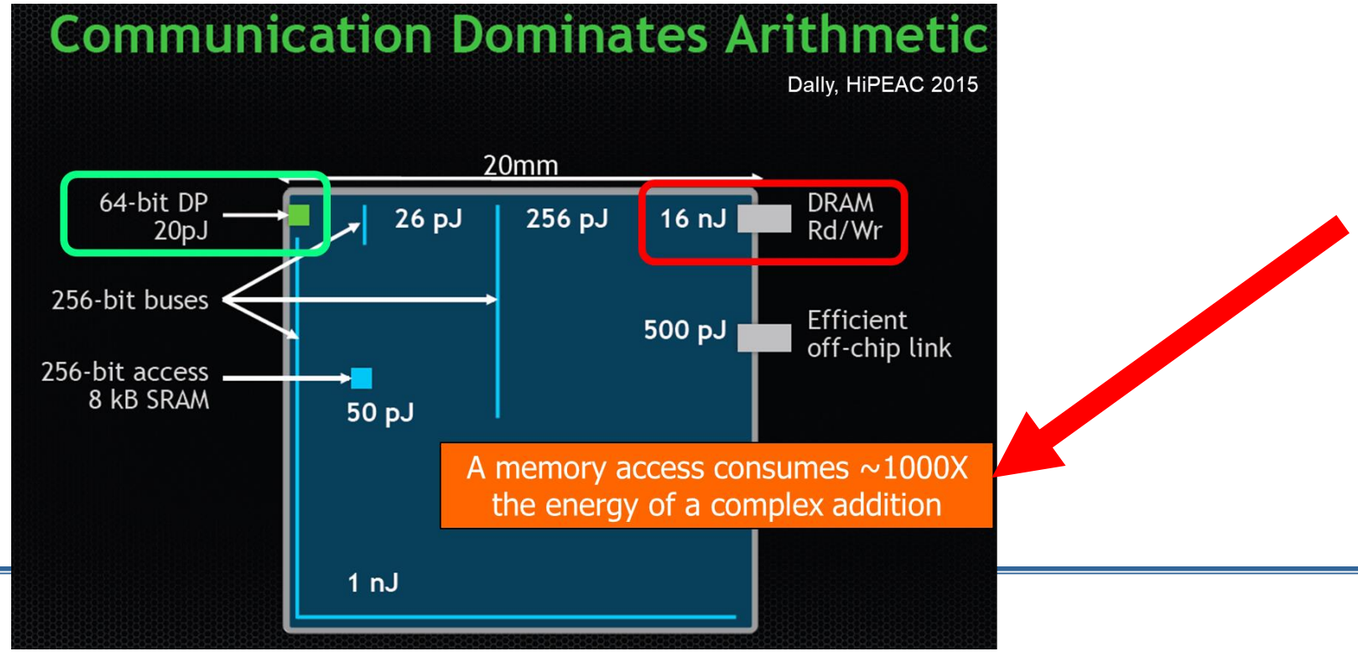
Yan Gu
UC Riverside

Charles McGuffey
Reed College

Phillip B. Gibbons
CMU

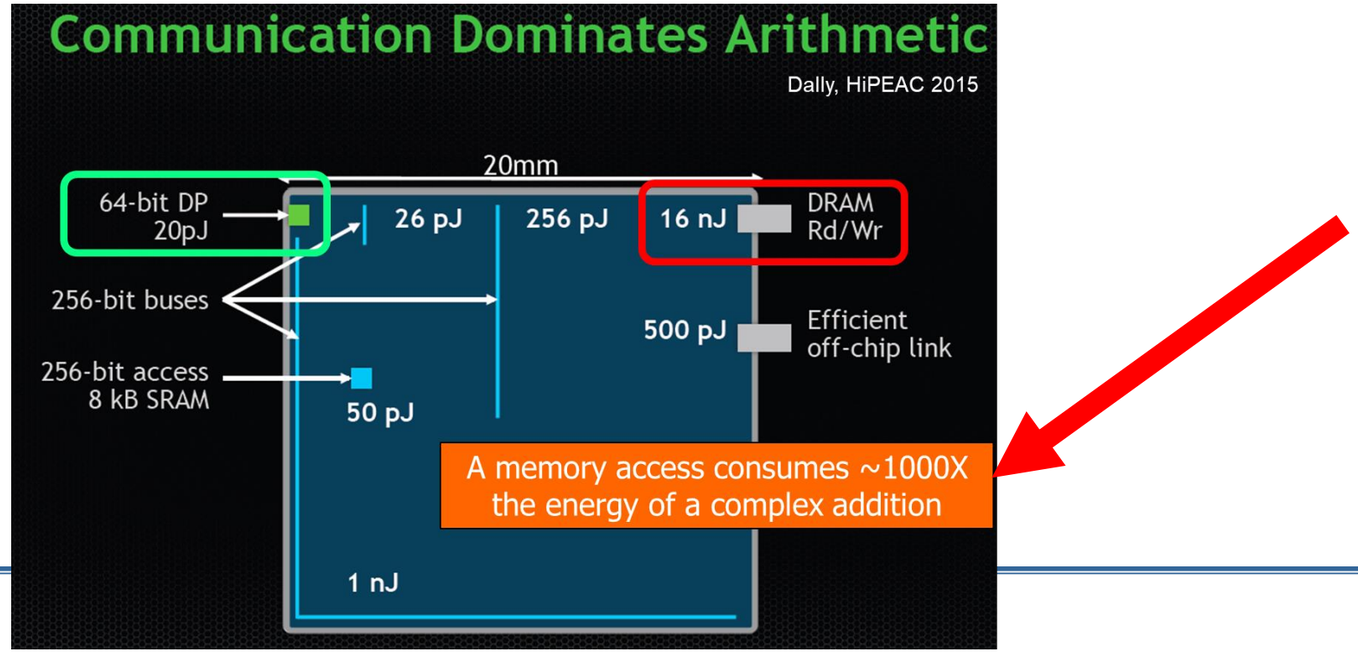
The Memory Wall Problem

- Memory Wall
  - Widening gap between the processor performance and memory access speed (memory bandwidth/latency).
  - High power consumption of data movement.
- Data movement becomes the key bottleneck.



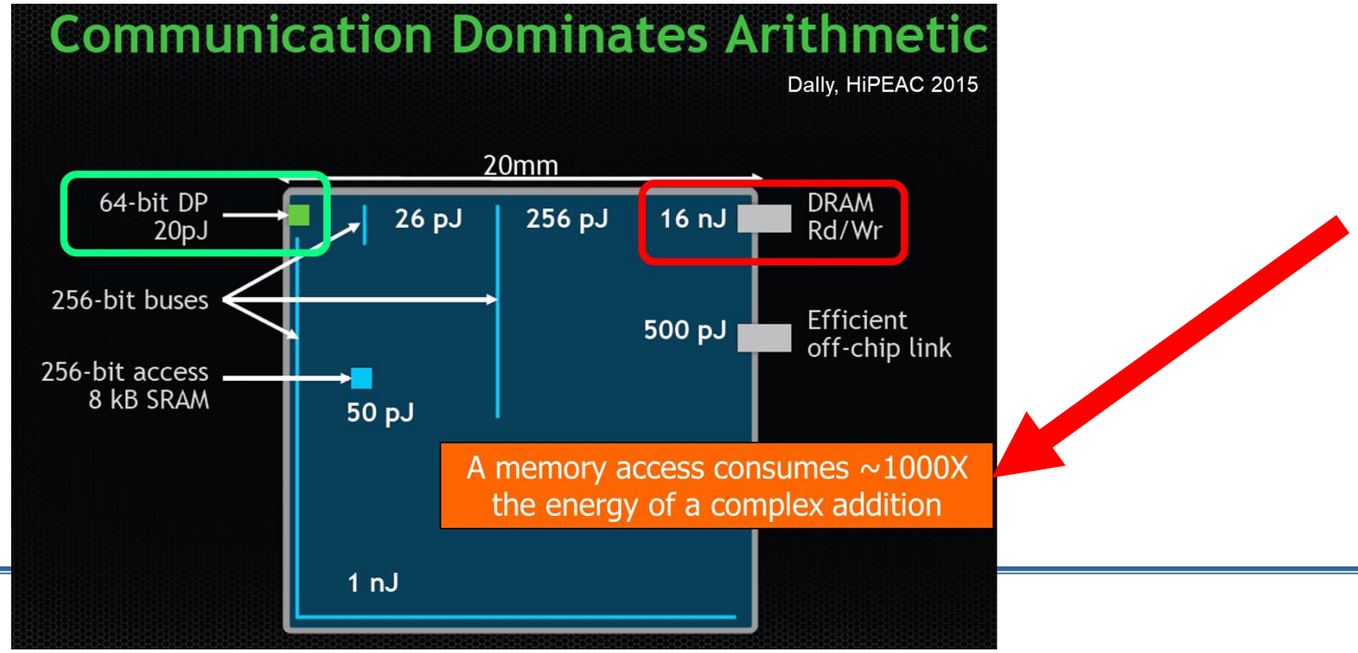
The Memory Wall Problem

- Memory Wall
  - Widening gap between the processor performance and memory access speed (memory bandwidth/latency).
  - High power consumption of data movement.
- Data movement becomes the key bottleneck.



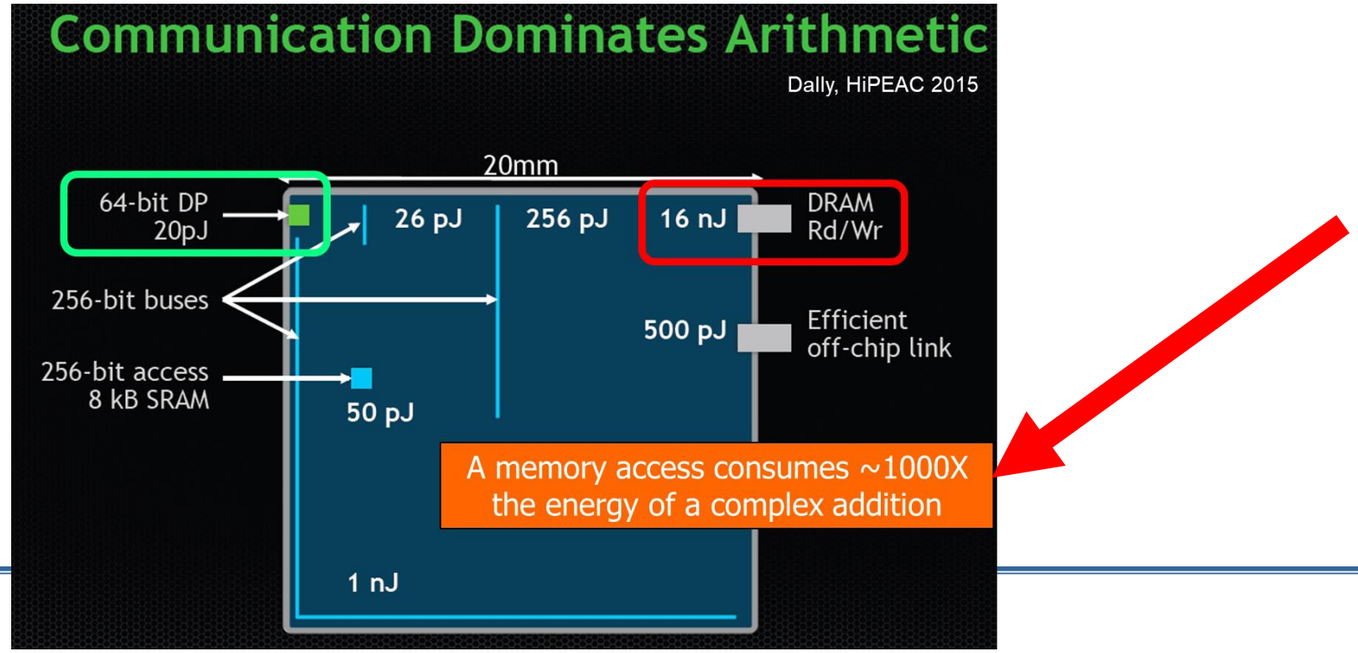
The Memory Wall Problem

- Memory Wall
  - Widening gap between the processor performance and memory access speed (memory bandwidth/latency).
  - High power consumption of data movement.
- Data movement becomes the key bottleneck.



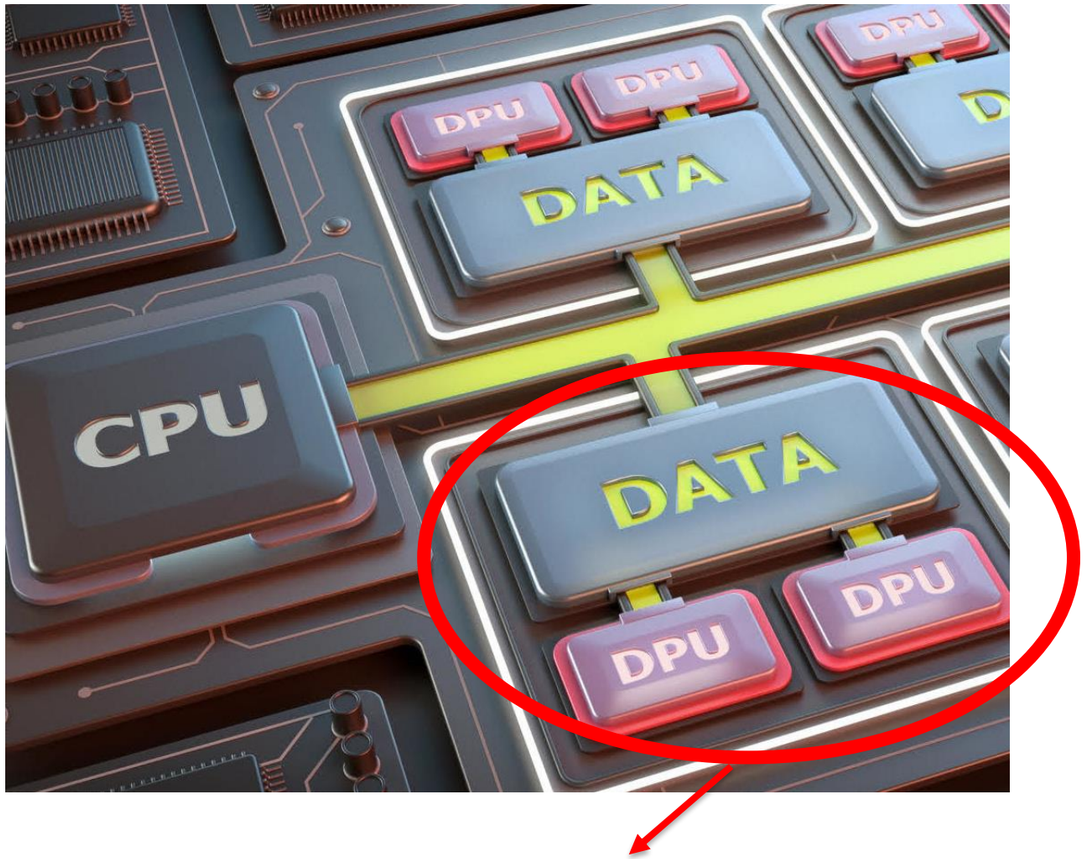
The Memory Wall Problem

- Memory Wall
  - Widening gap between the processor performance and memory access speed (memory bandwidth/latency).
  - High power consumption of data movement.
- **Data movement** becomes the key bottleneck.



Processing-in-Memory (PIM)

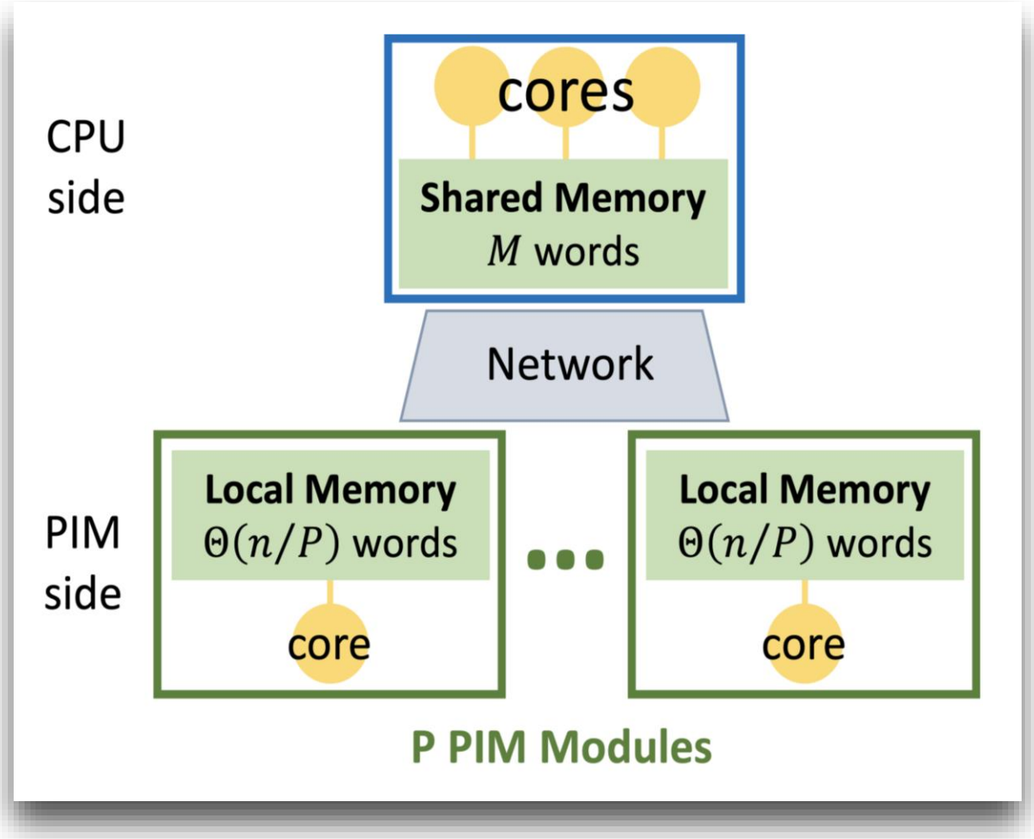
- Processing-in-memory enables computation to be pushed to memory
  - Appears in 1970s
  - Advance in memory tech.
  - Core(s) in each memory module



PIM module:
Data Processing Unit (PIM core)
& the local memory it can visit

The Processing-in-Memory Model*

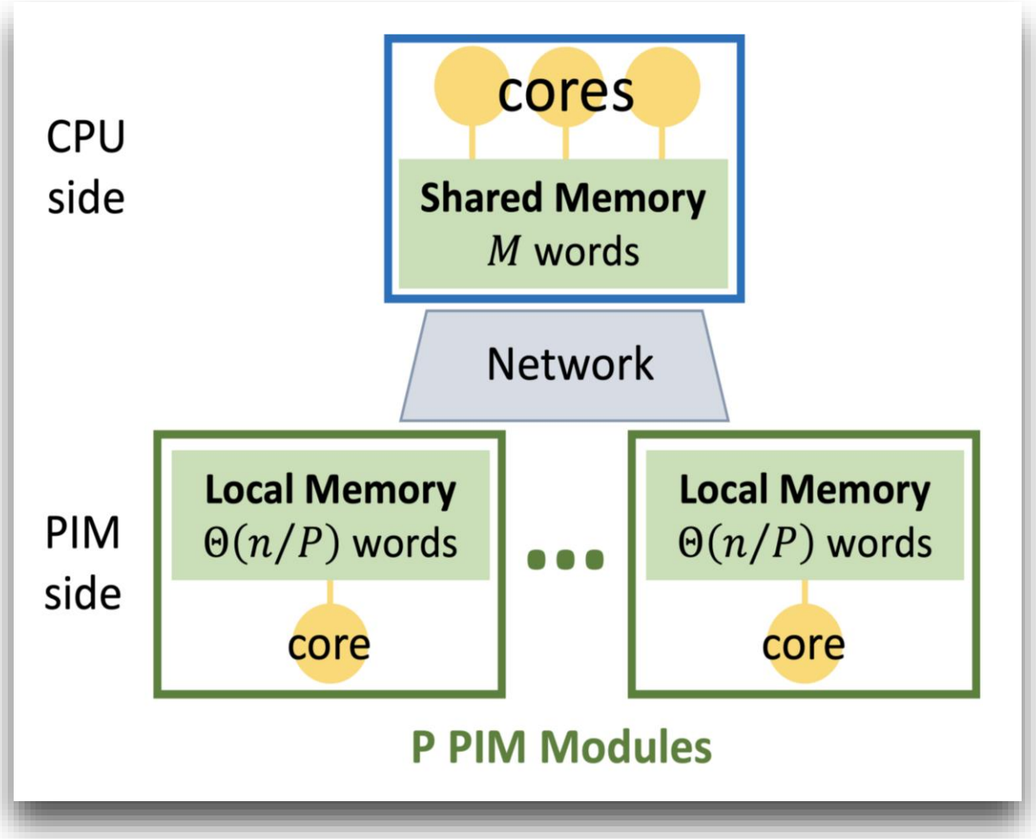
- One multicore CPU side
  - With a shared cache (LLC) of size M words
- P PIM modules in total
- Each PIM module
  - One PIM core that can only visit local memory & communicate with CPU through network
  - Local memory sized $\Theta(n/P)$** words



*** Kang, et al. The Processing-In-Memory Model. SPAA'21.**
**** n is the problem size / input data size.**

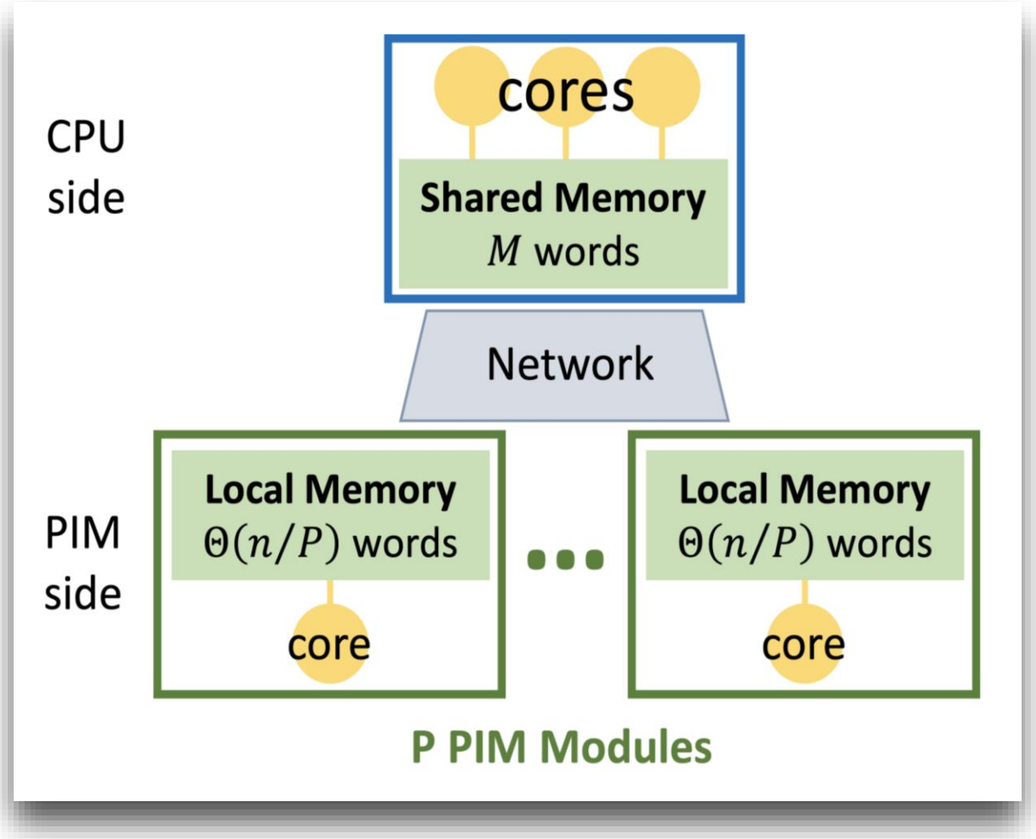
The Processing-in-Memory Model

- BSP style execution
  - Synchronous rounds
- Metrics
  - CPU work & span
  - PIM time: Maximum work on any PIM processor
  - IO time: Maximum number of word-sized messages from/to any PIM module



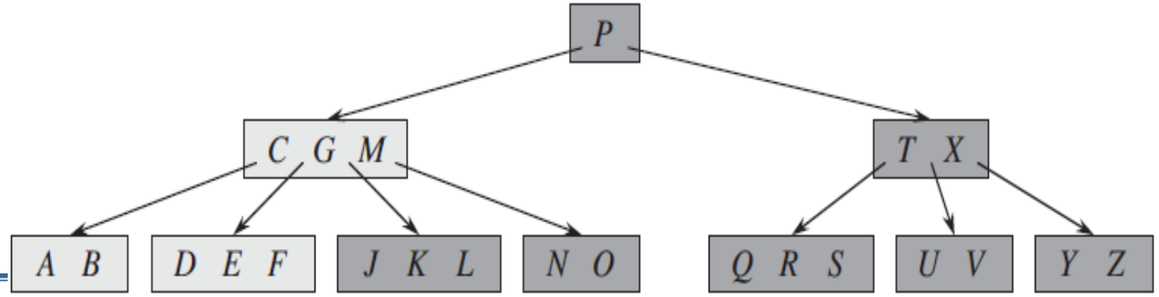
The Processing-in-Memory Model

- BSP style execution
  - Synchronous rounds
- Metrics
  - CPU work & span
  - PIM time: Maximum work on any PIM processor
  - IO time: Maximum number of word-sized messages from/to any PIM module



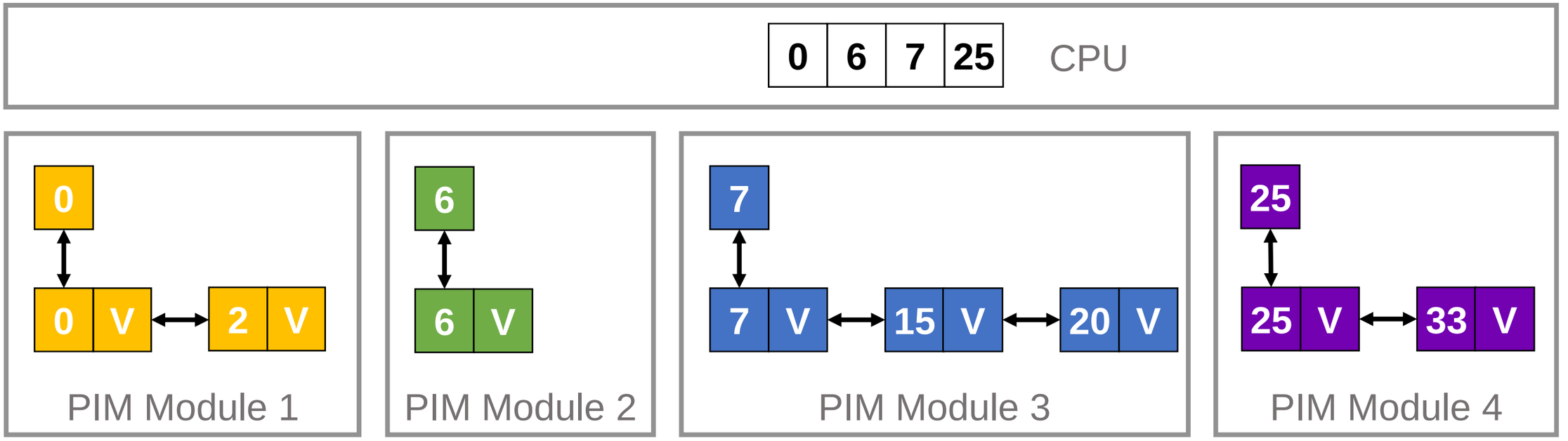
Pointer-Chasing Index

- Core block for in-memory storages.
  - Victim of its serial random memory access pattern (pointer-chase)
- Store key-value paired data
- Support operations:
  - Get(key)
  - Insert(key, value) & Delete(key)
  - Predecessor/Successor(key) & Range(Lkey, Rkey)
- Batch Parallel Execution
- Skip list
  - B-tree?



Prior Work: PIM-based Ordered Indexes*

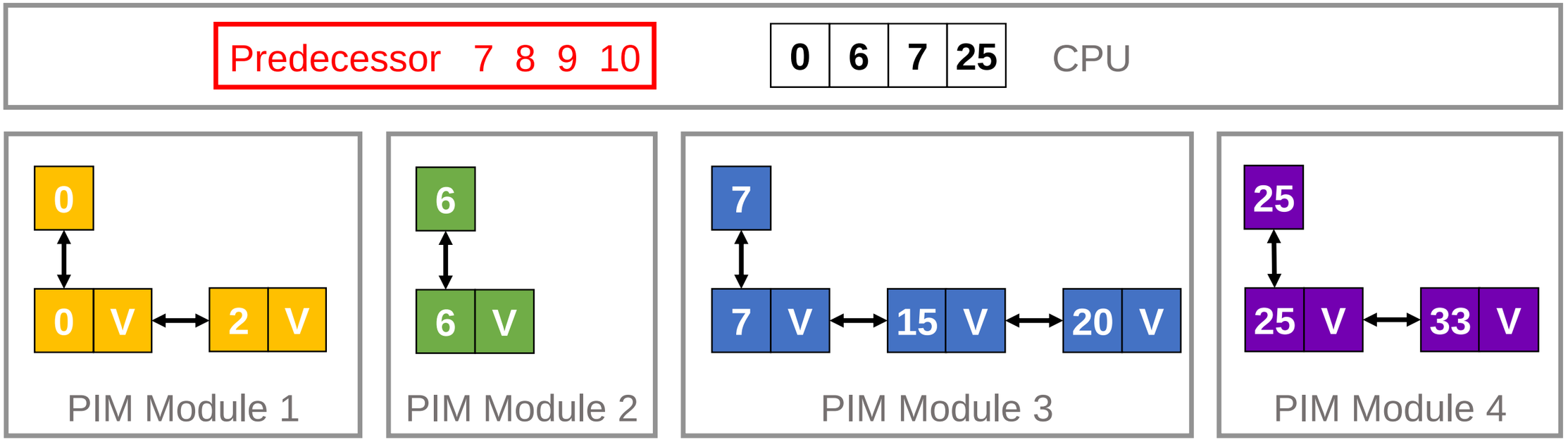
- Range-partitioning: Local skip lists for disjoint key ranges
- Pros:
  - Low data movement: Constant per point operation
  - Parallelism for keys in different ranges



*** Choe, et al. Concurrent Data Structures with Near-Data-Processing. SPAA'19.**

Drawbacks of Range Partitioning Indexes*

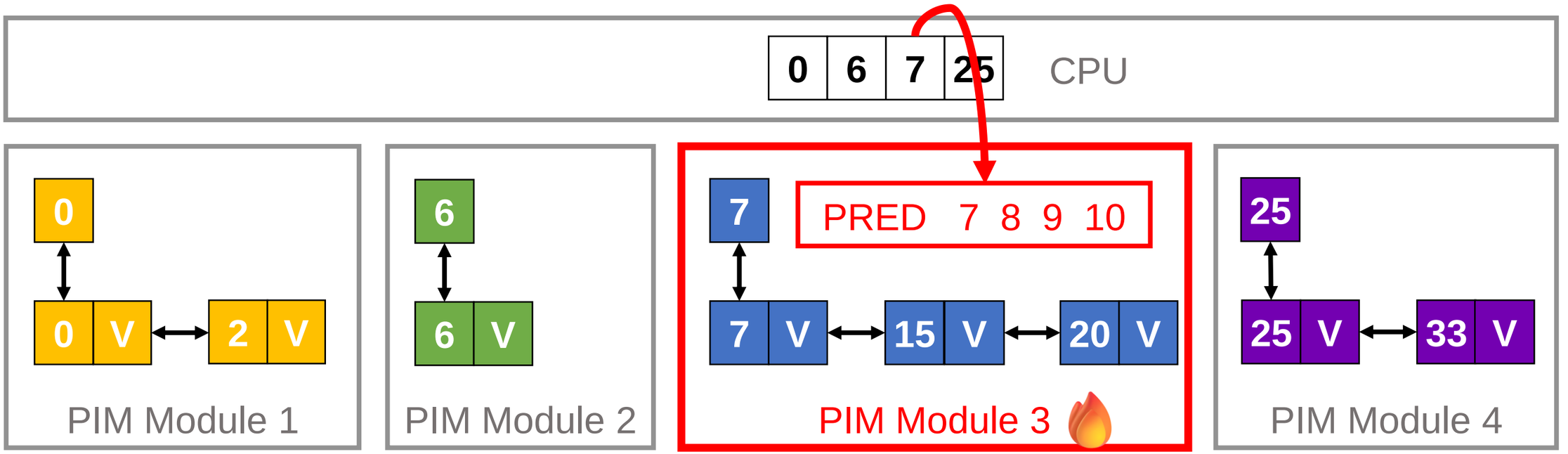
- Cons: Vulnerable to **skewed** operations batches
  - Query skew occur in real datasets: YCSB, …
  - Uneven distribution of keys cause **load imbalance**



*** Choe, et al. Concurrent Data Structures with Near-Data-Processing. SPAA'19.**

Drawbacks of Range Partitioning Indexes*

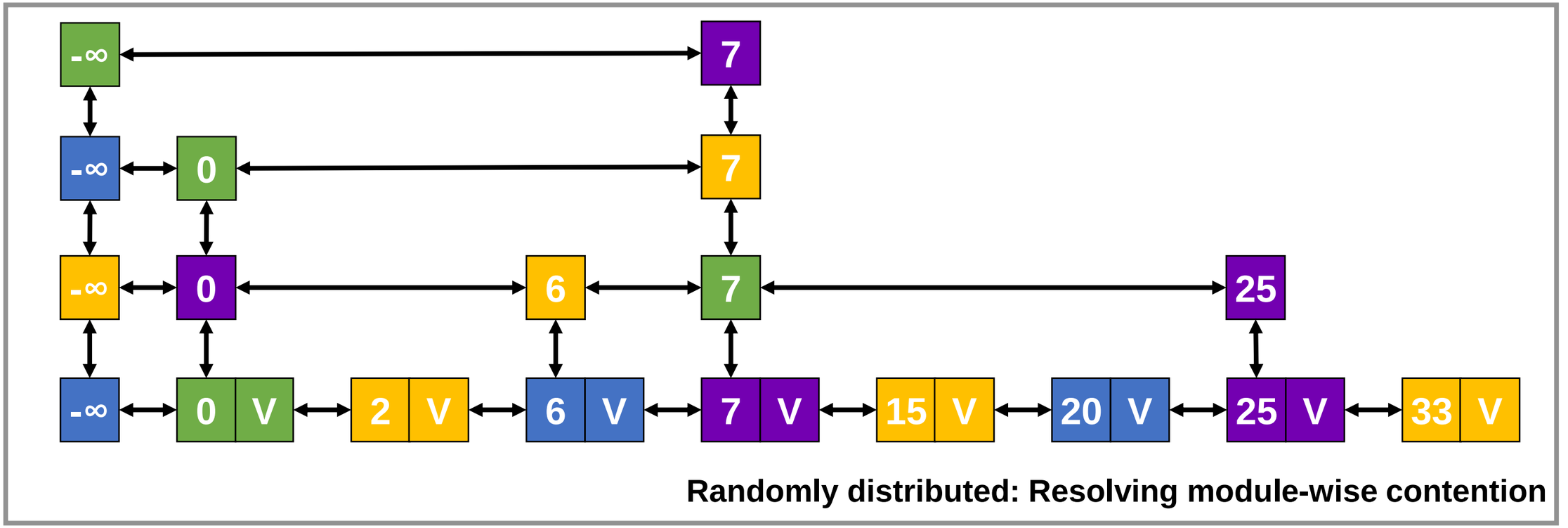
- Cons: Vulnerable to **skewed** operations batches
  - Query skew occur in real datasets: YCSB, …
  - Uneven distribution of keys cause **load imbalance**



*** Choe, et al. Concurrent Data Structures with Near-Data-Processing. SPAA'19.**

Alternative Approach: Random Mapping

Assuming 4 PIM modules (P = 4 as number of PIM modules)

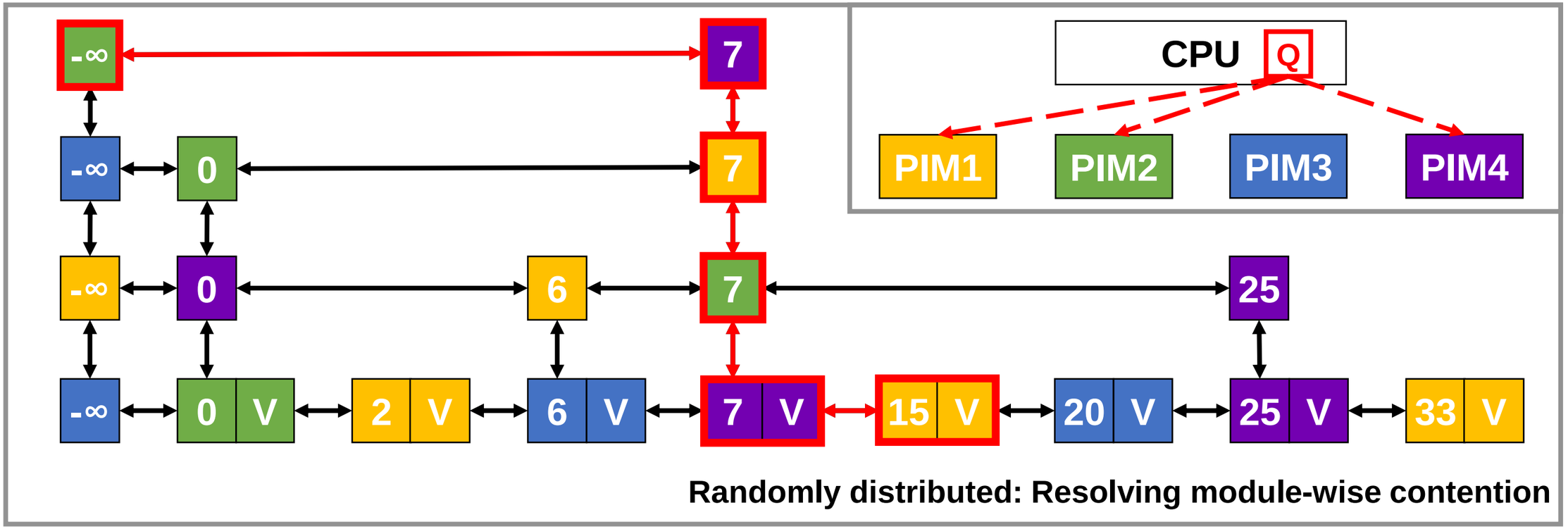


Pros: Resolving **module-wise contention**

Cons: Does not reduce communication [$O(\log n)$ in trees]

Alternative Approach: Random Mapping

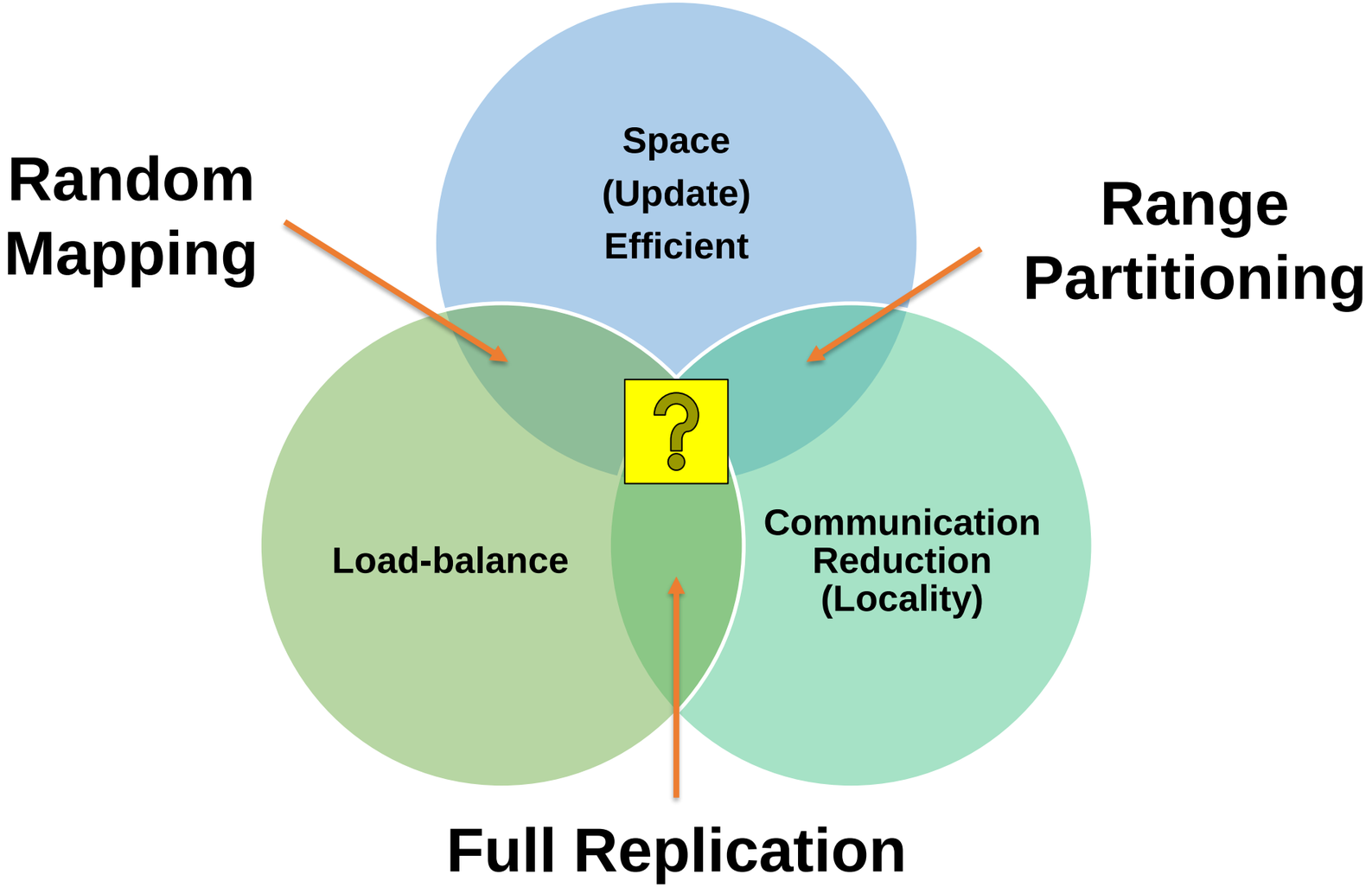
Assuming 4 PIM modules (P = 4 as number of PIM modules)



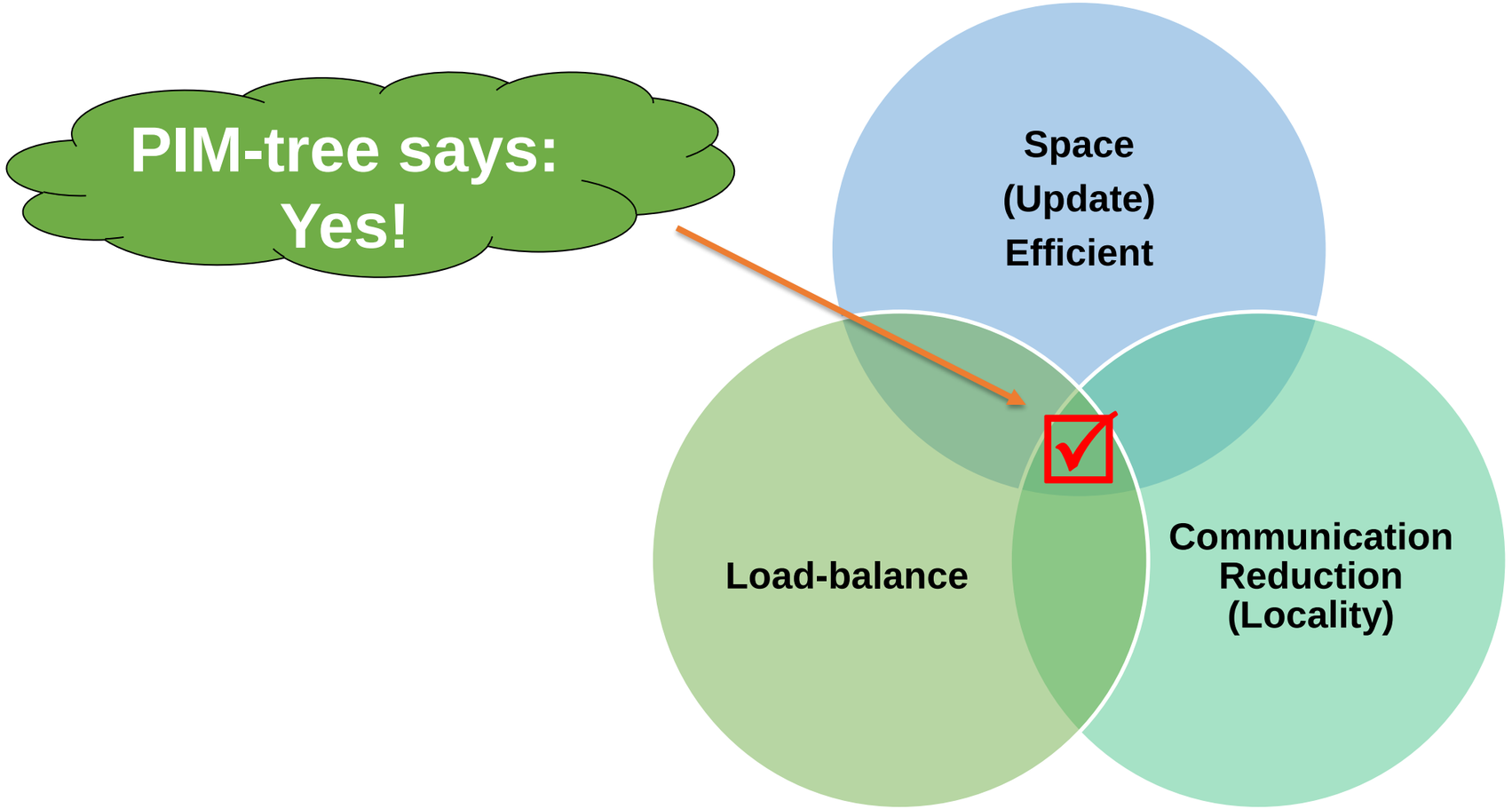
Pros: Resolving **module-wise contention**

Cons: Does not reduce communication [$O(\log n)$ in trees]

Designing PIM-based indexes



Designing PIM-based indexes



Achieving provable (theoretical) and practical efficiency in throughput, communication and space.

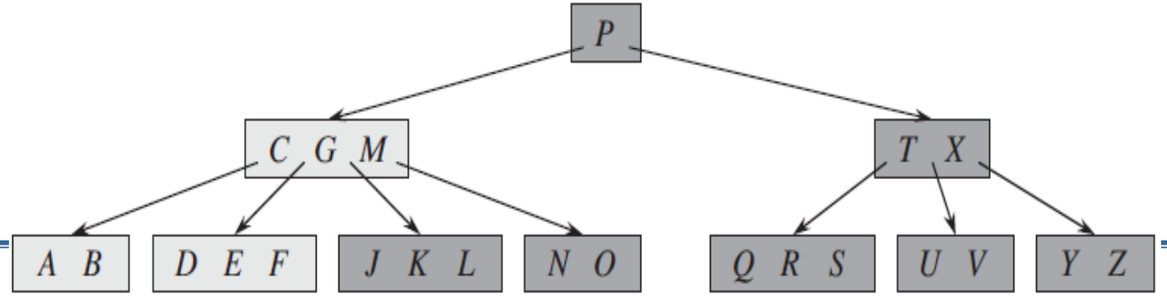
Our Work in a Nutshell

- **Replicated upper part**: From key insights of trees.
- **Push-pull search**: A lightweight design for load balancing.
- **Shadow subtree**: Provide vertical locality.
- **Chunking**: Provide horizontal locality.

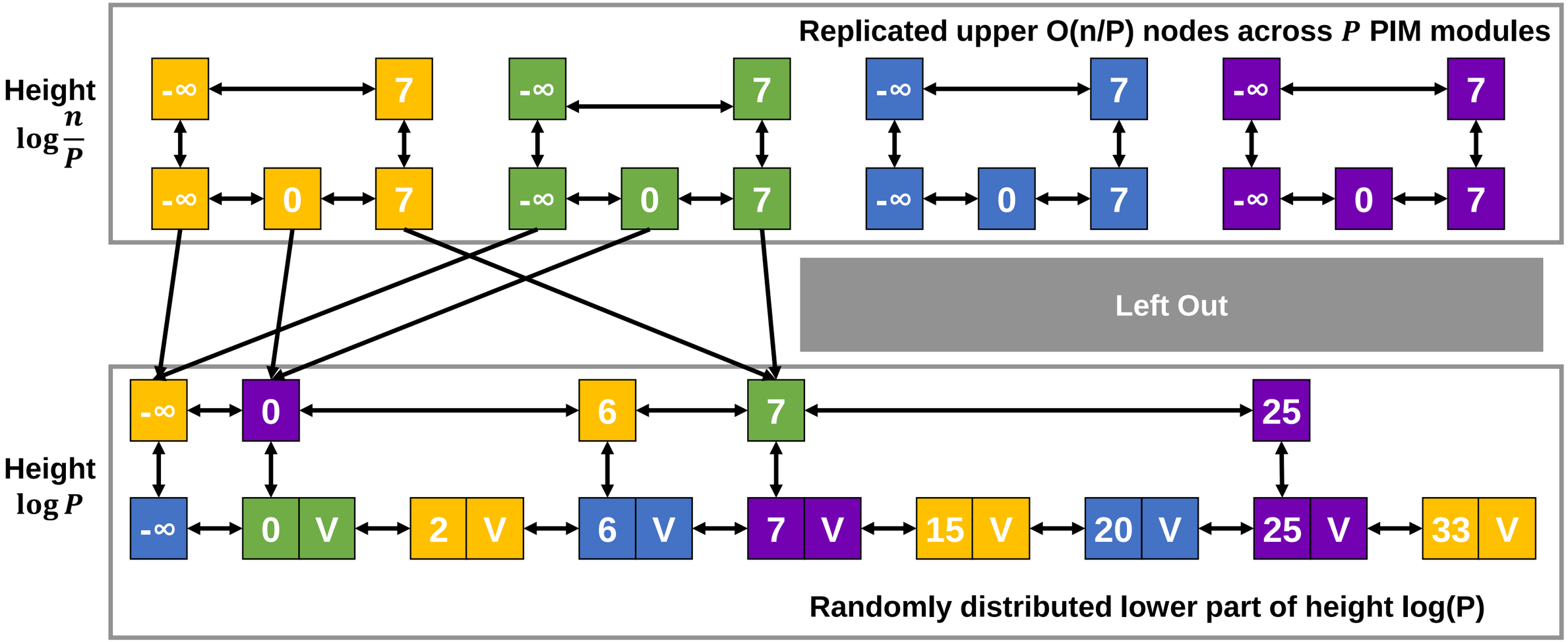
Taking Insight from Trees

- Key Observation in Tree-structured Indexes
  - The upper part of the tree is **frequently accessed** in search queries.
  - The size of the upper part is asymptotically **small**.
  - The upper part is **infrequently updated**.

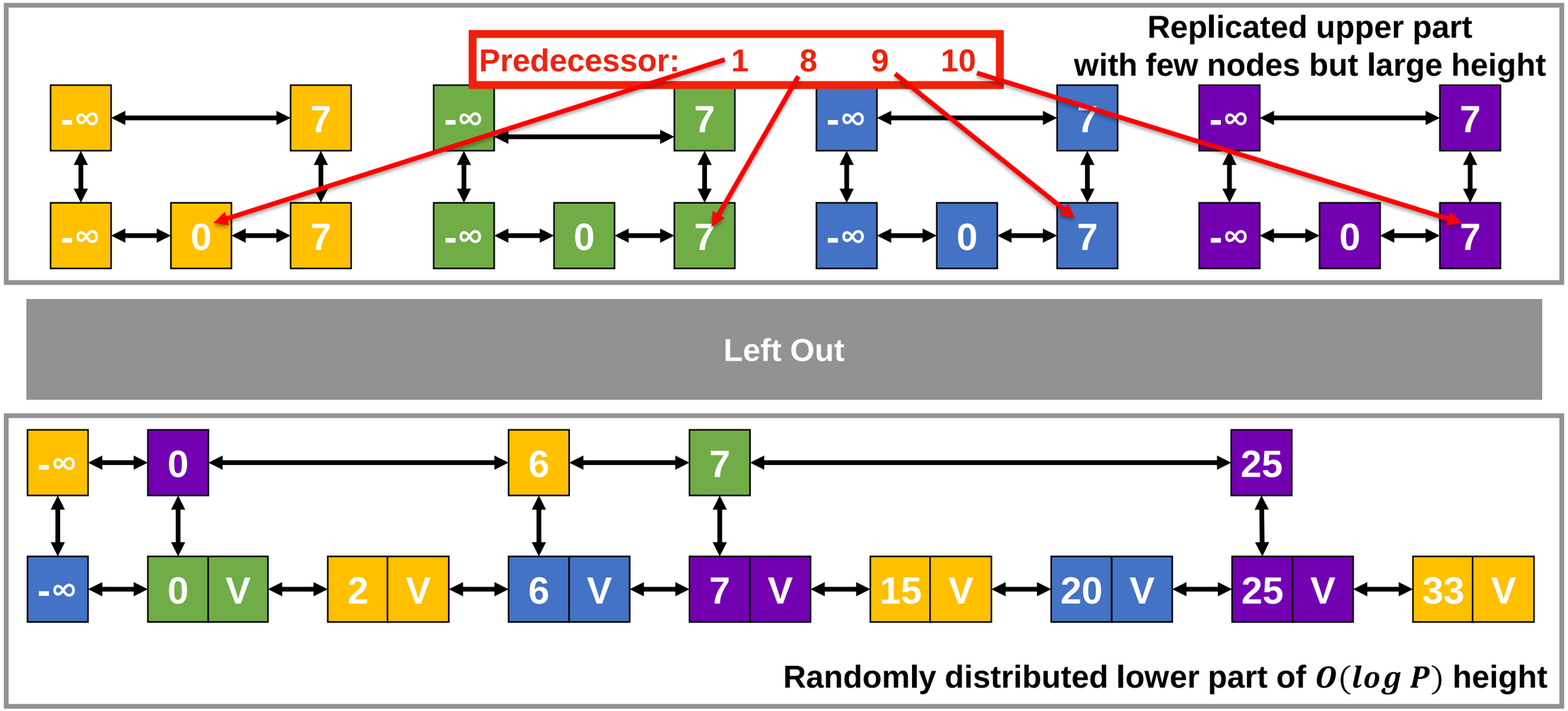
- **Replicating** the upper part in PIM modules should be beneficial.



Two-Layer Structure: Replicated upper part



Powerful Upper Part: Load-balance & Comm.



Low Communication: Only need to communicate $O(\log P)$.

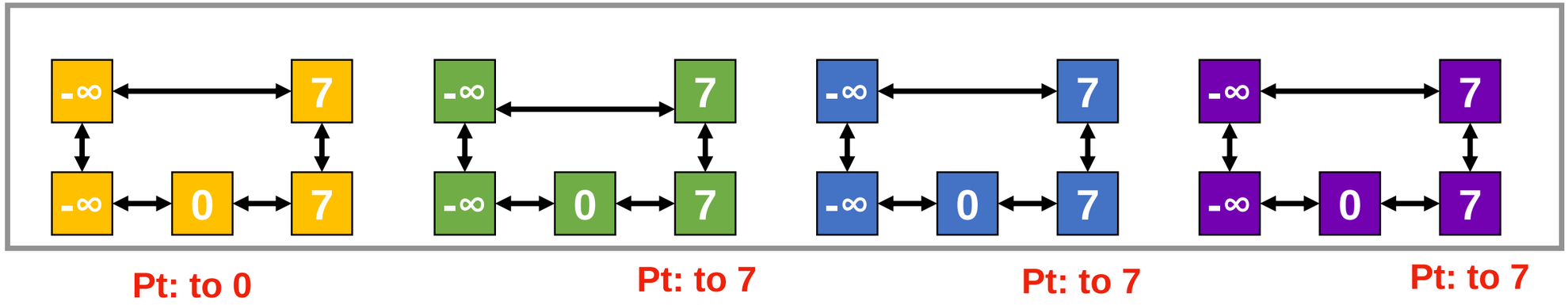
Challenge: Node-wise Contention on Lower Part

CPU side

PIM side

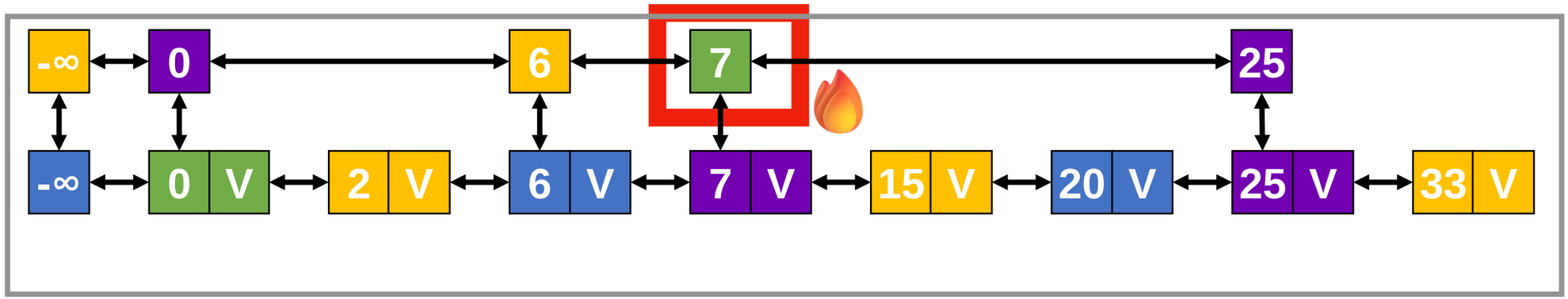
Predecessor:

1 8 9 10



Pt: to 0 Pt: to 7 Pt: to 7 Pt: to 7

1 8 9 10



Solution: Push-Pull Search

CPU side

Predecessor:

1 8 9 10

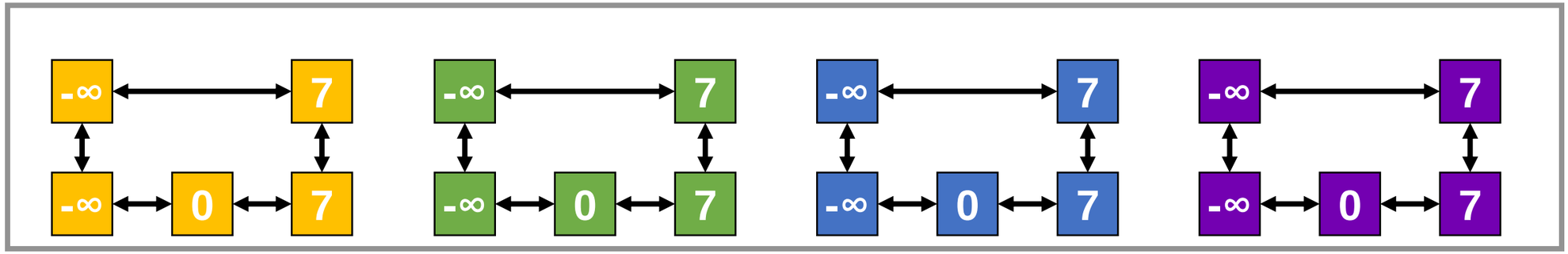
Pt: to 0 Pt: to 7

Pt: to 7 Pt: to 7

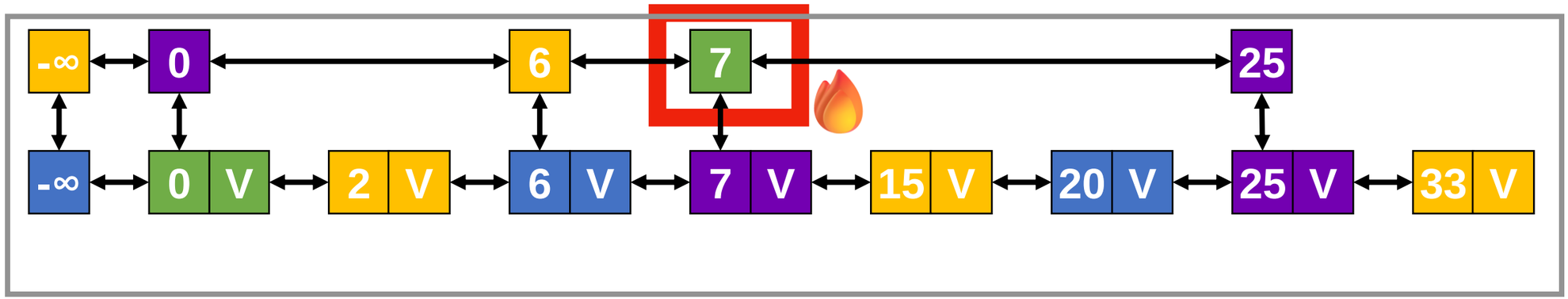
1 8 9 10

Pull data to CPU

PIM side

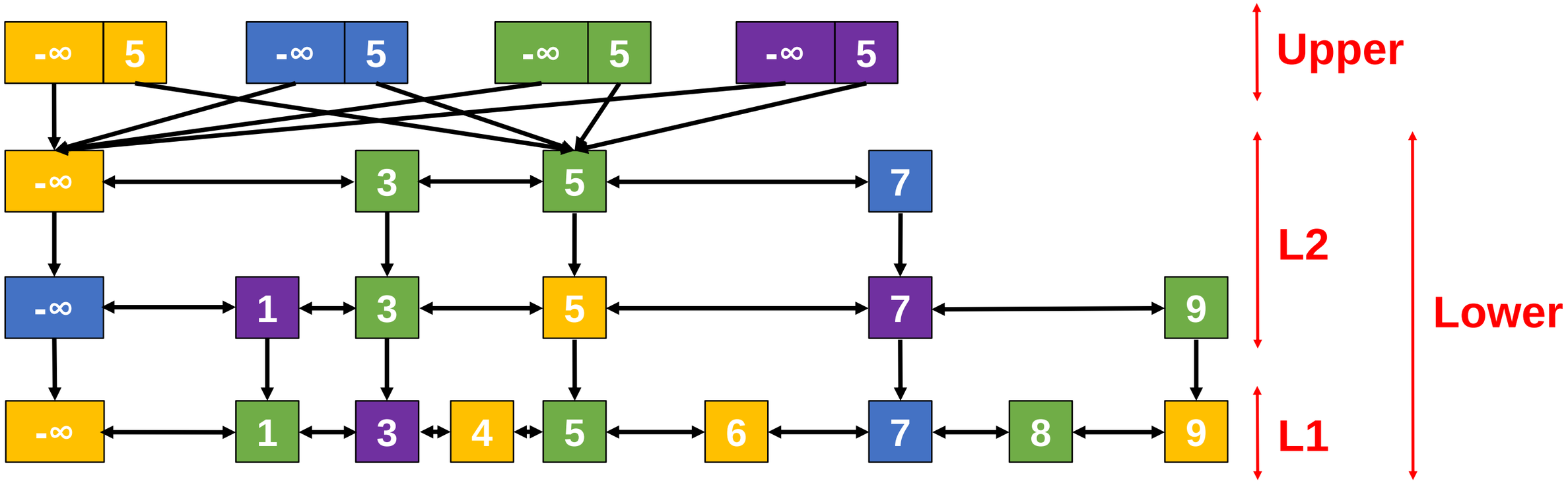


Push computation to PIM



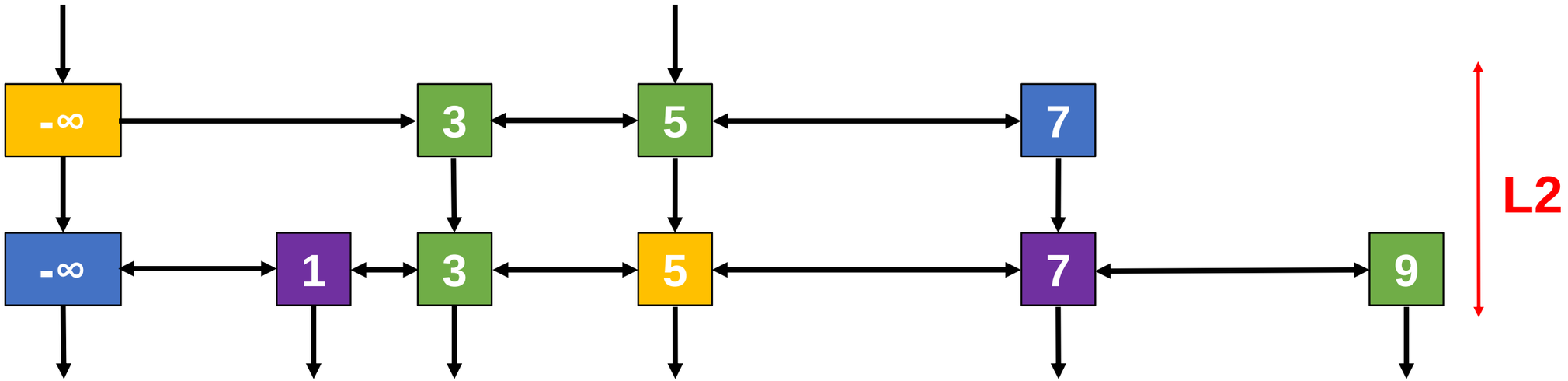
Shadow Subtree: Partial Replication

- Rebuild vertical locality in lower part by **partial** replication.
  - Due to space constraint and update cost.



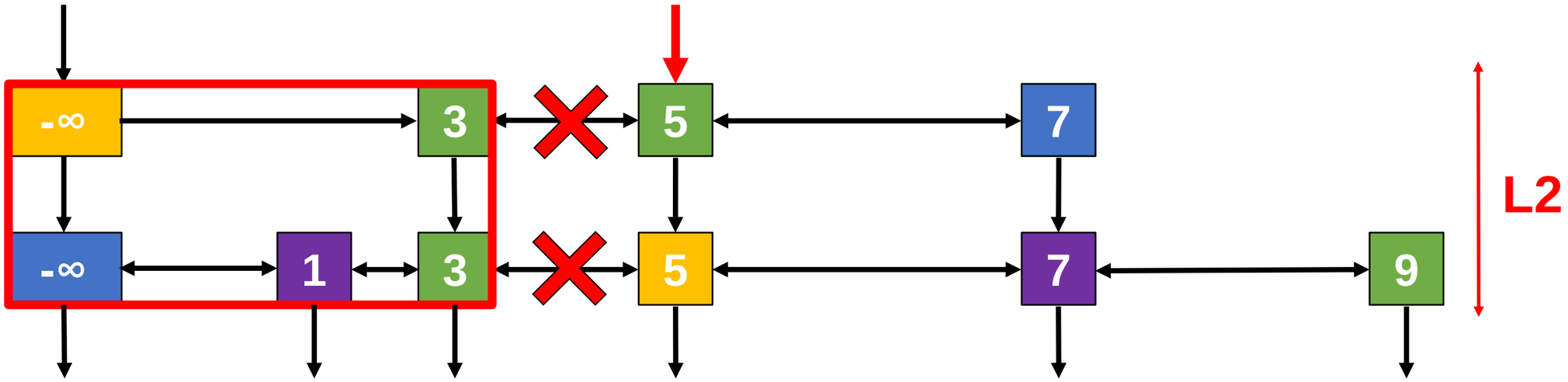
Shadow Subtree: Partial Replication

- Rebuild vertical locality in lower part by **partial** replication.
  - Due to space constraint and update cost.
- Basic Idea: L2 nodes cache their descendant subtree.



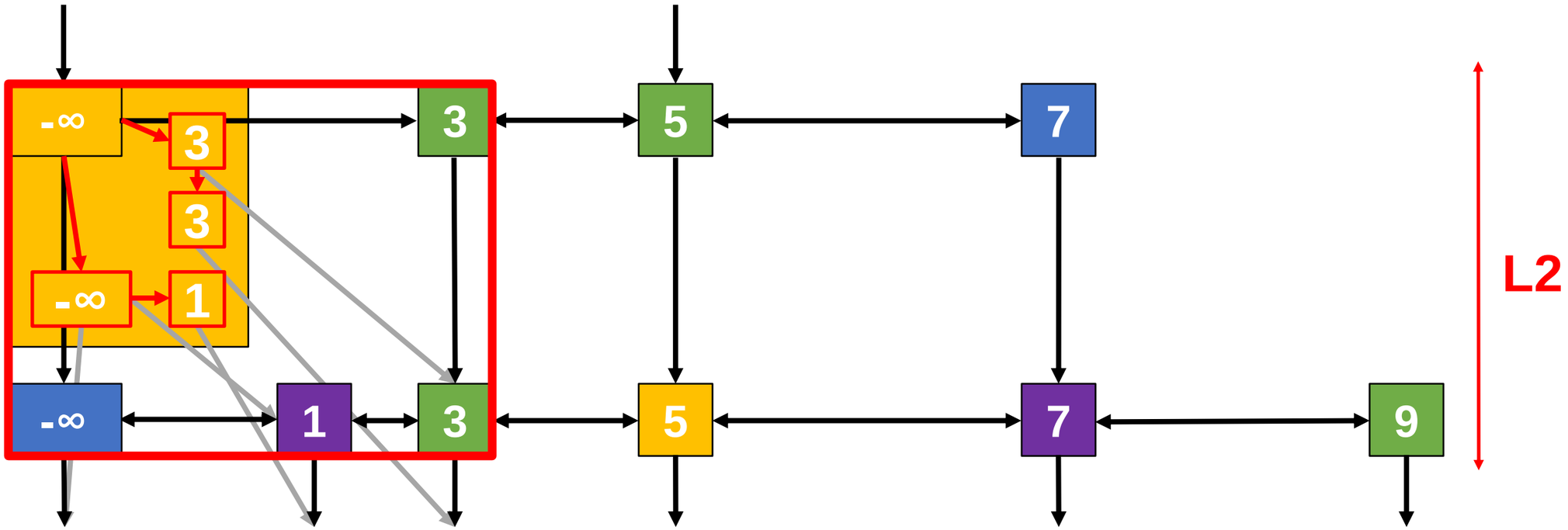
Shadow Subtree: Partial Replication

- Rebuild vertical locality in lower part by **partial** replication.
  - Due to space constraint and update cost.
- Basic Idea: L2 nodes cache their descendant subtree.



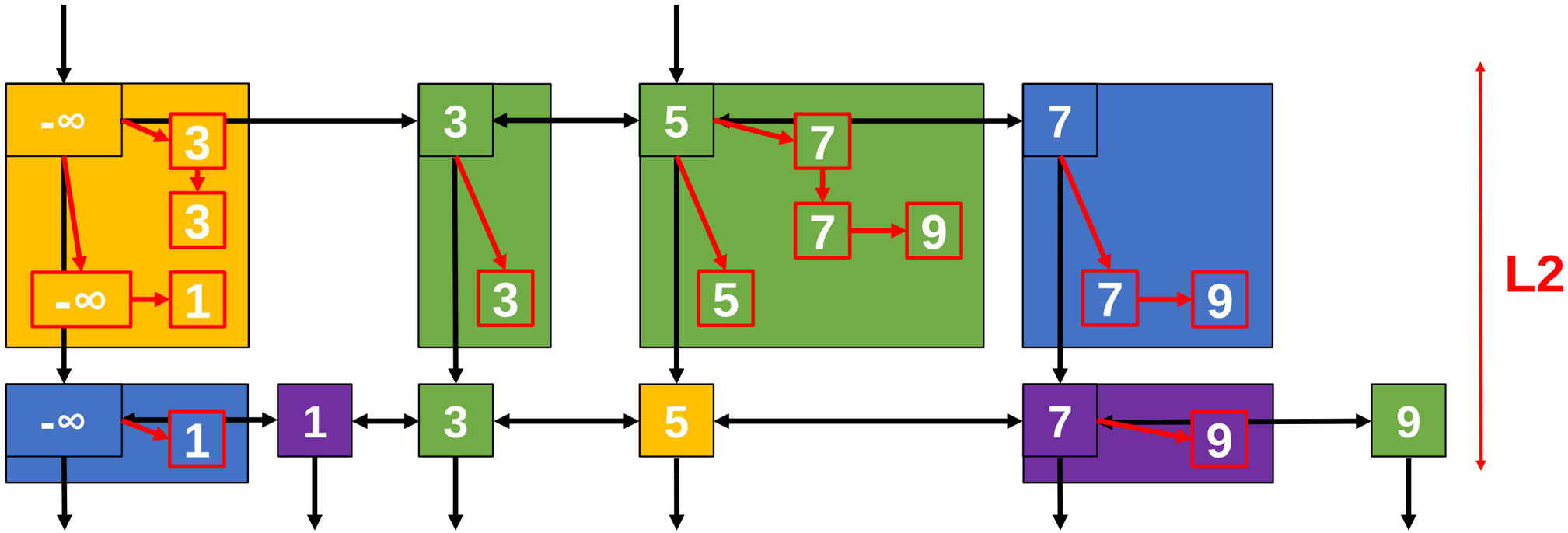
Shadow Subtree: Partial Replication

- Rebuild vertical locality in lower part by **partial** replication.
  - Due to space constraint and update cost.
- Basic Idea: L2 nodes cache their descendant subtree.



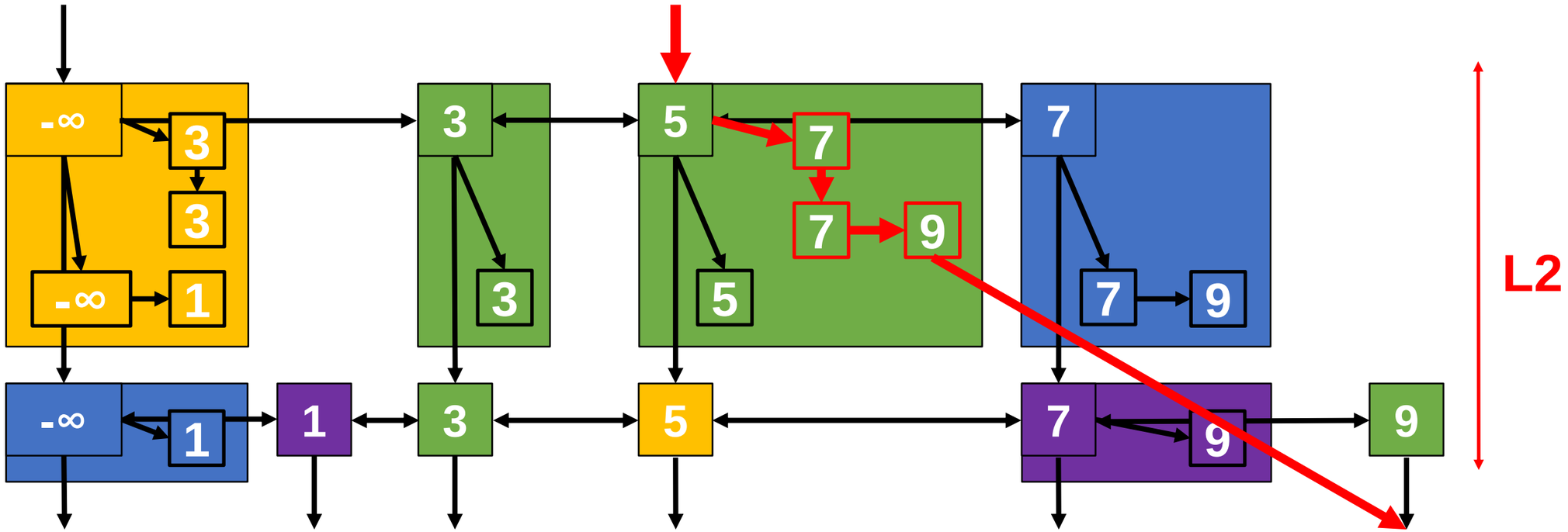
Shadow Subtree: Partial Replication

- Rebuild vertical locality in lower part by **partial** replication.
  - Due to space constraint and update cost.
- Basic Idea: L2 nodes cache their descendant subtree.



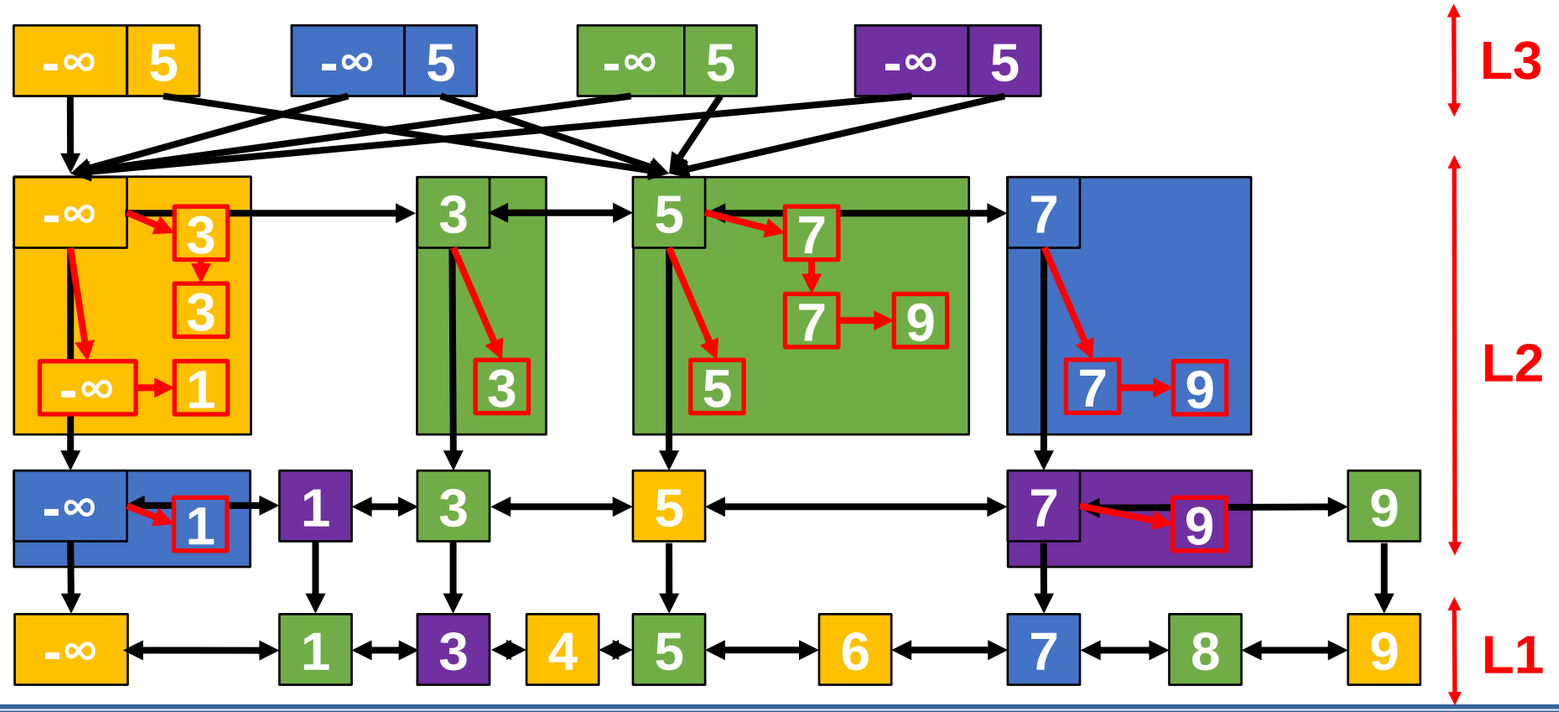
Shadow Subtree: Partial Replication

- Rebuild vertical locality in lower part by **partial** replication.
  - Due to space constraint and update cost.
- Basic Idea: L2 nodes cache their descendant subtree.



PIM-tree Structure: Three-layer

- Three-layer structure
  - Upper part (L3): **Fully replicated** in all PIM modules.
  - Middle part (L2): **Partially replicated** by Shadow subtree.
  - Lower part (L1): **Distributed** to random PIM modules.



Results of the Three-layer Structure

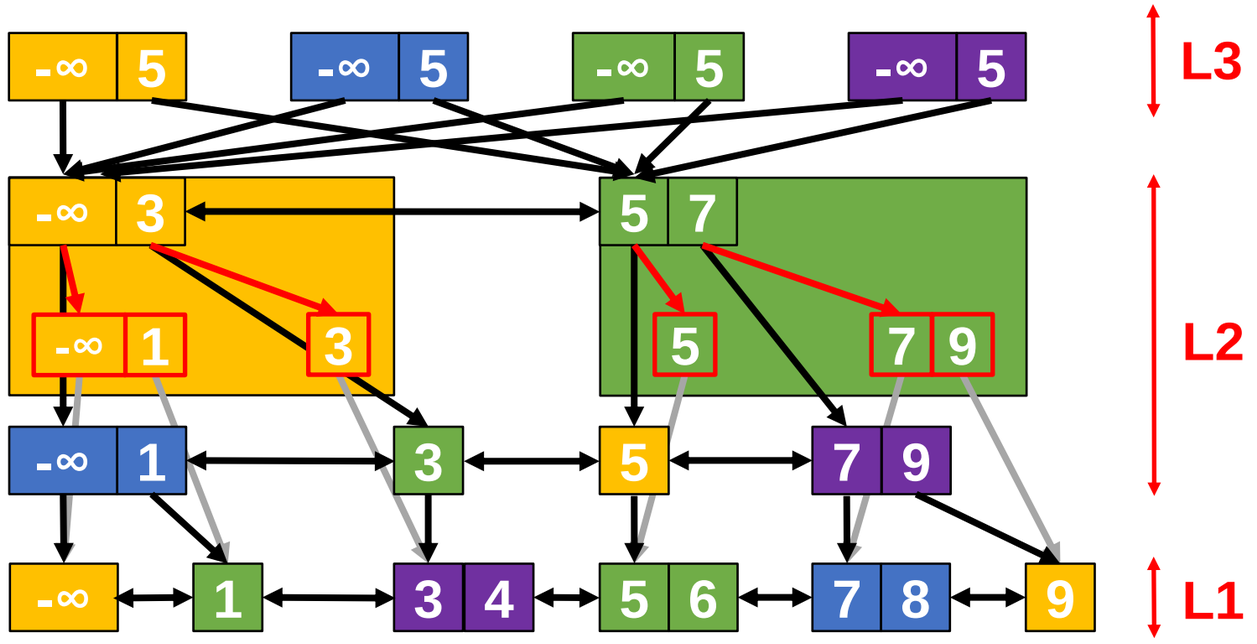
	L3	L2	L1
Height	$\log n - \log P$	$\log P - \log\log P$	$\log\log P$
Replication Strategy	Full Replication	Shadow Subtree	Distributed
Search Comm. (in words)	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(\log\log P)}$
# of Nodes in this layer	$\Theta(n/P)$	$\Theta(n/\log P)$	$\Theta(n)$
# of Replicas per Node	$O(P)$	$O(\log P)$	$O(1)$
Space	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Update Frequency	$1/P$	$1/\log P$	1
Update Comm. (post search)	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$

Results of the Three-layer Structure

	L3	L2	L1
Height	$\log n - \log P$	$\log P - \log\log P$	$\log\log P$
Replication Strategy	Full Replication	Shadow Subtree	Distributed
Search Comm. (in words)	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(\log\log P)}$
# of Nodes in this layer	$\Theta(n/P)$	$\Theta(n/\log P)$	$\Theta(n)$
# of Replicas per Node	$O(P)$	$O(\log P)$	$O(1)$
Space	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Update Frequency	$1/P$	$1/\log P$	1
Update Comm. (post search)	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$

Chunking : Further Horizontal Locality

- In the lower part, merge horizontal nodes chunks. stored on one PIM module.
- Cut all <u>horizontal</u> inter-module communication.
- Expected chunk size: B
  - Typically $B = 16$



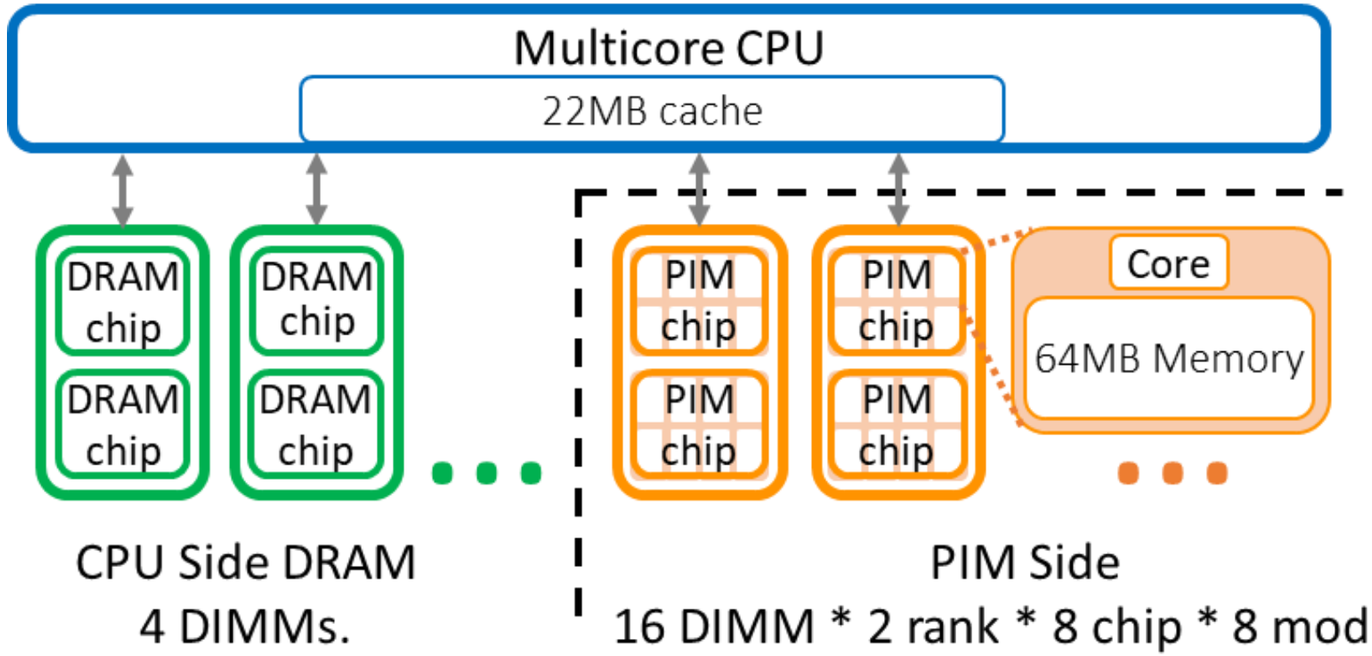
Results in The PIM Model

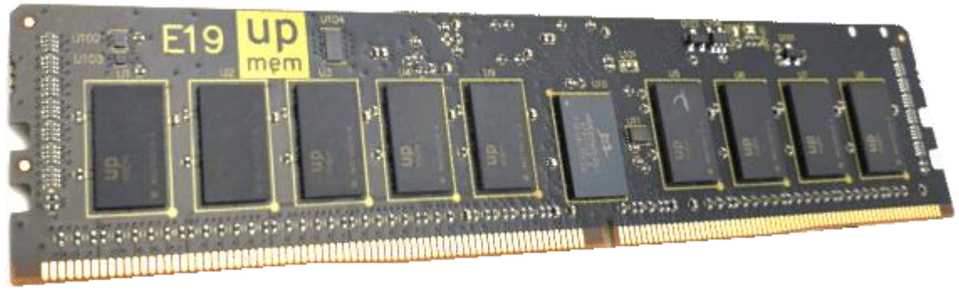
- Load-balance guaranteed* against adversary
- Reduced Communication <u>regardless of skew</u>
  - $\boldsymbol{O(\log_B \log_B P)}$ per operation for both search and update
  - 3 or 4 communication rounds in practice
- Linear space consumption $\Theta(n)$

* With high probability if the batch size is large enough.

Experimental Platform

- Upmem tech: https://www.upmem.com



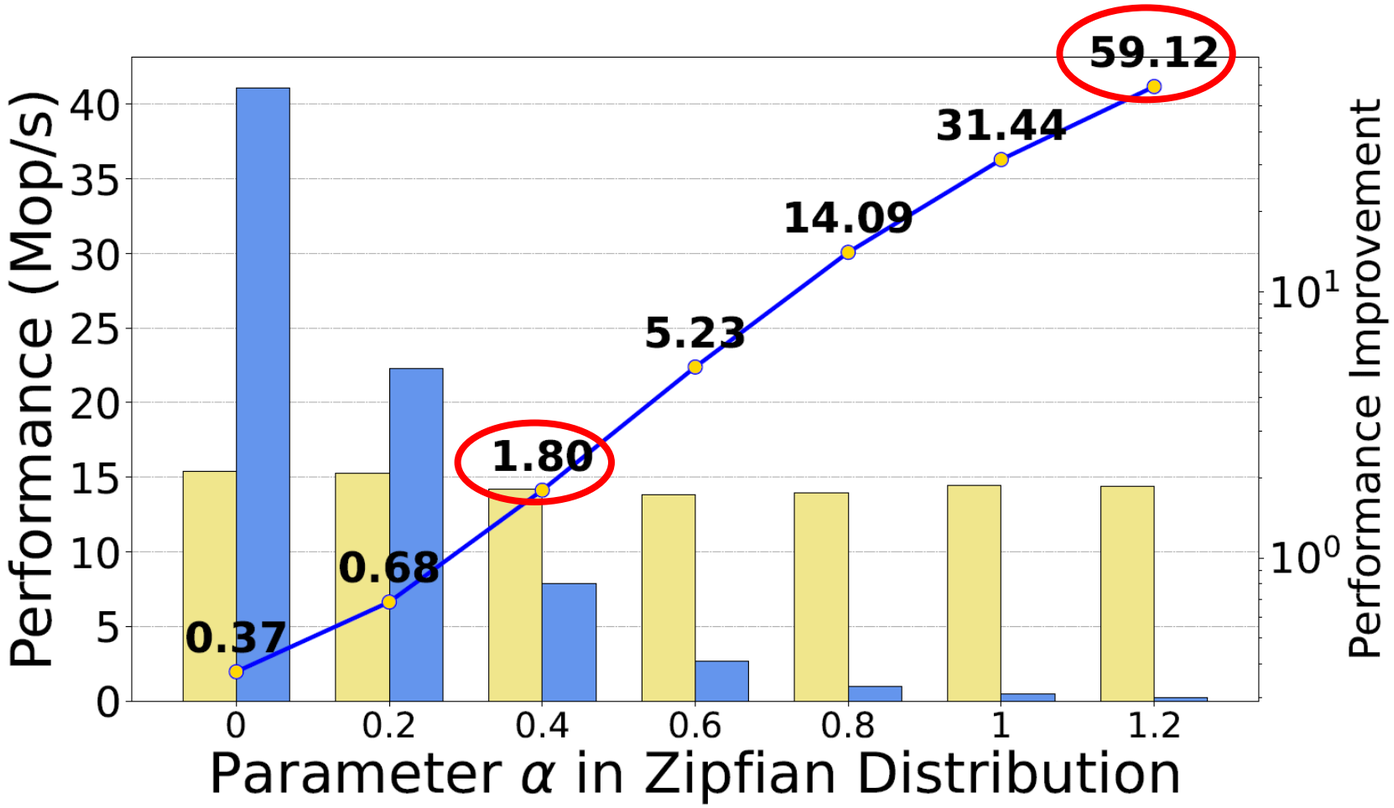


256GB in total.

Each PIM core is a 32-bit RISC processor @450MHz.

In this architecture, scheduling of PIM modules are done by the user program.

Results Comparing with Prior PIM-based Indexes



Predecessor() **Throughput**
on Zipf workloads

PIM-Tree

Partitioned Skip List

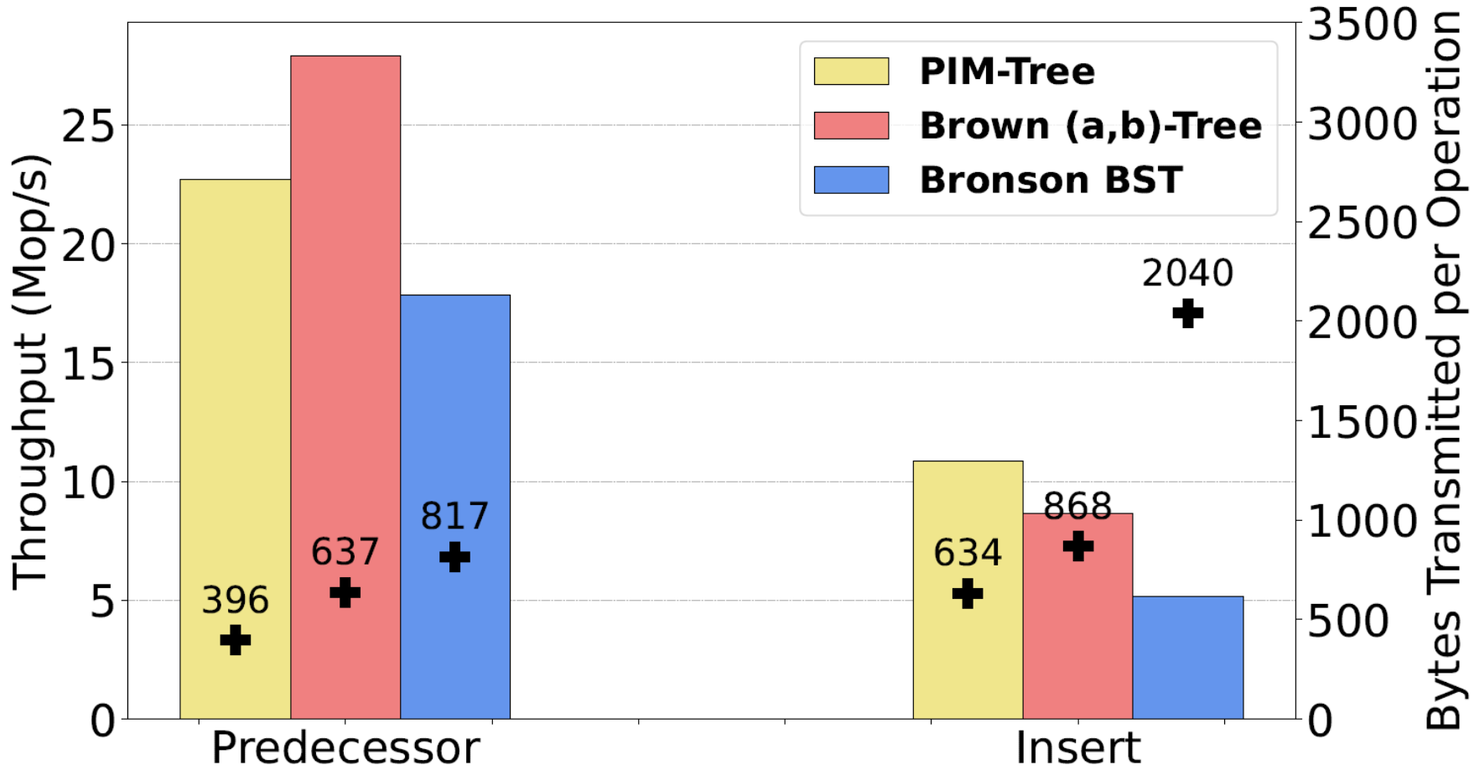
Improvements of PIM-Tree over Partitioned Skip List

Up to 59X improvement !

Results Comparing with CPU-based Indexes

Experiments on Wikipedia dataset Over SOTA Shared-memory Indexes

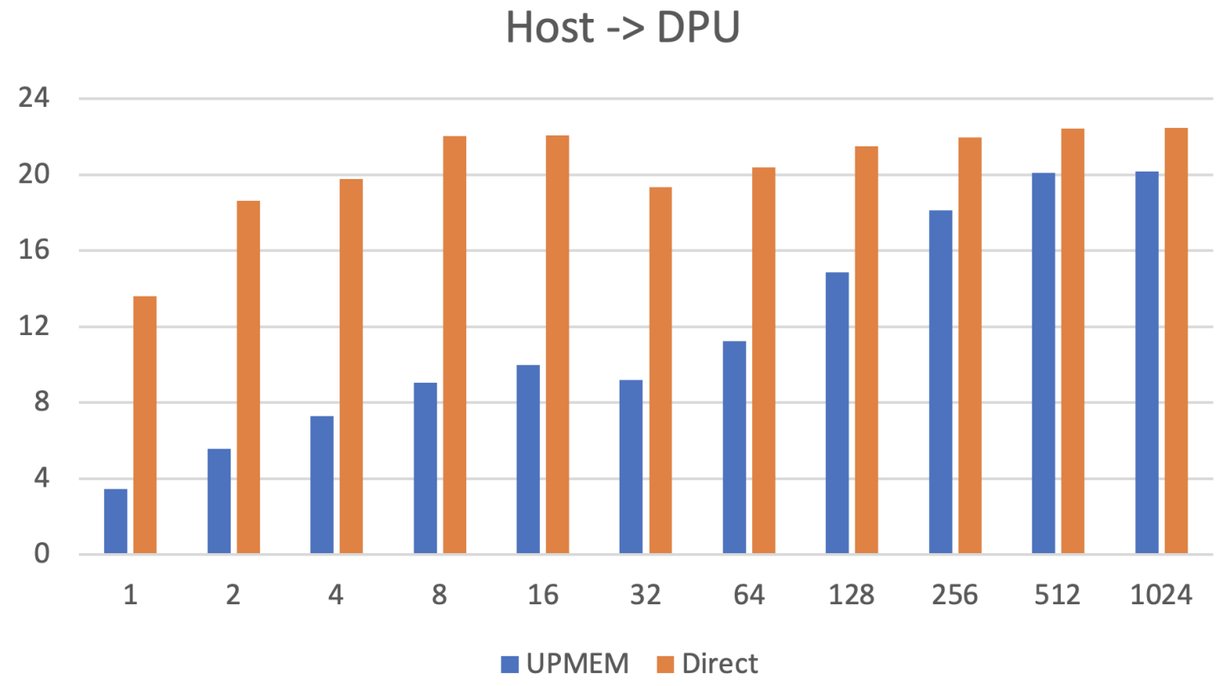
* Bars are throughputs; '+' are communications

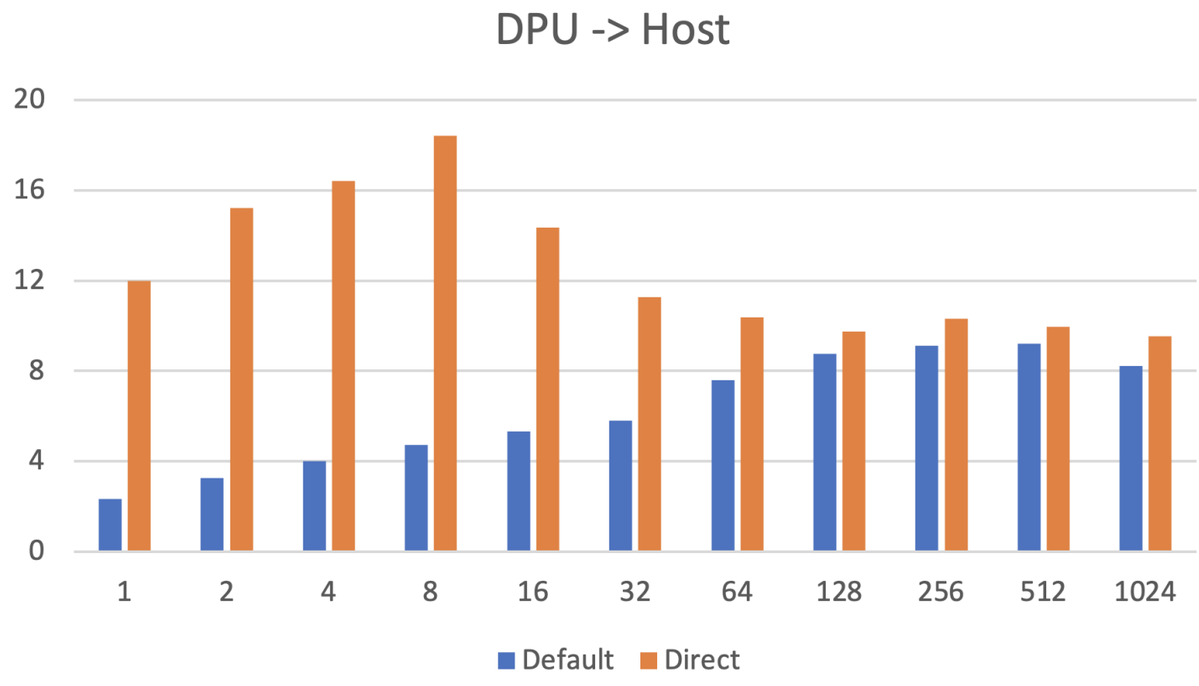


Less communication, faster inserts

New Early-Stage Results in Bandwidth

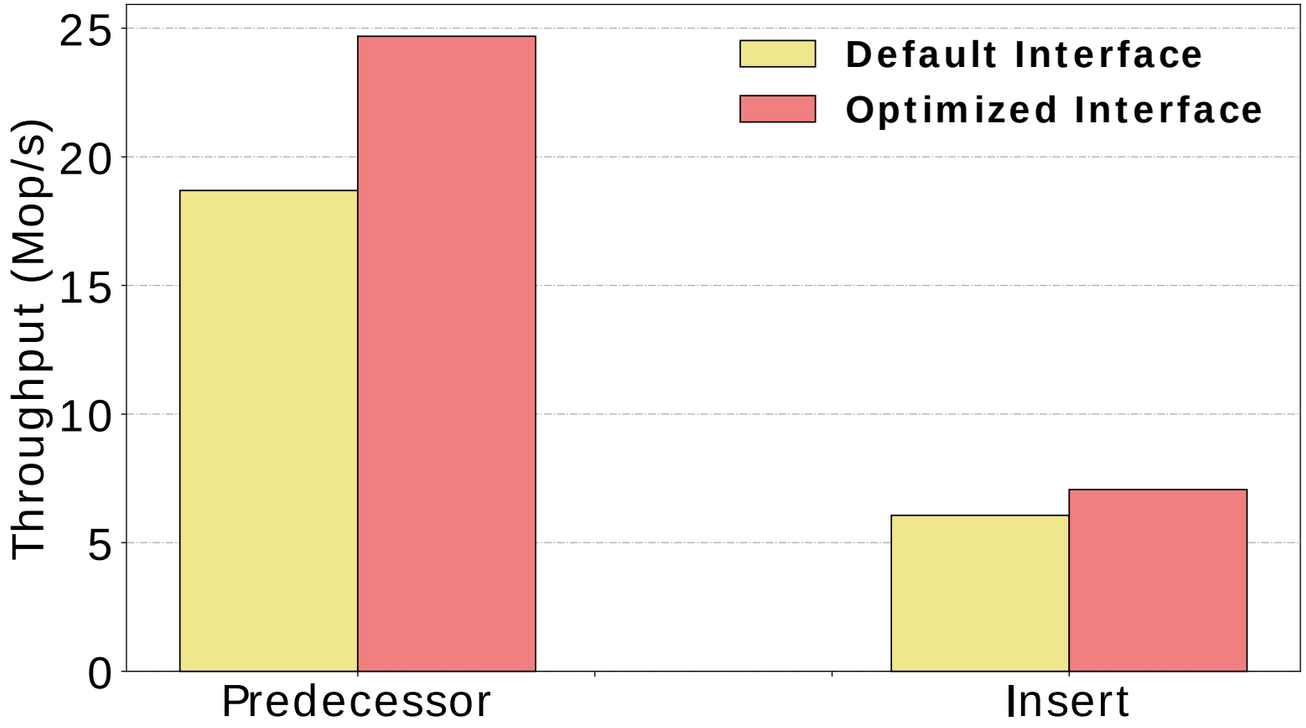
- Optimize communication for small batches
- Scatter/Gather buffers of 1~1024 KB with all DPUs
  - 32 ranks, 2000 PIM modules functioning





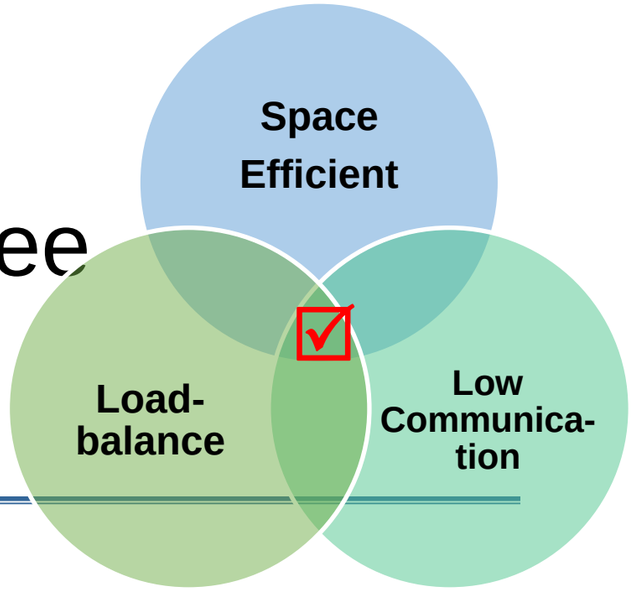
New Early-Stage Results in Throughput

- Applying new interface on Wikipedia dataset
  - 32 ranks, 2000 PIM modules functioning
  - upmem-sdk-light
    @ github

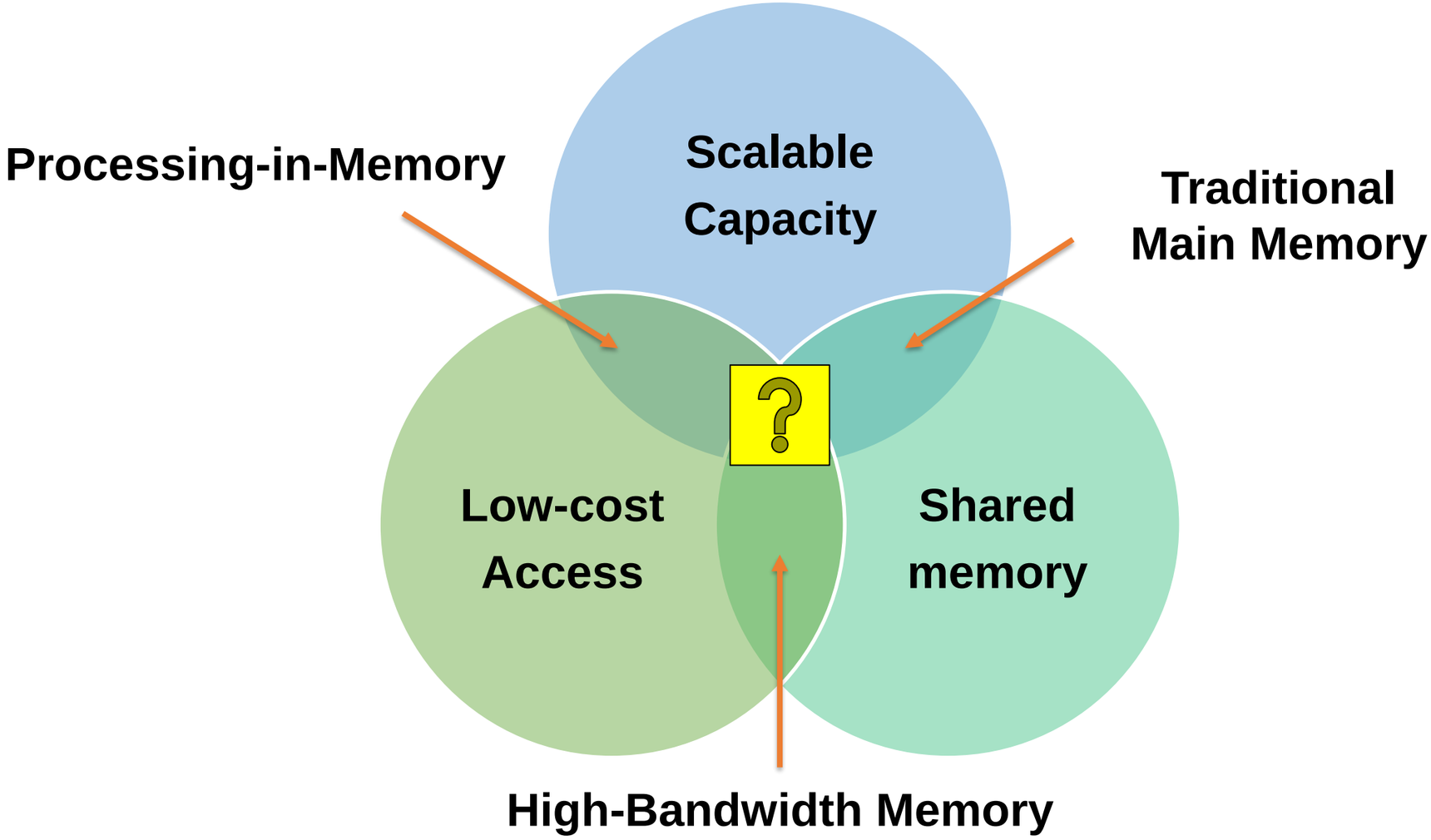


Key Takeaways

- PIM emerging as means of reducing data movement
- Prior works of PIM indexes vulnerable to skew
- PIM-tree is a skew-resistant database index
  - Three-level Structure: Different replication strategies
  - Push-pull search: Lightweight load balance
  - Chunking: Further reduction on communication
  - Provable and practical efficiency
- Codes: https://github.com/cmuparlay/PIM-tree



Ideal Memory



B-tree? Skip lists?

- Both are used in practice
  - Skip lists used in the memory part for LSM trees
- We believe our optimizations apply to both
  - B-tree can be interesting future work.
- In our implementation
  - L3 is implemented by a b-tree
  - L2 and L1 are implemented by skip lists
    - Enables easier parallel inserts/deletes