

PICCOLO

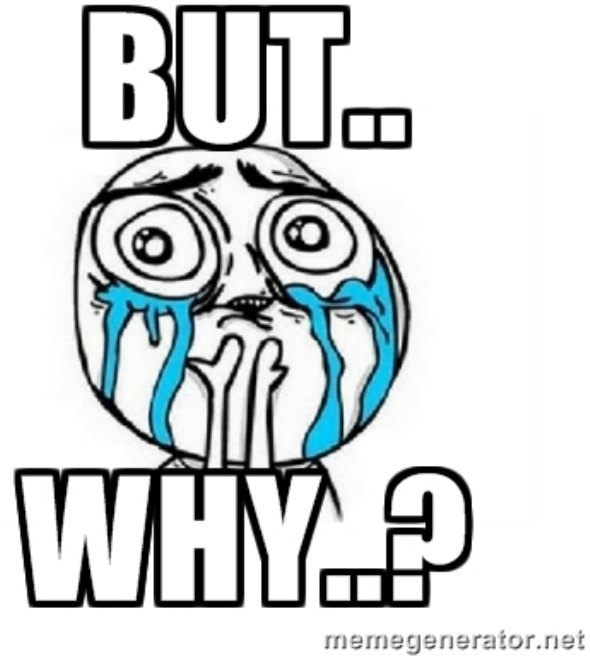
BUILDING FAST, DISTRIBUTED PROGRAMS
WITH PARTITIONED TABLES

Russell Power, Jinyang Li
New York University

OSDI 2010

What is PICCOLO?

- A data-centric programming model for applications that
 - Are distributed
 - Are **in-memory**
 - Access and mutate some shared intermediate state.
- Allows users to specify:
 - How is data partitioned?
 - Locality Policies



- MPI requires too much work!
 - Fine-grained control.
- Typical data-centric programming models
 - Are good for bulk processing of on-disk data.
 - Not for in-memory applications?
 - Read 1MB sequentially from memory – 0.00025 ms
 - Read 1MB sequentially from network – 0.01000 ms
 - Read 1MB sequentially from disk – 0.03000 ms
- Can we do better?
 - Yes, let the user figure out certain details.
 - But.. Why? Better performance.

Repeat until
convergence

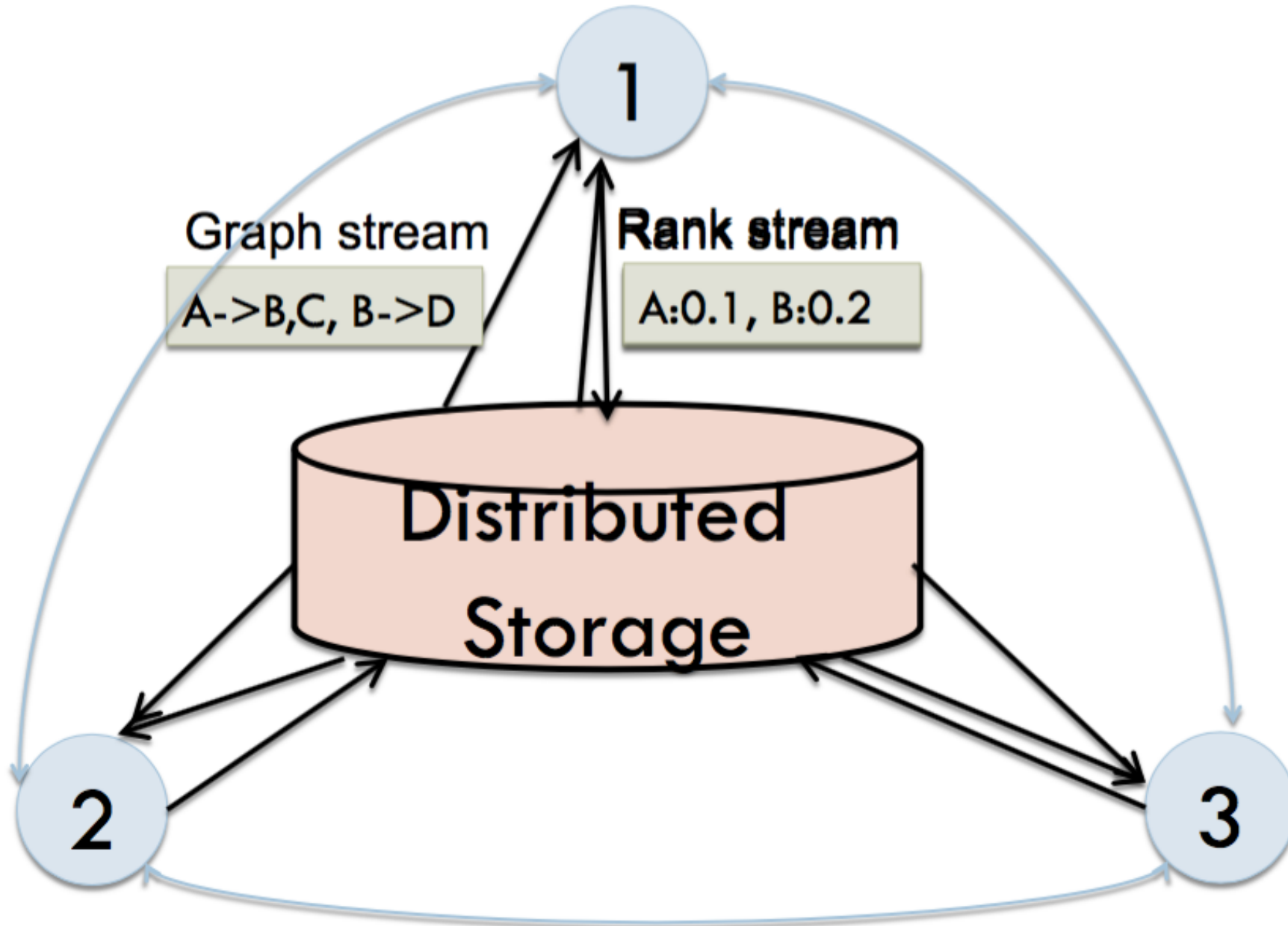
for each node X in graph:
for each edge $X \rightarrow Z$:
 $\text{next}[Z] += \text{curr}[X]$

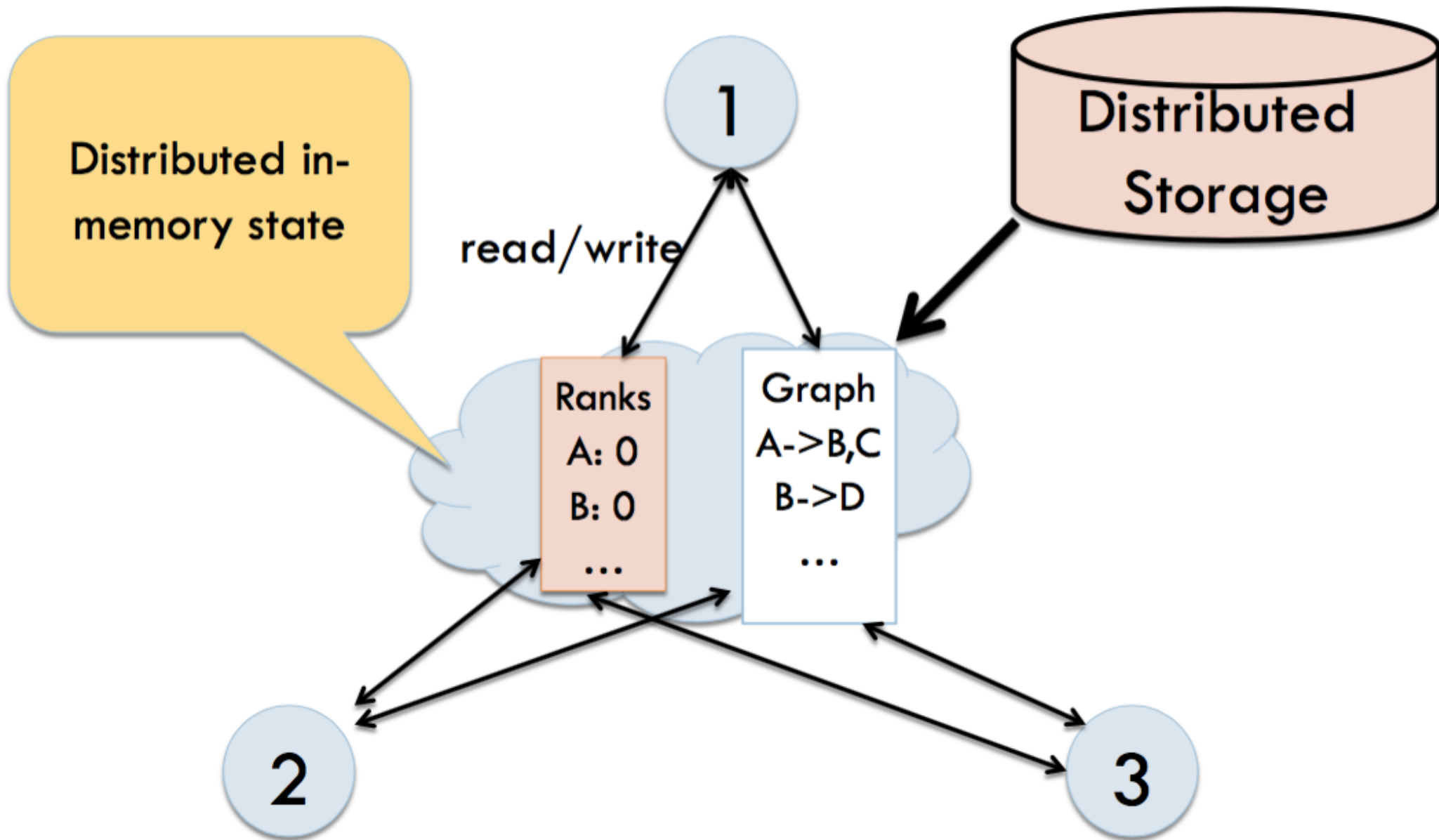
Input Graph

$A \rightarrow B, C, D$
$B \rightarrow E$
$C \rightarrow D$
...

Curr	Next
A: 0.25	A: 0.25
B: 0.17	B: 0.17
C: 0.22	C: 0.22
...	...

Fits in
memory!





Programming Model

- **Control** functions
 - Launch kernels
 - Create tables
 - Synchronize through barriers
 - Runs on one machine
- **Kernel** functions
 - Distributed, many instances are run together
 - Read and write to tables

Tables

- Key-value stores
 - Get(Key)
 - Put(Key, Value)
 - Update(Key, Value)
 - Flush()
- User defined accumulation functions
 - Commutative, Associative
 - Local – no access to global state.
 - Deal with write-write conflicts.
 - But.. Why? Write changes are buffered.


```
curr = Table(key=PageID, value=double)
next = Table(key=PageID, value=double)
```

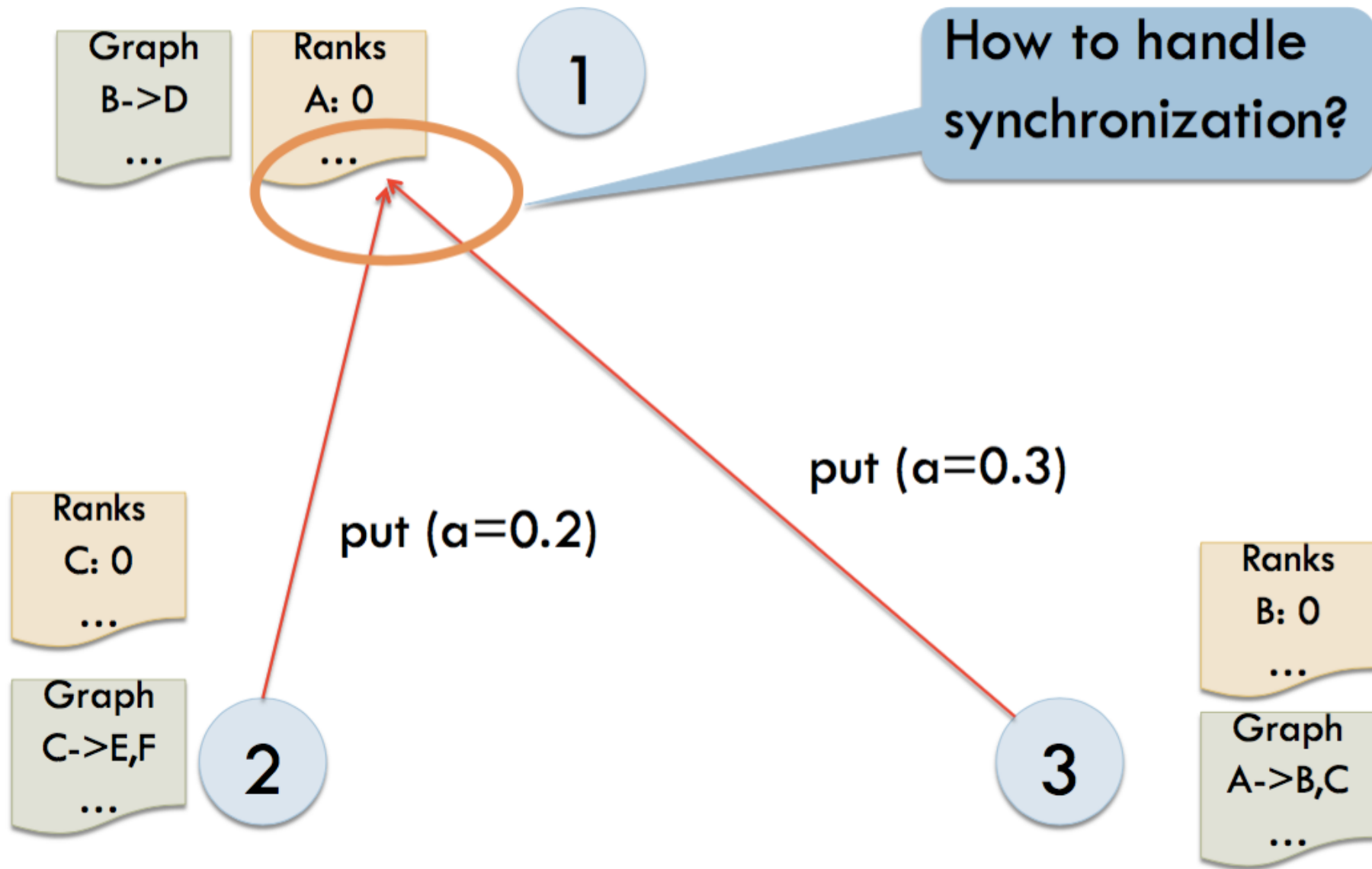
```
def pr_kernel(graph, curr, next):
    i = my_instance
    n = len(graph)/NUM_MACHINES
    for s in graph[(i-1)*n:i*n]:
        for t in s.out:
            next[t] += curr[s.id] / len(s.out)
```

```
def main():
    for i in range(50):
        launch_jobs(NUM_MACHINES, pr_kernel,
                    graph, curr, next)
        swap(curr, next)
        next.clear()
```

Jobs run by
many machines

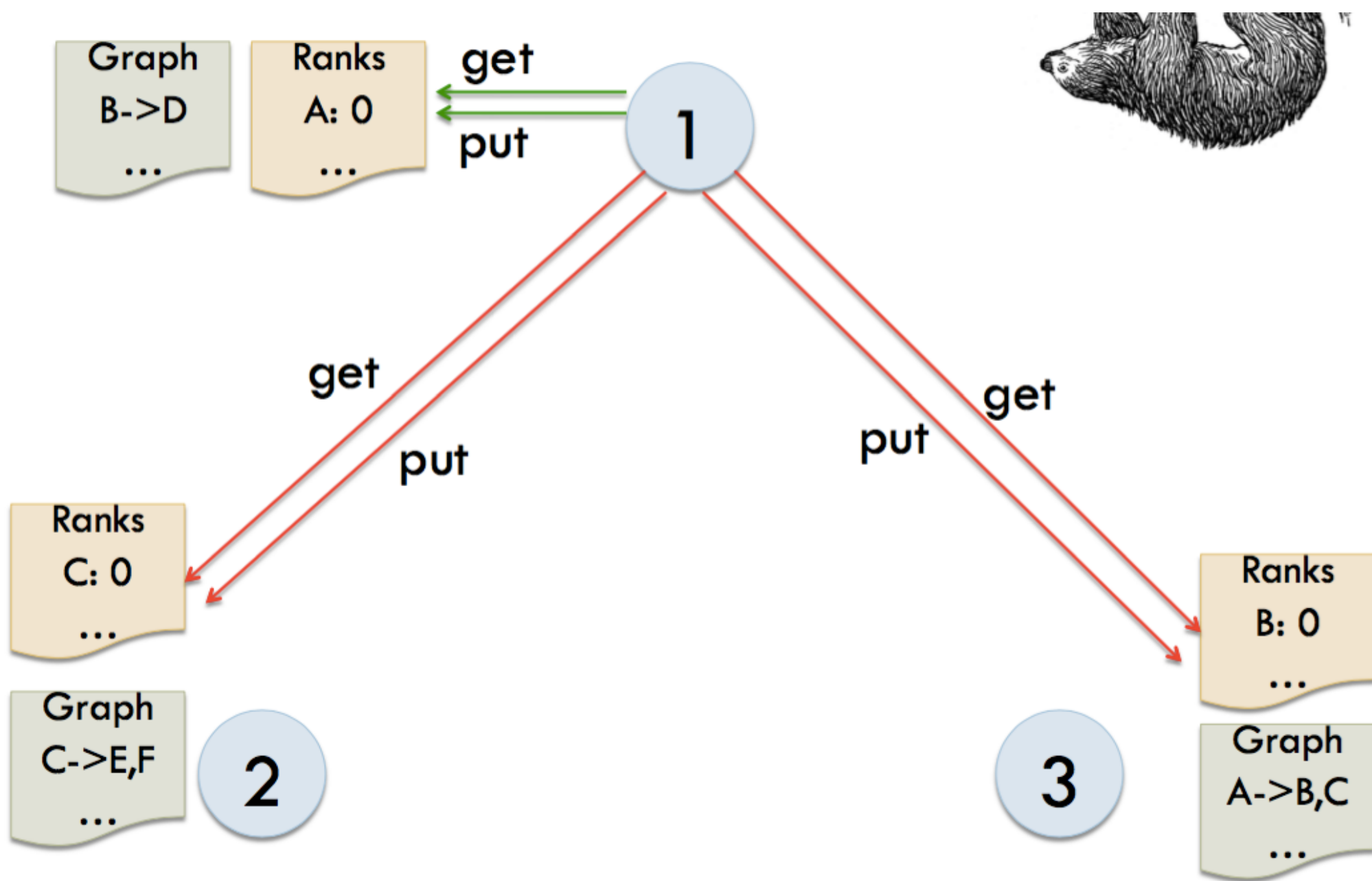
Controller launches
jobs in parallel

Run by a single
controller



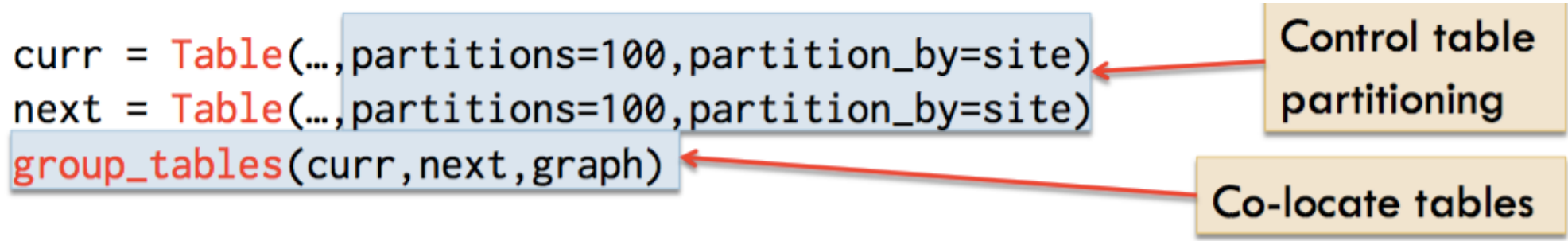
Partitioning and Locality

- Tables can be partitioned.
 - Assume each fragement fits in memory.
- Locality Preferences
 - Co-locate certain paritions of different tables.
 - Co-locate data and execution.



```
curr = Table(..., partitions=100, partition_by=site)
next = Table(..., partitions=100, partition_by=site)
group_tables(curr, next, graph)
```

Control table
partitioning

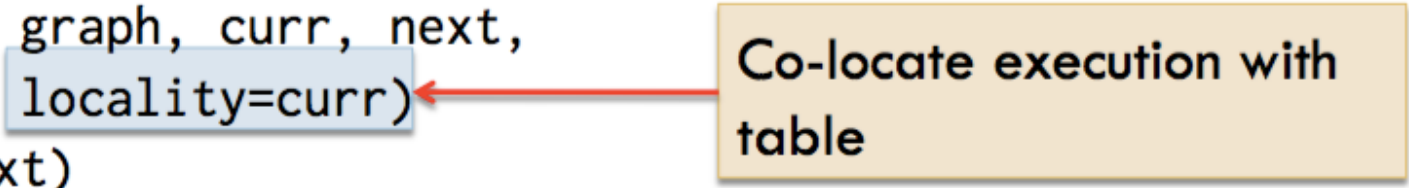


Co-locate tables

```
def pr_kernel(graph, curr, next):
    for s in graph.get_iterator(my_instance):
        for t in s.out:
            next[t] += curr[s.id] / len(s.out)
```

```
def main():
    for i in range(50):
        launch_jobs(curr.num_partitions,
                    pr_kernel,
                    graph, curr, next,
                    locality=curr)
    swap(curr, next)
    next.clear()
```

Co-locate execution with
table




Checkpoints

- Asynchronous
 - Takes as arguments – Tables, User-Data
- Synchronous
 - Takes as arguments – Time Interval, Tables, Callback


```
curr = Table(...,partition_by=site,accumulate=sum)
next = Table(...,partition_by=site,accumulate=sum)
group_tables(curr,next,graph)
```

Accumulation
via sum




```
def pr_kernel(graph, curr, next):
    for s in graph.get_iterator(my_instance)
        for t in s.out:
            next.update(t, curr.get(s.id)/len(s.out))
```

Update invokes
accumulation function



```
def main():
    for i in range(50):
        handle = launch_jobs(curr.num_partitions,
                               pr_kernel,
                               graph, curr, next,
                               locality=curr)
        barrier(handle)
        swap(curr, next)
        next.clear()
```

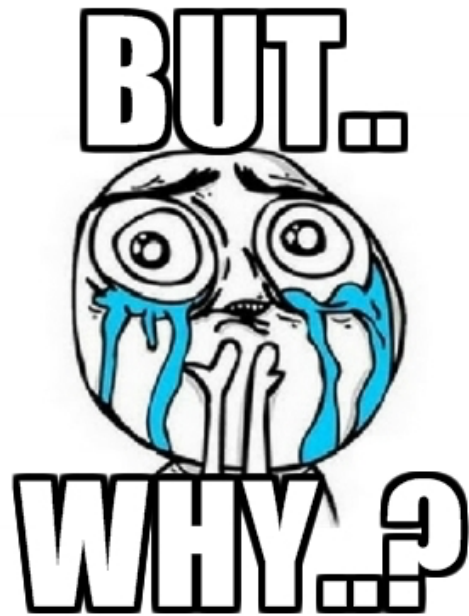
Explicitly wait
between iterations



System Design

- **Master** node
 - Control Thread
 - Assigns Kernels to Workers
 - The assignment is a “public announcement”.
- **Worker** nodes
 - Handle Kernel executions
 - Principle: Buffer as long as you can.
 - If need to read, then flush(), and read().
 - Why? Want that for a single thread, the chronology makes sense.

Load Balancing



memegenerator.net

- Initial allocation
 - Do round-robin.
 - If there is a distributed file, minimize inter-rack transfer.
- Dynamic Load Balancing
 - But .. Why?
 - Heterogeneous hardware configs
 - How?
 - Kill no running task.
 - Have to migrate table partitions.

Work Stealing

- If a worker is free, assign it a task from the busiest worker.
- Do larger tasks first.
 - Estimate task size by size of partition of table.



Table Partition Migration (OLD to NEW)

- Phase 1—
 - Master says BEGIN.
 - All workers flush changes to OLD, send new requests to NEW.
 - OLD pauses, relay requests to NEW. Then, transfer state.
 - NEW buffers requests, does not act.
 - ACKs from all.
- Phase 2—
 - Master says DO_IT_NOW to OLD and NEW.
 - All workers flush changes to OLD, send new requests to NEW.
 - OLD pauses, relay requests to NEW.
 - NEW buffers requests, does not act.
 - Now, OLD sends requests.
 - NEW now acts.

Fault Tolerance

- If one fails, restart all from the last checkpoint.
- Check-point
 - Need to save a consistent checkpoint without stopping the execution.
 - Chady-Lamport algorithm
 - Idea: Take a snapshot of state. Keep a log of changes.
 - When to do it?
 - Early. Log could be too large.
 - Late. Missed opportunity to concurrently do execution and checkpoint.
 - Do it when the first worker is done.

Experiments

- On
 - 12 node NYU network.
 - EC2
- Implements
 - PageRank – co-locate rank and graph.
 - Distributed Crawler – co-locate “polite”, “robots”, “sites”
 - K-Means
 - N-Body Simulation
 - Matrix Multiplication



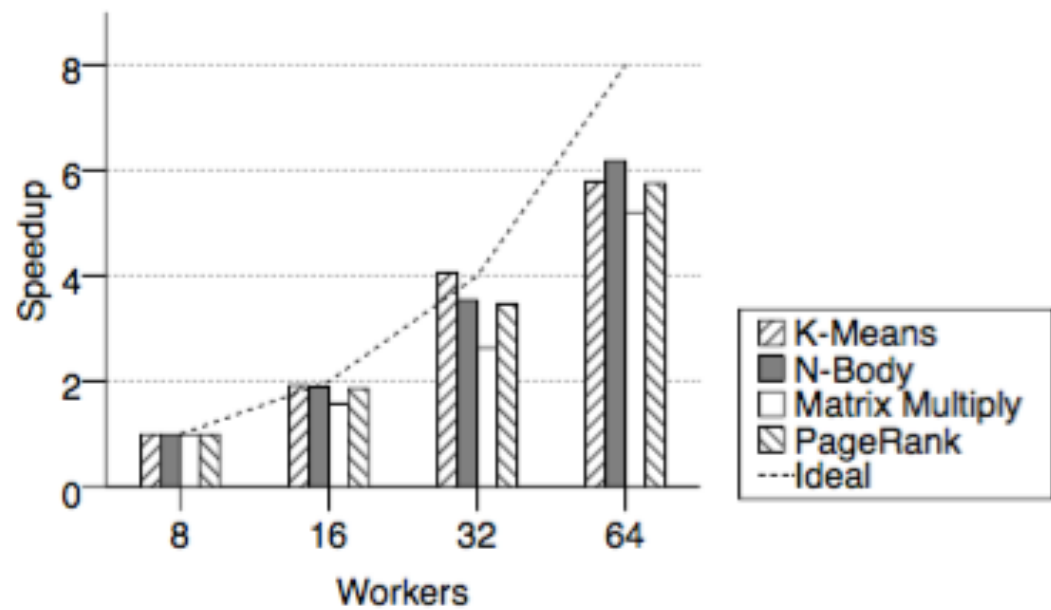


Figure 6: Scaling performance (fixed default input size)

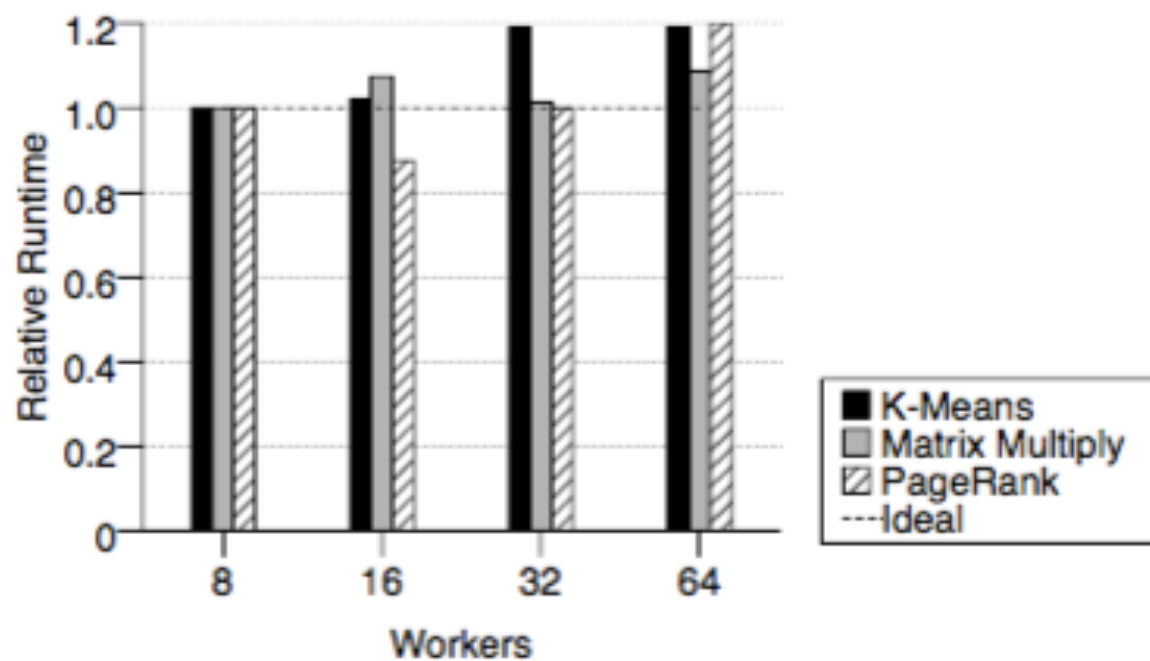


Figure 7: Scaling input size.

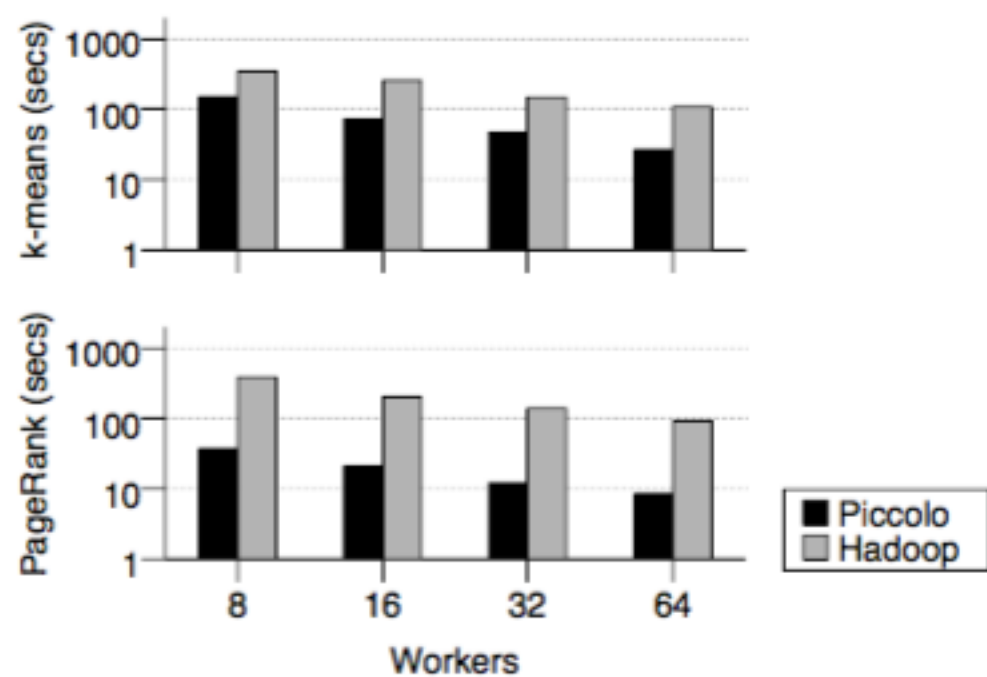


Figure 9: Per-iteration running time of PageRank and *k*-means in Hadoop and Piccolo (fixed default input size).

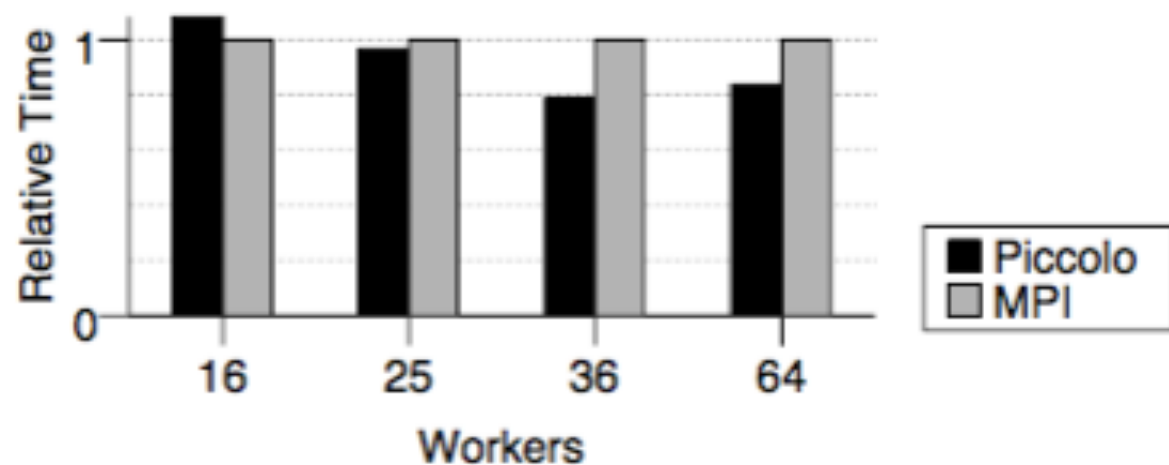


Figure 10: Runtime of matrix multiply, scaled relative to MPI.

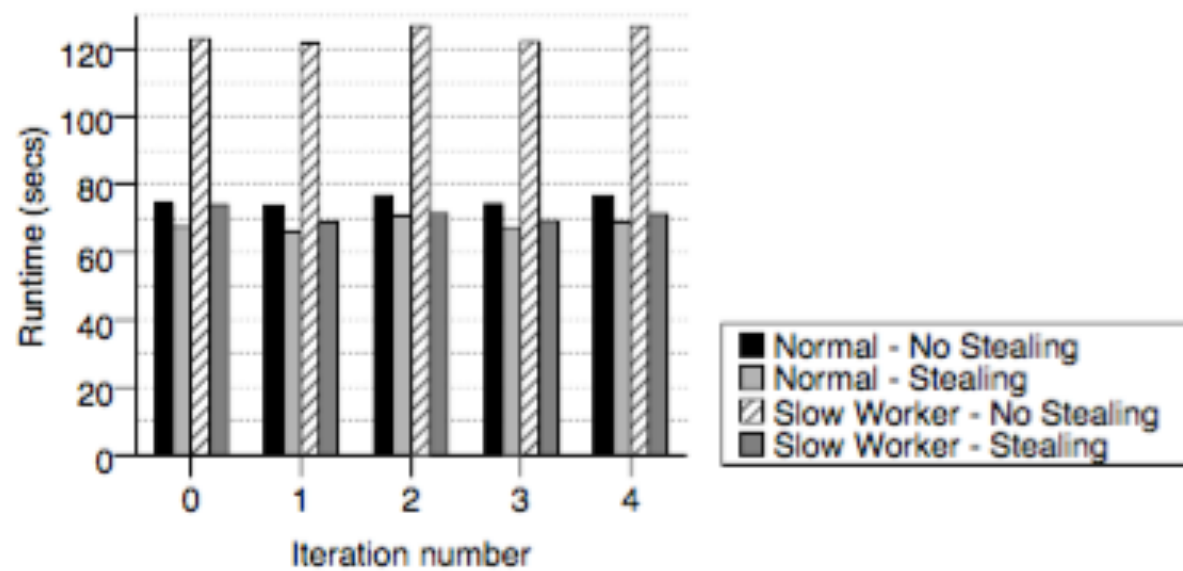


Figure 11: Effect of Work Stealing and Slow Workers

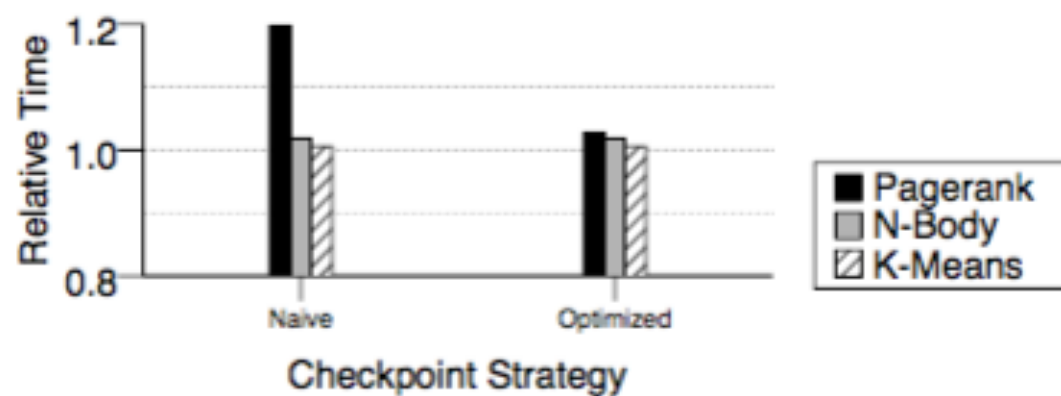


Figure 12: Checkpoint overhead. Per-iteration runtime is scaled relative to without checkpointing.

Strengths



- Natural model for some applications.
- Configurability allows for application tuning.
- 11x, 4x performance for PageRank, K-Means
- Not an all-purpose system; targets specific apps.
- Makes good choices as to what to delegate.
 - Checkpointing control variable is the user's job.

“Not-Strength”s



- The one-fails-all-do policy.
- Associative, Commutative accumulators?
- What if the master fails?
- Key-value interface. Multi-entry writes?
- Too reliant on the user?
- Checkpoint scalability?
- Compared to some other in-memory system?