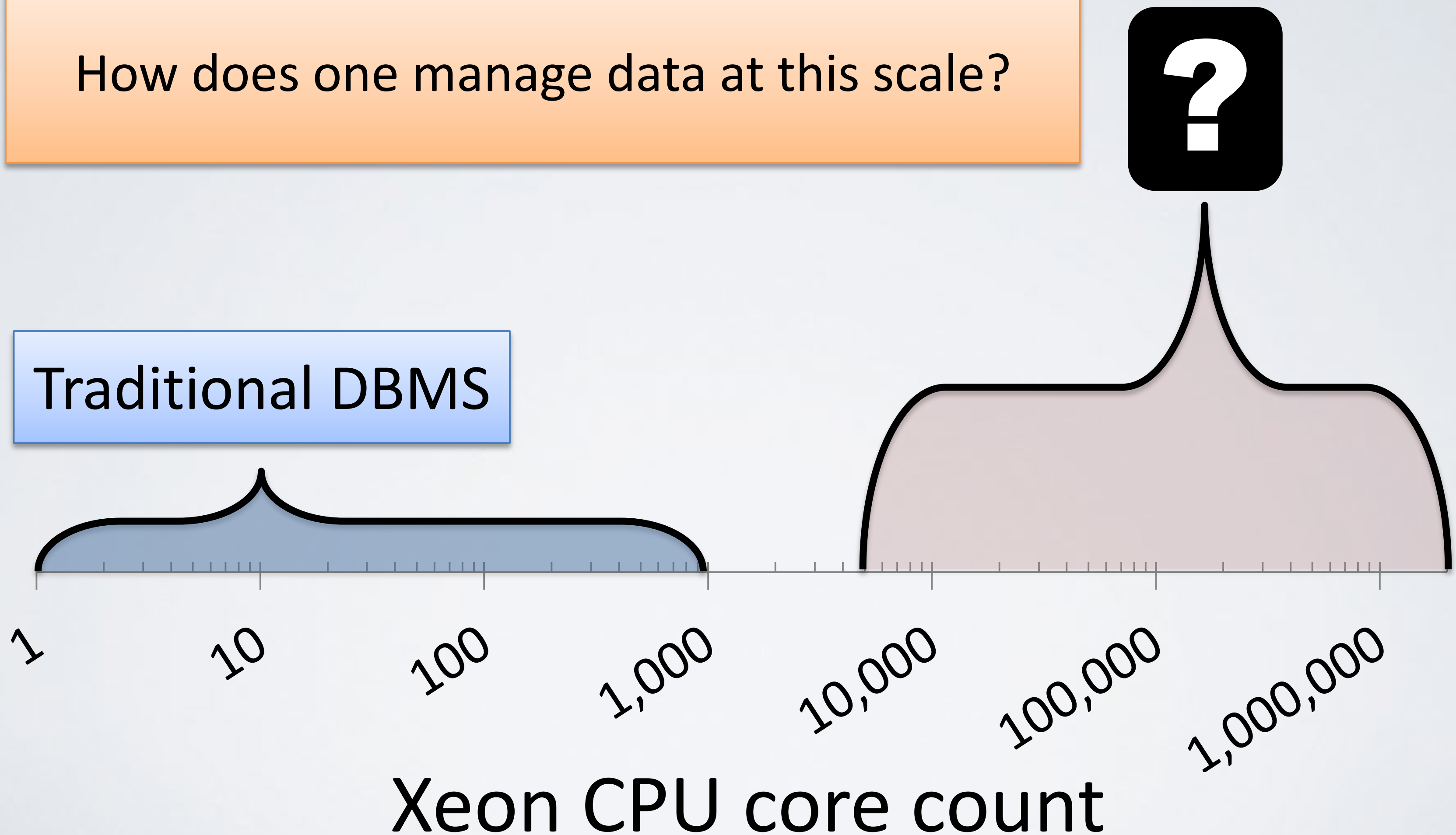
# Scaling database systems to high-performance computers

Spyros Blanas

THE OHIO STATE UNIVERSITY

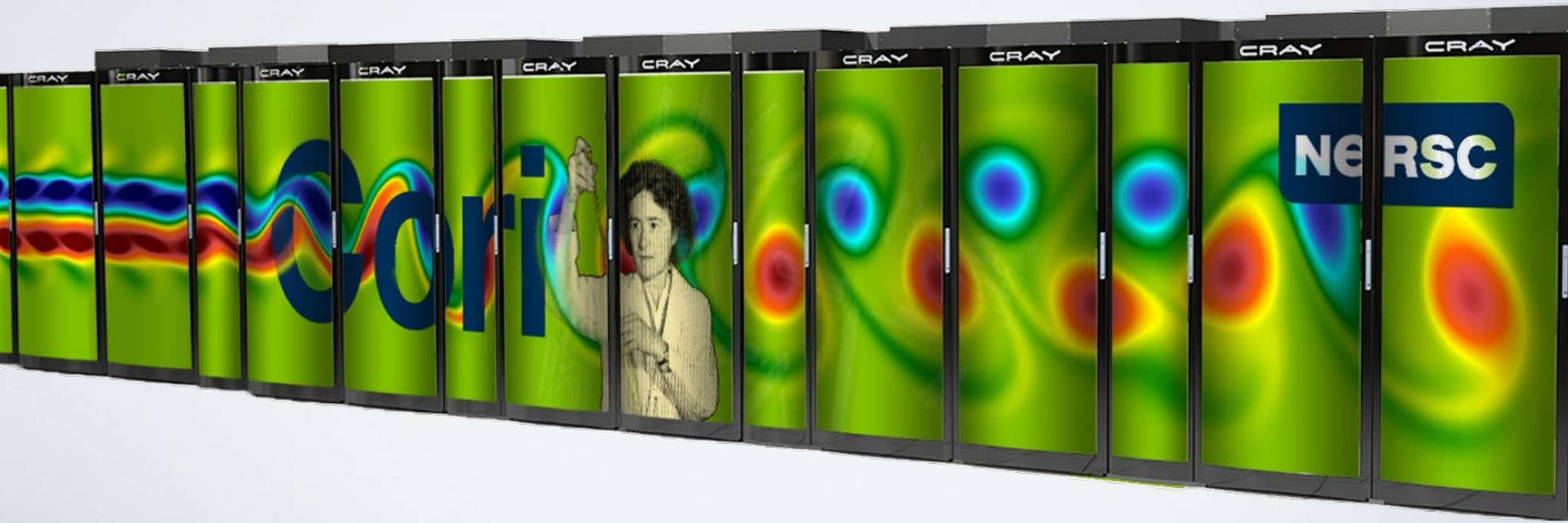
# Data processing at scale

How does one manage data at this scale?



# Data processing at scale

- Warehouse-scale computers
  - Scientific supercomputers
  - Cloud



# Data processing at scale

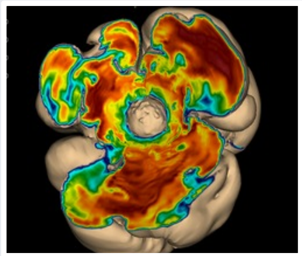
**Focus: Run one query as fast as possible on entire datacenter**

# Why not a database system?

**1** **Compute-intensive programs**

Supernovae detection

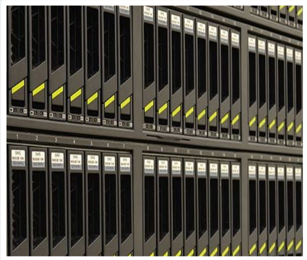
Disaster response



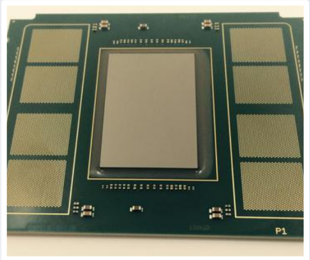
Plasma physics

Computational neuroscience

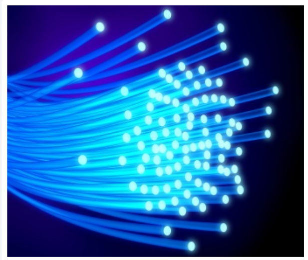
**2** **Scalability & efficiency**



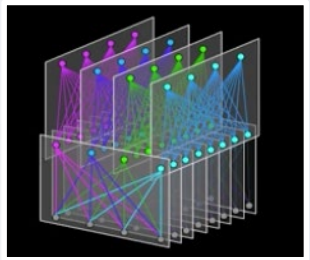
Massive I/O concurrency



3D-stacked DRAM

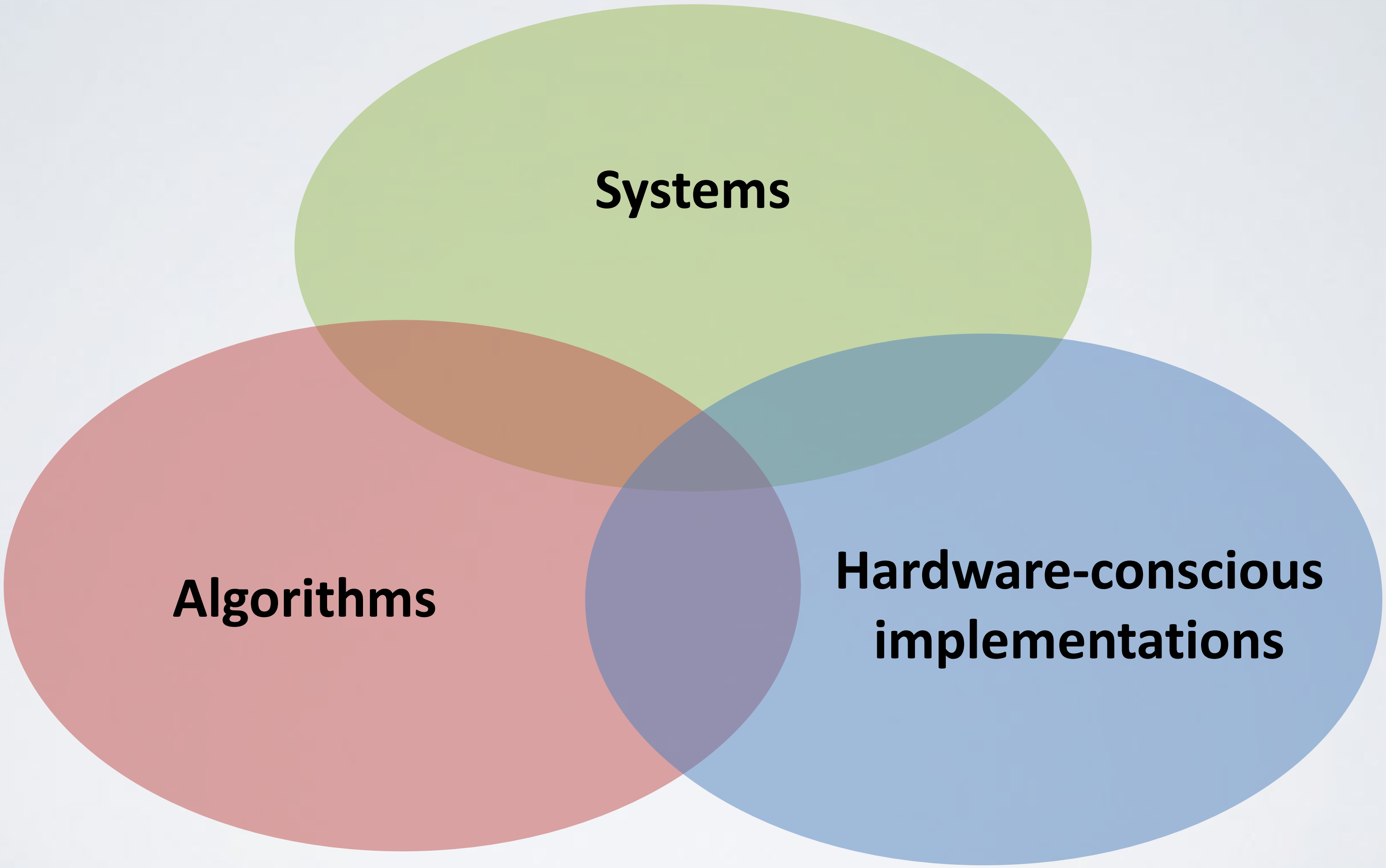


RDMA-capable networking



Unique network topologies

# Data processing at scale



# Take aways

❶ ArrayBridge: database processing over TB-sized HDF5 datasets

❷ GRASP: a network-aware **gr**eedy **a**ggregation **s**cheduling **p**rotocol based on dataset similarity

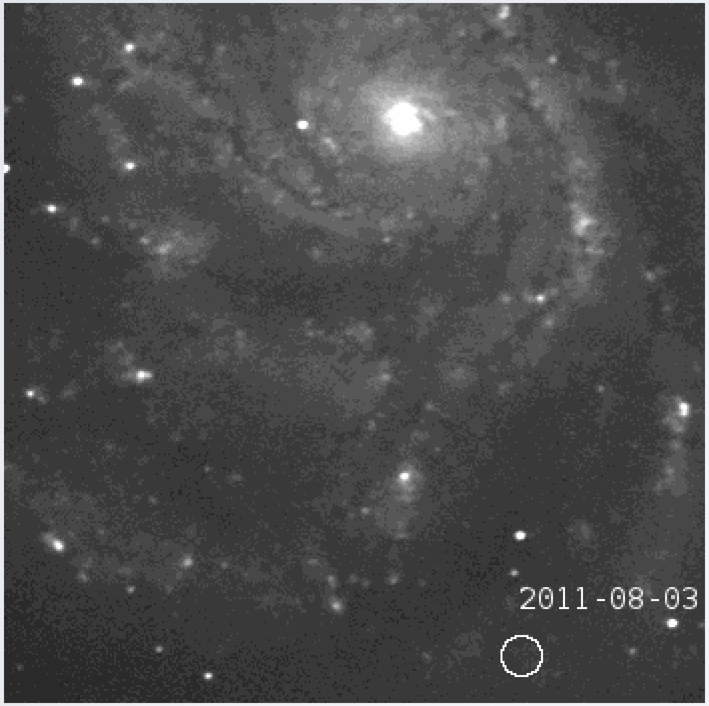
❸ An RDMA-based data shuffling operator exchanges data at line rate, 4× faster than MPI

# Supernovae detection pipeline

Telescope
↓
Image processing
↓
Image subtraction
↓
Classification

Image credit: PTF @ LBNL and ASAS-SN @ Ohio State

# Supernovae detection: Palomar Transient Factory (PTF)

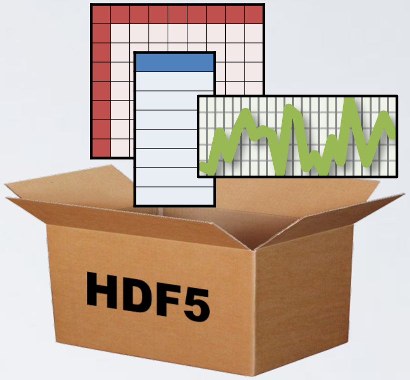



- Supernova caught 11 hours after explosion
  - 1 million times too dim to see with naked eye
- The 5th brightest supernova in 100 years


PTF11kly

# The HDF5 scientific file format

- Container for diverse scientific datasets

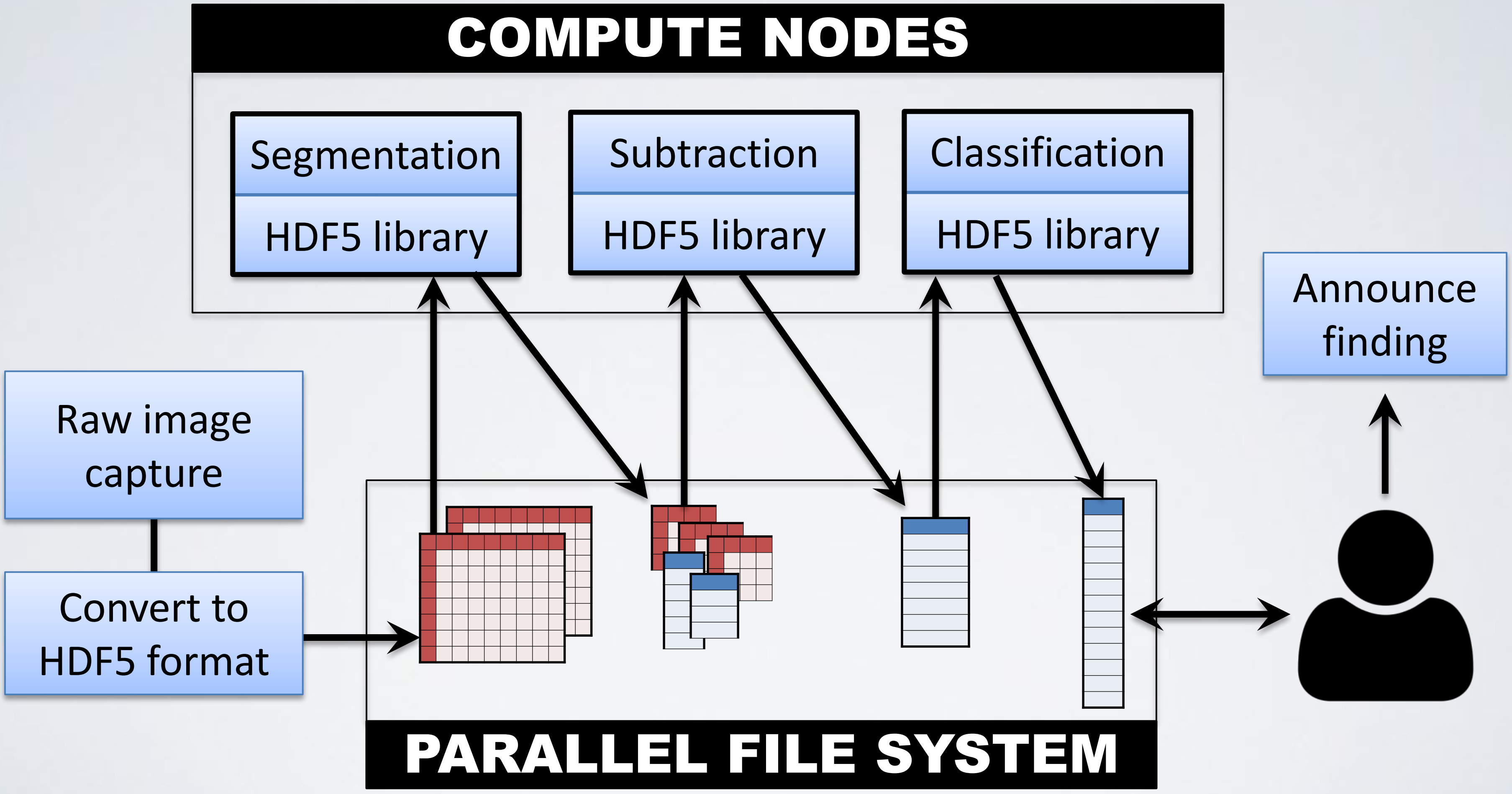



- Advantages vs. text files
  - Convenience
  - Compactness
  - Metadata support
  - Interoperability

**HDF5** is mature and widely supported

netCDF-4 uses HDF5

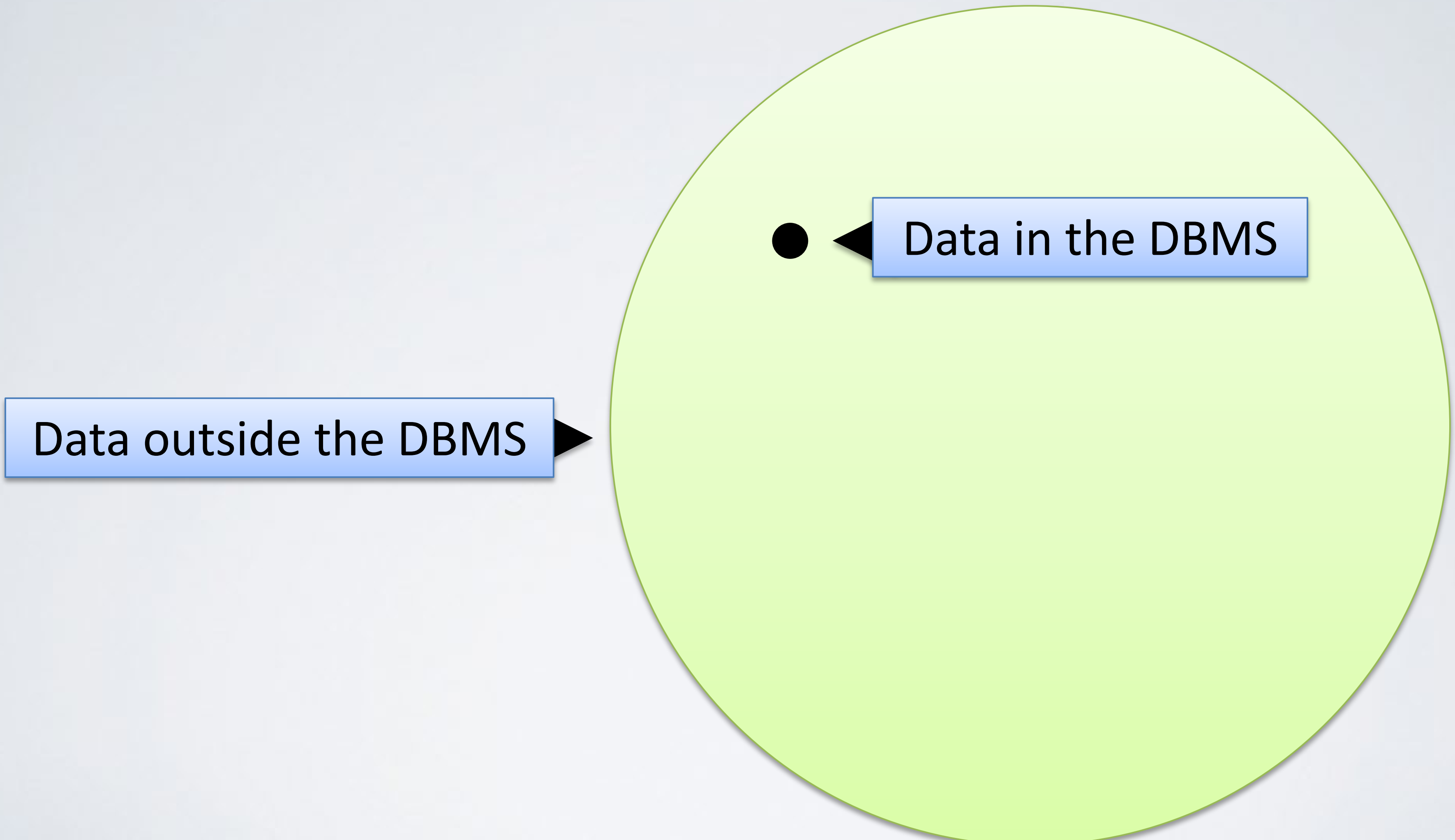
# The PTF data processing pipeline



# Status quo

- Data in the DBMS

# Status quo

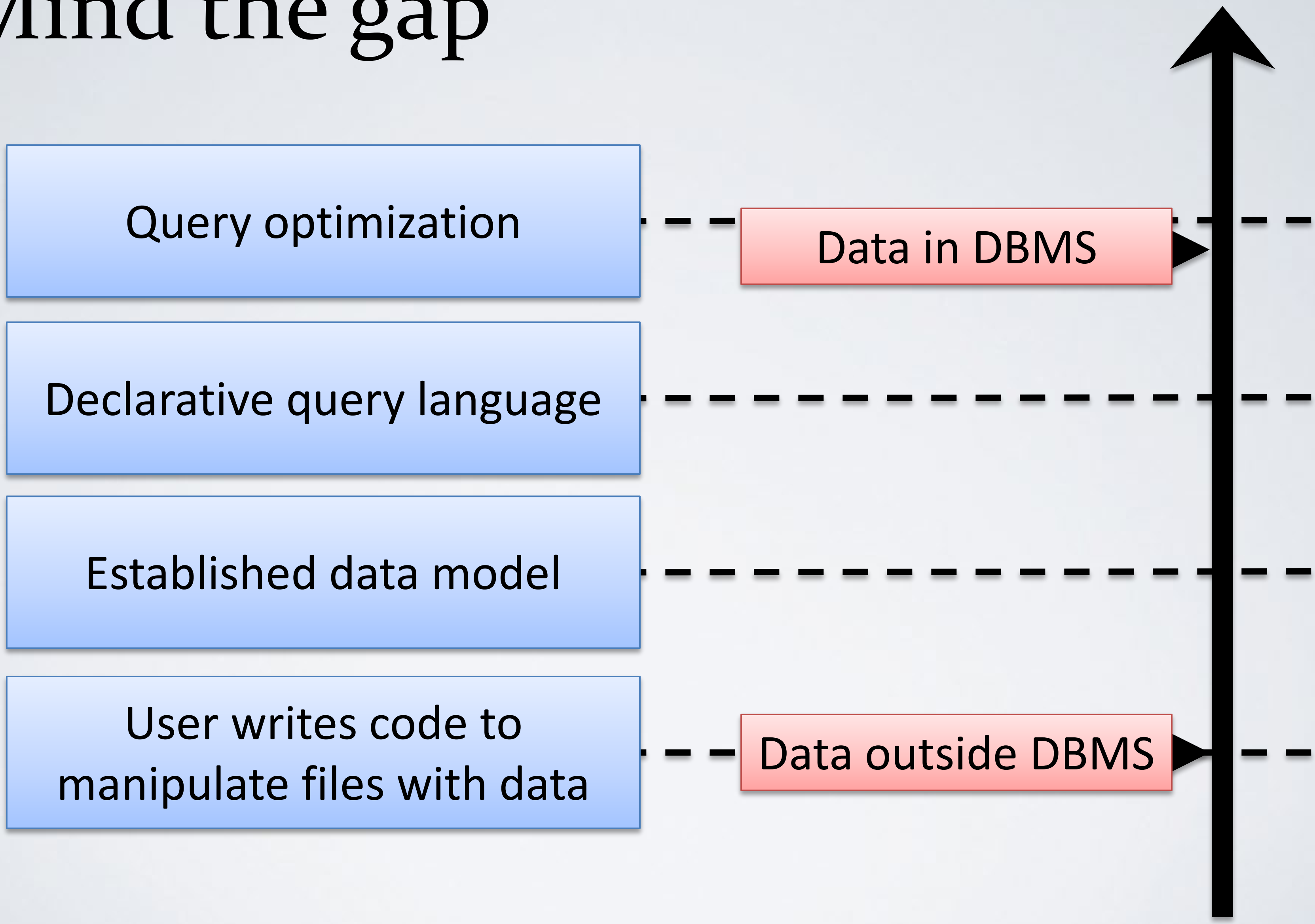


# Status quo

**Data management without database systems**

Users **write code** to run a simple analysis on massive data

# Mind the gap


Query optimization
Data in DBMS
Declarative query language
Established data model
User writes code to manipulate files with data
Data outside DBMS

# ArrayBridge

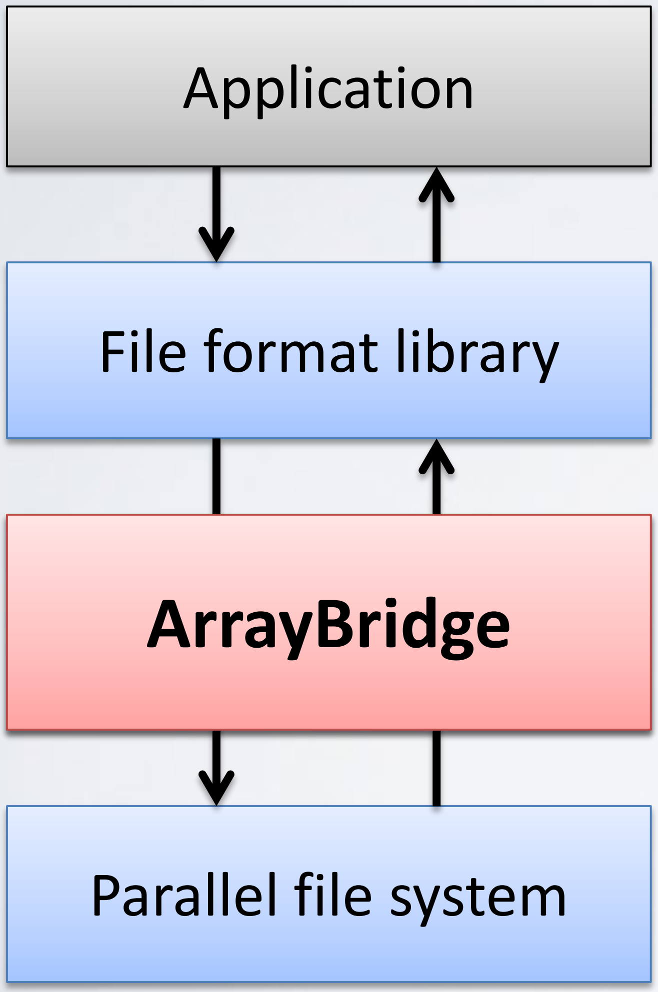
**Myth #1**

"Our data is so big that writing code is faster than using a DBMS"

**Contribution [ICDE'18]**

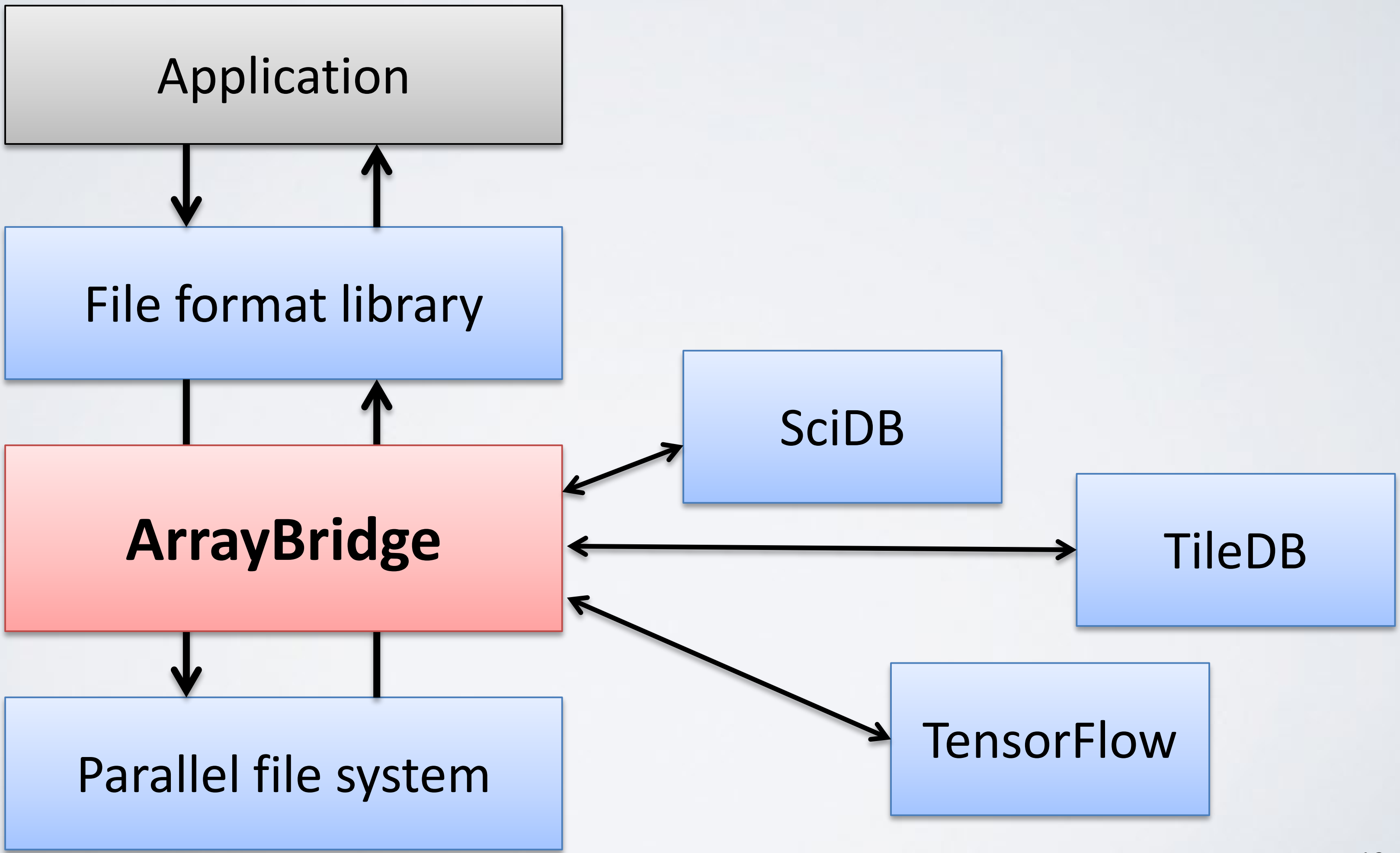
ArrayBridge: database processing directly on HDF5 data

# ArrayBridge

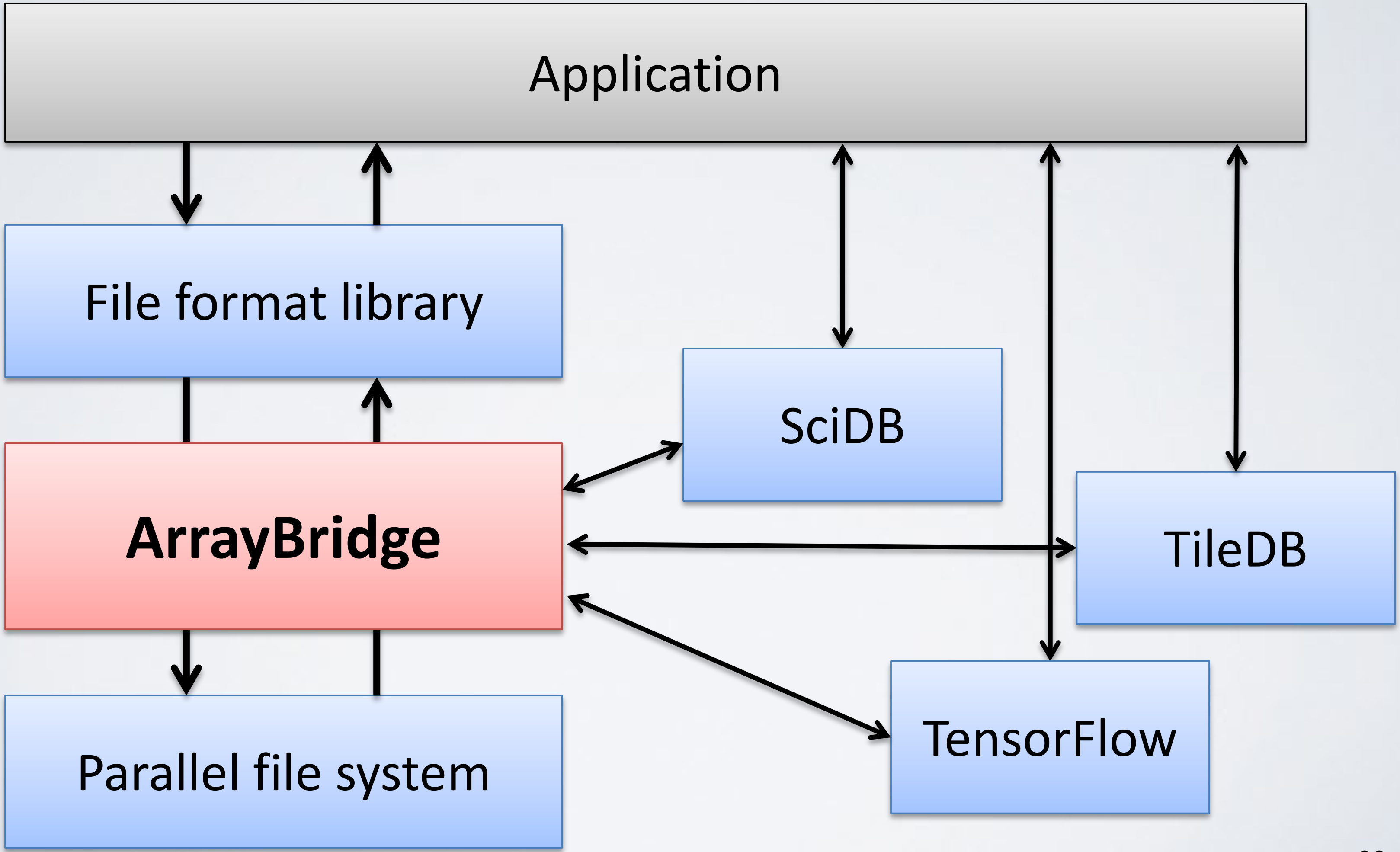



- Keep identical API for backwards compatibility
- Discover application-specific I/O patterns
- Inject optimizations into I/O path

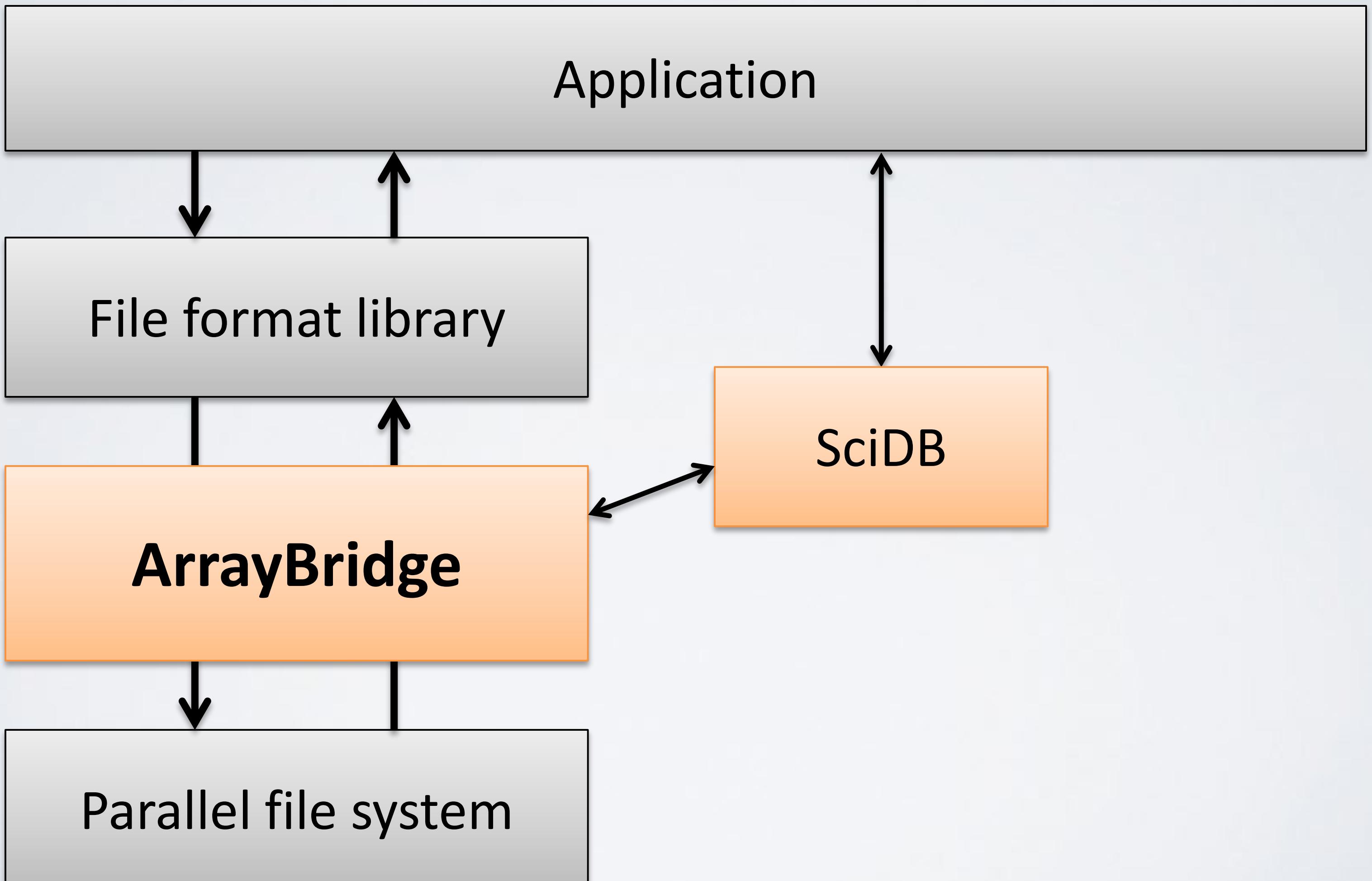
# ArrayBridge



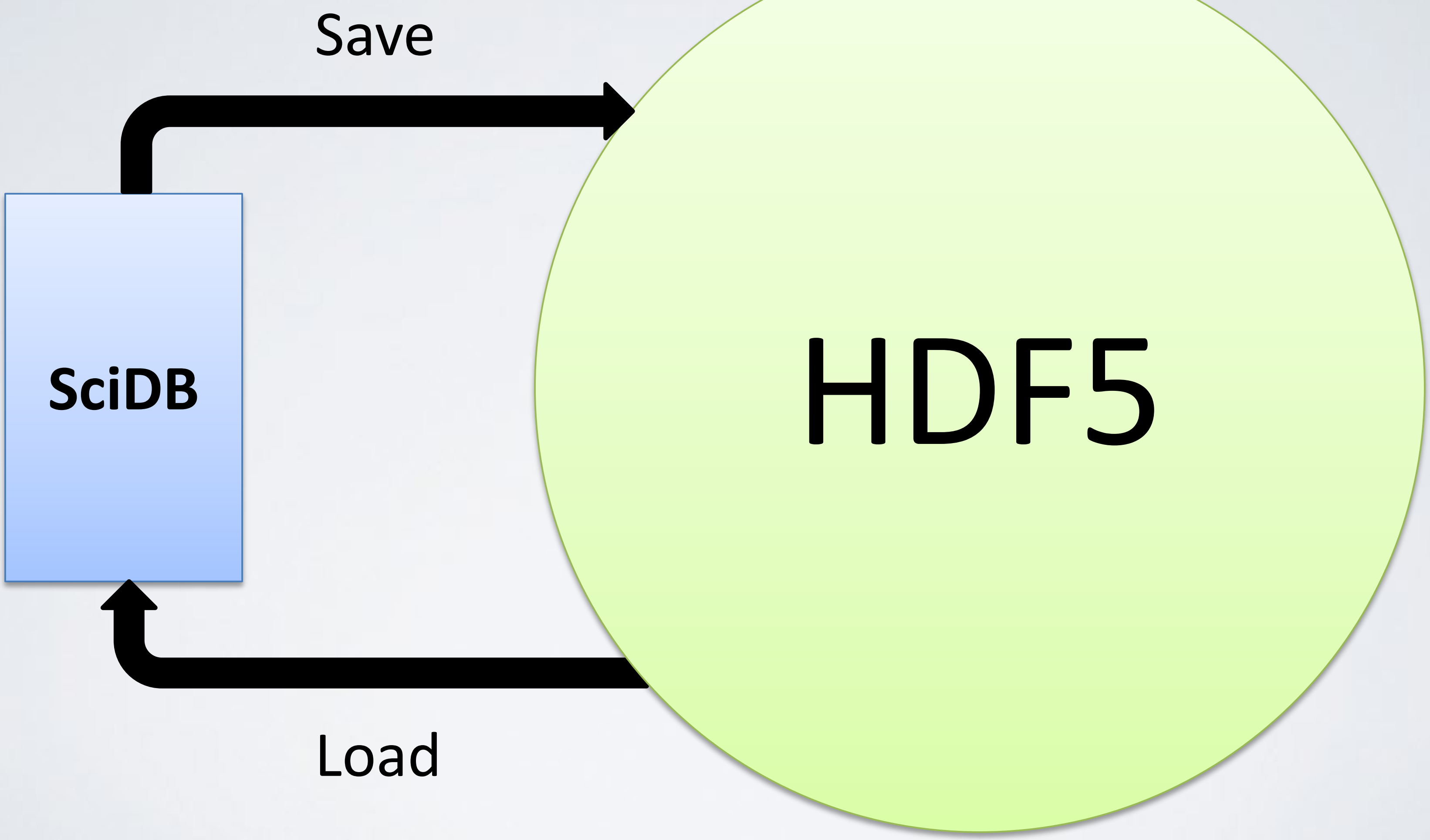
# ArrayBridge



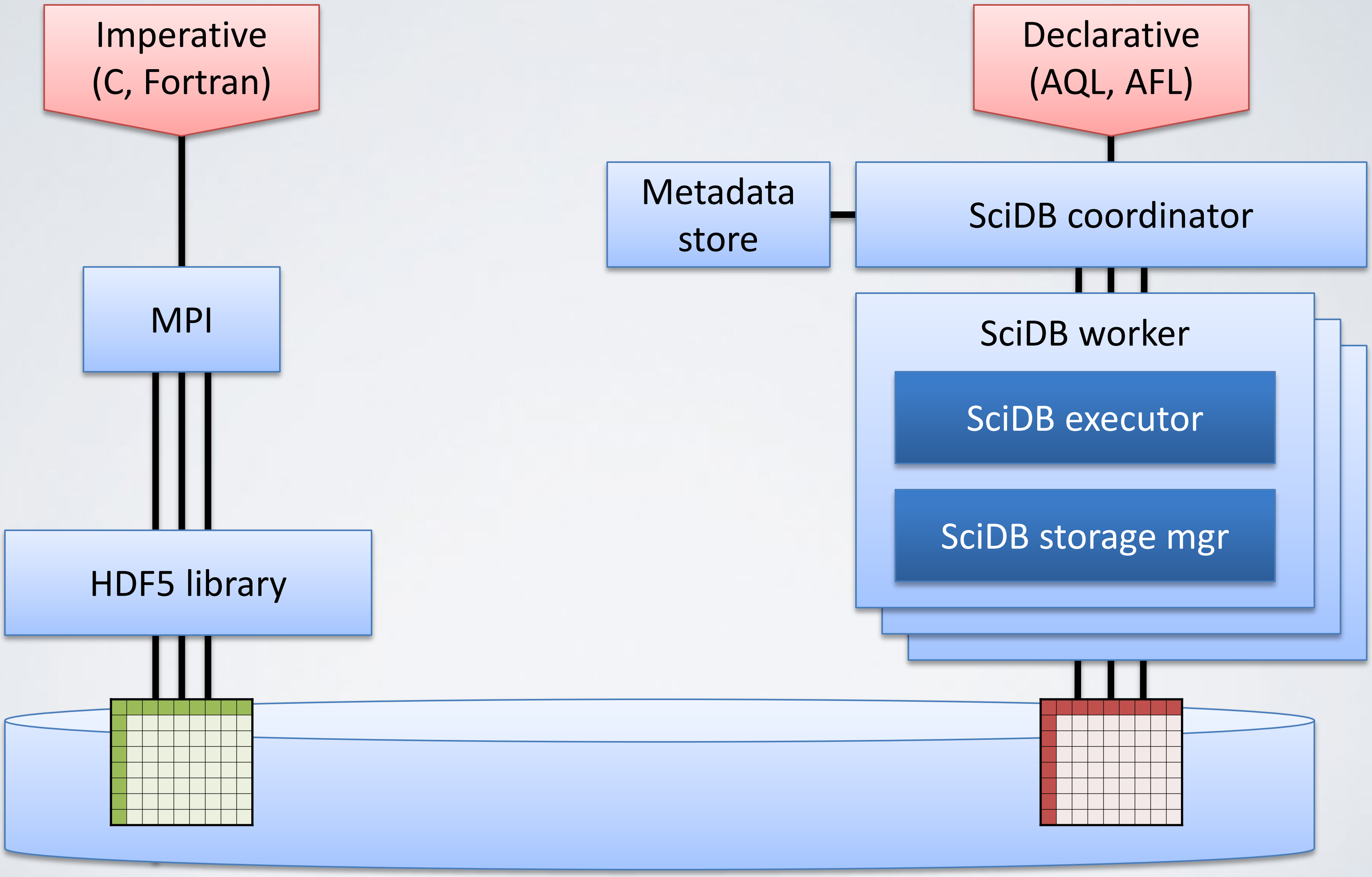
# SciDB meets scientific computing



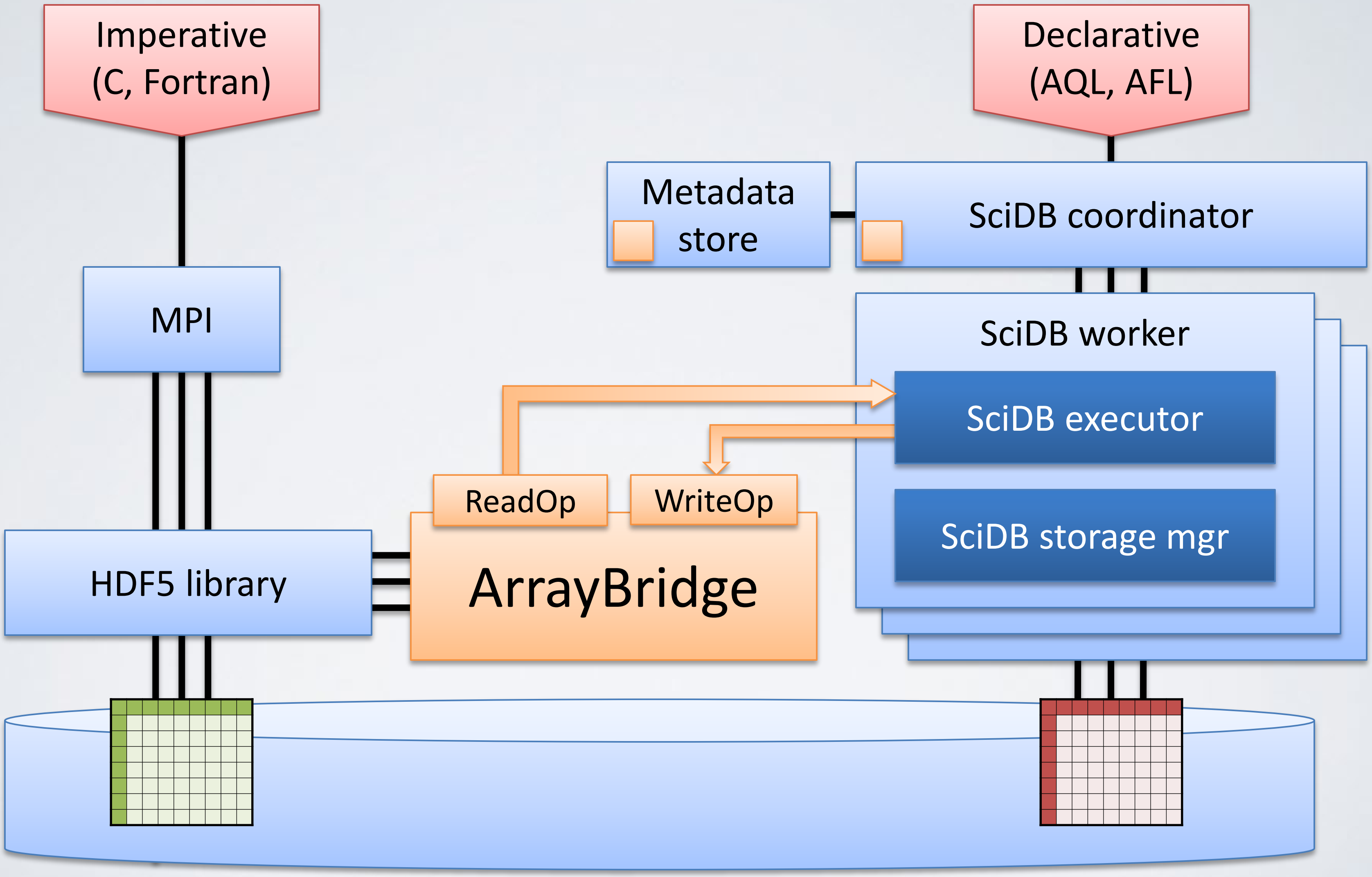
# SciDB meets scientific computing



# Current state: two silos

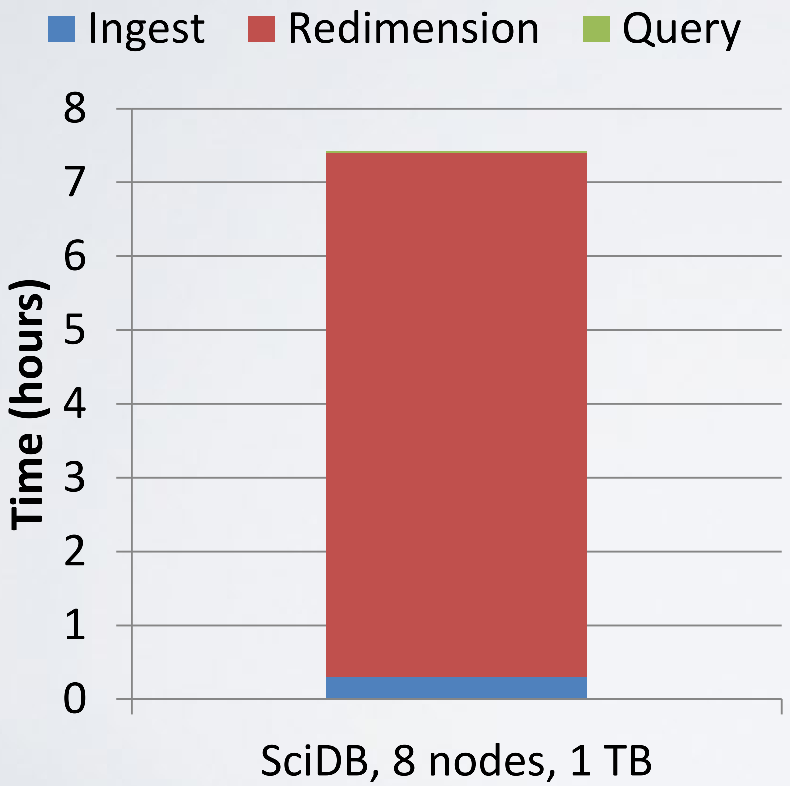


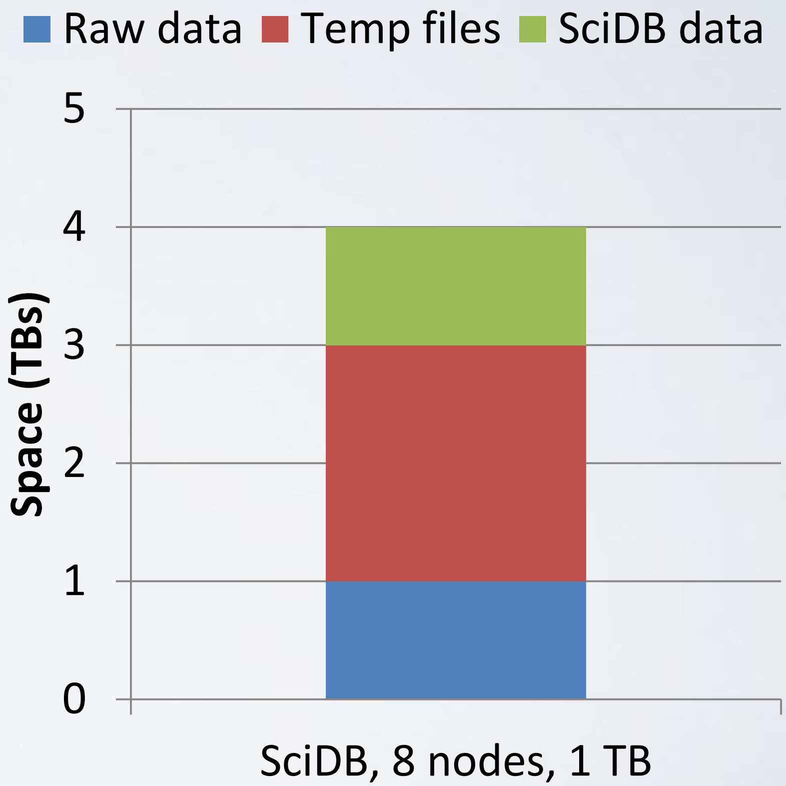
# ArrayBridge overview



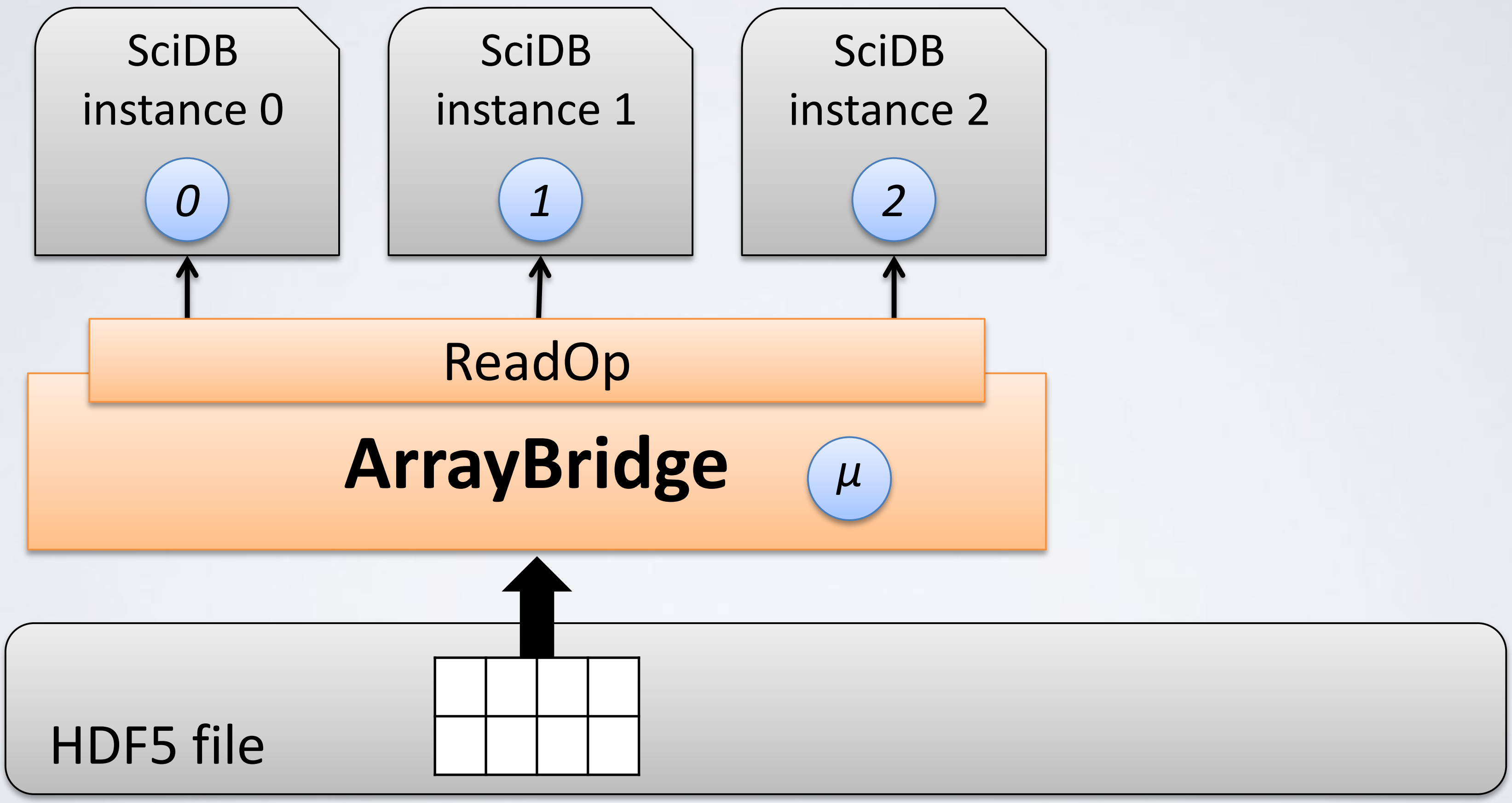
# Problem 1: Loading is slow in SciDB

Loading 1TB in SciDB takes 7.5 hours, 4TB of space

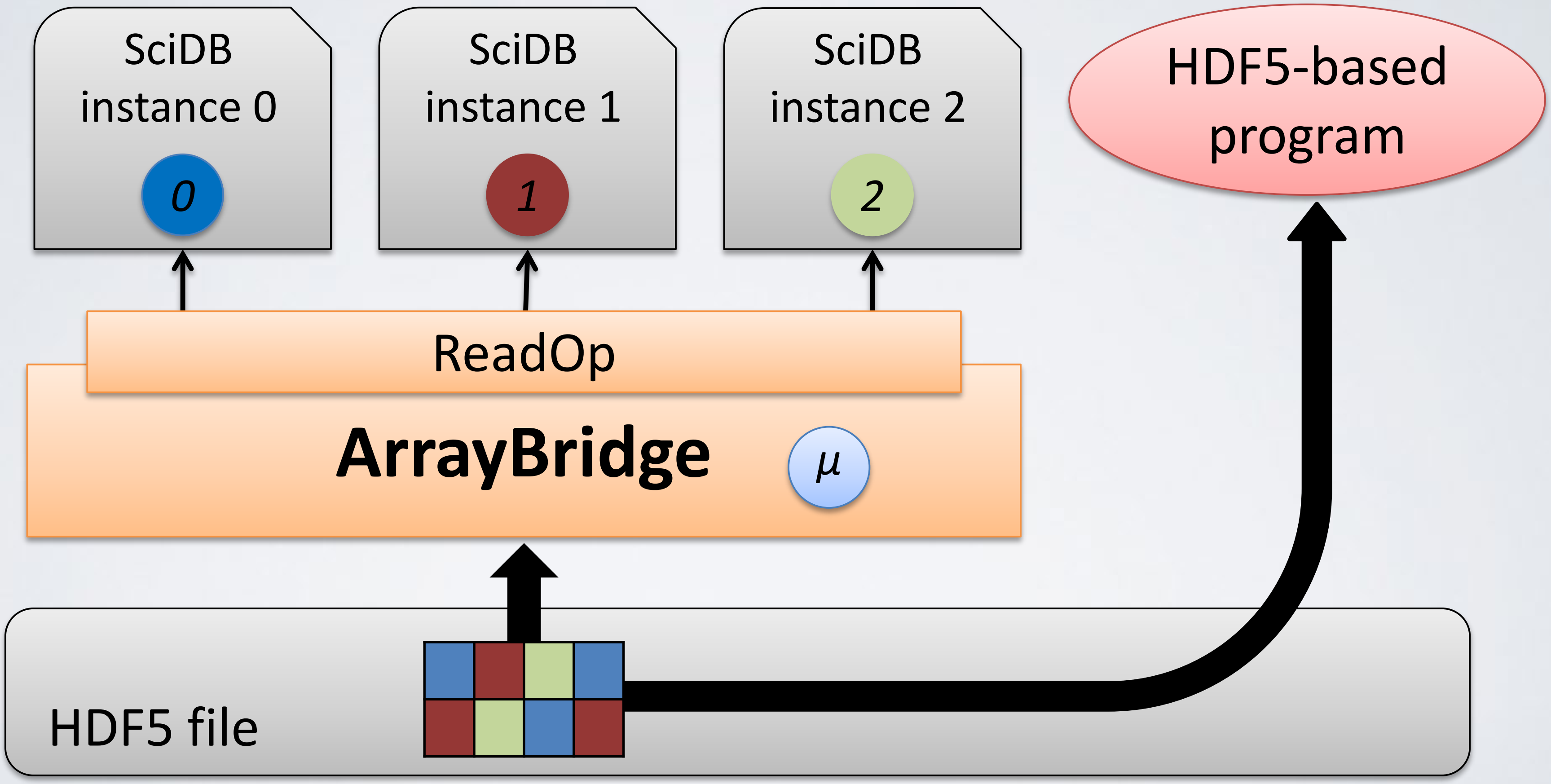


# Reading in ArrayBridge



# Reading in ArrayBridge



# Problem 2: Saving doesn't scale

Fundamental problem:
write concurrency

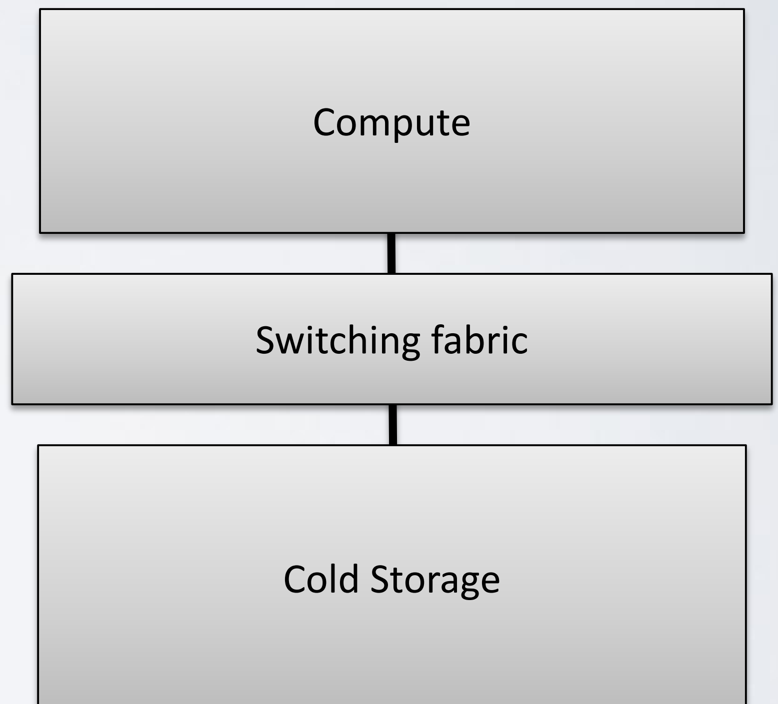
↓

SWMR limitation
(**S**ingle **W**riter, **M**any **R**eaders)

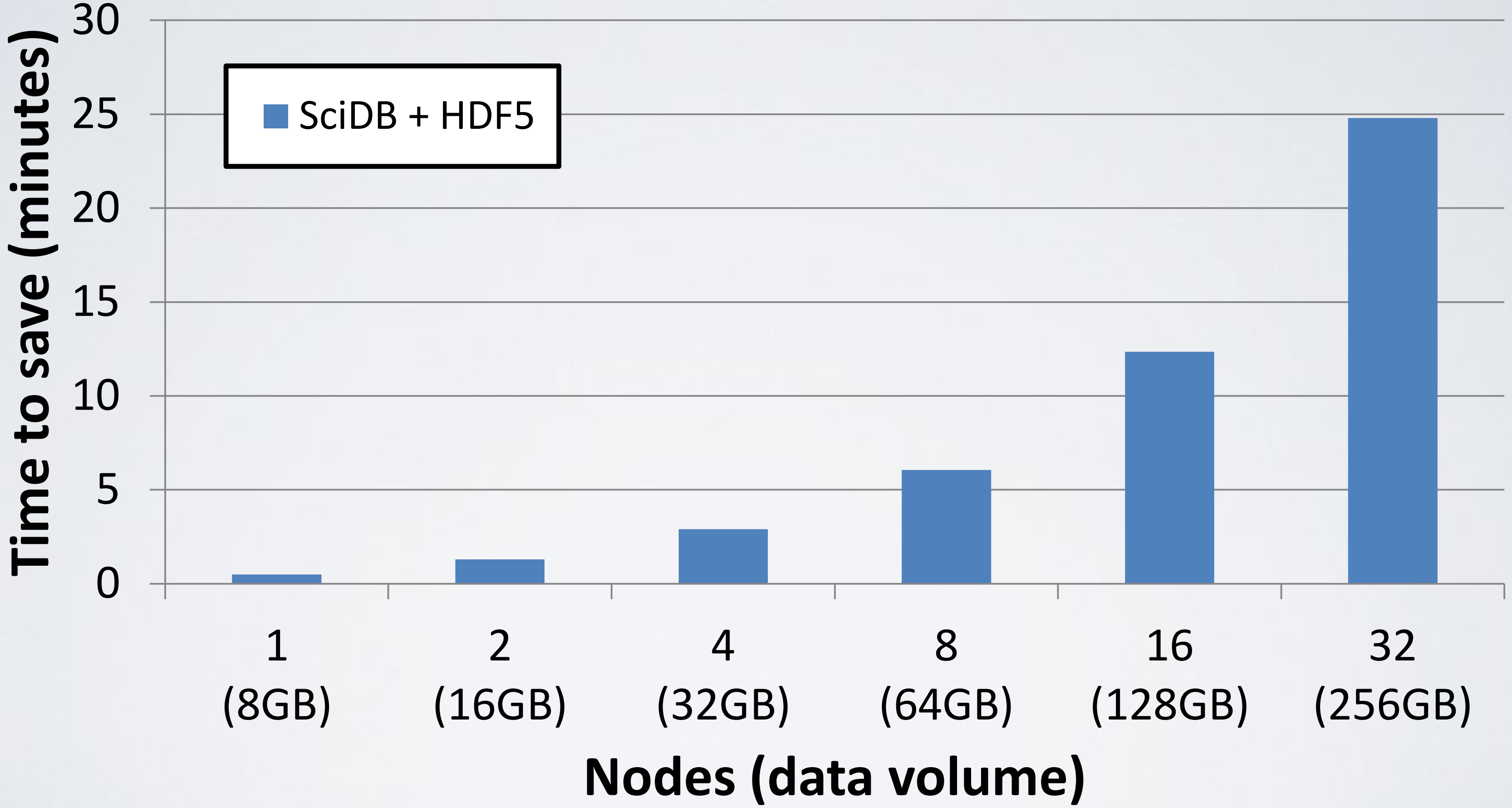
↓

write concurrency ≈ 1

**High-end computer**

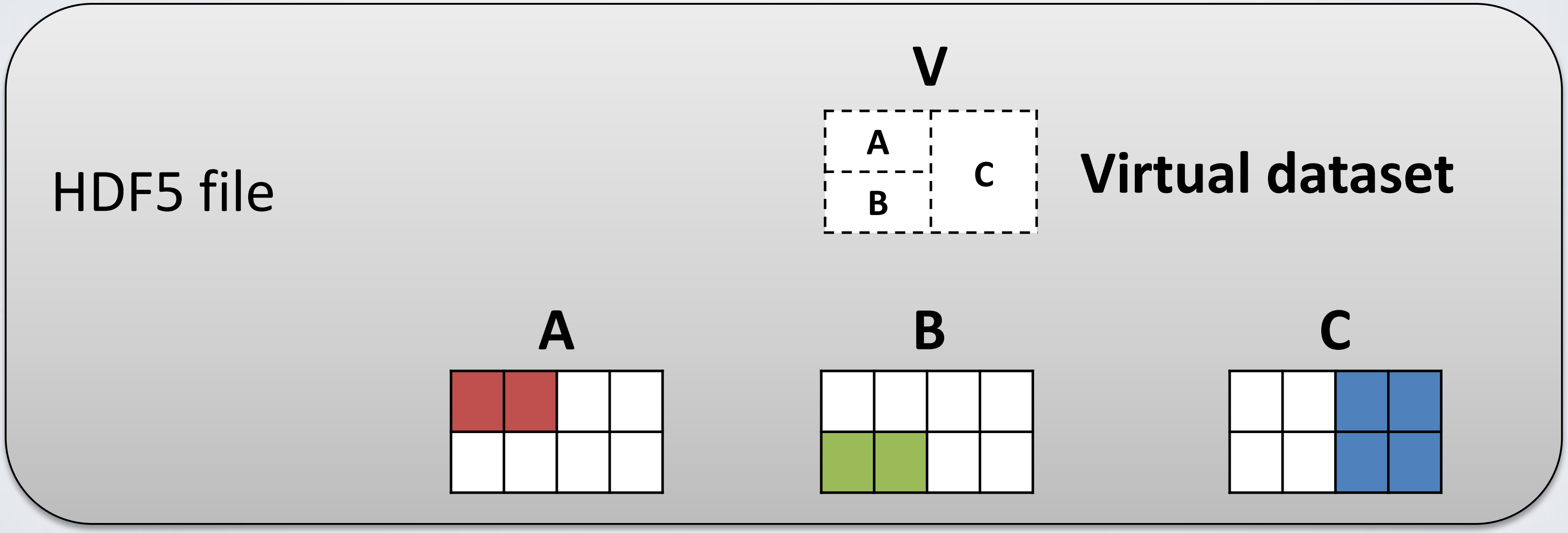


# Problem 2: Saving doesn't scale

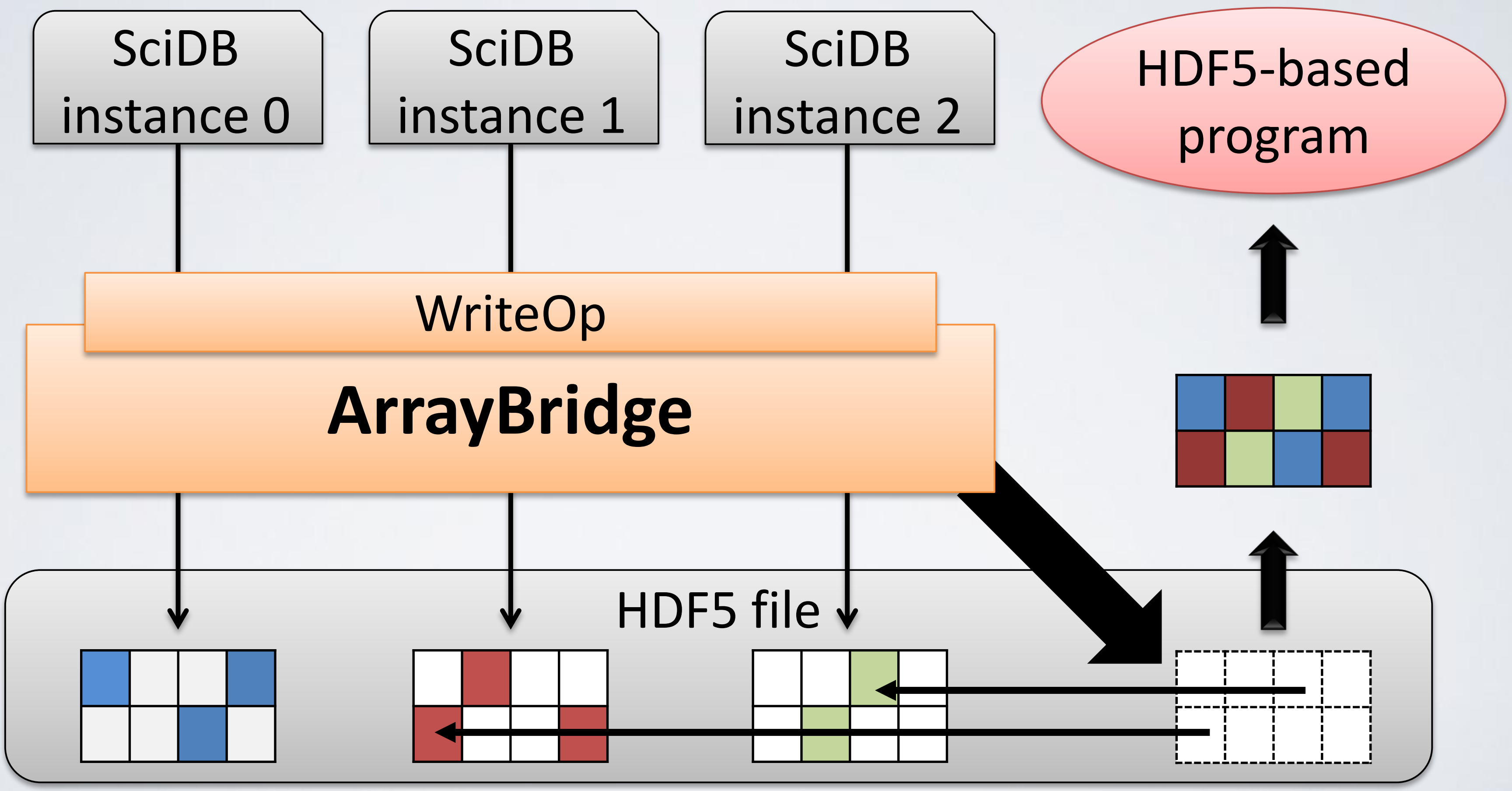


# Parallel writing in ArrayBridge

- ArrayBridge uses *virtual datasets* in HDF5
  - Recent (2016) feature, introduced in HDF5 1.10
  - Virtual dataset = a non-materialized view

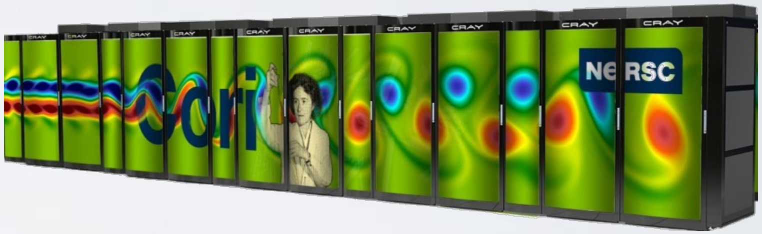


# Parallel writing in ArrayBridge



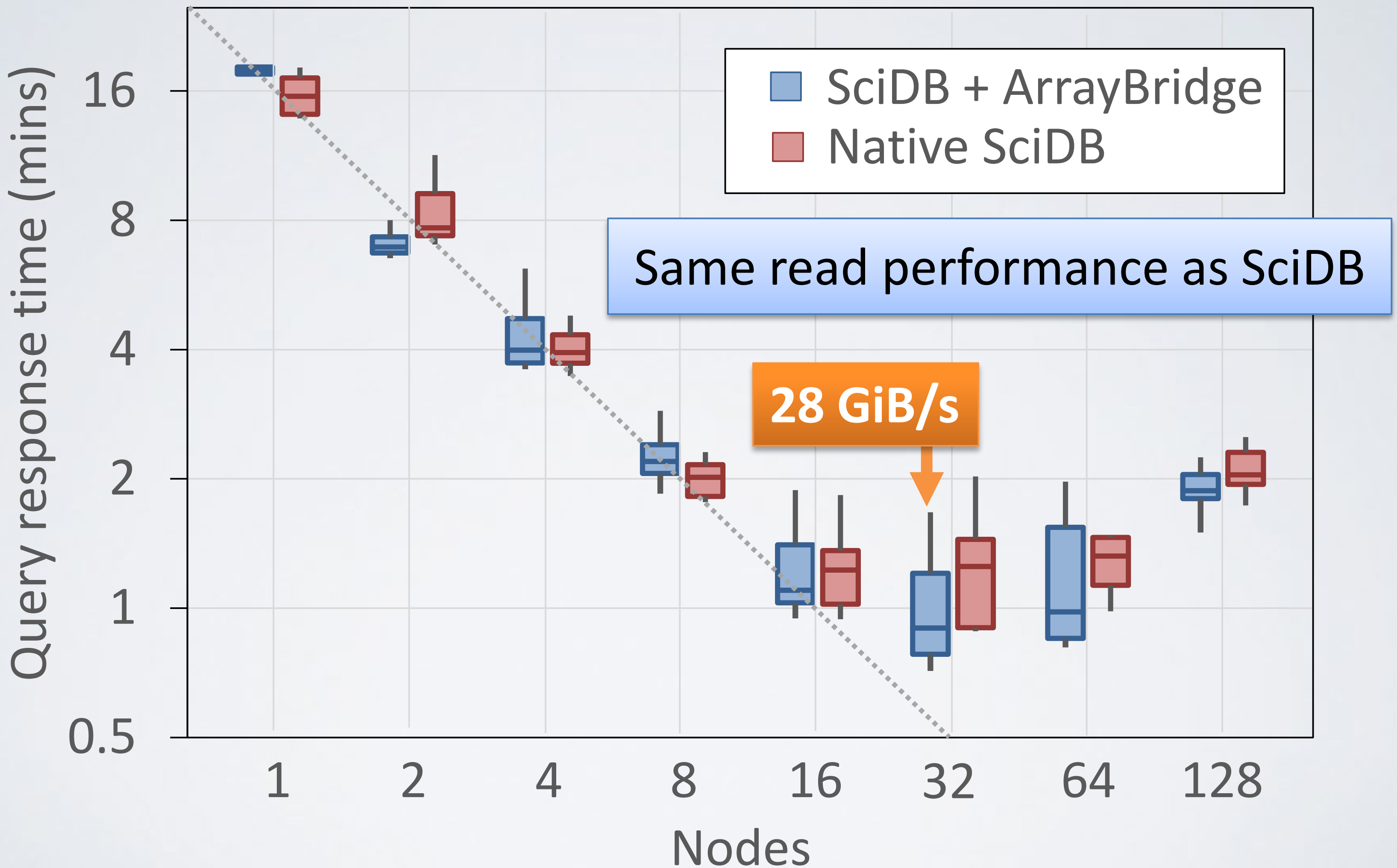
# Experimental evaluation

- High-performance computer, Cray XC40
  - 52,160 CPU cores
  - 204 TB memory
  - 10,168 HDDs
  - 30 PB cold storage

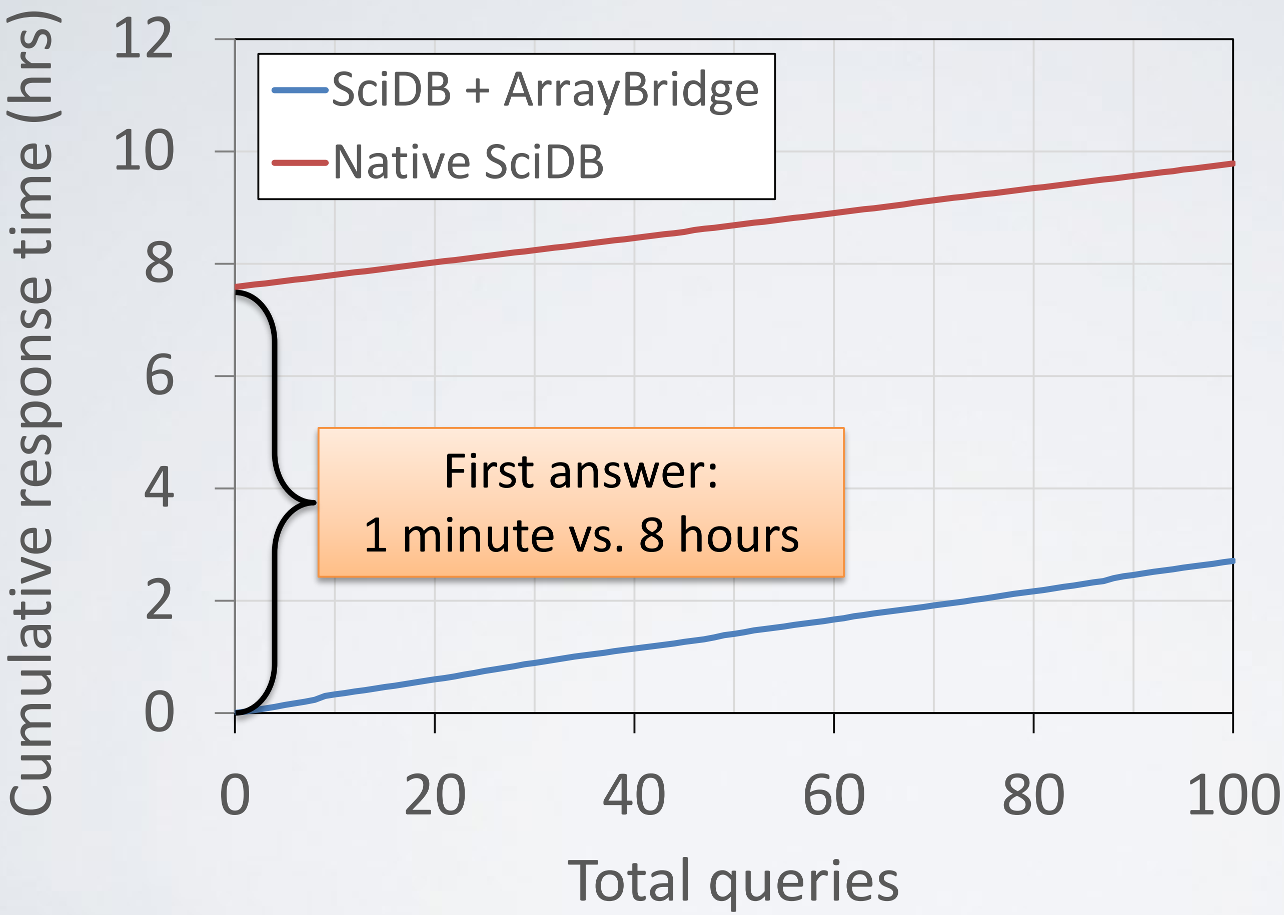



- Shared resource
  - Reporting variance when significant

# Read performance on 1.5 TB array

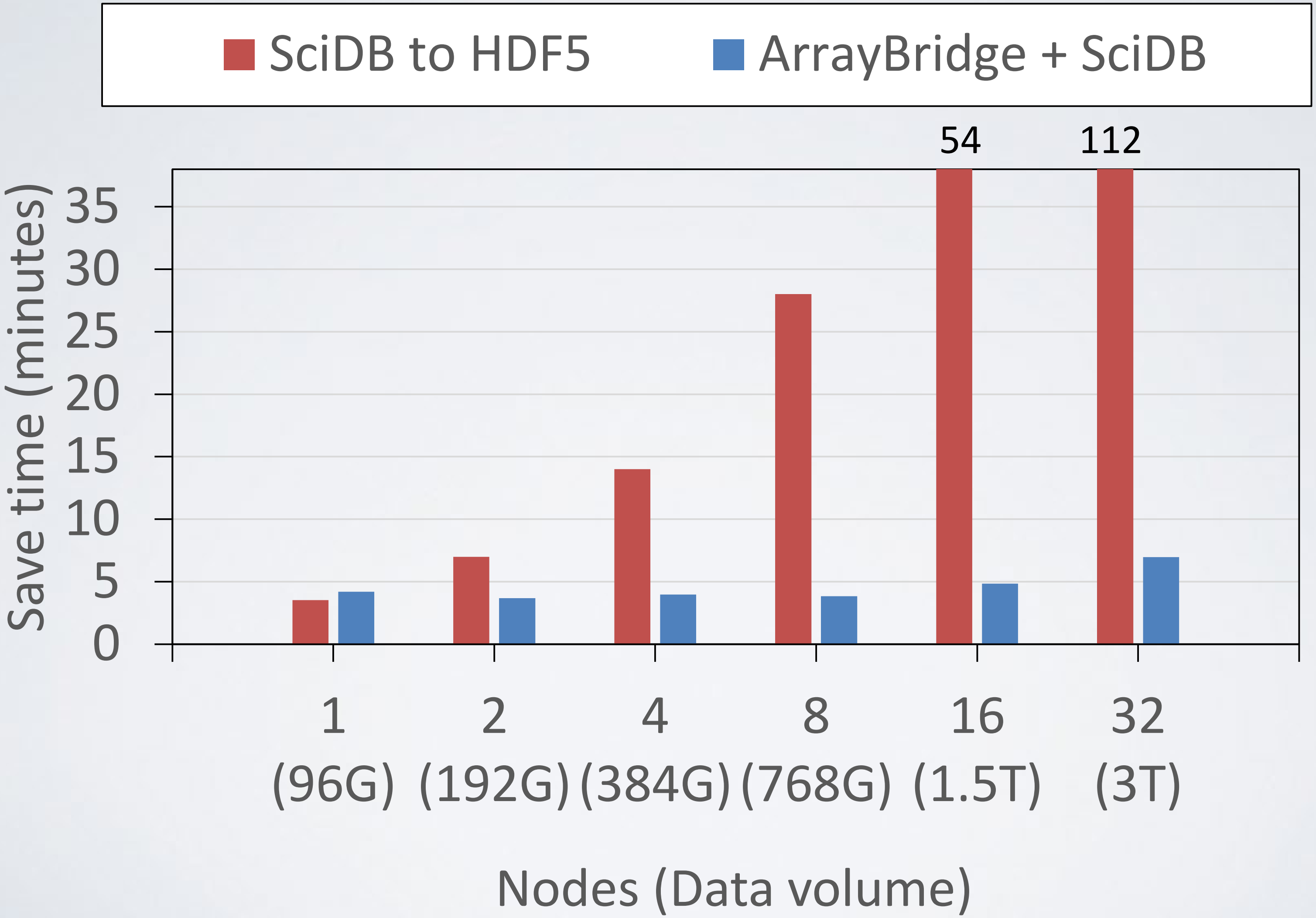


# Time to insight




- 1 TiB Data
- 100 consecutive aggregations

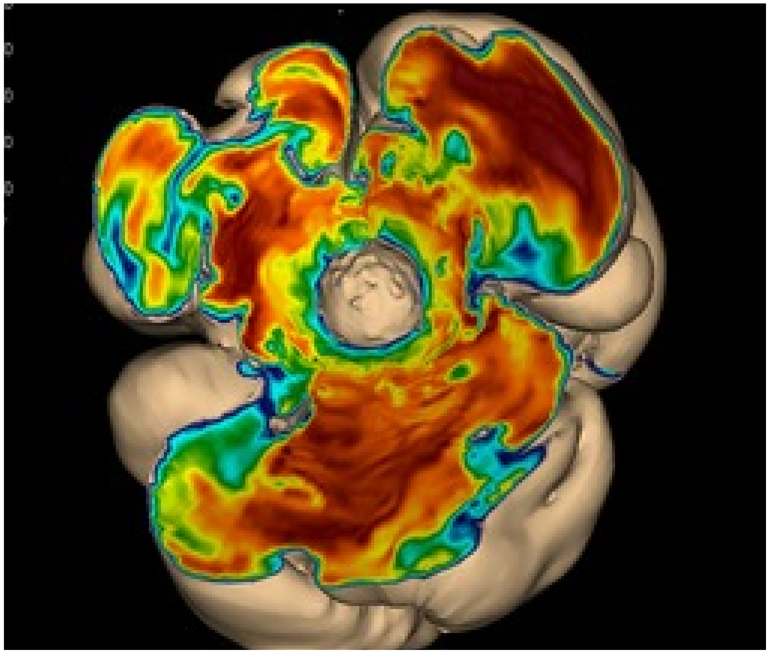
# Writing performance



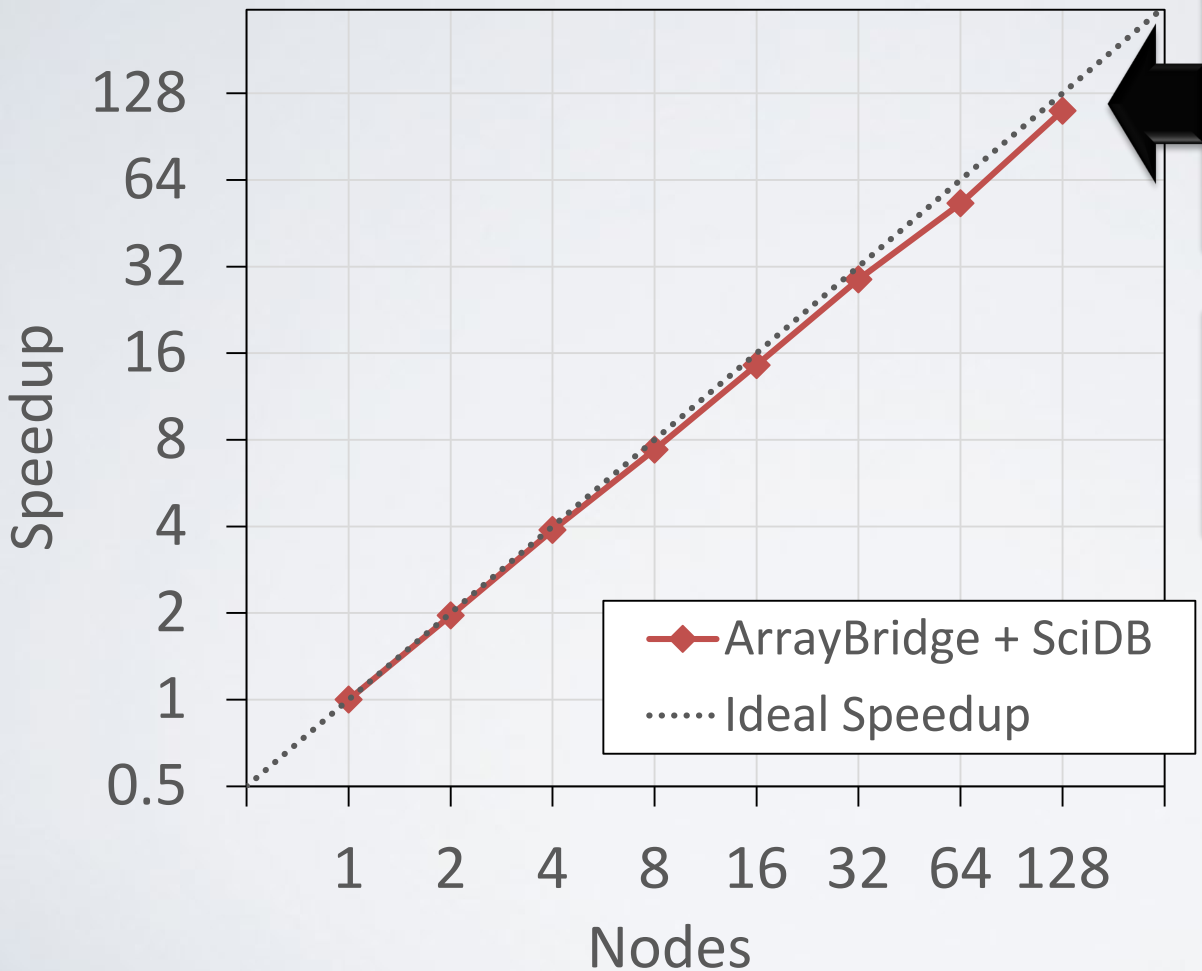
# Real dataset: VPIC

- Particle-in-cell simulation
- One 43TB array in HDF5
  - Per step of the simulation!
- Dataset:
  - Particle ID (1D)
  - Particle position (3D)
  - Particle velocity (3D)
- Query: Filter & Group By
  - Find high energy regions

**Plasma physics**



# Real dataset: VPIC




ArrayBridge + SciDB processes 43TB in **15 minutes**

Loading 43TB in SciDB takes **<u>10 days</u>**!

# ArrayBridge

**Contribution [ICDE'18]**

ArrayBridge: database processing on HDF5 files

**http://code.osu.edu/arraybridge**

**Opportunity ahead**

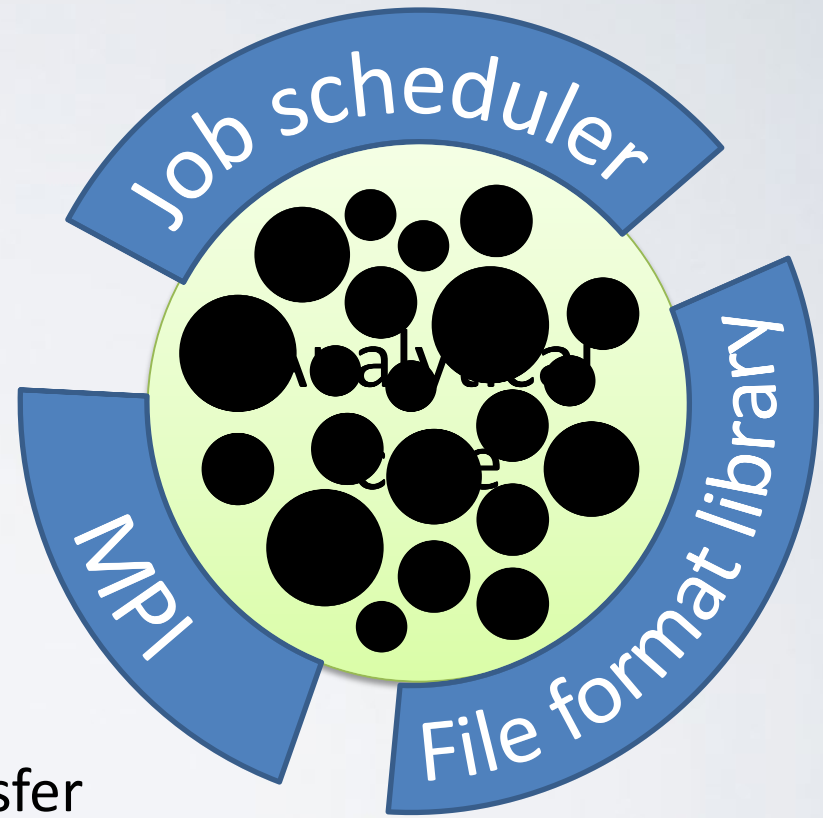
Transparent I/O management, cost-based optimization from imperative applications

# Smarter I/O for Large Array Processing

1. What are the I/O patterns of large-scale applications with complex data structures?
2. How to arrange complex objects in disk blocks?
3. How do we automate I/O optimization?

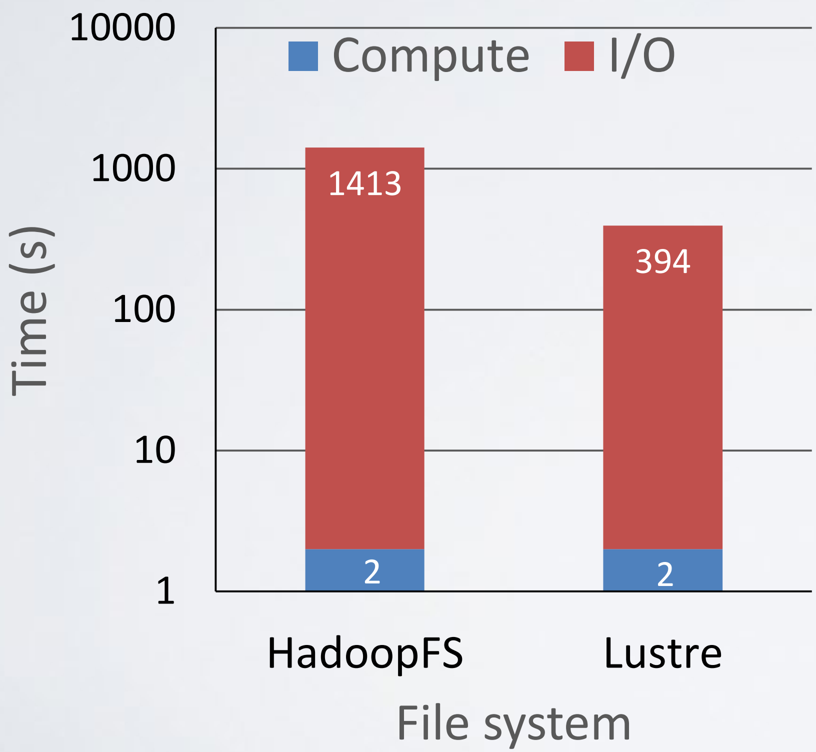
# Capture I/O patterns at scale

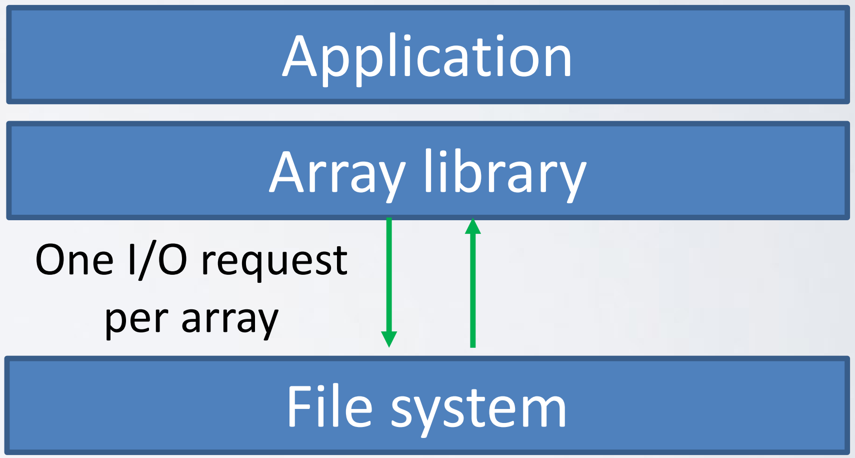
- Diverse analytical core
  - Ad-hoc reuse of older codes
  - Problem-specific optimizations
- Homogenous periphery, common tools for:
  - Parallelism & task management
  - Communication & data transfer
  - I/O to cold storage



# Small array challenge

- A file per array
  - I/O takes 200× to 700× longer than compute

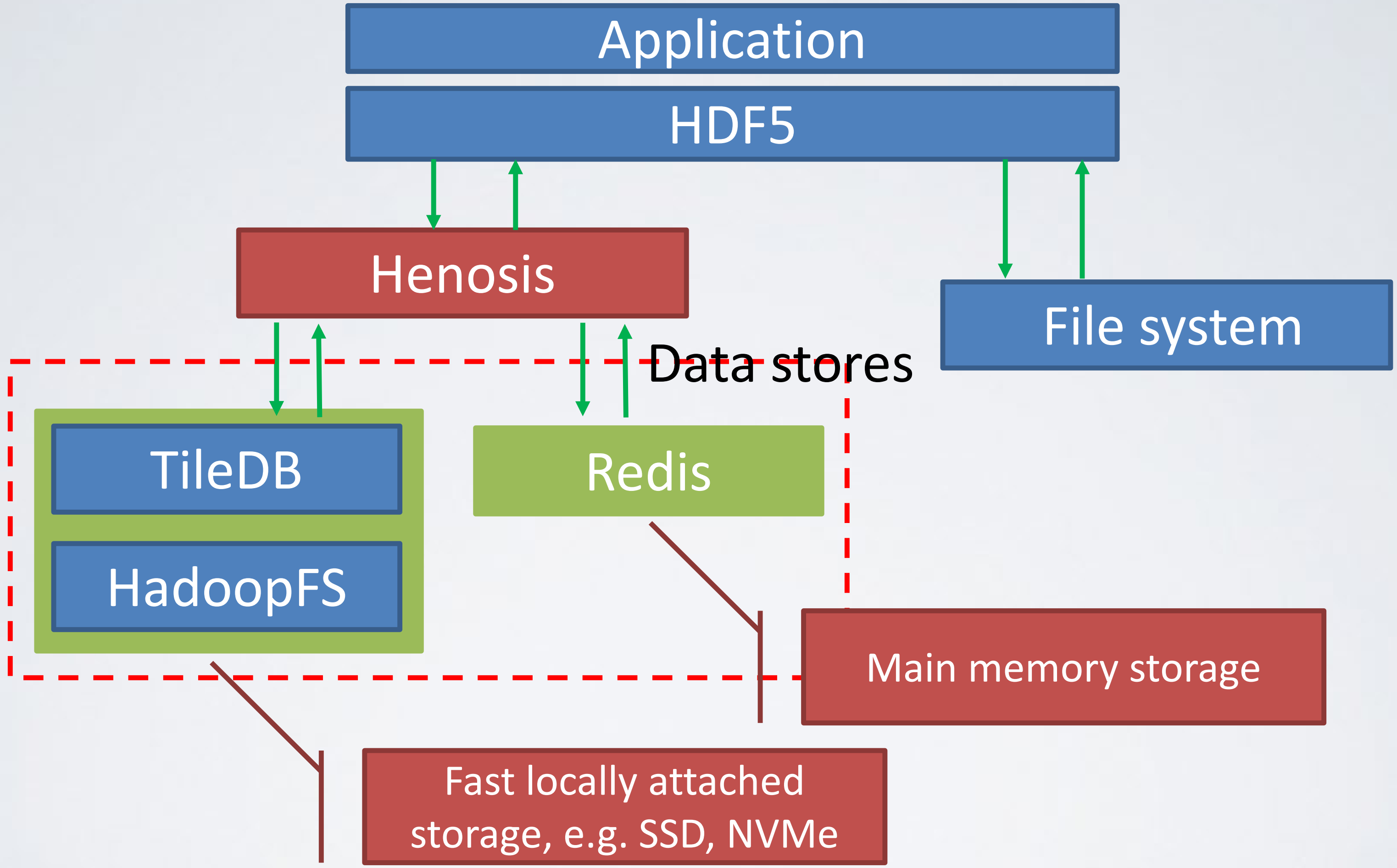


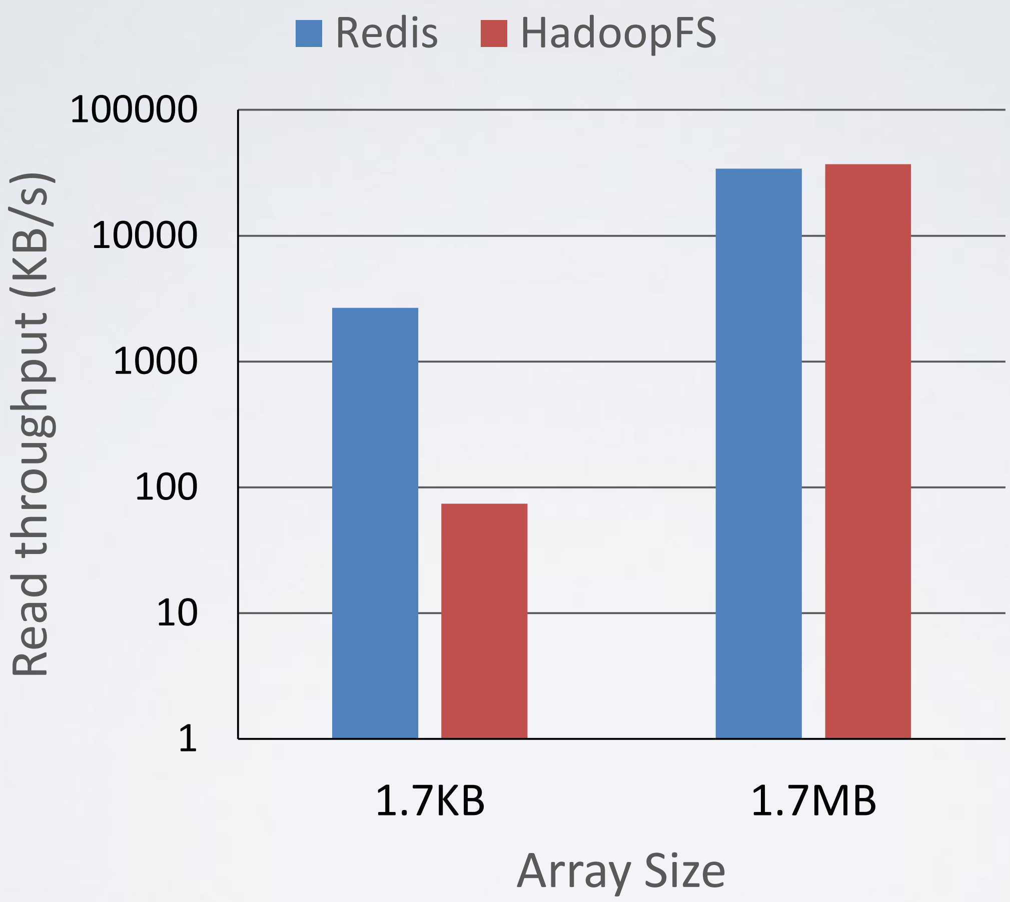
# Small array challenge

Too many small I/O requests!

# Software stack



# Throughput

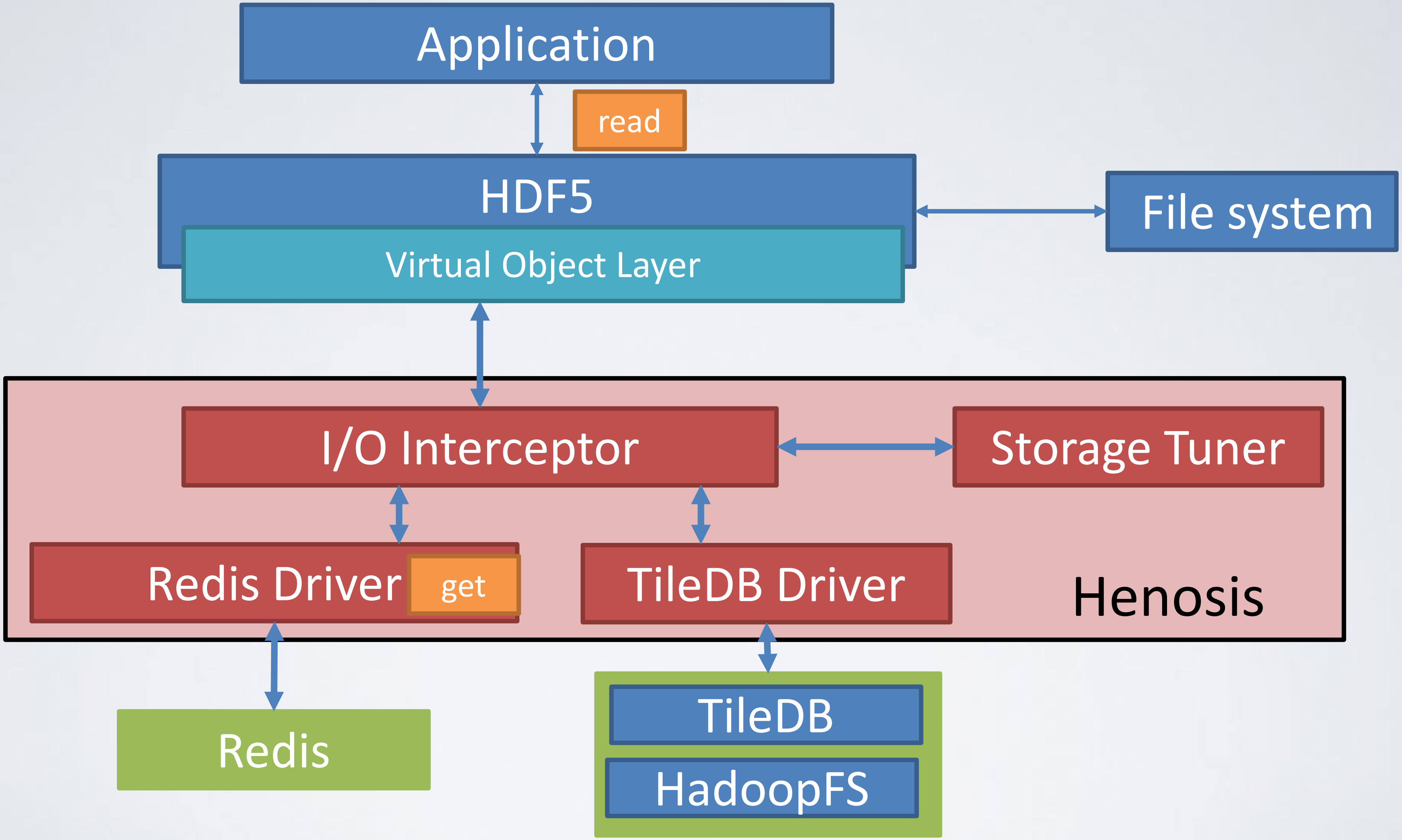


Redis
HadoopFS
100000
10000
1000
100
10
1
Read throughput (KB/s)
1.7KB
1.7MB
Array Size

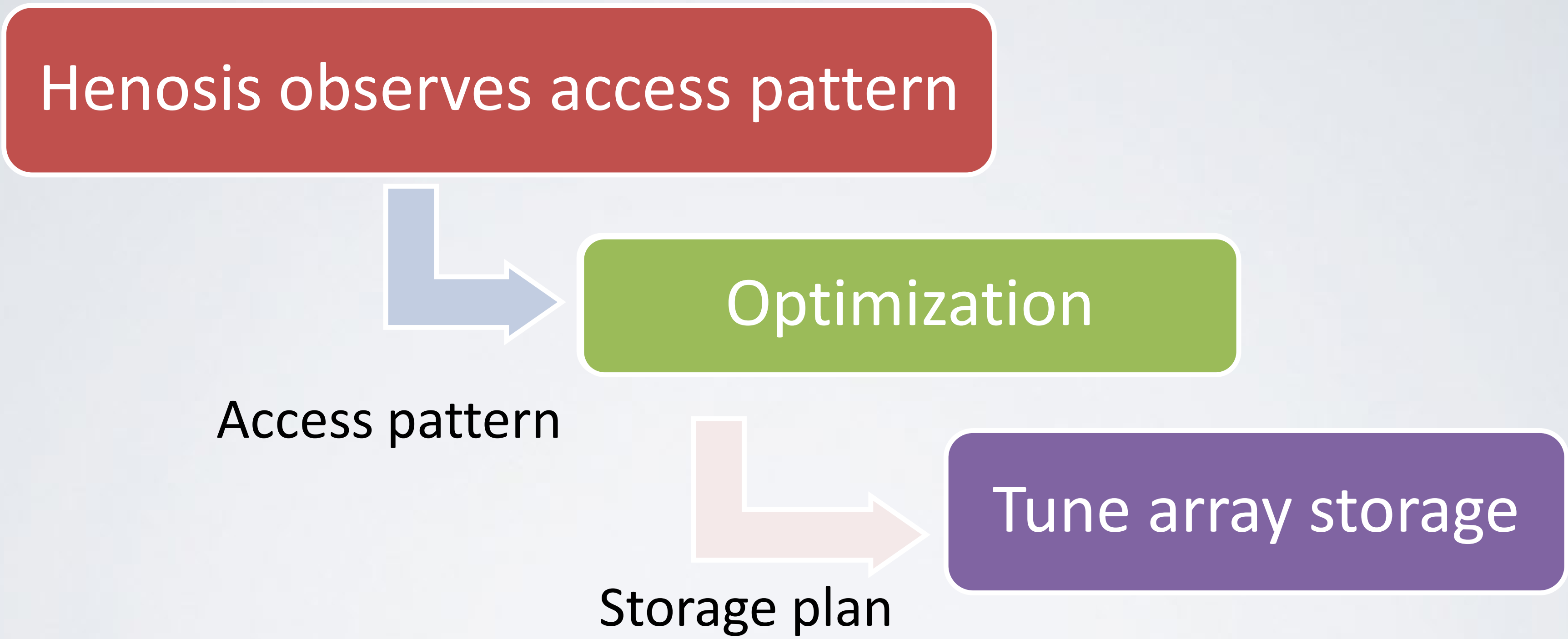
# Goals

- Store arrays on heterogeneous data stores
  - Without modifying applications
- Accelerate I/O
  - Improve the performance of each request
  - Reduce the number of requests
- Automatically decide the array storage layout
  - Which data store should an array be placed in?
  - How do we store small arrays in chunks?

# System architecture

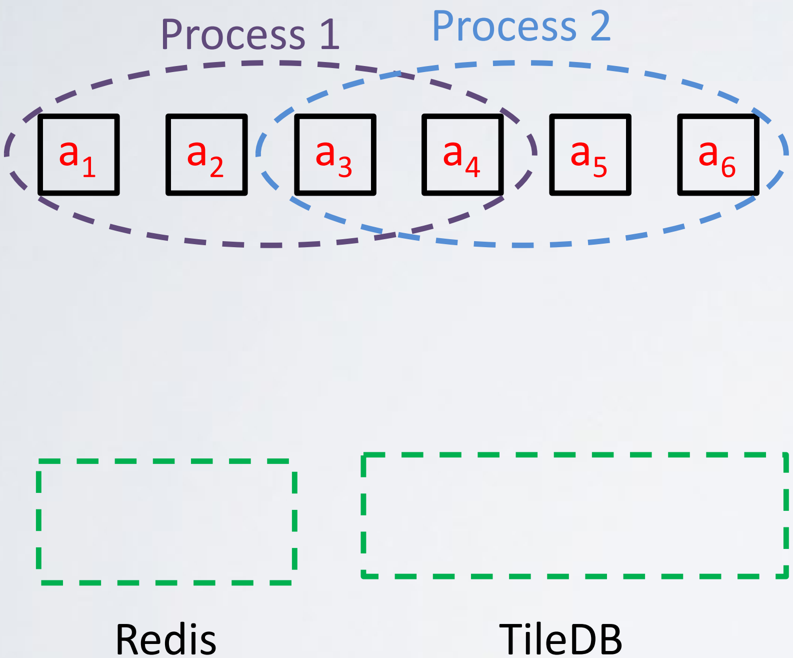


# Optimization workflow

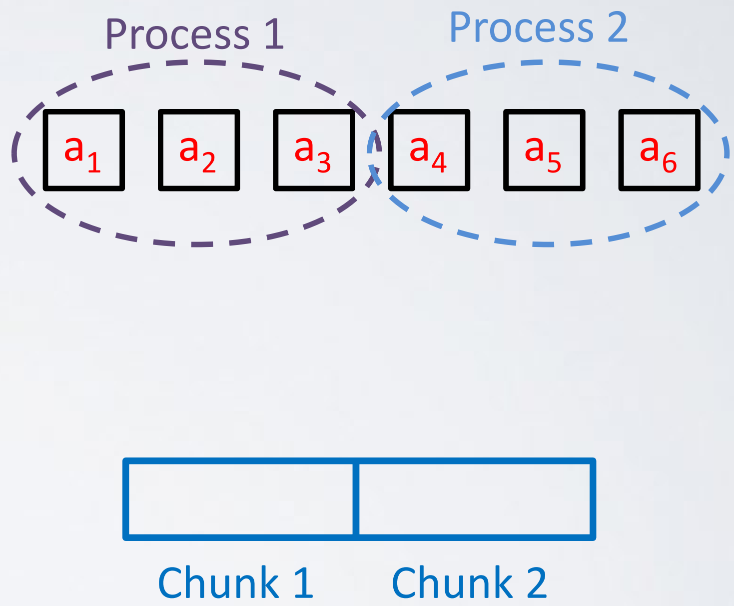


# I/O acceleration techniques

- Placement




- Consolidation



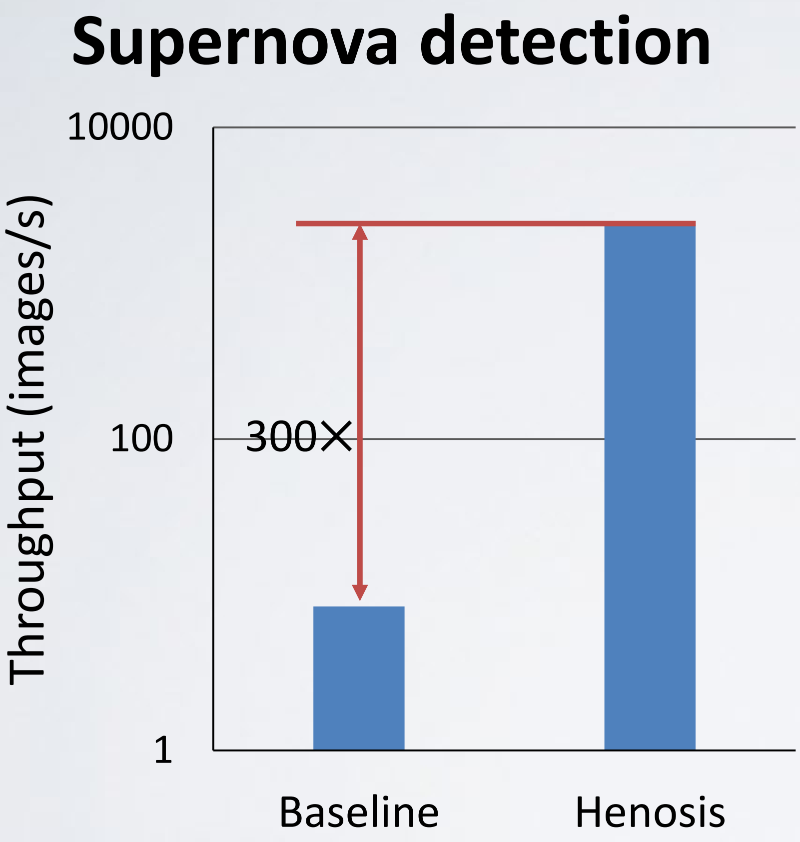
# Experimental evaluation

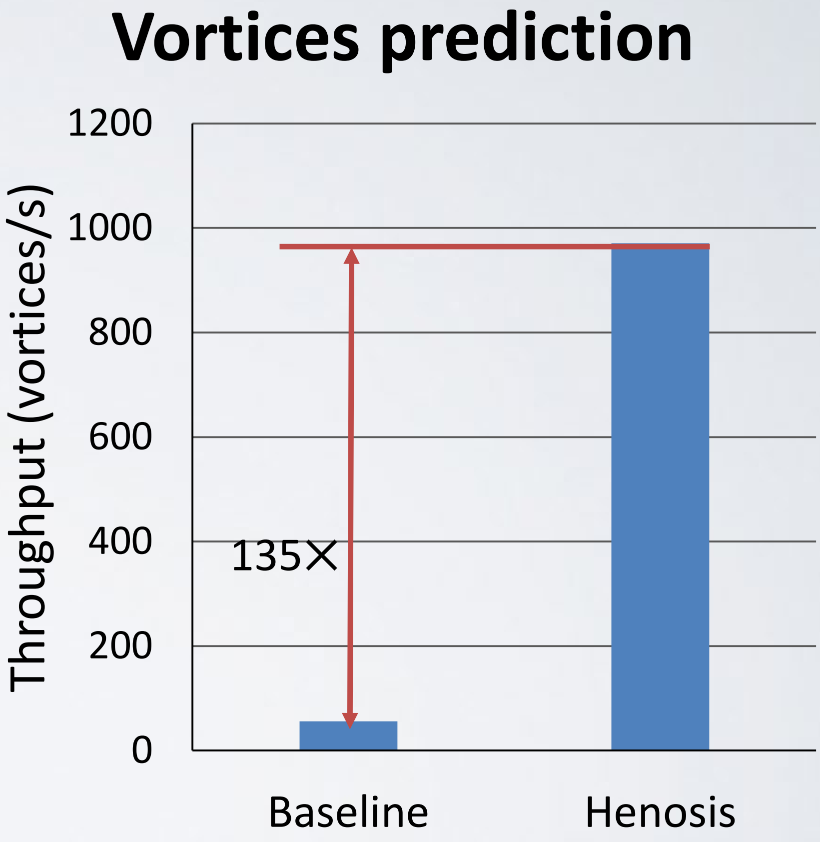
- How effective is consolidation?
  - Compare with reading small arrays directly
- What is the performance gain from optimization?
  - Compare with consolidation only
  - Compare with consolidating and placing independently
    - Consolidate-then-place
    - Place-then-consolidate

# Experiment setup

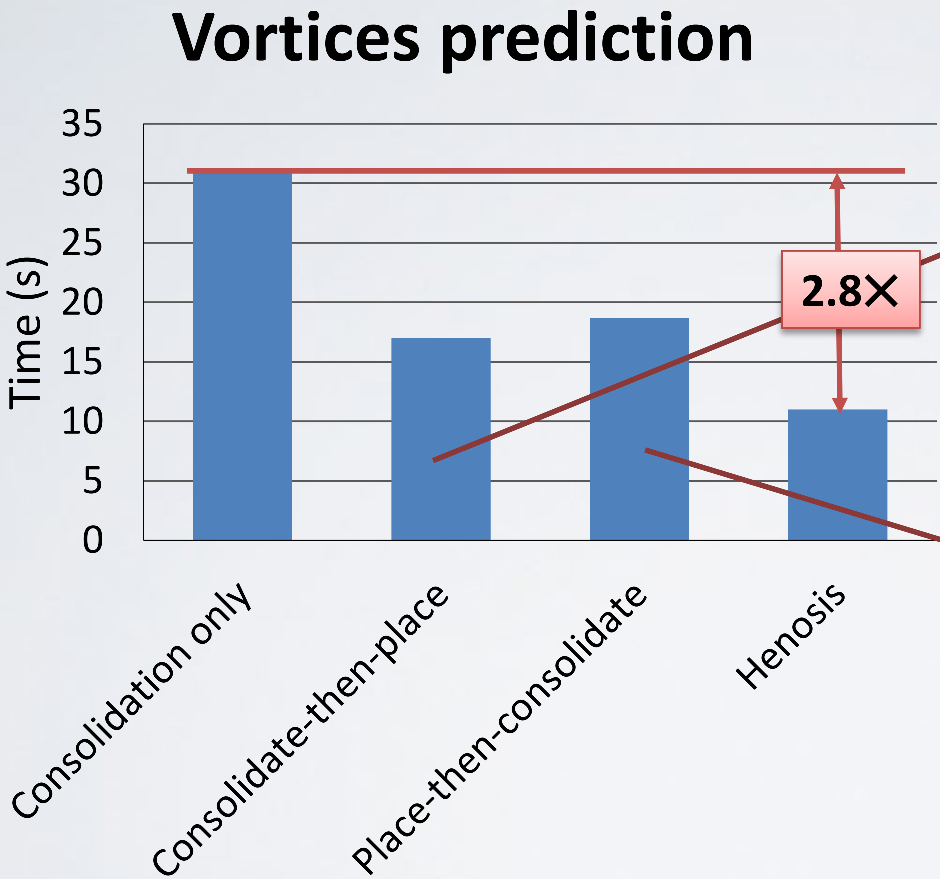
- Supernova detection
  - Dataset
    - 11,889 astronomy images
    - $21 \times 21$ pixels per image
  - Configuration
    - 9 nodes
- Vortices prediction
  - Dataset
    - 2000 timestamps
    - 160,000 vortices
    - 8KB/vortex
  - Configuration
    - 9 nodes

# Consolidation impact



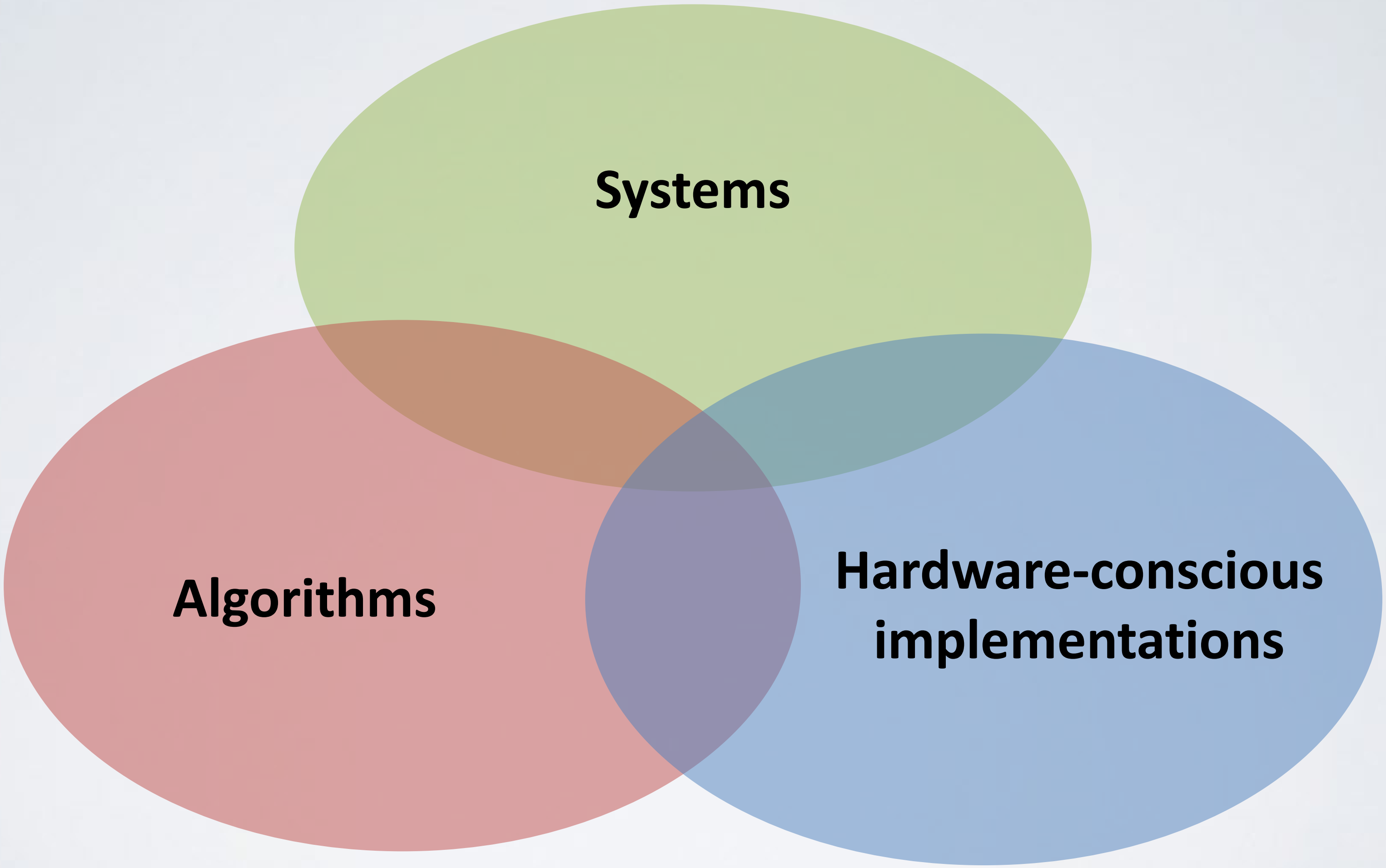
# Optimization impact




Consolidate-then-place tends to store unfrequently accessed small arrays in Redis

Place-then-consolidate tends to spread frequently co-accessed small arrays into two data stores

# Data processing at scale



# High-cardinality aggregation

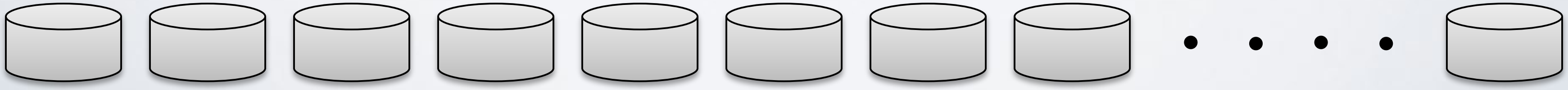
**Problem**

Aggregation based on repartitioning (1) does not use the network efficiently and (2) often transfers redundant data

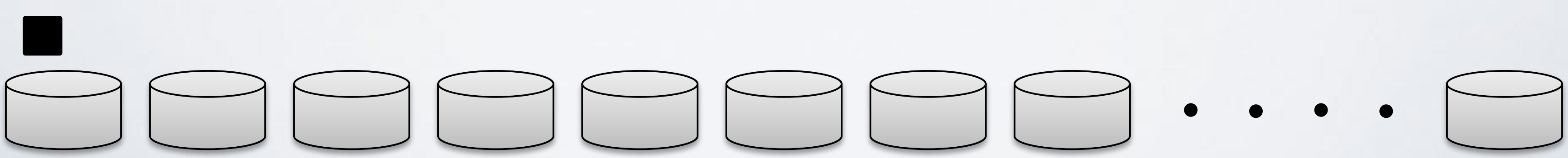
**Contribution [VLDB'19]**

GRASP, a **GR**eedy **A**ggregation **S**cheduling **P**rotocol: network-aware scheduling based on dataset similarity

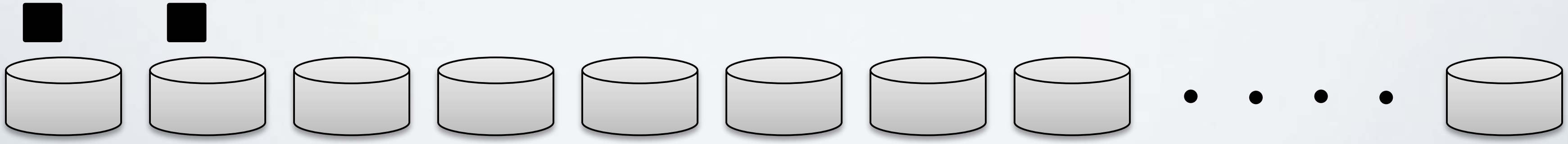
# Aggregation in MODIS



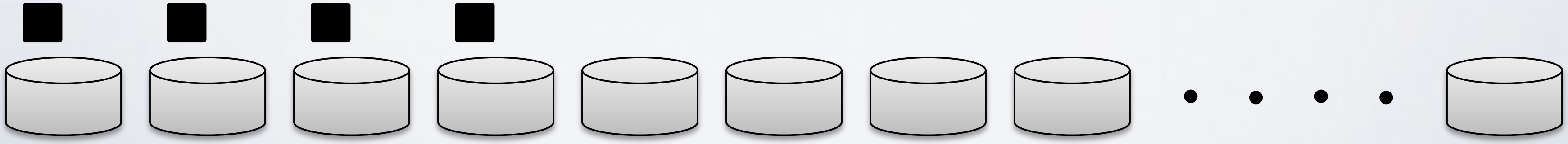
# Aggregation in MODIS

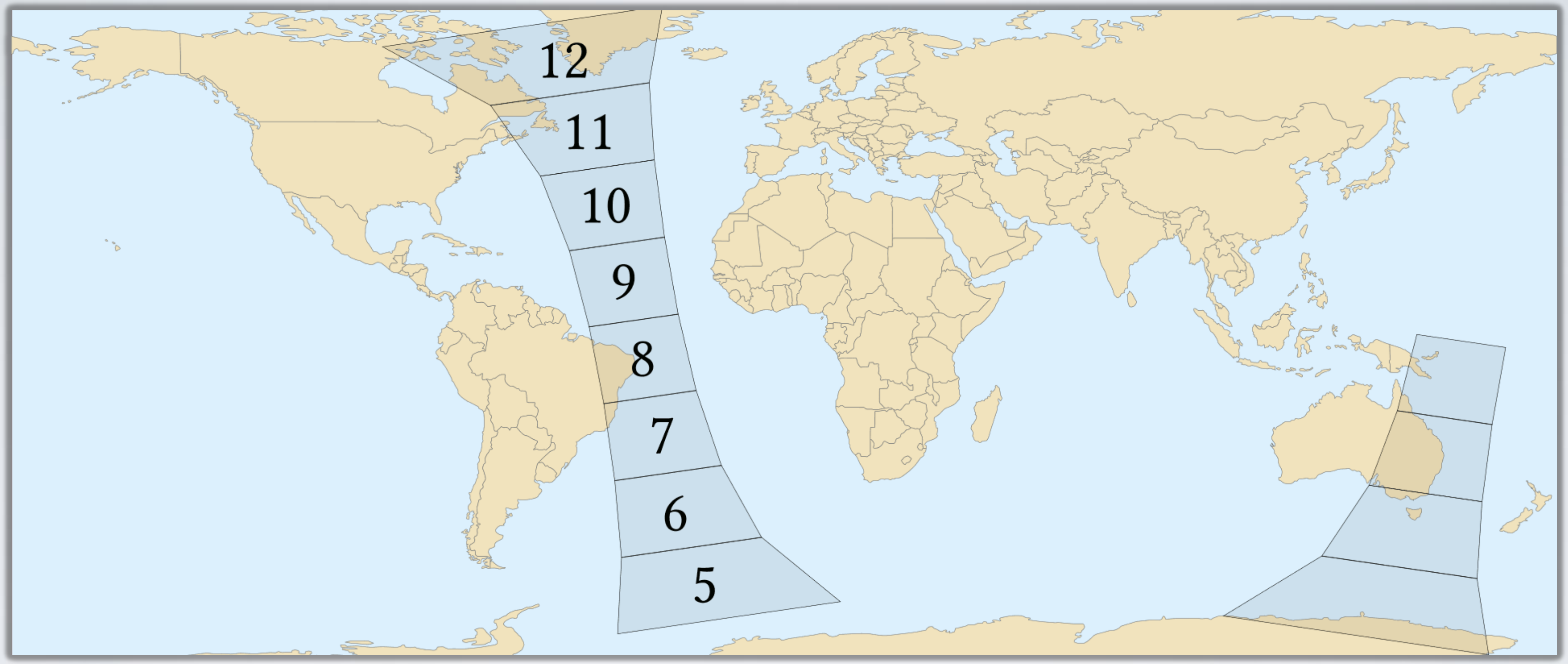
# Aggregation in MODIS

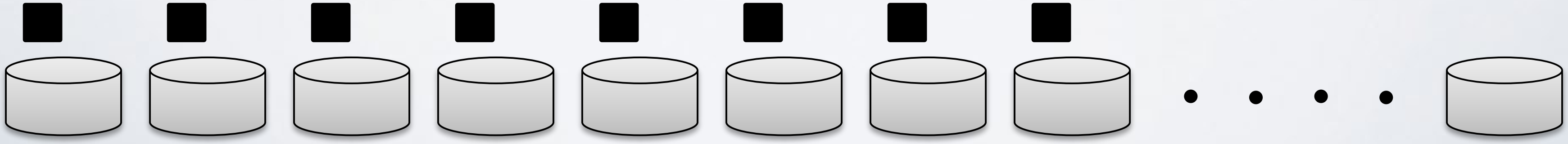



# Aggregation in MODIS

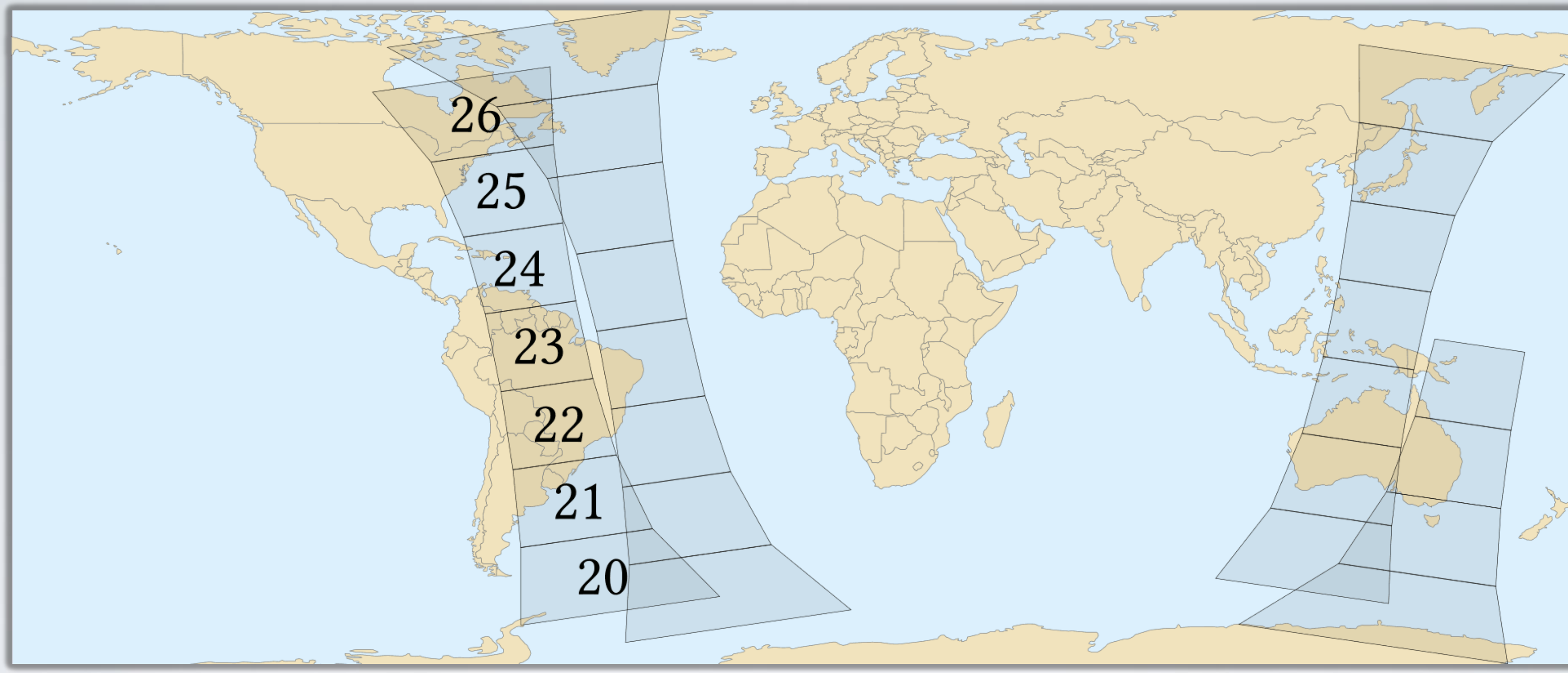



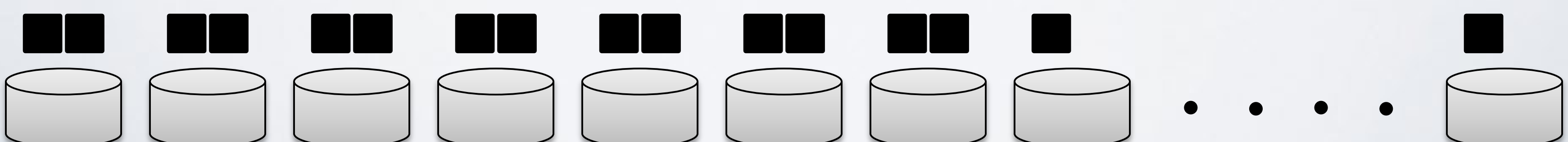
# Aggregation in MODIS



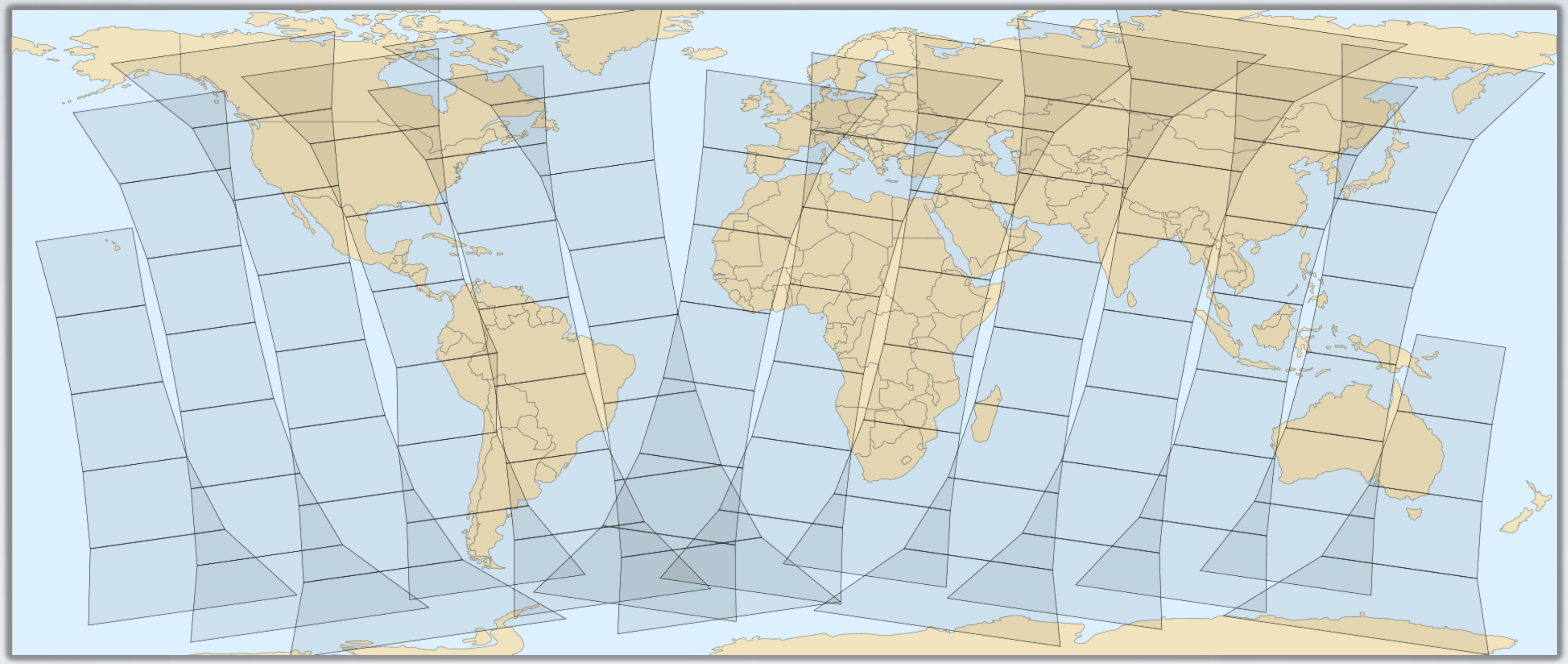



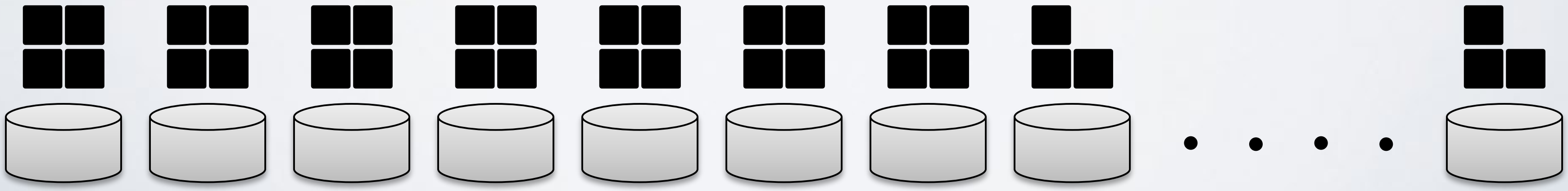
# Aggregation in MODIS



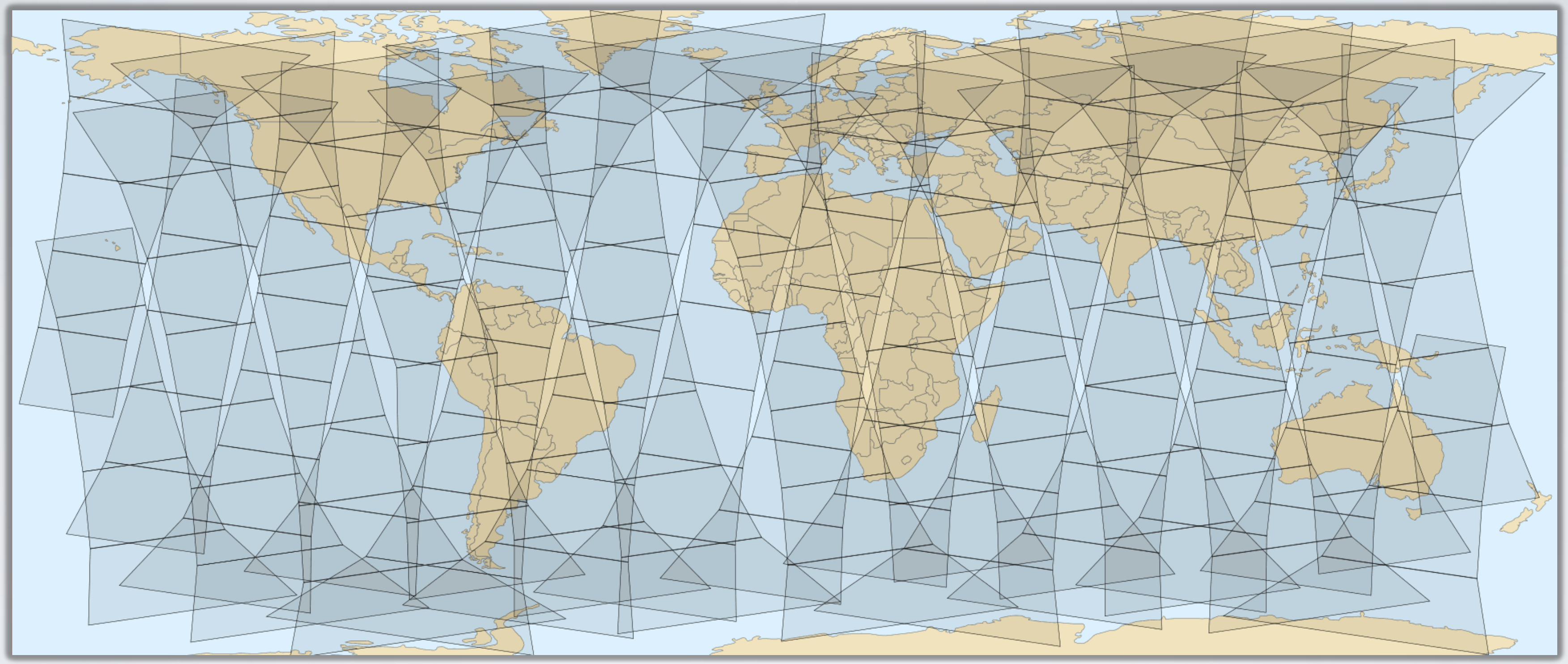



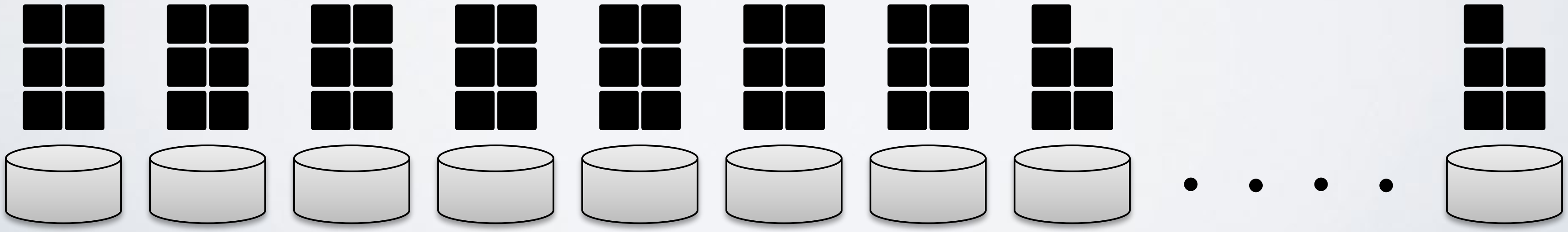
# Aggregation in MODIS



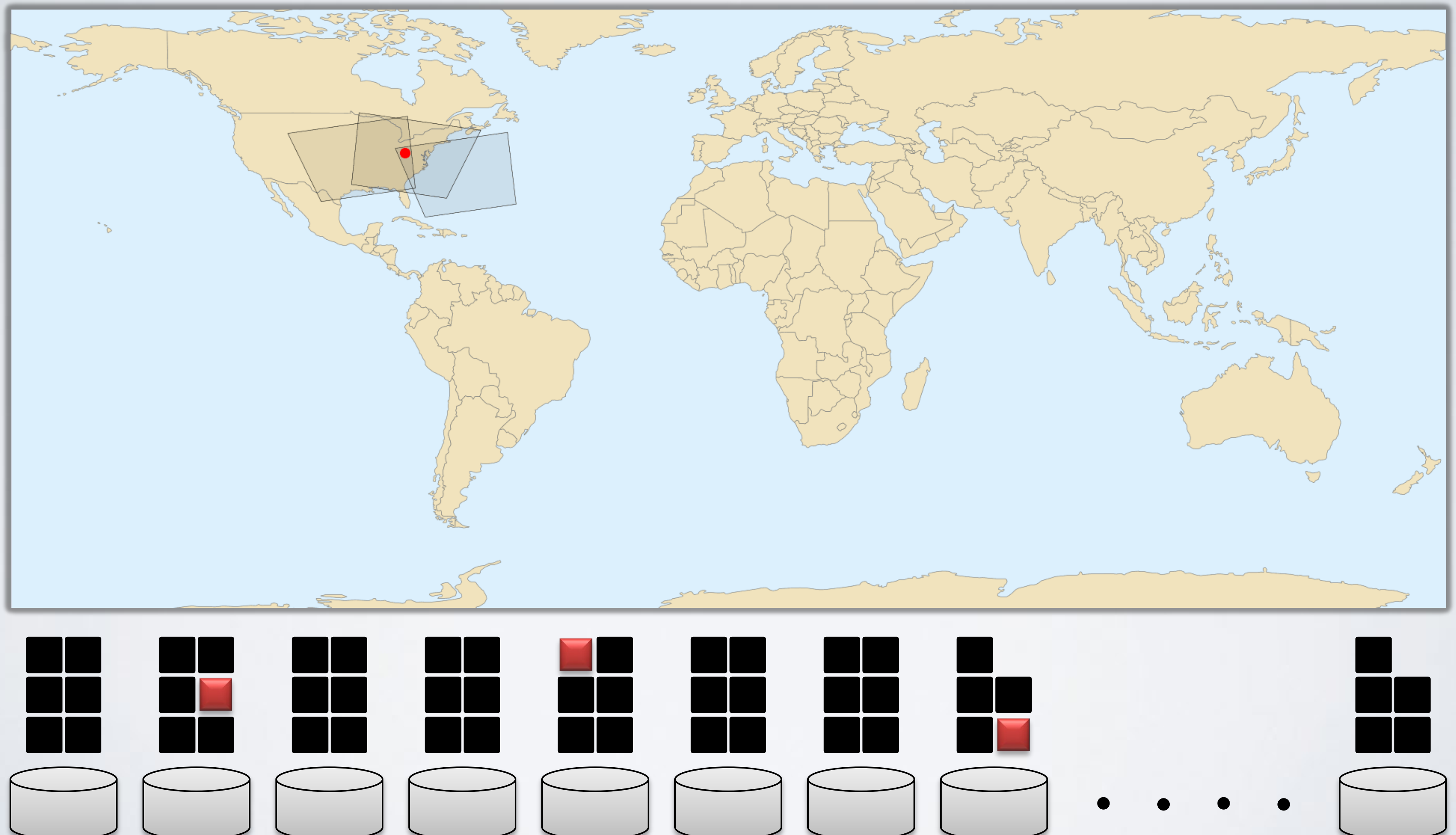


# Aggregation in MODIS

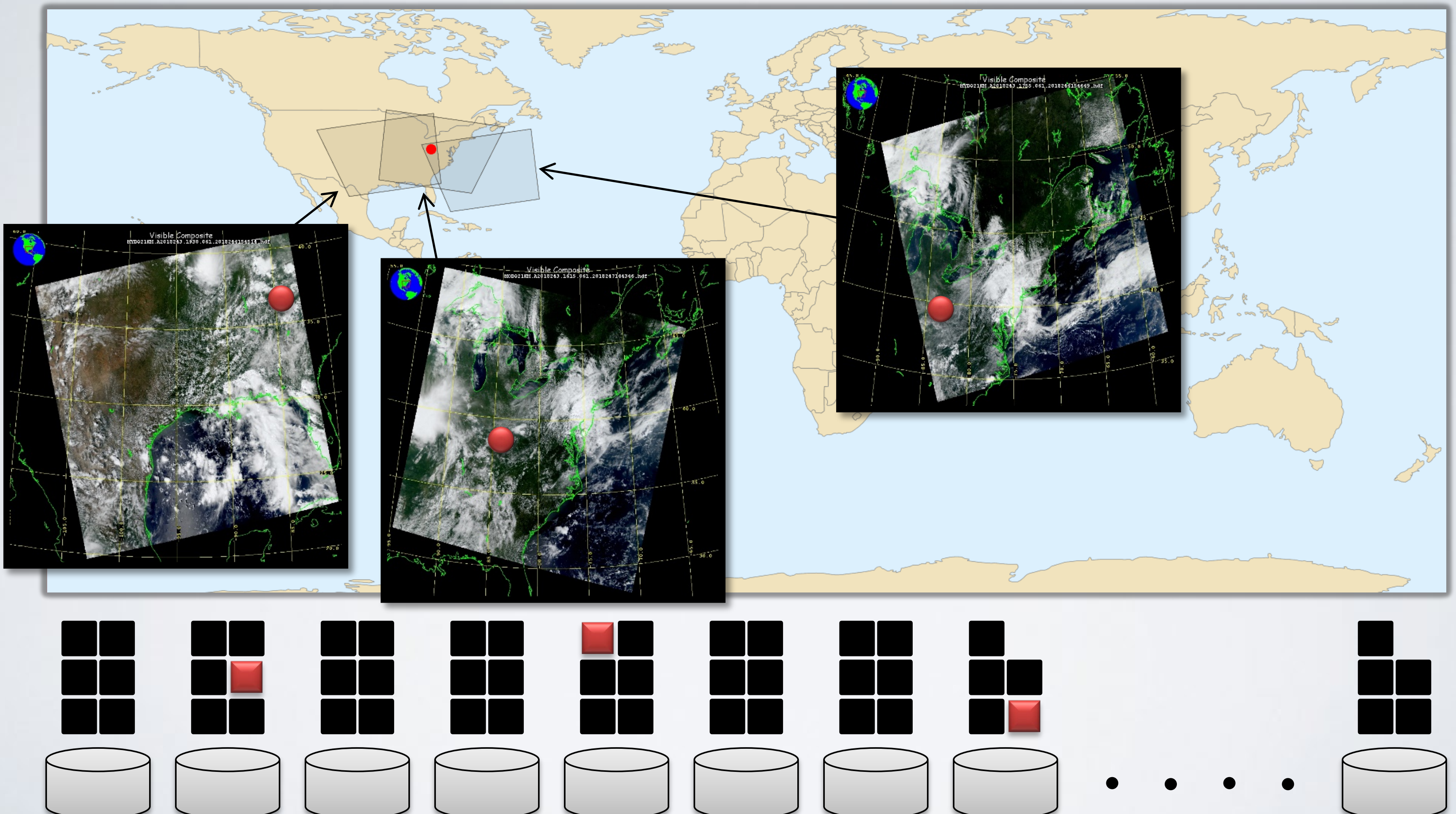




# Aggregation in MODIS

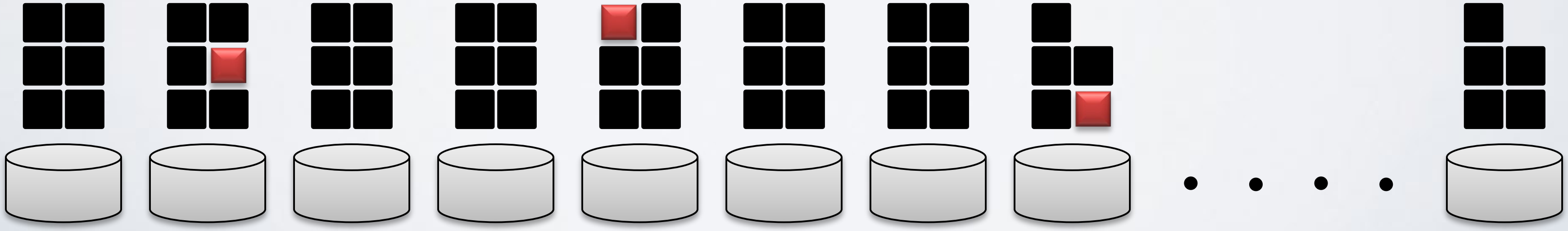


# Aggregation in MODIS



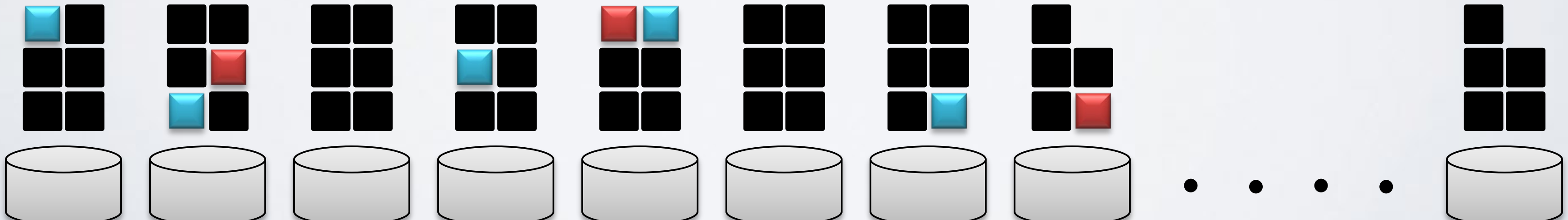
# Aggregation in MODIS

| Lat. | Lon. | |
|---|---|---|
| ... | ... | |
| -83.0 | 39.9 | |
| -83.0 | 40.0 | |
| -83.0 | 40.1 | |
| -83.0 | 40.2 | |
| ... | ... | |



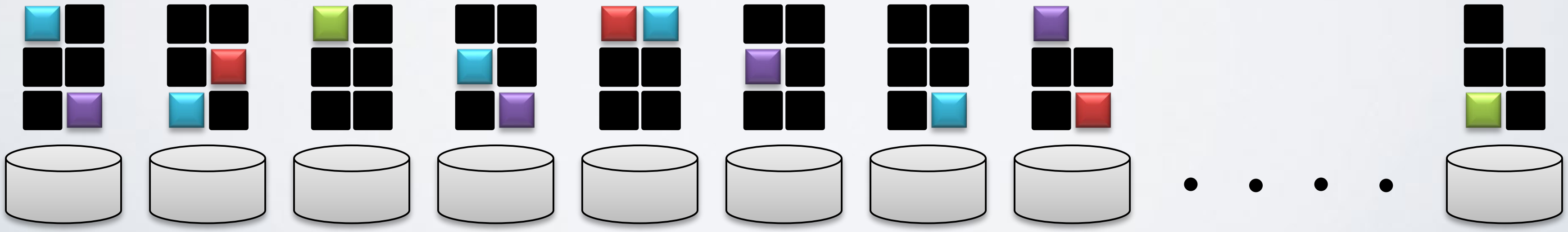
# Aggregation in MODIS

| Lat. | Lon. | |
|---|---|---|
| … | … | |
| -83.0 | 39.9 | |
| -83.0 | 40.0 | |
| -83.0 | 40.1 | |
| -83.0 | 40.2 | |
| … | … | |

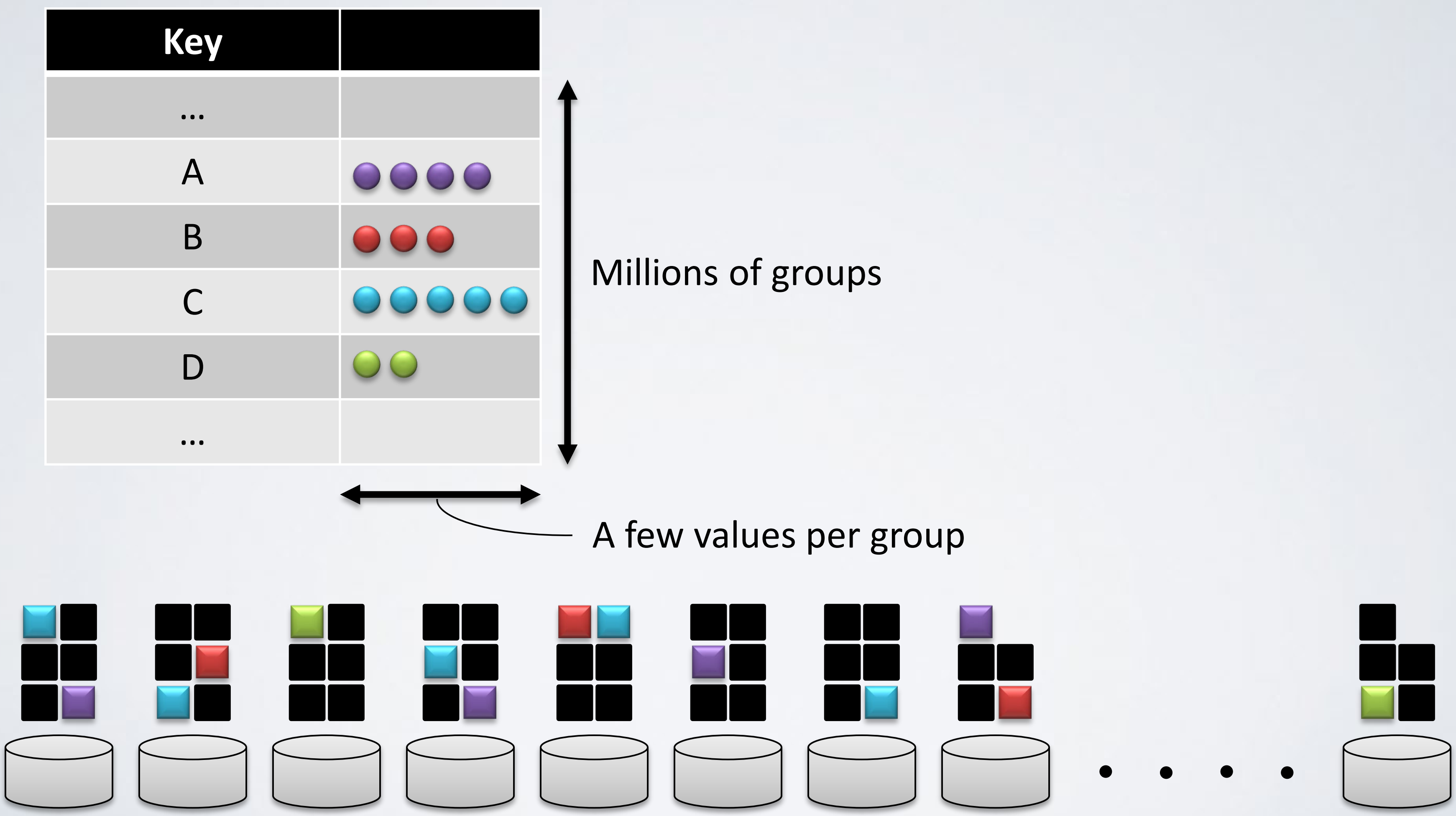


# Aggregation in MODIS

| Lat. | Lon. | |
|---|---|---|
| … | … | |
| -83.0 | 39.9 | |
| -83.0 | 40.0 | |
| -83.0 | 40.1 | |
| -83.0 | 40.2 | |
| … | … | |

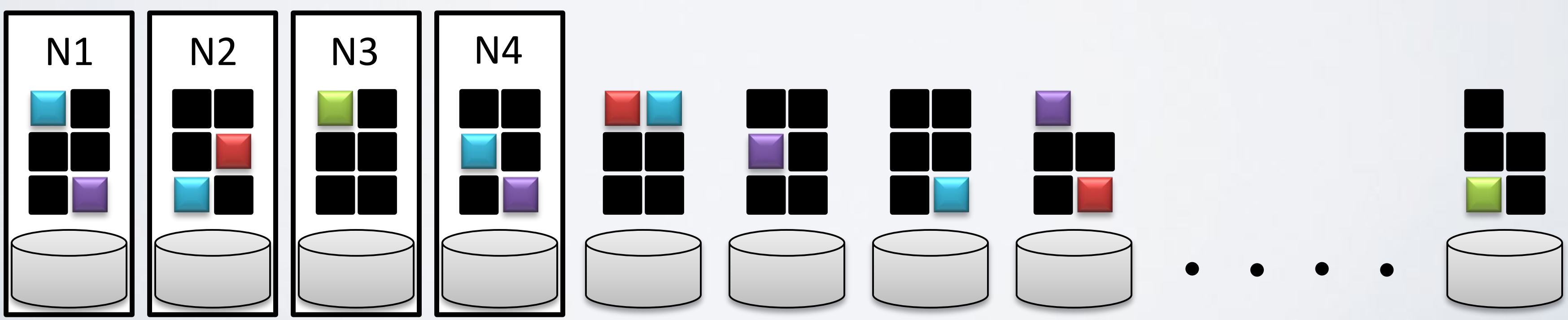


# Aggregation in MODIS



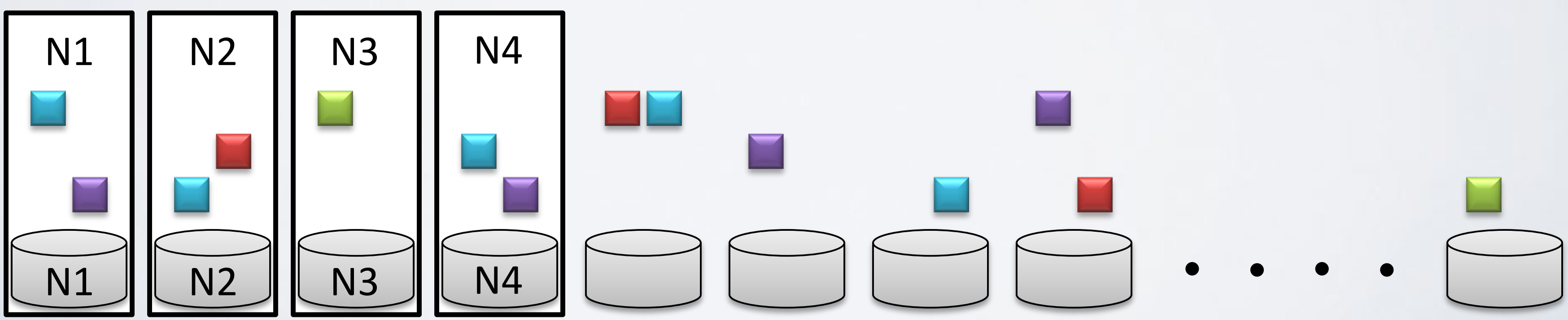
# Repartitioning during aggregation

| Key | | |
|---|---|---|
| … | | |
| A | | N1 |
| B | | N2 |
| C | | N3 |
| D | | N4 |
| … | | |



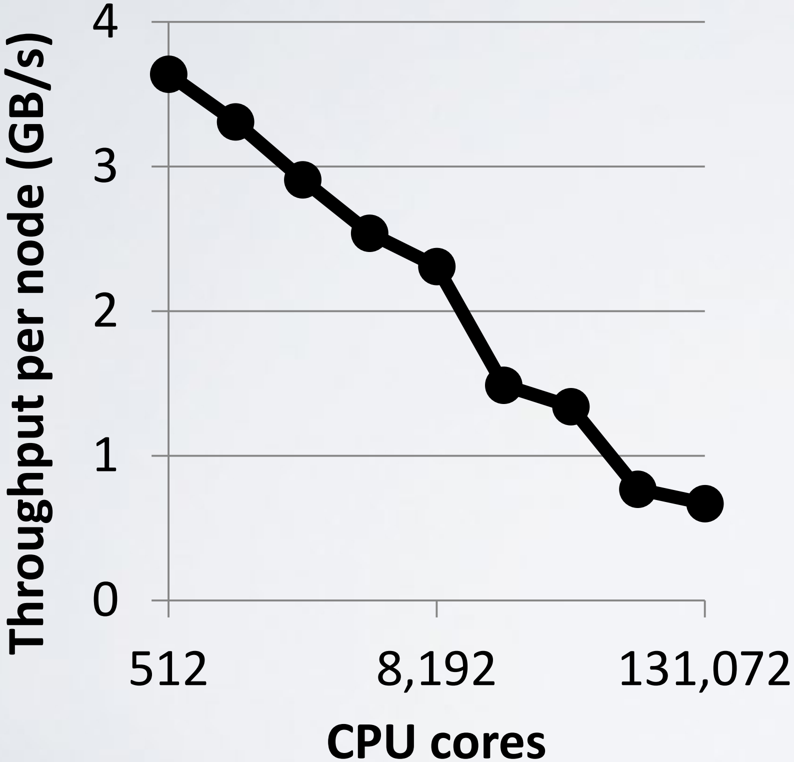
# Repartitioning during aggregation

| Key | | |
|---|---|---|
| … | | |
| A | | N1 |
| B | | N2 |
| C | | N3 |
| D | | N4 |
| … | | |



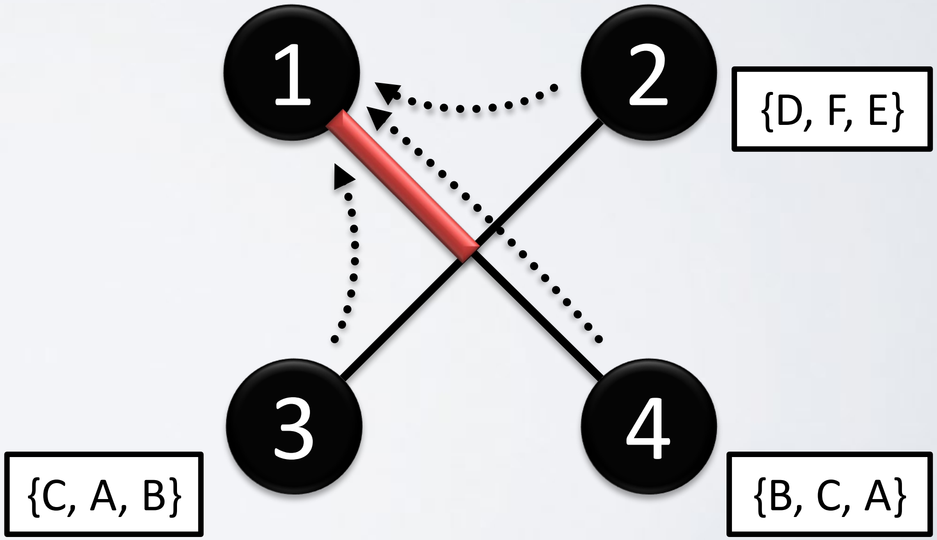
# Problems with repartitioning

**1. All-to-all does not scale**



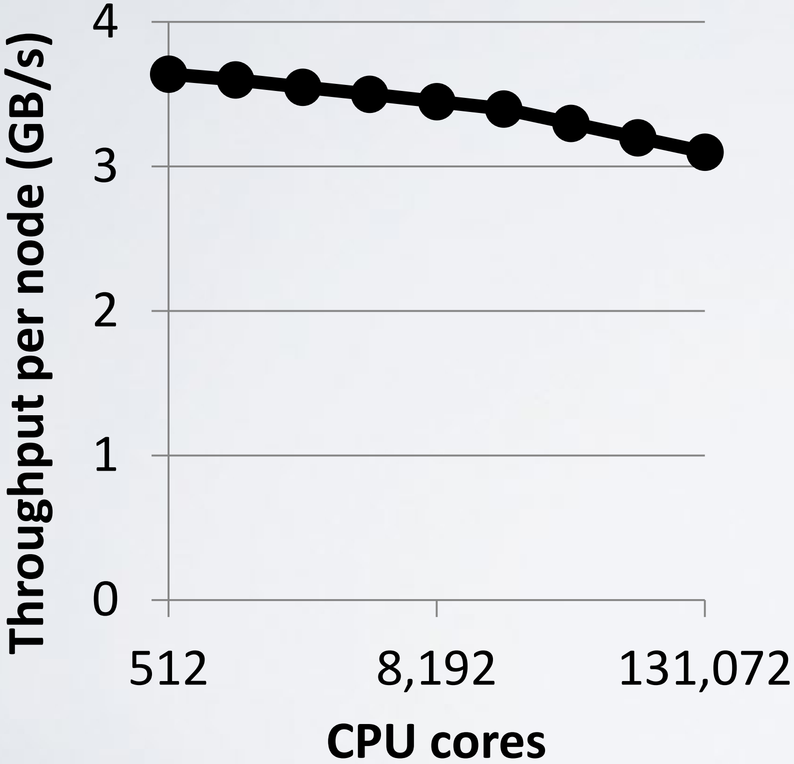

**2. Destination is overwhelmed**

| Communication | Time |
|---|---|
| 2 -> 1; 3 -> 1; 4 -> 1 | 9 |



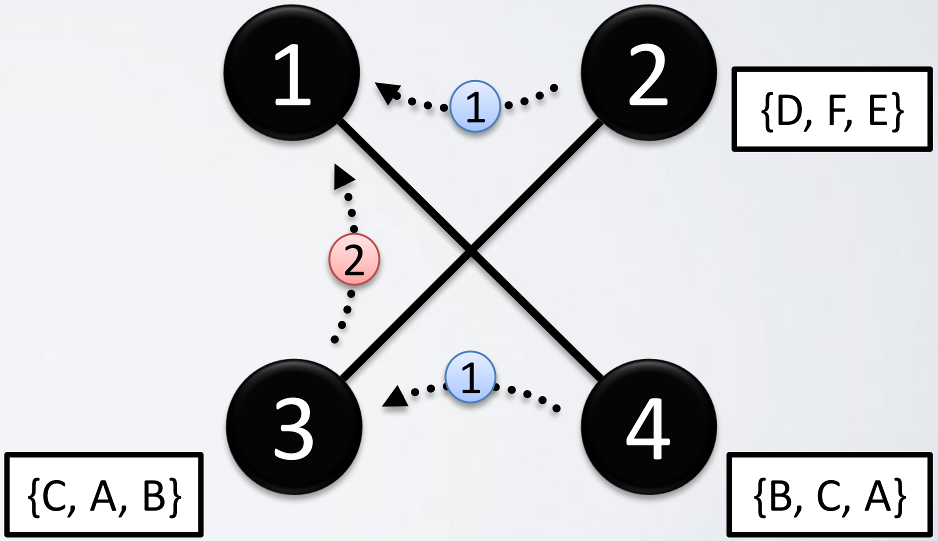
# GRASP: a GReedy Aggregation Scheduling Protocol

**1. Communicates in pairs**




**2. Leverages similarity**

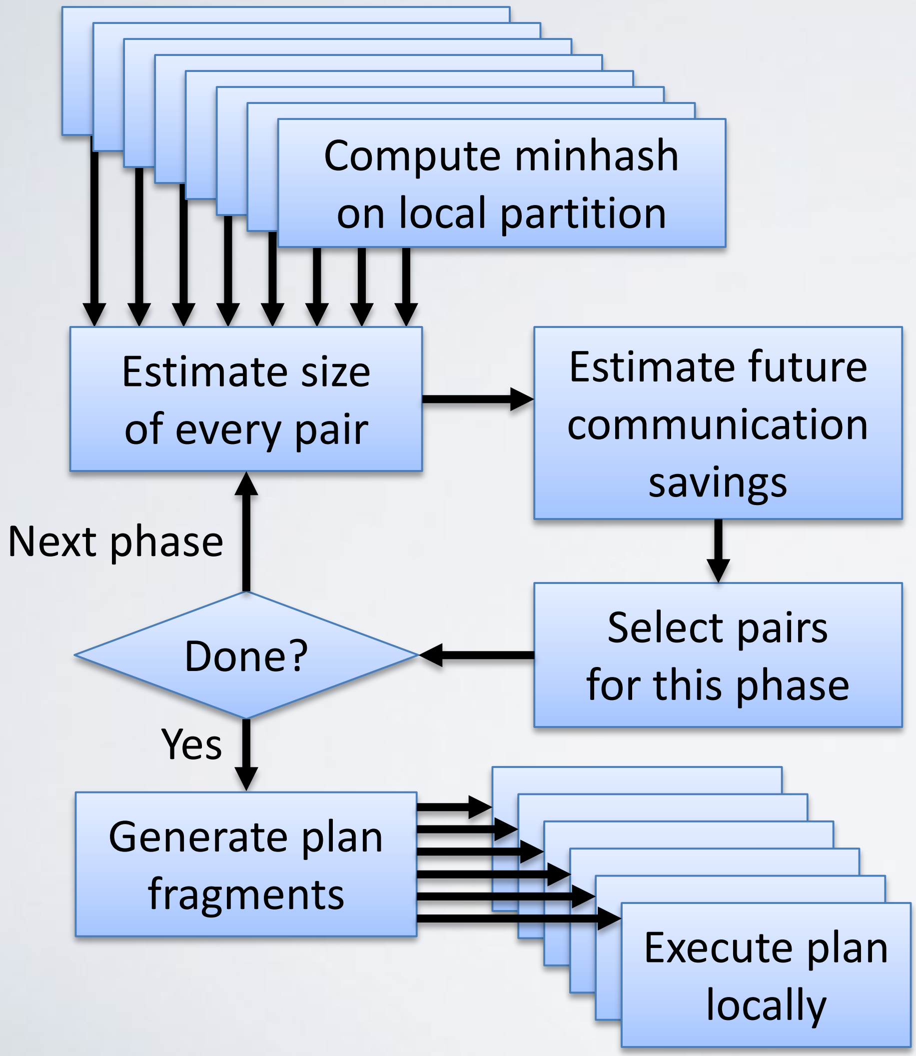
| Phase | Communication | Time |
|---|---|---|
| 1 | 2 -> 1; 4 -> 3 | **6** |
| 2 | 3 -> 1 | |

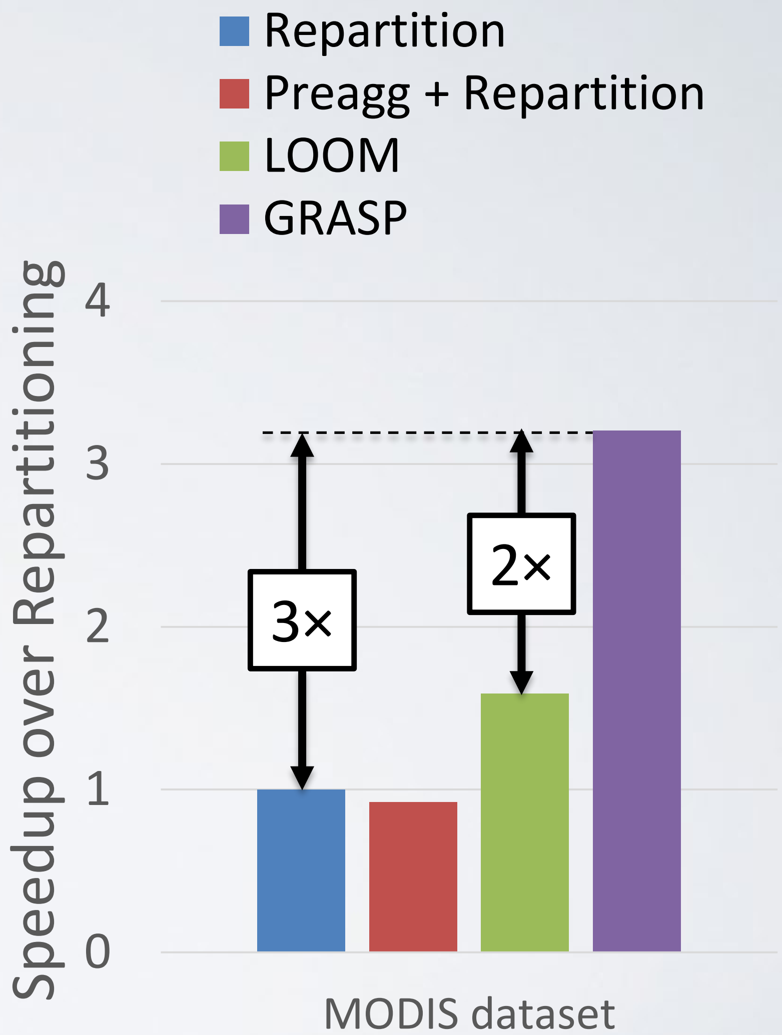


# GRASP

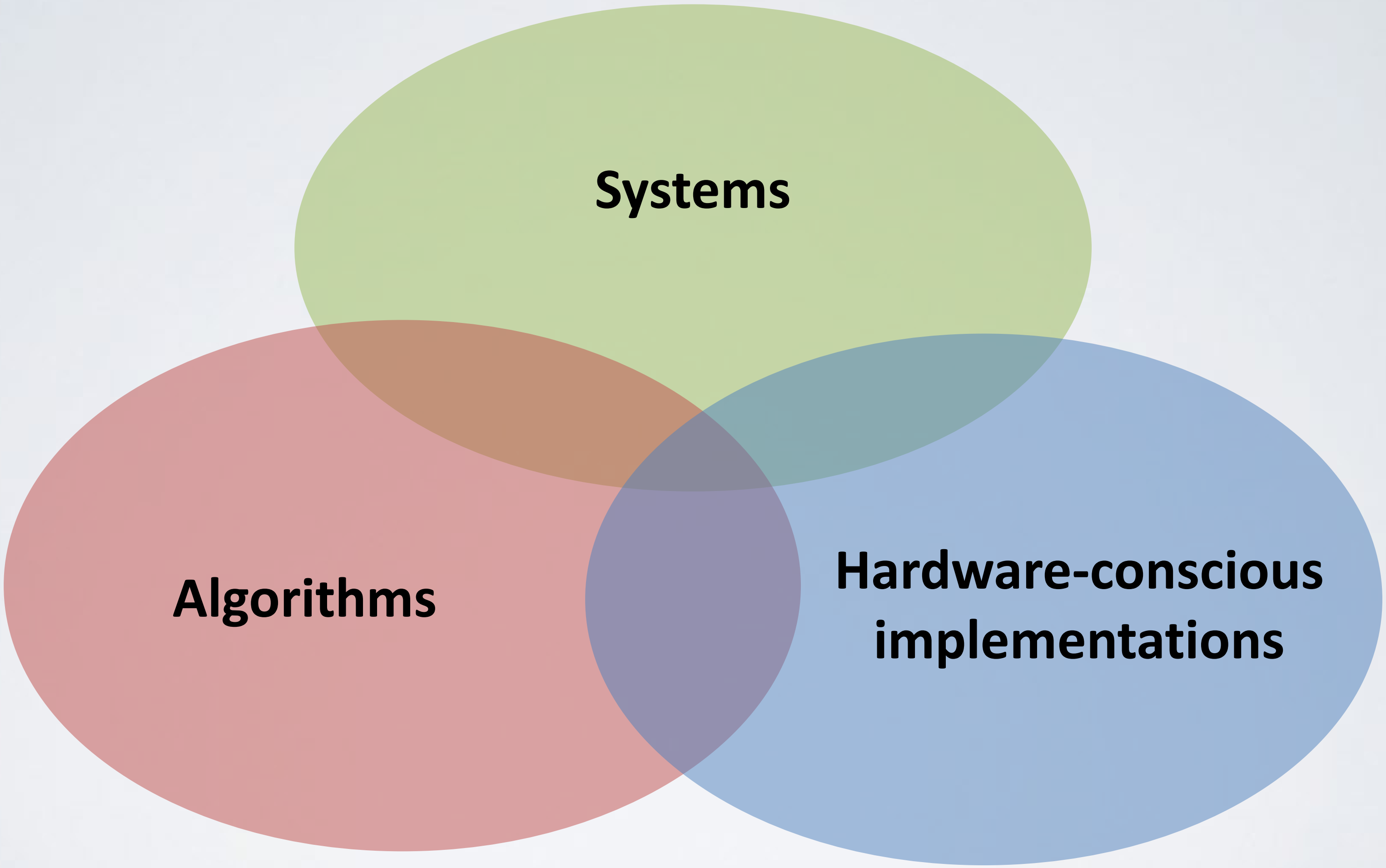
- SSE-hard to approximate optimal aggregation
  - At least as hard as Small Set Expansion problem
  - Hard to approximate within any constant factor, assuming SSE is hard to approximate
- GRASP is a heuristic that builds an aggregation tree based on data similarity
  - Prior work (LOOM, Orchestra) focuses on the network, not the data distribution

# GRASP

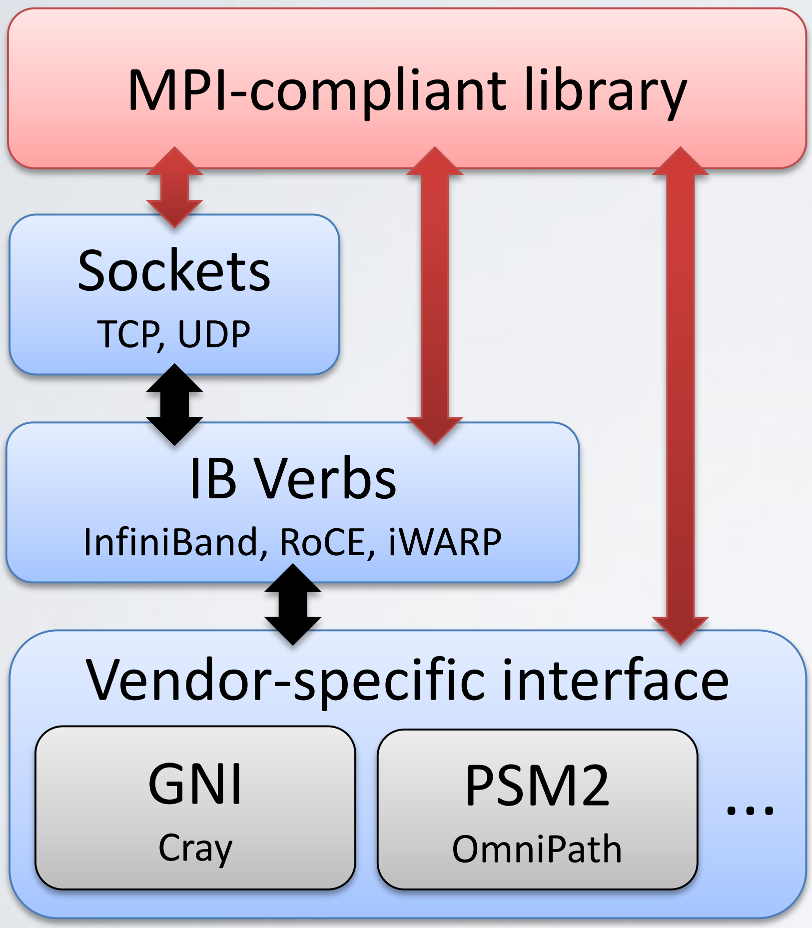

Compute minhash on local partition
Estimate size of every pair
Estimate future communication savings
Next phase
Done?
Select pairs for this phase
Yes
Generate plan fragments
Execute plan locally


Repartition
Preagg + Repartition
LOOM
GRASP
Speedup over Repartitioning
4
3
2
1
0
3×
2×
MODIS dataset

# Data processing at scale



# How to use a fast NIC?




+ Acceptable performance
+ Standardized API
- Opaque memory management
- Brittle performance (“user error”)

+ Ubiquitous
- Poor performance

+ Good performance
+ Standardized interface
- Limited application surface

+ Best performance
+ Rich feature set
- Engineering effort vs. benefit
- Vendor lock-in concerns

# Data shuffling with RDMA

**Myth #2**

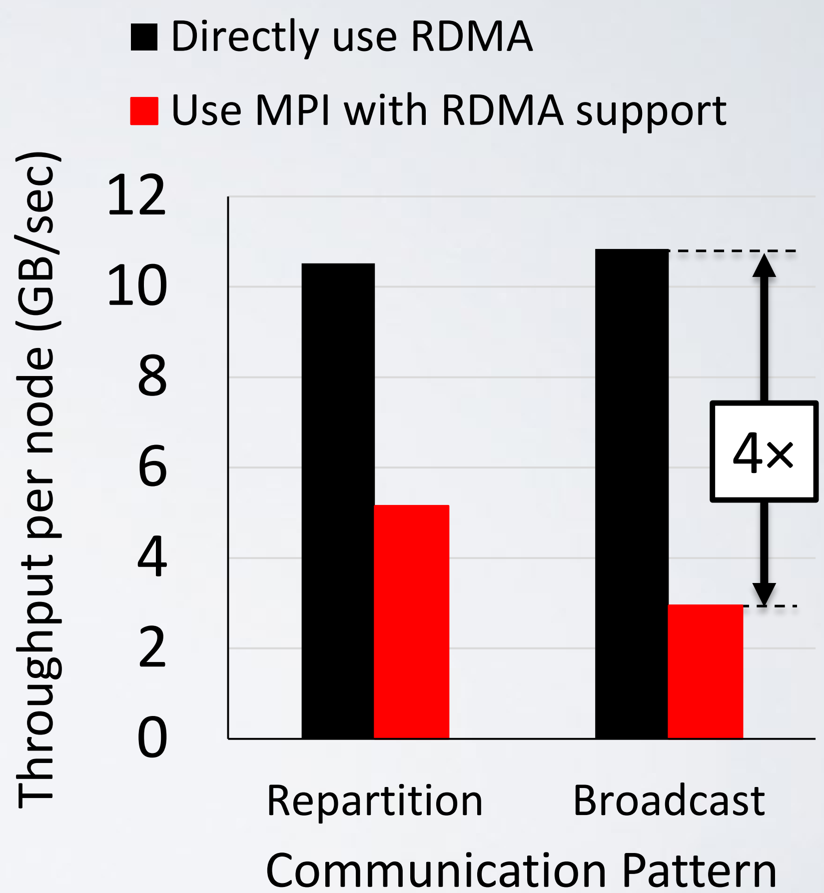
"MPI already uses RDMA, one can't go much faster"

**Contribution [EuroSys'17]**

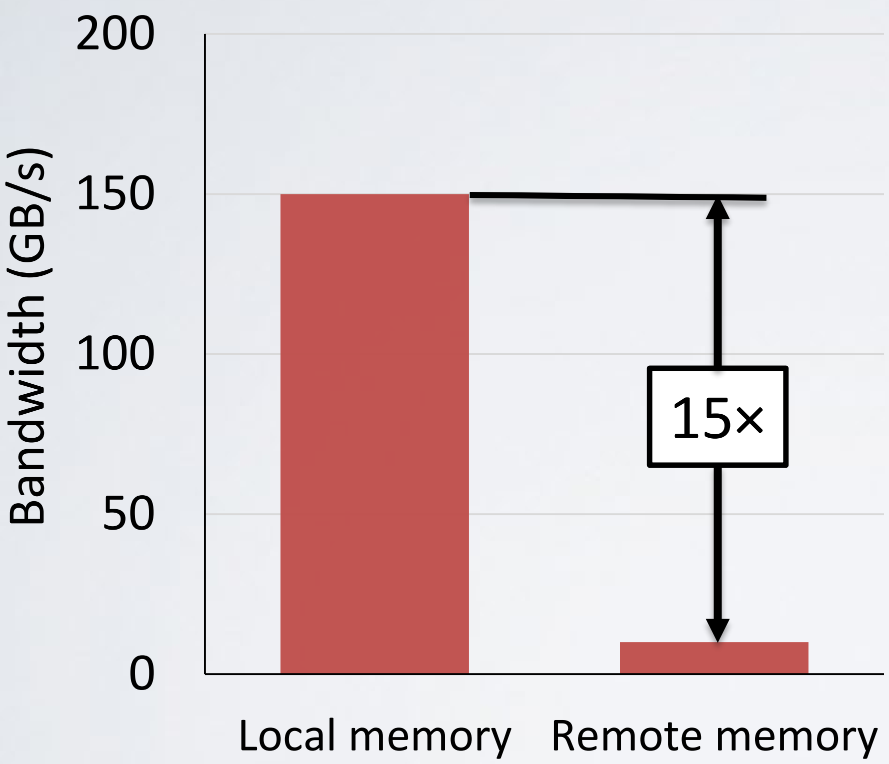
RDMA-based exchange operator, 4× faster than MPI

**Opportunity ahead**

Communication abstractions for data-intensive computing

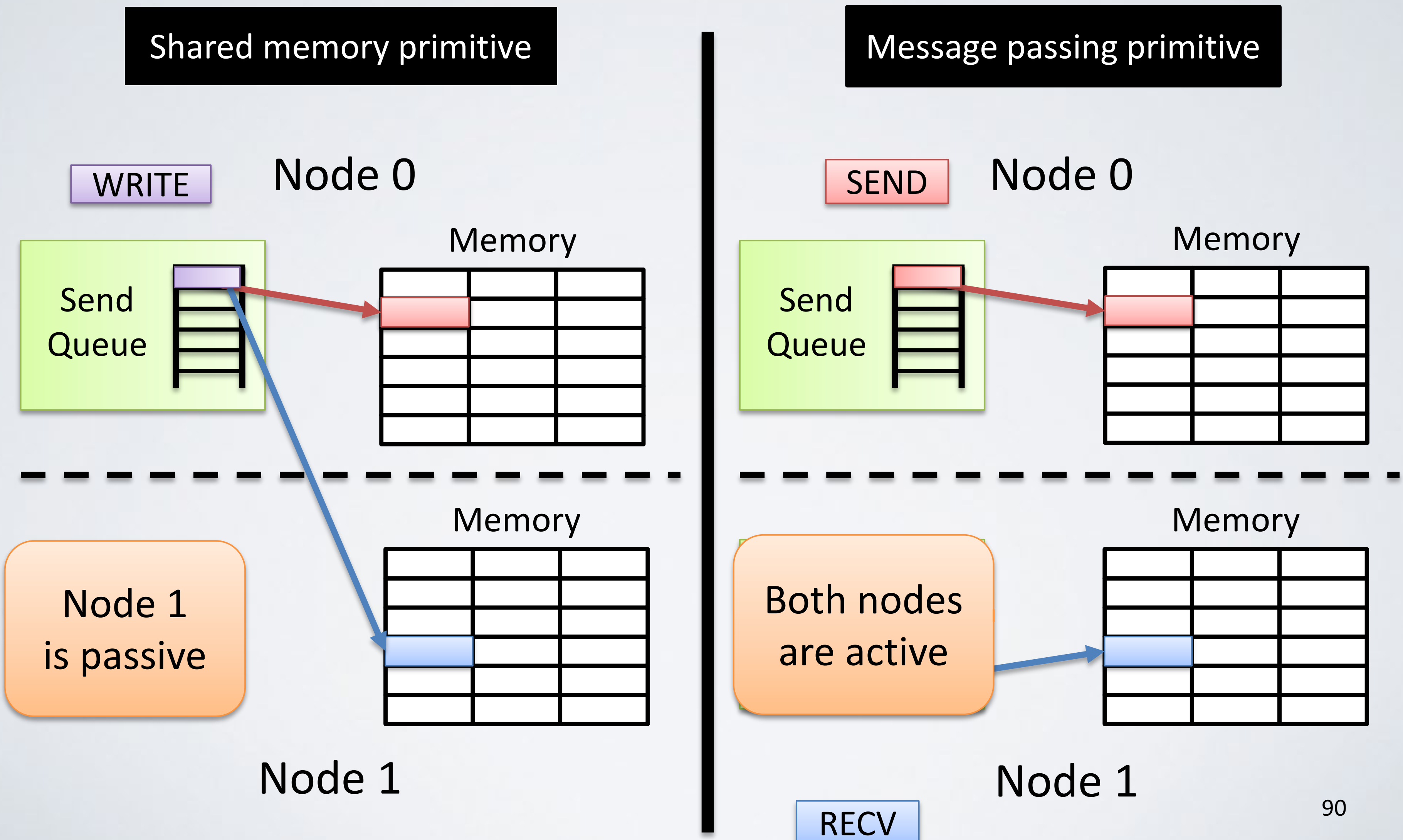


# Scaling the network is expensive




- The bottleneck is often network throughput
- Goal: Transfer data at line rate

# RDMA background



# Key questions from prior work

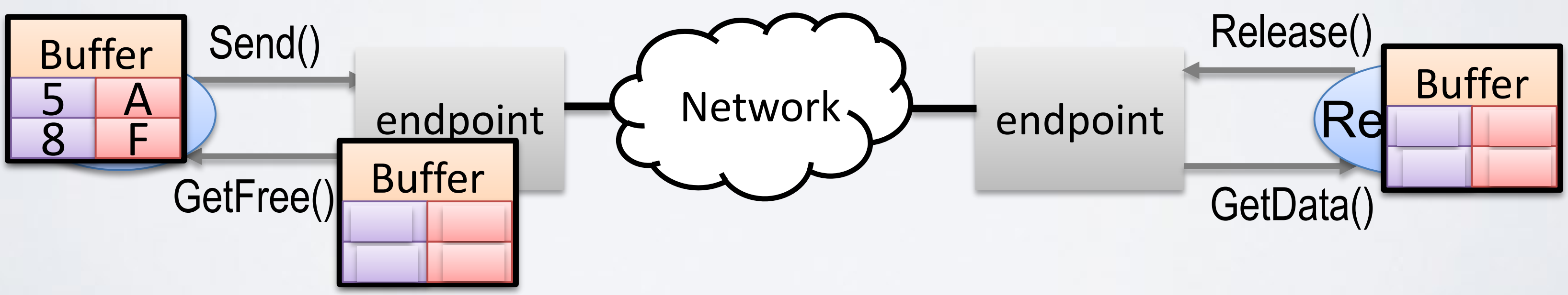
- Chen et al. [EuroSys'16] → Can one-sided primitives help?
- Kalia et al. [OSDI'16] → Is unreliable delivery tolerable?
- Barthels et al. [SIGMOD'15, VLDB'17] →
  - Is MPI good enough?
  - How to accelerate all queries, and not just joins?
- Rodiger et al. [VLDB'16] → How to avoid contention for the communication multiplexer?

# Challenges

- Isolate the complexity for RDMA
  - Manage memory registration
  - Anticipate packets may arrive out of order
  - Support different implementations: RDMA, MPI, IPoIB, …
- Identify promising design choices
  - Compare both two-sided and one-sided primitives
  - Consider both UD and RC transport
    - Balance between number of Queue Pairs and thread contention

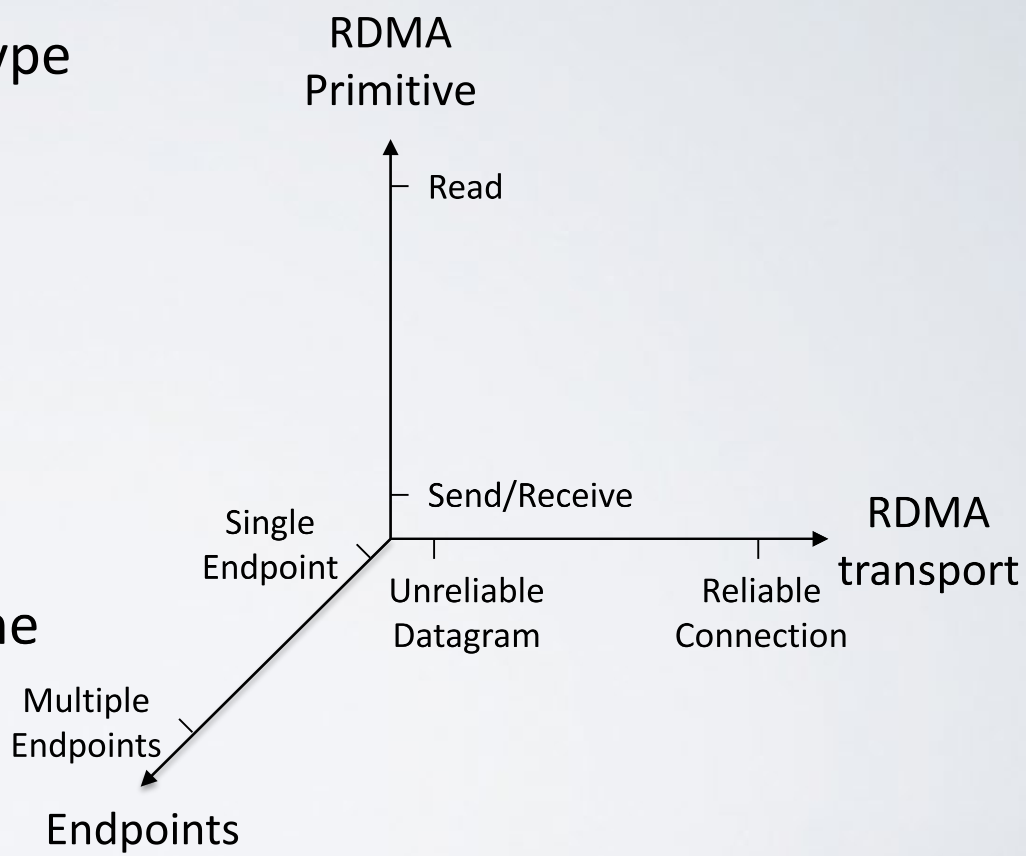
# The endpoint abstraction

- The endpoint hides the complexity of synchronization and memory management in RDMA communication
  - A uniform abstraction for communication
- One shuffle operator can have one or multiple endpoints
- All functions are thread-safe

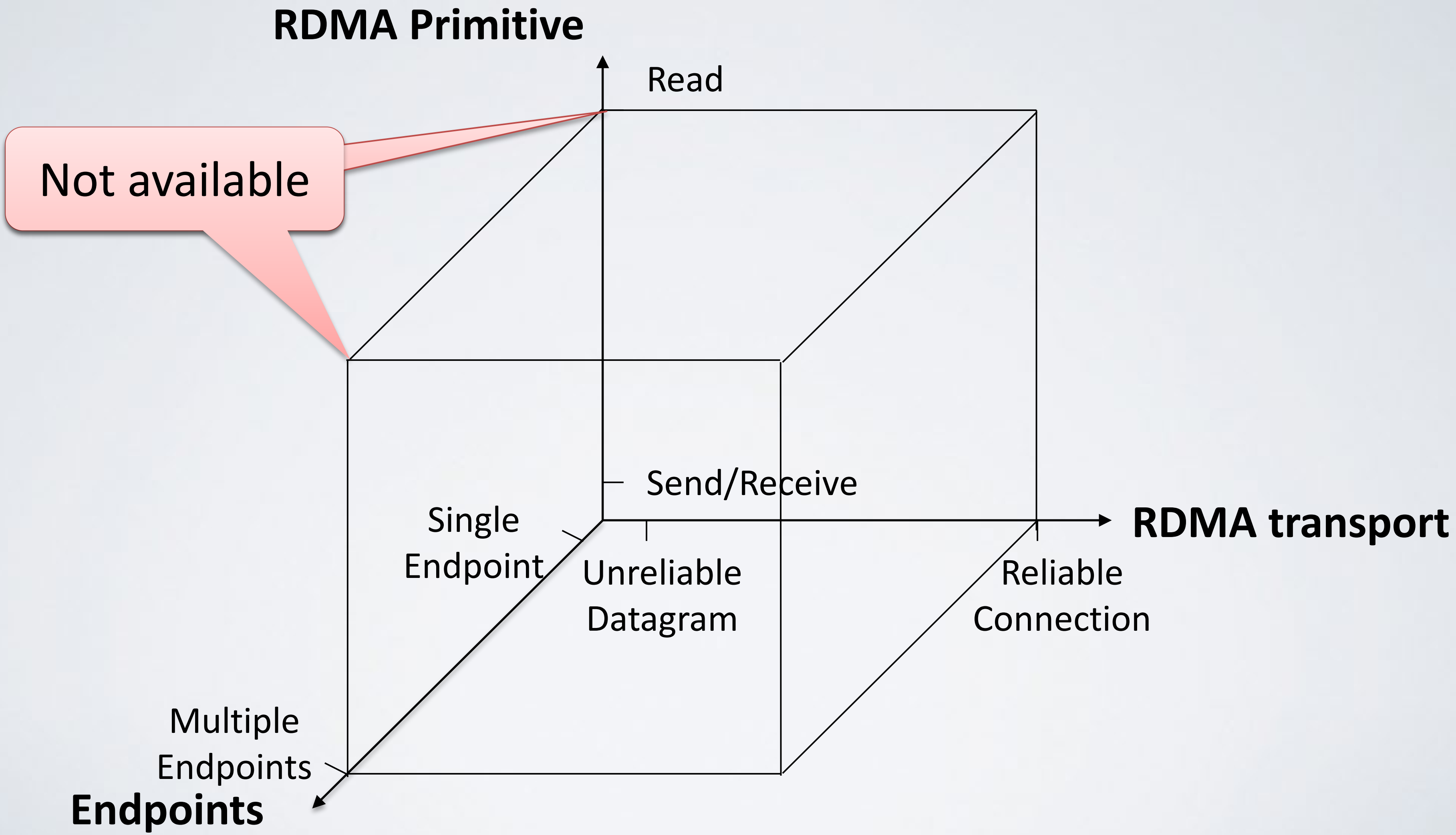


# Design choices

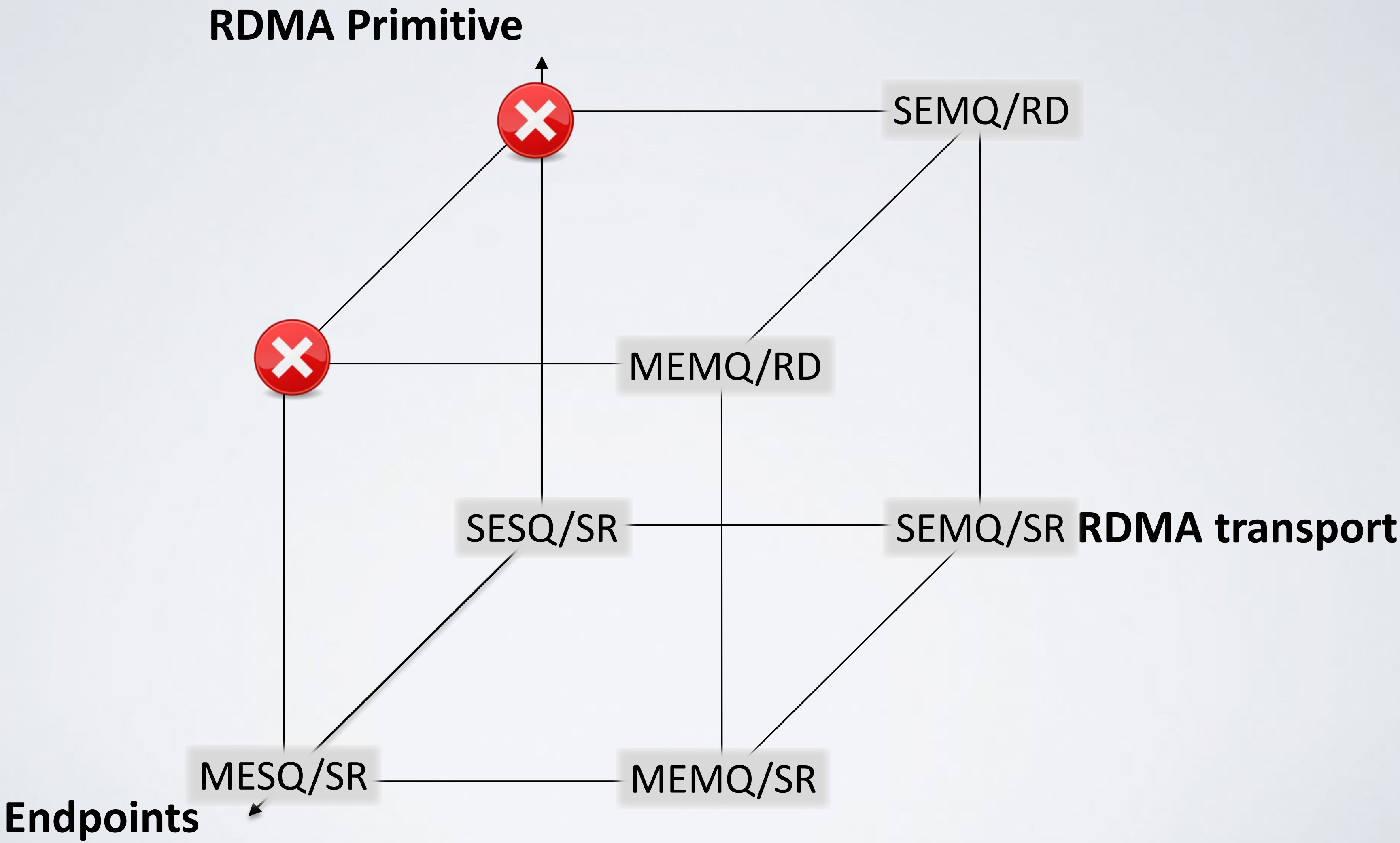
- RDMA transport service type
  - Unreliable Datagram (UD)
  - Reliable Connection (RC)
- RDMA primitive
  - RDMA Send/Receive (SR)
  - RDMA Read (RD)
- How many endpoints in the shuffling operator
  - Single Endpoint (SE)
  - Multiple Endpoints (ME)



# Design choices

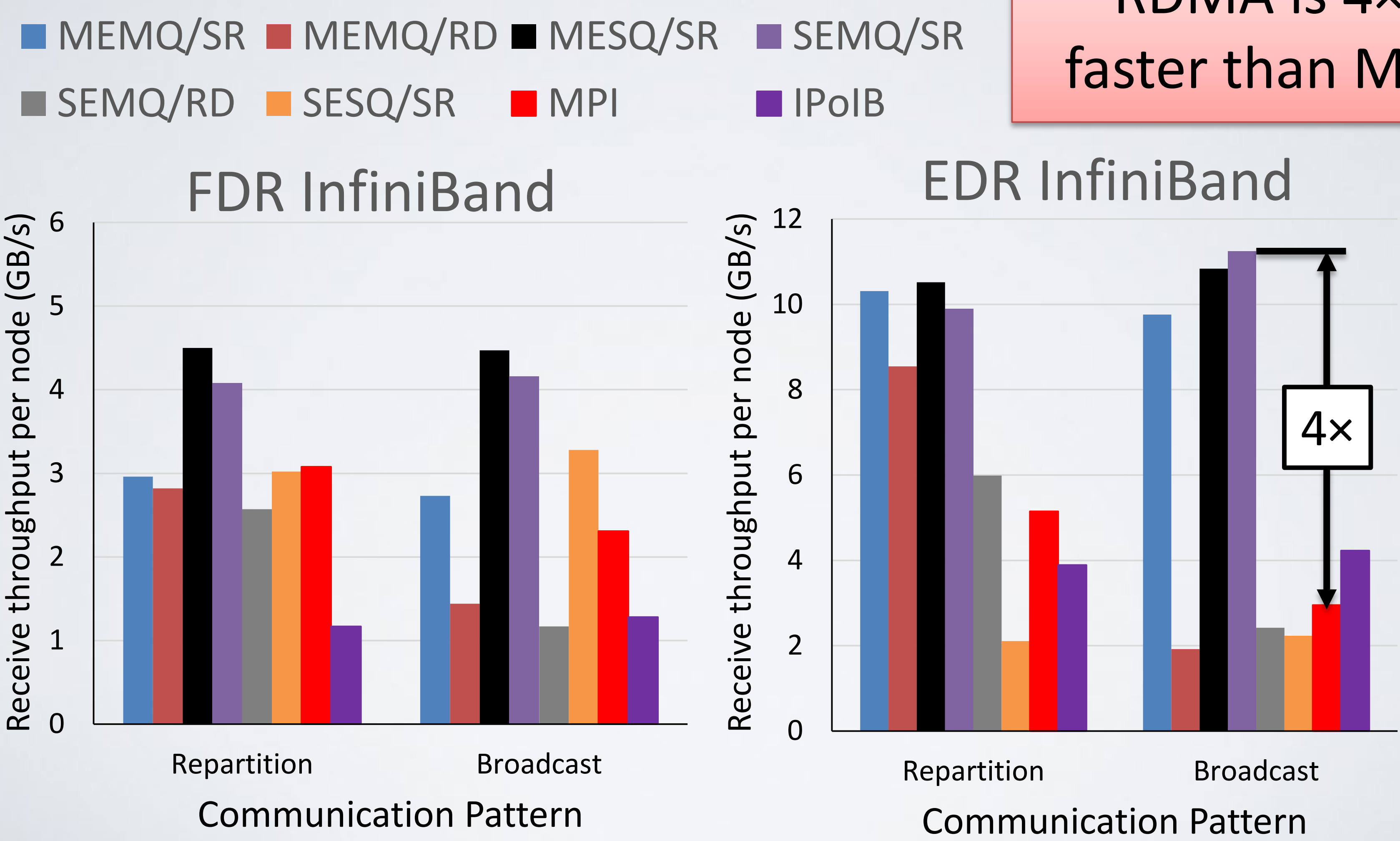


# Algorithms


RDMA Primitive
SEMQ/RD
MEMQ/RD
SESQ/SR
SEMQ/SR
RDMA transport
MESQ/SR
MEMQ/SR
Endpoints

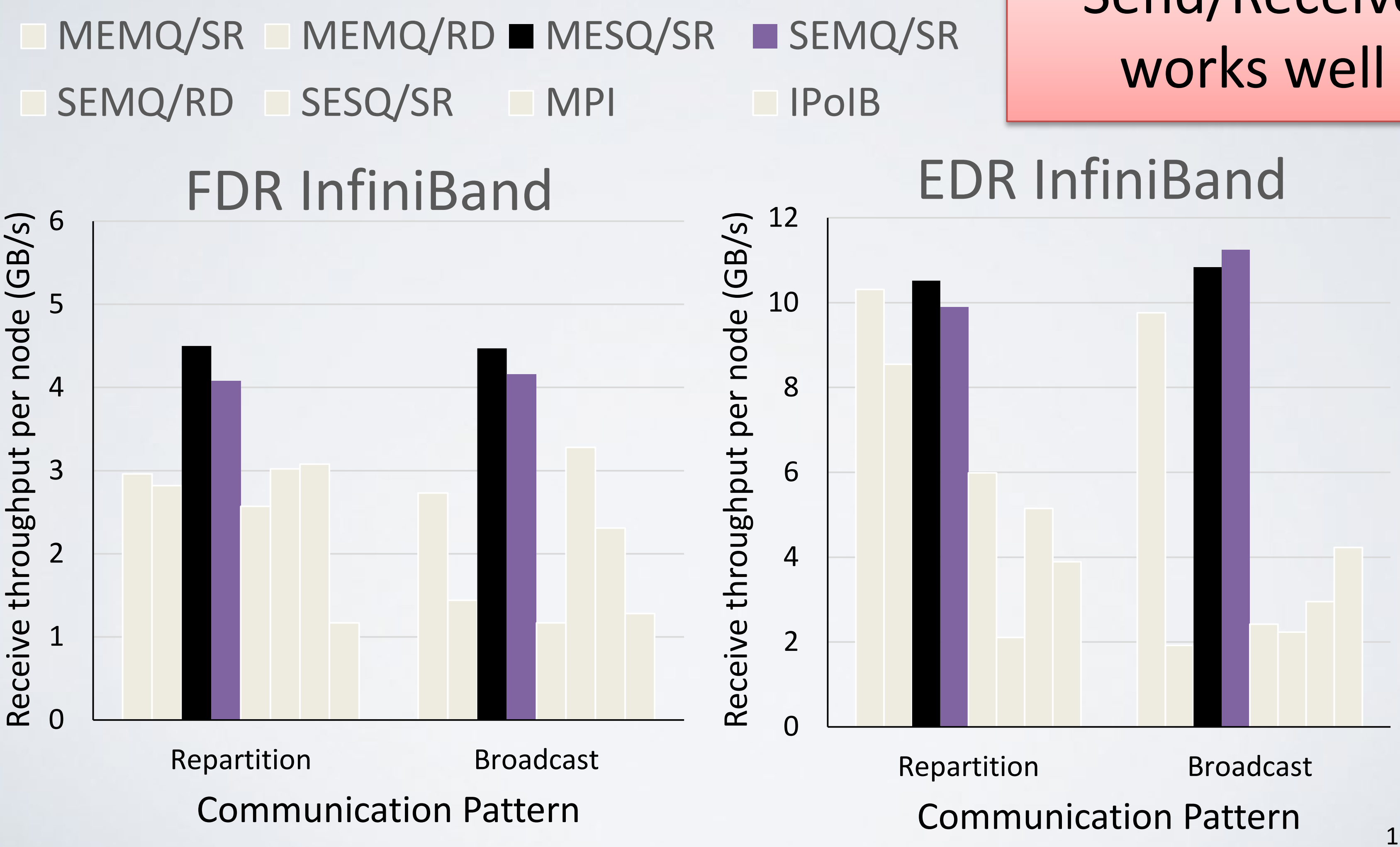
# Throughput comparison (16 nodes)

RDMA is 4× faster than MPI

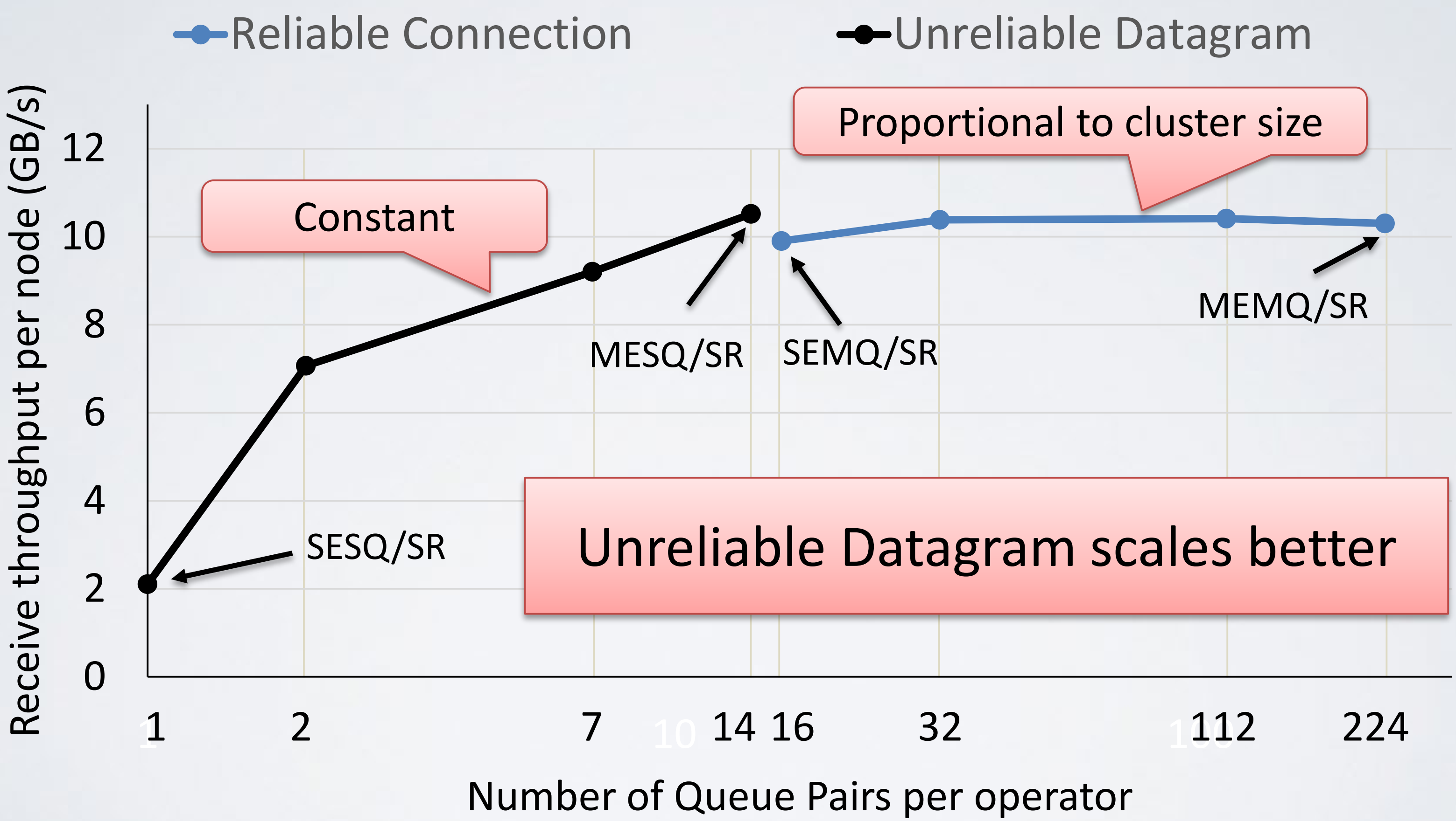


# Throughput comparison (16 nodes)

Send/Receive works well

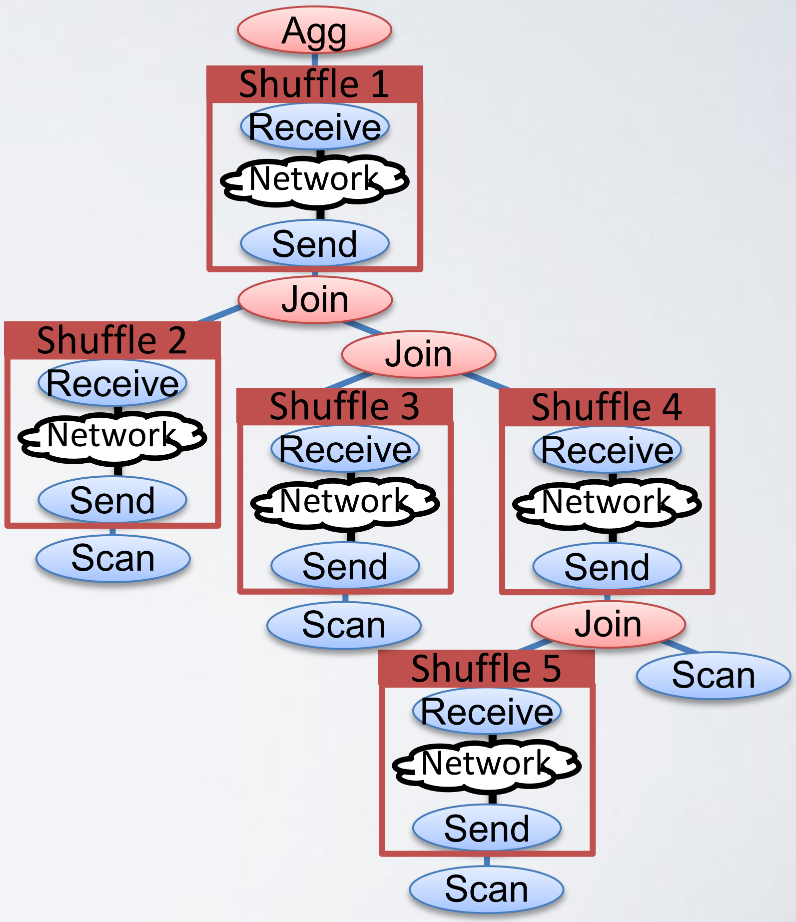


# Resources needed (16 nodes)



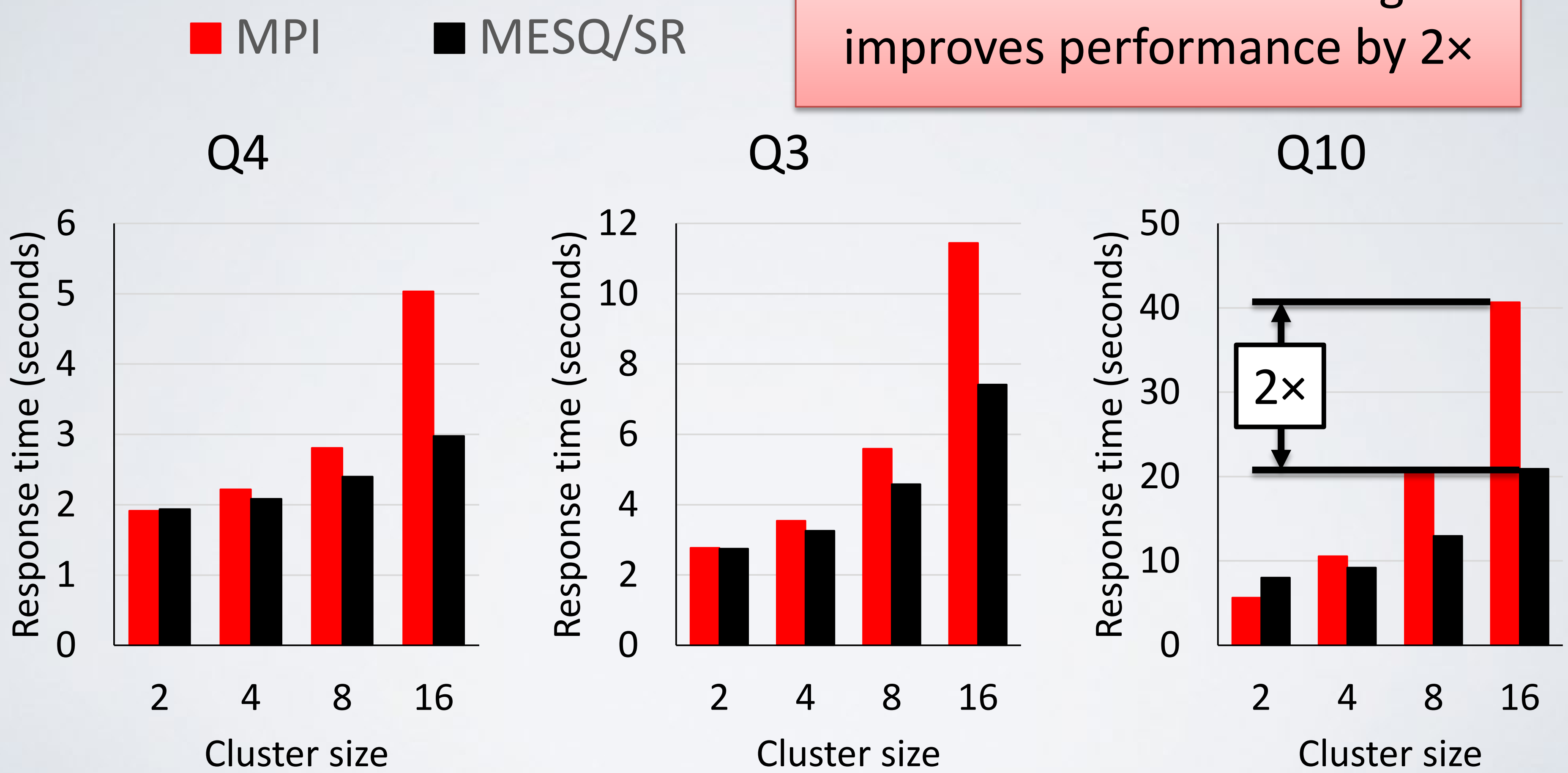
# Evaluation using TPC-H queries

- Evaluate with TPC-H Q3, Q4 and Q10
- Every node stores 100 GB of data in memory



# Evaluation using TPC-H queries

RDMA-aware shuffling improves performance by 2×



# What's wrong with MPI?

**Data-intensive computing**

- Thread-centric, heterogeneous
- Data-driven communication
- Transactional consistency on objects
- QoS: message priority, ordering
- Tolerate failures
- Elasticity, high availability

**MPI**

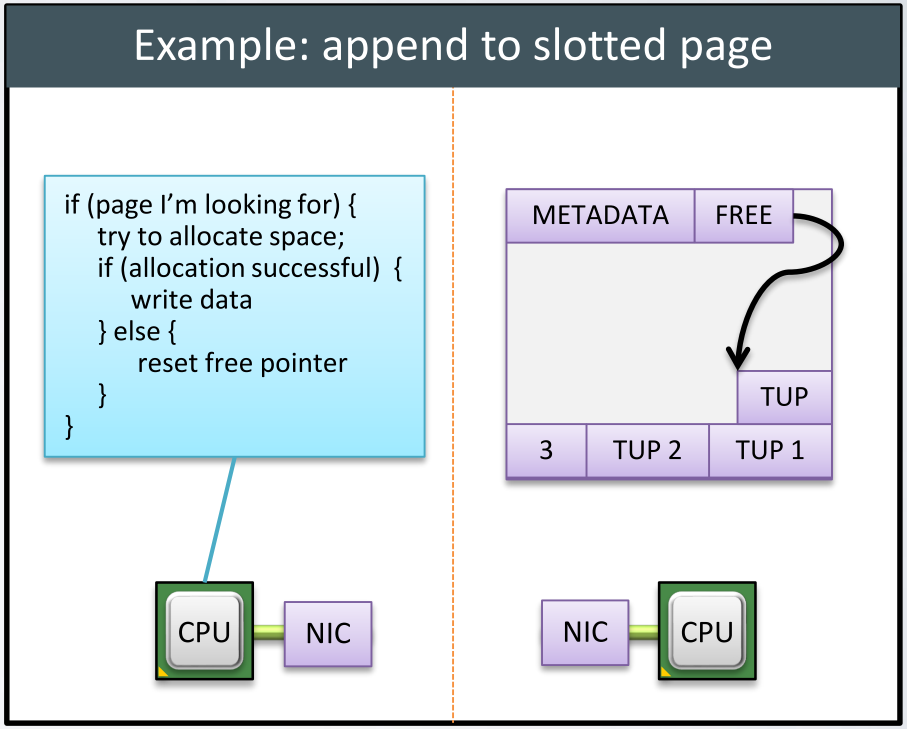
- Process-centric, homogeneous
- Collective operations within static communication group
- Consistency based on epochs and window locks
- Undefined by standard
- Error out
- Unsupported, very hard for general applications

**Need new communication abstractions for data-intensive computing**

# A proposal: RDMO

- **R**emote
  **D**irect
  **M**emory
  **O**perations

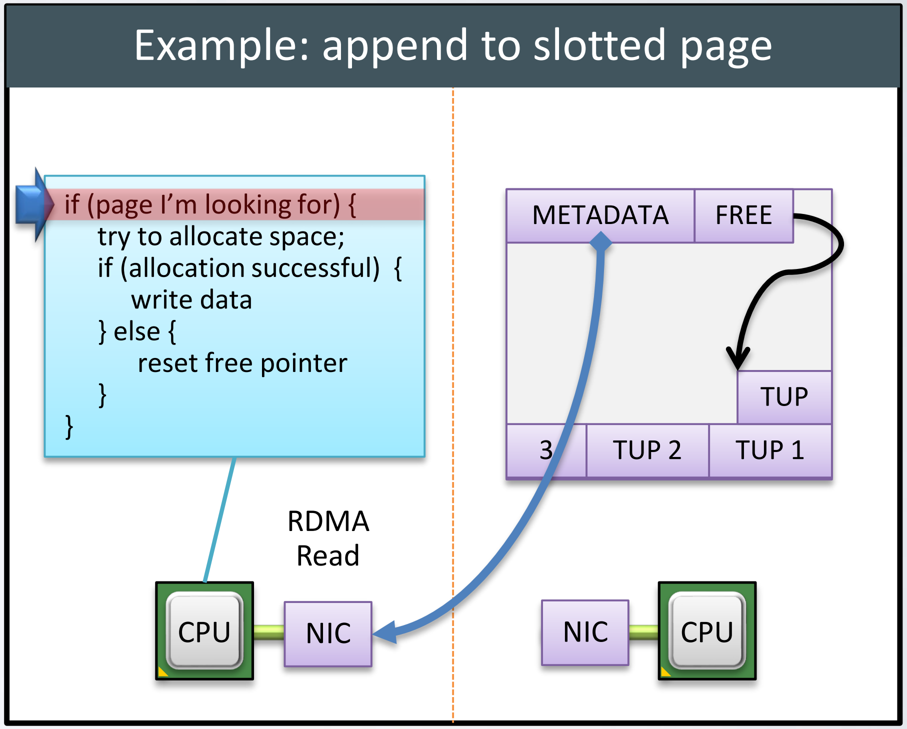
- Perform short sequences of read, write and atomics in a single round-trip



# A proposal: RDMO

- **R**emote
  **D**irect
  **M**emory
  **O**perations

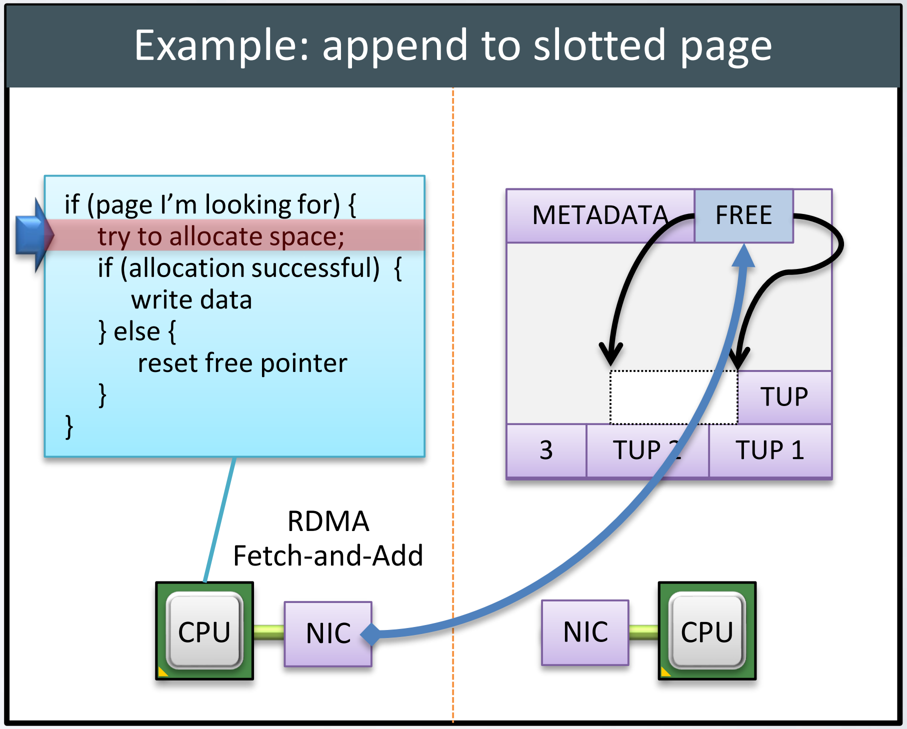
- Perform short sequences of read, write and atomics in a single round-trip



# A proposal: RDMO

- **R**emote
  **D**irect
  **M**emory
  **O**perations

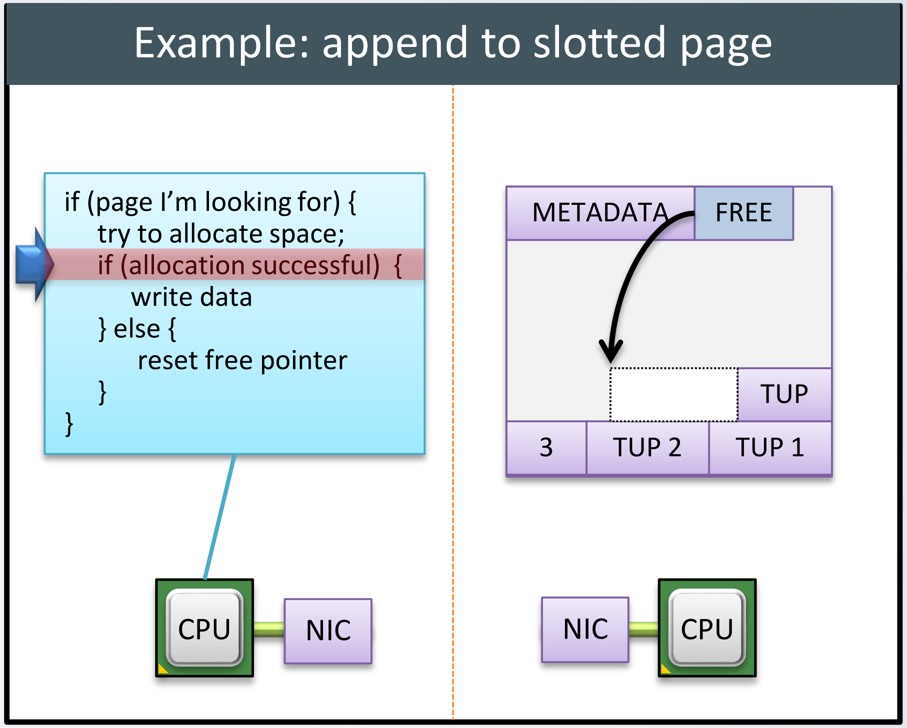
- Perform short sequences of read, write and atomics in a single round-trip



# A proposal: RDMO

- **R**emote
  **D**irect
  **M**emory
  **O**perations

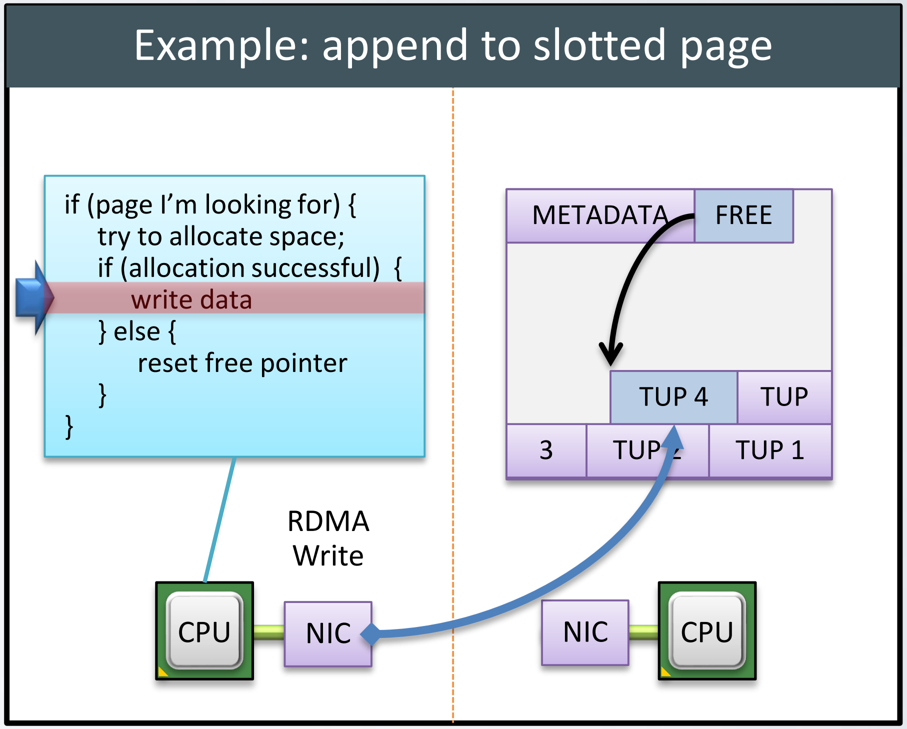
- Perform short sequences of read, write and atomics in a single round-trip



# A proposal: RDMO

- **R**emote
  **D**irect
  **M**emory
  **O**perations

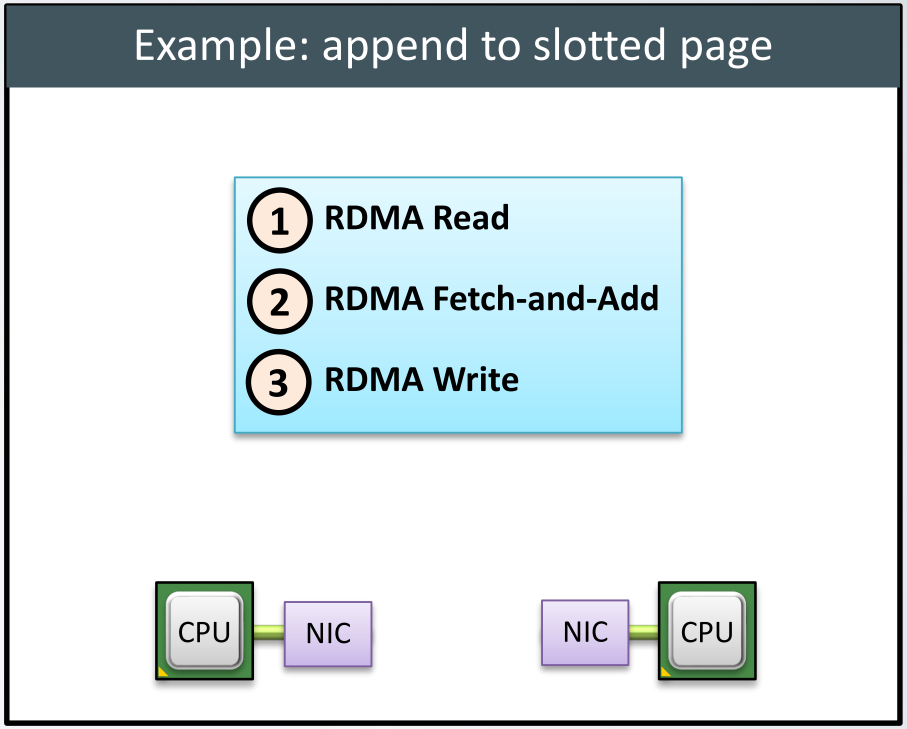
- Perform short sequences of read, write and atomics in a single round-trip



# A proposal: RDMO

- **<u>R</u>**emote
  **<u>D</u>**irect
  **<u>M</u>**emory
  **<u>O</u>**perations

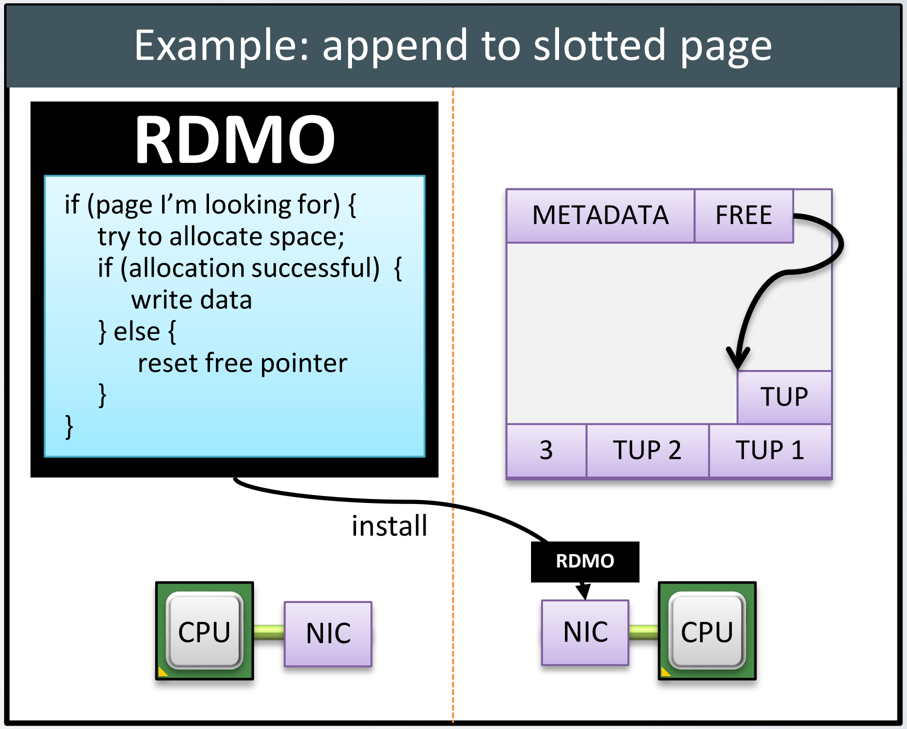
- Perform short sequences of read, write and atomics in a single round-trip



# A proposal: RDMO

- **R**emote
  **D**irect
  **M**emory
  **O**perations

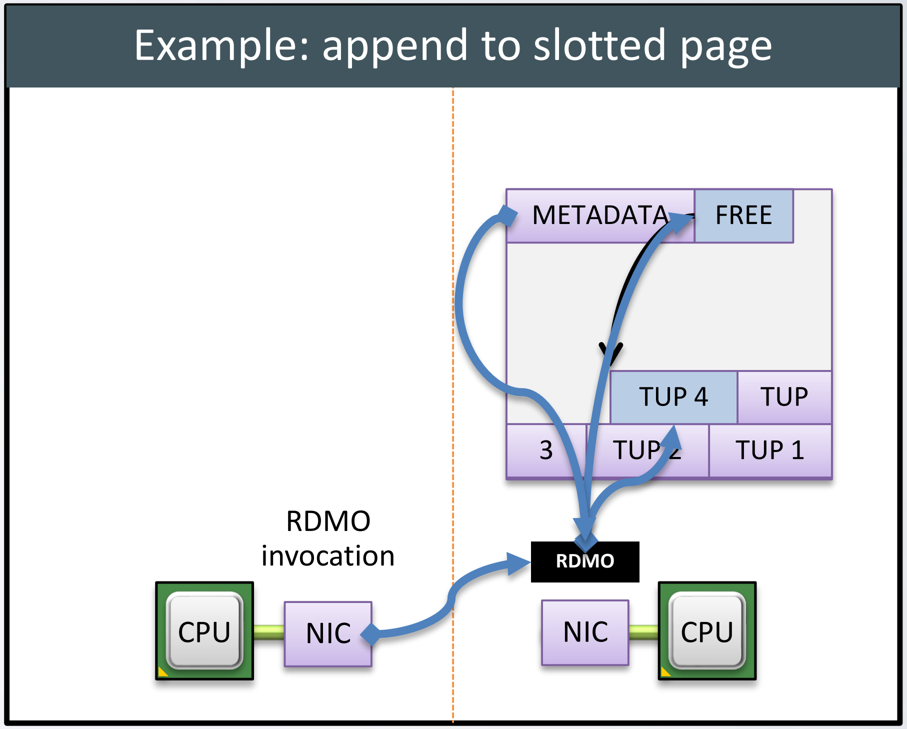
- Perform short sequences of read, write and atomics in a single round-trip



# A proposal: RDMO

- **R**emote
  **D**irect
  **M**emory
  **O**perations

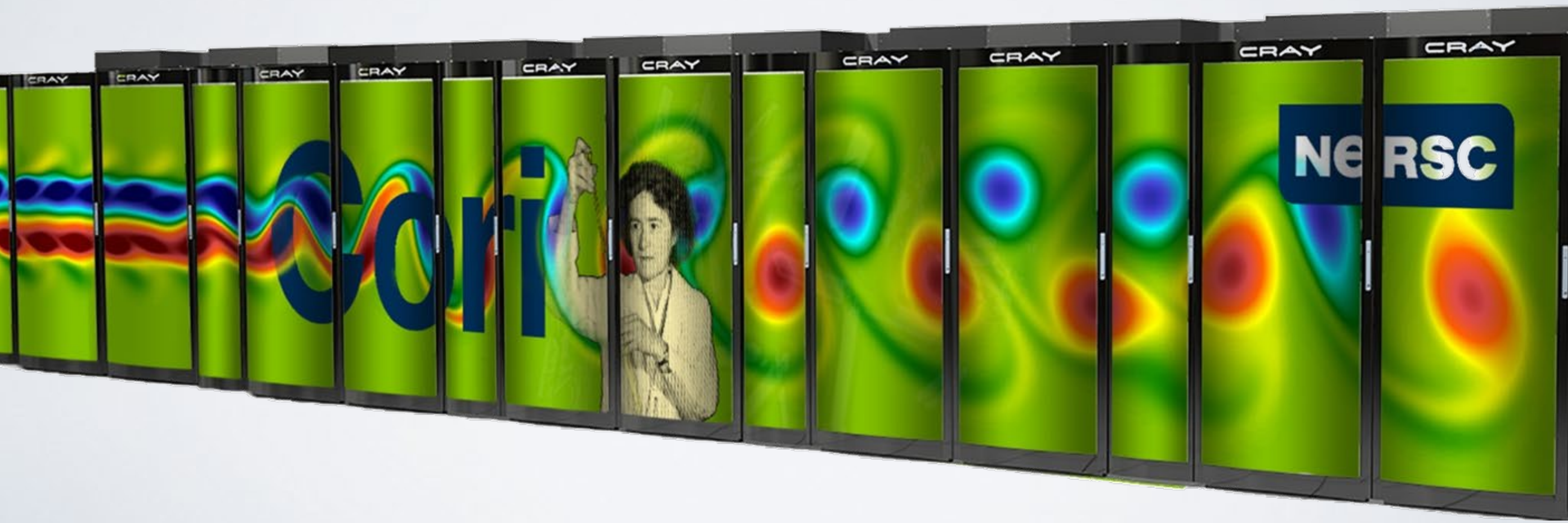
- Perform short sequences of read, write and atomics in a single round-trip



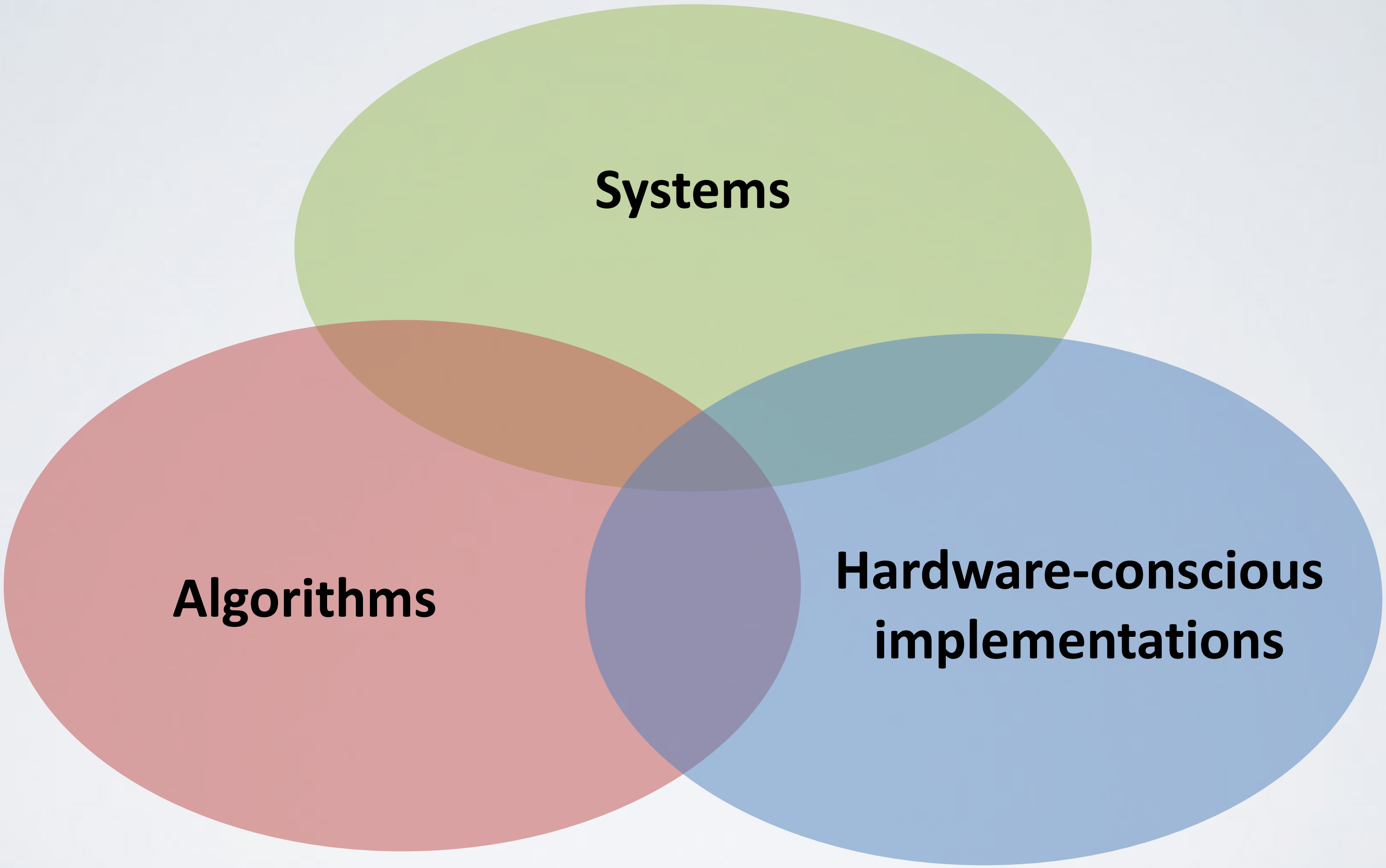
# Data processing meets massive scale

**GOAL**

Run **one query** as fast as possible
using a warehouse-scale computer



# Data processing meets massive scale



# Take aways

1. ArrayBridge: database processing over TB-sized HDF5 datasets

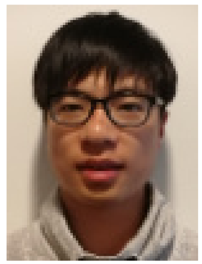
2. GRASP: a network-aware **gr**eedy **a**ggregation **s**cheduling **p**rotocol based on dataset similarity

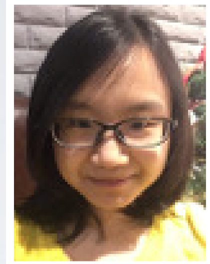
3. An RDMA-based data shuffling operator exchanges data at line rate, 4× faster than MPI

# Acknowledgments

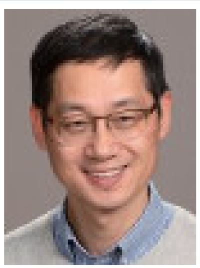
**Current and former students:**






**Collaborators, colleagues:**





**Funding, resources:**

NSF

U.S. DEPARTMENT OF ENERGY
Office of Science

Google

HUAWEI

ORACLE