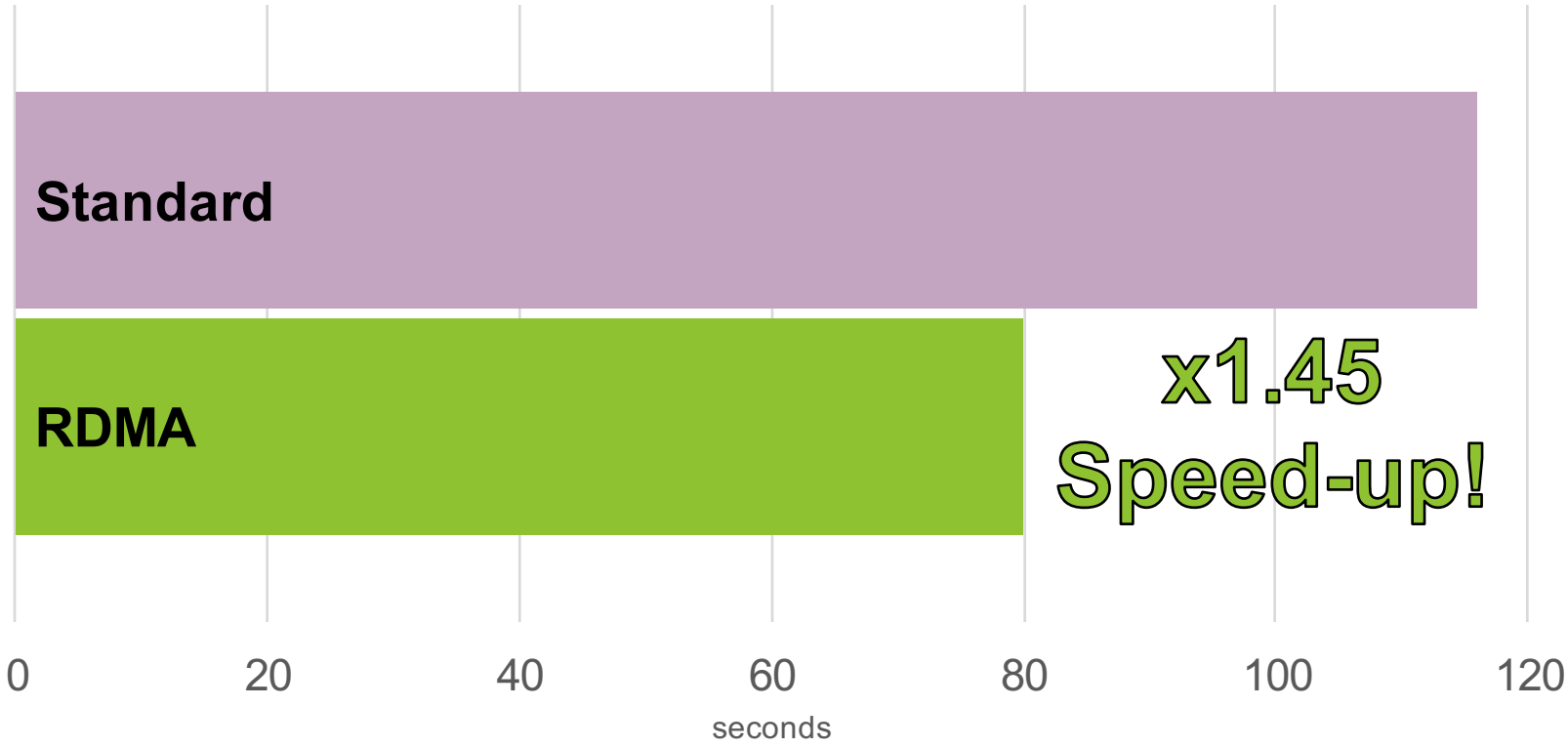
ACCELERATING SHUFFLE: A TAILOR-MADE RDMA SOLUTION FOR APACHE SPARK

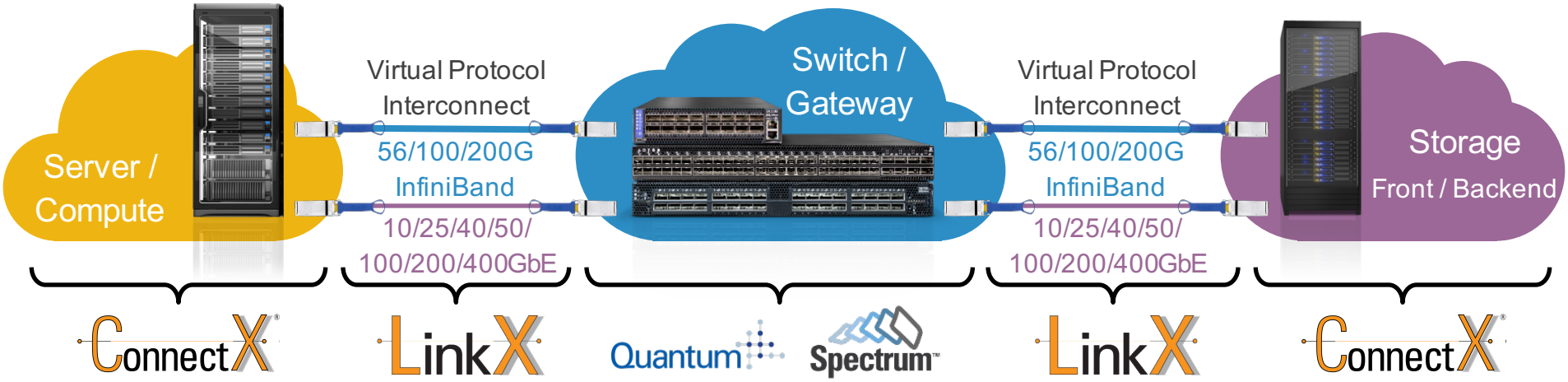
Yuval Degani, Mellanox Technologies

#EUres3

TeraSort accelerated with RDMA



- Founded 1999
- End-to-end designer and supplier of interconnect solutions: network adapters, switches, system-on-a-chip, cables, silicon and software
- 10-400 Gb/s Ethernet and InfiniBand

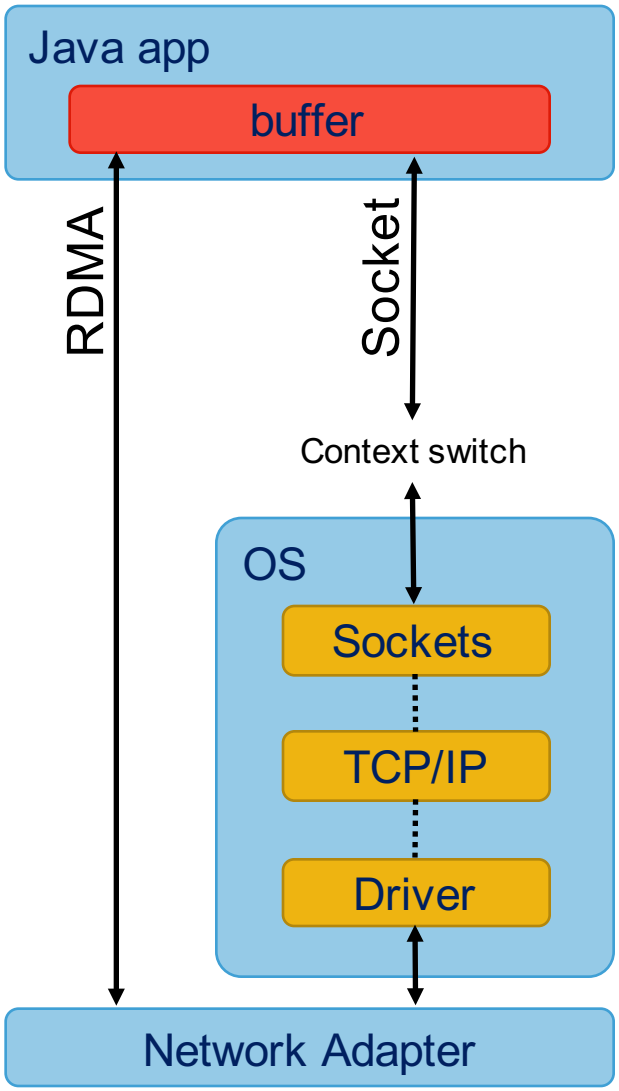


Agenda

- What's RDMA?
- Spark's Shuffle Internals
- SparkRDMA Shuffle Plugin
- Results
- What's next?

What's RDMA?

- Remote Direct Memory Access
  - Read/write from/to remote memory locations
- Zero-copy
- Direct hardware interface – bypasses the kernel and TCP/IP in IO path
- Flow control and reliability is offloaded in hardware
- Sub-microsecond latency
- Supported on almost all mid-range/high-end network adapters
- Growing cloud support
  - Already supported in Microsoft Azure (A, H instances)



Hardware acceleration in Big Data/Machine Learning platforms

- Hardware acceleration adoption is continuously growing
  - GPU integration is now standard
  - ASIC integration is spreading fast

- RDMA is already integrated in mainstream code of popular frameworks:
  - TensorFlow
  - Caffe2
  - CNTK

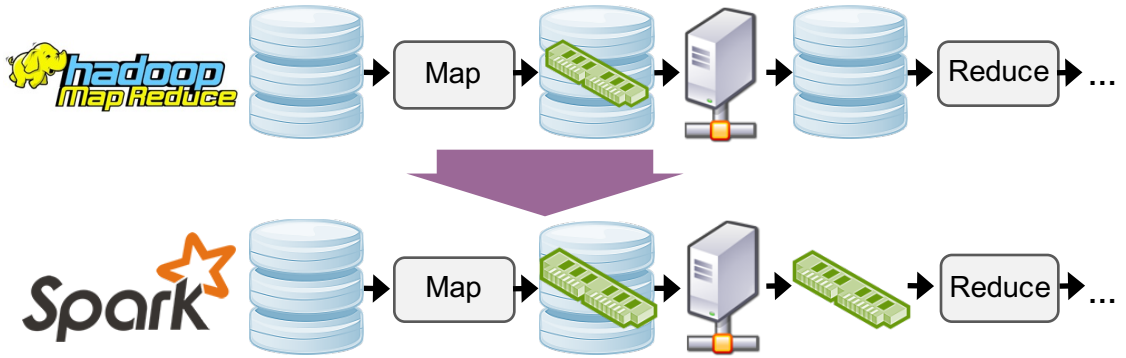
- Now it's Spark's turn to catch up

Under the hood

Spark's Shuffle Internals

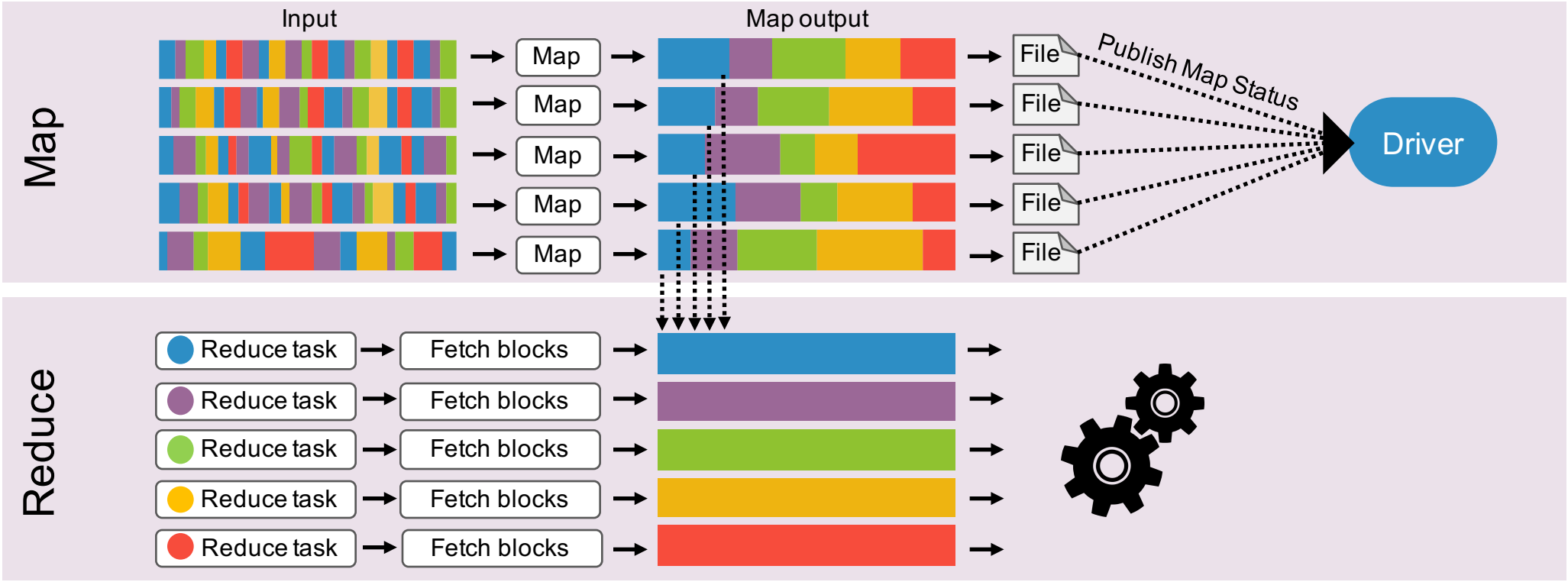
MapReduce vs. Spark

- Spark's in-memory model completely changed how shuffle is done
- In both Spark and MapReduce, map output is saved on the local disk (usually in buffer cache)
- In MapReduce, map output is then copied over the network to the destined reducer's local disk
- In Spark, map output is fetched from the network, on-demand, to the reducer's memory

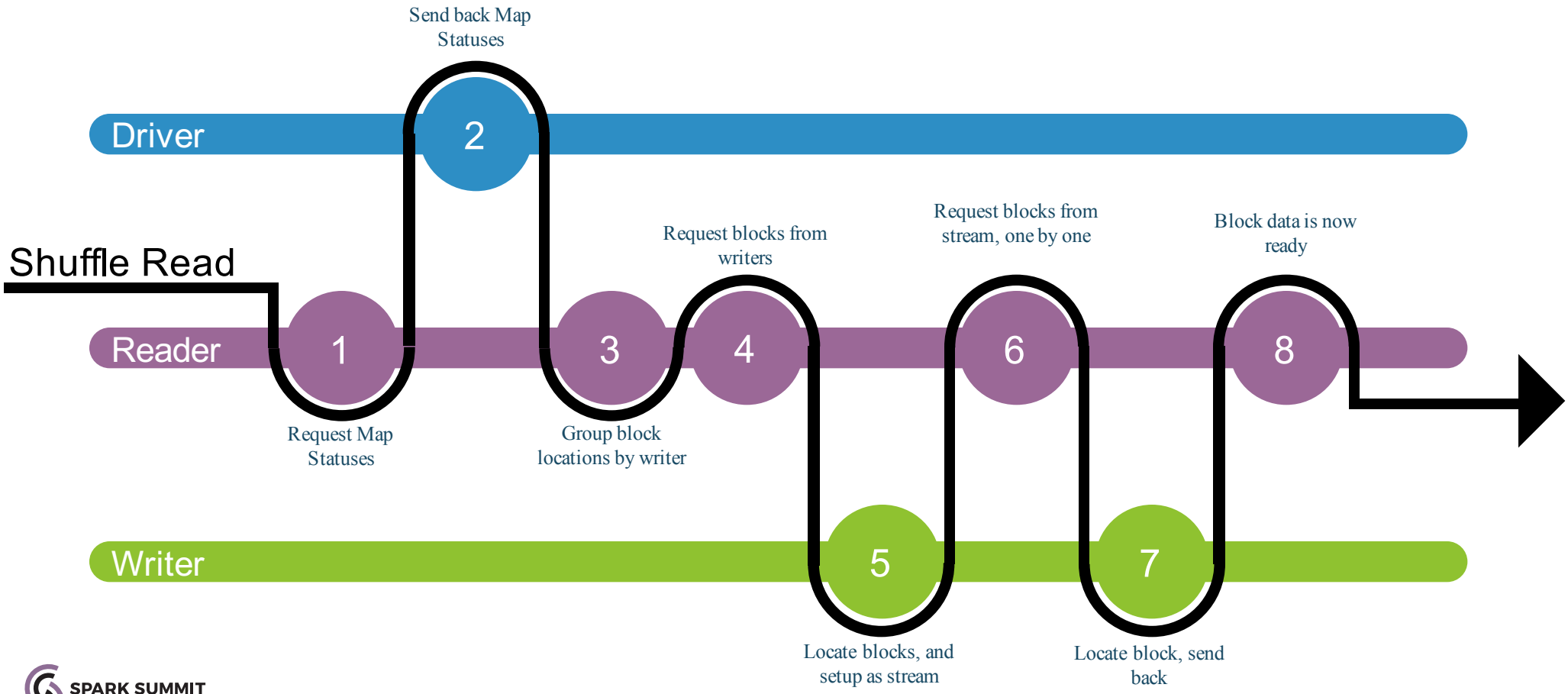


Memory-to-network-to-memory? RDMA is a perfect fit!

Spark's Shuffle Basics



Shuffle Read Protocol



The Cost of Shuffling

- Shuffling is very expensive in terms of CPU, RAM, disk and network IOs
- Spark users try to avoid shuffles as much as they can
- Speedy shuffles can relieve developers of such concerns, and simplify applications

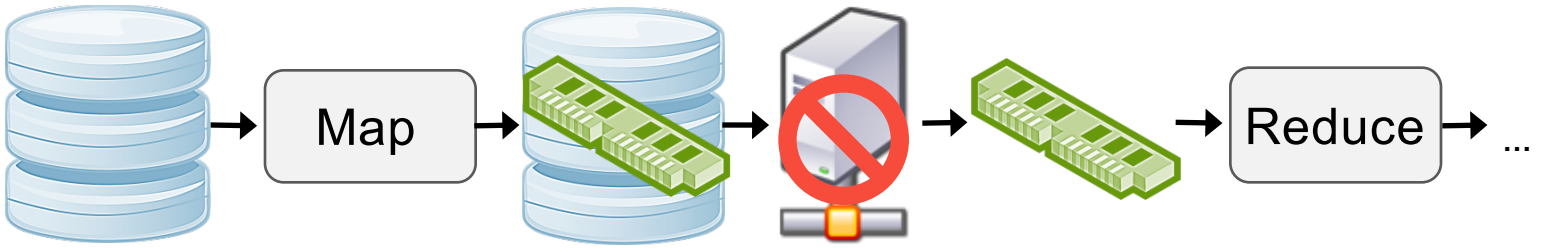
What's the opportunity here?

The Potential in Accelerating Shuffle

The Potential in Accelerating Shuffle

- Before setting off on the journey of adding RDMA to Spark, it was essential to estimate the ROI of such work
- What's better than a controlled experiment?

Goal	Quantify the potential performance gain in accelerating network transfers
Method	Bypass the network in Spark shuffle and compare to the original code: No network = maximum performance gain



Experiment: Bypassing The Network in Shuffle

- Do no fetch blocks from the network, instead, reuse a local sampled block over and over
- Compare to standard Spark, but with block reuse – so reduce data will be identical

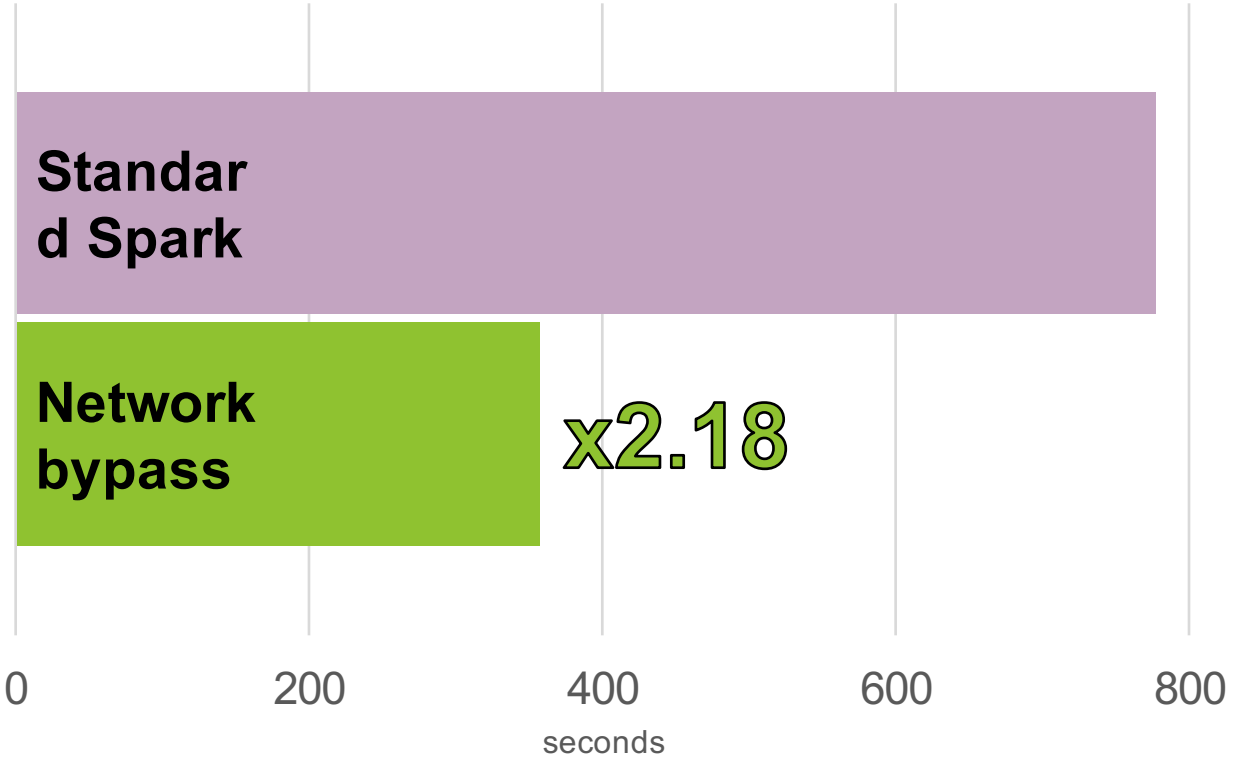
Results

Benchmark:

- HiBench TeraSort
- Workload: 600GB

Testbed:

- HDFS on Hadoop 2.6.0
- Spark 2.0.0
  - Standalone mode
  - Master + 30 workers
  - 28 cores per worker, 840 total
- Machine info:
  - Intel Xeon E5-2697 v3 @ 2.60GHz
  - 256GB RAM, 128GB of it reserved as RAMDISK
  - RAMDISK is used for Spark local directories and HDFS



Accelerating Shuffle with RDMA

SparkRDMA Shuffle Plugin

Design Goals

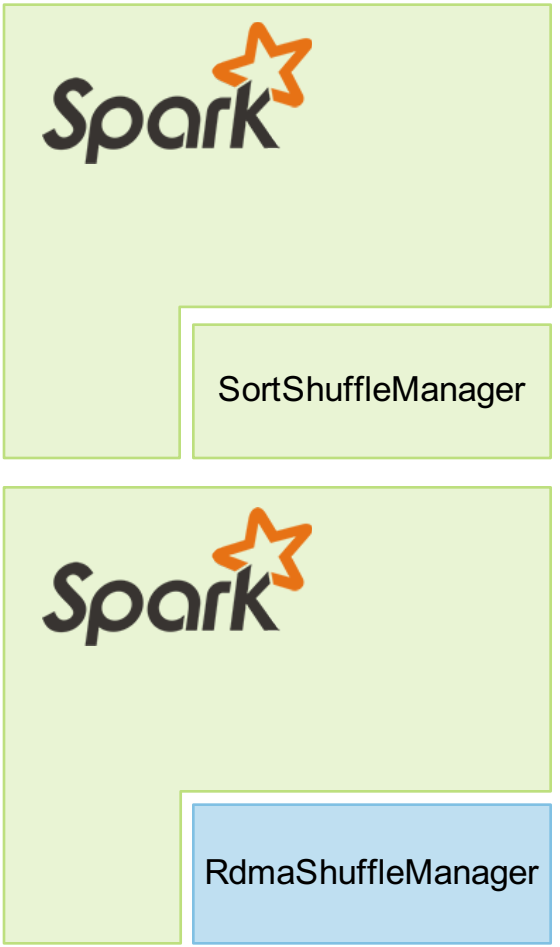
- Demonstrate significant improvements over standard Spark
- Seamlessly accelerate Shuffles with RDMA – no functional limitations
- Easy to use and deploy
- Minimize code impact

Design Approach

- Entire Shuffle-related communication is done with RDMA
  - RPC messaging for meta-data transfers
  - Block transfers
- SparkRDMA is an independent plugin
  - Implements the ShuffleManager interface
  - No changes to Spark's code – use with any existing Spark installation
- Reuse Spark facilities
  - Maximize reliability
  - Minimize impact on code
- RDMA functionality is provided by "DiSNI"
  - Open-source Java interface to RDMA user libraries
  - https://github.com/zrlio/disni
- No functionality loss of any kind, SparkRDMA supports:
  - Compression
  - Spilling to disk
  - Recovery from failed map or reduce tasks

ShuffleManager Plugin

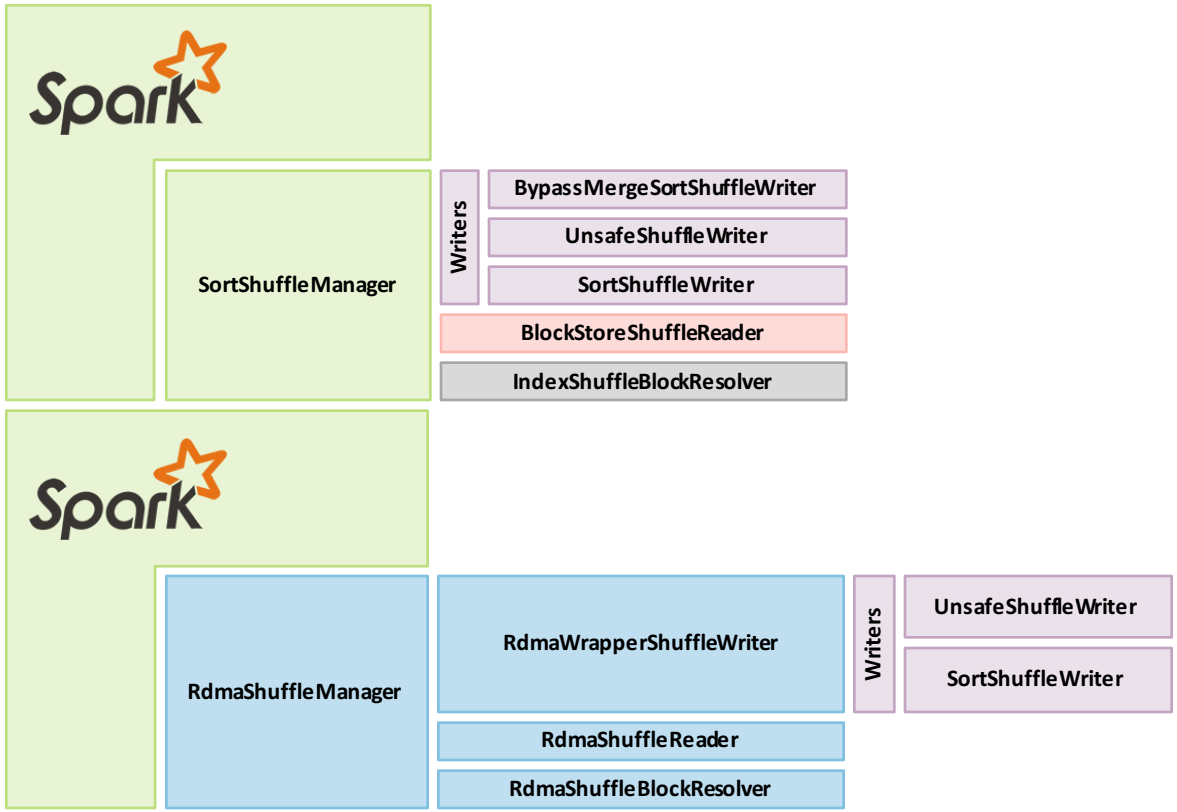
- Spark allows for external implementations of ShuffleManagers to be plugged in
  - Configurable per-job using: “spark.shuffle.manager”
- Interface allows proprietary implementations of Shuffle Writers and Readers, and essentially defers the entire Shuffle process to the new component
- SparkRDMA utilizes this interface to introduce RDMA in the Shuffle process



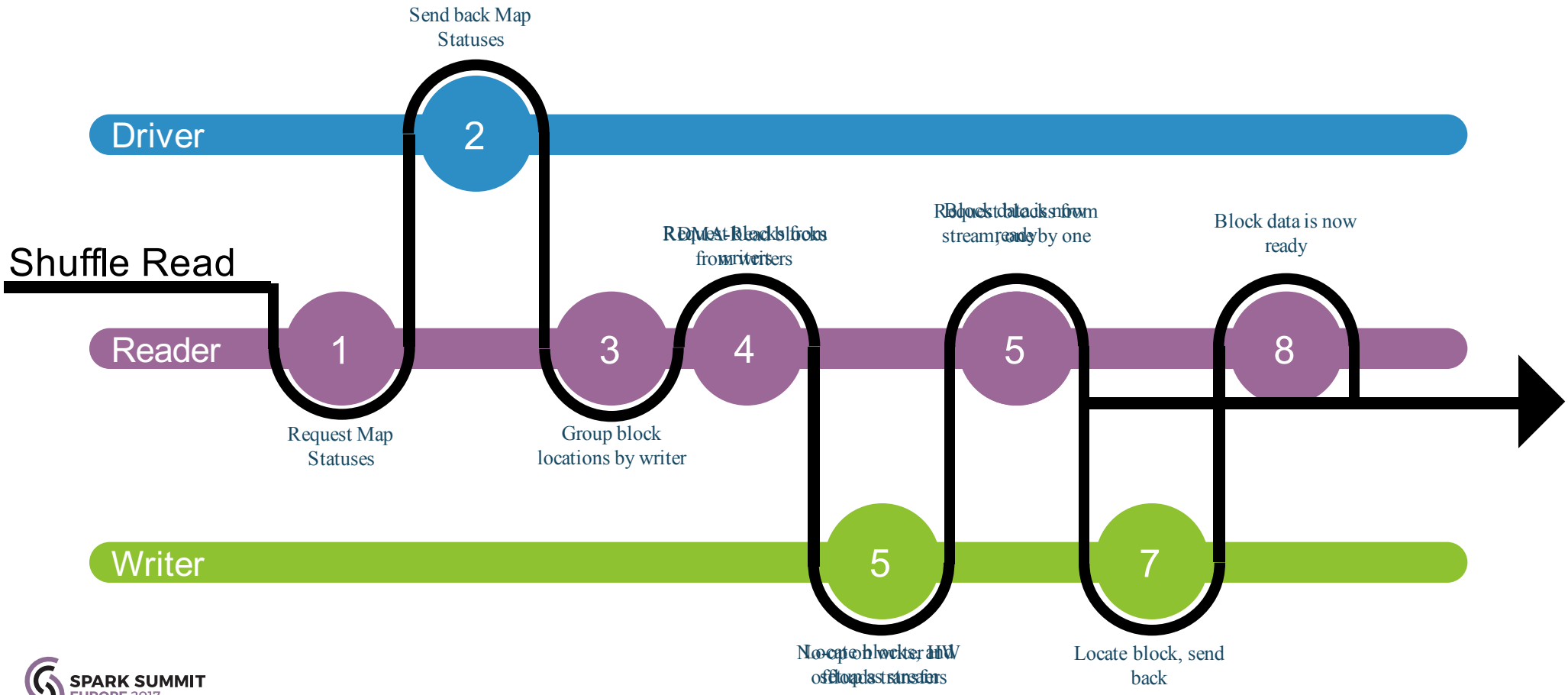
SparkRDMA Components

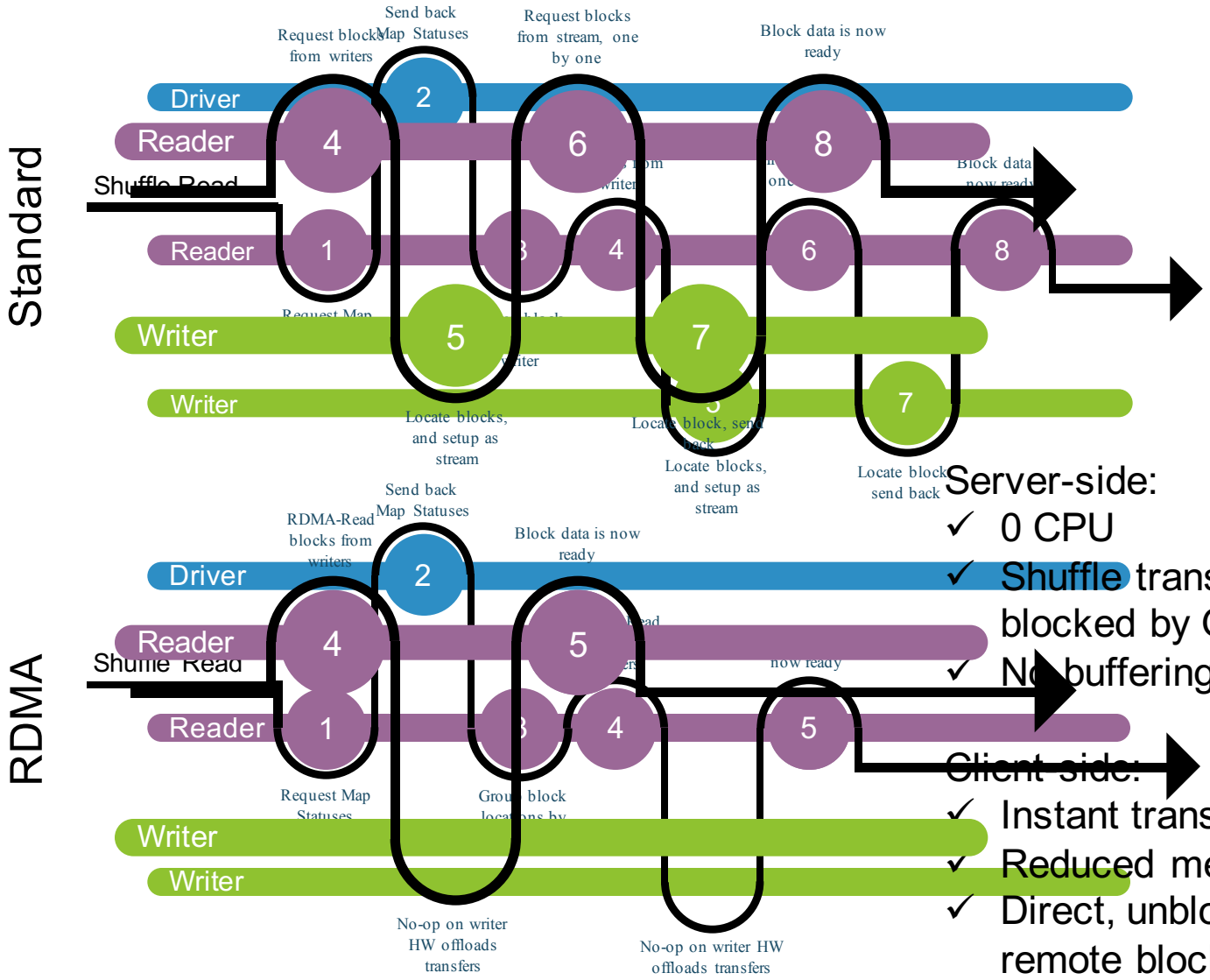
- SparkRDMA reuses the main Shuffle Writer implementations of mainstream Spark: Unsafe & Sort
- Shuffle data is written and stored identically to the original implementation

- All-new ShuffleReader and ShuffleBlockResolver provide an optimized RDMA transport when blocks are being read over the network



Shuffle Read Protocol – Standard vs. RDMA




Standard
Request blocks from writers
Send back Map Statuses
Request blocks from stream, one by one
Block data is now ready
Driver
Reader
Shuffle Read
Reader
Writer
Writer
Locate blocks, and setup as stream
Locate blocks, and setup as stream
Locate block send back
RDMA
RDMA-Read blocks from writers
Send back Map Statuses
Block data is now ready
Driver
Reader
Shuffle Read
Reader
Writer
Writer
Request Map Statuses
No-op on writer HW offloads transfers
No-op on writer HW offloads transfers
Server-side:
✓ 0 CPU
✓ Shuffle transfers are not blocked by GC in executor
✓ No buffering
Client-side:
✓ Instant transfers
✓ Reduced messaging
✓ Direct, unblocked access to remote blocks

Benefits

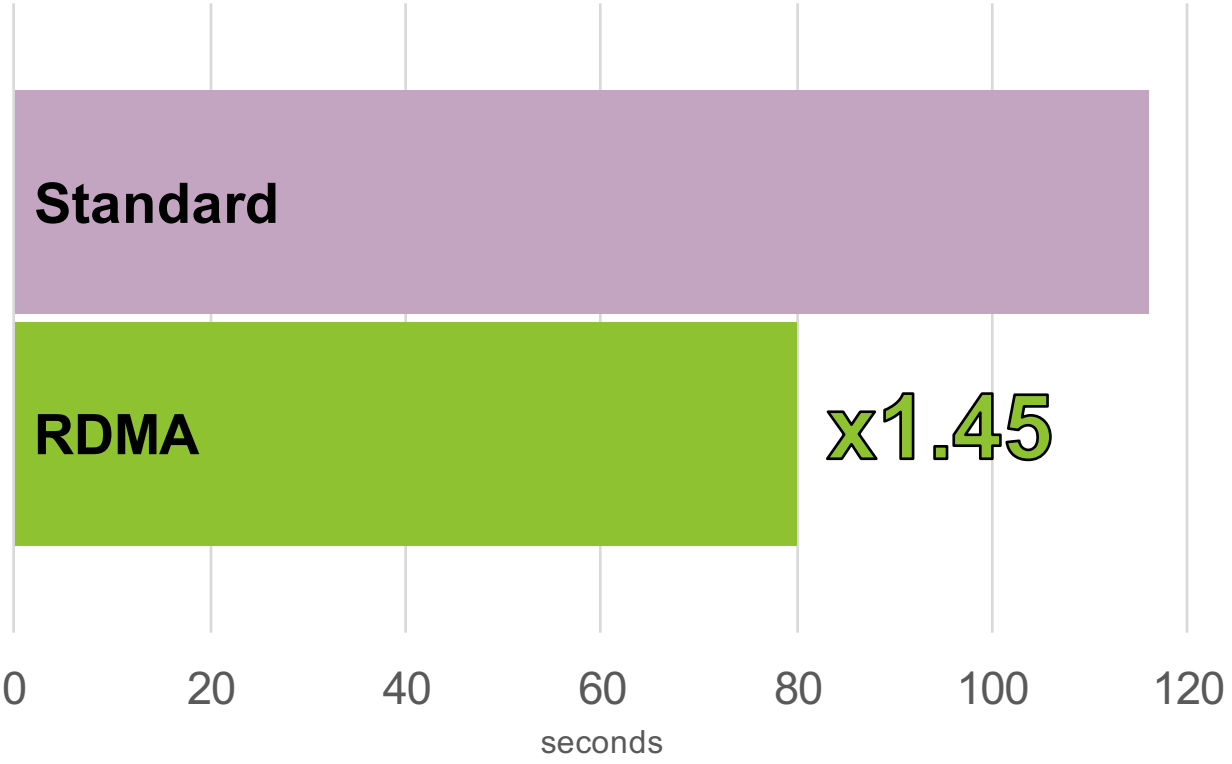
- Substantial improvements in:
  - Block transfer times: latency and total transfer time
  - Memory consumption and management
  - CPU utilization
- Easy to deploy and configure:
  - Packed into a single JAR file
  - Plugin is enabled through a simple configuration handle
  - Allows finer tuning with a set of configuration handles
- Configuration and deployment are on a per-job basis:
  - Can be deployed incrementally
  - May be limited to Shuffle-intensive jobs

Results

Performance Results: TeraSort

Testbed:

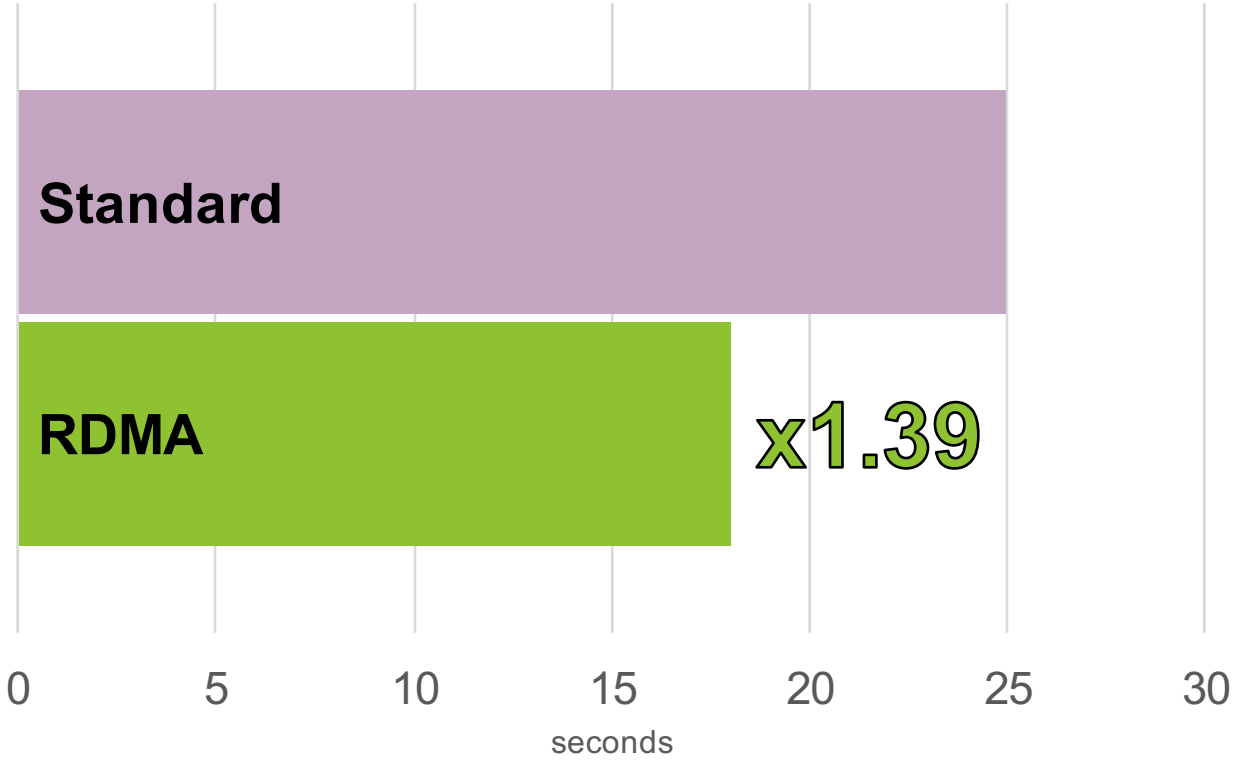
- HiBench TeraSort
  - Workload: 175GB
- HDFS on Hadoop 2.6.0
  - No replication
- Spark 2.0.0
  - 1 Master
  - 15 Workers
  - 28 active Spark cores on each node, 420 total
- Node info:
  - Intel Xeon E5-2697 v3 @ 2.60GHz
  - RoCE 100GbE
  - 256GB RAM
  - HDD is used for Spark local directories and HDFS



Performance Results: GroupBy

Testbed:

- GroupBy
  - 48M keys
  - Each value: 4096 bytes
  - Workload: 183GB
- Spark 2.0.0
  - 1 Master
  - 15 Workers
  - 28 active Spark cores on each node, 420 total
- Node info:
  - Intel Xeon E5-2697 v3 @ 2.60GHz
  - RoCE 100GbE
  - 256GB RAM
  - HDD is used for Spark local directories and HDFS



What's next

Roadmap

What's next?

- SparkRDMA official v1.0 release is publicly available on GitHub
- Integration to upstream Apache Spark

- Extend RDMA capabilities to more Spark components:
  - RDD accesses
  - Broadcast

- Also in the works: RDMA acceleration for HDFS

Open-source

- SparkRDMA is available at https://github.com/Mellanox/SparkRDMA
  - Quick installation guide
  - Wiki pages for advanced settings

- Vote for integrating SparkRDMA into mainstream Spark: https://issues.apache.org/jira/browse/SPARK-22229

- Feel free to reach out at: yuvaldeg@mellanox.com

Thank you.

#EUres3