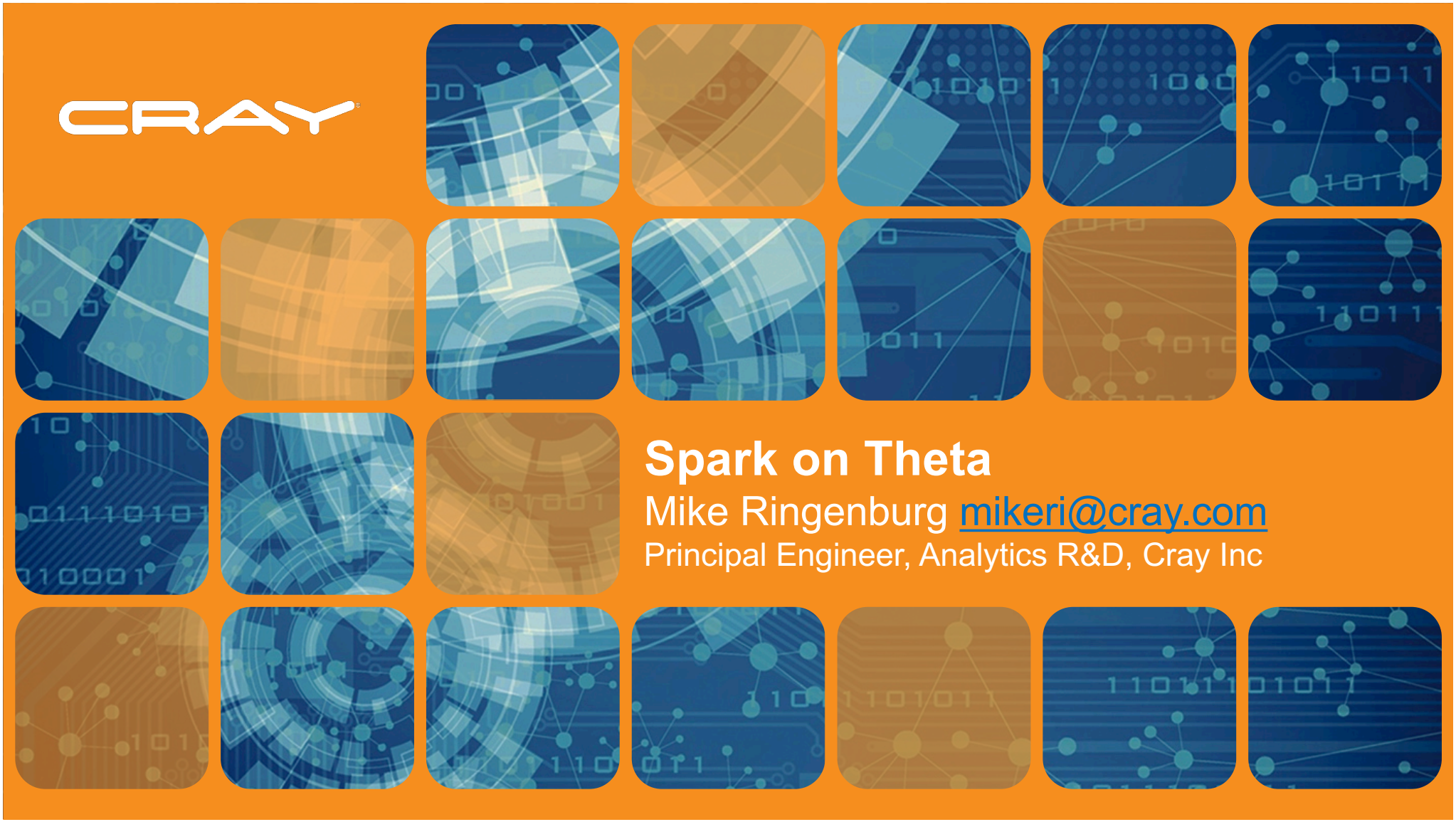
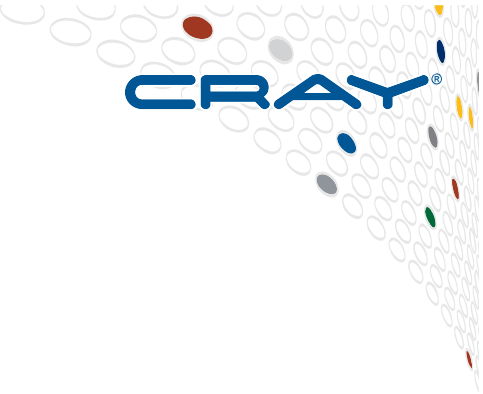


Spark on Theta

Mike Ringenburg mikeri@cray.com
Principal Engineer, Analytics R&D, Cray Inc

Agenda



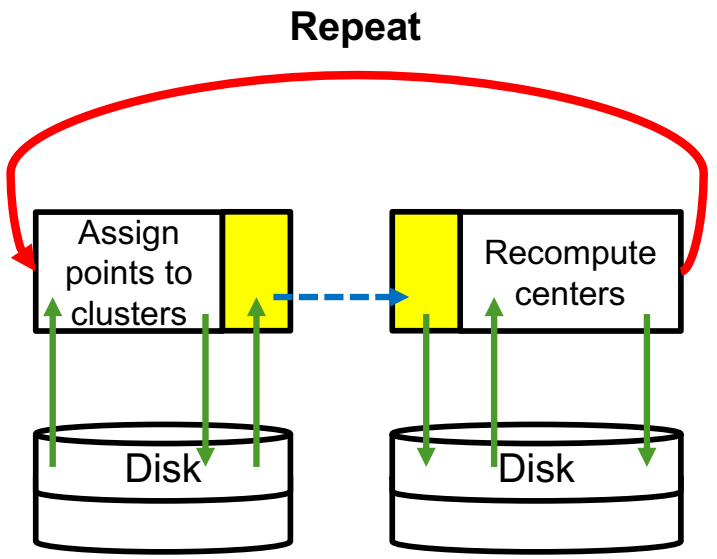
- **Introduction to Spark**
  - History and Background
  - Computation and Communication Model
- **Spark on the XC40**
  - Installation and Configuration
  - Local storage
- **Spark on Theta/KNL**
  - Tips
- **Questions?**

In the beginning, there was Hadoop MapReduce…

- **Simplified parallel programming model**
  - All computations broken into two parts
    - **Embarassingly parallel map phase**: apply single operation to every key,value-pair, produce new set of key,value-pairs
    - **Combining reduce phase**: Group all values with identical key, performing combining operation to get final value for key
  - Can perform multiple iterations for computations that require
  - I/O intensive
    - Map writes to local storage. Data shuffled to reducer's local storage, reduce reads.
    - Additional I/O between iterations in multi-iteration algorithms (map reads from HDFS, reduce writes to HDFS)
  - Effective model for many data analytics tasks
- **HDFS distributed file system (locality aware – move compute to data)**
- **YARN cluster resource manager**

Example: K-Means Clustering with MapReduce

- **Initially: Write out random cluster centers**
- **Map:**
  - Read in cluster centers
  - For each data point, compute nearest cluster center and write <key: nearest cluster, value: data point>
- **Reduce:**
  - For each cluster center (key) compute average of datapoints
  - Write out this value as new cluster center
- **Repeat until convergence (clusters don't change)**



MapReduce Problems

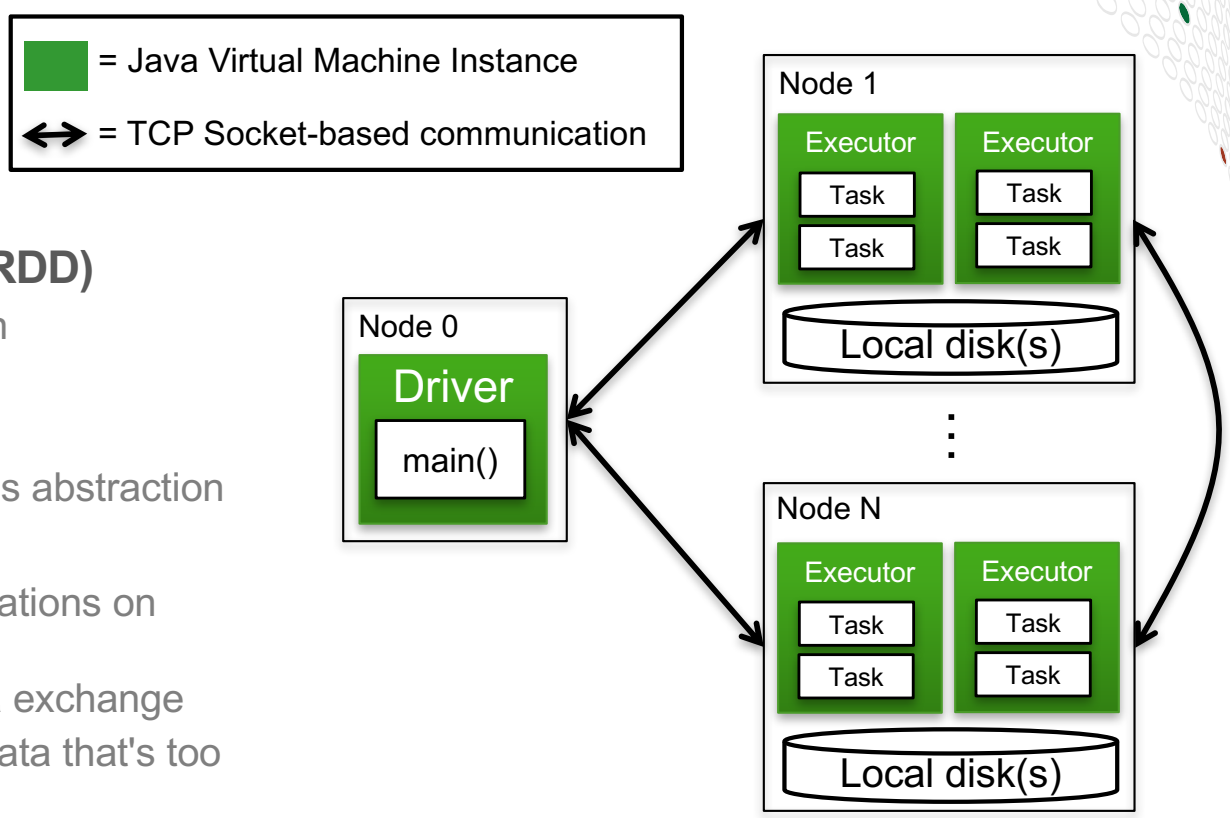
- **Gated on IO bandwidth, possibly interconnect as well**
  - Must write and read between map and reduce phases
  - Multiple iterations must write results in next time (e.g., new cluster centers)
- **No ability to persist reused data**
- **Must re-factor all computations as map then reduce (and repeat?)**

What is Spark?

- **Newer (2014) analytics framework**
  - Originally from Berkeley AMPLab/BDAS stack, now Apache project
  - Native APIs in Scala. Java, Python, and R APIs available as well.
  - Many view as successor to Hadoop MapReduce. Compatible with much of Hadoop Ecosystem.
- **Aims to address some shortcomings of Hadoop MapReduce**
  - More programming flexibility – not constrained to one map, one reduce, write, repeat.
  - Many operations can be pipelined into a single in-memory task
  - Can "persist" intermediate data rather than regenerating every stage

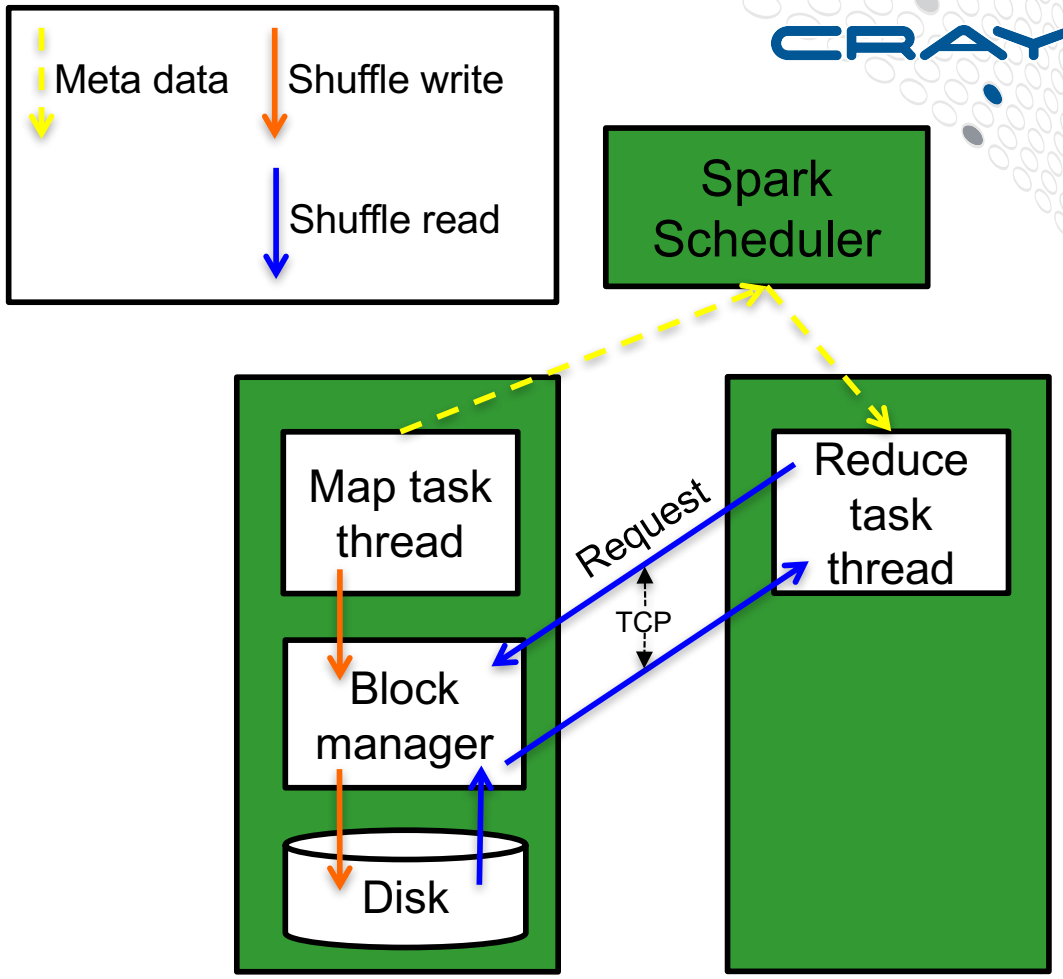
Spark Execution Model

- **Master-slave parallelism**
- **Driver (master)**
  - Executes main
  - Distributes work to executors
- **Resilient Distributed Dataset (RDD)**
  - Spark's original data abstraction
  - Partitioned amongst executors
  - Fault-tolerant via lineage
  - Dataframes/Datasets extend this abstraction
- **Executors (slaves)**
  - Lazily execute tasks (local operations on partitions of the RDD)
  - Global all-to-all shuffles for data exchange
  - Rely on local disks for spilling data that's too large, and storing shuffle data



Spark Communication Model (Shuffles)

- **All data exchanges between executors implemented via *shuffle***
  - Senders ("mappers") send data to block managers; block managers write to disks, tell scheduler *how much* destined for each reducer
  - Barrier until all mappers complete shuffle writes
  - Receivers ("reducers") request data from block managers *that have data for them*; block managers read and send



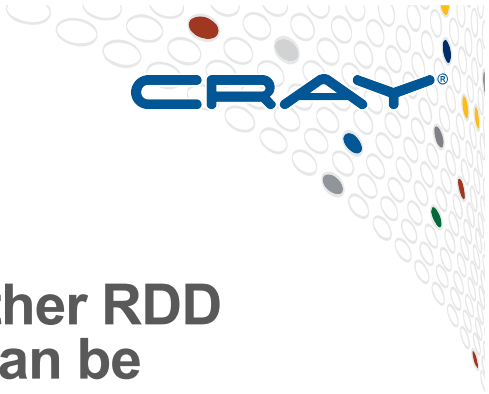
RDDs (and DataFrames/DataSets)

- **RDDs are original data abstraction of Spark**
  - DataFrames add structure to RDDs: named columns
  - DataSets add strong typing to columns of DataFrames (Scala and Java only)
  - Both build on the basic idea of RDDs
    - DataFrames were originally called SchemaRDDs
- **RDD data structure contains a description of the data, partitioning, and computation, but not the actual data … why?**
  - Lazy evaluation

Lazy Evaluation and DAGs

- **Spark is lazily evaluated**
  - Spark operations are only executed when and if needed
  - Needed operations: produce a result for driver, or produce a parent of needed operation (recursive)
- **Spark DAG (Directed Acyclic Graph)**
  - Calls to transformation APIs (operations that produce a new RDD/DataFrame from one or more parents) just add a new node to the DAG, indicating data dependencies (parents) and transformation operation
  - Action APIs (operations that return data) trigger execution of necessary DAG elements
- **Example shortly…**

Tasks, Stages, and Pipelining



- **If an RDD partition's dependencies are on a single other RDD partition (or on *co-partitioned* data), the operations can be *pipelined* into a single *task***
  - **Co-partitioned**: all of the parent RDD partitions are co-located with child RDD partitions that need them
  - **Pipelined**: Operations can occur as soon as the local parent data is ready (no synchronization)
  - **Task**: A pipelined set of operations
  - **Stage**: Execution of same task on all partitions
- **Every stage ends with a shuffle, an output, or returning data back to the driver.**
  - Global barrier between stages. All senders complete shuffle write before receivers request data (shuffle read)

Spark Example: Word Count

Load file

flatMap maps one value to (possibly) many, instead of one-to-one like map

groupByKey combines all key-value pairs with the same key (k, v1), …, (k,vn) into a single key-value pair (k, (v1, …, vn)).

Collect returns all elements to the driver

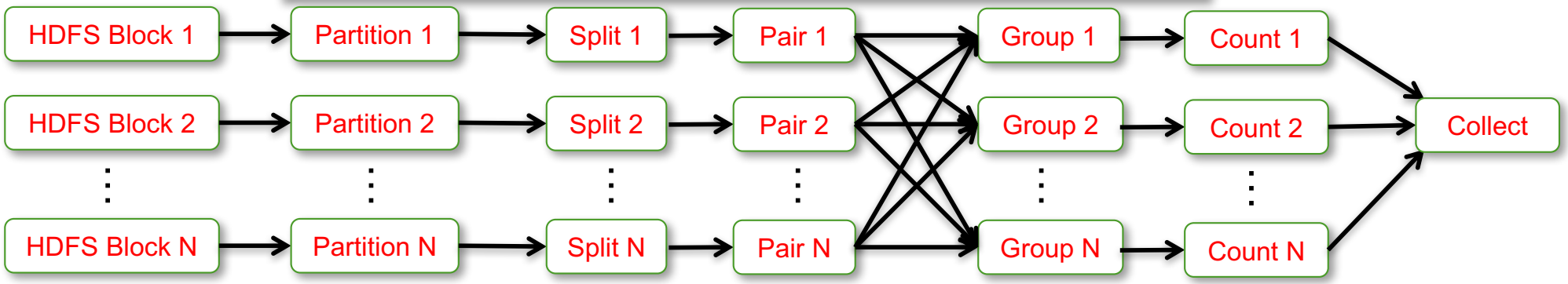
```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

- Let's like at a simple example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word

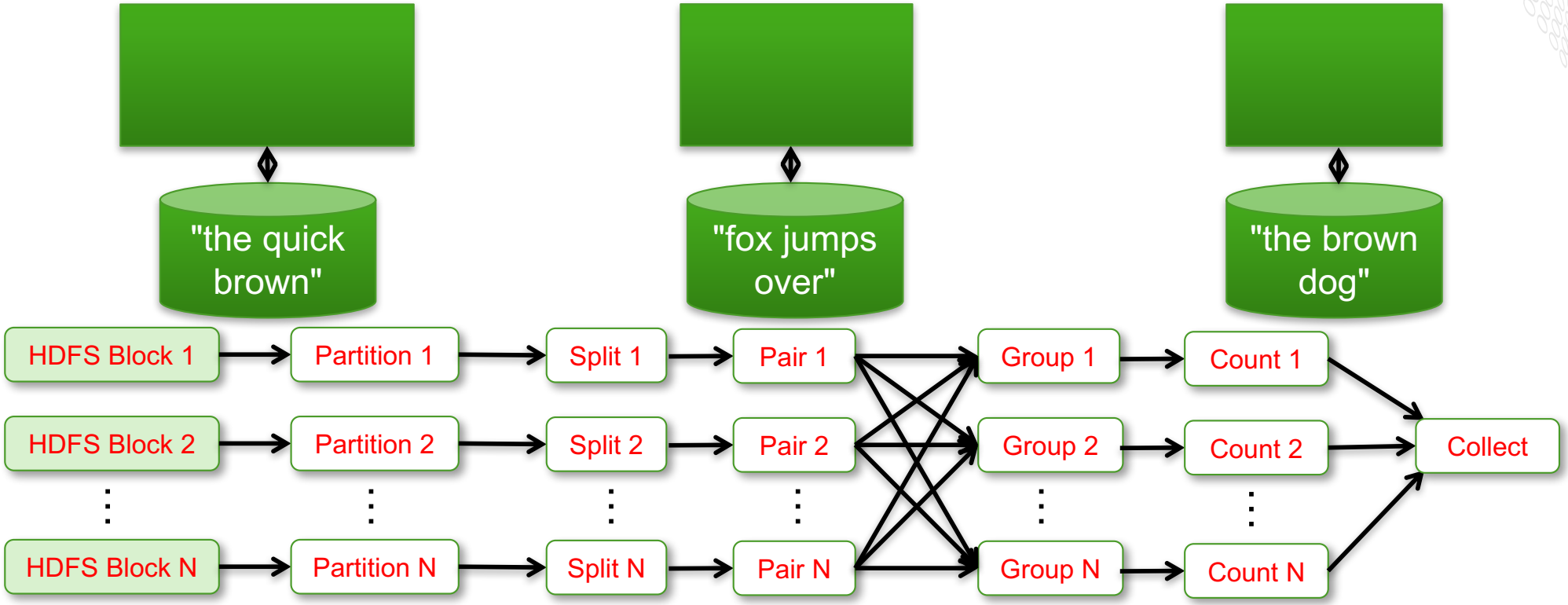
The Spark DAG

```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

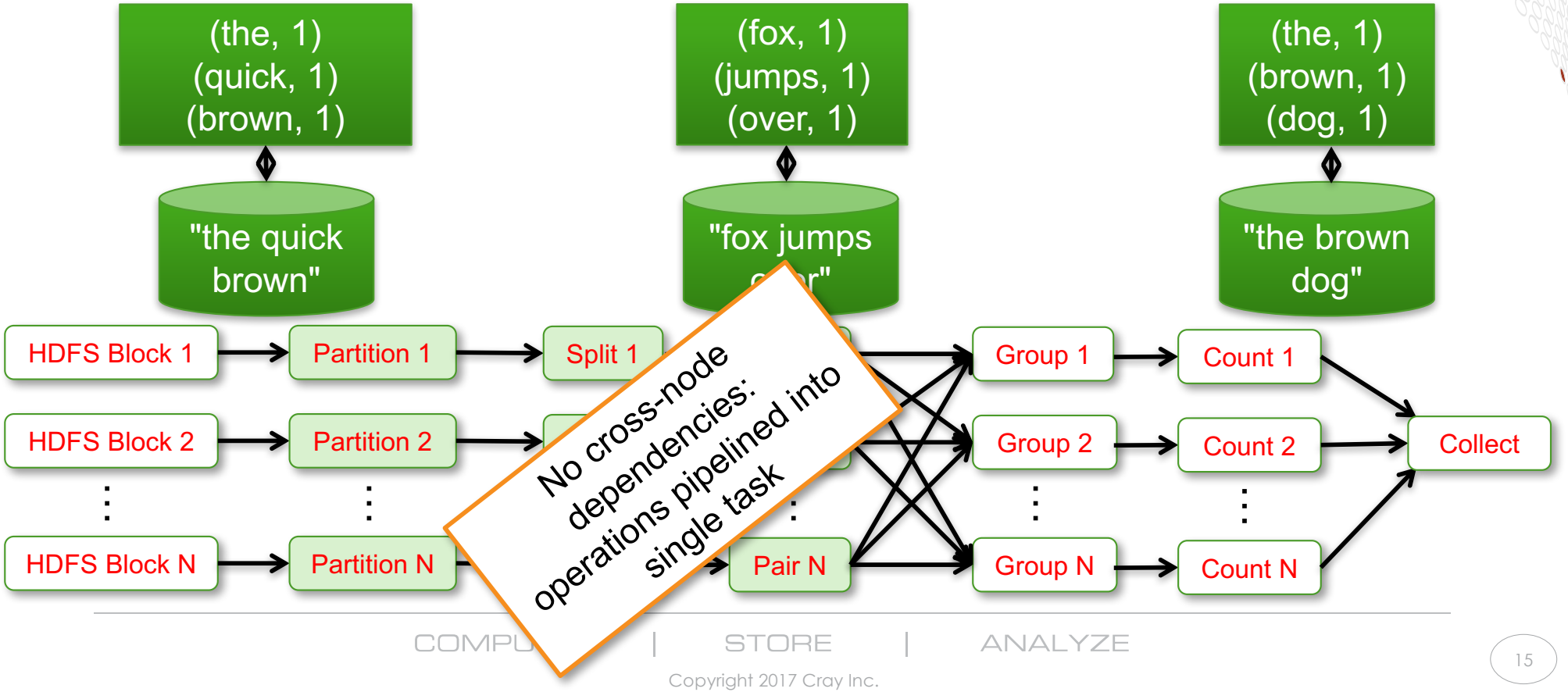
Execute!



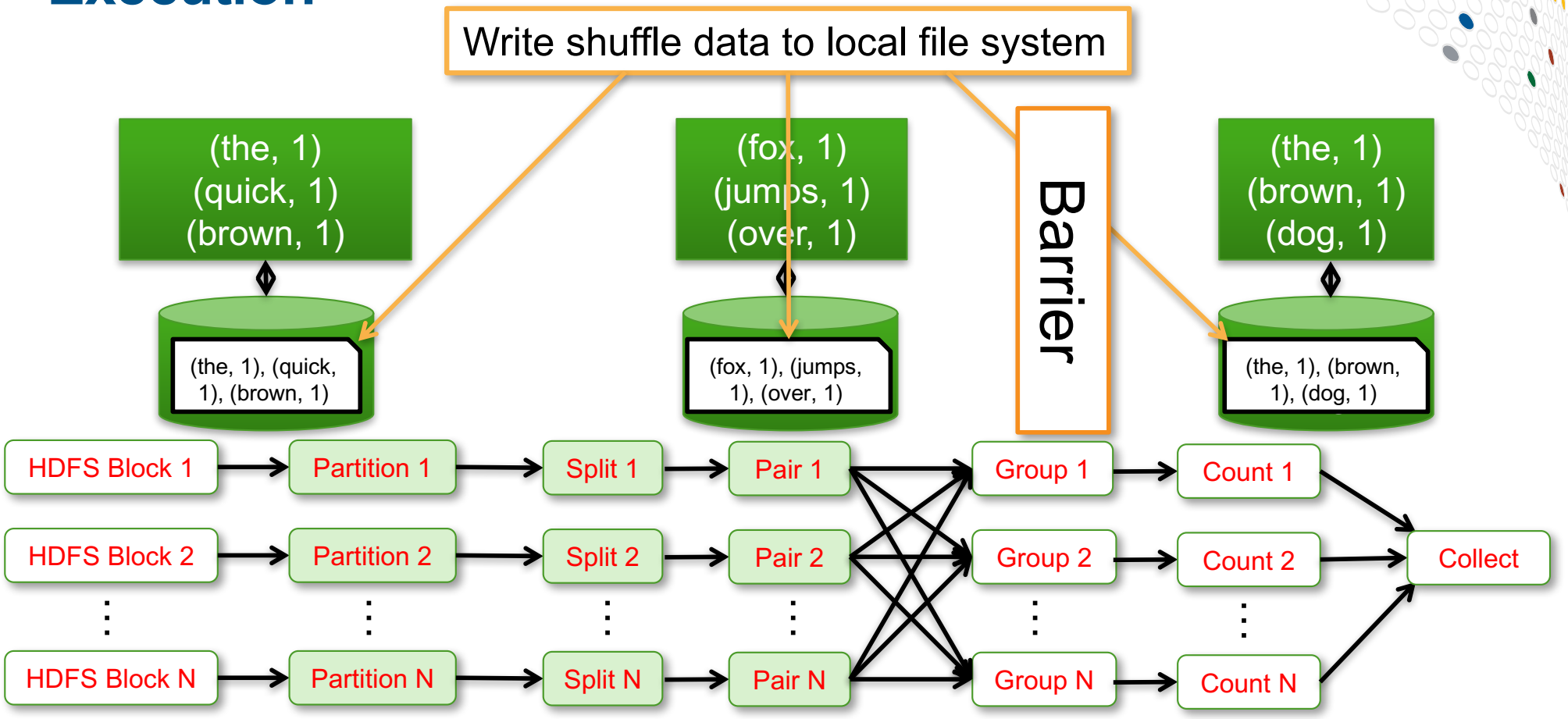
Execution



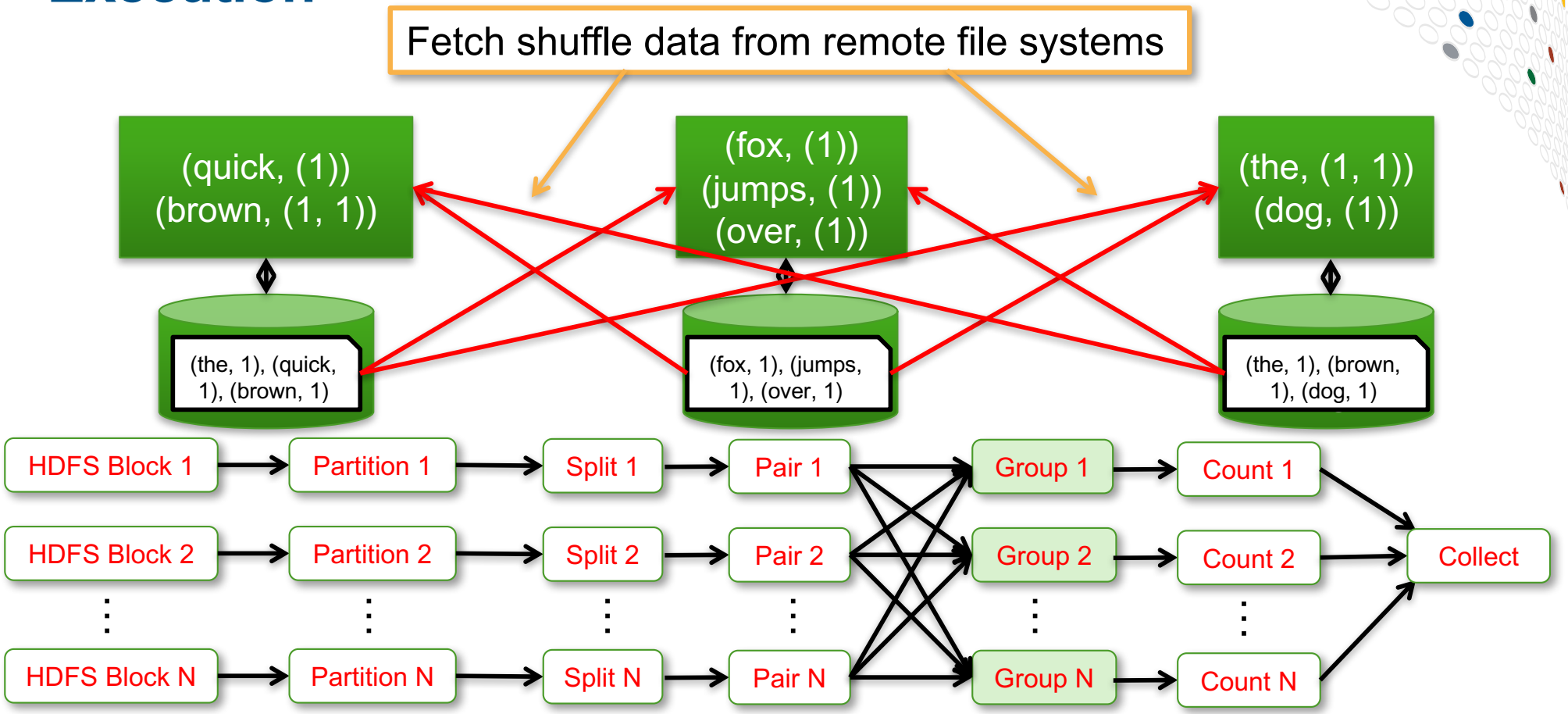
Execution



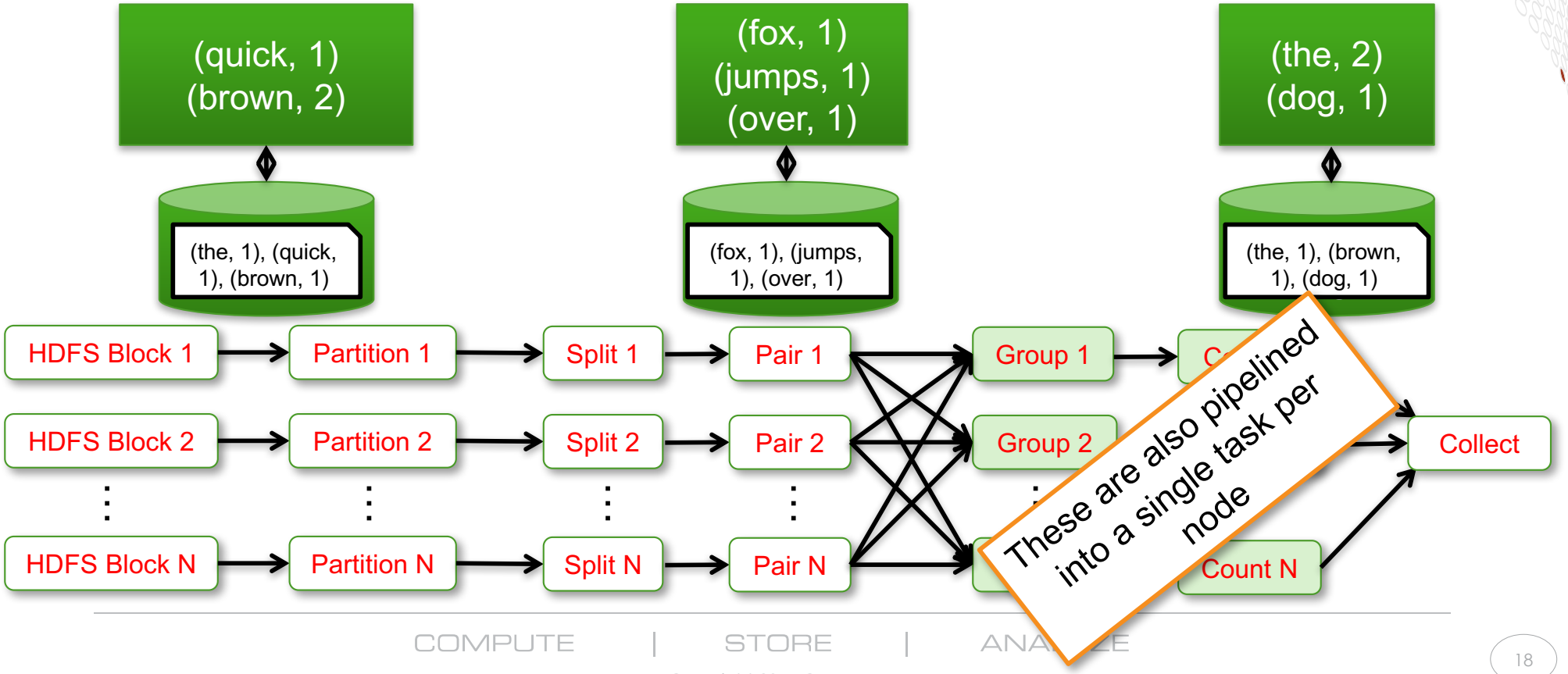
Execution



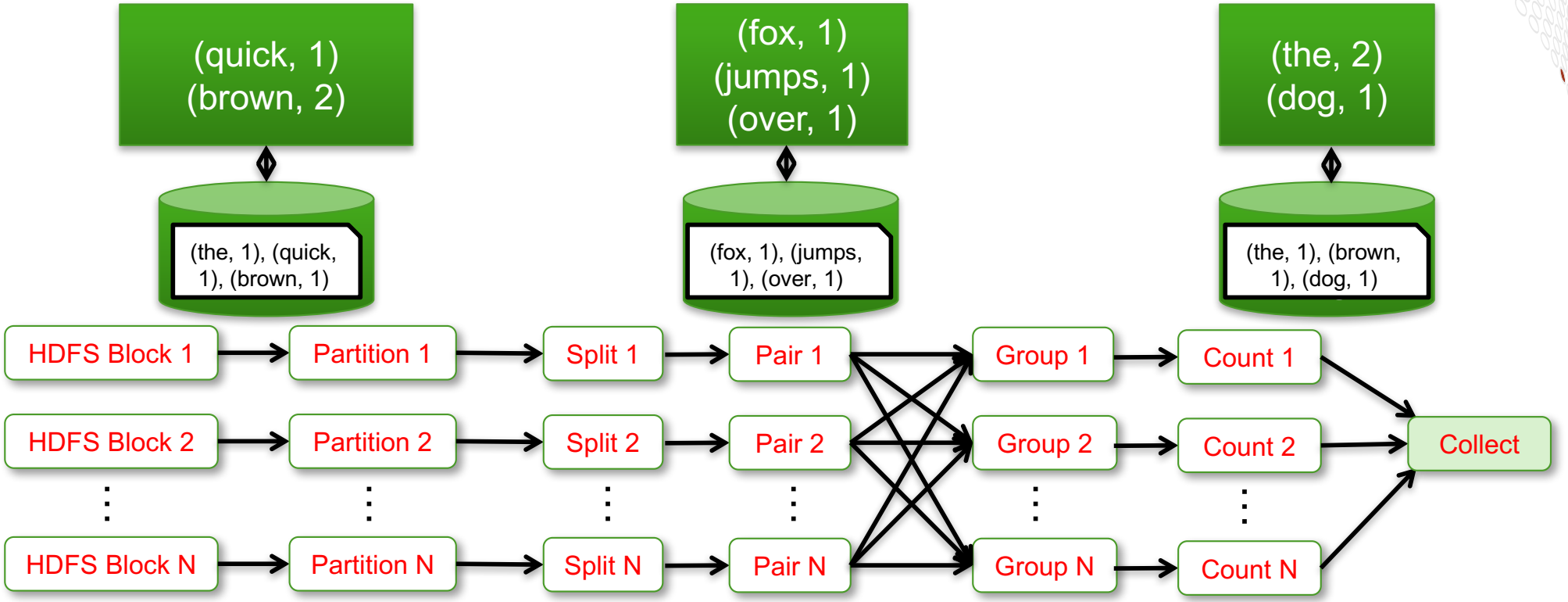
Execution



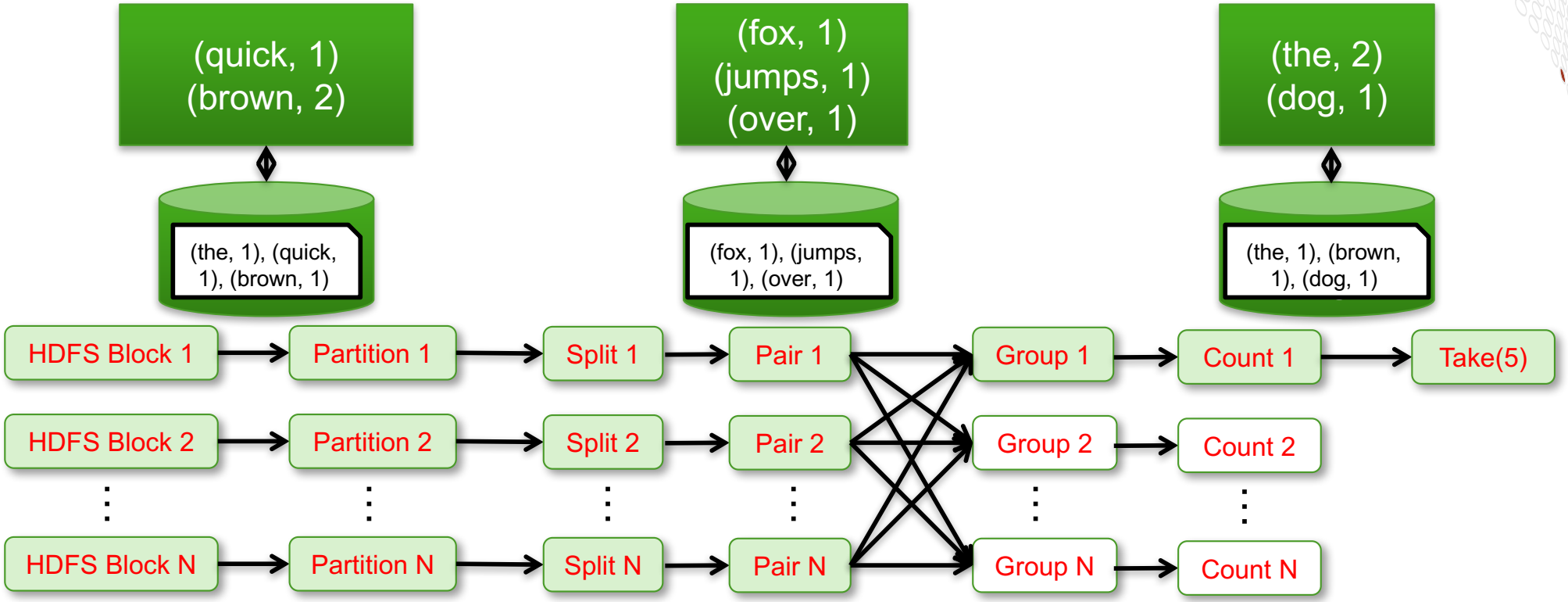
Execution

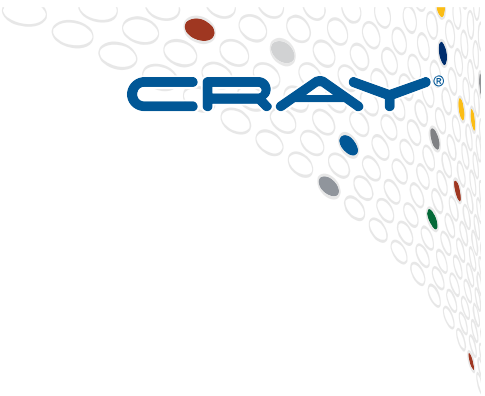


Execution



Execution





Spark on Cray XC

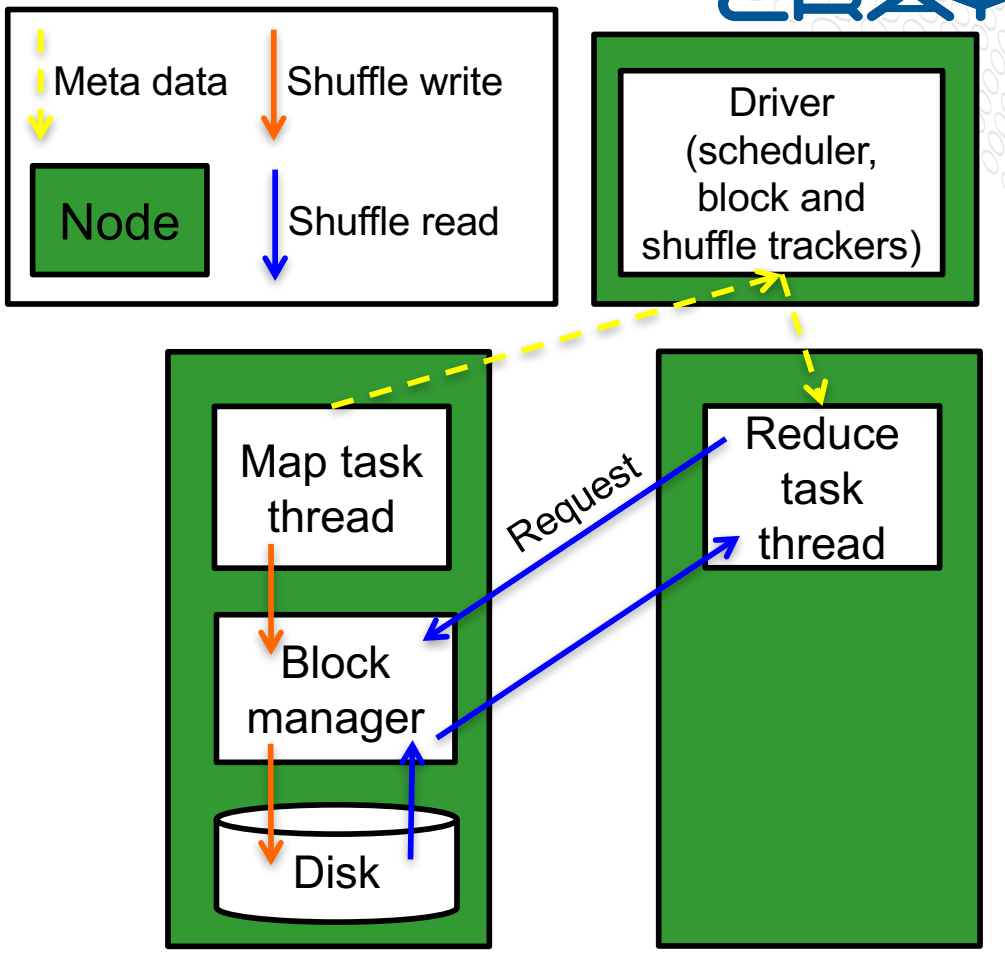
Spark on XC: Setup options

- **Cluster Compatibility Mode (CCM) option**
  - Set up and launch standalone Spark cluster in CCM mode, run interactively from mom node, or submit batch script
  - Exact details vary based on CLE version and workload manager
    - An example recipe can be found in:

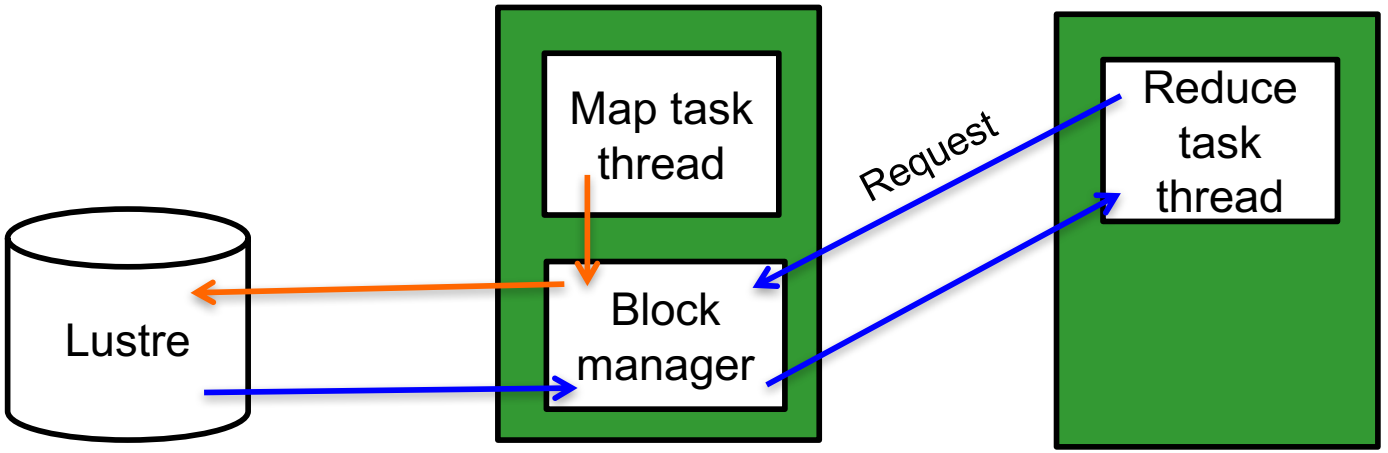
      "Experiences Running and Optimizing the Berkeley Data Analytics Stack on Cray Platforms", Maschhoff and Ringenburg, CUG 2015
- **Shifter option**
  - Shifter containerizer (think "Docker for XC") developed at NERSC
  - Acquire node allocation
    - Run master image on one node
    - Interactive image on another (or login)
    - Worker images on rest
  - Cray's analytics on XC product (in beta testing) uses this approach
- **Challenge: Lack of local storage for Spark shuffles and spills**

Reminder: Spark Shuffle – Standard Implementation

- Senders ("mappers") send data to block managers; block managers write to **local disks**, tell driver how much destined for each reducer
- Barrier until all mappers complete shuffle writes
- Receivers ("reducers") request data from block managers that have data for them; block managers read from **local disk** and send
- Key assumption: large, fast local block storage device(s) available on executor nodes

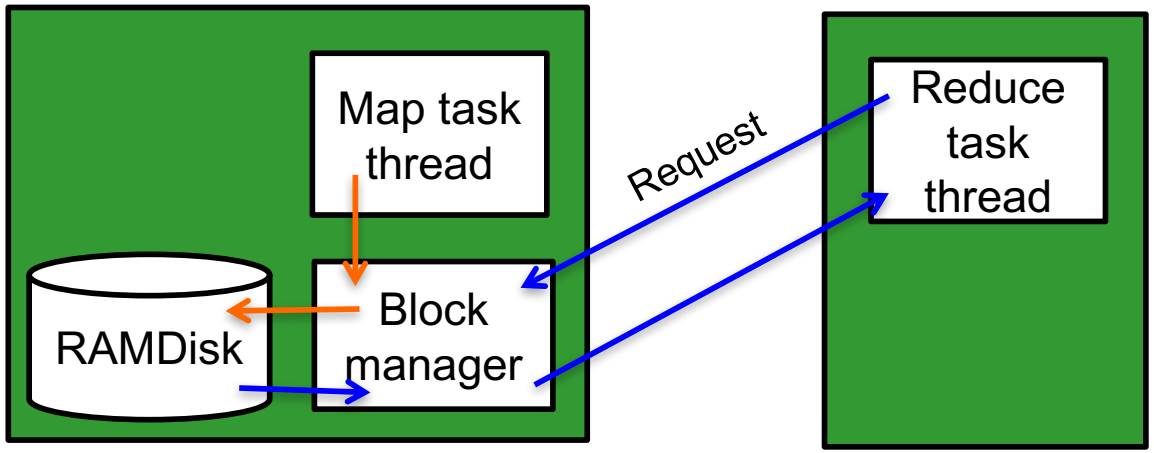


Shuffle on XC – Version 1



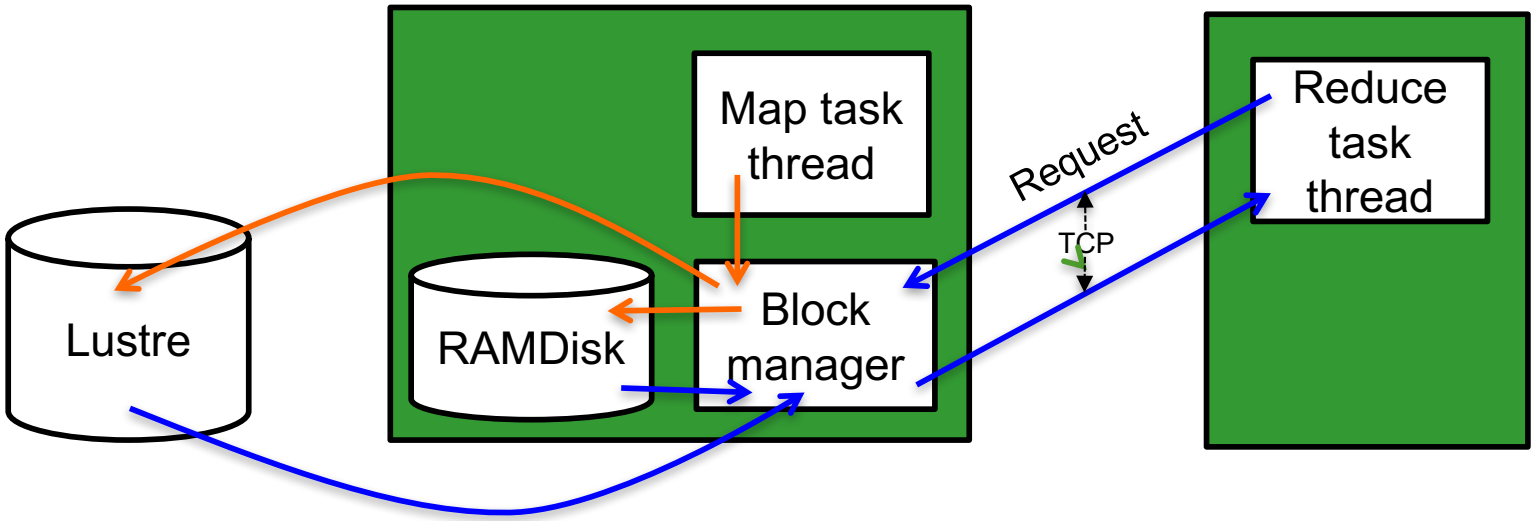
- **Problems: No local disk on standard XC40**
- **First try: Write to lustre instead**
  - Biggest Issue: Poor file access pattern for lustre (lots of small files, constant opens/closes). Creates a major bottleneck on Lustre Metadata Server (MDS).
  - Issue 2: Unnecessary extra traffic through network

Shuffle on XC – Version 2



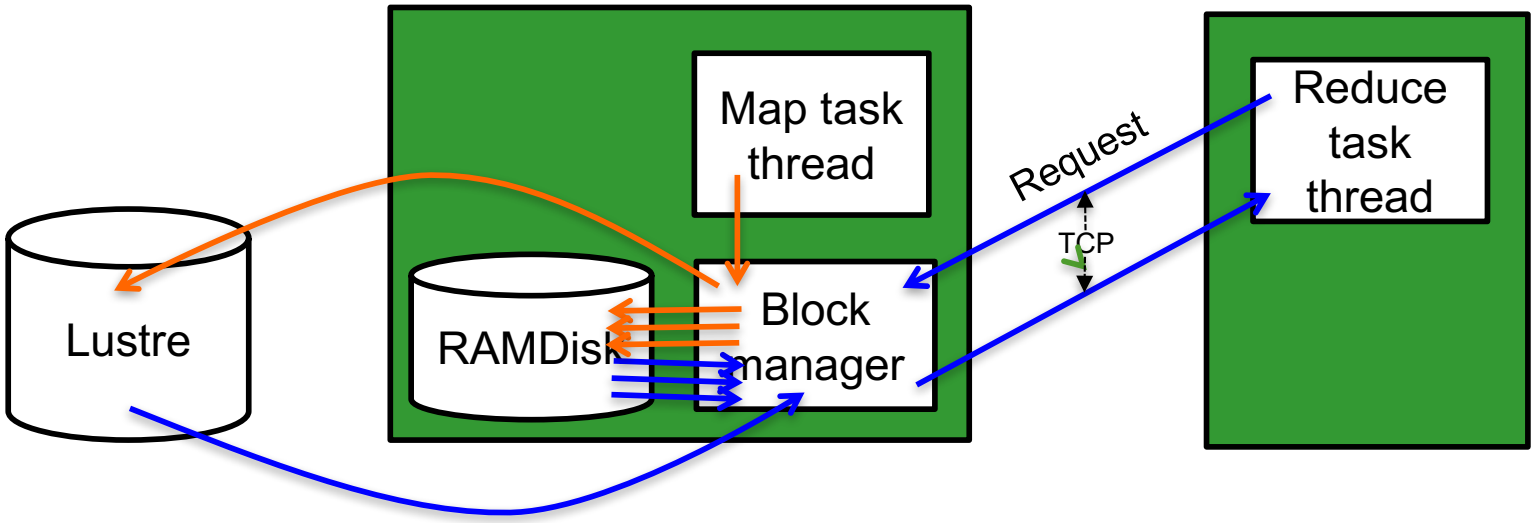
- **Second try: Write to RAMDisk**
  - Much faster, but …
  - Issues: Limited to lessor of: 50% of node DRAM or unused DRAM; Fills up quickly; Spark RAMDisk "flakiness"; takes away memory that could otherwise be allocated to Spark
  - Spark behaves unpredictably when it's local scratch space fills up (failures not always simple to diagnose)

Shuffle on XC – Version 3



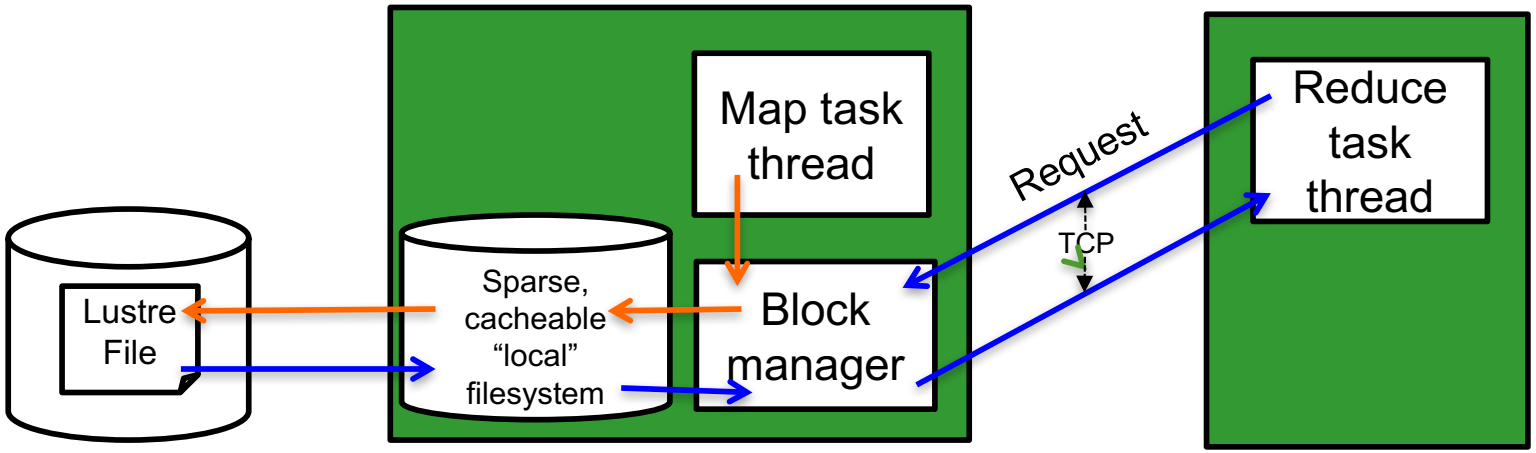
- **Third try: Write to RAMDisk and Lustre**
  - Set local directories to RAMdisk and lustre (can be list)
  - Initially fast and keeps working when RAMDisk full
  - Issues: Slow once RAMDisk fills; Round robin between directories (no bias towards faster RAM)

Shuffle on XC – Version 3



- **Third try: Write to RAMDisk and Lustre**
  - Set local directories to RAMdisk and lustre (can be list)
  - Initially fast and keeps working when RAMDisk full
  - Issues: Slow once RAMDisk fills; Round robin between directories (no bias towards faster RAM), *but can specify multiple RAM directories*

Shuffle on XC – with Shifter PerNodeCache



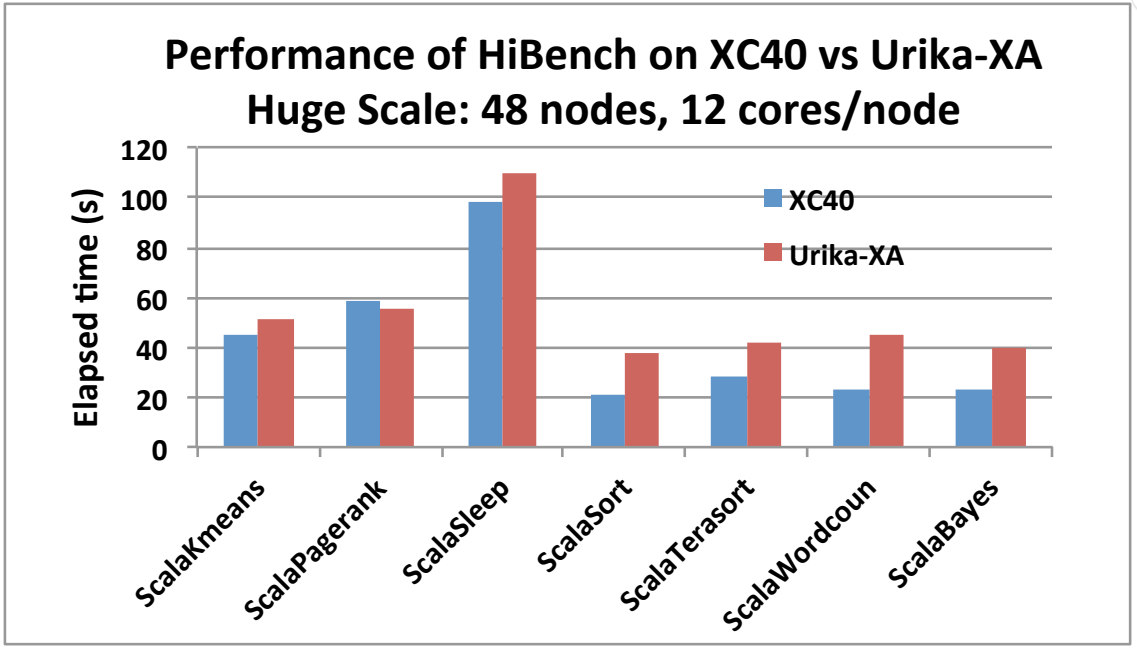
- **Shifter implementation: Per-node loopback file system**
  - NERSC's Shifter containerization (in Cray CLE6) provides optional loopback-mounted per-node temporary filesystem
  - Local to each node – fully cacheable
  - Backed by a single sparse file on Lustre – greatly reduced MDS load, plenty of capacity, doesn't waste space
  - Performance comparable to RAMDisk, without capacity constraints (Chaimov et al, CUG '16)
- **Cray's Analytics on XC project (in beta) will ship as a Shifter image, and use this approach**

Other Spark Configurations

- **Many config parameters … some of the more relevant:**
  - **spark.shuffle.compress**: Defaults to true. Controls whether shuffle data is compressed. In many cases with fast interconnect, compression and decompression overhead can cost more than the transmission time savings. However, can still be helpful if limited shuffle scratch space.
  - **spark.locality.wait**: Defaults to 3 (seconds). How long to wait for available resources on a node with data locality before trying to execute tasks on another node. Worth playing around with - decrease if seeing a lot of idle executors. Increase if seeing poor locality. (Can check both in history server.) Do not set to 0!

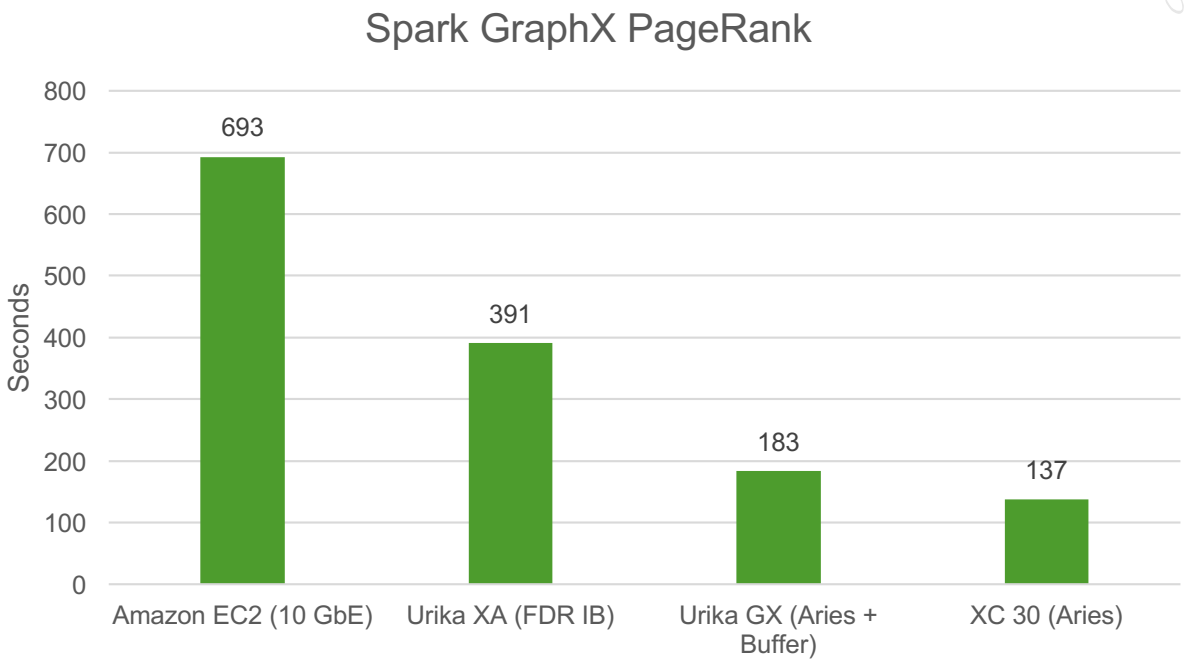
Spark Performance on XC: HiBench

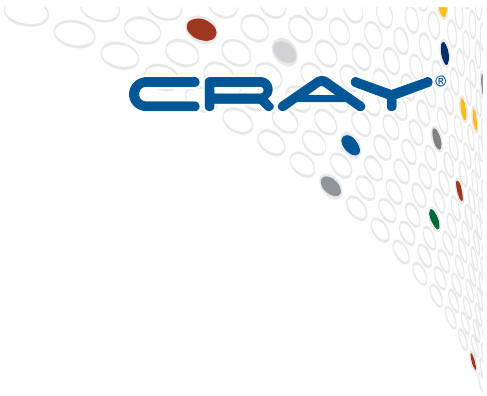
- **Intel HiBench**
  - Originally MapReduce, Spark added in version 4
- **Compared performance with Urika XA system**
  - XA: FDR Infiniband, XC40: Aries
  - Both: 32 core Haswell nodes
  - XA: 128 GB/node, XC40: 256 GB/node (problems fit in memory on both)
- **Similar performace on Kmeans, PageRank, Sleep**
- **XC40 faster for Sort, TeraSort, Wordcount, Bayes**



Spark Performance on XC: GraphX

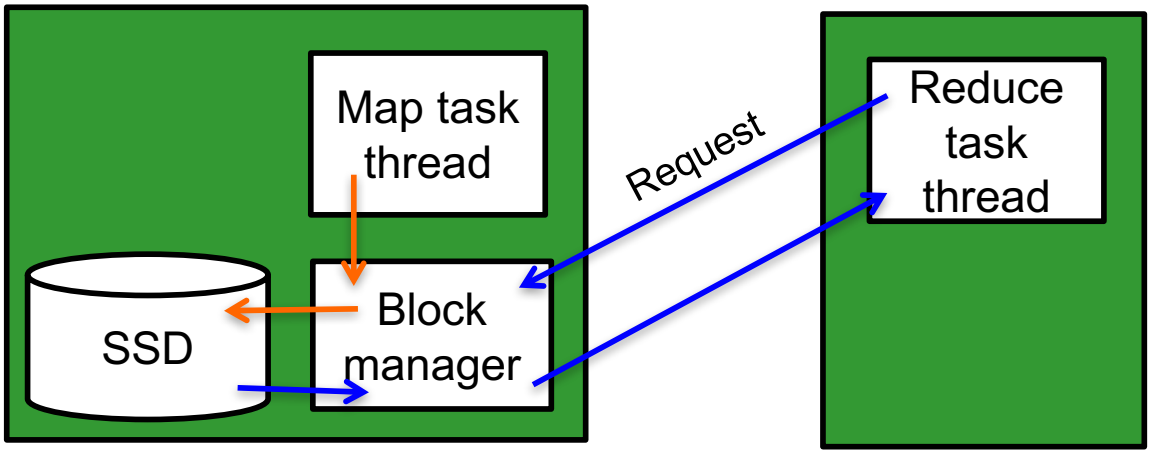
- **GraphX PageRank**
  - 20 iterations on Twitter dataset
  - Interconnect sensitive
- **GX has slightly higher latency and lower peak TCP bandwidth than XC due to buffer chip**





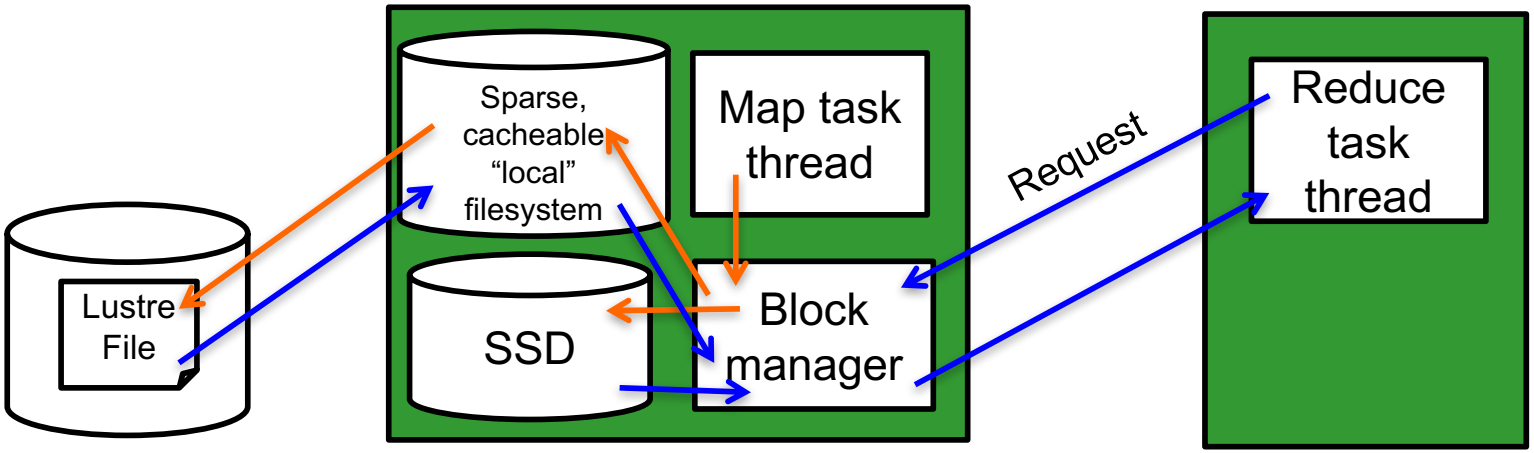
Spark on Theta

Shuffle on XC – Theta's SSDs



- **Another option, unique to Theta**
  - Theta system provides a 128GB SSD on every node, available for applications
  - Could be used as Spark local storage – this is what we do on Urika GX
    - Shown to be fast on Urika-GX
  - Slightly larger than max RAMdisk (128 vs 192/2=96), and no contention, but still a bit smaller than typically use.

Shuffle on XC – Theta's SSDs



- **Another option, unique to Theta**
  - Theta system provides a 128GB SSD on every node, available for applications
  - Could be used as Spark local storage – this is what we do on Urika GX
    - Shown to be fast on Urika-GX
  - Slightly larger than max RAMdisk (128 vs 192/2=96), and no contention, but still a bit smaller than typically use. Could be used in combination with RAMDisk, or Loopback w/ Shifter?

Spark on KNL

- **Cray and Intel have recently started a collaboration to investigate and improve Spark on KNL performance**
  - Java and Spark currently run
  - Performance vs Skylake varies from 20% slower to >4x slower
  - “Typical” benchmarks at larger sizes ~3x slower than a dual-socket Skylake node
    - Still early … just starting to benchmark and profile.
  - Looking at issues, profiling, attempting to identify causes and potential solutions.

Early findings and tips

- **Lots of skinny executors work better than fewer fatter executors**
  - On Xeon-based nodes this is not necessarily the case – fat often works nearly as well or occasionally better
  - On KNL, though, often find best results with 1-2 cores per executor
    - Make sure to adjust executor memory appropriately – all about memory/core
    - E.g., 64 executors with 1 core and 2GB each, rather than 1 executor with 64 cores and 128 GB
  - Skinny executors have better memory locality
  - Skinny executors also have less JVM overhead
  - JVM has issues scaling to many threads, e.g., https://issues.scala-lang.org/browse/SI-9823 (cache thrashing with isInstanceOf)
- **Hyperthreading generally not helpful for Spark (on either Xeon or Xeon Phi)**

Early findings and tips

- **Limit GC parallelism from JVM**
  - E.g., -XX:+UseParallelOldGC -XX:ParallelGCThreads=<N>, where N ≤ available threads/# executors
  - Especially important with lots of skinny JVMs
    - Otherwise each JVM will try to grab 5/8 total threads
- **MCDRAM configured as cache works best with Spark**
  - Seeing ~43% of accesses coming from MCDRAM, ~11% directly from DDR
  - Currently no ability in JVM to take advantage of MCDRAM in flat mode

Current Performance Gap

Benchmark	KNL	SKX-EP	KNL/SKX
Spark terasort large (3.2 GB)	136 sec	34 sec	4x
Spark terasort huge (32 GB)	343 sec	94 sec	3.6x
Spark terasort gigantic (320 GB)	2570 sec	800 sec	3.2x

- **Single node performance**
  - **KNL node**: 192 GB DDR4, 16 GB MCDRAM, 2TB P3700 PCIe SSD
  - **Skylake node**: Dual-socket node, 384GB DDR4, 2TB P3700 PCIe SSD

Summary

- **Spark runs well on XC systems**
  - Key is to intelligently configure local scratch directories
  - Analytics on XC project, including Spark, in Beta testing
    - Also planning to include Intel BigDL: Optimized Deep Learning in Spark
- **Cray and Intel collaborating to understand and improve performance of Spark on KNL/Xeon Phi systems**
  - Initial tips:
    - Skinny executors
    - Limit GC threads
  - On roadmap for future versions of Analytics on XC

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

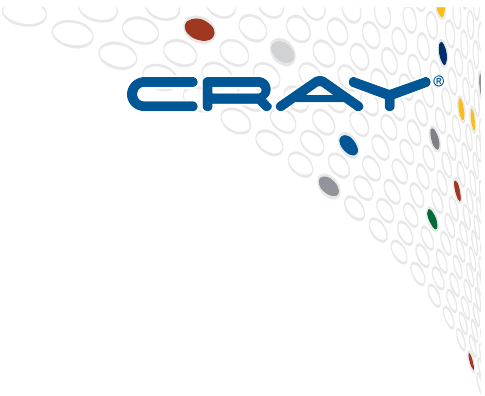
All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, REVEAL, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.



Questions?