



CRAY™

Spark and Alchemist

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 XC and Theta**
 - Installation and Configuration
 - Local storage
- **Spark on KNL**
- **Spark with Alchemist**
- **A Spark Deep Learning Use Case with BigDL**

In the beginning, there was Hadoop MapReduce...

- **Simplified parallel programming model**

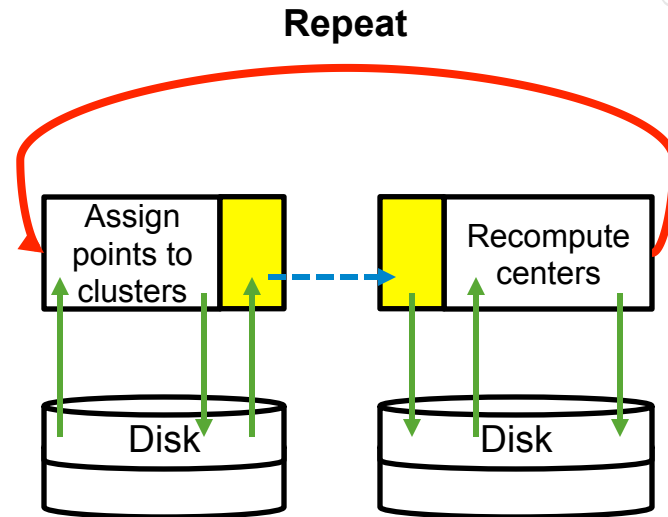
- All computations broken into two parts
 - **Embarrassingly parallel map phase:** apply single operation to every key-value pair, produce new set of key-value pairs
 - **Combining reduce phase:** Group all values with identical key, perform combining operation to get final value for each key
- Can perform multiple iterations for computations that require them
- I/O intensive
 - Map writes to local storage. Data shuffled to reducer's local storage, reduce reads from local storage.
 - 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

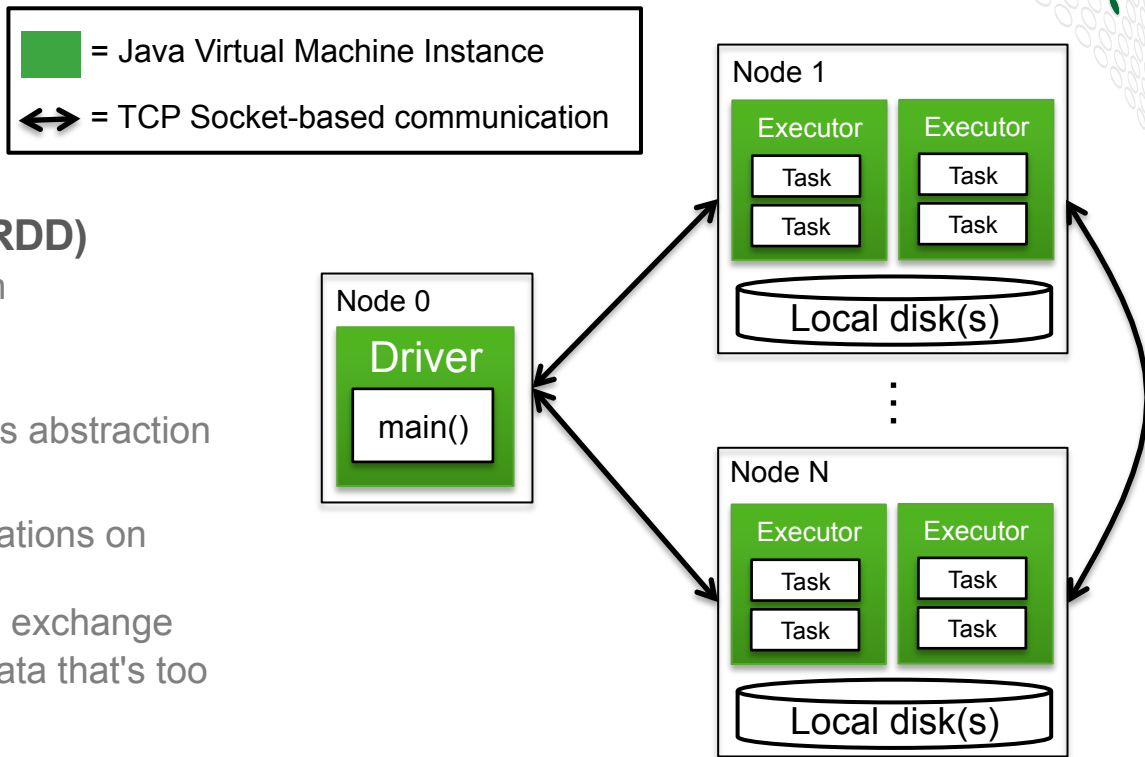
- **Gated on IO bandwidth, possibly interconnect as well**
 - Must write and read between map and reduce phases
 - Multiple iterations must read in previous results each time (e.g., new cluster centers)
- **No ability to persist reused data**
- **Must re-factor all computations as map then reduce, and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

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

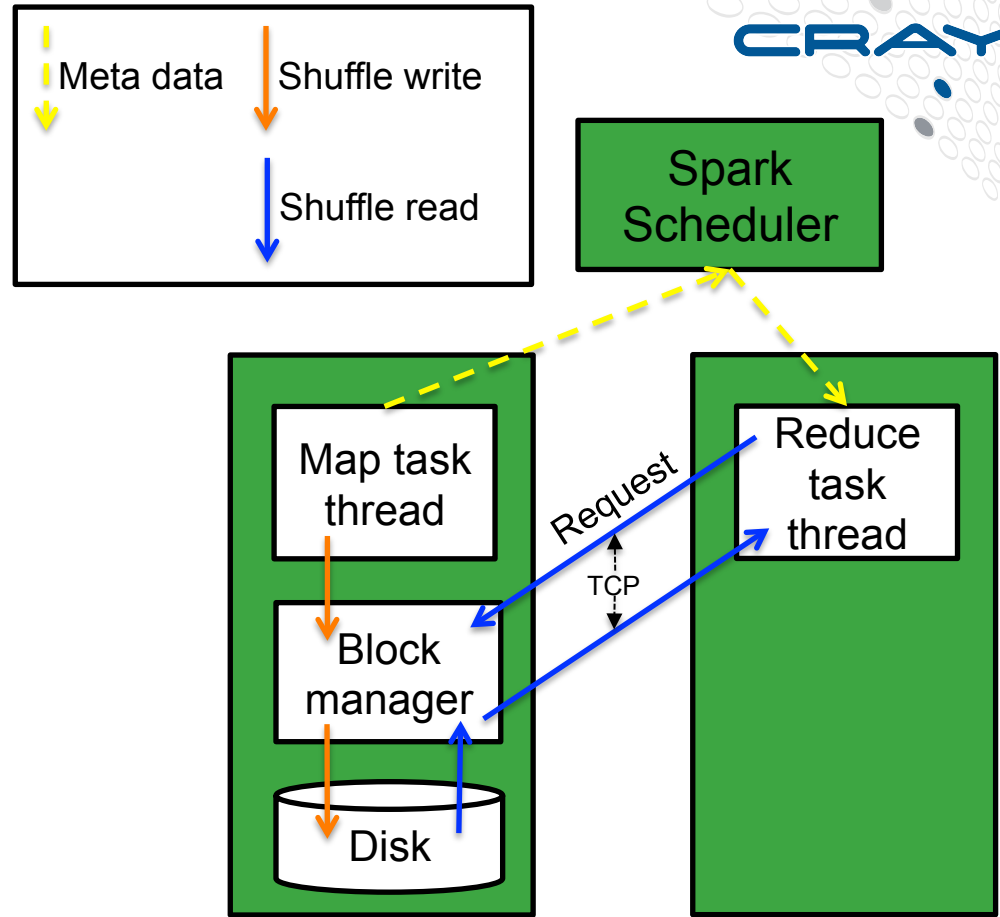
- **Master-slave parallelism**
- **Driver (master)**
 - Executes main
 - Distributes work to executors
- **Resilient Distributed Dataset (RDD)**
 - Spark's original data abstraction
 - Partitioned amongst executors
 - Fault-tolerant via lineage
 - Dataframes/Datasets extend this abstraction
- **Executors (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



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

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

Spark on Cray XC

Spark on XC: Typical 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
- **Container option**
 - Shifter container runtime (think “Docker for XC”) developed at NERSC
 - Acquire node allocation: run master image on one node, interactive image on another, worker images on rest
 - Cray’s Urika-XC analytics suite uses this approach
- **Challenge: Lack of local storage for Spark shuffles and spills.**
Options:
 - Lustre: slow, especially at scale, due to heavy metadata overhead (opens/closes)
 - Tmpfs ramdisk in /tmp: fast, but limited size and dependent on application memory footprint
 - Loopback filesystem on each node, backed by lustre file-per-node: dedicated space per node, solves lustre metadata problems (1 open/close per node)
 - Shifter has this capability built-in, used by Urika-XC

Spark on Theta: Alternative Setup Options

- **Theta differences**

- CCM not enabled, but does have ssh between nodes (--attrs=enable_ssh=1)
- Shifter not enabled
- Each node has a local 128 GB SSD (see Paul Coffman's I/O talk)

- **Option 1:**

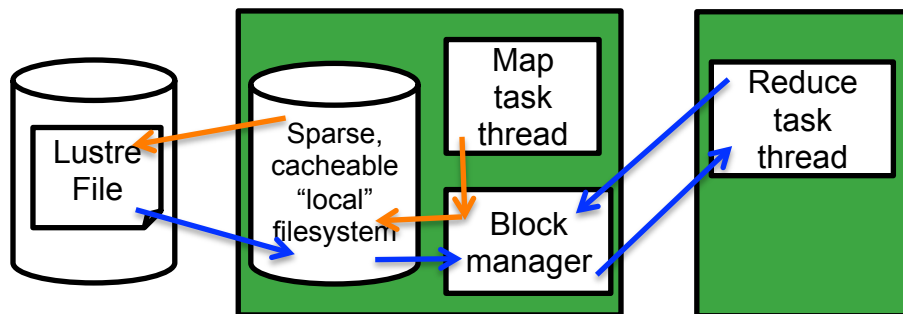
- Launches SPARK master and slaves on the theta Compute nodes
- `$ /PATH/TO/SPARK_JOB/submit-spark.sh -A datascience -t 10 -n 2 -q debug-cache-quad run-example SparkPi`

- **Option 2:**

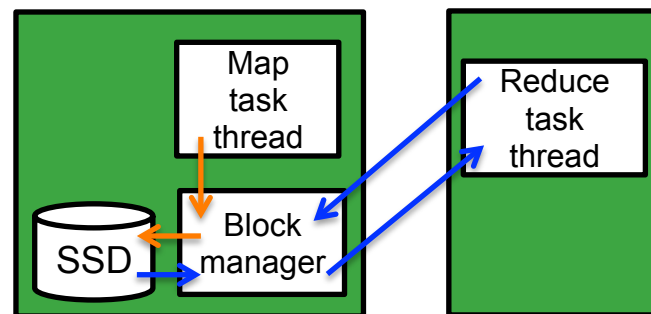
- Potentially using Singularity.

Shuffle on XC – Theta’s SSDs

Typical Loopback Setup



SSD-based Alternative



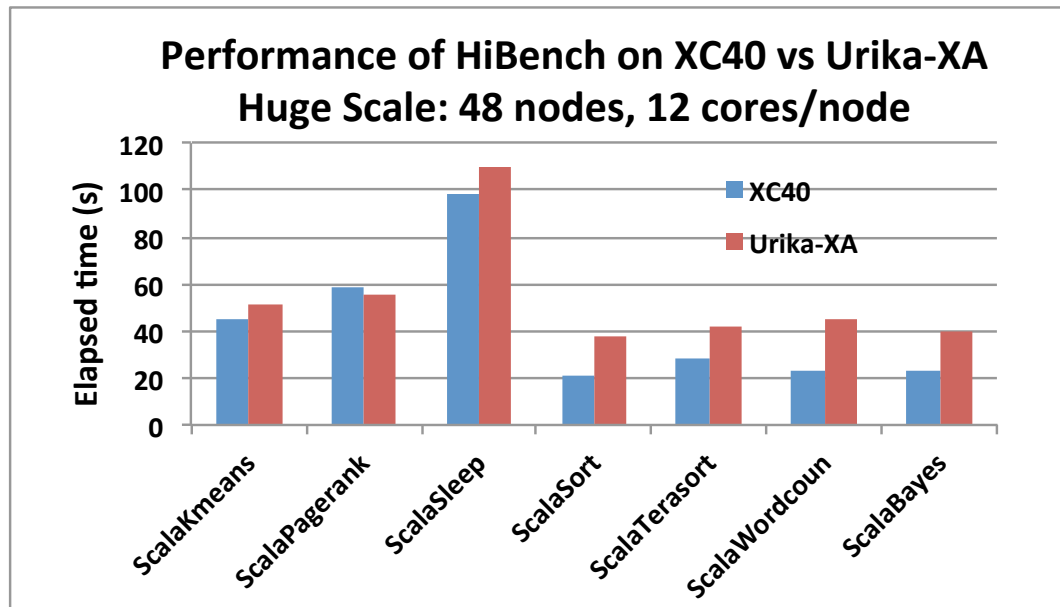
- **Typical approaches to storage: tmpfs RAMdisk or loopback filesystem**
- **Another option unique to Theta:**
 - Theta system provides a 128GB SSD on every node, available for applications
 - Could be used as Spark local storage – this is what we do on Urika GX
 - Larger than max RAMdisk, and no contention for space

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!

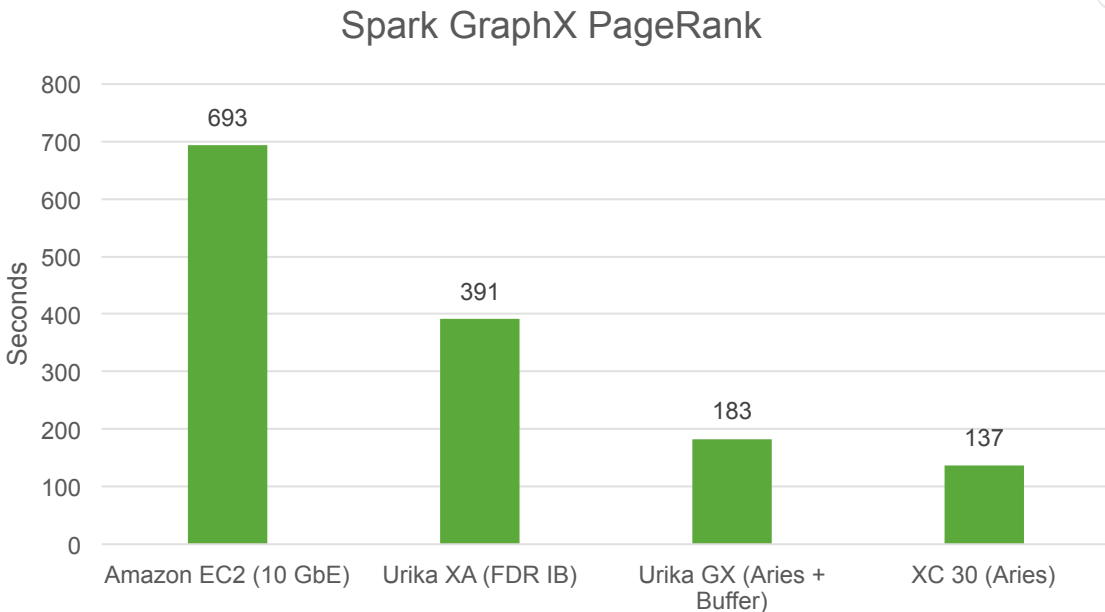
Spark Performance on XC: HiBench

- **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 performance 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 KNL

- **Java and Spark currently run**
 - JVM performance is fairly poor – garbage collection on KNLs is a big chunk of this
- **Performance vs Skylake varies from 20% slower to ~4x slower**
- **“Typical” benchmarks at larger sizes ~3x slower than a dual-socket Skylake node**
 - Can sometimes shrink by carefully tuning parameters

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)
 - On KNL – play around with these settings – can be very sensitive
- **Hyperthreading generally not helpful for Spark**



Early findings and tips

- **Limit GC parallelism from JVM**
 - E.g., `-XX:+UseParallelOldGC -XX:ParallelGCThreads=<N>`, where $N \leq \text{available threads}/\# \text{ executors}$
 - 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

Summary

- **Spark runs well on XC systems**
 - Key is to intelligently configure local scratch directories
- **KNL has some challenges, but can run reasonably with some tuning**
 - JVM on KNL seems to be the biggest issue ... is there a way to remove that from the equation?
 - Perhaps, at least partially ...

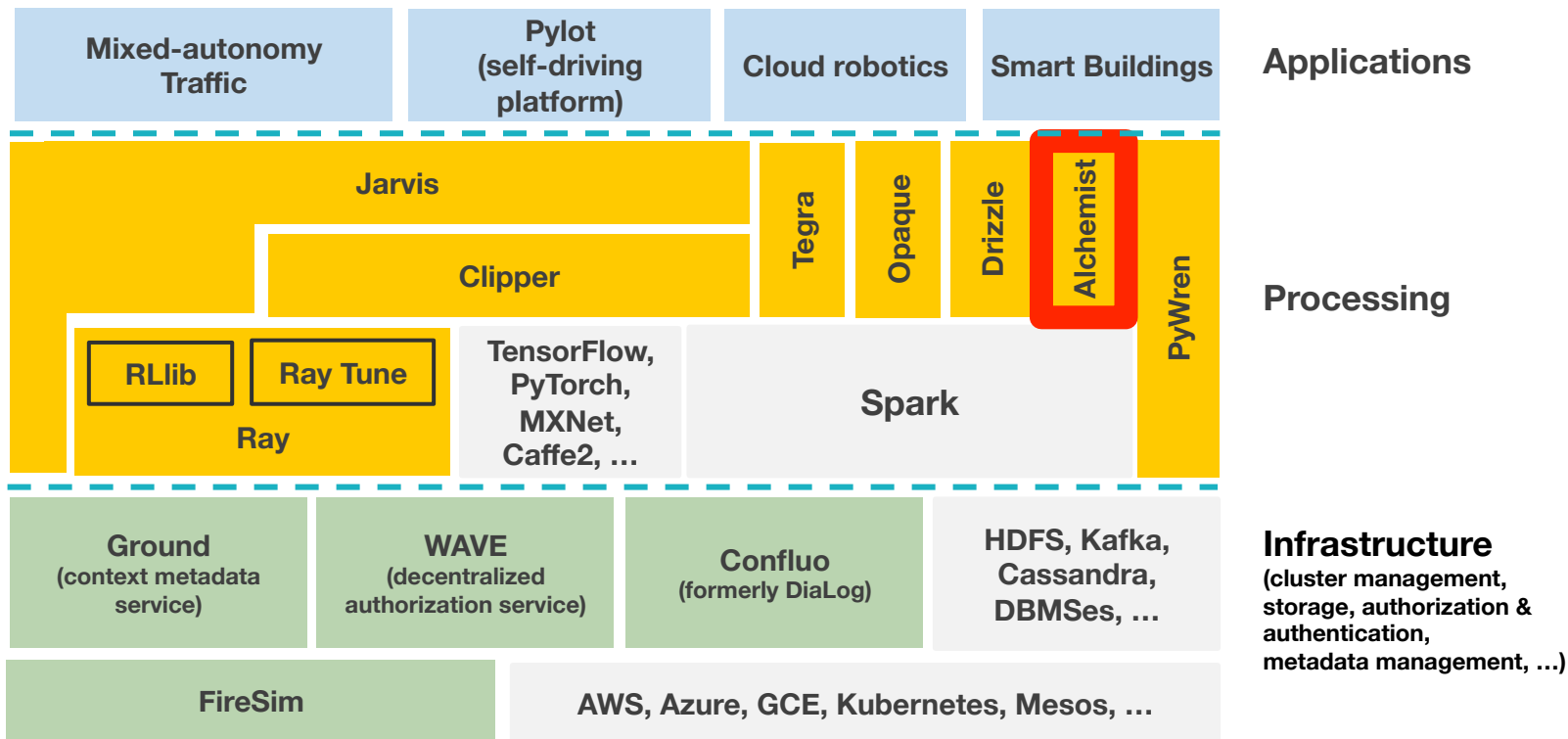
Alchemist

An Apache Spark ↔ MPI Interface

A Collaboration of Cray and the UC Berkeley RiseLab (Alex Gittens, Kai Rothauge, Michael W. Mahoney, Shusen Wang, Jey Kottalam)

Slides courtesy Kai Rothauge

RISE Stack

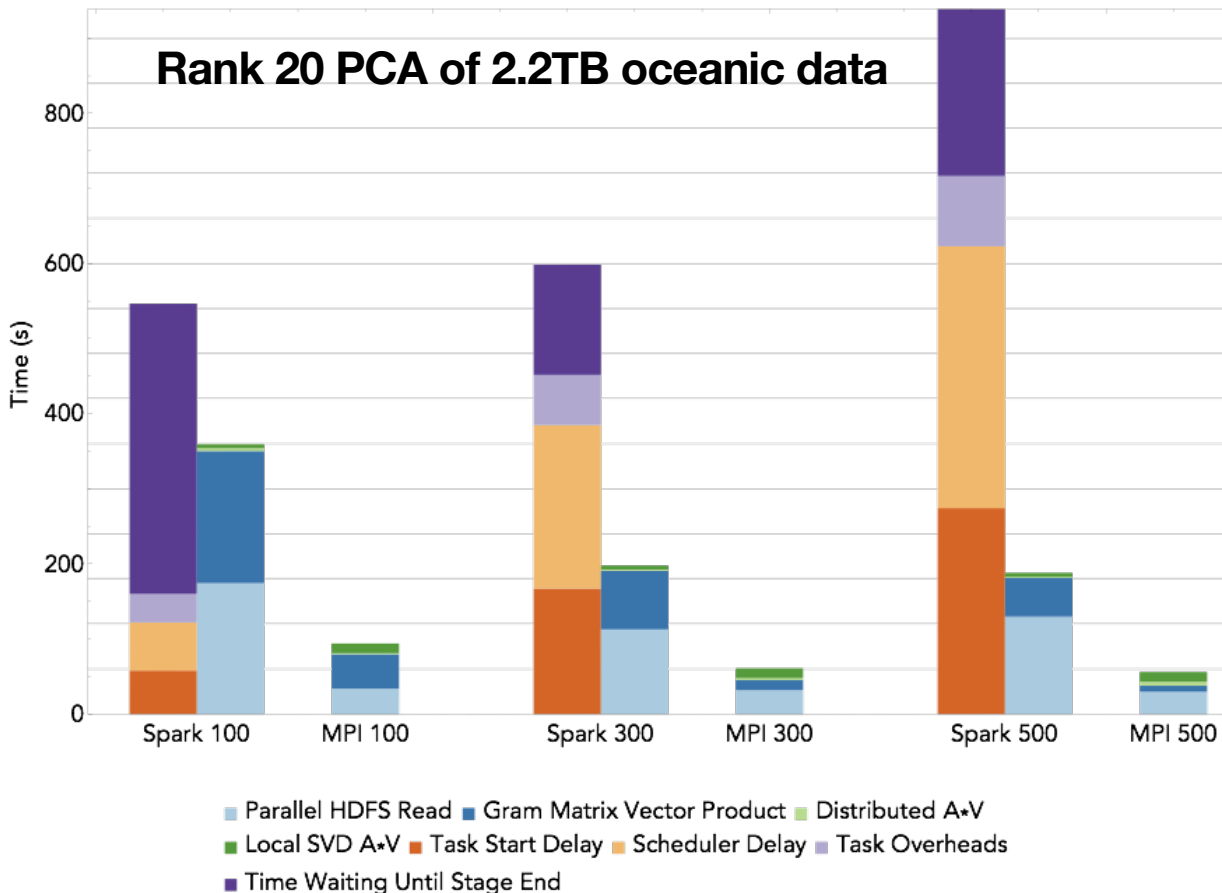


MPI vs Spark

- Cray and AMPLab performed case study for numerical linear algebra on Spark vs. MPI
- Why do linear algebra in Spark?
 - **Pros:**
 - Faster development, easier reuse
 - One abstract uniform interface (RDD)
 - An entire ecosystem that can be used before and after the NLA computations
 - Spark can take advantage of available local linear algebra codes
 - Automatic fault-tolerance, out-of-core support
 - **Con:**
 - Classical MPI-based linear algebra implementations will be faster and more efficient

MPI vs Spark

- Performed a case study for numerical linear algebra on Spark vs. MPI:
 - Matrix factorizations considered include Principal Component Analysis (PCA)
 - Data sets include
 - Oceanic data: 2.2 TB
 - Atmospheric data: 16 TB






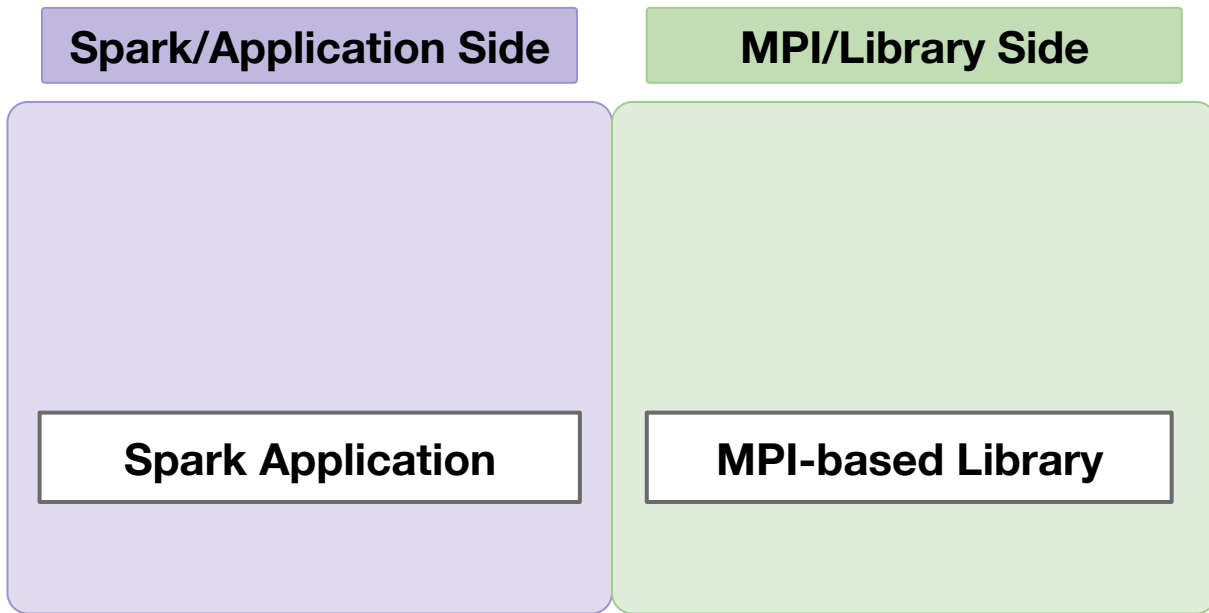
A. Gittens et al. "Matrix factorizations at scale: A comparison of scientific data analytics in Spark and C+MPI using three case studies", 2016 IEEE International Conference on Big Data (Big Data), pages 204–213, Dec 2016.

MPI vs Spark: Lessons learned

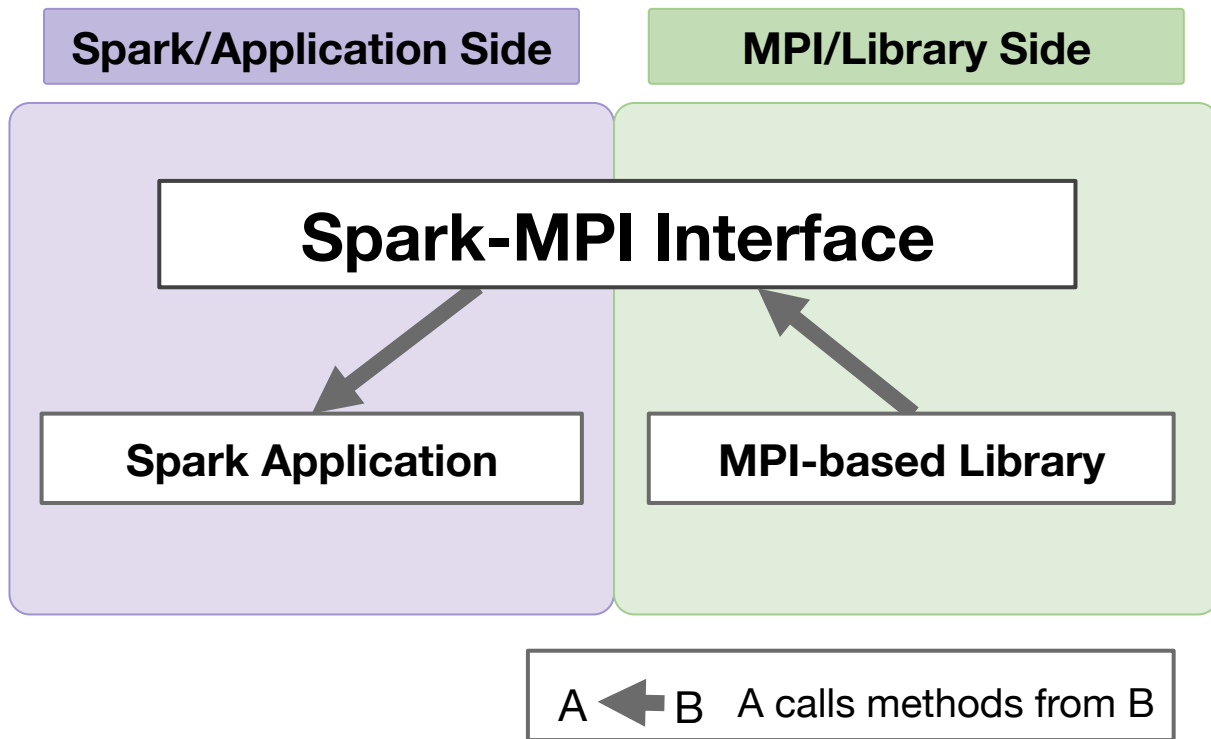
- With favorable data (tall and skinny) and well-adapted algorithms, linear algebra in Spark is 2x-26x slower than MPI when I/O is included
- Spark's overheads are orders of magnitude higher than the actual computations
 - Overheads include time until stage end, scheduler delay, task start delay, executor deserialize time, inefficiencies related to running code via JVM
- **The gaps in performance suggest it may be better to interface with MPI-based codes from Spark**

Alchemist

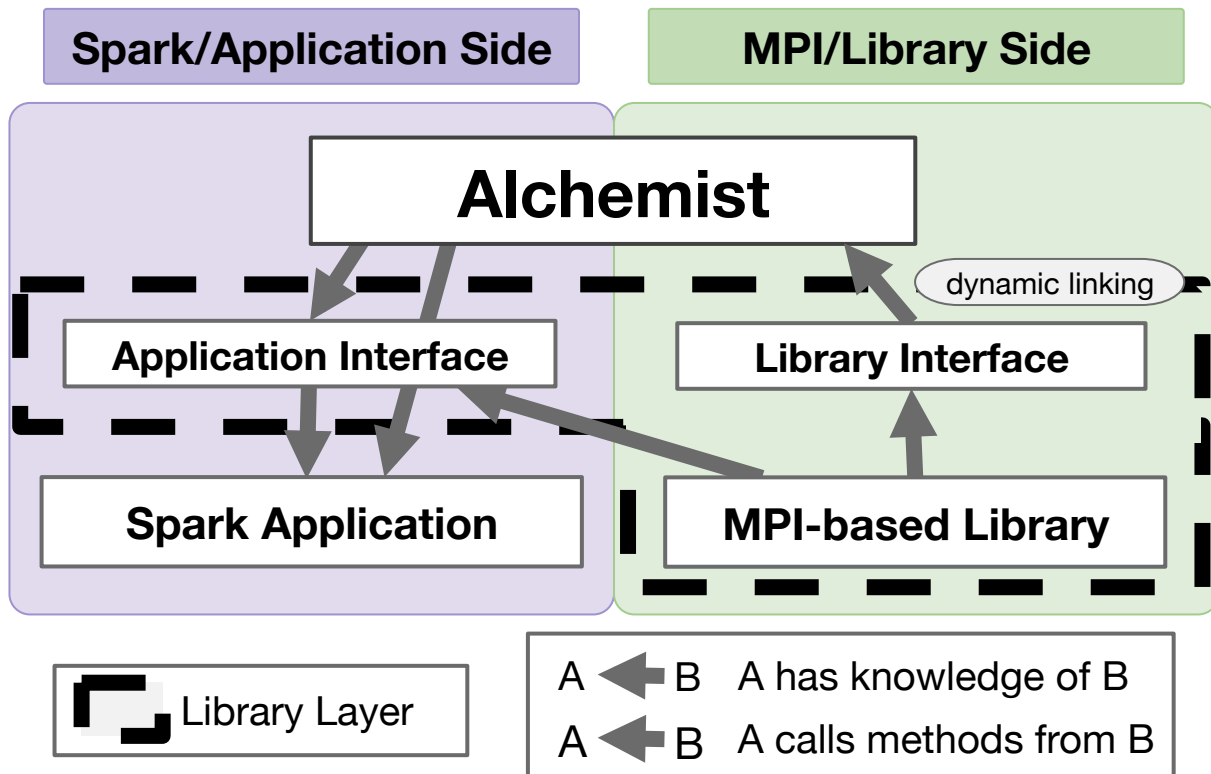
- Interface between Apache Spark and *existing* MPI-based libraries for NLA, ML, etc.
- Design goals include making the system *easy to use, efficient, and scalable*
- Two main tasks:
 - Send distributed **input** matrices from Spark to MPI-based libraries (**Spark => MPI**)
 - Send distributed **output** matrices back to Spark (**Spark <= MPI**)
- Want as little overhead as possible when transferring data between Spark and a library
- Three possible approaches:
 - File I/O (e.g. HDFS) **too slow!** 
 - Use shared memory buffers, Apache Ignite, Alluxio, etc. **extra copy in memory** 
 - Use in-memory transfer, send data between processes using sockets 



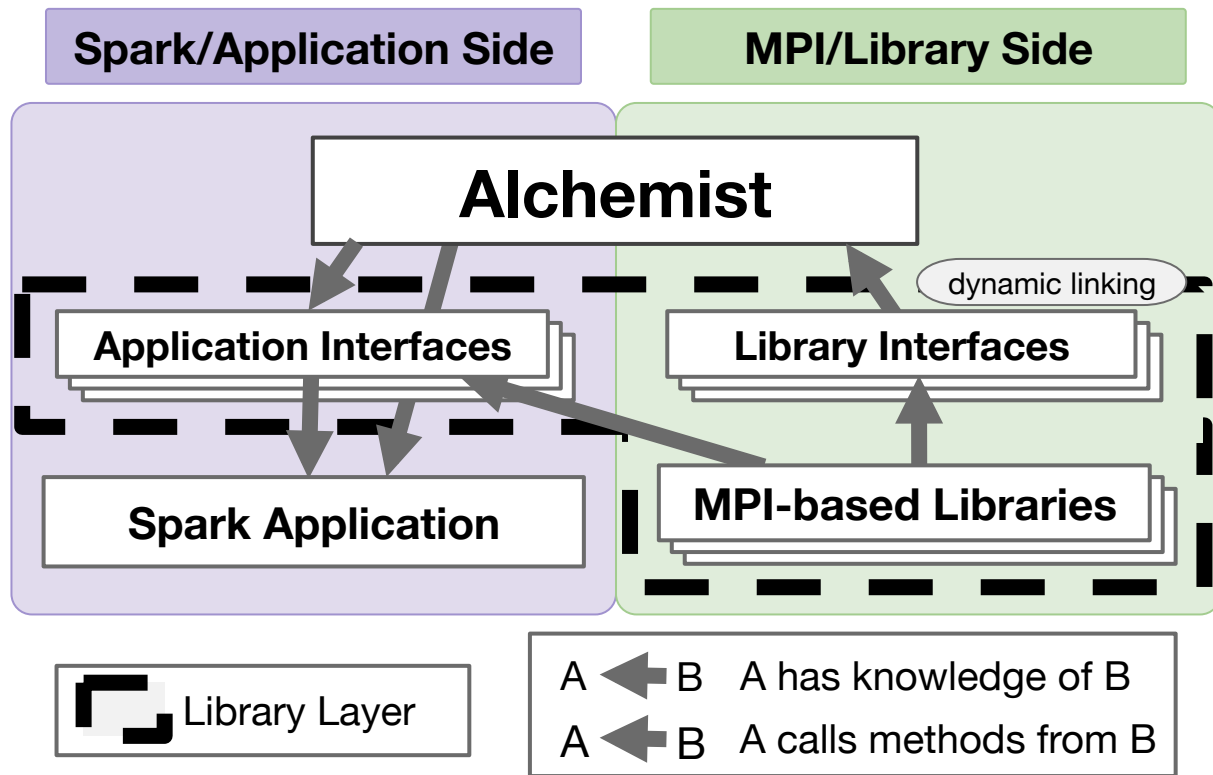
Naive Approach to combining Spark and MPI



Alchemist Architecture

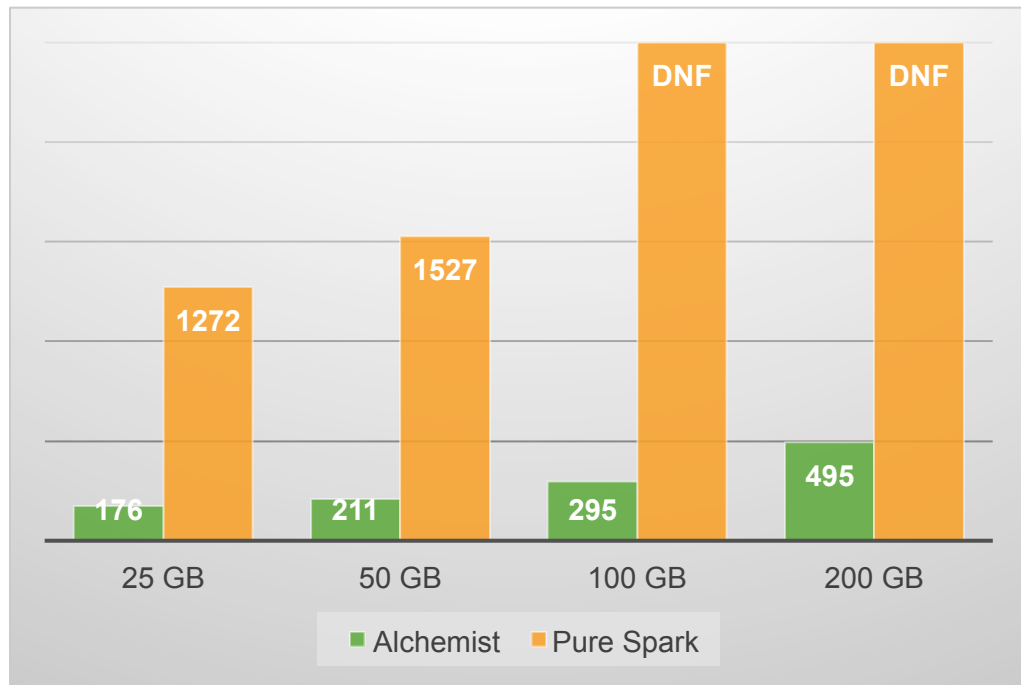


Alchemist Architecture



Truncated SVD Alchemist vs Pure Spark

- Use Alchemist and MLib to get rank 20 truncated SVD
- Setup:
 - 30 KNL nodes, 96GB DDR4, 16GB MCDRAM
 - Spark: 22 nodes; Alchemist: 8 nodes
 - A: m-by-10K, where m = 5M, 2.5M, 1.25M, 625K, 312.5K
 - Ran jobs for at most 60 minutes (3600 s)
- Alchemist times include data transfer



Future Work

- Support for sparse matrices
- Support for MPI-based libraries built on ScaLAPACK
- Enable running on AWS EC2
- Ray + MPI?
- Still research code. Numerous code improvements in progress:
 - Improved error handling and testing
 - Support for additional Spark distributed matrix layouts
 - Improved communication between Spark and libraries by using locality information
 - Cray team working closely with Berkeley team to harden and productize code, and add it to Cray's analytics stack.

Try it out at github.com/alexgittens/alchemy

A Spark Deep Learning Use Case



Precipitation Nowcasting

- **Problem: Predict precipitation locations and rates at a regional level over a short timeframe**
 - Neighborhood level predictions
 - T+0 – T+6 hours
- **Standard Approach: Numerical Weather Prediction**
 - Physics based simulations
 - High computational cost limits performance and accessibility
- **Cutting edge approach: Deep Learning**
 - Predict rainfall by learning from historical data
 - Heavy computation occurs ahead of time
 - Pre-Trained models can be deployed as soon as data is available

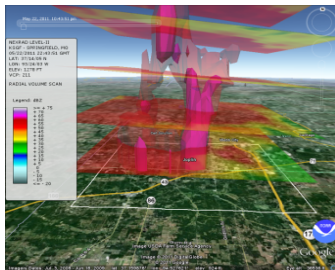
Motivation

- **Increase the quality and availability of very short term (0-1 hour) precipitation forecasts**
 - Will it rain on my walk home from work if I leave right now?
 - Which bike-route should I take to avoid the rain?
- **Improve tracking quality of severe precipitation events**
 - Where do we issue severe weather warning?
 - Is a flash flood imminent? Do we need to evacuate?
- **Gain insights into the full deep learning workflow**
- **Accelerate the integration of deep learning in operational meteorology**

Data Processing Pipeline

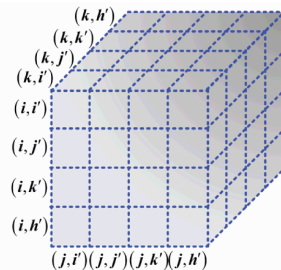
Data Collection

- Historical Radar Data (NETCDF)
- Geographical Region (Eg:- Seattle)
- Days with over 0.1 inches of precipitation, info from NOAA - NCDC
- Radar scans every 5-10 minutes throughout the day



Transformation

- Raw radial data structure converted to evenly spaced Cartesian grid (Tensors with float 32)
- Resolution scaling and clipping
- Configure dimensionality
- Sequencing
- 2 channels - Reflectivity, Velocity
- Uses Py-ART package



Sampling

- Time-series
- Inputs and Labels
- Random sampling



COMPUTE

STORE

ANALYZE

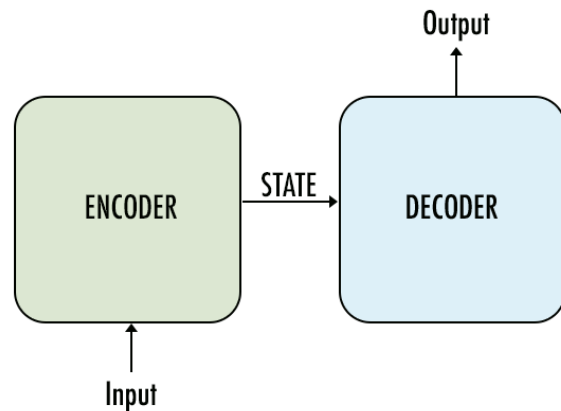
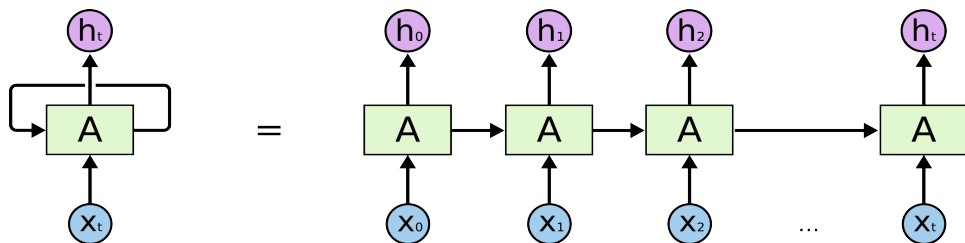
Neural Network Model

- **Convolutional Recurrent Neural Network**

- Convolutional Neural Network – Spatial Patterns
- Recurrent Neural Network – Temporal Patterns
- ConvLSTM – Convolutional Long Short-Term Memory Network

- **Sequence to Sequence**

- Encoder Decoder
- Use recent history to predict future changes



COMPUTE

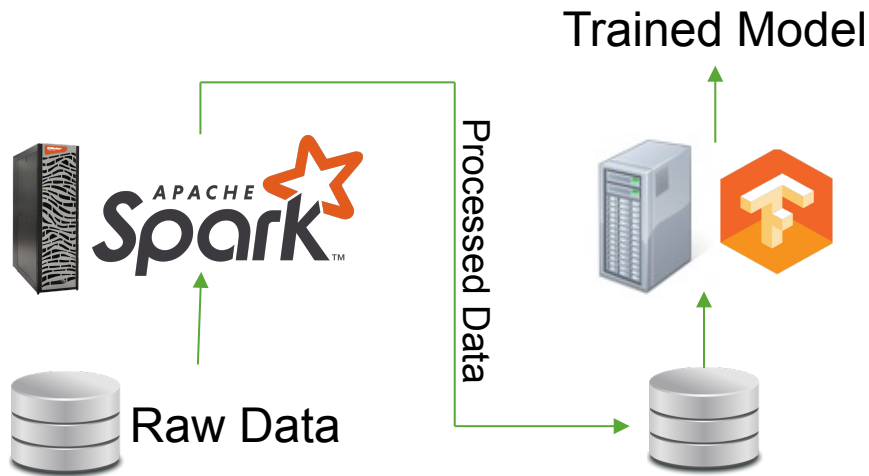
STORE

ANALYZE

Implementation: Tensorflow + Spark

- **Separate workflows – no integration**
 - Forced overhead – data movement
 - Distinct data pipelines
- **Data processing – highly distributed analytics platform**
- **DL Training implementation – dense compute platform**

- **Pro:**
 - Specialized hardware
 - good individual performance
- **Con:**
 - Productivity loss
 - Fragmented workflow



COMPUTE

STORE

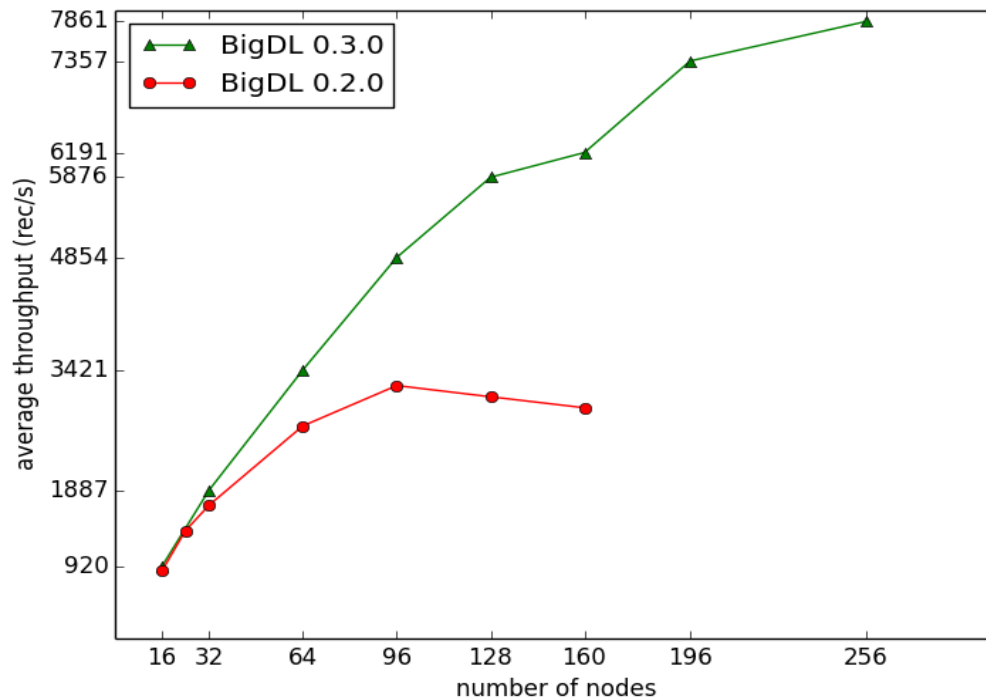
ANALYZE

Intel BigDL – Scalable Deep Learning

- **Distributed Deep Learning Library**
- **Natively integrated with Spark**
 - Single Spark Context
 - Dataset stays in memory
 - Effortless distributed training
- **Optimized with MKL-DNN libraries**
- **Interface similar to Torch**
 - Stacked NN layers
 - Define a very complex model in very few lines
- **Quickly integrate Deep Learning and Machine Learning into Spark-based data analytics workloads**



BigDL Training Scaling



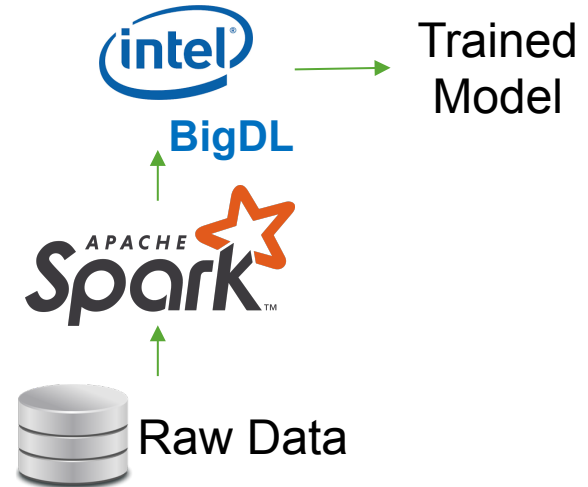
COMPUTE

STORE

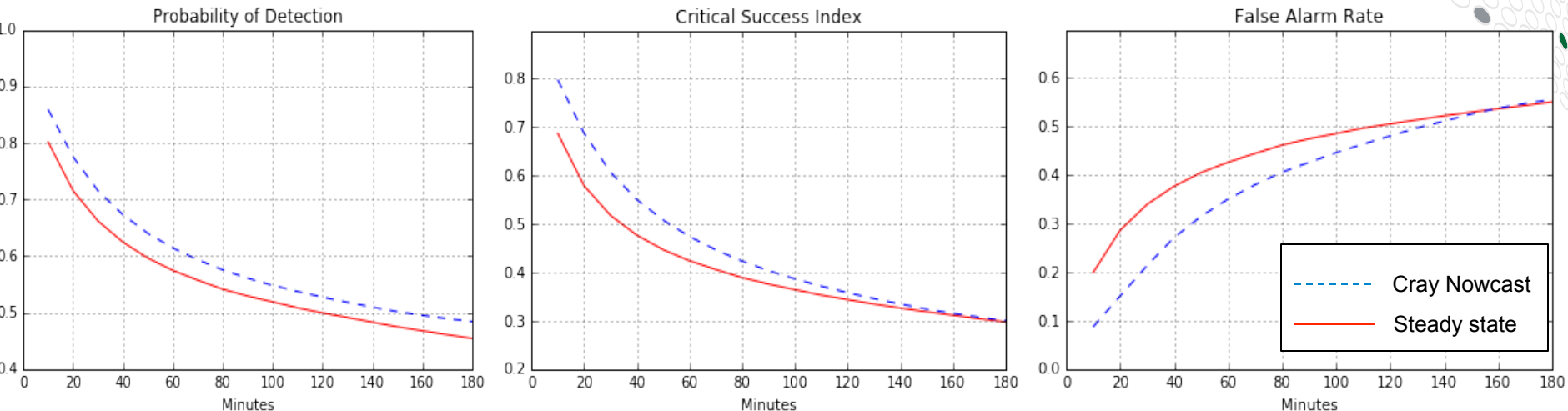
ANALYZE

Implementation: BigDL on Spark

- **Singular workflow**
 - Data processing on spark flows directly into the training process with BigDL
- **HPC scale with Urika-XC**
 - High performance compute nodes excel at data analytics
 - MKL, MKL-DNN provide suitable optimization for DL workloads
 - Suite of analytics tools to aid in development
- **Pros:**
 - Single platform
 - Highly productive development environment
 - Effortless distribution
- **Cons:**
 - Less flexible expressive Deep Learning tools
 - Less flexible compute environment



Results over time vs. steady state



- **False Alarm Rate: Fraction of false alarms to predicted precipitation**
 - $FAR = \text{false-alarms} / (\text{hits} + \text{false-alarms})$
- **Probability of Detection: Fraction of hits to observed precipitation**
 - $POD = \text{hits} / (\text{hits} + \text{misses})$
- **Critical Success Index: Fraction of hits to predicted and observed precipitation**
 - $CSI = \text{hits} / (\text{hits} + \text{misses} + \text{false-alarms})$
 - Penalizes both misses and false alarms

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?