Design High-Performance and Scalable Big Data Systems

Siddhartha Srivastava
srivastava.190@osu.edu

Network-Based Computing for HPC, Big Data, Deep Learning, and Cloud
Department of Computer Science and Engineering
The Ohio State University
Accelerating Spark with RDMA for Big Data Processing: Early Experience

Xiaoyi Lu, Md. Wasi-ur-Rahman, Nusrat Islam,

Dipti Shankar and DK Panda

Department of Computer Science and Engineering
The Ohio State University
Outline

- Introduction
- Problem Statement
- Proposed Design
- Performance Evaluation
- Conclusion
Introduction : Spark

- An in-memory processing framework
  - Iterative machine learning
  - Interactive data analytics
  - Scala based implementation
  - Master Slave, HDFS, Zookeeper
Introduction: Spark

- Scalable and communication intensive
  - Wide dependencies between Resilient Distributed Datasets (RDDs)
  - MapReduce-like shuffle operations to repartition RDDs
  - Same as Hadoop, Sockets based communication
  - Master Slave, HDFS, Zookeeper
Problem Statement

- Is it worth it?
  - Is the performance improvement potential high enough, if we can successfully adapt RDMA to Spark?
  - A few percentage points or orders of magnitude?

- How difficult is it to adapt RDMA to Spark?
  - Can RDMA be adapted to suit the communication needs of Spark?
  - Is it viable to have to rewrite portions of the Spark code with RDMA?

- Can Spark applications benefit from an RDMA-enhanced design?
  - What are the performance benefits that can be achieved by using RDMA for Spark applications on modern HPC clusters?
  - Can RDMA-based design benefit applications transparently?
Proposed Design: Architecture Overview

- **Design Goals**
  - High Performance
  - Keep existing Spark architecture and interface intact
  - Minimum code changes

- **Approaches**
  - Plug-in based approach to extend shuffle framework in Spark
    - RDMA Shuffle Server + RDMA Shuffle fetcher
    - 100 lines of code change
  - RDMA based shuffle engine
SEDA-based Data Shuffle Plug-in

- SEDA - Staged Event Driven Architecture
- A set of stages connected by queues
- A dedicated thread pool will be in charge of processing events on the corresponding queue
  - Listener, Receiver, Handlers, Responders
- Performing admission controls on these event queues
- High throughput through maximally overlapping different processing stages as well as maintain default task-level parallelism in Spark
RDMA based Shuffle Engine

● Connection Management
  ○ Alternative designs
    ■ Pre-connection
      ● Hide the overhead in the initialization stage
      ● Before the actual communication, pre-connect processes to each other
      ● Sacrifice more resources to keep all of these connections alive
    ■ Dynamic Connection Establishment and Sharing
      ● A connection will be established if and only if an actual data exchange is going to take place
      ● A naive dynamic connection design is not optimal, because we need to allocate resources for every data block transfer
      ● Advanced dynamic connection scheme that reduces the number of connection establishments
      ● Spark uses multi-threading approach to support multi-task execution in a single JVM à Good chance to share connections!
      ● How long should the connection be kept alive for possible re-use?
      ● Time out mechanism for connection destroy
RDMA based Shuffle Engine

● Data Transfer
  ○ Each connection is used by multiple tasks (threads) to transfer data concurrently
  ○ Packets over the same communication lane will go to different entities in both server and fetcher sides
  ○ Alternative designs
    ■ Perform sequential transfers of complete blocks over a communication lane -> Keep the order
      ● Cause long wait times for some tasks that are ready to transfer data over the same connection
    ■ Non-blocking and Out-of-order Data Transfer
      ● Chunking data blocks
      ● Non-blocking sending over shared connections
      ● Out-of-order packet communication
      ● Guarantee both performance and ordering
      ● Efficiently work with the dynamic connection management and sharing mechanism
RDMA based Shuffle Engine

- Buffer Management
  - On-JVM-Heap vs. Off-JVM-Heap Buffer Management
  - Off-JVM-Heap
    - High-Performance through native IO
    - Shadow buffers in Java/Scala
    - Registered for RDMA communication
  - Flexibility for upper layer design choices
    - Support connection sharing mechanism à Request ID
    - Support packet processing in order à Sequence number
    - Support non-blocking send à Buffer flag + callback
Performance Evaluation: Setup

- **Hardware**
  - Intel Westmere Cluster (A)
    - Up to 9 nodes
    - Each node has 8 processor cores on 2 Intel Xeon 2.67 GHz quad-core CPUs, 24GB main memory
    - Mellanox QDR HCAs (32Gbps) + 10GigE
  - TACC Stampede Cluster (B)
    - Up to 17 nodes
    - Intel Sandy Bridge (E5-2680) dual octa-core processors, running at 2.7GHz, 32GB main memory
    - Mellanox FDR HCAs (56Gbps)

- **Software**
  - Spark 0.9.1, Scala 2.10.4 and JDK 1.7.0
  - GroupBy Test
Performance Evaluation on Setup A

- For 32 cores, up to 18% over IPoIB (32Gbps) and up to 36% over 10GigE
- For 64 cores, up to 20% over IPoIB (32Gbps) and 34% over 10GigE
Performance Evaluation on Setup B

- For 128 cores, up to 83% over IPoIB (56Gbps)
- For 256 cores, up to 79% over IPoIB (56Gbps)
**Conclusion**

- Three major conclusions
  - RDMA and high-performance interconnects can benefit Spark.
  - Plug-in based approach with SEDA-/RDMA-based designs provides both performance and productivity.
  - Spark applications can benefit from an RDMA enhanced design.
High Performance Design of Apache Spark with RDMA and Its Benefits on Various Workloads

Xiaoyi Lu, Dipti Shankar, Shashank Gugnani and DK Panda

Department of Computer Science and Engineering
The Ohio State University
Outline

- Problem Statement
- Proposed Design
- Performance Evaluation
- Conclusion
Problem Statement

- Previous paper showed that Spark + RDMA = Better Performance

- But something changed in between the previous paper and this paper
Problem Statement

- Yahoo researchers found out that for 46K mappers and 46K reducers, 2 billion files were generated on the clusters.
- Hash-based design
  - Calculated the amount of reducers as the amount of partitions and created a separate file for each partition
  - It then loops through the records, calculates the target partition for each of the records, and stores the record to the corresponding partition file
  - This worked for smaller file system but for larger file system, the amount of file created was too huge.
- This issue was fixed by Spark in 1.2.0 release
- Spark introduced two new shuffle design
  - Sort
  - Tungsten-Sort
Problem Statement

- Can RDMA benefit the new shuffle architecture in Spark?
- Sort-based, Hash-based, and Tungsten-Sort based shuffle: Can RDMA-based design improve all three shuffle schemes?
- What are the performance characteristics of Spark over RDMA in bare-metal and SR-IOV based virtual environment over IB?
- What are the performance benefits we can get for various workloads?
- Can RDMA based design for Spark combine with other techniques easily and transparently to deliver more benefits?
Proposed Design: Architecture Overview

- Design Goals
  - High Performance
  - Keep existing Spark architecture and interface intact
  - Plug in based approach to extend shuffle framework in Spark
  - Spark shuffle related functions are implemented in the shuffle manager
  - The new implementation should be able work with all the shuffle managers

Fig. 2: Architecture of RDMA-based Apache Spark
Shuffle Managers in Spark

- Changes in Spark 1.2.0
  - Spark 1.2.0 changed the default shuffle manager to Sort shuffle manager
  - It outputs a single file ordered by reducer id and indexed
  - Data can be fetched using the index

- Changes in Spark 1.4.0
  - Introduced a new shuffle manager called Tungsten Sort
  - Operated directly on serialized binary data without deserializing it
Implementation

- BlockTransferService was extended to include RDMA Shuffle Server and RDMA Shuffle Client
  - These components implement necessary functions in Java and Scala
  - Use RDMA for data transfer via Java Native Interface (JNI) which bypasses the Java socket layer
  - JNI can offload data transfer tasks in Scala or Java layer to native RDMA based data shuffle engine
  - Extending BlockTransferService does not change the existing Spark architecture
Flow of RDMA based Data Shuffle

- How does ShuffleManager and BlockTransferService work together to complete the data shuffle process?
  - When reduce task is reading records from ShuffleManager, ShuffleManager will initialize a ShuffleReader
  - ShuffleBlockFetcherIterator is an abstraction of ShuffleReader. It can read data locally or remotely from other nodes by sending a request through BlockTransferService
  - If RDMA based service is configured, RDMAShuffleClient will send a request to RDMAShuffleServer
  - RDMAShuffle server will deserialize the message to get block id and use the id to get block data information from BlockManager
  - BlockManager calls ShuffleBlockResolver to get the data
  - RDMABlockStreamManager register an iterator of ManagedBuffers with a communication channel to client

Fig. 3: Flow of RDMA-based data shuffle in Spark
Design of RDMA based Data Shuffle

- Client and Server both follow Staged Event Driven Architecture (SEDA) to achieve high throughput
  - RDMA Shuffle Client has worker thread pool and reader thread pool to handle transmission of block requests, receiving block data, and callback processing for incomplete data
  - RDMA Shuffle Server consists of a reader thread pool that receives data requests from all connections and a handler thread pool for request handling.
  - The number of threads in each pool are configurable.
Design of RDMA based Shuffle Engine

• Connection Management
  ○ Alternative designs
    ■ Pre-connection
      • Hide the overhead in the initialization stage
      • Before the actual communication, pre-connect processes to each other
      • Sacrifice more resources to keep all of these connections alive
    ■ Dynamic Connection Establishment and Sharing
      • A connection will be established if and only if an actual data exchange is going to take place
      • A naive dynamic connection design is not optimal, because we need to allocate resources for every data block transfer
      • Advanced dynamic connection scheme that reduces the number of connection establishments
      • Spark uses multi-threading approach to support multi-task execution in a single JVM à Good chance to share connections!
      • How long should the connection be kept alive for possible re-use?
      • Time out mechanism for connection destroy
Design of RDMA based Data Shuffle

Fig. 4: RDMA-based shuffle engine design
Performance Evaluation

- The RDMA scheme can reduce job execution time by 36% and achieve 52% communication performance improvement. This can be explained as data shuffle read involves a large amount of communication.
Performance Evaluation

- RDMA based design can reduce the execution time of GroupBy test by 33%. Similarly it reduces the execution time of SortBy by 37%.

- RDMA based design can reduce the execution time of GroupBy test by 79% and 74% on 32 and 64 nodes. Similarly it reduces the execution time of SortBy by 77% and 73% on 32 and 64 nodes.

Fig. 6: Performance Evaluation for RDD Benchmarks over SR-IOV enabled Virtualized Cluster on Chameleon

Fig. 7: Performance Evaluation for RDD Benchmarks on SDSC Comet
Performance Evaluation

- RDMA based design can reduces the execution time of Sort test by 36% and 38% on 32 and 64 nodes. Similarly it reduces the execution time of TeraSort by 16% and 23% on 32 and 64 nodes.

- RDMA based design can reduces the execution time of PageRank by 43% and 46% on 32 and 64 nodes. Similarly it reduces the execution time of SparkSQL by 32% and 30% on 16GB and 32 GB data size.
The RDMA based plugin approach for Apache Spark gave very positive results. It can achieve up to 79% performance improvements for Spark RDD operation benchmarks (GroupBy, SortBy). Up to 38% performance improvement for batch workload. Up to 46% performance improvement for graph processing workload. Up to 32% performance improvement for SQL queries.
GraphLab: A New Framework For Parallel Machine Learning

Yucheng Low, Joseph Gonzalez, Aapo Kyrola,

Danny Bickson, Carlos Guestrin and Joseph Hellerstein*

Carnegie Mellon University

*UC Berkeley
Outline

- Motivation
- Input, Output and Components
- Consistency Models
- Implementation
- Case Studies
Motivation

● There is an exponential growth in the scale of Machine Learning and Data Mining (MLDM) algorithms

● Designing, implementing, and testing MLDM at large-scale are challenging due to:
  ○ Synchronization
  ○ Deadlocks
  ○ Scheduling
  ○ Distributed state management
  ○ Fault-tolerance

● The interest on analytics engines that can execute MLDM algorithms automatically and efficiently is increasing
  ○ MapReduce is inefficient with iterative jobs (common in MLDM algorithms)
  ○ Pregel cannot run asynchronous problems (common in MLDM algorithms)
Input, Output and Components

- GraphLab assumes problems modelled as graphs
- It adopts two phases, the initialization phase and execution phase
Components of the GraphLab Engine: The Data Graph

- The graph lab engine has three parts
  1. The data graph, which represents the user program state at a cluster machine
Components of the GraphLab Engine: The Update Function

- The graph lab engine has three parts

2. The update function, which involves two main sub-functions:
   2.1- Altering data within a scope of a vertex
   2.2- Scheduling future update functions at neighboring vertices

The scope of a vertex $v$ (i.e., $S_v$) is the data stored in $v$ and in all $v$’s adjacent edges and vertices.
Components of the GraphLab Engine: The Update Function

- The graph lab engine has three parts

2. The update function, which involves two main sub-functions:

2.1- Altering data within a scope of a vertex

2.2- Scheduling future update functions at neighboring vertices

---

**Algorithm: The GraphLab Execution Engine**

**Input:** Data Graph $G = (V, E, D)$

**Input:** Initial Vertex set $\tau = \{v_1, v_2, \ldots\}$

**while** $\tau$ is not empty do

1. $v \leftarrow \text{RemoveNext}(\tau)$

2. $(\tau', S_v) \leftarrow f(v, S_v)$

3. $\tau \leftarrow \tau \cup \tau'$

**Output:** Modified Data Graph $G' = (V, E', D')$
Components of the GraphLab Engine: The Sync Operation

- The GraphLab engine incorporates three main parts:
  3. The sync operation, which maintains global statistics describing data stored in the data-graph

- Global values maintained by the sync operation can be written by all update functions across the cluster machines
- A mutual exclusion mechanism is applied by the sync operation to avoid write-write conflicts
- For scalability reasons, the sync operation is not enabled by default
Consistency Model in GraphLab
Consistency Model in GraphLab

- GraphLab guarantees sequential consistency
  - Provides the same result as a sequential execution of the computational steps

- User-defined consistency models
  - Full Consistency model is used
  - Vertex Consistency model is used and update function only access local vertex data
  - Edge Consistency model is used and update function does not modify data in adjacent vertices
Consistency Model in GraphLab
Implementation

- GraphLab was implemented in C++ using pThreads.
- The data consistency models were implemented using race free and deadlock free ordered locking protocols.
- To attain maximum performance, parallel memory allocation, concurrent random number generation and cache efficiency are already addressed.
- To achieve these level of performance, ML community would have to repeatedly overcome many of the same challenges needed to build GraphLab.
Case Studies (CO-EM)

- Co-EM is a semi-supervised learning algorithm for named entity recognition.
- Given a list of noun phrases, contexts, and co-occurrence counts for each NP-CT pair in a training corpus, Co-EM tries to estimate the probability that each entity belongs to a particular class.
- The CoEM update function is pretty fast so it stresses GraphLab to manage massive amount of fine-grained parallelism.
- With 16 processors, GraphLab could complete 3 iterations in 30 minutes whereas it took Hadoop around 7.5 hours with 95 cpus.
Case Studies (Gibbs Sampling)

Figure 5: MRF Inference (a) The speedup of the Gibbs sampler using three different schedules. The planned set schedule enables processors to safely execute more than one color simultaneously. The round robin schedule executes updates in a fixed order and relies on the edge consistency model to maintain sequential consistency. The plan set scheduler does not apply optimization and therefore suffers from substantial synchronization overhead. (b) The distribution of vertices over the 20 colors is strongly skewed resulting in a high sequential set schedule. (c) The sampling rate per processor plotted against the number of processor provides measure of parallel overhead which is substantially reduced by the plan optimization in the set scheduler. (d) The speedup for Loopy BP is improved substantially by the Splash. (e) The efficiency of the GraphLab framework as function of the number of processors.

- Running Gibbs Sampling on 16 processors gave a speedup of 15x
Conclusion

- GraphLab is a new parallel abstraction which achieves a high level of usability, expressiveness and performance

- GraphLab supports representation of structured data dependencies, iterative computation and flexible scheduling
Thank You!

Siddhartha Srivastava
srivastava.190@osu.edu