Accelerating Spark Shuffle with RDMA

Liu Bing[1] Liu Fang[2] Xiao Nong[1] Chen Zhiguang[1]

[1]National University of Defense Technology
[2]Sun Yat-Sen University

October 12, 2018

Shuffle Operations in Apache Spark

- When executing an application with Spark, it runs many jobs in parallel.
- These jobs are divided into stages based on the shuffle boundary.

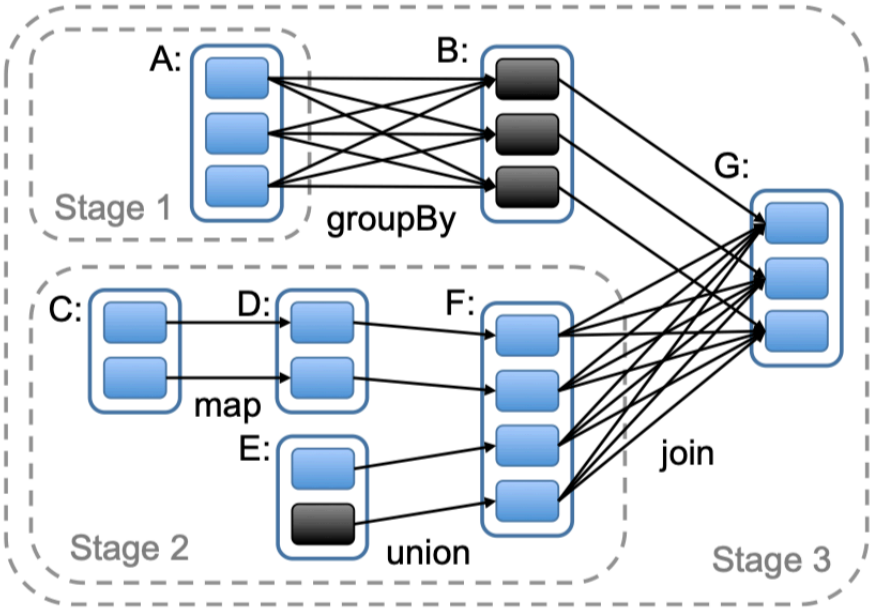

Figure: Example of how Spark computes job stages. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage.[Zaharia2012]

Shuffle Phase is Time-consuming

- Shuffle data across the stage in a cluster is time-consuming
  - The entire working set, which is usually a large fraction of the input data, must be transferred across the network.
  - It requires many remote files and network I/Os.[Davidson2013]

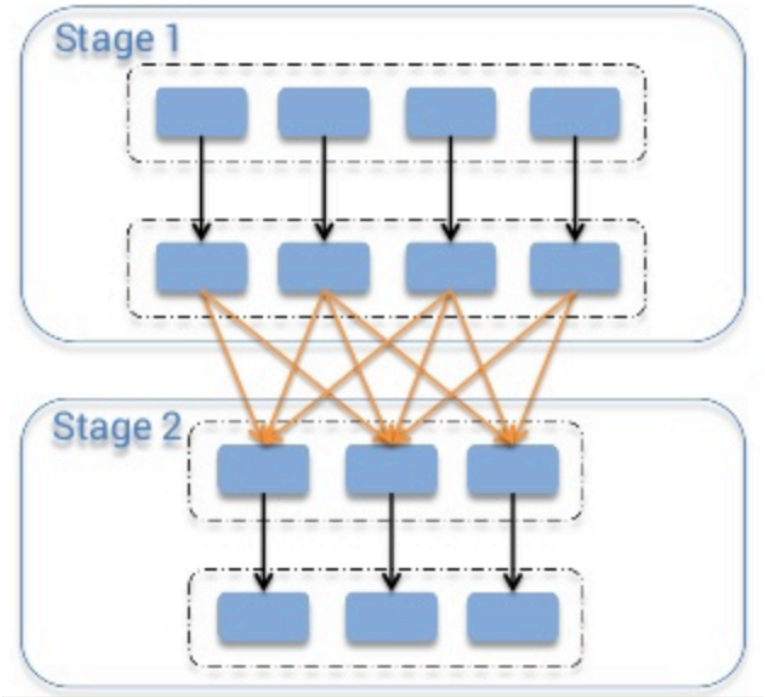

Figure: Shuffle Phase

Netty Involves Many Data Copies

- The latest Apache Spark takes Netty to perform data shuffle.
  - Netty is written with Java Sockets.
  - It needs multiple data copies and CPU involvement.
  - It could not fully take advantage of the performance benefits provided by high-speed interconnects.

What is RDMA?

- RDMA — **R**emote **D**irect **M**emory **A**ccess
- "RDMA" — networking technologies that have a software interface with three features:
  1. Remote DMA
  2. Asynchronous work queues
  3. Kernel bypass
- InfiniBand / RoCE(RDMA over Converged Ethernet) / iWARP, etc.
- RDMA is increasing its presence in datacenters as the hardware becomes cheaper.[Mitchell2013]

Challenges

- how to keep the existing Spark interface intact?
  - adapt the plugin based approach
- how to reduce the overhead of RDMA memory registration?
  - use tiering memory pool

Architecture Overview

- Choosing the plugin based approach to extend current shuffle framework in Spark:
  - Accelerate Spark workloads using RDMA technology
  - Keep the existing Spark architecture and interface intact
  - Can benefit existing Spark applications transparently

Architecture Overview

- Redesign the communication layer of Spark shuffle with RDMA
- A hybrid approach:
  1. using RDMA for data shuffle via Java Native Interface (JNI)
  2. all other Spark operations go through the Netty module

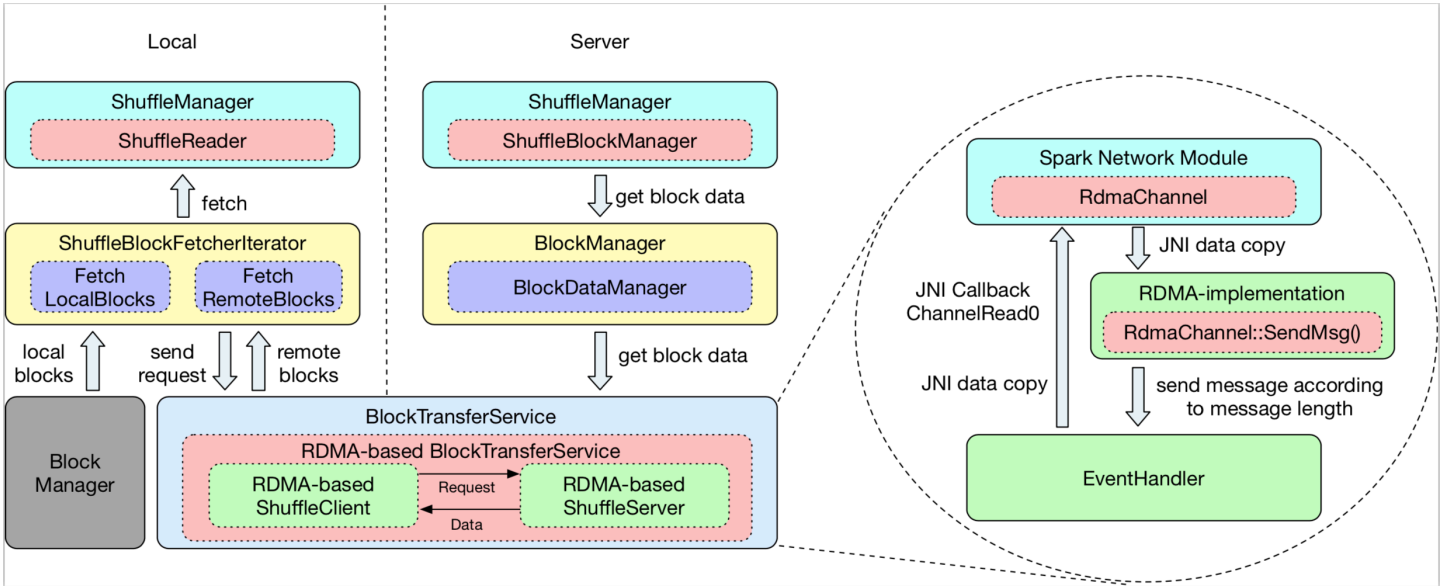

Figure: Flow of RDMA-based data shuffle in Spark.

Major features of our RDMA-based design

1. RDMA Connection Established
2. Tiering Memory Pool: avoid frequent memory allocation
3. Out-of-order Data Transmission: transfer messages of different size using different mechanisms

Tiering Memory Pool

- using RDMA for data transmission requires registering memory to RNIC
- and the operation of registering and deregistering memory includes a significant overhead[Frey2009]
- We use *boost.pool* to implement a tiering memory pool.

Tiering Memory Pool

- registering a large piece of memory in advance, with the chunk's size ranging from 32B, 1KB, 32KB, 1MB and 32MB
- implement our own allocator, including *malloc* and *free*
  - *malloc* — allocate memory and register memory
  - *free* — only called when application finished

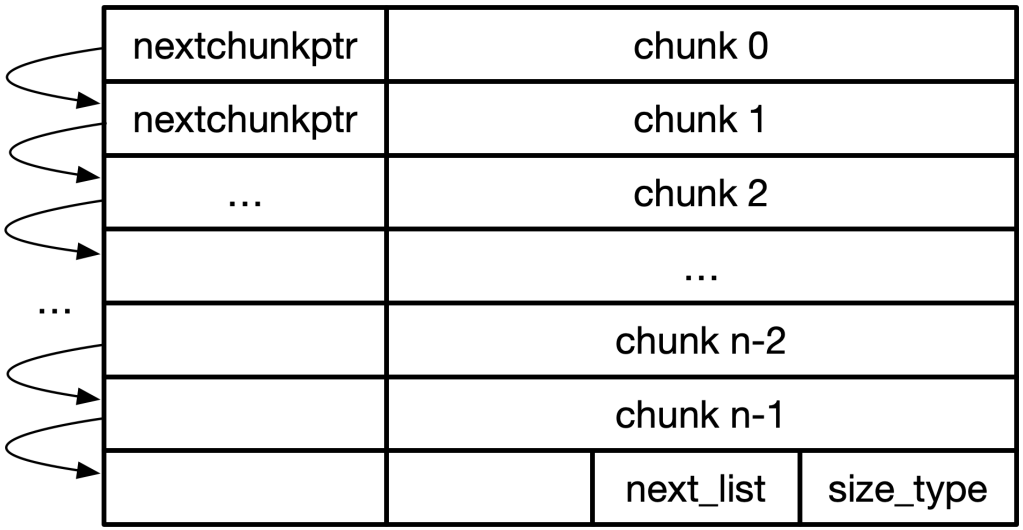

Figure: Block and chunks in boost.pool.

Out-of-order Data Transmission

- use the unique id to ensure data processing correctly
- choose a threshold value to divide messages into small and large
  - in the experiment section, we hard-code 1KB
- for small messages, we just use two-side SEND/RECV
- for large messages, we use one-side READ
- NOW, we are working with WRITE and WRITE_WITH_IMM.

Data Transfer procedure

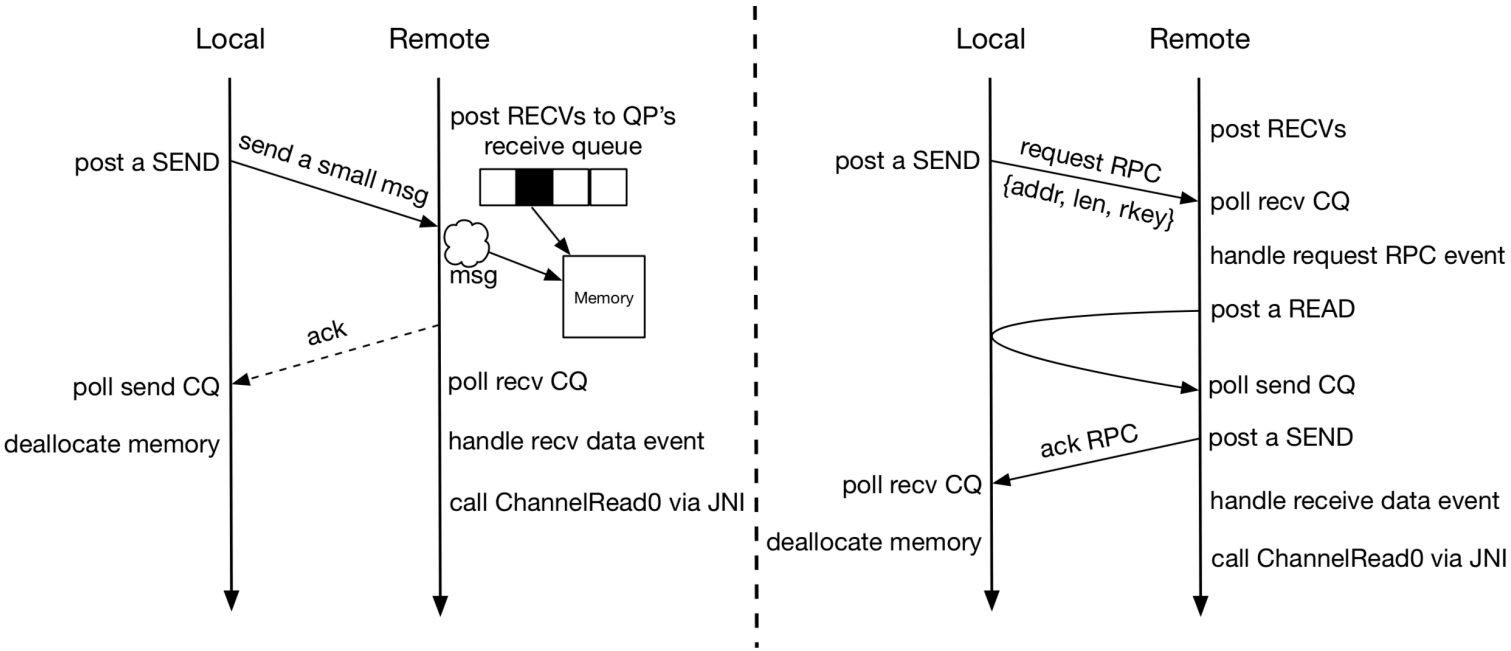

Figure: The data transfer procedure.

Setup

- All of our experiments are conducted on a small cluster of 5 machines. One is master and the other is worker.

CPU	2.10 GHz Intel Xeon E5-2620L 6-core processors
OS	Red Hat Enterprise Linux Server 6 with kernel 2.6.32
Connections	InfiniBand FDR using Mellanox MT27500 ConnectX-3 HCA (40Gb/s) and 1Gb/s Ethernet
Memory	128 GB
Hard Disk	2 TB
Spark Version	1.6.0
Hadoop Version	2.7.3

Table: Environment setup.

Comparison with

1. our RDMA-based Shffule design
2. the default Spark based on Netty over IPoIB
3. Crail-Spark-IO — a recent open-source Spark shuffle plugin from IBM
   - based on DaRPC[stuedi2014]
   - using Spark 2.0.0 to meet its building requirements

Crail-Spark-IO

- shuffle operation becomes as simple as writing and reading files

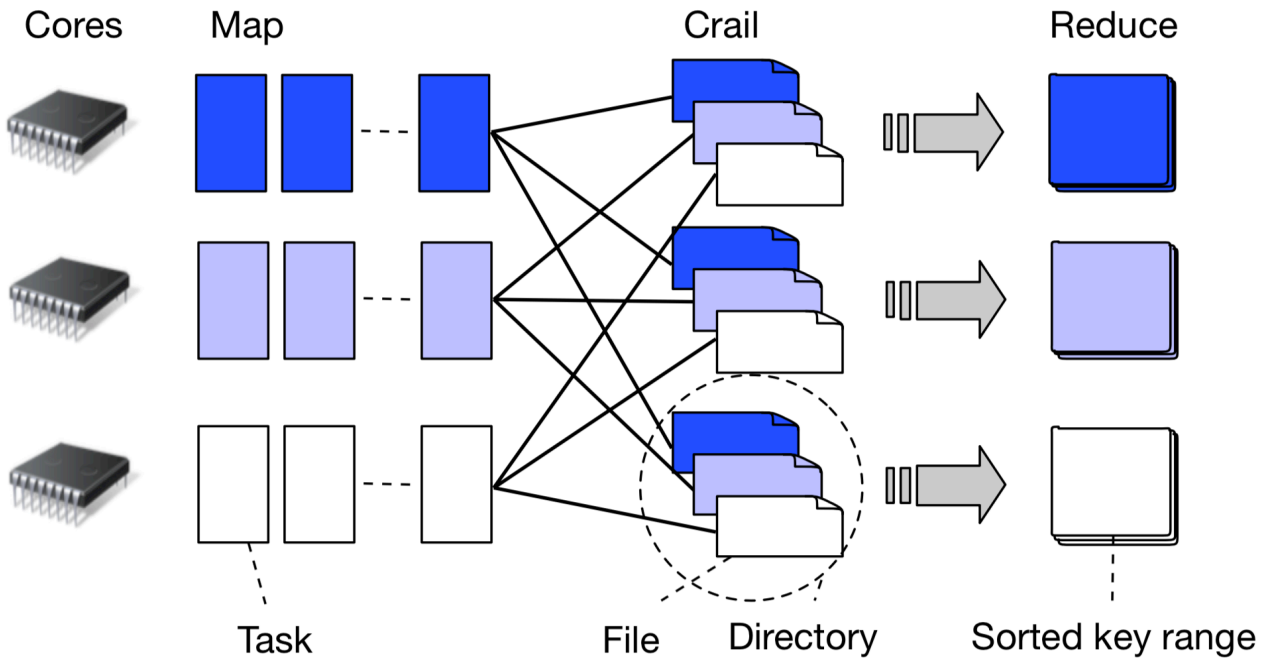

Figure: The Crail shuffle module.[Stuedi2017]

Workloads

- We have chosen four workloads in two categories:
  1. RDD benchmarks (GroupBy and SortBy)
  2. Iterative algorithms (TriangleCount and SVM in SparkBench[Li2015])

Evaluation of Shuffle Read Block Time

- Shuffle Read Block Time — the time that tasks spent blocked waiting for shuffle data to be read from remote machines
- In general, compared with the Netty-based shuffle design, the RDMA-based shuffle design performs better.

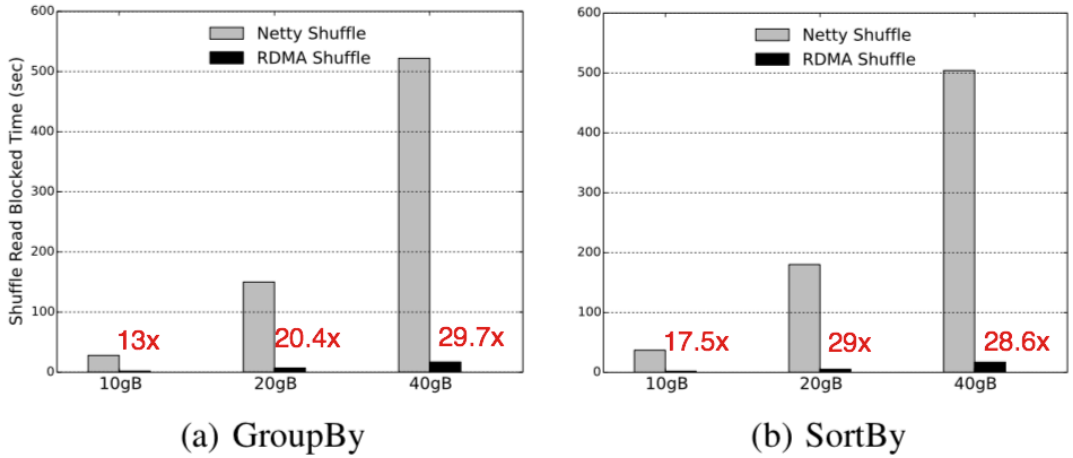

Figure: Shuffle read blocked time evaluation for GroupBy and SortBy.

Performance Evaluation of RDD Benchmarks

- RDMA-based design performs better than the default design
- Compared with Crail-Spark-IO, our implementation can get 25% and 9% improvement for GroupBy and SortBy, respectively.

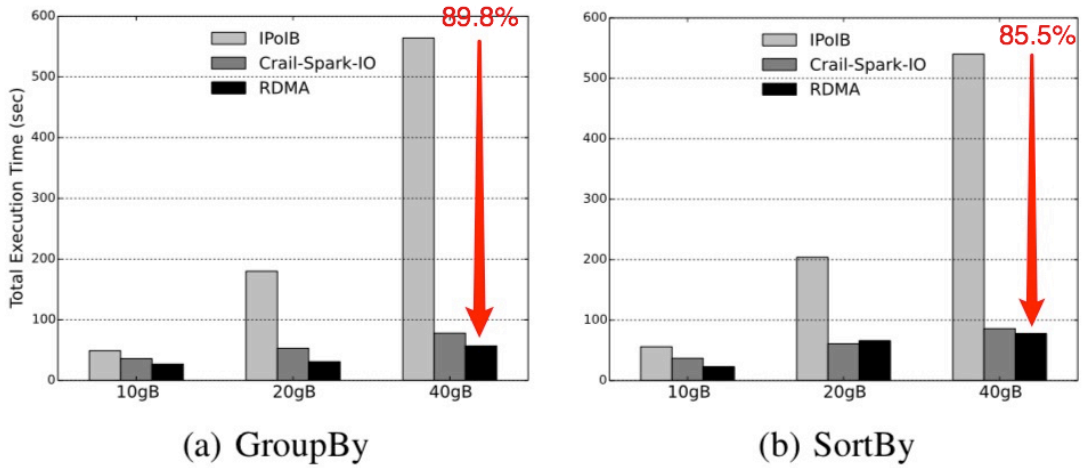

Figure: Performance evaluation for RDD benchmarks.

Performance Evaluation with Iterative Algorithms

- RDMA-based design performs better than the default design.
- Compared with Crail-Spark-IO, when the data size is small, we can get about 41% and 44% improvement for 1gB and 2gB.
- In SVM, when data size is 10gB, the shuffle data is small.

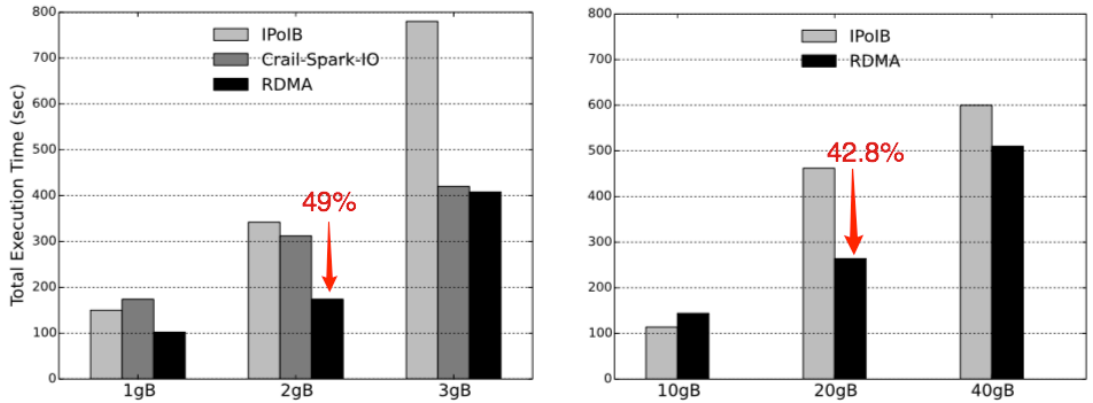

Figure: Performance evaluation for iterative algorithms.

Summary

- Shuffling phase is time-consuming because it requires many remote files and network I/Os.
- RDMA enables zero-copy transfers, reducing latency and CPU overhead.
- A hybrid approach that incorporates communication over conventional sockets as well as RDMA over InfiniBand:
  1. tiering memory pool — avoid frequent memory allocation
  2. SEND/RECV and READ — transfer small and large messages
- Our accelerations to the data transfer layer through this RDMA-based shuffle can benefit Spark workloads transparently.

Thank you!