EC-Shuffle: Dynamic Erasure Coding Optimization for Efficient and Reliable Shuffle in Spark

Xin Yao, Cho-Li Wang, Mingzhe Zhang
Department of Computer Science
The University of Hong Kong
Outline

• Background
  • Basic fault tolerance techniques
  • Erasure coding and FTI
  • Motivation of EC-Shuffle

• Design
  • Two EC-based Fault-tolerance Mechanisms
  • Dynamic Encoding Selection at Runtime

• EC-Shuffle in Spark

• Conclusion

• Q & A
Background

- Sources of data loss
  - Nodes/Processes crash
  - Network partition
  - The tail problem: Timeout
- Fault-tolerance in current systems
  - Storage systems: Storage data/Objects
  - Computation systems: Intermediate data at runtime
    - **Difference**: More communication among data partitions (or blocks) in computation systems
General Fault Tolerance Techniques

- Fault Tolerance Techniques
  1. Lineage: a list of functions
  2. Local checkpoints: non-volatile storage
  3. Replication: Multiple remote copies

Recover a program with n wide transformations when r failures happened

<table>
<thead>
<tr>
<th></th>
<th>Storage overhead</th>
<th>Replay cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lineage</td>
<td>Few (record operations)</td>
<td>All computation of the program</td>
</tr>
<tr>
<td>Local checkpoints</td>
<td>O(n): checkpoint the immediate data of each wide transformation</td>
<td>Only recompute the lost data of each wide transformation, some data partition can be recovered from checkpoints</td>
</tr>
<tr>
<td>Replication</td>
<td>O(r): r copies of the last RDD</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Fault-tolerance Mechanism for Shuffle in Spark

- Distributed computations
- Shuffle runtime
  - Default:
    - Save the checkpoint of the intermediate shuffle data
    - Only recompute the lost data
  - RDDPersist():
    - Remote replication
    - Checkpoints in HDFS

The most expensive operations

(1) Map
(2) Aggregation
(3) Partition
(4) Shuffle
Erasure Coding

- Encoding process
  - Generate parity chunks
- Decoding process
  - Deconstruct original data
- Advantages:
  - Low storage overhead: r/k
  - High throughput (Intel report)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Single Core Throughput of Reed Solomon EC (10+4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Platinum 8180 Processor</td>
<td>12.7 GB/s</td>
</tr>
<tr>
<td>Intel Xeon Processor E5-2650 v4</td>
<td>5.3 GB/s</td>
</tr>
<tr>
<td>Intel Xeon Processor E5-2650 v3</td>
<td>5.3 GB/s</td>
</tr>
<tr>
<td>Intel Xeon Processor D-1541</td>
<td>4.6 GB/s</td>
</tr>
</tbody>
</table>

HDFS supports Erasure Coding in Hadoop 3.x

RDMA+EC: Mellanox BlueField™ and ConnectX®-5

GPUs as Storage System Accelerators
Fault Tolerance Interface

- Runtime of FTI
  - Divides the system in groups of k processes.
  - Generate r parity chunks in each group (k data chunks)
  - The highest reliability level (r=k). In most case, we do not need such a high reliability level (r << k)

- Recovery
  - Concerns: network traffic of collecting data chunks in erasure coding is still heavy
Motivations of EC-Shuffle

- Network traffic becomes serious concern
- Network bandwidth
  - 0.1-1GB/s (inter-machine) vs ≥20GB/s (CPU-RAM)
- Latency
  - 10,000-100,000 ns (inter-machine) vs 100 ns (CPU-RAM)
Motivations of EC-Shuffle

• Two targets of EC-Shuffle:
  1. Two new encoding schemes to reduce heavy network traffic
     • Forward-coding
     • Backward-coding
  2. Reduce the total size of generated parity chunks
     • Encoding scheme selection
A EC-Shuffle with M mappers and N reducers in Spark, the reliability level is r.

**Design -- System Overview**

**Mapper**
- ecShuffleManager (ecShuffleWriter)

**Reducer**
- ecShuffleManager (ecShuffleReader)

**Dependency**
- Get block data
  - Block 1
  - Block 2
  - Block i
  - Block N
- Parity 1
  - ... Parity r

**Network Traffic**
- ecShuffleBlockResolver
- ecShuffleBlockFetcherIterator

**Partition**
- X1
- X2
- Parity

**Aggregation**
- A
- B
- Parity

**Difference between EC-Shuffle and FTI:**

1. FTI gather all data at one node before encoding; in EC-Shuffle, each Mapper/Reducer only perform encoding on its own data partition.

2. FTI has extra data transfer; EC-Shuffle only has parity transfer.

3. EC-Shuffle could generate a smaller size of parity chunks.
Design -- Forward-coding Scheme

- Pre-shuffle (encoding process)
  - For each RDD partition: decide the sub-partitions send to different receivers.
  - Compute the parity chunks on each mapper.
- Shuffle
  - Reducer reads data blocks from mappers
  - Parity chunks are send to different nodes/processes
- Post-shuffle
  - Combine the data chunks from mappers

- Memory overhead: $r (=2)$ parity chunks
- Extra Network traffic: same with memory overhead

Partition (one-to-N operation), $N = 4$, $r = 2$
Design -- Forward-coding Scheme

- **Recovery**
  - If the lost data can be found at the mapper side, it can fetch the data again remotely.
  - Or else fetch data chunks and parity chunks from other nodes to construct the lost data.

- **Network traffic:** one chunk (a sub-partition) or four chunks (a RDD partition at the sender).
Design -- Backward-coding Scheme

- Pre-shuffle
  - For each RDD partition: decide the sub-partitions send to different receivers.

- Shuffle
  - Reducer reads data blocks from mappers

- Post-shuffle (encoding process)
  - Compute the parity chunks on each receiver.
  - Combine the data chunks from mappers
  - Parity chunks are send to different nodes/processes

- Memory overhead: $r (=2)$ parity chunks
- Extra Network traffic: same with memory overhead

Aggregation (M-to-one operation), $M = 4, r = 2$
Design -- Backward-coding Scheme

• Recovery
  • data chunks and parity chunks from other nodes to construct the lost data

• Network traffic: four chunks (a RDD partition at the receiver).

• If a irrecoverable error occurs during the shuffle phase: similar to the replication, RDD will be recovered from the last wide transformation and replay only narrow transformations
Both Forward-coding and Backward-coding can be used in M-to-N wide transform operation (M=4, N=2, r=1)

- **Forward-coding**
  - Network traffic: 8 data chunks
  - Network traffic: 4 parity chunks
  - Parity chunks: 4 parity chunks

- **Backward-coding**
  - Network traffic: 2 data chunks
  - Network traffic: 2 data chunks
  - Parity chunks: 2 data chunks

**FTI**
- Network traffic: 8 data chunks
- Parity chunks: 2 parity chunks
Both Forward-coding and Backward-coding can be used in M-to-N wide transform operation
\((M=2, N=4, r=1)\)

**FTI**

- **FTI**
  - Network traffic: 8 data chunks
  - Parity chunks: 4 parity chunks

**Forward-coding**

- **Forward-coding**
  - Network traffic: 2 parity chunks
  - Parity chunks: 2 parity chunks

**Backward-coding**

- **Backward-coding**
  - Network traffic: 4 parity chunks
  - Parity chunks: 4 parity chunks
Overhead comparison

<table>
<thead>
<tr>
<th></th>
<th>Remote Replication</th>
<th>FTI</th>
<th>FC</th>
<th>BC</th>
<th>EC-Shuffle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Traffic</td>
<td>$r^*M^*X$</td>
<td>$r^*M^*X / (M+r-1)*X$</td>
<td>$r^*X^*M$</td>
<td>$r^*X$</td>
<td>$\text{Min}{r^*X^*M^N, r^*X}$</td>
</tr>
<tr>
<td>Memory Usage</td>
<td>$r^*M^*X$</td>
<td>$r^*X$</td>
<td>$r^*X^*M$</td>
<td>$r^*X$</td>
<td>$\text{Min}{r^*X^*M^N', r^*X}$</td>
</tr>
</tbody>
</table>

- Network traffic:
  - When $M < N$: $\text{RR} \geq \text{FTI} > \text{BC} > \text{FC} = \frac{r^*X^*M}{N}$
  - When $M = N$: $\text{RR} \geq \text{FTI} > \text{BC} = \text{FC} = r^*X$
  - When $M > N$: $\text{RR} \geq \text{FTI} > \text{FC} > \text{BC} = r^*X$

- Memory usage (for storing copies/parities):
  - When $M < N$: $\text{RR} > \text{FTI} = \text{BC} > \text{FC} = \frac{r^*X^*M}{N}$
  - When $M = N$: $\text{RR} > \text{FTI} = \text{BC} = \text{FC} = r^*X$
  - When $M > N$: $\text{RR} > \text{FC} > \text{FTI} = \text{BC} = r^*X$

**Notation**

<table>
<thead>
<tr>
<th>$M$</th>
<th>The number of mappers/senders</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>The number of reducers/receivers</td>
</tr>
<tr>
<td>$X$</td>
<td>The size of data partition in each mapper/sender</td>
</tr>
<tr>
<td>$r$</td>
<td>The reliability level</td>
</tr>
</tbody>
</table>
(1) Large numbers of mappers and reducers in the shuffle:
   • Cause long delay when recover the lost data by decoding many chunks.
   • Divide $N/M$ data chunks into several groups, each group has $k$ data chunks and generates $r$ parity chunks.

(2) Data Chunks in Different Lengths:
   • Append a zero-byte array to each short chunks
   • Light overhead in most applications
Experiments

• Hardware:
  • A spark cluster with 64 physical nodes
  • 1Gb/s Ethernet
• Spark configurations
• Intelligent Storage Acceleration Library
  • Reed-Solomon code
  • ISAL-Java-SDK (implemented via JNI)
• Benchmark: BigDataBench 4.0
  • Network-intensive: Sort, PageRank
  • Computing-intensive: Matrix Multiply

The performance of ISAL-Java-SDK in our cluster

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Ubuntu 18.04</td>
<td>Intel isa-l</td>
<td>2.19</td>
</tr>
<tr>
<td>Kernel</td>
<td>4.15.0-22</td>
<td>Java</td>
<td>1.8.0_191</td>
</tr>
<tr>
<td>GCC</td>
<td>7.3.0</td>
<td>Hadoop</td>
<td>2.7.5</td>
</tr>
<tr>
<td>Maven</td>
<td>3.5.2</td>
<td>Spark</td>
<td>2.3.0</td>
</tr>
<tr>
<td>Yasm</td>
<td>1.3.0.0</td>
<td>BigDataBench</td>
<td>4.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intel Xeon CPU E5540</th>
<th>Encoding speed</th>
<th>Decoding speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 + 1</td>
<td>2.56 GB/s</td>
<td>4.28 GB/s</td>
</tr>
<tr>
<td>10 + 4</td>
<td>1.31 GB/s</td>
<td>1.57 GB/s</td>
</tr>
</tbody>
</table>
Evaluation -- Different Workload Types

- Sort (M=N=96)
  - Data Traffic: 43% reduction
  - Extra memory cost: 84% reduction

- Matrix Multiply (M=N=200)
  - Data Traffic: 45% reduction
  - Extra memory cost: 89.1% reduction

- PageRank (M=4, N=8)
  - Data Traffic: 37% reduction
  - Extra memory cost: 75% reduction

Data Size (GB)

Time (s)

Spark
Spark with Persist
Spark with FTI
Spark with EC-Shuffle
Evaluation -- Different Workload Types

- With EC-Shuffle, the network-intensive workloads has more performance improvement than the computing-intensive workloads.
  - a shuffle has heavy network traffic, but with few computation workloads
  - more shuffle operation
- Load balance is important to determine the total size of parity chunks.
  - Align data chunks, larger parity chunks
- (Limitation) Erasure coding performs not well on encoding data chunks in small size.
  - KMeans and SVMWithSGD

![Graph showing data traffic and extra memory cost for different Spark configurations.](image)
Evaluation -- Recovery

- Application: PageRank for 10 iterations
  - $M = 4$, $N = 8$
- Replication/FTI/EC-Shuffle can instantly recover the runtime data (only one retry stage)
  - FTI needs 4 RDD partition to recover the lost data
**Application:** MatrixMultiply

- \( M = N = 200 \)

**Traffic:**

- FTI: a full RDD + (r-1) parity chunks
- EC-Shuffle: r parity chunks

**Computation**

- With the same speed, encoding big data chunks cause a longer time

The generated parity data size in FTI and EC-Shuffle
Evaluation -- Dynamic Encoding Strategy

The generated parity data size in FTI and EC-Shuffle

<table>
<thead>
<tr>
<th>M, N</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1:1.008:1.000</td>
<td>1:0.542</td>
<td>1:0.290</td>
<td>1:0.294</td>
<td>1:0.298</td>
<td><strong>1:0.199</strong></td>
</tr>
<tr>
<td>4</td>
<td>1:1.006:1.012</td>
<td>1:1.077</td>
<td>1:0.593</td>
<td>1:0.608</td>
<td>1:0.676</td>
<td>1:0.551</td>
</tr>
<tr>
<td>8</td>
<td>1:1.014:1.055</td>
<td>1:1.074</td>
<td>1:1.139</td>
<td>1:1.263</td>
<td>1:1.134</td>
<td>1:1.078</td>
</tr>
</tbody>
</table>

### Findings:
- When \( M < N \): Forward-coding save up to 80%
- When \( M = N \): The same
- When \( M > N \): backward-coding

### Limitations:
- Imbalanced workloads
- the group size \( k \) (10) -> 8 or 16
Limitation

(1) Load imbalance:
   - Combine with other works (try to balance the workloads)
   - Data chunks in different sizes: append zero-valued bytes to data chunks for calculating the parity chunks
   - Multi-Phrase encoding: it avoid allocating zero-valued bytes but it will cause new concern to encode data chunks in small size.

(2) Small-size shuffle block:
   - EC performs bad to encode small-size data
Future Work

(1) Streaming Computing widely use the uncoordinated checkpointing technique, e.g., Flink.
- FTI needs synchronization among $k$ processes to collect their data chunks.
- In EC-Shuffle, each process encode its data chunks **independently**.

(2) Operations in irregular communication pattern:
EC-Shuffle only transfers the parity chunks to provide fault-tolerance.
EC-Shuffle executes coding processes in parallel at multiple mappers/receivers.
EC-Shuffle presents dynamic encoding strategies, which can generate the minimum size of parity chunks. It outperforms FTI by reducing up to 80% of the total size of the parity chunks.
THANK YOU
Q & A