SSS: An Implementation of Key-value Store based MapReduce Framework

Hirotaka Ogawa (AIST, Japan)
Hidemoto Nakada
Ryousei Takano
Tomohiro Kudoh
MapReduce

• A promising programming tool for implementing large-scale data-intensive apps
• Essentially provides a data-parallel computing model
  – Map
    • Spreads a segment of a single array computation over multiple processors
    • Performs each computation on the relevant processor
  – Reduce
    • Aggregates distributed reduction variables
    • Performs computation over them
• (Theoretically) most SPMD-type apps can be realized by MR model
Extends MapReduce to HPC

• User can develop HPC apps faster and easier than ever!
  – Provides a higher level programming model than parallel programming languages, e.g., HPF, OpenMP
  – Provides simpler communication topologies and synchronization model than message-passing libraries, e.g., MPI

• But there are limitations in MapReduce
  – Sacrificing runtime performance
  – Fixed workflow
Why?

- Semantic gap between MR data model and the input/output data format:
  - MR apps handle KV data
  - Backend DFS provides an interface to large data files
- No opportunity to reuse the internal KV data:
  - These KV data exist only in the MR runtime
  - Considerable overhead for reading/writing large amount of data, in particular, iterative apps
- Flexible workflows, e.g., reusing KV data (incl. the int. KV data) across multiple maps and reduces, are infeasible!
Related Work

• Twister
  – Map tasks can read input data from:
    • Distributed in-memory cache
    • Local disks

• MRAP
  – Map tasks can read input data from:
    • Preprocessed files optimized to efficient access
    • Original files

• Partial Solutions
  – Cannot handle the intermediate KV data
  – Users need to determine which data is cacheable (immutable)
Our Solution

MapReduce Runtime
- M M M R R
- Task Manager
- KV KV KV KV KV
- Internal Data (Cache) Manager

Serialized Files
- KV KV KV KV
- File System

Storage-side Cache
- KV KV KV KV

Storage System
- KV KV KV KV

MapReduce Runtime
- M M M R R
- Task Manager

KV KV KV KV KV

File System
- KV KV KV KV

Storage-side Cache
- KV KV KV KV

Storage System
- KV KV KV
SSS: Distributed KVS based MapReduce

- Our first prototype of KVS-based MapReduce
- Hadoop-compatible API
- Distributed KVS-centric
  - Scale-out
    - Horizontally adding new nodes
  - Owner computes
    - Each map and reduce is distributed to the node where the target KV data is stored
  - Shuffle-and-Sort phase is not required
  - On-memory cache
    - Enables reuse of KV data across multiple maps and reduces
  - Flexible workflows are supported
    - Any combination of map and reduce, including iterative apps, are available

Why?
How MapReduce runs in SSS

Map -> Grouping -> Reduce
SSS: Distributed KVS based MapReduce

- Our first prototype of KVS-based MapReduce
- Hadoop-compatible API
- Distributed KVS-based
  - Scale-out
    - Horizontally adding new nodes
  - Owner computes
    - Each map and reduce is distributed to the node where the target KV data is stored
  - Shuffle-and-Sort phase is not required
  - On-memory cache
    - Enables reuse of KV data across multiple maps and reduces
  - Flexible workflows are supported
    - Any combination of map and reduce, including iterative apps, are available

How?
To limit the memory usage and hide the latency of KVS, SSS runtime employs pipelining technique.
Packed-SSS Map Thread Pipeline

- To reduce the number of KV data
  - Stores KV data into thread-local buffer
  - Converts them to a single large KV data
  - Stores it to the KVS
Preliminary Evaluation

• Environment
  – Number of nodes: 16 + 1 (master)
  – CPUs per node: Intel Xeon W5590 3.33GHz x 2
  – Memory per node: 48GB
  – OS: CentOS 5.5 x86_64
  – Storage: Fusion-io ioDrive Duo 320GB
  – NIC: Mellanox ConnectX-II 10G

• MapReduce implementations
  – SSS
  – Packed-SSS
  – Hadoop (replica count is 1, to avoid unintended replications)
Benchmarks

• Word count
  – 128 text files
    • Total file size: 12.5GiB
    • Each file size: almost 100MiB
  – No combiners employed
  – Input: Coarse grain, Output: Fine grain

• Iterative identity map and reduce
  – Both of map and reduce generate a set of KV data same as their inputs
  – Iteratively 8 times applied
  – 8 million keys
    • Total amount of KV data: almost 128MiB
  – Input/Output: Fine grain
DistCopy: Distributing 12.5GiB text files for WordCount

Our KVS is not slower than HDFS
Parallelization is very effective due to ioDrive + 10GbE
WordCount

Running Time [sec]

Trials

1 2 3 4 5

Hadoop
SSS
packed-SSS

12% faster
3x faster
Iterative Identity Map/Reduce

- **2.9x faster**
- **10x faster**

<table>
<thead>
<tr>
<th>Iteration Count</th>
<th>Running Time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hadoop</td>
</tr>
<tr>
<td>2</td>
<td>SSS</td>
</tr>
<tr>
<td>3</td>
<td>packed-SSS</td>
</tr>
<tr>
<td>4</td>
<td>Hadoop</td>
</tr>
<tr>
<td>5</td>
<td>SSS</td>
</tr>
<tr>
<td>6</td>
<td>packed-SSS</td>
</tr>
<tr>
<td>7</td>
<td>Hadoop</td>
</tr>
<tr>
<td>8</td>
<td>SSS</td>
</tr>
</tbody>
</table>

Legend:
- Hadoop
- SSS
- packed-SSS
Numbers of KV data/files

- **WordCount**

<table>
<thead>
<tr>
<th></th>
<th># Map Inputs</th>
<th># Intermediate</th>
<th># Reduce Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSS</td>
<td>128</td>
<td>1.5 billion</td>
<td>2.7 million</td>
</tr>
<tr>
<td>Packed-SSS</td>
<td>128</td>
<td>2,048</td>
<td>16</td>
</tr>
<tr>
<td>Hadoop</td>
<td>128 files</td>
<td>~256 files</td>
<td>16 files</td>
</tr>
</tbody>
</table>

- **Iterative Identity Map/Reduce**

<table>
<thead>
<tr>
<th></th>
<th># Map Inputs</th>
<th># Intermediate</th>
<th># Reduce Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSS</td>
<td>8 million</td>
<td>8 million</td>
<td>8 million</td>
</tr>
<tr>
<td>Packed-SSS</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Hadoop</td>
<td>128 files</td>
<td>128 files</td>
<td>128 files</td>
</tr>
</tbody>
</table>
Conclusion

• SSS is our first prototype of KVS-based MapReduce
• Distributed KVS centric
  – Scale-out
  – Owner computes
  – Shuffle-and-Sort phase is not required
  – On-memory cache
  – Flexible workflows are supported
• Hadoop-compatible API
• Runtime performance is better than Hadoop, but not enough (we think)
Future Work

• Performance
• Fault-tolerance
• More comprehensive benchmarks
  – To identify the characteristics and feasibility to various class of HPC and data-intensive apps
• Higher-level programming tool
  – Pig, Szl, DryadLINQ, HAMA, R, etc.
  – We have already implemented our own Sawzall-clone running on top of Hadoop and SSS
Thank you!
Matrix Multiply by MR

Matrix A (blocked)

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td>1,4</td>
<td></td>
</tr>
</tbody>
</table>

Matrix B (blocked)

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td>1,4</td>
<td></td>
</tr>
</tbody>
</table>

Block multiply

- \( C_{11}, A_{11} \times B_{11} \)
- \( C_{11}, A_{12} \times B_{21} \)
- \( C_{11}, A_{13} \times B_{31} \)
- \( C_{11}, A_{14} \times B_{41} \)

Block add

- \( C_{11}, A_{11} \times B_{11} + A_{12} \times B_{21} + A_{13} \times B_{31} + A_{14} \times B_{41} \)
- \( C_{12}, A_{11} \times B_{12} + A_{12} \times B_{22} + A_{13} \times B_{32} + A_{14} \times B_{42} \)
- \( C_{13}, A_{11} \times B_{13} + A_{12} \times B_{23} + A_{13} \times B_{33} + A_{14} \times B_{43} \)

Block add

- \( C_{11}, A_{11} \times B_{11} + A_{12} \times B_{21} + A_{13} \times B_{31} + A_{14} \times B_{41} \)
- \( C_{12}, A_{11} \times B_{12} + A_{12} \times B_{22} + A_{13} \times B_{32} + A_{14} \times B_{42} \)
- \( C_{13}, A_{11} \times B_{13} + A_{12} \times B_{23} + A_{13} \times B_{33} + A_{14} \times B_{43} \)