Review and Q&A for
MapReduce: Simplified Data Processing on Large Clusters [1]

James Worsham
Spring 2019
CSE 6350
MapReduce: Technical Overview

• “MapReduce is a programming model and an associated implementation for processing and generating large data sets”[1]

• Map functions generate intermediate keys from key/value pairs

• Reduce functions merge intermediate results together

• Example use cases:
  • Word count in large collection of documents
  • Distributed grep across large collection of inputs
  • Distributed sorting
Question 1

Compared with traditional parallel programming models, such as multithreading and MPI, what are major advantages of MapReduce?

Answer

- A major advantage is *automatic parallelization and execution*
  - Abstracts difficulties of distributed execution
    - Input partitioning
    - Distributed scheduling
    - Fault tolerance
    - Load balancing

- Allows developers to build distributed computations without needing experience with the system nuances
  - Business logic is the only concern
Use Figure 1 to explain a MR program’s execution.
Answer 2

1. Split input files into M pieces (typically 16-64 MB)
   Fork many instances of program on cluster
1. Split input files into $M$ pieces (typically 16-64 MB)
   Fork many instances of program on cluster
2. Master assigns map and reduce tasks to workers
1. Split input files into $M$ pieces (typically 16-64 MB) 
   Fork many instances of program on cluster

2. Master assigns map and reduce tasks to workers

3. Map workers read contents of assigned input split 
   Runs each KV pair through *Map* function 
   Intermediate KV outputs buffered in memory

---

[1]
Answer 2

1. Split input files into $M$ pieces (typically 16-64 MB)
   - Fork many instances of program on cluster

2. Master assigns map and reduce tasks to workers

3. Map workers read contents of assigned input split
   - Runs each KV pair through *Map* function
   - Intermediate KV outputs buffered in memory

4. Buffered pairs partitioned into $R$ regions
   - Periodically, partitions written to local disk
   - Master is told of intermediate file locations
   - Master forwards locations to reduce workers

Figure 1: Execution overview
1. Split input files into $M$ pieces (typically 16-64 MB)
   Fork many instances of program on cluster
2. Master assigns map and reduce tasks to workers
3. Map workers read contents of assigned input split
   Runs each KV pair through *Map* function
   Intermediate KV outputs buffered in memory
4. Buffered pairs partitioned into $R$ regions
   Periodically, partitions written to local disk
   Master is told of intermediate file locations
   Master forwards locations to reduce workers
5. Reduce worker remotely reads intermediate file
   Reduce worker sorts (groups) intermediate keys

![Figure 1: Execution overview](image-url)
1. Split input files into M pieces (typically 16-64 MB)  
   Fork many instances of program on cluster
2. Master assigns map and reduce tasks to workers
3. Map workers read contents of assigned input split  
   Runs each KV pair through Map function  
   Intermediate KV outputs buffered in memory
4. Buffered pairs partitioned into R regions  
   Periodically, partitions written to local disk  
   Master is told of intermediate file locations  
   Master forwards locations to reduce workers
5. Reduce workers remotely read intermediate files  
   Reduce workers sort (group) intermediate keys
6. Reduce workers pass each unique key and corresponding values to Reduce function  
   Reduce function output appended to output files

Figure 1: Execution overview
1. Split input files into M pieces (typically 16-64 MB) 
   Fork many instances of program on cluster
2. Master assigns map and reduce tasks to workers
3. Map workers read contents of assigned input split 
   Runs each KV pair through Map function 
   Intermediate KV outputs buffered in memory
4. Buffered pairs partitioned into R regions 
   Periodically, partitions written to local disk 
   Master is told of intermediate file locations 
   Master forwards locations to reduce workers
5. Reduce workers remotely read intermediate files 
   Reduce workers sort (group) intermediate keys
6. Reduce workers pass each unique key and corresponding values to Reduce function 
   Reduce function output appended to output files
7. When all workers are done, master wakes up 
   Control returned to user program
Question 3

Describe how MR handles worker and master failures.

<table>
<thead>
<tr>
<th>Master</th>
<th>Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoints periodically persisted</td>
<td>Master pings all workers and expects responses</td>
</tr>
<tr>
<td>State and ID of workers</td>
<td>If no response received, worker marked as failed</td>
</tr>
<tr>
<td>Locations of intermediate files</td>
<td>If worker fails:</td>
</tr>
<tr>
<td>Size of intermediate files</td>
<td>All in-progress tasks rescheduled</td>
</tr>
<tr>
<td>If master task dies:</td>
<td>All completed <em>map</em> tasks rescheduled</td>
</tr>
<tr>
<td>New copy started from last checkpoint</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When a map task is re-scheduled:</td>
</tr>
<tr>
<td></td>
<td>All reduce workers are notified</td>
</tr>
<tr>
<td></td>
<td>Workers read from new node</td>
</tr>
<tr>
<td></td>
<td>(If data not already loaded from failed worker)</td>
</tr>
</tbody>
</table>
The implementation of MapReduce enforces a barrier between the Map and Reduce phases, i.e., no reducers can proceed until all mappers have completed their assigned workload. For higher efficiency, is it possible for a reducer to start its execution earlier, and why? (clue: think of availability of inputs to reducers)
In the presented Google implementation...

- It is *not* possible for a reducer to start prior to barrier saturation
- The issue is the *shuffling* phase
  - Reducers are remotely collecting and sorting intermediate data
  - Once all the KV pairs are sorted, `reduce()` is ran
  - Data excluded from results if `reduce()` ran before all inputs are ready
Answer 4

*Current MapReduce Implementation
What if we did this without a barrier?

1. Intermediate data from $M_1$ is ready
What if we did this without a barrier?

1. Intermediate data from M₁ is ready
2. R₁ collects and sorts k₁
R₁ runs reduce()
Final output file appended
What if we did this without a barrier?

1. Intermediate data from $M_1$ is ready
2. $R_1$ collects and sorts $k1$
   $R_1$ runs $reduce()$
   Final output file appended
3. Now $M_2$ and $M_3$ are ready
What if we did this without a barrier?

1. Intermediate data from $M_1$ is ready
2. $R_1$ collects and sorts $k1$
   $R_1$ runs $reduce()$
   Final output file appended
3. Now $M_2$ and $M_3$ are ready
4. If we run reduce for $K1$ again, we have conflicting output results
Questions?
References
