Motivation

• MapReduce systems were motivated by the problem of performing data mining and data analytics on large sets of data.
  o The operations are almost always embarrassingly parallel, but are also typically data-intensive not computation-intensive

• MapReduce systems can be either implemented in a cluster, where all the computers are connected in a high speed local area network, or in a grid system where computers across the Internet can perform operations on portion of the data.
Overview

• MapReduce is a programming model and associated implementation

• The name originates from the "Map" and "Reduce" primitive operations in functional programming languages (esp. Lisp, also Haskell, Python, etc.)

• Pioneered by Google; Google's proprietary MapReduce implementation is integrated with other Google inventions: GFS, BigTable, etc.
Applications

Suitable Problem Types:
• Web crawling and/or text processing
• Log analysis
• Machine learning and/or sophisticated data mining
• Archiving large sets of data
• Image processing and manipulation
• Analyzing and processing large sets of messages
• High Performance Computing (for example: Setting up with and using Sun Grid Engine for parallel processing)
Programming Model

The MapReduce paradigm, as apparent in the name, consists of two principal operations: map and reduce.

**Map:** Apply the map function to each input key/value pair to produce a list of intermediate key/value pairs.
- **Input:** collection of key/value pairs
- **Output:** list of intermediate key/value pairs

**Reduce:** Collect all values sharing the same intermediate key and apply the reduce function to yield a single output.
- **Input:** list of intermediate key/value pairs
- **Output:** list of output values
MapReduce Example
Example Walk-Through

- An example problem: Find the maximum temperature of every year during last 100 years with 10 worker nodes using the daily log file of 100 years and sort it by temperature.

- Map Step: Each worker parses raw temperature data into (date, temperature) pairs

- Reduce Step: Years are partitioned between workers and each worker finds the maximum in their set. Tasks are assigned in order of year, which implicitly sorts.
  - Each worker generates an output file with sorted entries
  - Input: (year, temperature) list --> Output: sorted output file of years and max temperatures
Input a large set of data that is accessible by all nodes

Prior to Map Step, the data is split into $M$ partitions and mapped to $P$ processors automatically. Partitioning highly abstracted by default but can be customized

One node is designated as a master; the master will perform partitioning and dynamically assign work units

Map function is applied by map workers to all $M$ partitions; data written to worker's local disks and only the location on local disk relayed to master

Intermediate key/value pairs are split into $R$ partitions for computation; partitioning is performed with a hash function

- Reduce Worker = hash(intermediate key) mod $R$
- Easy attempt to create a uniform distribution of work

Reduce workers use remote procedure calls to collect data from map workers' local disks.

Reduce workers sort intermediate key/value pairs by key and apply reduce function; the output is written to $R$ files
Data Flow Diagram
Benefits of MapReduce

The MapReduce programming model not only provides the ability to process large distributed data sets; the other main advantage is the abstraction layer provided by the architecture

• The programmer is abstracted from load balancing, partitioning, and scheduling. Very little knowledge of parallel programming is required

• However, the architecture must make up for this and has to handle many parallel programming considerations:
  o Fault Tolerance
  o Communication Overhead
  o Grain Size Selection
Fault Tolerance Overview

- Fault tolerance initially appears to be a triviality; however, Google commonly uses clusters with 1000+ nodes
  - Each node may have multiple hard disks/processors
  - Probability of node failure is non-trivial
- Three different fault tolerance scenarios:
  - Worker failures
  - Master failure
  - "Stragglers"
Master & Worker Fault Tolerance

• To detect worker failures, the master periodically pings each worker and checks for an acknowledgement
  o Regular "heartbeat"
• If a worker fails, the task allocated to it is simply rescheduled on a different processing element
• Master failures are more problematic, but also less likely; since there is only one master to many hundred workers.
  o Master writes checkpoints periodically as failsafe
• To be fault tolerant, all work must be completed and all work must be completed correctly (regardless of whether or not it was a re-scheduled execution)
• To ensure correctness all file write operations are atomic
  o Output written to temporary files
  o Renamed to output files as final atomic operation
Handling "Stragglers"

• MapReduce abstraction also accounts for "stragglers", nodes that do not fail but take an unusually long time to complete the assigned work unit.

• Possible Causes:
  o Other tasks may be scheduled on machine
  o Contention on network
  o Degraded components

• When the MapReduce operation is nearly complete, copies of the tasks still in flight are duplicated and scheduled again on different processing elements.

• Increases utilization of computational resources but significantly improved completion speed (result shown later).
Reducing Communication

• For the problem sizes considered, the transfer of the intermediate results between map and reduce workers certainly will strain network

• The Google File System (GFS) provides a foundation to mitigate this
  o GFS divides each file into 64MB blocks which are stored redundantly on different machines (typically 3)

• MapReduce scheduler takes this location information into account and attempts to schedule tasks on or near one of the machines where the data resides

• MapReduce programming model also provides an optional Combiner function
  o Similar to Reduce function, but performed on mapping machine
  o Some data is combined before being sent over the network
  o Can also reduce contention at Reduce nodes if there are many more Map nodes (and improve load balancing)
Grain Size Selection

- The MapReduce model abstracts the selection of grain size from the user, though the problem itself constrains grain size selection
  - User will specify the number of output files $R$; which potentially constrains the number of reduce workers
- Grain size also limited by storage at the master node and demands on the MapReduce scheduler running at the master
  - Master Storage: $O(M+R)$
  - Scheduler States: $O(M*R)$
- Google research publication indicates that for a typical size MapReduce problem:
  - $M = 200,000$ (Map Tasks / # input partitions)
  - $R = 5,000$ (Reduce Tasks / # output files)
  - $P = 2,000$ (Processing Elements)
- Dynamic allocation of smaller tasks also improves fault tolerance; since it is easier to reschedule tasks
Real World Examples

• Google, Yahoo and other search engines use MapReduce to crawl and index the web

• Facebook uses MapReduce to store internal logs and perform data analytics on its users. Its 1,100 machine cluster and 300 machine cluster process 15PB of data.

• Microsoft uses MapReduce for natural language search.

• New York Times used MapReduce in 100 Amazon EC2 instances to convert 4TB of high resolution raw tiff images into 11 million PDFs in 24 hours in an inefficient way, as the data is not local and it cost only $240 for the computation.

• Almost every major tech company uses MapReduce. Details on how many different companies use a popular implementation of MapReduce called hadoop can be found here [http://wiki.apache.org/hadoop/PoweredBy](http://wiki.apache.org/hadoop/PoweredBy)
Implementations

- MapReduce implementations are available in many several high level languages such as Java, Python and C++.

- Map Reduce Implementations:
  - Hadoop
  - Pig
  - Hive
Results at Google

MapReduce Results for Parallel Sort:
• 1 TB of data sorted by 1,800 machines
• Less than 50 lines of user code

(a) Normal execution  (b) No backup tasks  (c) 200 tasks killed
Comparison to Parallel Computing

Table 1: MapReduce Comparison [2]

<table>
<thead>
<tr>
<th></th>
<th>MapReduce</th>
<th>Parallel Computing (ex. MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of Problems</td>
<td>Mainly Data-Intensive</td>
<td>Mainly CPU-Intensive</td>
</tr>
<tr>
<td>Thread / Process Coupling</td>
<td>Minimal</td>
<td>Minimal to Maximal</td>
</tr>
<tr>
<td>Programming Patterns</td>
<td>Just One</td>
<td>Anything</td>
</tr>
<tr>
<td>Programming Effort</td>
<td>Very Small</td>
<td>Medium to Large</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Automatic</td>
<td>Manual</td>
</tr>
</tbody>
</table>
Conclusion

• The MapReduce architecture abstracts the user from many parallel programming decisions
• The price of this abstraction is paid by the architecture, which must make its own efforts to provide fault tolerance, reduce communication overhead, and select an appropriate grain size
• MapReduce systems are used by many commercial entities and open source implementations are readily available
Questions?
References

