MapReduce
Parallel Data Processing

Kai Shen

Big Data Processing

- Fifty billions of indexed web pages
  - 500 terabytes if 10KB per page
- Processing large datasets is time consuming
- Fortunately, many of the big data processing tasks (e.g., counting words) are embarrassingly parallel
  - Distribute the work across a cluster of commodity machines and do that in parallel

Challenges of Big Data Processing

- Parallel programming (what have you done before?)
- Parallel programming is challenging
  - Co-ordination and communication among nodes
  - Load balancing necessary for scalable performance (particularly for large number of nodes)
- Distributed systems are subject to faults
  - Nodes (hardware or software) may fail ⇒ a small per-node failure probability leads to significant chance for something to fail in a large system
  - Inexplicable performance anomaly may arise ⇒ a small anomaly probability on a component leads to significant chance for something abnormal in a large system
- Good engineering is possible, but very hard

What do we need?

- Ideally to have a programming interface and system support such that:
  - the interface is easy to program;
  - the programming interface is suitable for many big data processing applications;
  - the underlying system can automatically support data movement, load balancing, and fault-tolerance behind this interface.
- A realization includes a programming interface and associated system support
- MapReduce is one such realization [Dean and Ghemawat 2004]
  - first introduced at Google for large-scale web data processing
MapReduce Programming Interface

- Two-stage data processing
  - Data can be divided into many chunks
  - A map task processes input data and generates local results for one or a few chunks
  - A reduce task aggregates and merges local results from multiple map tasks
- Data is always represented as a set of key-value pairs
  - Key helps grouping for the reduce tasks
  - Though key is not always needed (for some applications, or for the input data), a consistent data represent eases the programming interface

MapReduce Programming Example

- Count the occurrences of individual words in bunch of web pages
  - Map task: find words in one or a few files
    - Input: <key = page url, value = page content>
      - <"www.golf.com", "golf in florida ...">
      - ...
    - Output: <key = word, value = word count>
      - <"golf", 1>
      - <"florida", 1>
      - ...
  - Reduce task: compute total word counts across multiple files
    - Input/output: <key = word, value = word count>

Dependency in MapReduce

- Dependency/synchronization is honored by the system support
  - Essential for application correctness
  - Too much dependency/synchronization complicates system support; limits performance and scalability

System Support

- Dean’s paper, Figure 1
  - Besides getting all the work done, the underlying system can optimize the performance
    - It assigns work in a load-balanced fashion
    - It may monitor the progress of work on each node to re-balance the work
    - It may adjust work assignment to minimize data movement
System Support

- The underlying system also supports fault tolerance
  - It can monitor potential failures and performance anomaly (not finishing something that should've been finished); re-launch a task if necessary
  - Since most work is done by the master, master failure needs special handing (periodic checkpointing and restart from checkpointed state)
- Performance optimization and fault-tolerance are hidden behind the simple programming interface

Usage Examples

- Web data processing (inside Google)
  - Count words
  - Generate inverted web indexes: set of words, and list of matching web pages for each word
  - Identify popular queries from tons of search logs
  - Data clustering for Google news personalization [Das et al., WWW2007]
  - ... ...

Usage Examples

- Distributed grep
- Distributed sort
- Distributed matrix vector multiplication

\[
\begin{array}{c|c|c|c}
A & B & C \\
\hline
\text{X} & = & \\
\end{array}
\]

Usage Examples

- Big data machine learning
- An example: K-means clustering
  - Each iteration of K-means: given K centers, each sample is grouped into the nearest center
  - Can be done in MapReduce
    - Map: find the nearest center for one or a few samples
    - Reduce: aggregate the results
**MapReduce Implementations**

- MapReduce has multiple implementations
- Original MapReduce implementation at Google
  - Not shared with the public
  - Implemented in C/C++, but support many interfaces including Java
- Hadoop
  - Open source implementation
  - Implemented in Java, only support Java interface?
- Phoenix
  - Also open source
  - MapReduce on a single multi-core machine
    - Does it make sense? MapReduce vs. threads?

**Applications that don’t fit**

- MapReduce supports limited semantics
  - The key success of MapReduce depends on the assumption that the dominant part of data processing can be divided into a large number of independent map tasks
- What applications don’t fit this?
  - Those with complex dependencies --- Gaussian Elimination to solve a linear system of equations; k-means actually wasn’t a great fit; other iterative machine learning computations, ...

**Alternative Parallel Data Processing**

- Good old ways:
  - Distributed applications (synchronized over sockets) to process data on multiple machines
  - Multi-threaded processing to take advantage of multiple CPU cores on a single machine
- Something in between---ideally to have a programming interface and system support such that:
  - the interface is easy to program;
  - the programming interface is suitable for many big data processing applications;
  - the underlying system can automatically support data movement, load balancing, and fault-tolerance behind this interface.