MapReduce
Parallel Data Processing

Kai Shen

Big Data Processing

- http://www.worldwidewebsize.com
- Twenty billions of indexed web pages
  - 200 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 (pthreads and MPI) 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 representation 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
- Dependency/synchronization in MapReduce
  - Map tasks are independent from each other, can all run in parallel
  - A map task must finish before the reduce task that processes its result
  - In many cases, reduce tasks are commutative

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

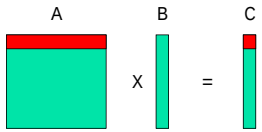
- The underlying system also supports fault tolerance
  - It can monitor potential failures and performance anomaly (not finishing something that should've been finished); re-launch a task if necessary
  - Since most work is done by the master, master failure needs special handling (periodic checkpointing 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
  - … …

Usage Examples

- Distributed grep
- Distributed sort
- Distributed matrix vector multiplication



Usage Example: PageRank

- PageRank scores each page
  - the score is the converged probability for a random web surfer to arrive at the page
- Scores transfer through links (random click by the surfer):
  - If pages B, C are the pages that link to A, then PR(A) = PR(B)/L(B) + PR(C)/L(C)
- Damping factor (for stability):
  - The surfer has a tendency to stay at where he/she is
  - PR(A) = (1-d)/N + d(PR(B)/L(B) + PR(C)/L(C))
- A linear system of equations (N variables, N linear equations)

Usage Example: PageRank

- PR(A) = (1-d)/N + d(PR(B)/L(B) + PR(C)/L(C))
- Iterative solver:
  - Start from an initial PR vector (1/N for each page)
  - Compute a new PR vector by the above linear transformation
  - Keep repeating the linear transformation until the PR converges (very small change after further linear transformation)
    - Even with the large, complex web graph, often converging after dozens of iterations
- Each iteration is a matrix-vector multiplication, so the whole computation is a series of matrix-vector multiplication

Usage Examples in Machine Learning

- 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
- Expectation maximization
  - Data clustering for Google news personalization [Das et al., WWW2007]

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?

Distributed File System

- Big data applications typically run on a distributed file system
  - Too much data that must be spread over multiple nodes
    ⇨ a distributed file system tracks where the data is
  - A single file can be chopped up to blocks for distribution (allowing parallel accesses)
  - Replication for reliability

Google File System

- Google File System (GFS) works on such assumptions
  - A modest number of large files (100s of MBs or GBs or more)
  - Reads are more often than writes
  - "Big data" accesses are common; throughput is more important than latency
  - Build on many cheap commodity components (don't buy exotic hardware, reliability through replication)
- Design/implementation
  - Chunk-based, a single large file contains multiple chunks which are distributed (allowing parallel accesses)
  - One master node and many chunk servers
  - Based on large chunks (64MB per chunk)

Google File System

- Challenges
  - Replication consistency for writes
  - Fault tolerance with a single master

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; graph (social network) processing, …

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 (though a bit more complex than Mapreduce);
  - the programming interface is suitable for more big data processing applications;
  - the underlying system can automatically support data movement, load balancing, and fault-tolerance behind this interface.