Data Analysis with MapReduce

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Motivation: Large Scale Data Processing

• Want to process many terabytes of raw data
  —documents found by a web crawl
  —web request logs

• Produce various kinds of derived data
  —inverted indices
    – e.g. mapping from words to locations in documents
  —representations of graph structure of documents
  —summaries of number of pages crawled per host
  —most frequent queries in a given day
  —...
Problem Characteristics

• Input data is large

• Need to parallelize computation so it takes reasonable time
  —often need thousands of CPUs
What Application Developers Want

- Automatic parallelization & distribution of computation
- Fault tolerance
- Clean and simple programming abstraction
  - parallel programming for the masses …
- Monitoring and status tools
  - monitor computation progress
  - adjust resource provisioning, if necessary
Solution: MapReduce Programming Model

- Inspired by map and reduce primitives in Lisp
  - mapping a function \( f \) over a sequence \( x \ y \ z \) yields \( f(x) f(y) f(z) \)
  - reduce function combines sequence of elements using a binary op

- Many data analysis computations can be expressed as
  - applying a map operation to each logical input record
    - produce a set of intermediate (key, value) pairs
  - applying a reduce to all intermediate pairs with same key

- Simple programming model using an application framework
  - user supplies map and reduce operators
  - messy implementation details handled by the framework
    - parallelization
    - fault tolerance
    - data distribution
    - load balance
Example: Count Word Occurrences

**Pseudo Code**

```java
map(String input_key, String value):
    // input_key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator values):
    // output_key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParsesInt(v);
    Emit(AsString(result));
```

Supports lists of values too large to fit in memory.
Applying the Framework

• Fill in a MapReduce specification object with
  — names of input and output files
  — map and reduce operators
  — optional tuning parameters

• Invoke the MapReduce framework to initiate the computation
  — pass the specification object as an argument
Benefits of the MapReduce Framework

• Functional model
  —simple and powerful interface
  —automatic parallelization and distribution of computations
  —enables re-execution for fault tolerance

• Implementation achieves high performance
What about Data Types?

- Conceptually, map and reduce have associated types
  - map \((k_1, v_1) \rightarrow \text{list}(k_2, v_2)\)
  - reduce\((k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)\)

- Input keys and values
  - drawn from a different domain than output keys and values

- Intermediate keys and values
  - drawn from the same domain as output keys and values

- Google MapReduce C++ implementation
  - passes strings to all user defined functions: simple and uniform
  - have user convert between strings and appropriate types
Example: Count Word Occurrences

Pseudo Code

map(String input_key, String value):

// input_key: document name
// value: document contents

for each word w in value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator values):

// output_key: a word
// values: a list of counts

int result = 0;
for each v in values:
    result += ParseInt(v);
    Emit(AsString(result));
What about Data Types?

• Conceptually, map and reduce have associated types
  —map \((k_1, v_1) \rightarrow \text{list}(k_2, v_2)\)
  —reduce\((k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)\)

• Input keys and values
  —drawn from a different domain than output keys and values

• Intermediate keys and values
  —drawn from the same domain as output keys and values

• Google MapReduce C++ implementation
  —passes strings to all user defined functions: simple and uniform
  —have user convert between strings and appropriate types
Example: Count Word Occurrences

Pseudo Code

map(String input_key, String value):
    // input_key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator values):
    // output_key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);  // Interpret values as integers
    Emit(AsString(result));
MapReduce Examples - I

- Distributed “grep” (pattern search)
  - **map**: emits a line if it matches a supplied pattern
  - **reduce**: identity function - copies intermediate data to the output

- Count of URL access frequency
  - **map**: processes logs of web page requests, outputs a sequence of <URL, 1> tuples
  - **reduce**: adds together all values for the same URL and emits a <URL, total count> pair

- Reverse web-link graph
  - **map**: outputs <target, source> pairs for each link to a target URL found in a page named source
  - **reduce**: concatenates the list of all source URLs associated with a given target URL
    - emits the pair: <target, list of sources> (an adjacency list representation of the graph)
MapReduce Examples - II

• Term-vector per host

  summarize the most important words that occur in a document or a set of documents as a list of <word, frequency> pairs

  —map: emits a <hostname, term vector> pair for each input document
    - hostname is extracted from the URL of the document
  —reduce: passed all per-document term vectors for a given host
    - adds term vectors together
    - throws away infrequent terms, emits a <hostname, term vector> pair

• Inverted Index

  —map: parses each document, emits a sequence of <word, document ID> pairs
  —reduce: accepts all pairs for a given word, sorts the corresponding document IDs, emits a <word, list(document IDs)>
    - set of all output pairs forms a simple inverted index
    - easy to augment this to keep track of word positions
MapReduce Examples - III

• Distributed sort
  — map: extracts key from each record; emits a <key, record> pair
  — reduce: emits all pairs unchanged

  — resulting pairs will be in sorted order
  — this property depends upon partitioning facilities and ordering properties guaranteed by the MapReduce framework
Implementation Considerations

- Many different implementations are possible
- Right choice depends on environment, e.g.
  - small shared memory machine
  - large NUMA multiprocessor
  - larger collection of networked machines
  - dual-processor x86, Linux, 2-4GB memory
  - commodity network: 100Mb/1Gb Ethernet per machine
    - much less than full bisection bandwidth
  - thousands of machines: failure common
  - storage:
    - inexpensive disks attached directly to machines
    - distributed file system to manage data on these disks
      replication provides availability and reliability on unreliable h/w
Execution Overview

1. **Initiation**
   - MapReduce library splits input files into M pieces typically 16-64 MB per piece (controllable by a parameter)
   - starts up copies of the program on a cluster (1 master + workers)

2. **Master assigns** M map tasks and R reduce tasks to workers

3. **Worker assigned a map task** reads contents of input split
   - parses key/value pairs out of input data and passes them to map
   - intermediate key/value pairs produced by map: buffer in memory

4. Periodically write pairs to local disk; partition into R regions; pass locations of buffered pairs to master for reducers

5. **Reduce worker uses** RPC to read intermediate data from remote disks; sort pairs by key

6. **Iterate over sorted intermediate data; call reduce; append output to final output file for this reduce partition**

7. **When all is complete, notify user program**
• For each map and reduce task, store
  —state (idle, in-progress, completed)
  —identity of worker machine (for non-idle tasks)
• For each completed map task
  —store locations and sizes of R intermediate files produced by map
  —information updated as map tasks complete
  —pushed incrementally to workers that have in-progress reduce tasks
Logical Overview of Execution

Data store 1 → map → ... → Data store n

(key 1, values...) (key 2, values...) (key 3, values...) (key 1, values...) (key 2, values...) (key 3, values...)

== Barrier == : Aggregates intermediate values by output key

key 1, intermediate values → reduce → final key 1 values

key 2, intermediate values → reduce → final key 2 values

key 3, intermediate values → reduce → final key 3 values
Logical Execution

Figure credit: http://labs.google.com/papers/mapreduce-osdi04-slides/index-auto-0007.html
Execution Realization

Figure credit: http://labs.google.com/papers/mapreduce-osdi04-slides/index-auto-0008.html
Execution Timeline

- Many more tasks than machines
- Pipeline data movement with map execution

Figure credit: http://labs.google.com/papers/mapreduce-osdi04-slides/index-auto-0009.html
Fault Tolerance: Worker Failure

• Detecting failure
  — master pings worker periodically
  — if no answer after a while, master marks worker as failed

• Coping with failure
  — any map tasks for worker reset to idle state; may be rescheduled
  — worker’s completed map tasks re-execute on failure
    – data on local disk of failed worker is inaccessible
    – any reducers attempting to read notified of the re-execution

Fault tolerance example:
  — network maintenance on a cluster caused groups of 80 machines at a time to become unreachable for several minutes
  — master simply re-executed work for unreachable machines and continued to make forward progress
What about Master Failure?

- Master could periodically write checkpoints of master data structures
- If master dies, another could be recreated from checkpointed copies of its state
- In practice
  — only a single master
  — failure would be rare
  — implementation currently aborts MapReduce if master fails
  — client could check this condition and retry the computation
Exploiting Locality

- Network bandwidth is a scarce commodity

- Data is stored on local disks of machines
  - GFS divides files into 64MB blocks
  - stores several copies (typically 3) on different machines

- MapReduce master
  - attempts to map worker onto a machine that contains a replica of input data
  - if impossible, attempts to map task near a replica
    - on machine attached to same network switch

Locality management example:
- when running large MapReduce operations on a significant fraction of machines in a cluster, most input data is local and consumes no network bandwidth
Task Granularity

• Divide map phase into M pieces; reduce phase into R pieces
• Ideally, M and R much larger than number of worker machines
• Dynamically load balance tasks onto workers
• Upon failure
  —the many map tasks performed by a worker can be distributed among other machines
• How big are M and R?

Task granularity in practice
—e.g., M = 200K, R=5K, using 2000 worker machines
Coping with “Stragglers”

• Problem: a slow machine at the end of the computation could stall the whole computation

• When a MapReduce is nearing completion, schedule redundant “backup” executions of in-progress tasks

Backup task execution in practice

—significantly reduces time for large MapReduce computations
—a sorting example took 44% longer without backup tasks
Combiner

- When there is significant repetition in intermediate keys —e.g. instances of `<the, 1>` in word count output, it is useful to partially merge data locally before sending it across the network to a reducer

- Combiner
  —function executed on each machine that performs a map task
  —typically the same code as the reducer

- Significantly speeds up “certain classes of MapReduce operations”
Input and Output Types

- Can supply a “reader” for new input type
  — e.g. reader interface might read records from a database
- Output types can produce outputs in different formats as well
Refinements

• Partitioning function
  —typically hash(key) mod R
    – tends to give balanced partitions
  —user can supply their own if certain properties desired
    – hash(Hostname(URL)) mod R: all URLs from same host end up in same output file

• Ordering guarantees
  —within a given partition, all intermediate values processed in order: simplifies creating sorted output

• Real world issues
  —skip “bad records”
  —side effects - write extra files “atomically” and idempotently
  —master provides status pages via HTTP
    – enable users to predict run time, decide to add more resources, gain insight into performance issues
MapReduce at Google (2004)

- Large-scale machine learning and graph computations
- Clustering problems for Google News
- Extraction of data to produce popular queries (Zeitgeist)
- Extracting properties of web pages
  - e.g. geographical location for localized search
- Large-scale indexing
  - 2004: indexing crawled documents
    - data size > 20 TB
    - runs indexing as a sequence of 5-10 MapReduce operations
  - experience
    - applications smaller than ad-hoc indexing by 4x
    - readily modifiable because programs are simple
    - performance is good: keep conceptually distinct thing separate rather than forcing them together
    - indexing is easy to operate: e.g. fault tolerant; easy to improve performance by adding new machines
- Hundreds of thousands of MapReduce calculations/day!
MapReduce Examples at Google

- Extracting the set of outgoing links from a collection of HTML documents and aggregating by target document
- Stitching together overlapping satellite images to remove seams and to select high-quality imagery for Google Earth
- Generating a collection of inverted index files using a compression scheme tuned for efficient support of Google search queries
- Processing all road segments in the world and rendering map tile images that display these segments for Google Maps
Performance Anecdotes I

• Cluster
  — ~1800 nodes
    – 2 2GHz Xeon, 4GB memory, 2 160GB IDE drives, 1Gb Ethernet
  — network 2-level tree-shaped switched Ethernet
    – ~100-200Gbps aggregate bandwidth at the root

• Benchmarks executed on a roughly idle cluster

  grep: scan through $10^{10}$ 100-byte records (~1TB) for a relatively rare 3-character pattern
  – split input into 64MB pieces, $M=15000$, $R = 1$ (one output file)
  – time = ~150 seconds
Performance Anecdotes II

sort 10^{10} 100-byte records (~1TB)
- consists of < 50 lines of user code
- split input into 64MB pieces, M=15000, R=4000
- partitioning function uses initial bytes to put it into one of R pieces
- input rate higher than shuffle or output rate: on local disk
- output phase makes 2 replica for availability
- time = 891 seconds

read rate

comm rate

write rate
MapReduce is a Success

• Reasons for its success
  — easy even for users lacking experience with parallel systems
    – insulates user from complexity
      parallelization, fault tolerance, locality opt., load balancing
  — large variety of computations expressible using MapReduce
    – sorting, data mining, machine learning, etc.
  — implementation scales to large commodity clusters
    – makes efficient use of thousands of machines

• Lessons
  — restricting programming model simplifies tackling parallelization, fault tolerance, distribution
  — network bandwidth is a scarce resource
    – locality optimizations to save network bandwidth are important
      read data from local disk; write intermediate data to local disk
  — redundant execution
    – reduces impact of slow machines, machine failures, data loss
#include "mapreduce/mapreduce.h"

class WordCounter : public Mapper {
public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while ((i < n) && isspace(text[i])) i++;
            // Find word end
            int start = i;
            while ((i < n) && !isspace(text[i])) i++;
            if (start < i) Emit(text.substr(start,i-start),"1");
        }
    }
};

REGISTER_MAPPER(WordCounter);
#include "mapreduce/mapreduce.h"

class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the 
    // same key and add the values 
    int64 value = 0;
    while (! input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }
    // Emit sum for input->key() 
    Emit(IntToString(value));
  }
};

REGISTER_REDUCER(Adder);
#include "mapreduce/mapreduce.h"

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);
    MapReduceSpecification spec;

    // Store list of input files into "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }

    // Specify the output files:
    // /gfs/test/freq-00000-of-00100
    // /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");

    // Optional: do partial sums within map
    // tasks to save network bandwidth
    out->set_combiner_class("Adder");

    // Tuning parameters: use at most 2000
    // machines and 100 MB memory per task
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);

    // Now run it
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();

    // Done: 'result' structure contains info
    // about counters, time taken, number of
    // machines used, etc.

    return 0;
}
MapReduce Evolution

• September 2010
  — Google announced that its new search infrastructure “Caffeine” is no longer based on MapReduce
    – MapReduce supported batch indexing scheme
    – Caffeine supports incremental indexing
      use “Bigtable” to represent WWW index
      sparse, distributed, persistent multi-dimensional sorted map
      \( (row:string, column:string, time:int64) \rightarrow string \)
      analyze the WWW in small pieces
      supports incremental updates to index without entire rebuild

• December 2011
  — Open source Apache Hadoop 1.0.0 - http://hadoop.apache.org
    – Hadoop File System
    – Hadoop MapReduce
    – Hadoop Common - support utilities
References - I


- Introduction to Parallel Programming with MapReduce, Google Code University

- Seminar presentation on “MapReduce Theory and Implementation” by Christophe Bisciglia et al. Summer 2007
  —http://code.google.com/edu/submissions/mapreduce/llm3-mapreduce.ppt
References - II

- Hadoop Map/Reduce Tutorial
  —http://hadoop.apache.org/core/docs/r0.19.1/mapred_tutorial.html

  —http://research.google.com/archive/bigtable.html