L22: SC Report, Map Reduce

November 23, 2010

Map Reduce

- What is MapReduce?
- Example computing environment
- How it works
- Fault Tolerance
- Debugging
- Performance
- Google version = Map Reduce; Hadoop = Open source

What is MapReduce?

- Parallel programming model meant for large clusters
  - User implements Map() and Reduce()
- Parallel computing framework
  - Libraries take care of EVERYTHING else
    - Parallelization
    - Fault Tolerance
    - Data Distribution
    - Load Balancing
- Useful model for many practical tasks (large data)

Functional Abstractions Hide Parallelism

- Map and Reduce
- Functions borrowed from functional programming languages (eg. Lisp)
- Map()
  - Process a key/value pair to generate intermediate key/value pairs•
- Reduce()
  - Merge all intermediate values associated with the same key

Example: Counting Words

- Map()
  - Input <filename, file text>
  - Parses file and emits <word, count> pairs
    - eg. <"hello", 1>
- Reduce()
  - Sums values for the same key and emits <word, TotalCount>
    - eg. <"hello", (3 5 2 7)> => <"hello", 17>

Example Use of MapReduce

- Counting words in a large set of documents

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

reduce(string key, iterator values)
    //key: word
    //values: list of counts
    int results = 0;
    for each v in values
        result += ParseInt(v);
    Emit(AsString(result));
```

How MapReduce Works

- User to do list:
  - indicate:
    - Input/output files
    - **M**: number of map tasks
    - **R**: number of reduce tasks
    - **W**: number of machines
  - Write *map* and *reduce* functions
  - Submit the job
- This requires no knowledge of parallel/distributed systems!!!
- What about everything else?

Data Distribution

- Input files are split into **M** pieces on distributed file system
  - Typically ~ 64 MB blocks
- Intermediate files created from *map* tasks are written to local disk
- Output files are written to distributed file system

Assigning Tasks

- Many copies of user program are started
- Tries to utilize data localization by running *map* tasks on machines with data
- One instance becomes the Master
- Master finds idle machines and assigns them tasks

Execution (map)

- *Map* workers read in contents of corresponding input partition
- Perform user-defined *map* computation to create intermediate <key,value> pairs
- Periodically buffered output pairs written to local disk
  - Partitioned into **R** regions by a partitioning function

Partition Function

- Example partition function: hash(key) mod **R**
- **Why do we need this?**
- **Example Scenario:**
  - **Want to do word counting on 10 documents**
  - **5 *map* tasks, 2 *reduce* tasks**

Execution (reduce)

- Reduce workers iterate over ordered intermediate data
  - Each unique key encountered – values are passed to user's reduce function
  - eg. <key, [value1, value2,..., valueN]>
- Output of user's *reduce* function is written to output file on global file system
- When all tasks have completed, master wakes up user program

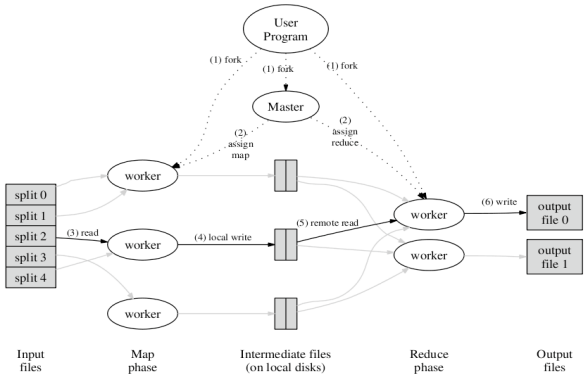
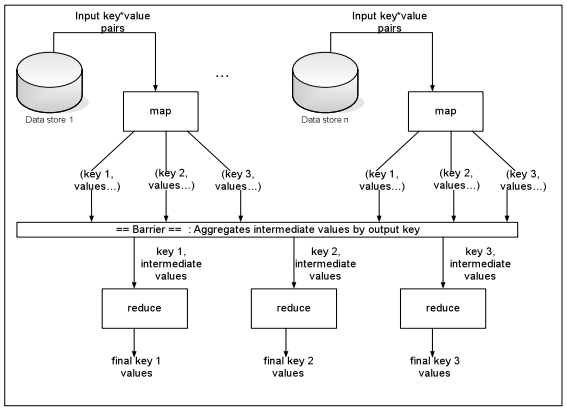


Figure 1: Execution overview

Observations

- No *reduce* can begin until *map* is complete
- Tasks scheduled based on location of data
- If *map* worker fails any time before *reduce* finishes, task must be completely rerun
- Master must communicate locations of intermediate files
- MapReduce library does most of the hard work for us!



Fault Tolerance

- Workers are periodically pinged by master
  - No response = failed worker
- Master writes periodic checkpoints
- On errors, workers send "last gasp" UDP packet to master
  - Detect records that cause deterministic crashes and skips them

Fault Tolerance

- Input file blocks stored on multiple machines
- When computation almost done, reschedule in-progress tasks
  - Avoids "stragglers"

Debugging

- Offers human readable status info on http server
  - Users can see jobs completed, in-progress, processing rates, etc.
- Sequential implementation
  - Executed sequentially on a single machine
  - Allows use of gdb and other debugging tools

MapReduce Conclusions

- Simplifies large-scale computations that fit this model
- Allows user to focus on the problem without worrying about details
- Computer architecture not very important
  - Portable model

References

- Jeffery Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters
- Josh Carter, http://multipart-mixed.com/software/mapreduce_presentation.pdf
- Ralf Lammel, Google's MapReduce Programming Model – Revisited
- http://code.google.com/edu/parallel/mapreduce-tutorial.html

- RELATED
  - Sawzall
  - Pig
  - Hadoop