

MapReduce: Simplified Data Processing on Large Clusters

- Framework developed by Jeff Dean and Sanjay Ghemawat, both working at Google
- Developed in early 2004
- Model very successful in Google – Thousands of programs have been written in this restricted programming model

Motivation

- Many computations involve processing of huge amount of data (tera-bytes of data)
- Such raw data can be
 - Crawled documents
 - Web request logs
- Computations essentially perform similar tasks
- Processing is distributed across 1000s of machines
- Simple Interface needed by developers to run programs in parallel

MapReduce: An Abstraction

- MapReduce: An abstraction that allows developers to express distributed computations easily
- Hides details about parallelization, fault-tolerance, data distribution, load-balancing etc.
- Abstraction inspired by two *Lisp* primitives, *map()* and *reduce()*

map() and *reduce()*

- *map* in Lisp:
 - Input is a function f and a sequence of values $\langle v_1, v_2, v_3, \dots, v_n \rangle$
 - *map* applies the function to each value to compute $\langle f(v_1), f(v_2), f(v_3), \dots, f(v_n) \rangle$
- *reduce* in Lisp:
 - Input is a sequence $\langle v_1, v_2, \dots, v_n \rangle$ and an operation
 - combines all elements using the operation
 - ex. can use '+' to add all elements $(v_1 + v_2 + \dots + v_n)$

Characteristics of Programs

- Characteristics of Programs expressible in this model:
 - Usually scan huge amount of data
 - Derive some intermediate values by doing a first-scan
 - Use intermediates to calculate final values
- Examples of programs expressible in MapReduce model:
 - Distributed Grep
 - URL Access Frequency count

Programming model

- Input & Output: Each a set of key/value pairs
- Programmer specifies two functions:
 - `map (input_key, input_value) -> list(output_key, intermediate_value)`
 - `reduce (output_key, list(intermediate_value)) -> list(output_value)`
- A library was built to achieve this functionality

Example: Word Count

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

Example Contd.

```
reduce(String output_key, Iterator intermediate_values):  
    // output_key: a word  
    // output_values: a list of counts  
{  
    int result = 0;  
    for each v in intermediate_values:  
        result += ParseInt(v);  
    Emit(AsString(result));  
}
```

Computing Environment at Google

- x86 Dual-processor machines running Linux with 2-4 GB RAM
- No. of machines in cluster were typically in thousands
- 100 MB/sec network bandwidth
- Data stored on cheap IDE disks
- Scheduler maps user jobs to a set of available machines

How MapReduce Works

- The input file split into blocks of 16MB to 64MB
- Many copies of program started on cluster
- One copy is Master and the rest are Workers
- M map tasks and R reduce tasks
- Master picks idle workers and assigns work
- Map function parses input & passes (key, value) pairs to *map()* function
- Intermediate data buffered to local disks, partitioned into R regions and location conveyed to master
- Upon location becoming available, reduce worker makes RPC calls, reading buffered data and grouping together values for the same key
- Reduce worker passes (unique_key, <v1, v2, v3.....>) to *reduce()* function
- Output of *reduce()* appended to final output file for this reduce partition

Execution Overview

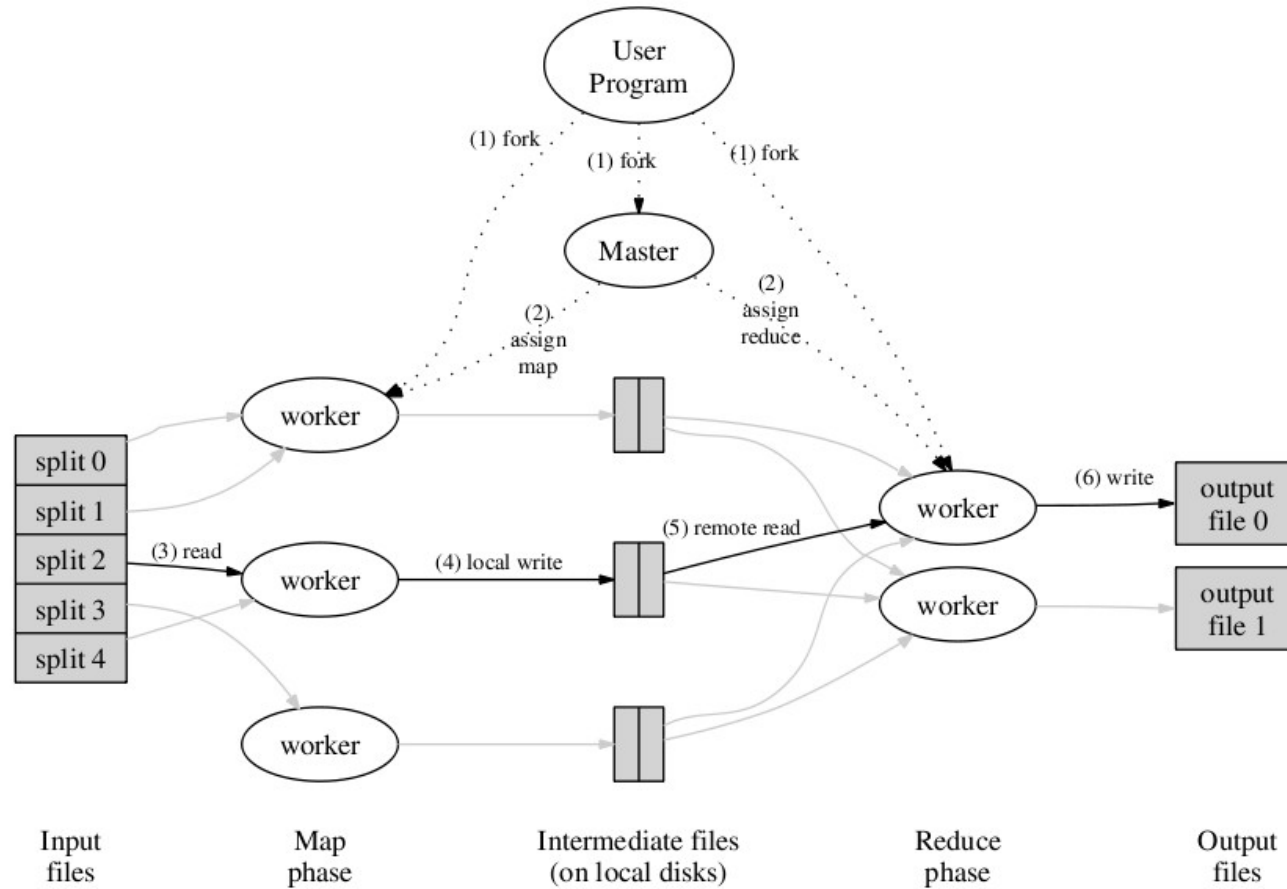


Figure 1: Execution overview

Fault tolerance

- When Worker fails:
 - Idea is to re-execute the work done by failed worker
 - All incomplete *map* or *reduce* tasks reset to initial idle state
 - Finished *map* tasks also reset to initial idle state
 - On re-scheduling of map tasks, workers executing *reduce* tasks notified of the change
- When Master fails:
 - MapReduce computation aborted

Refinements

- 'Stragglers' – Machines slowing down at the end of MapReduce operation
- Towards the end, multiple executions of the same in-progress task are invoked
- Significant reduction in time to complete MapReduce operation

Refinements

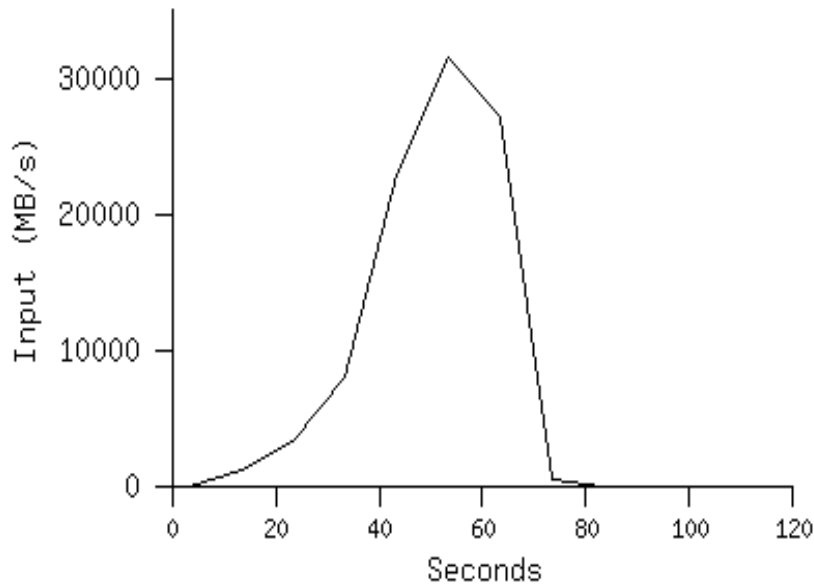
- Input data divided into 64 MB blocks
- Each block replicated on several machines
- Most data read locally, so no network bandwidth
- Master schedules map task on a worker which -
 - has the replica of input data
 - or is near to a machine having the input data

Performance Evaluation

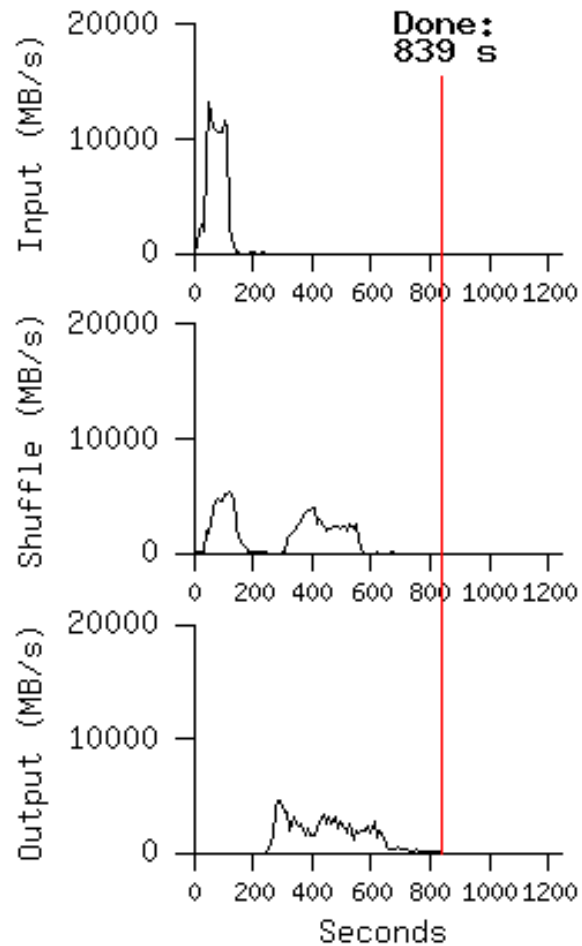
- System tested on
 - 1800 machines
 - Each having two 2GHz processors, 4GB RAM, two 160 GB IDE disks and a gigabit Ethernet link
 - Tested at a time when low workload
- Two programs were run – Grep and Sort

Grep

- Scans 10^{10} 100-byte records searching for a 3-character pattern
- $M = 15,000$ and $R = 1$
- Rate increases as more machines are added
- Rate drops as the map tasks finish



Sort



- Sorts 10^{10} 100-byte records
- $M = 15,000$ & $R = 4000$
- Input rate $>$ Shuffle rate $>$ Output rate
- Input rate for sort $<$ Input rate for grep

Conclusion

- MapReduce provides an interface to execute large computations on a cluster of machines in reasonably small time
- Simple to use
- Programmers need not worry about implementation details such as parallelization, fault tolerance
- Many real world tasks are expressible in this model

References

Jeffery Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters: Paper and Slides

<http://code.google.com/edu/parallel/mapreduce-tutorial.html>