Google’s MapReduce

Simplified Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat

Presented by Laurie James
Summary

What?

– General-purpose library for large-scale distributed data processing;
– Fault-tolerant;
– Hides implementation details from programmers.

Why?

– Google processes vast quantities of data...
  • And has large clusters of machines.
– Writing elegant code for distributed processing is tricky.
Writing MapReduce code

The programmer defines two functions:

- map(k1, v1) -> list(k2,v2)
  * Takes input as a key/value pair, applies the function code
  * Returns a list of ‘intermediate’ k/v pairs.
- reduce(k2,(list v2)) -> list(v2)
  * Iterates over the list of values, applying the reduce function as necessary.

MapReduce groups all equal intermediate keys to be passed into reduce
Code example: word frequency

```java
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: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
        Emit(AsString(result));
```
Implementations

• Many different implementations to suit different architectures;

• They describe the process for Google’s cluster:
  – 100s-1000s of networked machines;
  – Locally networked - Gigabit ethernet;
  – Distributed filesystem (GFS)

• Not entirely applicable to other designs – refined by trial & error
Execution model

• MapReduce library picks a `master node’.
• And splits input into $M$ map tasks, and $R$ reduce tasks.
  – $M,R$ user defined.
    • Optimally, $M$ splits input into ~16-64MB tasks;
    • $R$ a small multiple of number of machines;
    • $O(M*R)$ memory usage on the master.

• Input files are then distributed across the cluster...
• And MapReduce tasks are spawned on each node.
Execution model (cont’d)

• All nodes initially idle;

• The master assigns idle workers a map or a reduce task.

• If a worker receives a map, it:
  – Parses out k/v pairs, runs these through the map function;
  – Buffers and periodically writes intermediate k/v pairs;
  – Location of intermediate output sent to the master.

• If a worker receives a reduce, it:
  – Gets the intermediate data location from the master;
  – Pulls this over the network;
  – Sorts and iterates over values, applying reduce function;
  – Writes the end result to one of R final output files.
So far, so theoretical...

Above process is good, but we don’t live in a perfect world.

Machine failures:
- Are pretty likely in large clusters!
- Workers are periodically pinged;
  - If they timeout, the task is reallocated.
  - (Even if the worker is a completed map task – local data!)

Great, but what if the master dies?
- They assume it doesn’t!
- Only one machine, so failure is unlikely.
- But possible to write configuration stores as ‘checkpoints’.
- MapReduce operation fails
Stragglers - she just won’t run any faster!

‘Stragglers’ are a significant problem in large clusters.
   – Could be due to poor hardware or slow IO
   – A few slow machines significantly increase completion time.

So start ‘backup’ tasks for remaining processes when nearly done.
   – Little (~4%) overhead, large performance increase
Refinements

• Network bandwidth is scarce
  – Split the input data multiple times across many nodes
  – Master tries to assign maps on nodes with a local copy of the relevant data;
  – Failing that, a node where it’s close.

• Reduce tasks are split with a `partitioning function’
  – Default: \((\text{hash(key)} \mod R)\)
  – But users can specify their own
    • E.g. \((\text{hash(hostname(url/key))} \mod R)\)
    • To group all data from the same hostname into an output file
Another refinement...

`Combiner’ functions useful where we have many of the same intermediate k/v. E.g. (the, 1).

- Combiner performs a local reduce prior to writing the intermediate keys.
- Allegedly significantly increases performance.
  - By writing less intermediate k/v pairs, so less I/O?
Bugs & Debugging

• Deterministic bugs repeatedly crashing an operation;
  – MapReduce will never complete.
  – If an op crashes twice, the master skips that record.

• Can also run MapReduce locally (no distributed debugging).

• Master runs an internal webserver.
  – Provides auxiliary information:
    • x/y tasks completed
    • Bytes in/out
    • # failed nodes/operations
    • Among others...
Performance

• Benchmarked with a cluster:
  – ~1800 machines;
  – 2x2Ghz CPUs;
  – ~3GB available memory;
  – Gigabit ethernet.

• Two benchmarking procedures:
  – Grep for a 3-char string in 1TB data;
  – Sort 1TB data (‘Terrasort’).

• Tasks representative of normal MapReduce usage:
  – Extract infrequent data from large dataset;
  – Parse/reorganise large collection of data.
Distributed Grep

- Total time of ~140s
- Of which 60s is startup overhead...
- Slow ‘warm-up’ while adding more machines.
- 30GB/s peak on 1734 workers.

Figure 2: Data transfer rate over time
1TB Sort (50LoC(!))

- Takes \(~890\)s. (40s startup)
- Best prior time – 1057s.
- Throughput is < half that of Grep
  - Because sorting requires heavy I/O of intermediates.
The trouble with stragglers...

- Same, but with backup tasks disabled.
- Vast majority of work done by ~800s (as we’d expect...)
- But the last 5 tasks take an extra 300s to finish.
- Total of 1283s – 44% increase.
Murder.

• Same task again, but killing off 200 workers.
• New tasks allocated, takes a total of 933s.
• Only 5% time increase.
Conclusions/findings

• Particularly useful in some domains:
  – Distributed grep;
  – Counting URL hits from server logs;
  – Term-vectors per host;
  – Distributed sort;
• Makes life easier for Google engineers.
• Code consolidation – one function 3800->700 LoC.
• Increases worker efficiency.
• Conserving bandwidth is important.
• Library is well liked/used.
Comments/criticisms

• Lots of unnecessary explanation of their own environment/clusters.
• Little in-depth discussion of using the library.
  – But perhaps more suited to a technical manual...

• No real comparison of benchmarks against existing solutions!
  – Not impressive if previous benchmark was done on 6 P2s!
Thank you!

Questions...