MapReduce
Simplified Data Processing on Large Clusters

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

What do we do when there is too much data to process?

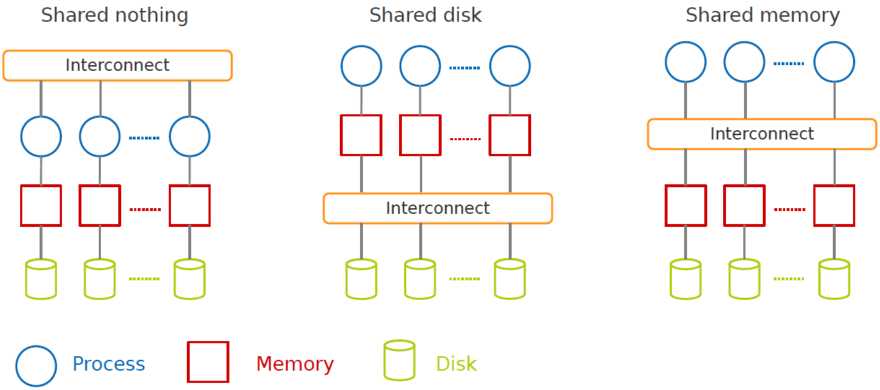
Scale Up vs. Scale Out (1/2)

- Scale up or scale vertically: adding resources to a single node in a system.
- Scale out or scale horizontally: adding more nodes to a system.

Scale Up vs. Scale Out (2/2)

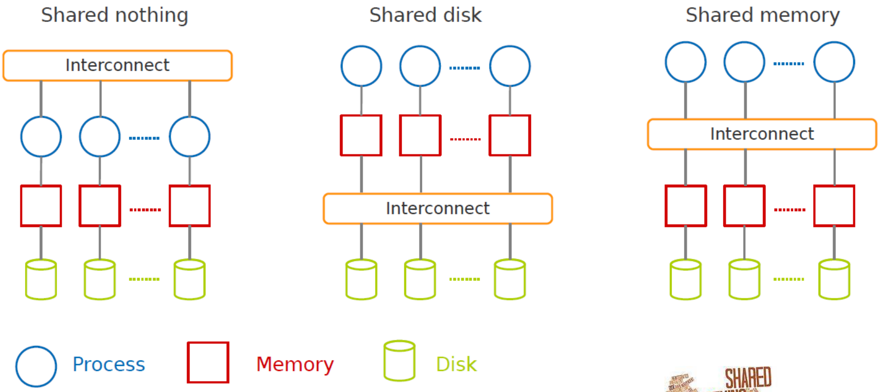
- Scale up: more expensive than scaling out.
- Scale out: more challenging for fault tolerance and software development.

Taxonomy of Parallel Architectures



DeWitt, D. and Gray, J. "Parallel database systems: the future of high performance database systems". ACM Communications, 35(6), 85-98, 1992.

Taxonomy of Parallel Architectures



DeWitt, D. and Gray, J. "Parallel database systems: the future of high performance database systems". ACM Communications, 35(6), 85-98, 1992.

MapReduce

- A shared nothing architecture for processing large data sets with a parallel/distributed algorithm on clusters of commodity hardware.

Challenges

- How to distribute computation?
- How can we make it easy to write distributed programs?
- Machines failure.

Idea

- Issue:
  - Copying data over a network takes time.

Idea

- Issue:
  - Copying data over a network takes time.

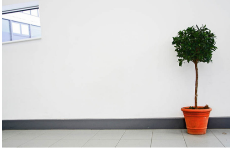
- Idea:
  - Bring computation close to the data.
  - Store files multiple times for reliability.

Simplicity

- Don't worry about parallelization, fault tolerance, data distribution, and load balancing (MapReduce takes care of these).

- Hide system-level details from programmers.

Simplicity!



MapReduce Definition

- A programming model: to batch process large data sets (inspired by functional programming).

MapReduce Definition

- A programming model: to batch process large data sets (inspired by functional programming).

- An execution framework: to run parallel algorithms on clusters of commodity hardware.

Programming Model

Warm-up Task (1/2)

- We have a huge text document.
- Count the number of times each distinct word appears in the file
- Application: analyze web server logs to find popular URLs.

Warm-up Task (2/2)

- File too large for memory, but all $\langle \texttt{word}, \texttt{count} \rangle$ pairs fit in memory.

Warm-up Task (2/2)

- File too large for memory, but all $\langle \texttt{word}, \texttt{count} \rangle$ pairs fit in memory.
- `words(doc.txt) | sort | uniq -c`
  - where `words` takes a file and outputs the words in it, one per a line

Warm-up Task (2/2)

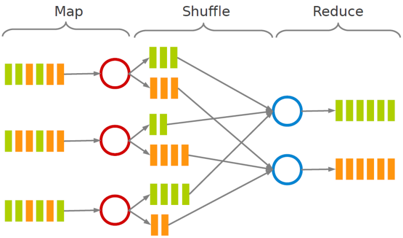
- File too large for memory, but all $\langle \texttt{word}, \texttt{count} \rangle$ pairs fit in memory.
- `words(doc.txt) | sort | uniq -c`
  - where `words` takes a file and outputs the words in it, one per a line
- It captures the essence of MapReduce: great thing is that it is naturally parallelizable.

MapReduce Overview

- `words(doc.txt) | sort | uniq -c`

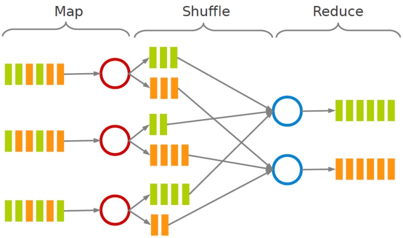
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.



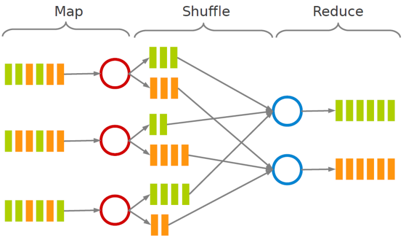
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.



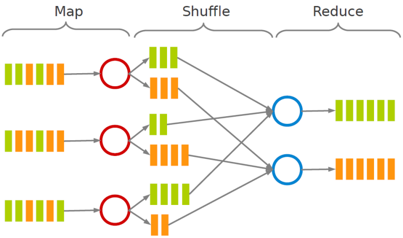
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.



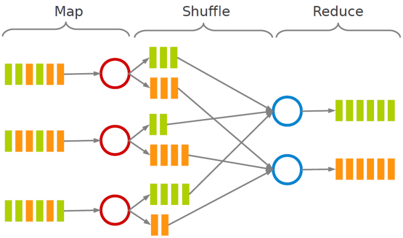
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.



MapReduce Overview

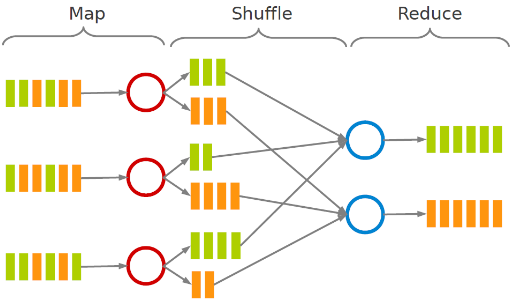
- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.
- Write the result.



MapReduce Dataflow

- map function: processes data and generates a set of intermediate key/value pairs.

- reduce function: merges all intermediate values associated with the same intermediate key.



Example: Word Count

- Consider doing a word count of the following file using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

Example: Word Count - map

- The map function reads in words one a time and outputs (word, 1) for each parsed input word.
- The map function output is:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Example: Word Count - shuffle

- The shuffle phase between map and reduce phase creates a list of values associated with each key.
- The reduce function input is:

```
(Bye, (1))
(Goodbye, (1))
(Hadoop, (1, 1)
(Hello, (1, 1))
(World, (1, 1))
```

Example: Word Count - reduce

- The reduce function sums the numbers in the list for each key and outputs (word, count) pairs.

- The output of the reduce function is the output of the MapReduce job:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

Combiner Function (1/2)

- In some cases, there is significant repetition in the intermediate keys produced by each map task, and the reduce function is commutative and associative.

Machine 1:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
```

Machine 2:

```
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Combiner Function (2/2)

- Users can specify an optional combiner function to merge partially data before it is sent over the network to the reduce function.
- Typically the same code is used to implement both the combiner and the reduce function.

Machine 1:

```
(Hello, 1)
(World, 2)
(Bye, 1)
```

Machine 2:

```
(Hello, 1)
(Hadoop, 2)
(Goodbye, 1)
```

Example: Word Count - map

```
public static class MyMap extends Mapper<...> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);

    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
  }
}
```

Example: Word Count - reduce

```
public static class MyReduce extends Reducer<...> {
  public void reduce(Text key, Iterator<...> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;

    while (values.hasNext())
      sum += values.next().get();

    context.write(key, new IntWritable(sum));
  }
}
```

Example: Word Count - driver

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = new Job(conf, "wordcount");

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setMapperClass(MyMap.class);
  job.setCombinerClass(MyReduce.class);
  job.setReducerClass(MyReduce.class);

  job.setInputFormatClass(TextInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);

  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.waitForCompletion(true);
}
```

Example: Word Count - Compile and Run (1/2)

```
# start hdfs
> hadoop-daemon.sh start namenode
> hadoop-daemon.sh start datanode

# make the input folder in hdfs
> hdfs dfs -mkdir -p input

# copy input files from local filesystem into hdfs
> hdfs dfs -put file0 input/file0
> hdfs dfs -put file1 input/file1

> hdfs dfs -ls input/
input/file0
input/file1

> hdfs dfs -cat input/file0
Hello World Bye World

> hdfs dfs -cat input/file1
Hello Hadoop Goodbye Hadoop
```

Example: Word Count - Compile and Run (2/2)

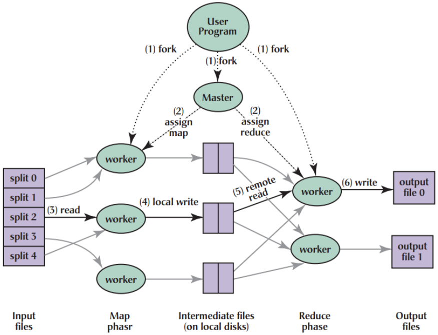
```
> mkdir wordcount_classes

> javac -classpath
$HADOOP_HOME/share/hadoop/common/hadoop-common-2.2.0.jar:
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.2.0.jar:
$HADOOP_HOME/share/hadoop/common/lib/commons-cli-1.2.jar
-d wordcount_classes sics/WordCount.java

> jar -cvf wordcount.jar -C wordcount_classes/ .

> hadoop jar wordcount.jar sics.WordCount input output

> hdfs dfs -ls output
output/part-00000

> hdfs dfs -cat output/part-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

Execution Engine

MapReduce Execution (1/7)

- The user program divides the input files into M splits.
  - A typical size of a split is the size of a HDFS block (64 MB).
  - Converts them to key/value pairs.

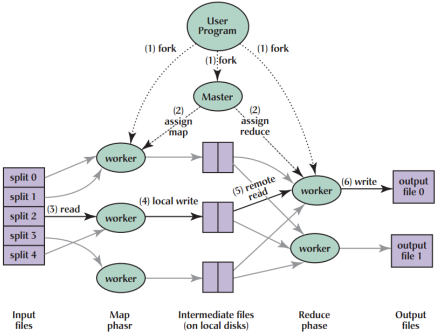
- It starts up many copies of the program on a cluster of machines.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

MapReduce Execution (2/7)

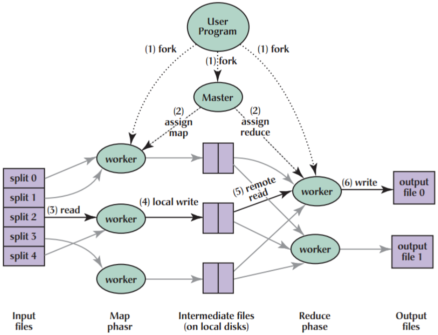
- One of the copies of the program is master, and the rest are workers.
- The master assigns works to the workers.
  - It picks idle workers and assigns each one a map task or a reduce task.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

MapReduce Execution (3/7)

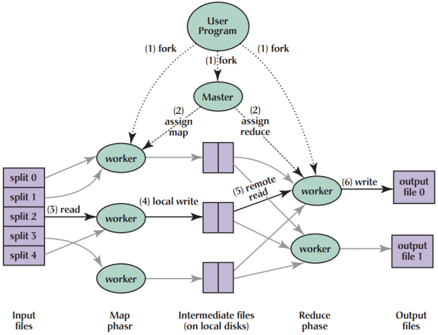
- A map worker reads the contents of the corresponding input splits.
- It parses key/value pairs out of the input data and passes each pair to the user defined map function.
- The intermediate key/value pairs produced by the map function are buffered in memory.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

MapReduce Execution (4/7)

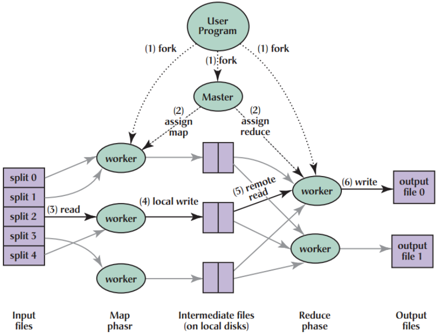
- The buffered pairs are periodically written to local disk.
  - They are partitioned into R regions (hash(key) mod R).
- The locations of the buffered pairs on the local disk are passed back to the master.
- The master forwards these locations to the reduce workers.



J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", ACM Communications 51(1), 2008.

MapReduce Execution (5/7)

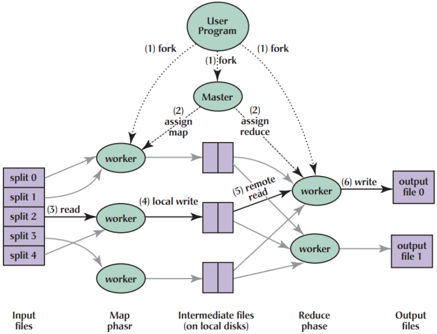
- A reduce worker reads the buffered data from the local disks of the map workers.
- When a reduce worker has read all intermediate data, it sorts it by the intermediate keys.



J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters”, ACM Communications 51(1), 2008.

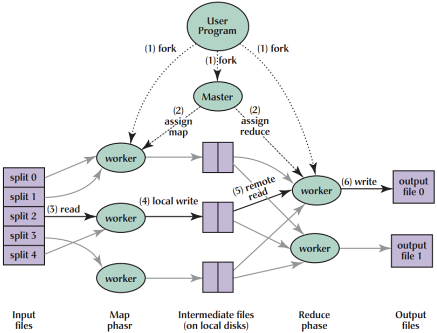
MapReduce Execution (6/7)

- The reduce worker iterates over the intermediate data.
- For each unique intermediate key, it passes the key and the corresponding set of intermediate values to the user defined reduce function.
- The output of the reduce function is appended to a final output file for this reduce partition.



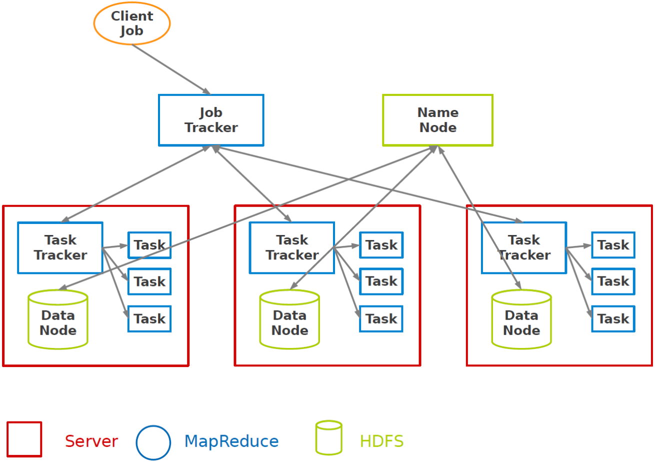
MapReduce Execution (7/7)

- When all map tasks and reduce tasks have been completed, the master wakes up the user program.



J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters”, ACM Communications 51(1), 2008.

Hadoop MapReduce and HDFS



Fault Tolerance - Worker

- Detect failure via periodic heartbeats.
- Re-execute in-progress map and reduce tasks.
- Re-execute completed map tasks: their output is stored on the local disk of the failed machine and is therefore inaccessible.
- Completed reduce tasks do not need to be re-executed since their output is stored in a global filesystem.

Fault Tolerance - Master

- State is periodically checkpointed: a new copy of master starts from the last checkpoint state.

MapReduce Weaknesses and Solving Techniques

W1: Access to Input Data

- Scanning the entire input to perform the map-side processing.
- Initiating map tasks on all input partitions.
  - Accessing only a subset of input data would be enough in certain cases.
- Lack of selective access to data.
- High communication cost.

S1: Access to Input Data

- Efficient access to data.
- Indexing data: Hadoop++, HAIL
- Intentional data placement: CoHadoop
- Data layout: Llama, Cheetah, RCFile, CIF

W2: Redundant Processing and Recomputation

- Performing similar processing by different jobs over the same data.
  - Jobs are processed independently: redundant processing

- No way to reuse the results produced by previous jobs.
  - Future jobs may require those results: recompute everything

S2: Redundant Processing and Recomputation

- Batch processing of jobs: MRShare
- Result sharing and materialization: ReStore
- Incremental processing: Incoop

W3: Lack of Early Termination

- Map tasks must process the entire input data before any reduce task can start processing.
- Some jobs may need only sampling of data.
- Quick retrieval of approximate results.

S3: Lack of Early Termination

- Sampling: EARL
- Sorting: RanKloud

W4: Lack of Iteration

- MapReduce programmers need to write a sequence of MapReduce jobs and coordinate their execution, in order to implement an iterative processing.
- Data should be reloaded and reprocessed in each iteration.

S4: Lack of Iteration

- Looping, caching, pipelining: Stratosphere, Haloop, MapReduce online, NOVA, Twister, CBP, Pregel, PrIter
- Incremental processing: Stratosphere, REX, Differential dataflow

W5: Lack of Interactive and Real-Time Processing

- Various overheads to guarantee fault-tolerance that negatively impact the performance.
- Many applications require fast response times, interactive analysis, and online analytics.

S5: Lack of Interactive and Real-Time Processing

- Streaming, pipelining: Dremel, Impala, Hyracks, Tenzing
- In-memory processing: PowerDrill, Spark/Shark, M3R
- Pre-computation: BlikDB

Summary

- Programming model: Map and Reduce
- Execution framework
- Batch processing
- Shared nothing architecture

Questions?