MapReduce and (Apache) Spark

Lecture 19
May 6, 2025

To Dos

Program #7 results

Reading for next time

Final project assigned

Main Points

- MapReduce
    - API
    - System organization
    - Google File System
- Limitations of MapReduce
- (Apache) Spark
    - API
- Hadoop vs. Spark

Different Programming Environments

- MPI
  - Have to explicitly orchestrate data communication
- Pthreads / OpenMP
  - Allows sharing of data
  - Limited number of threads
- GPUs
  - Read-only data sharing except within thread blocks
  - Computation oriented, limited by memory bandwidth

Motivation for MapReduce

- Data parallel computations on large, mostly read-only data sets distributed over large number of machines

Motivation for MapReduce

- Data parallel computations on large, mostly read-only data sets distributed over large number of machines
- Considerations
  - How do you use commodity resources that fail frequently?
  - How do you parallelize the work?
  - How do you deal with distribution of work?
  - How do you load balance work to achieve lower latency?

Solution: MapReduce

- Programmer describes the work in a data parallel fashion
  - Map
    - input key/value pair → set of intermediate key/value pairs
  - Reduce
    - set of all intermediate key/value pairs with same key → key/value pair
- Submit job to scheduling system
- Underlying system handles all the other issues:
  - Distributing the work
  - Distributing the data
  - Dealing with hardware failures
  - Load balance
  - Locality

map from Functional Programming Languages

From Wikipedia: map is the name of a higher-order function that applies a given function to each element of a functor, e.g. a list, returning a list of results in the same order.

- Takes function F
- Takes a list [a1, a2,...,an]
- Produces [F(a1), F(a2), ...F(an)]

```
map (fn x=>x+1, [1,2,3,4,5]); (* "map" sucessor func to list *)
val it = [2,3,4,5,6] : int list
```

Example 1: Counting Strings

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

Example 2: Inverted Index

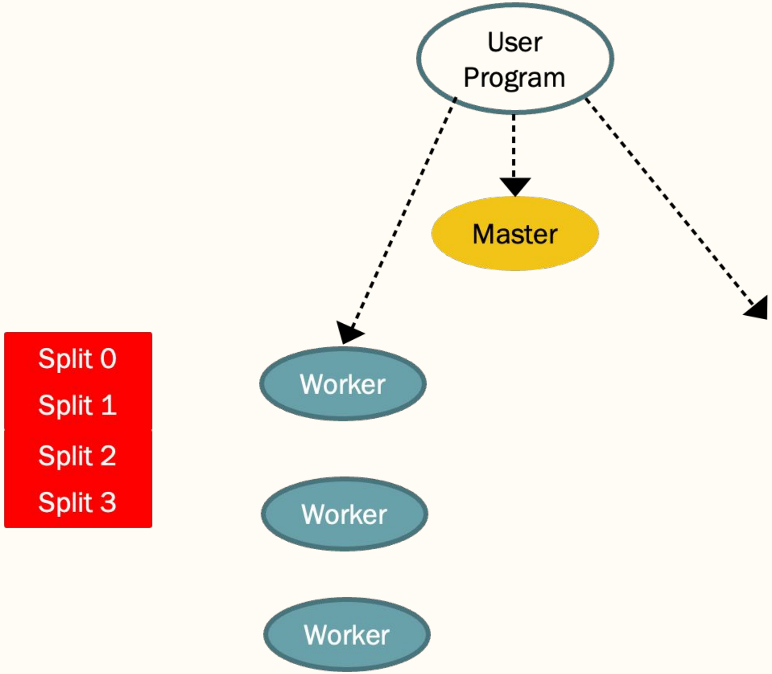
- Map (Document name, file contents)
  - Emits sequence of <word, document ID> pairs
- Reduce
  - Emits <word, list(document ID)> pairs

Execution of MapReduce

- Partition input into set of M splits
  - Splits processed in parallel
  - Each split typically 16-64 MB

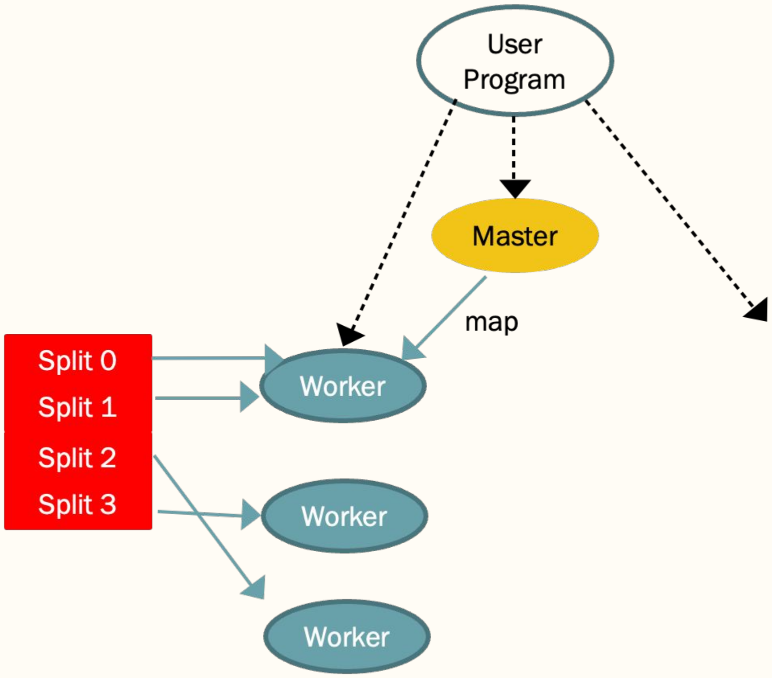
Execution of MapReduce

- Partition input into set of M splits
  - Splits processed in parallel
  - Each split typically 16-64 MB
- Start many copies of program on cluster
  - Master task
  - M map tasks
  - R reduce tasks
  - M+R >> workers



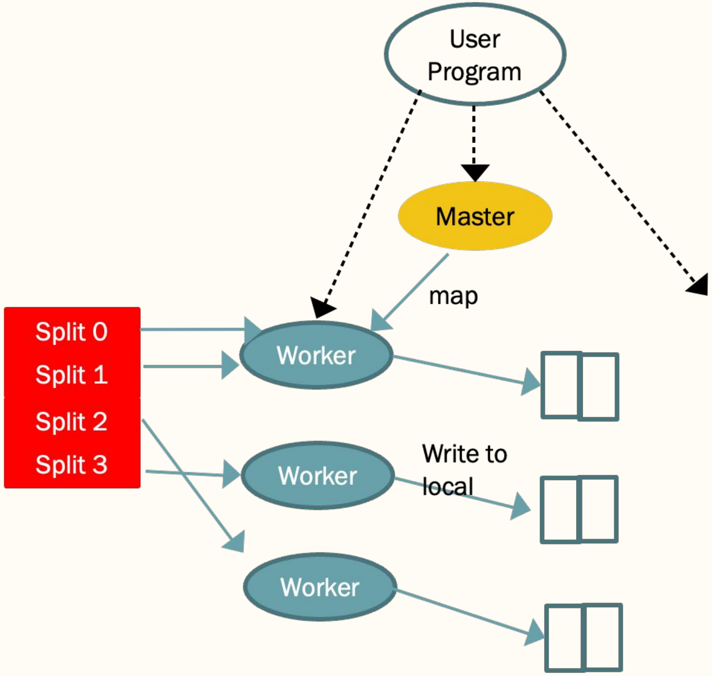
Execution of MapReduce

- Partition input into set of M splits
  - Splits processed in parallel
  - Each split typically 16-64 MB
- Start many copies of program on cluster
  - Master task
  - M map tasks
  - R reduce tasks
  - M+R >> workers
- Master assigns maps tasks to workers



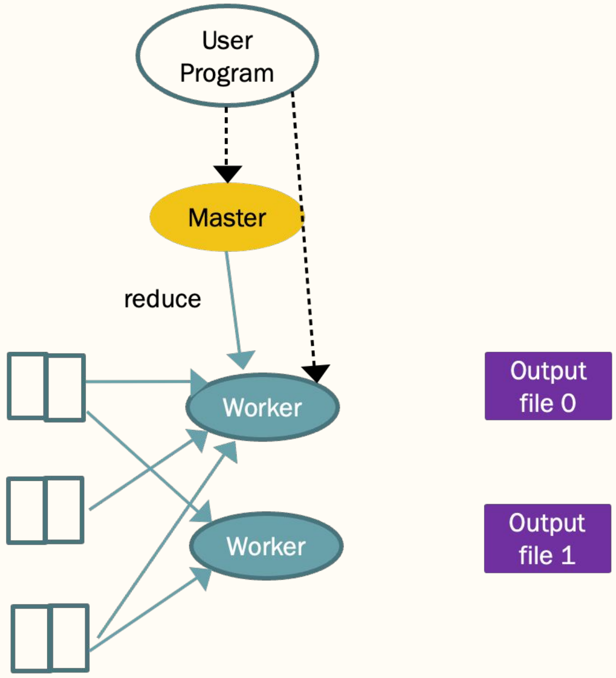
Execution of MapReduce

- Partition input into set of M splits
  - Splits processed in parallel
  - Each split typically 16-64 MB
- Start many copies of program on cluster
  - Master task
  - M map tasks
  - R reduce tasks
  - M+R >> workers
- Master assigns maps tasks to workers
- Map tasks apply map function to input key/value pairs and write intermediate values to local memory
- Map results written to local disk, partitioned into R regions based on partition function, tell master



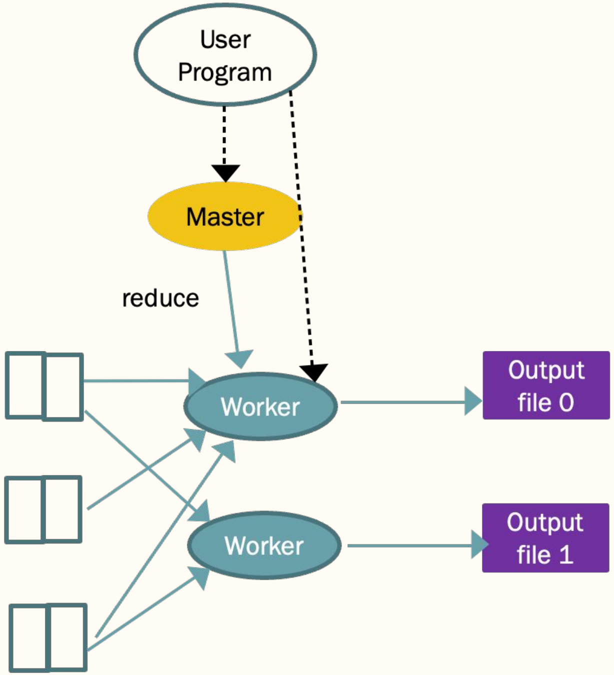
Execution of MapReduce (2)

- Master tells reduce tasks where to get data. Use RPC calls to get data from map tasks' disks



Execution of MapReduce (2)

- Master tells reduce tasks where to get data. Workers use RPC calls to get data from map tasks' disks
- Reduce tasks
  - Sort by intermediate keys
  - Apply reduce function
  - Append output to file



Example Partition Function

```
unsigned long MR_DefaultHashPartition(char *key, int num_partitions) {
    unsigned long hash = 5381;
    int c;

    while ((c = *key++) != '\0')
        hash = hash * 33 + c;
    return hash % num_partitions;
}
```

https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/concurrency-mapreduce

Fault Tolerance

- Ping workers periodically to establish status
- Re-execute map tasks on failure
- Atomic commits of map and reduce task outputs
  - Map buffers output locally and notifies master of local file names
  - Reduce buffers into local file and then atomically renames temporary file to output file

Locality

- Master tries to assign map tasks at node where data is replicated or nearby
- Combiner functions
    - Executed on map task machine
    - Combines partial results from this map task before writing results to local intermediate file

Straggler Tasks

- Some tasks take a really long time to execute
    - Maybe because of failed node
    - Maybe because of overloaded node
- When few tasks left, master schedules backup executions of in-progress tasks

Motivation for Google File System

Large files frequently read sequentially

Files frequently read-only after creation

File appends from potentially multiple writers

Commodity parts mean frequent failures

Google File System Characteristics

- Redundancy and Fault Tolerance
- Large files
- Optimized for large sequential reads
- Support for append-only writes
- Prioritize bandwidth over latency

Mechanisms

- Files divided into large chunks (64MB)
- Master orchestrates disbursal of meta data to clients and coordination of chunk servers
- Chunk servers respond to data requests from clients
- Chunks are replicated

Keeping data consistent

- Meta data handled by single master so no race conditions there
- Random access writes possible but not optimized
- Record append performed atomically
- File regions are defined as consistent or inconsistent
    - Applications must deal with occasional inconsistent data
        - E.g. Duplicate entries
- Master appoints primary replica to determine order of updates to chunk and orchestrate those updates happening at replicas

Limitations of MapReduce

Acyclic data flow

- No iterative jobs
- No interactive analysis

Spark

- Insight: Some applications reuse working set of data across multiple operations
- Abstraction 1: Resilient Distributed Datasets (RDDs)
  - Read-only collection of objects partitioned across set of machines that can be rebuilt in case of failures
  - Can explicitly cache RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations
- Abstraction 2: Parallel operations on datasets
- Additional features: 2 types of shared variables
  - Read-only broadcast variables
  - Accumulators

RDD

Constructed from

- File
- Dividing collection of data (e.g., array) into slices
- Transforming existing RDD (e.g., use map or filter)
- Change persistence of RDD
  - Cache (hint to keep in memory)
  - Save (writes to filesystem)

Parallel Operations

- Reduce
  - Combine dataset elements using associative function
- Collect
  - Send all elements to driver program
- Foreach
  - Pass each element through user provided function
- Invoke operations like map, filter, reduce by passing closures (functions) to Spark

Shared Variables

- Broadcast variables
  - Variable saved to file
  - Can be cached by Spark at worker node
- Accumulator variables
  - Each worker has separate copy of accumulator and initialized to 0
  - After task completes, worker sends message to driver program containing updates made to accumulator
  - Driver applies updates

Example

```
val file = spark.textFile(“hdfs://…”)           // create RDD
val errs = file.filter(_.contains(“ERROR”))   // transform RDD
val ones = errs.map(_ => 1)                     // map each line to 1
val count = ones.reduce(_ + _)              // add up 1s in reduce
```

Example

```
val points = spark.textFile(…).map(parsePoint).cache()  // create RDD

var w = Vector.random(D) // d-dimensional vector
// update w
for(i <- 1 to ITERATIONS) {
    val grad = spark.accumulator(new Vector(D))
    for(p <- points) { // foreach runs in parallel
        val s = (1/(1+exp(-p.y*(w dot p.x))) -1)*p.y
        grad += s * p.x
    }
    w -= grad.value
}
```

AI Overview

Hadoop is a big data framework focused on storing and processing massive datasets, while Spark is a fast, distributed computing framework that excels at real-time data processing and machine learning. Hadoop primarily uses disk storage and MapReduce for processing, whereas Spark uses in-memory processing with Resilient Distributed Datasets (RDDs).