DistributedMapReduce
Updates

• Updates distributed to your project 8 repositories
  • do Team->Update
  • try submitting
  • should submit to project 9

• Deadline Friday
  • late deadline Monday, 10% penalty
  • don’t expect much help over the weekend
Core Mastery track

- Implement DistributedMapReduce
- But use threads for workers
  - number of threads = # of compute nodes requested
- Don’t need to handle combiner
- Code will be hand audited for correct synchronization
DistributedMapReduce

- Master computation
- MapperWorker
- ReducerWorker
Mapper Worker

- worker id
- FileRange
- Mapper
- Partitioner
- Combiner (optional, may be null)
- numberPartitions (you can decide if you want to include this, may be useful)
What is does (leaving out combiner for now)

- Call setup on Mapper
- Invokes the Reader with the FileRange
- FileIndex, String values written by Reader to Mapper
- output pair from Mapper gets written to file determined by applying partition function to key
- Call cleanup on Mapper
- close all output files
Which output files?

- With 4 partitions, worker 17 will write to
  - phase1/0/17
  - phase1/1/17
  - phase1/2/17
  - phase1/3/17
- There is no phase 2
ReducerWorker

- worker id
- partition number
- list of successful map workers or list of failed map workers
- Reducer
- Combiner (optional, may be null)
What reducer worker does

• Read all files from successful jobs in the directory for it’s partition
  • phase1/partition#
• Put the results into an accumulating pair writer
• create a reducer context that writes to output/workerId
• call reducer.setup
• For key, List<value> pairs from accumulating pair writer, call
  reducer.reduce
• call reducer.cleanup
• close output stream
What the master does

• Initialize phase1 and output directories
• Listen via multicast until we have the requested number of unique ServerLocations
• Compute FileRanges
• Send out MapperWorkers to WorkerRunners until all ranges computed
• Send out ReducerWorkers until all partitions reduced
Master Constructor

- `baseDir` -- directory that contains input, phase1 and output directories
- `mapperClass` -- class for creating mappers
- `combinerClass` -- class for creating combiners; may be null
- `reducerClass` -- class for creating reducers
- `partitionerClass` -- class for creating partitioners; must define constructor that takes an int argument
Master runJob method parameters

- int mapTasks -- parameter to FileSplitCalculator
- int reduceTasks -- number of partitions
- int computeNodes -- number of unique ServerLocations collected via multicast to serve as WorkerRunners
Sending out a job

- For each workerRunner, create a thread. In that thread
  - get task. If task already completed, skip it and go back for another.
  - enqueue task for retry
  - mark worker as failed
  - send job out to worker
  - if job successful and we are the first job to complete that task, remove worked from set of failed workers
Keep track of tasks and workers

- Queue of tasks to be done
- Set of successfully completed tasks
- Set of “failed” workers
  - work id added to set when created
  - removed if worker is first to successfully complete task
Tasks

- When a task is completed, don’t try to remove it from a queue
  - too expensive: need to iterate through the queue
- Instead, maintain a set of successfully completed tasks
- When you remove something from the queue, check to see if it has already been completed
  - only way to completely remove a task from the queue
- Otherwise, re-enqueue the task for another attempt
Task queues

- Simplest: circular queue: when you remove a task that hasn’t been completed, add it back to the end of the queue
  - downside: initial reexecution are of jobs running the longest, likely to complete soon
First reexecution is for jobs that have been running the longest
Actually sending out a job

- Create a socket to the ServerLocation
- Create an ObjectOutputStream from the socket output stream
- write and flush a worker
- Call HeartBeat.waitForSuccess on the socket
Combiners

- Combiners can be used in two places: Mappers and Reducers
CombiningAccumulatingPairWriter

- Have the Mapper write to a CombininingAccumulatingPairWriter
- Like a AccumulatingPairWriter, except
  - if list gets too long (16 items?), it combines them down to a single item
  - getResult reduces the list of results for each key, returns a Map<K,V> rather than Map<K,List<V>>>
CombiningPairWriter

- A decorator: wraps around another PairWriter, combined results get written to that
- if too many keys, find oldest key, combine values, write key and combined value to delegate
- if list of values gets too long (16?) combine values
- when done, iterate through key, list<value> pairs, combine values, write key, combinedValue to delegate
LinkedHashMap

• Very cool data structure

• Like a hash map, but allows you to iterate through the elements either in insertion order or access order

• When a new element is added to the map, it calls removeEldestEntry(Map.Entry e). You can override that method to decide whether to delete that entry.

• i.e., a LRU (least recently used) cache