CS 6240: Parallel Data Processing in MapReduce

Mirek Riedewald

Course Information

- Homepage: http://www.ccs.neu.edu/home/mirek/classes/2012-F-CS6240/
  - Announcements
  - Lecture handouts
  - Office hours
- Homework management through Blackboard
- Prerequisites: CS 5800/CS 7800, or consent of instructor

Grading

- Homework/project: 60%
- Midterm 30%
- Participation 10%
  - Ask/answer in class; answer questions on Piazza
- No copying or sharing of homework solutions!
  - But you can discuss general challenges and ideas
- Material allowed for exams
  - Any handwritten notes (originals, no photocopies)
  - Printouts of lecture summaries distributed by instructor

Instructor Information

- Instructor: Mirek Riedewald (332 WVH)
  - Office hours: Tue 4:00-5:30pm
  - Post questions on Piazza
  - Email for appointment if you cannot make it during office hours (or stop by for 1-minute questions)
- TA: Alper Okcan (472 WVH)

Course Materials

- Hadoop: The Definitive Guide by Tom White
- Hadoop in Action by Chuck Lam
  - Both available from Safari Books Online at http://0-proquest.safaribooksonline.com.ilsprod.lib.neu.edu/
  - Use your myNEU credentials
- Other resources mentioned in syllabus and class homepage

Course Content and Objectives

- How to process Big Data
  - Different from traditional approaches to parallel computation for smaller data
- Learn important fundamentals of selected approaches
  - Current trends and architectures
  - Parallel programming in (raw) MapReduce
    - Programming model and Hadoop open source implementation
  - Creating data processing workflows with Pig Latin
  - HBase for storing and managing big data
  - MapReduce versus SQL and other related approaches
- Various problem types and design patterns

Course Content and Objectives

- Gain an intuition for how to deal with big-data problems
- Hands-on MapReduce practice
  - Writing MapReduce programs and running them on the Amazon Cloud
  - Understanding the system architecture and functionality below MapReduce
  - Learning about limitations of MapReduce
- Might produce publishable research

7

Words of Caution 1

- We can only cover a small part of the parallel computation universe
  - Do not expect all possible architectures, programming models, theoretical results, or vendors to be covered
  - Explore complementary courses in CCIS and ECE
- This really is an algorithms course, not a basic programming course
  - But you will need to do a lot of non-trivial programming

8

Words of Caution 2

- This is still a fairly a new course, so expect rough edges like too slow/fast pace, uncertainty in homework load estimation
- There are few certain answers, as people in research and leading tech companies are trying to understand how to deal with big data
- We are working with cutting edge technology
  - Bugs, lack of documentation, new Hadoop API
- In short: you have to be able to deal with inevitable frustrations and plan your work accordingly…
- …but if you can do that and are willing to invest the time, it will be a rewarding experience

9

Running Your Code

- You need to set up an account with Amazon Web Services (AWS)
- Requires a credit card
- We give you $100 in credit for this course
- Should be sufficient for all assignments
  - Develop and test on your laptop
  - Deploy once you are confident things work
  - Monitor your job and make sure it terminates as expected

10

How to Succeed

- Attend the lectures and take your own notes
  - Helps remembering (compared to just listening)
  - Capture lecture content more individually than our handouts
  - Free preparation for exams
- Go over notes, handouts, book soon after lecture
  - Try to explain material to yourself or friend
- Look at content from previous lecture right before the next lecture to "page-in the context"

11

How to Succeed

- Ask questions during the lecture
  - Even seemingly simple questions show that you are thinking about the material and are genuinely interested
- Work on the HW assignment as soon as it comes out
  - Can do most of the work on your own laptop
  - Time to ask questions and deal with unforeseen problems
  - We might not be able to answer all last-minute questions right before the deadline
- Students with disabilities: contact me by September 18

12

What Else to Expect?

- Need strong Java programming skills
  - Code for Hadoop system is in Java
  - Hadoop supports other languages, but use at your own risk (we cannot help you and have not tested it)
- Need strong algorithms background
  - Analyze problems and solve them using an unfamiliar framework
- Basic understanding of important system concepts
  - File system, processes, network basics, computer architecture

13

Why Focus on MapReduce?

- MapReduce is viewed as one of the biggest breakthroughs for processing massive amounts of data.
- It is widely used at technology leaders like Google, Yahoo, Facebook.
- It has huge support by the open source community.
- Amazon provides special support for setting up Hadoop MapReduce clusters on its cloud infrastructure.
- It plays a major role in current database research conferences (and many other research communities)

14

Let us first look at some recent trends and developments that motivated MapReduce and other approaches to parallel data processing.

15

Why Parallel Processing?

- Answer 1: big data

16

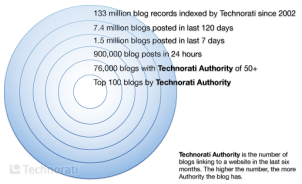
How Much Information?

- Source: http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/execsum.htm
- 5 exabytes (10^{18}) of new information from print, film, optical storage in 2002
  - 37,000 times Library of Congress book collections (17M books)
- New information on paper, film, magnetic and optical media doubled between 2000 and 2003
- Information that flows through electronic channels—telephone, radio, TV, Internet—contained 18 exabytes of new information in 2002

17

Web 2.0

- Billions of Web pages, social networks with millions of users, millions of blogs
  - How do friends affect my reviews, purchases, choice of friends
  - How does information spread?
  - What are "friendship patterns"
    - Small-world phenomenon: any two individuals likely to be connected through short sequence of acquaintances



18

Facebook Statistics

- 955M active users (June '12), 81% outside US/Canada
- More than 100 petabytes of photos and videos
- August 2011: 30 billion pieces of content (web links, news stories, blog posts, notes, photo albums, etc.) shared each month
  - Avg. user created 90 pieces of content per month

19

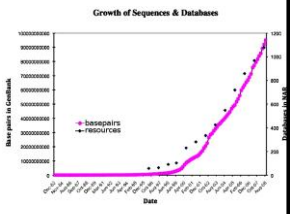
Business World

- Fraudulent/criminal transactions in bank accounts, credit cards, phone calls
  - Billions of transactions, real-time detection
- Retail stores
  - What products are people buying together?
  - What promotions will be most effective?
- Marketing
  - Which ads should be placed for which keyword query?
  - What are the key groups of customers and what defines each group?
- Spam filtering

20

eScience Examples

- Genome data
- Large Hadron Collider
  - Petabytes of raw data per year
- SkyServer
  - 818 GB, 3.4 billion rows
- DataONE
  - "Universal access to data about life on earth and the environment"
- Cornell Lab of Ornithology
  - 107M observations, 100s of attributes



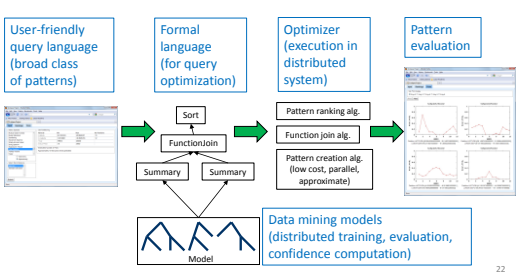
Source: Nature

21

Our Scolopax Project

- Search for patterns in prediction models based on user preferences

Make this as easy and fast as Web search



22

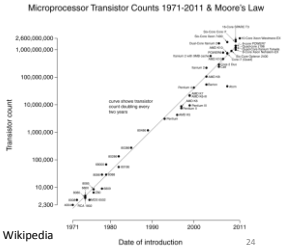
Why Parallel Processing?

- Answer 1: big data
- Answer 2: hardware trends

23

The Good Old Days

- Moore's Law: number of transistors that can be placed inexpensively on an integrated circuit doubles about every 2 years
- Computational capability improved at similar rate
  - Sequential programs became automatically faster
- Parallel computing never became mainstream
  - Reserved for high-performance computing niches

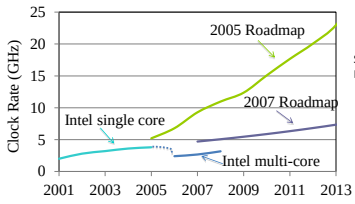


Source: Wikipedia

24

"New" Realities

- "Party" ended around 2004
- Heat issues prevent higher clock speeds
- Clock speed remains below 4 GHz



Source: Dave Patterson, UCB

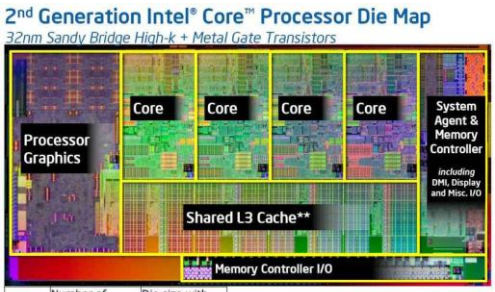
25

Multi-Core CPUs

- Clock speed stagnates, but number of cores increases
  - Core is like a processor, but shares chip with other cores
  - Cores typically share some cache, memory bus, access to same main memory
- Need to keep multiple cores busy to exploit additional transistors on chip
  - Multi-threaded applications

26

Processor Example (Source: Intel)



Die	Number of Transistors (mio)	Die size with Scribe (mm2)
4+2	995	216
2+2	624	149
2+1	504	131

Typical Multi-Core Properties

- Each core has some local cache (e.g., L1, L2)
- The cores share some cache (e.g., L3)
- All cores access same memory through bus
- Misses become much more expensive from L1 to L3, even more when accessing memory

28

Important Numbers (Source: Google's Jeff Dean @LADIS'09)

L1 cache reference	0.5
Branch mispredict	5
L2 cache reference	7
Mutex lock/unlock	25
Main memory reference	100
Compress 1 KB with Zippy	3,000
Send 2 KB over 1 Gbps network	20,000
Read 1 MB sequentially from memory	250,000
Round trip within same data center	500,000
Disk seek	10,000,000
Read 1 MB sequentially from disk	20,000,000
Send packet CA -> Holland -> CA	150,000,000

All times in ns.

29

Other Trends

- Datacenter as a computer
  - Hundreds to tens of thousands of commodity machines for large-scale data processing
- Cloud computing
  - Often powered by data center(s)
- GPU computing
  - Initially developed for fast parallel graphics computations, now also used for general computations
- Parallel data processing is becoming mainstream

30

Parallel Architectures

- Multi-core chips
- Datacenter as a computer

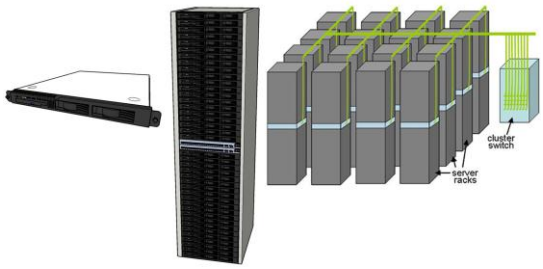
31

Warehouse-Scale Computer (WSC)

- Hundreds or thousands of commodity PCs
  - Better cost per unit of computational capability than specialized hardware due to economies of scale of commodity hardware
  - Easy to "scale out" by adding more machines
- Organized in racks in data centers
- Relatively homogenous hardware and system software platform with common system management layer
  - Often run smaller number of very large applications like Internet services

32

Basic Architecture



Source: Barroso, Holzle (2009)

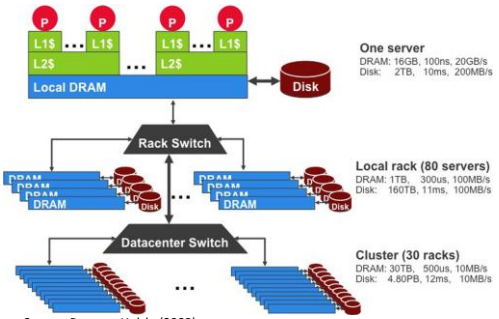
33

Typical Specs

- Low-end servers in 1U enclosure in 7' rack
- Rack-level switch with 1- or 10-Gbps links
- Connected by one or more cluster switches
  - Can include >10,000 servers
- Local (cheap) disks on each server
  - Managed by global distributed file system
- Might have Network Attached Storage (NAS) devices for more centralized storage solution

34

Storage Hierarchy



Source: Barroso, Holzle (2009)

35

Programming WSCs

- Build cluster infrastructure and services that hide architecture complexity from developers
  - Program it like a single big computer, but avoid inefficient code
- Need easy way to keep hundreds or thousands of CPUs busy
- Handle failures transparently
  - With 1000 commodity machines, failures are the norm, not the exception
  - Developers want to focus on their application, not how to deal with failures of hardware and low-level services
- This is where MapReduce comes into the picture!

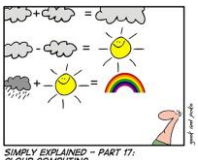
36

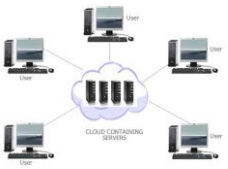
Parallel Architectures

- Multi-core chips
- Datacenter as a computer
- Cloud computing

The Cloud

- Many different versions of Clouds
- Common idea: customers use virtual resources without knowing details about underlying hardware
  - Could run on cluster, multiple data centers, or large parallel machine
- Typical use 1: reserve virtual machines to create virtual cluster
- Typical use 2: connect through Web browser and run favorite application
- Typical use 3: build own app on top of services offered by Cloud provider
  - Database, document management, Web design, workflow, analytics





Cloud Computing

- Goal: Move data and programs from desktop PCs and corporate server rooms to "compute cloud"
- Related buzzwords: on-demand computing, software as a service (SaaS), Internet as platform
- Starts to replace shrink-wrap software
  - MSFT Word on desktop PC vs. Google Docs

Back to the Future…

- 1960s: service bureaus, time-sharing systems
  - Hub-and-spoke configuration: terminal access through phone lines, central site for computation
- 1980s: PCs "liberate" programs and data from central computing center
  - Customization of computing environment
  - Client-server model

…or not?

- Cloud is not the same as 1960's hub
  - Client can communicate with many servers at the same time
  - Servers can communicate with each other
- Still, functions migrate to distant data centers
  - "Core" and "fringe"
    - Storage, computing, high bandwidth, and careful resource management in core
    - End users initiate requests from fringe

Why Clouds?

- High price of total control
  - Software installation, configuration, and maintenance
  - Maintenance of computing infrastructure
  - Difficult to grow and shrink capacity on demand
- Easier software development
  - Replaces huge variety of operating environments by computing platform of vendor's choosing
  - But: server interaction with variety of clients
- Easier to deploy updates and bug fixes
- Easier to leverage multi-core, parallel systems
  - Single instance of Word cannot utilize 100 cores, but 100 instances of Word can

Example Cloud Offerings

- Document processing
  - Google Docs: word processor, spreadsheet, presentations
  - Adobe: Acrobat.com, Photoshop Express
  - Microsoft Office 365
- Enterprise applications
  - Salesforce.com: customer relationship management, sales marketing apps
  - Microsoft Dynamics CRM, IBM Tivoli Live
- Cloud infrastructure
  - Amazon Web Services: storage, computing as needed (pay as you go)
  - IBM Smart Cloud, Google App Engine, Force.com, Microsoft Azure
- Cloud OS
  - User interface in Web browser
  - New browser wars: browser as new Cloud OS

43

Challenges

- Scalability
  - More users, complex interactions between applications
- Many-to-many communication
  - Client invokes programs on multiple servers, server talks to multiple clients
- Browser is limited compared to traditional OS
  - Limited functionality
  - Fewer development tools

44

More Challenges

- Heterogeneous environment
  - Database backend with SQL
  - JavaScript, HTML at client
  - Server app written in PHP, Java, Python
  - Information exchanged as XML
- New role for open source movement?
  - Open source word processor vs. running a service

45

Biggest Problems

- Privacy, security, reliability
  - What if the service is not accessible?
  - Who owns the data?
  - Lose access to data if bill not paid?
  - Guarantee that deleted documents are really gone?
  - How aggressive about protecting data, e.g., against government access?
  - How to know if data is leaked to third party?

46

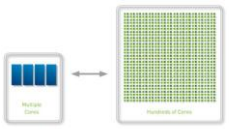
Parallel Architectures

- Multi-core chips
- Datacenter as a computer
- Cloud computing
- GPU computing

47

GPU vs. CPU

- Optimized for massively parallel processing
  - Graphics processing
- Challenge: how to create applications for 100s of cores?
  - Example: NVIDIA developed CUDA
  - Used widely for general-purpose computations

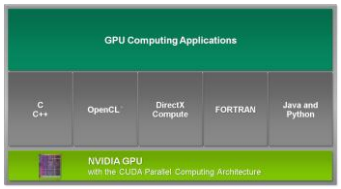


Source: NVIDIA

48

CUDA (Source: NVIDIA)

- CUDA programming model provides abstractions for data and task parallelism
  - Programmer can express parallelism in high-level languages such as C, C++, Fortran or driver APIs such as OpenCL™ and DirectX™-11 Compute
  - Programming model guides programmers to partition the problem into coarse sub-problems that can be solved independently in parallel
  - Fine grain parallelism in the sub-problems is then expressed such that each sub-problem can be solved cooperatively in parallel.



49

Course Content in a Nutshell

- In big-data processing, usually the same computation needs to be applied to a lot of data
  - Possibly many such steps (think "workflow")
- Divide the work between multiple processors
  - Make sure you can handle data transfer efficiently
- Combine intermediate results from multiple processors

50

Why This Is Not So Easy

- How can the work be partitioned?
- What if too much intermediate data is produced?
- How do we start up and manage 1000s of jobs?
- How do we get large data sets to processors or move processing to the data?
- How do we deal with slow responses and failures?

51

More Problems

- Shared resources limit scalability
  - Cost of managing concurrent access
- Shared-nothing architectures still need communication for processes to share data
- Easy to get into problems like deadlocks and race conditions
- It is generally difficult to reason about the behavior and correctness of concurrent processes
  - Especially when failures are part of the model
- Inherent tradeoff between consistency, availability, and partition tolerance (Brewer's Conjecture)

52

What Can We Do?

- Work at the right level of abstraction
  - Too low-level: difficult to write programs, e.g., to deal with locks; need to customize code for different systems
  - Too high-level: poor performance if control for crucial bottleneck is "abstracted away"
- Use more declarative style of programming
  - Define WHAT needs to be computed, not HOW this is done at the low level
  - Well-known success story: SQL and databases

53

Recipes for Success

- Use hardware that can scale out, not just up
  - Doubling the number of commodity servers is easy, but buying a double-sized SMP machine is not.
- Have data located near the processors
  - Sending petabytes around is not a good idea
- Avoid centralized resources that are likely bottlenecks, e.g., single shared memory bus for many cores
- Read and write data sequentially
  - Assume random I/O takes 20 msec, disk streams data sequentially at 100 MB/sec, and record size is 1 KB
  - During 1 random I/O, can read 2000 records sequentially
- MapReduce does all this, and its level of abstraction seems to have hit a sweet spot

54

Algorithms First

- No matter which parallel programming model we use, we first need to understand what part of a computation can be performed in parallel
- More precisely…

55

Writing Parallel Programs

- Analyze problem and identify what can be done in parallel
  - Dependencies: if I need data D as input for a task, then I cannot run this task and the creation of D in parallel.
  - Coordination requirements: when do parallel tasks have to communicate and how much data is sent?
  - Best sequential algorithm might not be easy to parallelize—find alternative solutions
- Create an efficient implementation
  - Make sure solution is a good fit for the given architecture and programming model

56

Examples

- Let us look at some examples to get a feeling for the challenges

57

Sum Of Integers

- Compute sum of a large set of integers
- Sequential: simple for-loop (scan)
- Parallel: assign chunk of data to each processor to compute local sum, then add them together
- Algorithmically easy, but…
  - Where do the chunks come from? Partitioning data file into multiple chunks might take as long as sequential computation.
  - What if data transfer is the bottleneck? Then pushing k chunks from disk to k cores might not be a good idea.
  - Who computes final sum and how do the local sums get there?

58

Word Count

- Count the number of occurrences of each word in a large document
- Sequential: read document sequentially, update counters for each word
  - Need data structure, e.g., hash map, to keep track of counts
- Parallel: each processor does this for a chunk using local data structure, then counts are aggregated
- Improvement (?): use shared data structure for counts
  - Good: no "replication", no need for final summation step
  - Bad: need to coordinate access to shared data structure, not a good fit for shared-nothing architecture
- What if some documents are much larger than others?
  - Need to deal with data skew, e.g., break up large documents

59

Median

- Find the median of a set of integers
- Holistic aggregate function
  - Chunk assigned to a processor might contain mostly smaller or mostly larger values, and the processor does not know this without communicating extensively with the others
- Parallel implementation might not do much better than sequential one
- Efficient approximation algorithms exist

60

Parallel Office Tools

- Parallelize Word, Excel, email client?
- Need to rewrite them as multi-threaded applications
  - Seem to naturally have low degree of parallelism
- Leverage economies of scale: *n* processors (or cores) support *n* desktop users by hosting the service in the Cloud
  - E.g., Google docs

61

Before exploring parallel algorithms in more depth, how do we know if our parallel algorithm or implementation actually does well or not?

62

Measures Of Success

- If sequential version takes time t, then parallel version on n processors should take time t/n
  - Speedup = sequentialTime / parallelTime
  - Note: job, i.e., work to be done, is fixed
- Response time should stay constant if number of processors increases at same rate as "amount of work"
  - Scaleup = workDoneParallel / workDoneSequential
  - Note: time to work on job is fixed

63

Things to Consider: Amdahl's Law

- Consider job taking sequential time 1 and consisting of two sequential tasks taking time t_1 and $1\text{-}t_1$, respectively
- Assume we can perfectly parallelize the first task on n processors
  - Parallel time: $t_1/n + (1 - t_1)$
- Speedup = $1 / (1 - t_1(n\text{-}1)/n)$
  - t_1=0.9, n=2: speedup = 1.81
  - t_1=0.9, n=10: speedup = 5.3
  - t_1=0.9, n=100: speedup = 9.2
  - Max. possible speedup for t_1=0.9 is 1/(1-0.9) = 10

64

Implications of Amdahl's Law

- Parallelize the tasks that take the longest
- Sequential steps limit maximum possible speedup
  - Communication between tasks, e.g., to transmit intermediate results, can inherently limit speedup, no matter how well the tasks themselves can be parallelized
- If fraction x of the job is inherently sequential, speedup can never exceed 1/x
  - No point running this on too many processors

65

Performance Metrics

- Total execution time
  - Part of both speedup and scaleup
- Total resources consumed
- Total amount of money paid
- Total energy consumed
- Optimize some combination of the above
  - E.g., minimize total execution time, subject to a money budget constraint

66

Popular Solution: Load Balancing

- Avoid overloading one processor while other is idle
  - Careful: if better balancing increases total load, it might not be worth it
  - Careful: optimizes for response time, but not necessarily other metrics like $ paid
- Static load balancing
  - Need cost analyzer like in DBMS
- Dynamic load balancing
  - Easy: Web search
  - Hard: join

Let's see how MapReduce works.

MapReduce

- Proposed by Google in research paper
  - Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004
- MapReduce implementations such as Hadoop differ in details, but main principles are the same

Overview

- MapReduce = programming model and associated implementation for processing large data sets
- Programmer essentially just specifies two (sequential) functions: map and reduce
- Program execution is automatically parallelized on large clusters of commodity PCs
  - MapReduce could be implemented on different architectures, but Google proposed it for clusters

Overview

- Clever abstraction that is a good fit for many real-world problems
- Programmer focuses on algorithm itself
- Runtime system takes care of all messy details
  - Partitioning of input data
  - Scheduling program execution
  - Handling machine failures
  - Managing inter-machine communication

Programming Model

- Transforms set of input key-value pairs to set of output values (notice small modification compared to paper)
- Map: (k1, v1) → list (k2, v2)
- MapReduce library groups all intermediate pairs with same key together
- Reduce: (k2, list (v2)) → list (k3, v3)
  - Usually zero or one output value per group
  - Intermediate values supplied via iterator (to handle lists that do not fit in memory)

Example: Word Count

- Insight: can count each document in parallel, then aggregate counts
- Final aggregation has to happen in Reduce
  - Need count per word, hence use word itself as intermediate key (k2)
  - Intermediate counts are the intermediate values (v2)
- Parallel counting can happen in Map
  - For each document, output set of pairs, each being a word in the document and its frequency of occurrence in the document
  - Alternative: output (word, 1) for each word encountered

Word Count in MapReduce

Count number of occurrences of each word in a document collection:

```
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 += v;
 Emit(key, result);
```

Almost all the coding needed
(need also MapReduce specification object with names of input and output files, and optional tuning parameters)

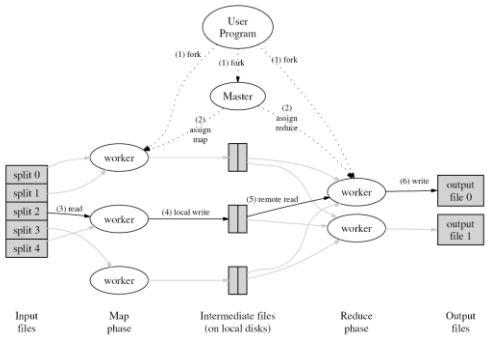
Execution Overview

- Data is stored in files
  - Files are partitioned into smaller splits, typically 64MB
  - Splits are stored (usually also replicated) on different cluster machines
- Master node controls program execution and keeps track of progress
  - Does not participate in data processing
- Some workers will execute the Map function, let's call them mappers
- Some workers will execute the Reduce function, let's call them reducers

Execution Overview

- Master assigns map and reduce tasks to workers, taking data location into account
- Mapper reads an assigned file split and writes intermediate key-value pairs to local disk
- Mapper informs master about result locations, who in turn informs the reducers
- Reducers pull data from appropriate mapper disk location
- After map phase is completed, reducers sort their data by key
- For each key, the Reduce function is executed and output is appended to final output file
- When all reduce tasks are completed, master wakes up user program

Execution Overview



Master Data Structures

- Master keeps track of status of each map and reduce task and who is working on it
  - Idle, in-progress, or completed
- Master stores location and size of output of each completed map task
  - Pushes information incrementally to workers with in-progress reduce tasks

Handling Mapper Failures

- Master pings every worker periodically
- Workers who do not respond in time are marked as failed
- Mapper's in-progress and completed tasks are reset to idle state
  - Can be assigned to other mapper
  - Completed tasks are re-executed because result is stored on mapper's local disk
- Reducers are notified about mapper failure, so that they can read the data from the replacement mapper

79

Handling Reducer Failures

- Failed reducers identified through ping as well
- Reducer's in-progress tasks are reset to idle state
  - Can be assigned to other reducer
  - No need to restart completed reduce tasks, because result is written to distributed file system

80

Handling Master Failure

- Failure unlikely, because it is just a single machine
- Can simply abort MapReduce computation
  - Users re-submit aborted jobs when new master process is up
- Alternative: master writes periodic checkpoints of its data structures so that it can be re-started from checkpointed state

81

Semantics with Failures

- If map and reduce are deterministic, then output is identical to non-faulting sequential execution
  - For non-deterministic operators, different reduce tasks might see output of different map executions
- Relies on atomic commit of map and reduce outputs
  - In-progress task writes output to private temp file
  - Mapper: on completion, send names of all temp files to master (master ignores if task already complete)
  - Reducer: on completion, *atomically* rename temp file to final output file (needs to be supported by distributed file system)

82

Practical Considerations

- Conserve network bandwidth ("Locality optimization")
  - Schedule map task on machine that already has a copy of the split, or one "nearby"
- Create backup tasks to deal with machines that take unusually long for the last in-progress tasks ("stragglers")

83

Refinements

- User-defined partitioning functions for reduce tasks
  - Default: assign key K to reduce task *hash(K) mod R*
  - Use *hash(Hostname(urlkey)) mod R* to have URLs from same host in same output file
  - We will see others in future lectures
- Combiner function to reduce mapper output size
  - Pre-aggregation at mapper for reduce functions that are commutative and associative
  - Often (almost) same code as for reduce function

84

Careful With Combiners

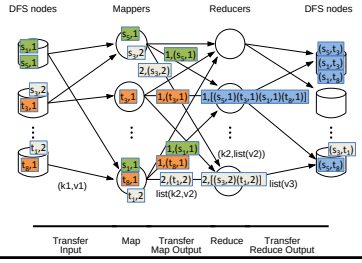
- Consider Word Count, but assume we only want words with count > 10
  - Reducer computes total word count, only outputs if greater than 10
  - Combiner = Reducer? No. Combiner should not filter based on its local count!
- Consider computing average of a set of numbers
  - Reducer should output average
  - Combiner has to output (sum, count) pairs to allow correct computation in reducer

Equi-Join in MapReduce

- Given two data sets $S=(s_1,s_2,\ldots)$ and $T=(t_1,t_2,\ldots)$ of integers, find all pairs (s_i,t_j) where $s_i.A=t_j.A$
- Can combine the s_i and t_j only in Reduce
  - To ensure that the right tuples end up in the same Reduce invocation, use join attribute A as intermediate key (k2)
- Map needs to output (s.A, s) for each S-tuple s (similar for T-tuples)
  - Also adds a flag indicating if the tuple is from S or T

Equi-Join in MapReduce

- Join condition: S.A=T.A
- Map(s) = (s.A, (s, "S")); Map(t) = (t.A, (t, "T"))
- Reduce computes Cartesian product of set of S-tuples and set of T-tuples with same key



Comments

- Programming model might appear very limited
- But, map and reduce can do anything with their input
  - Could implement a Turing machine inside…
  - …which could compute anything, but…
  - …would not result in a good parallel implementation.
- Challenge: find best MapReduce implementation for a given problem

Basic MapReduce Program Design

- Tasks that can be performed independently on a data object, large number of them: Map
- Tasks that require combining of multiple data objects: Reduce
- Sometimes it is easier to start program design with Map, sometimes with Reduce
- Select keys and values such that the right objects end up together in the same Reduce invocation
- Might have to partition a complex task into multiple MapReduce sub-tasks

Choosing M and R

- M = number of map tasks, R = number of reduce tasks
- Larger M, R: creates smaller tasks, enabling easier load balancing and faster recovery (many small tasks from failed machine)
- Limitation: O(M+R) scheduling decisions and O(M·R) in-memory state at master
  - Very small tasks not worth the startup cost
- Recommendation:
  - Choose M so that split size is approximately 64 MB
  - Choose R a small multiple of the number of workers; alternatively choose R a little smaller than #workers to finish reduce phase in one "wave"

Grep

- Find all lines matching some pattern
- No need to combine anything
  - Reduce is not needed, i.e., just identity function
- Map takes line and outputs it if it matches the pattern
- Map could also take an entire document and emit all matching lines
  - Not a good idea if there is a single large document, but works well if there are many documents

91

Reverse Web-Link Graph

- For each URL, find all pages (URLs) pointing to it (incoming links)
- Problem: Web page has only outgoing links
- Need all (anySource, P) links for each page P
  - Suggests Reduce with P as the key, source as value
- Map: for page *source*, create all (*target*, *source*) pairs for each link to a *target* found in page
- Reduce: since *target* is key, will receive all sources pointing to that target

92

Inverted Index

- For each word, create list of documents (document IDs) containing it
- Same as reverse Web-link graph problem
  - "Source URL" is now "document ID"
  - "Target URL" is now "word"
- Can augment this to create list of (document ID, position) pairs for each word
  - Map emits (word, (document ID, position)) while parsing a document

93

Distributed Sorting

- Can Map do pre-sorting and Reduce the merging?
  - Use *set* of input records as Map input
  - Map pre-sorts it and single reducer merges them
  - Does not scale!
- We need to get multiple reducers involved
  - What should we use as the intermediate key?

94

Distributed Sorting, Revisited

- Quicksort-style partitioning
- For simplicity, consider case with 2 machines
  - Goal: each machine sorts about half of the data
- Assuming we can find the median record, assign all smaller records to machine 1, all others to machine 2
- Sort locally on each machine, then "concatenate" output

95

Partitioning Sort in MapReduce

- Consider 2 reducers for simplicity
- Run MapReduce job to find approximate median of data
  - Hadoop also offers InputSampler
    - Writes the keys that define the partitions, to be used by TotalOrderPartitioner
    - Runs on client and downloads input data splits, hence only useful if data is sampled from few splits, i.e., splits themselves should contain random data samples
- Map outputs (sortKey, record) for an input record
- All sortKey < median are assigned to reduce task 1, all others to reduce task 2, using a partitioner
- Reduce sorts its assigned set of records

96

Partitioning Sort in MapReduce

- MapReduce has class Partitioner<KEY, VALUE>
  - Method int getPartition(KEY key, VALUE value, int numPartitions) allows assigning keys to partitions
- Example for numPartitions = 2
  - Partition 1 gets all numbers less than median
  - Partition 2 gets all larger numbers
- What about concatenating the output?
  - Not necessary, except for many small files (big files are broken up anyway)
- Generalizes obviously to more reducers

97

MapReduce and Key Sorting

- MapReduce environment guarantees that for each reduce task the assigned set of intermediate keys is processed in key order
  - After receiving all (key2, val2) pairs from mappers, reducer sorts them by key2, then calls Reduce on each (key2, list(val2)) group in order
- Can leverage this guarantee for partitioning sort
  - Reduce simply emits the records unchanged
  - No need for user sort code in Reduce function!

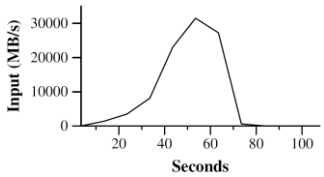
98

Experiments

- 1800 machine cluster
  - 2 GHz Xeon, 4 GB memory, two 160 GB IDE disks, gigabit Ethernet link
  - Less than 1 msec roundtrip time
- Grep workload
  - Scan 10^{10} 100-byte records, search for rare 3-character pattern, occurring in 92,337 records
  - M=15,000 (64 MB splits), R=1

99

Grep Progress Over Time



- Rate at which input is scanned as more mappers are added
- Drops as tasks finish, done after 80 sec
- 1 min startup overhead beforehand
  - Propagation of program to workers
  - Delays due to distributed file system for opening input files and getting information for locality optimization

100

Sort

- Sort 10^{10} 100-byte records (~1 TB of data)
- Less than 50 lines user code
- M=15,000 (64 MB splits), R=4000
- Use key distribution information for intelligent partitioning
- Entire computation takes 891 sec
  - 1283 sec without backup task optimization (few slow machines delay completion)
  - 933 sec if 200 out of 1746 workers are killed several minutes into computation

101

MapReduce at Google (2004)

- Machine learning algorithms, clustering
- Data extraction for reports of popular queries
- Extraction of page properties, e.g., geographical location
- Graph computations
- Google indexing system for Web search (>20 TB of data)
  - Sequence of 5-10 MapReduce operations
  - Smaller simpler code: from 3800 LOC to 700 LOC for one computation phase
  - Easier to change code
  - Easier to operate, because MapReduce library takes care of failures
  - Easy to improve performance by adding more machines

102

Summary

- Programming model that hides details of parallelization, fault tolerance, locality optimization, and load balancing
- Simple model, but fits many common problems
  - User writes Map and Reduce function
  - Can also provide combine and partition functions
- Implementation on cluster scales to 1000s of machines
- Open source implementation, Hadoop, is available

MapReduce relies heavily on the underlying distributed file system. Let's take a closer look to see how it works.

The Distributed File System

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003

Motivation

- Abstraction of a single global file system greatly simplifies programming in MapReduce
- MapReduce job just reads from a file and writes output back to a file (or multiple files)
- Frees programmer from worrying about messy details
  - How many chunks to create and where to store them
  - Replicating chunks and dealing with failures
  - Coordinating concurrent file access at low level
  - Keeping track of the chunks

Google File System (GFS)

- GFS in 2003: 1000s of storage nodes, 300 TB disk space, heavily accessed by 100s of clients
- Goals: performance, scalability, reliability, availability
- Differences compared to other file systems
  - Frequent component failures
  - Huge files (multi-GB or even TB common)
  - Workload properties
    - Design system to make important operations efficient

Data and Workload Properties

- Modest number of large files
  - Few million files, most 100 MB+
  - Manage multi-GB files efficiently
- Reads: large streaming (1 MB+) or small random (few KBs)
- Many large sequential append writes, few small writes at arbitrary positions
- Concurrent append operations
  - E.g., Producer-consumer queues or many-way merging
- High sustained bandwidth more important than low latency
  - Bulk data processing

File System Interface

- Like typical file system interface
  - Files organized in directories
  - Operations: create, delete, open, close, read, write
- Special operations
  - Snapshot: creates copy of file or directory tree at low cost
  - Record append: concurrent append guaranteeing atomicity of each individual client's append

Architecture Overview

- 1 master, multiple chunkservers, many clients
  - All are commodity Linux machines
- Files divided into fixed-size chunks
  - Stored on chunkservers' local disks as Linux files
  - Replicated on multiple chunkservers
- Master maintains all file system metadata: namespace, access control info, mapping from files to chunks, chunk locations

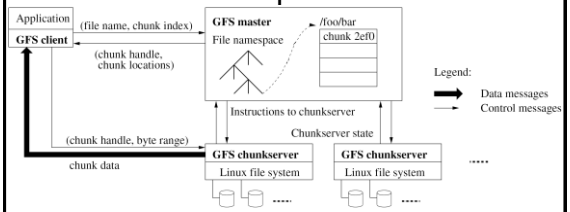
Why a Single Master?

- Simplifies design
- Master can make decisions with global knowledge
- Potential problems:
  - Can become bottleneck
    - Avoid file reads and writes through master
  - Single point of failure
    - Ensure quick recovery

High-Level Functionality

- Master controls system-wide activities like chunk lease management, garbage collection, chunk migration
- Master communicates with chunkservers through HeartBeat messages to give instructions and collect state
- Clients get metadata from master, but access files directly through chunkservers
- No GFS-level file caching
  - Little benefit for streaming access or large working set
  - No cache coherence issues
  - On chunkserver, standard Linux file caching is sufficient

Read Operation



- Client: from (file, offset), compute chunk index, then get chunk locations from master
  - Client buffers location info for some time
- Client requests data from nearby chunkserver
  - Future requests use cached location info
- Optimization: batch requests for multiple chunks into single request

Chunk Size

- 64 MB, stored as Linux file on a chunkserver
- Advantages of large chunk size
  - Fewer interactions with master (recall: large sequential reads and writes)
  - Smaller chunk location information
    - Smaller metadata at master, might even fit in main memory
    - Can be cached at client even for TB-size working sets
- Disadvantage: fewer chunks => fewer options for load balancing
  - Fixable with higher replication factor
  - Address hotspots by letting clients read from other clients

Practical Considerations

- Number of chunks is limited by master's memory size
  - Only 64 bytes metadata per 64 MB chunk; most chunks full
  - Less than 64 bytes namespace data per file
- Chunk location information at master is not persistent
  - Master polls chunkservers at startup, then updates info because it controls chunk placement
  - Eliminates problem of keeping master and chunkservers in sync (frequent chunkserver failures, restarts)

115

Consistency Model

- GFS uses a relaxed consistency model
- File namespace updates are atomic (e.g., file creation)
  - Only handled by master, using locking
  - Operations log defines global total order
- State of file region after update
  - Consistent: all clients will always see the same data, regardless which chunk replica they access
  - Defined: consistent and reflecting the entire update

116

Relaxed Consistency

- GFS guarantees that after a sequence of successful updates, the updated file region is defined and contains the data of the last update
  - Applies updates to all chunk replica in same order
  - Uses chunk version numbers to detect stale replica (when chunk server was down during update)
- Stale replica are never involved in an update or given to clients asking the master for chunk locations
- But, client might read from stale replica when it uses cached chunk location data
  - Not all clients read the same data
  - Can address this problem for append-only updates

117

Leases, Update Order

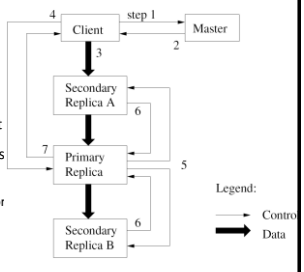
- Leases used for consistent update order across replicas
  - Master grants lease to one replica (primary)
  - Primary picks serial update order
  - Other replicas follow this order
- Lease has initial timeout of 60 sec, but primary can request extensions from master
  - Piggybacked on HeartBeat messages
  - Master can revoke lease (e.g., to rename file)
  - If no communication with primary, then master grants new lease after old one expires

118

Updating a Chunk

1. Who has lease?
2. Identity of primary and secondary replicas
3. Push data to all replicas
4. After receiving all acks, send write request to primary who assigns it a serial number
5. Primary forwards write request to all other replicas
6. Secondaries ack update success
7. Primary replies to client
   1. Also reports errors
   2. Client retries steps 3-7 on error

- Large writes broken down into chunks



119

Data Flow

- Decoupled from control flow for efficient network use
- Data pipelined linearly along chain of chunkservers
  - Full outbound bandwidth for fastest transfer (instead of dividing it in non-linear topology)
  - Avoids network bottlenecks by forwarding to "next closest" destination machine
  - Minimizes latency: once chunkserver receives data, it starts forwarding immediately
    - Switched network with full-duplex links
    - Sending does not reduce receive rate
    - 1 MB distributable in 80 msec

120

Namespace Management

- Want to support concurrent master operations
- Solution: locks on regions of namespace for proper serialization
  - Read-write lock for each node in namespace tree
    - Operations lock all nodes on path to accessed node
      - For operation on /d1/d2/leaf, acquire read locks on /d1 and /d1/d2, and appropriate read or write lock on /d1/d2/leaf
    - File creation: read-lock on parent directory
  - Concurrent updates in same directory possible, e.g., multiple file creations
  - Locks acquired in consistent total order to prevent deadlocks
    - First ordered by level in namespace tree, then lexicographically within same level

Replica Placement

- Goals: scalability, reliability, availability
- Difficult problem
  - 100s of chunkservers spread across many machine racks, accessed from 100s of clients from the same or different racks
  - Communication may cross network switch(es)
  - Bandwidth into or out of a rack may be less than aggregate bandwidth of all the machines within the rack
- Spread replicas across racks
  - Good: fault tolerance, reads benefit from aggregate bandwidth of multiple racks
  - Bad: writes flow through multiple racks
- Master can move replicas or create/delete them to react to system changes and failures

Lazy Garbage Collection

- File deletion immediately logged by master, but file only renamed to hidden name
  - Removed later during regular scan of file system namespace
  - Batch-style process amortizes cost and is run when master load is low
- Orphaned chunks identified during regular scan of chunk namespace
- Chunkservers report their chunks to master in HeartBeat messages
- Master replies with identities of chunks it does not know
  - Chunkserver can delete them
- Simple and reliable: lost deletion messages (from master) and failures during chunk creation no problem
- Disadvantage: difficult to finetune space usage when storage is tight, e.g., after frequent creation/deletion of temp files
  - Solution: use different policies in different parts of namespace

Stale Replicas

- Occur when chunkserver misses updates while it is down
- Master maintains chunk version number
  - Before granting new lease on chunk, master increases its version number
  - Informs all up-to-date replicas of new number
    - Master and replicas keep version number in persistent state
  - This happens before client is notified and hence before it can start updating the chunk
- When chunkservers report their chunks, they include version numbers
  - Older than on master: garbage collect it
  - Newer than on master: master must have failed after granting lease; master takes higher version to be up-to-date
- Master also includes version number in reply to client and chunkserver during update-process related communication

Achieving High Availability

- Master and chunkservers can restore state and start in seconds
- Chunk replication
- Master replication, i.e., operation log and checkpoints
- But: only one master process
  - Can restart almost immediately
  - Permanent failure: monitoring infrastructure outside GFS starts new master with replicated operation log (clients use DNS alias)
- Shadow masters for read-only access
  - May lag behind primary by fraction of a sec

Experiments

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

- Chunkserver metadata mostly checksums for 64 KB blocks
  - Individual servers have 50-100 MB of metadata
  - Reading this from disk during recovery is fast

Results

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

- Clusters had been up for 1 week at time of measurement
- A's network configuration has max read rate of 750 MB/s
  - Actually reached sustained rate of 580 MB/s
- B's peak rate is 1300 MB/s, but applications never used more than 380 MB/s
- Master not a bottleneck, despite large number of ops sent to it

127

Summary

- GFS supports large-scale data processing workloads on commodity hardware
- Component failures treated as norm, not exception
  - Constant monitoring, replicating of crucial data
  - Relaxed consistency model
  - Fast, automatic recovery
- Optimized for huge files, appends, large sequential reads
- High aggregate throughput for concurrent readers and writers
  - Separation of file system control (through master) from data transfer (between chunkservers and clients)

128

Now that we covered the basics of MapReduce, let's look at some Hadoop specifics.

129

Working With Hadoop

- Mostly based on Tom White's book "Hadoop: The Definitive Guide", 3rd edition
- Note: We will use the new org.apache.hadoop.mapreduce API, but…
  - Many existing programs might be written using old API org.apache.hadoop.mapred
  - Some old libraries might only support the old API

130

Important Terminology

- NameNode daemon
  - Corresponds to GFS Master
  - Runs on master node of the Hadoop Distributed File System (HDFS)
  - Directs DataNodes to perform their low-level I/O tasks
- DataNode daemon
  - Corresponds to GFS chunkserver
  - Runs on each slave machine in the HDFS
  - Does the low-level I/O work

131

Important Terminology

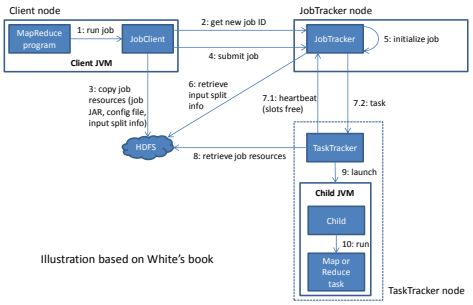
- Secondary NameNode daemon
  - One per cluster to monitor status of HDFS
  - Takes snapshots of HDFS metadata to facilitate recovery from NameNode failure
- JobTracker daemon
  - MapReduce master in Google paper
  - One per cluster, usually running on master node
  - Communicates with client application and controls MapReduce execution in TaskTrackers

132

Important Terminology

- TaskTracker daemon
  - MapReduce worker in Google paper
  - One TaskTracker per slave node
  - Performs actual Map and Reduce execution
  - Can spawn multiple JVMs to do the work
- Typical setup
  - NameNode and JobTracker run on cluster head node
  - DataNode and TaskTracker run on all other nodes
  - Secondary NameNode runs on dedicated machine or on cluster head node (usually not a good idea, but ok for small clusters)

Anatomy of MapReduce Job Run



Job Submission

- Client submits MapReduce job through Job.submit() call
  - waitForCompletion() submits job and polls JobTracker about progress every sec, outputs to console if changed
- Job submission process
  - Get new job ID from JobTracker
  - Determine input splits for job
  - Copy job resources (job JAR file, configuration file, computed input splits) to HDFS into directory named after the job ID
  - Informs JobTracker that job is ready for execution

Job Initialization

- JobTracker puts ready job into internal queue
- Job scheduler picks job from queue
  - Initializes it by creating job object
  - Creates list of tasks
    - One map task for each input split
    - Number of reduce tasks determined by mapred.reduce.tasks property in Job, which is set by setNumReduceTasks()
- Tasks need to be assigned to worker nodes

Task Assignment

- TaskTrackers send heartbeat to JobTracker
  - Indicate if ready to run new tasks
  - Number of "slots" for tasks depends on number of cores and memory size
- JobTracker replies with new task
  - Chooses task from first job in priority-queue
    - Chooses map tasks before reduce tasks
    - Chooses map task whose input split location is closest to machine running the TaskTracker instance
      - Ideal case: data-local task
  - Could also use other scheduling policy

Task Execution

- TaskTracker copies job JAR and other configuration data (e.g., distributed cache) from HDFS to local disk
- Creates local working directory
- Creates TaskRunner instance
- TaskRunner launches new JVM (or reuses one from another task) to execute the JAR

Monitoring Job Progress

- Tasks report progress to TaskTracker
- TaskTracker includes task progress in heartbeat message to JobTracker
- JobTracker computes global status of job progress
- JobClient polls JobTracker regularly for status
- Visible on console and Web UI

Handling Failures: Task

- Error reported to TaskTracker and logged
- Hanging task detected through timeout
- JobTracker will automatically re-schedule failed tasks
  - Tries up to mapred.map.max.attempts many times (similar for reduce)
  - Job is aborted when task failure rate exceeds mapred.max.map.failures.percent (similar for reduce)

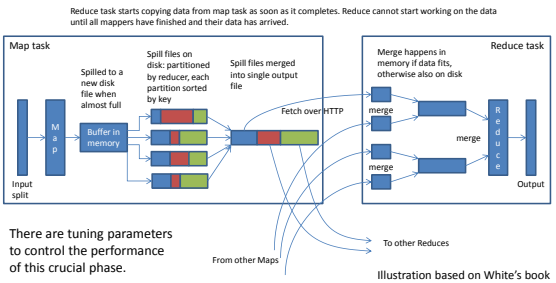
Handling Failures: TaskTracker and JobTracker

- TaskTracker failure detected by JobTracker from missing heartbeat messages
  - JobTracker re-schedules map tasks and not completed reduce tasks from that TaskTracker
- Hadoop cannot deal with JobTracker failure
  - Could use Google's proposed JobTracker take-over idea, using ZooKeeper to make sure there is at most one JobTracker

Moving Data From Mappers to Reducers

- Shuffle and sort phase = synchronization barrier between map and reduce phase
- Often one of the most expensive parts of a MapReduce execution
- Mappers need to separate output intended for different reducers
- Reducers need to collect their data from all mappers and group it by key
- Keys at each reducer are processed in order

Shuffle and Sort Overview



NCDC Weather Data Example

- Raw data has lines like these (year, temperature in **bold**)
  - 006701199099999**1950**051507004+68750+023550FM-12+038299999V0203301N00671220001CN9999999N9**+00**001+99999999999
  - 004301199099999**1950**051512004+68750+023550FM-12+038299999V0203201N00671220001CN9999999N9**+00**221+99999999999
- Goal: find max temperature for each year
  - Map: emit (year, temp) for each year
  - Reduce: compute max over temp from (year, (temp, temp,…)) list

Map

- Hadoop's Mapper class
  - Org.apache.hadoop.mapreduce.Mapper
- Type parameters: input key type, input value type, output key type, and output value type
  - Input key: line's offset in file (irrelevant)
  - Input value: line from NCDC file
  - Output key: year
  - Output value: temperature
- Data types are optimized for network serialization
  - Found in org.apache.hadoop.io package
- Work is done by the map() method

145

Map() Method

- Input: input key type, input value type (and a Context)
  - Line of text from NCDC file
  - Converted to Java String type, then parsed to get year and temperature
- Output: written using Context
  - Uses output key and value types
- Only write (year, temp) pair if the temperature is present and quality indicator reading is OK

146

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MaxTemperatureMapper
 extends Mapper<LongWritable, Text, Text, IntWritable> {

 private static final int MISSING = 9999;

 @Override
 public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

  String line = value.toString();
  String year = line.substring(15, 19);
  int airTemperature;
  if (line.charAt(87) == '+') {                    // parseInt doesn't like leading plus signs
   airTemperature = Integer.parseInt(line.substring(88, 92));
  } else {
   airTemperature = Integer.parseInt(line.substring(87, 92));
  }
  String quality = line.substring(92, 93);

  if (airTemperature != MISSING && quality.matches("[01459]")) {
   context.write(new Text(year), new IntWritable(airTemperature));
  }
 }
}
```

147

Reduce

- Implements org.apache.hadoop.mapreduce.Reducer
- Input key and value types must match Mapper output key and value types
- Work is done by reduce() method
  - Input values passed as Iterable list
  - Goes over all temperatures to find the max
  - Result pair is written by using the Context
    - Writes result to HDFS, Hadoop's distributed file system

148

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
 extends Reducer<Text, IntWritable, Text, IntWritable> {

 @Override
 public void reduce(Text key, Iterable<IntWritable> values, Context context)
   throws IOException, InterruptedException {

  int maxValue = Integer.MIN_VALUE;
  for (IntWritable value : values) {
   maxValue = Math.max(maxValue, value.get());
  }
  context.write(key, new IntWritable(maxValue));
 }
}
```

149

Job Configuration

- Job object forms the job specification and gives control for running the job
- Specify data input path using addInputPath()
  - Can be single file, directory (to use all files there), or file pattern
  - Can be called multiple times to add multiple paths
- Specify output path using setOutputPath()
  - Single output path, which is a directory for all output files
- Set mapper and reducer class to be used
- Set output key and value classes for map and reduce functions
  - For reducer: setOutputKeyClass(), setOutputValueClass()
  - For mapper (omit if same as reducer): setMapOutputKeyClass(), setMapOutputValueClass()
- Can set input types similarly (default is TextInputFormat)
- Method waitForCompletion() submits job and waits for it to finish

150

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemperature {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

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

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

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

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```


Extension: Combiner Functions

- Recall earlier discussion about combiner function
  - Pre-reduces mapper output before transfer to reducers
  - Does not change program semantics
- Usually (almost) same as reduce function, but has to have same output type as Map
- Works only for some reduce functions that can be incrementally computed
  - MAX(5, 4, 1, 2) = MAX(MAX(5, 1), MAX(4, 2))
  - Same for SUM, MIN, COUNT, AVG (=SUM/COUNT)


```
public class MaxTemperatureWithCombiner {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCombiner <input path> " + "<output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");

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

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

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

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Note: combiner here is identical to reducer class.

Extension: Custom Partitioner

- Partitioner determines which keys are assigned to which reduce task
- Default HashPartitioner essentially assigns keys randomly
- Create custom partitioner by implementing your own getPartition() method of Partitioner in org.apache.hadoop.mapreduce

MapReduce Development Steps

1. Write Map and Reduce functions
   - Create unit tests
2. Write driver program to run a job
   - Can run from IDE with small data subset for testing
   - If test fails, use IDE for debugging
   - Update unit tests and Map/Reduce if necessary
3. Once program works on small test set, run it on full data set
   - If there are problems, update tests and code accordingly
4. Fine-tune code, do some profiling

Local (Standalone) Mode

- Runs same MapReduce user program as cluster version, but does it sequentially
- Does not use any of the Hadoop daemons
- Works directly with local file system
  - No HDFS, hence no need to copy data to/from HDFS
- Great for development, testing, initial debugging

Pseudo-Distributed Mode

- Still runs on single machine, but now simulates a real Hadoop cluster
  - Simulates multiple nodes
  - Runs all daemons
  - Uses HDFS
- Main purpose: more advanced testing and debugging
- You can also set this up on your laptop

157

Extension: MapFile

- Sorted file of (key, value) pairs with an index for lookups by key
- Must append new entries in order
  - Can create MapFile by sorting SequenceFile
- Can get value for specific key by calling MapFile's get() method
  - Found by performing binary search on index
- Method getClosest() finds closest match to search key

158

Extension: Counters

- Useful to get statistics about the MapReduce job, e.g., how many records were discarded in Map
- Difficult to implement from scratch
  - Mappers and reducers need to communicate to compute a global counter
- Hadoop has built-in support for counters
- See ch. 8 in Tom White's book for details

159

Hadoop Job Tuning

- Choose appropriate number of mappers and reducers
- Define combiners whenever possible
  - But see also later discussion about local aggregation
- Consider Map output compression
- Optimize the expensive shuffle phase (between mappers and reducers) by setting its tuning parameters
- Profiling distributed MapReduce jobs is challenging.

160

Hadoop and Other Programming Languages

- Hadoop Streaming API to write map and reduce functions in languages other than Java
  - Any language that can read from standard input and write to standard output
- Hadoop Pipes API for using C++
  - Uses sockets to communicate with Hadoop's task trackers

161