CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Warehouse-Scale Computing, MapReduce, and Spark

Instructors:
Krste Asanović & Randy H. Katz
http://inst.eecs.berkeley.edu/~cs61c/

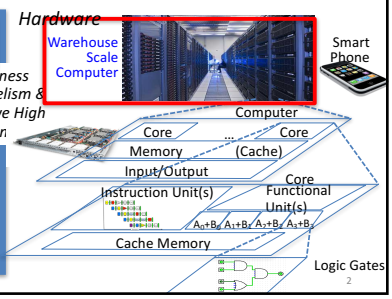
New-School Machine Structures

Software

- Parallel Requests
  Assigned to computer
  e.g., Search "Randy Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instruction
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Perforn

Hardware



Agenda

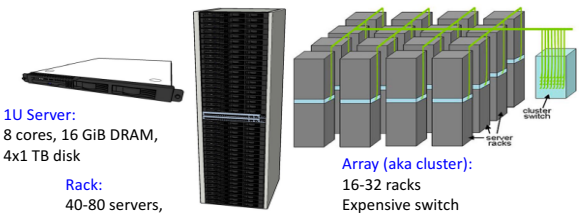
- Warehouse-Scale Computing
- Cloud Computing
- Request-Level Parallelism (RLP)
- Map-Reduce Data Parallelism
- And, in Conclusion …

Agenda

- Warehouse-Scale Computing
- Cloud Computing
- Request Level Parallelism (RLP)
- Map-Reduce Data Parallelism
- And, in Conclusion …

Google's WSCs

WSC Architecture



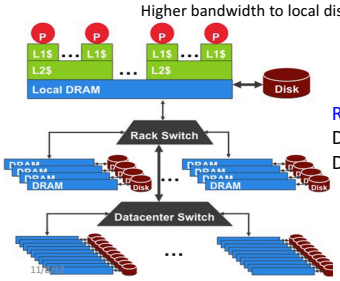
1U Server:
8 cores, 16 GiB DRAM,
4x1 TB disk

Rack:
40-80 servers,
Local Ethernet (1-10Gbps) switch
(30$/1Gbps/server)

Array (aka cluster):
16-32 racks
Expensive switch
(10X bandwidth → 100x cost)

WSC Storage Hierarchy

Lower latency to DRAM in another server than local disk
Higher bandwidth to local disk than to DRAM in another server

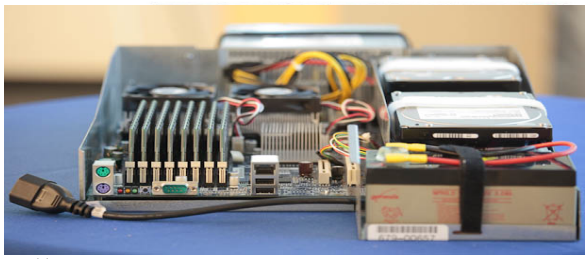


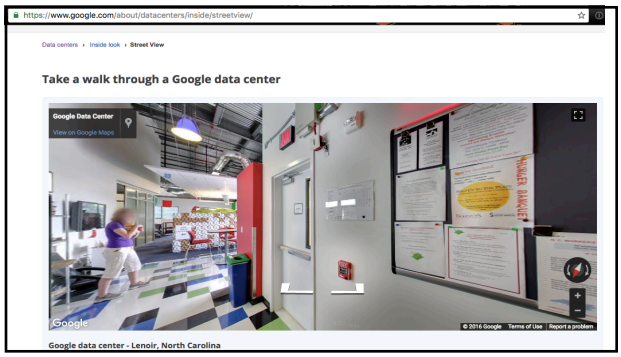
1U Server:
DRAM: 16GB, 100ns, 20GB/s
Disk: 2TB, 10ms, 200MB/s

Rack (80 severs):
DRAM: 1TB, 300µs, 100MB/s
Disk: 160TB, 11ms, 100MB/s

Array (30 racks):
DRAM: 30TB, 500µs, 10MB/s
Disk: 4.80PB, 12ms, 10MB/s

Google Server Internals





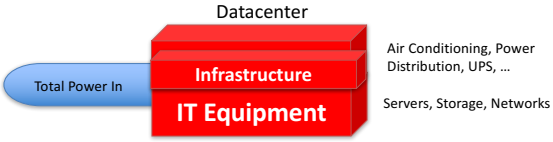
Power Usage Effectiveness

- Energy efficiency
  - Primary concern in the design of WSC
  - Important component of the total cost of ownership
- Power Usage Effectiveness (PUE):

$$\frac{\textbf{Total Building Power}}{\textbf{IT equipment Power}}$$

  - Power efficiency measure for WSC
  - Not considering efficiency of servers, networking
  - Perfection = 1.0
  - Google WSC's PUE = 1.2

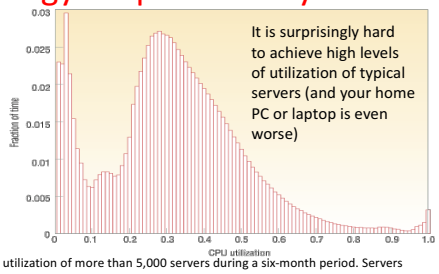
Power Usage Effectiveness



PUE = Total Power/IT Power
PUE = 2.5

Energy Proportionality

"The Case for Energy-Proportional Computing," Luiz André Barroso, Urs Hölzle, *IEEE Computer* December 2007



It is surprisingly hard to achieve high levels of utilization of typical servers (and your home PC or laptop is even worse)

Figure 1. Average CPU utilization of more than 5,000 servers during a six-month period. Servers are rarely completely idle and seldom operate near their maximum utilization, instead operating most of the time at between 10 and 50 percent of their maximum

Energy-Proportional Computing

"The Case for Energy-Proportional Computing," Luiz André Barroso, Urs Hölzle, *IEEE Computer* December 2007

Energy Efficiency = Utilization/Power

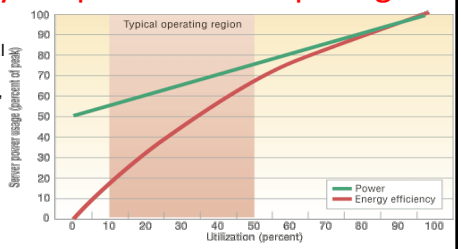


Figure 2. Server power usage and energy efficiency at varying utilization levels, from idle to peak performance. Even an energy-efficient server still consumes about half its full power when doing virtually no work.

13

Energy Proportionality

"The Case for Energy-Proportional Computing," Luiz André Barroso, Urs Hölzle, *IEEE Computer* December 2007

Energy Efficiency = Utilization/Power

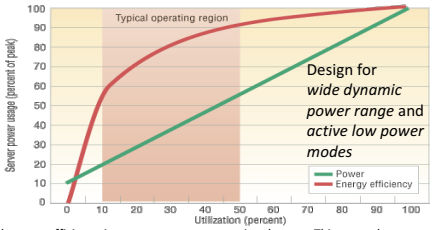


Figure 4. Power usage and energy efficiency in a more energy-proportional server. This server has a power efficiency of more than 80 percent of its peak value for utilizations of 30 percent and above, with efficiency remaining above 50 percent for utilization levels as low as 10 percent.

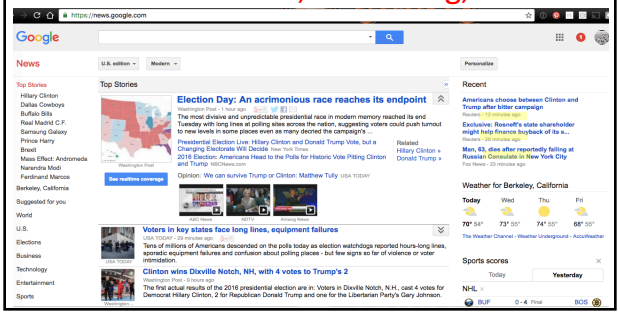
14

Agenda

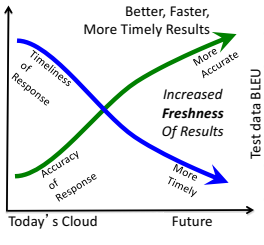
- Warehouse Scale Computing
- Cloud Computing
- Request Level Parallelism (RLP)
- Map-Reduce Data Parallelism
- And, in Conclusion …

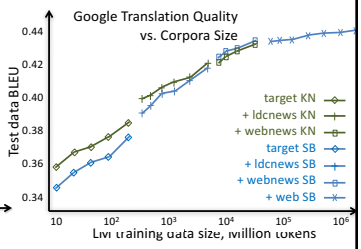
11/8/17 15

Scaled Communities, Processing, and Data



Quality and Freshness





11/8/17 17

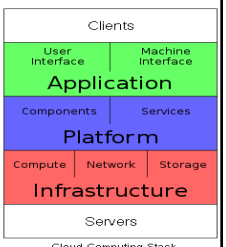
Cloud Distinguished by …

- Shared platform with illusion of isolation
  - Collocation with other tenants
  - Exploits technology of VMs and hypervisors (next lectures!)
  - At best "fair" allocation of resources, but not true isolation
- Attraction of low-cost cycles
  - Economies of scale driving move to consolidation
  - Statistical multiplexing to achieve high utilization/efficiency of resources
- Elastic service
  - Pay for what you need, get more when you need it
  - But no performance guarantees: assumes uncorrelated demand for resources

11/8/17 18

Cloud Services

- **SaaS**: deliver apps over Internet, eliminating need to install/run on customer's computers, simplifying maintenance and support
  - E.g., Google Docs, Win Apps in the Cloud
- **PaaS**: deliver computing "stack" as a service, using cloud infrastructure to implement apps. Deploy apps without cost/complexity of buying and managing underlying layers
  - E.g., Hadoop on EC2, Apache Spark on GCP
- **IaaS**: Rather than purchasing servers, software, data center space or net equipment, clients buy resources as an outsourced service. Billed on utility basis. Amount of resources consumed/cost reflect level of activity
  - E.g., Amazon Elastic Compute Cloud, Google Compute Platform



Cloud Computing Stack

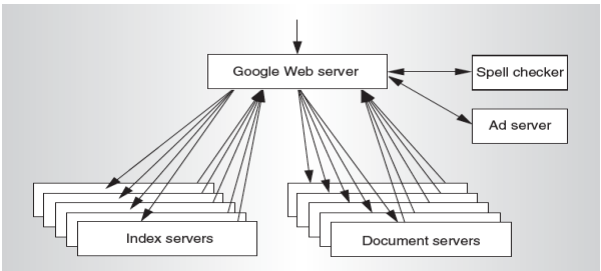
Agenda

- Warehouse Scale Computing
- Cloud Computing
- Request-Level Parallelism (RLP)
- Map-Reduce Data Parallelism
- And, in Conclusion …

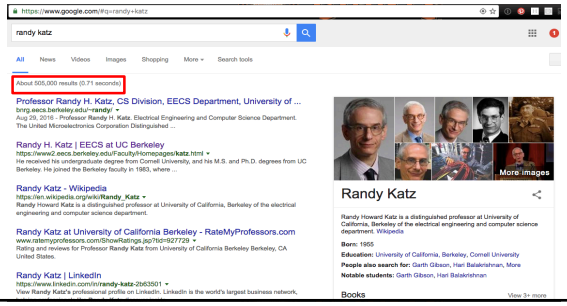
Request-Level Parallelism (RLP)

- Hundreds of thousands of requests per second
  - Popular Internet services like web search, social networking, …
  - Such requests are largely independent
    - Often involve read-mostly databases
    - Rarely involve read-write sharing or synchronization across requests
- Computation easily partitioned across different requests and even within a request

Google Query-Serving Architecture



Web Search Result



Anatomy of a Web Search (1/3)

- Google "Randy Katz"
  1. Direct request to "closest" Google Warehouse-Scale Computer
  2. Front-end load balancer directs request to one of many clusters of servers within WSC
  3. Within cluster, select one of many Google Web Servers (GWS) to handle the request and compose the response pages
  4. GWS communicates with Index Servers to find documents that contain the search words, "Randy", "Katz", uses location of search as well
  5. Return document list with associated relevance score

Anatomy of a Web Search (2/3)

- In parallel,
  - Ad system: books by Katz at Amazon.com
  - Images of Randy Katz
- Use docids (document IDs) to access indexed documents
- Compose the page
  - Result document extracts (with keyword in context) ordered by relevance score
  - Sponsored links (along the top) and advertisements (along the sides)

Anatomy of a Web Search (3/3)

- Implementation strategy
  - Randomly distribute the entries
  - Make many copies of data (aka "replicas")
  - Load balance requests across replicas
- ***Redundant copies*** of indices and documents
  - Breaks up hot spots, e.g., "Justin Bieber"
  - Increases opportunities for ***request-level parallelism***
  - Makes the system more ***tolerant of failures***

Administrivia

- Project 3: Performance has been released!
  - Due Monday, November 20
  - Compete in a performance context (Proj5) for extra credit!
- HW5 was released; due Wednesday Nov 15
- Regrade requests are open
  - Due this Friday (Nov 10)
  - **Please** consult the solutions and GradeScope rubric first

Agenda

- Warehouse Scale Computing
- Cloud Computing
- Request Level Parallelism (RLP)
- Map-Reduce Data Parallelism
- And, in Conclusion …

Data-Level Parallelism (DLP)

- SIMD
  - Supports data-level parallelism in a single machine
  - Additional instructions & hardware (e.g., AVX)

  e.g., Matrix multiplication in memory
- DLP on WSC
  - Supports data-level parallelism across ***multiple machines***
  - MapReduce & scalable file systems

Problem Statement

- How process large amounts of raw data (crawled documents, request logs, …) every day to compute derived data (inverted indices, page popularity, …) when computation conceptually simple but input data large and distributed across 100s to 1000s of servers so that finish in reasonable time?
- Challenge: Parallelize computation, distribute data, tolerate faults without obscuring simple computation with complex code to deal with issues

Solution: MapReduce

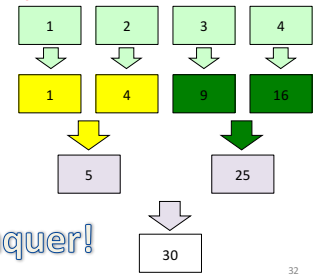
- Simple data-parallel ***programming model*** and ***implementation*** for processing large datasets
- Users specify the computation in terms of
  - a ***map*** function, and
  - a ***reduce*** function
- Underlying runtime system
  - Automatically ***parallelize*** the computation across large scale clusters of machines
  - ***Handles*** machine ***failure***
  - ***Schedule*** inter-machine communication to make efficient use of the networks

Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," 6th *USENIX Symposium on Operating Systems Design and Implementation, 2004*. (optional reading, linked on course homepage – a digestible CS paper at the 61C level)

Inspiration: Map & Reduce Functions, ex: Python

Calculate : $\sum_{n=1}^{4} n^2$

```
A = [1, 2, 3, 4]
def square(x):
  return x * x
def sum(x, y):
  return x + y
reduce(sum, map(square, A))
```



Divide and Conquer!

MapReduce Programming Model

- ***Map***: `(in_key, in_value)` → `list(interm_key, interm_val)`

```
map(in_key, in_val):
  // DO WORK HERE
  emit(interm_key,interm_val)
```

  - Slice data into "shards" or "splits" and distribute to workers
  - Compute set of intermediate key/value pairs
- ***Reduce***: `(interm_key, list(interm_value))` → `list(out_value)`

```
reduce(interm_key, list(interm_val)):
  // DO WORK HERE
  emit(out_key, out_val)
```

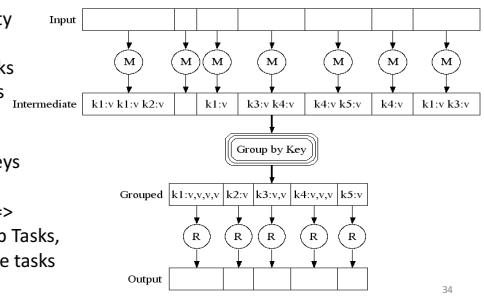
  - Combines all intermediate values for a particular key
  - Produces a set of merged output values (usually just one)

MapReduce Execution

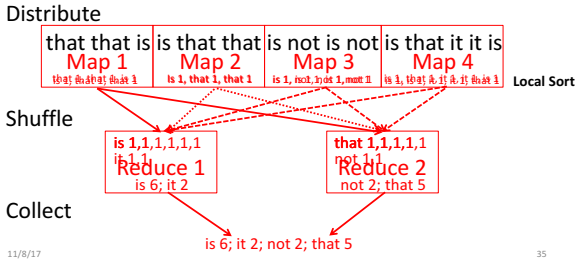
Fine granularity tasks: many more map tasks than machines

Bucket sort to get same keys together

2000 servers =>
≈ 200,000 Map Tasks,
≈ 5,000 Reduce tasks



MapReduce Word Count Example



MapReduce Word Count Example

Task of counting the number of occurrences of each word in a large collection of documents

User-written ***Map*** function reads the document data and parses the words. For each word, it writes the (key, value) pair of (word, 1). The word is treated as the intermediate key and the associated value of 1 means that we saw the word once.

Map phase: (doc name, doc contents) → list(word, count)

`// "I do I learn" → [("I",1),("do",1),("I",1),("learn",1)]`

```
map(key, value):
  for each word w in value:
    emit(w, 1)
```

MapReduce Word Count Example

Task of counting the number of occurrences of each word in a large collection of documents.

Intermediate data is then sorted by MapReduce by keys and the user's ***Reduce*** function is called for each unique key. In this case, Reduce is called with a list of a "1" for each occurrence of the word that was parsed from the document. The function adds them up to generate a total word count for that word.

Reduce phase: (word, list(counts)) → (word, count_sum)

```
// ("I", [1,1]) → ("I",2)
reduce(key, values):
  result = 0
  for each v in values:
    result += v
  emit(key, result)
```

37

38

The Combiner (Optional)

- One missing piece for our first example:
  - Many times, the output of a single mapper can be "compressed" to save on bandwidth and to distribute work (usually more map tasks than reduce tasks)
  - To implement this, we have the combiner:

```
combiner(interm_key,list(interm_val)):
    // DO WORK (usually like reducer)
    emit(interm_key2, interm_val2)
```

39

Our Final Execution Sequence

- Map – Apply operations to all input key, val
- Combine – Apply reducer operation, but distributed across map tasks
- Reduce – Combine all values of a key to produce desired output

40

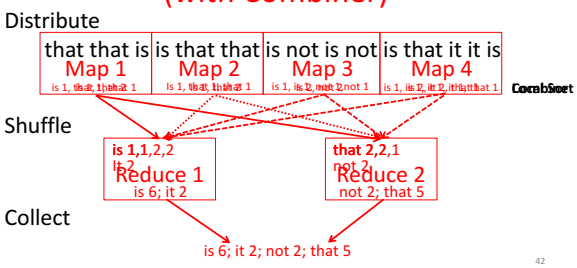
MapReduce Processing Example: Count Word Occurrences

- Pseudo Code: for each word in input, generate <key=word, value=1>
- Reduce sums all counts emitted for a particular word across all mappers

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1"); // Produce count of words
combiner: (same as below reducer)
reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // intermediate_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v); // get integer from key-value
  Emit(output_key, result);
```

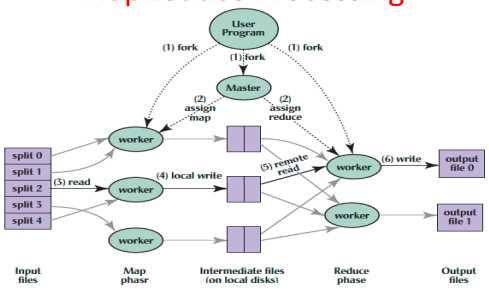
41

MapReduce Word Count Example (with Combiner)



42

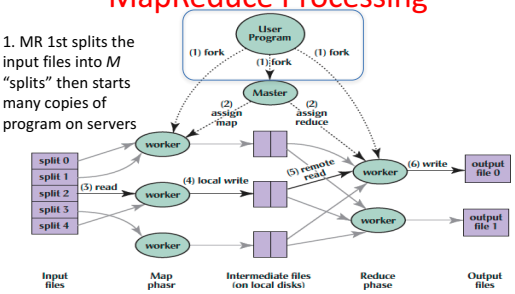
MapReduce Processing



Shuffle phase

MapReduce Processing

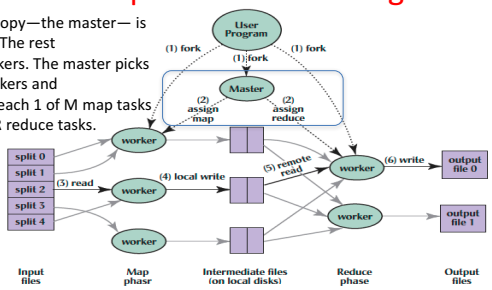
1. MR 1st splits the input files into *M* "splits" then starts many copies of program on servers



Shuffle phase

MapReduce Processing

2. One copy—the master— is special. The rest are workers. The master picks idle workers and assigns each 1 of M map tasks or 1 of R reduce tasks.

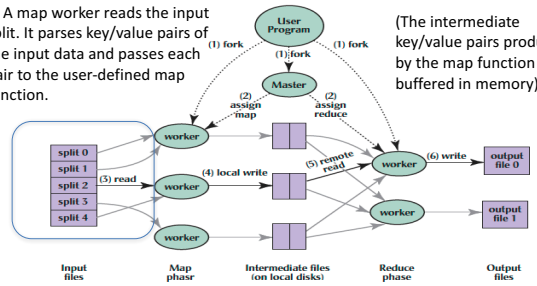


Shuffle phase

MapReduce Processing

3. A map worker reads the input split. It parses key/value pairs of the input data and passes each pair to the user-defined map function.

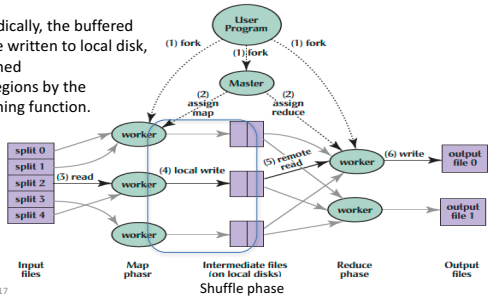
(The intermediate key/value pairs produced by the map function are buffered in memory)



Shuffle phase

MapReduce Processing

4. Periodically, the buffered pairs are written to local disk, partitioned into *R* regions by the partitioning function.

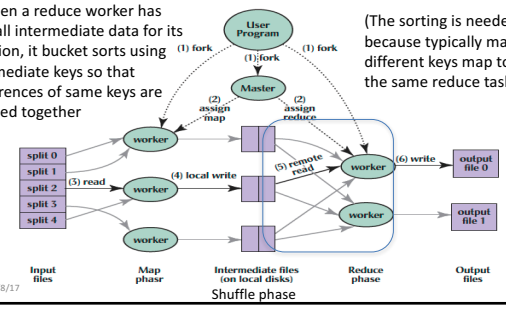


Shuffle phase

MapReduce Processing

5. When a reduce worker has read all intermediate data for its partition, it bucket sorts using intermediate keys so that occurrences of same keys are grouped together

(The sorting is needed because typically many different keys map to the same reduce task)

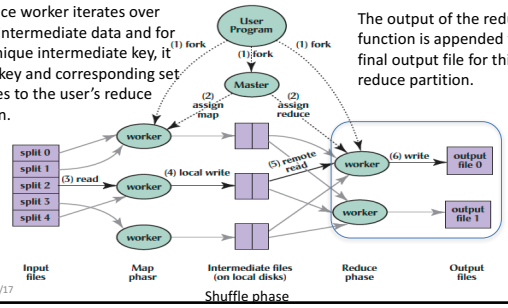


Shuffle phase

MapReduce Processing

6. Reduce worker iterates over sorted intermediate data and for each unique intermediate key, it passes key and corresponding set of values to the user's reduce function.

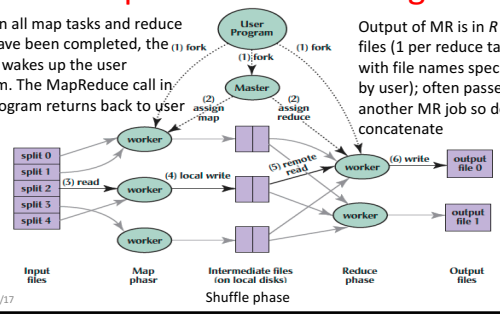
The output of the reduce function is appended to a final output file for this reduce partition.



MapReduce Processing

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. The MapReduce call in user program returns back to user code.

Output of MR is in *R* output files (1 per reduce task, with file names specified by user); often passed into another MR job so don't concatenate



Big Data Frameworks: Hadoop & Spark

- Apache Hadoop
  - Open-source MapReduce Framework
  - Hadoop Distributed File System (HDFS)
  - MapReduce Java APIs
- Apache Spark
  - Fast and general engine for large-scale data processing.
  - Originally developed in the AMP lab at UC Berkeley
  - Running on HDFS
  - Provides Java, Scala, Python APIs for
    - Database
    - Machine learning
    - Graph algorithm

WordCount in Hadoop's Java API

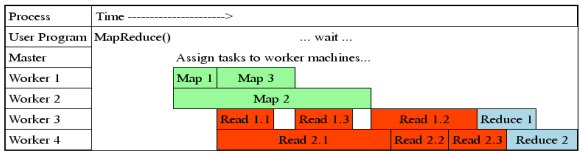
```
public static void main(String[] args) throws IOException {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(WCMap.class);
    conf.setCombinerClass(WCReduce.class);
    conf.setReducerClass(WCReduce.class);
    conf.setInputPath(new Path(args[0]));
    conf.setOutputPath(new Path(args[1]));
    JobClient.runJob(conf);
}

public class WCMap extends MapReduceBase implements Mapper {
    private static final IntWritable ONE = new IntWritable(1);
    public void map(WritableComparable key, Writable value,
                    OutputCollector output,
                    Reporter reporter) throws IOException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            output.collect(new Text(itr.next()), ONE);
        }
    }
}

public class WCReduce extends MapReduceBase implements Reducer {
    public void reduce(WritableComparable key, Iterator values,
                       OutputCollector output,
                       Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += ((IntWritable) values.next()).get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```


Word Count in Spark's Python API

```
// RDD: primary abstraction of a distributed collection of items
file = sc.textFile("hdfs://…")
// Two kinds of operations:
// Actions: RDD → Value
// Transformations: RDD → RDD
// e.g. flatMap, Map, reduceByKey
file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
```

See http://spark.apache.org/examples.html

MapReduce Processing Time Line



- Master assigns map + reduce tasks to "worker" servers
- As soon as a map task finishes, worker server can be assigned a new map or reduce task
- Data shuffle begins as soon as a given Map finishes
- Reduce task begins as soon as all data shuffles finish
- To tolerate faults, reassign task if a worker server "dies"

Show MapReduce Job Running

- ~41 minutes total
  - ~29 minutes for Map tasks & Shuffle tasks
  - ~12 minutes for Reduce tasks
  - 1707 worker servers used
- Map (Green) tasks read 0.8 TB, write 0.5 TB
- Shuffle (Red) tasks read 0.5 TB, write 0.5 TB
- Reduce (Blue) tasks read 0.5 TB, write 0.5 TB

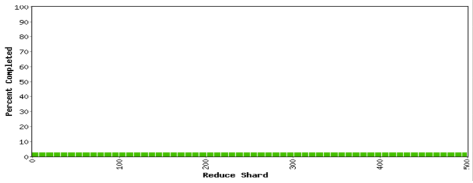
55

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 00 min 18 sec

323 workers; 0 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	0	323	878934.6	1314.4	717.0
Shuffle	500	0	323	717.0	0.0	0.0
Reduce	500	0	0	0.0	0.0	0.0



Counters

Variable	
Mapped (MB/s)	72.5
Shuffle (MB/s)	0.0
Output (MB/s)	0.0
doc-index-hits	145825686
docs-indexed	506631
dups-in-index-merge	0
mr-operator-calls	508192
mr-operator-	506631

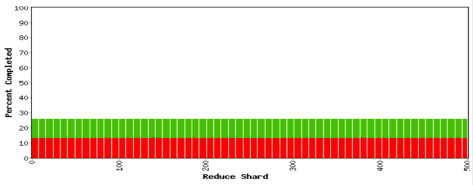
56

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 05 min 07 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	1857	1707	878934.6	191995.8	113936.6
Shuffle	500	0	500	113936.6	57113.7	57113.7
Reduce	500	0	0	57113.7	0.0	0.0



Counters

Variable	
Mapped (MB/s)	699.1
Shuffle (MB/s)	349.5
Output (MB/s)	0.0
doc-index-hits	5004411944
docs-indexed	17290135
dups-in-index-merge	0
mr-operator-calls	17331371
mr-operator-outputs	17290135

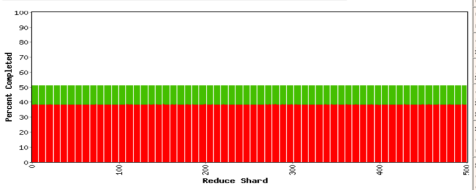
57

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 10 min 18 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	5354	1707	878934.6	406020.1	241058.2
Shuffle	500	0	500	241058.2	196362.5	196362.5
Reduce	500	0	0	196362.5	0.0	0.0



Counters

Variable	
Mapped (MB/s)	704.4
Shuffle (MB/s)	371.9
Output (MB/s)	0.0
doc-index-hits	5000364228
docs-indexed	17300709
dups-in-index-merge	0
mr-operator-calls	17342493
mr-operator-outputs	17300709

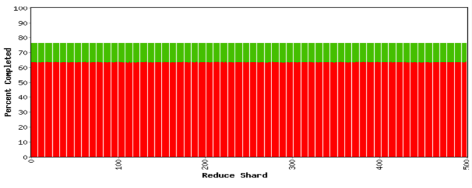
58

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 15 min 31 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	8841	1707	878934.6	621608.5	369459.8
Shuffle	500	0	500	369459.8	326986.8	326986.8
Reduce	500	0	0	326986.8	0.0	0.0



Counters

Variable	
Mapped (MB/s)	706.5
Shuffle (MB/s)	419.2
Output (MB/s)	0.0
doc-index-hits	4982870667
docs-indexed	17229926
dups-in-index-merge	0
mr-operator-calls	17272056
mr-operator-outputs	17229926

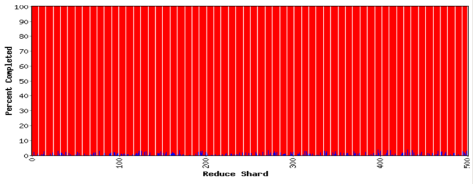
59

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 29 min 45 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	195	305	523499.2	523389.6	523389.6
Reduce	500	0	195	523389.6	2685.2	2742.6



Counters

Variable	
Mapped (MB/s)	0.3
Shuffle (MB/s)	0.5
Output (MB/s)	45.7
doc-index-hits	2313178
docs-indexed	7936
dups-in-index-merge	0
mr-merge-calls	1954105
mr-merge-outputs	1954105

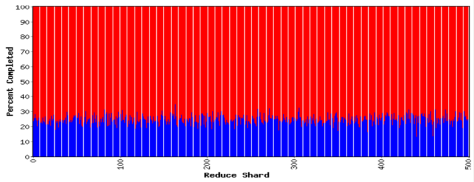
60

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 31 min 34 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	133837.8	136929.6



Counters

Variable	
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.1
Output (MB/s)	1238.8
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	51738599
mr-merge-outputs	51738599

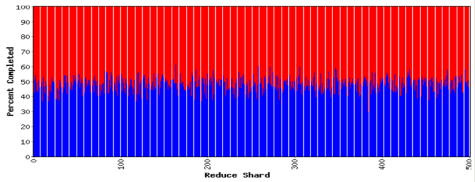
61

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 33 min 22 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	263283.3	269351.2



Counters

Variable	
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.0
Output (MB/s)	1225.1
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	51842100
mr-merge-outputs	51842100

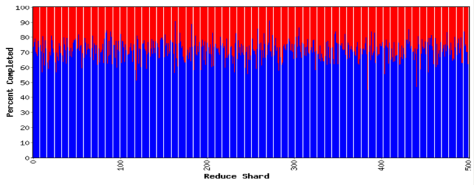
62

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 35 min 08 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	390447.6	399457.2



Counters

Variable	
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.0
Output (MB/s)	1222.0
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	51640600
mr-merge-outputs	51640600

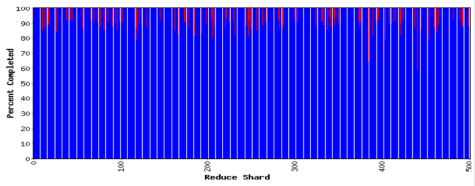
63

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 37 min 01 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	520468.6	520468.6
Reduce	500	406	94	520468.6	512265.2	514373.3



Counters

Variable	
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.0
Output (MB/s)	849.5
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	35083350
mr-merge-outputs	35083350

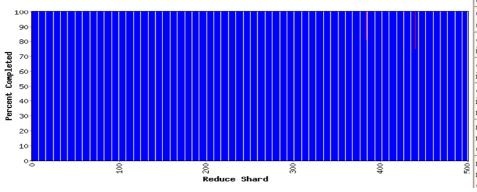
64

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 38 min 56 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	519781.8	519781.8
Reduce	500	498	2	519781.8	519394.7	519440.7



Counters

Variable	
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.0
Output (MB/s)	9.4
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	394792
mr-merge-outputs	394792

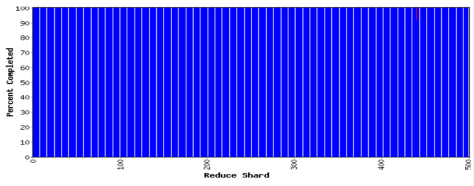
65

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 40 min 43 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	519774.3	519774.3
Reduce	500	499	1	519774.3	519735.2	519764.0



Counters

Variable	
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.0
Output (MB/s)	1.9
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	73442
mr-merge-outputs	73442

66

And, in Conclusion ...

- Warehouse-Scale Computers (WSCs)
  - New class of computers
  - Scalability, energy efficiency, high failure rate
- Cloud Computing
  - Benefits of WSC computing for third parties
  - "Elastic" pay as you go resource allocation
- Request-Level Parallelism
  - High request volume, each largely independent of other
  - Use replication for better request throughput, availability
- MapReduce Data Parallelism
  - Map: Divide large data set into pieces for independent parallel processing
  - Reduce: Combine and process intermediate results to obtain final result
  - Hadoop, Spark

67