



## UMD DATA605 - Big Data Systems

### 8.1: Cluster Architecture

- **Instructor:** Dr. GP Saggese - [gsaggese@umd.edu](mailto:gsaggese@umd.edu)

# MapReduce: Applications

---

- Major classes of applications
  - Text tokenization, indexing, and search
  - Processing of large data structures
  - Data mining and machine learning
  - Link analysis and graph processing

# Example: Language Model

---

- Statistical machine translation
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents
- Large Language Models
  - OpenAI GPT
- Very easy with MapReduce
  - Map
    - Extract (5-word sequence, count) from document
  - Reduce
    - Combine the counts

# Cost Measures for Distributed Algorithms

- **Quantify the cost of a parallel algorithm in terms of:**
- Communication cost
  - = total I/O of all processes
  - Related to disk usage as well
- Elapsed communication cost
  - = max I/O along any path (critical path)
- Elapsed computation cost
  - = end-to-end running time of algorithm
  - It is the wall-clock time using parallelism
- **Total cost**
  - = what you pay as rent to your “friendly” neighborhood cloud provider
  - CPU + disk + I/O used
  - Either CPU, disk, I/O cost dominates → ignore the others
- **In this case, the big-O notation is not the most useful**
  - The actual cost matters and not the asymptotic cost!
  - Multiplicative constant matters
  - Adding more machines is always an option

# MapReduce Cost Measures

---

- **For a 'map-reduce' algorithm:**
- **Communication cost**
  - = total I/O of all processes
  - input file size
  - $2 \times$  (sum of the sizes of all files passed from Map processes to Reduce processes) [You need to write and read back the data]
  - the sum of the output sizes of the Reduce processes
- **Elapsed communication cost**
  - = max of I/O along any path
  - sum of the largest input + output for any Map process, plus the same for any Reduce process
- **Elapsed computation cost**
  - = end-to-end running time of algorithm
  - Ideally all Map and Reduce processes end at the same time
  - Workload is "perfectly balanced"

# Example: Join By MapReduce

- Compute the natural join  $R(A,B) \bowtie S(B,C)$  joining on column **B**
- **R** and **S** are stored in files as pairs **(a, b)** or **(b, c)**
- Use a hash function  $h$  from B-values to  $h(b)$  in  $[1, \dots, k]$
- **Map task**
  - Transform an input tuple  $R(a, b)$  into key-value pair  $(h(b), (a, R))$
  - Each input tuple  $S(b, c) \rightarrow (h(b), (c, S))$
- **GroupBy task**
  - Each key-value pair with key  $b$  is sent to Reduce task  $h(b)$
  - Hadoop does this automatically; just tell it what  $h$  is
- **Reduce task**
  - Matches all the pairs  $(b, (a, R))$  with all  $(b, (c, S))$  to get  $(a, b, c)$
  - Output  $(a, c)$

R	A	B	$\bowtie$
	a1	b1	
	a2	b1	
	a3	b2	
S	a4	b3	=
	B	C	
	b2	c1	
	b2	c2	
	b3	c3	
	A	C	
	a3	c1	
	a3	c2	
	a4	c3	

# Cost of MapReduce Join

- **Total communication cost**

- = total I/O of all processes
- =  $O(|R| + |S| + |R \bowtie S|)$
- You need to read all the data and then write the result
- It doesn't matter how you split the computation

- **Elapsed communication cost**

- We put a limit  $s$  on the amount of input or output that any one process can have, e.g.,
  - What fits in main memory
  - What fits on local disk
- =  $O(s)$
- We're going to pick the number of Map and Reduce processes so that the I/O limit  $s$  is respected

- **Computation cost**

- =  $O(|R| + |S| + |R \bowtie S|)$
- Using proper indexes there is no shuffle
- So computation cost is like communication cost

# UMD DATA605 - Big Data System

---

- Storing and Computing Big Data
- MapReduce Framework
- (Apache) Hadoop
- Algorithms
- **MapReduce vs DBs**

# History

---

- Abstract ideas about MapReduce have been known before Google's MapReduce paper
- **The strength of MapReduce comes from simplicity, ease of use, and performance**
  - Declarative design
  - User specifies what is to be done, not how many machines to use, etc. . .
  - Many times commercial success comes from making something simple to use
- MapReduce can be implemented using user-defined aggregates in PostgreSQL quite easily
  - See MapReduce and Parallel DBMSs by Stonebraker et al., 2010
- No database system can come close to the performance of MapReduce infrastructure
  - E.g., RDBMSs
  - Can't scale to that degree
  - Are not as fault-tolerant
  - Designed to support ACID
    - Most MapReduce applications don't care about ACID consistency

# History

---

- **MapReduce**
- Is very good at doing what it was designed for
  - If the application maps well to MapReduce, one can achieve optimal theoretical speed-up
- May not be ideal for more complex tasks
  - E.g., no notion of “query optimization”, e.g., operator order optimization
  - The sequence of MapReduce tasks makes it procedural within a single machine
- Assumes a single input
  - E.g., joins are tricky to do, but doable
- Much work in recent years on extending the basic MapReduce functionality, e.g.,
  - Hadoop Zoo
  - E.g., Spark, Dask, Ray

# Hadoop Ecosystem (aka Hadoop Zoo)

- Pig
  - High-level data-flow language and execution framework for parallel computation
- HBase
  - Scalable, distributed database
  - Supports structured data storage for large tables (like Google BigTable)
- Cassandra
  - Scalable multi-master database with no single points of failure
- Hive
  - Data warehouse infrastructure
  - Provide data summarization and ad-hoc querying
- ZooKeeper High-performance coordination service for distributed applications
- YARN, Kafka, Storm, Spark, Solr,

