CS 61C: Great Ideas in Computer Architecture

MapReduce

Guest Lecturer: Justin Hsia

Review of Last Lecture

- Performance – latency and throughput
- Warehouse Scale Computing
  - Example of parallel processing in the post-PC era
  - Servers on a rack, rack part of cluster
  - Issues to handle include load balancing, failures, power usage (sensitive to cost & energy efficiency)
  - PUE = Total building power / IT equipment power

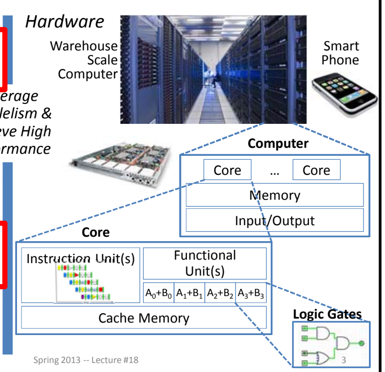
Great Idea #4: Parallelism

Today's Lecture

Software

- Parallel Requests
  Assigned to computer
  e.g. Search "Garcia"
- Parallel Threads
  Assigned to core
  e.g. Lookup, Ads
- Parallel Instructions
  > 1 instruction @ one time
  e.g. 5 pipelined instructions
- Parallel Data
  > 1 data item @ one time
  e.g. add of 4 pairs of words
- Hardware descriptions
  All gates functioning in parallel at same time

Leverage Parallelism & Achieve High Performance



Agenda

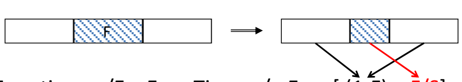
- Amdahl's Law
- Request Level Parallelism
- Administrivia
- MapReduce
  - Data Level Parallelism

Amdahl's (Heartbreaking) Law

- Speedup due to enhancement E:

$$\text{Speedup w/E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/E}}$$

- **Example:** Suppose that enhancement E accelerates a fraction F (F<1) of the task by a factor S (S>1) and the remainder of the task is unaffected



- Exec time w/E = Exec Time w/o E × [(1-F) + F/S]

  Speedup w/E = 1 / [(1-F) + F/S]

Amdahl's Law

- Speedup = $\dfrac{1}{(1 - F) + \dfrac{F}{S}}$

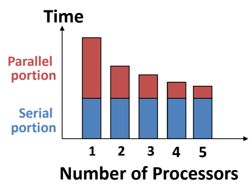
  Non-sped-up part: $(1 - F)$ Sped-up part: $\dfrac{F}{S}$

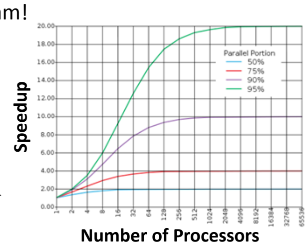
- **Example:** the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

$$\frac{1}{0.5 + \frac{0.5}{2}} = \frac{1}{0.5 + 0.25} = 1.33$$

Consequence of Amdahl's Law

- The amount of speedup that can be achieved through parallelism is limited by the non-parallel portion of your program!





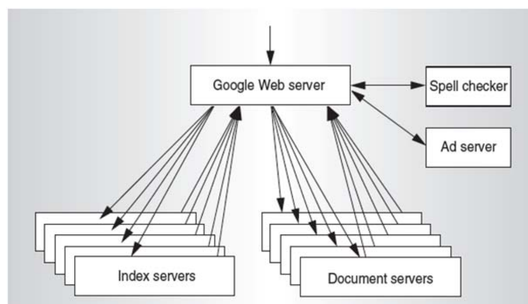
Agenda

- Amdahl's Law
- Request Level Parallelism
- Administrivia
- MapReduce
  - Data Level Parallelism

Request-Level Parallelism (RLP)

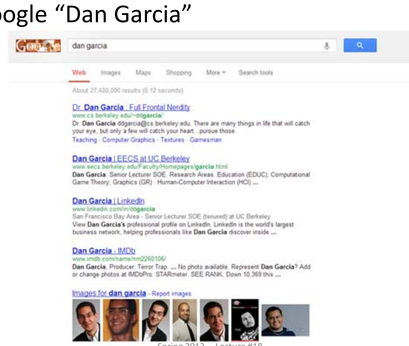
- Hundreds or thousands of requests per second
  - Not your laptop or cell-phone, but popular Internet services like web search, social networking, …
  - Such requests are largely independent
    - Often involve read-mostly databases
    - Rarely involve strict read–write data sharing or synchronization across requests
- Computation easily partitioned within a request and across different requests

Google Query-Serving Architecture



Anatomy of a Web Search

- Google "Dan Garcia"



Anatomy of a Web Search (1 of 3)

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

Anatomy of a Web Search (2 of 3)

- In parallel,
  - Ad system: run ad auction for bidders on search terms
  - Get images of various Dan Garcias
- 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 of 3)

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

Agenda

- Amdahl's Law
- Request Level Parallelism
- Administrivia
- MapReduce
  - Data Level Parallelism

Administrivia

- Midterm not graded yet
  - Please don't discuss anywhere until tomorrow!
- Lab 6 is today and tomorrow
- HW3 due this Sunday (3/10)
  - Finish early because Proj2 is being released this week!
- Twitter Tech Talk on Hadoop/MapReduce
  - Thu, 3/7 at 6pm in the Woz (430 Soda)

Agenda

- Amdahl's Law
- Request Level Parallelism
- Administrivia
- MapReduce
  - Data Level Parallelism

Data-Level Parallelism (DLP)

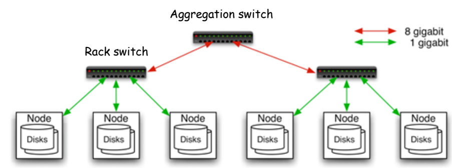
- Two kinds:
  1) Lots of data on many disks that can be operated on in parallel (e.g. searching for documents)
  2) Lots of data in memory that can be operated on in parallel (e.g. adding together 2 arrays)

1) Lab 6 and Project 2 do DLP across many servers and disks using MapReduce
2) Lab 7 and Project 3 do DLP in memory using Intel's SIMD instructions

What is MapReduce?

- Simple data-parallel programming model designed for scalability and fault-tolerance
- Pioneered by Google
  - Processes > 25 petabytes of data per day
- Popularized by open-source Hadoop project
  - Used at Yahoo!, Facebook, Amazon, …

Typical Hadoop Cluster

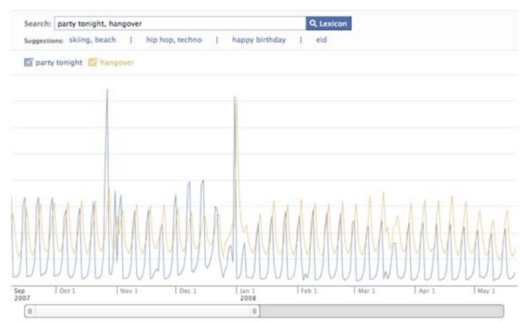


- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- Node specs (Yahoo terasort):
  8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

What is MapReduce used for?

- At Google:
  - Index construction for Google Search
  - Article clustering for Google News
  - Statistical machine translation
  - For computing multi-layer street maps
- At Yahoo!:
  - "Web map" powering Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

Example: Facebook Lexicon



www.facebook.com/lexicon (no longer available)

MapReduce Design Goals

1. **Scalability to large data volumes:**
   - 1000's of machines, 10,000's of disks
2. **Cost-efficiency:**
   - Commodity machines (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance (fewer administrators)
   - Easy to use (fewer programmers)

Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, Jan 2008.

MapReduce Processing: "Divide and Conquer" (1/2)

- Apply Map function to user supplied record of key/value pairs
  - Slice data into "shards" or "splits" and distribute to workers
  - Compute set of intermediate key/value pairs
  - `map(in_key,in_val) -> list(out_key,interm_val)`
- Apply Reduce operation to all values that share same key in order to combine derived data properly
  - Combines all intermediate values for a particular key
  - Produces a set of merged output values
  - `reduce(out_key,list(interm_val)) -> list(out_val)`

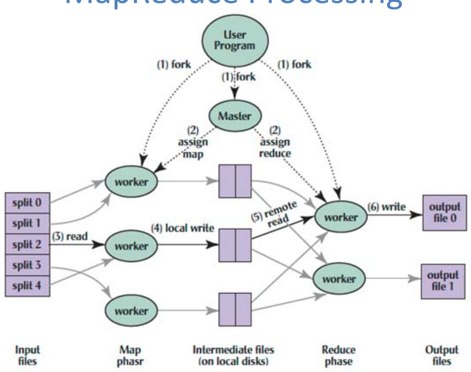
MapReduce Processing: "Divide and Conquer" (2/2)

- User supplies Map and Reduce operations in functional model
  - Focus on problem, let MapReduce library deal with messy details
  - Parallelization handled by framework/library
  - Fault tolerance via re-execution

Execution Setup

- Map invocations distributed by partitioning input data into M *splits*
  - Typically 16 MB to 64 MB per piece
- Input processed in parallel on different servers
- Reduce invocations distributed by partitioning intermediate key space into R pieces
  - e.g. hash(key) mod R
- User picks M >> # servers, R > # servers
  - Big M helps with load balancing, recovery from failure
  - One output file per R invocation, so not too many

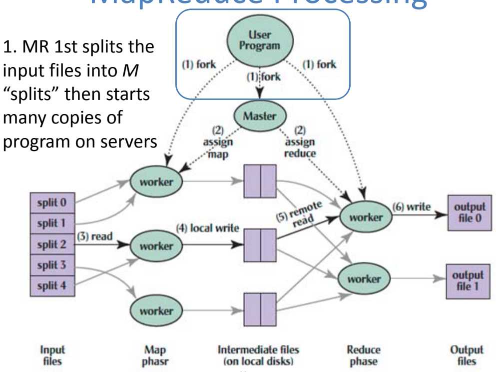
MapReduce Processing



Shuffle phase

MapReduce Processing

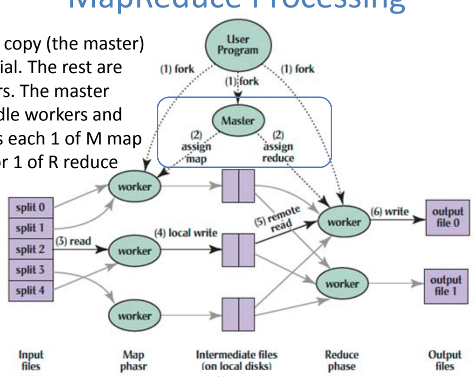
1. MR 1st splits the input files into *M* "splits" then starts many copies of program on servers



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.

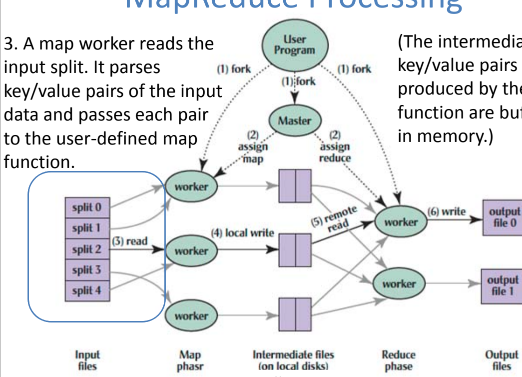


Shuffle phase

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.

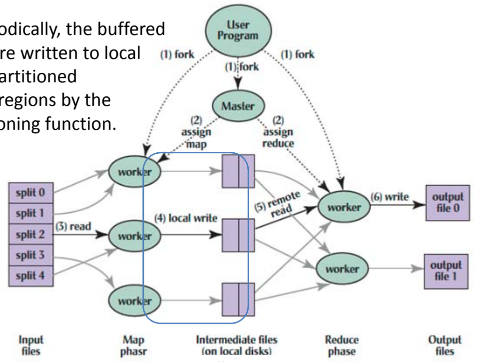
(The intermediate key/value pairs produced by the map function are buffered in memory.)



Shuffle phase

MapReduce Processing

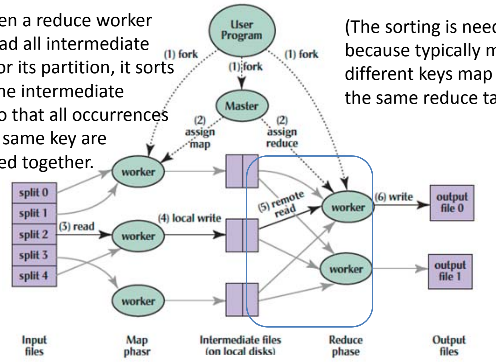
4. Periodically, the buffered pairs are written to local disk, partitioned into *R* regions by the partitioning function.



MapReduce Processing

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

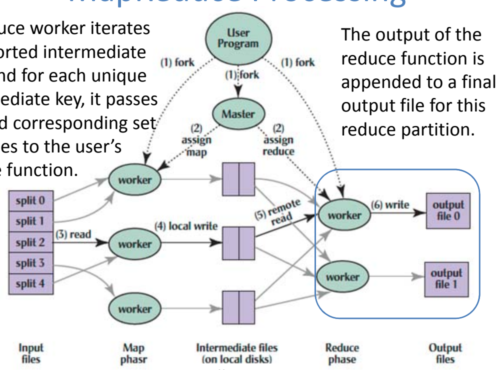
(The sorting is needed because typically many different keys map to the same reduce task)



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.

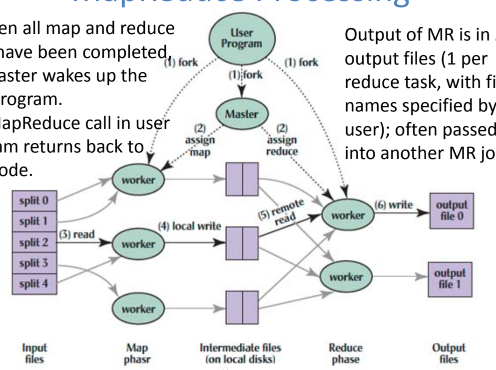
The output of the reduce function is appended to a final output file for this reduce partition.



MapReduce Processing

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

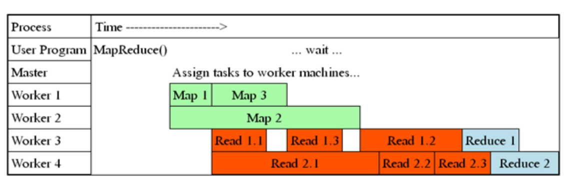
Output of MR is in *R* output files (1 per reduce task, with file names specified by user); often passed into another MR job.



What Does the Master Do?

- For each map task and reduce task
  - State: idle, in-progress, or completed
  - Identity of worker server (if not idle)
- For each completed map task
  - Stores location and size of R intermediate files
  - Updates files and size as corresponding map tasks complete
- Location and size are pushed incrementally to workers that have in-progress reduce tasks

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"

MapReduce Processing Example: Count Word Occurrences (1/2)

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

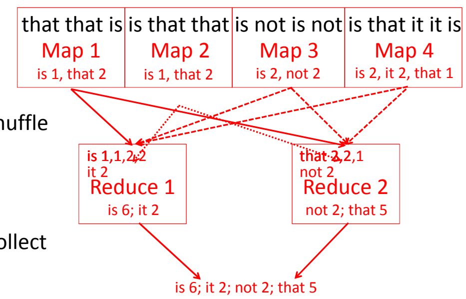
```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1"); // Produce count of words

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(AsString(result));
```


MapReduce Processing Example: Count Word Occurrences (2/2)

Distribute

Shuffle

Collect



MapReduce Failure Handling

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Master failure:
  - Protocols exist to handle (master failure unlikely)
- Robust: lost 1600 of 1800 machines once, but finished fine

MapReduce Redundant Execution

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time
  - 3% more resources, large tasks 30% faster

Question: Which statements are NOT TRUE about about MapReduce?

a) MapReduce divides computers into 1 master and N-1 workers; masters assigns MR tasks

b) Towards the end, the master assigns uncompleted tasks again; 1st to finish wins

c) Reducers can start reducing as soon as they start to receive Map data

d) Reduce worker sorts by intermediate keys to group all occurrences of same key

Question: Which statements are NOT TRUE about about MapReduce?

a) MapReduce divides computers into 1 master and N-1 workers; masters assigns MR tasks

b) Towards the end, the master assigns uncompleted tasks again; 1st to finish wins

c) Reducers can start reducing as soon as they start to receive Map data

d) Reduce worker sorts by intermediate keys to group all occurrences of same key

Summary

- Amdahl's Law
- Request Level Parallelism
  - High request volume, each largely independent
  - Replication for better throughput, availability
- Map Reduce Data Parallelism
  - Divide large data set into pieces for independent parallel processing
  - Combine and process intermediate results to obtain final result