Principles of Software Construction

MapReduce

Josh Bloch    Charlie Garrod
Administrivia

• Charlie and his wife Jess had a baby girl!
  – Her name is Elowen
• Homework 5c due tonight
  – But you can turn it in until Friday
• Homework 6 coming soon to an Internet near you
  – But not until after Thanksgiving
• Have a happy and relaxing Thanksgiving
Key concepts from Thursday

• Network programming in Java is pretty straightforward
  – Key classes for TCP: InetAddress, Socket, ServerSocket
• Distributed systems provide scalability and reliability
• But they also provide complexity and headaches
Ancient history of computing

- 1950s – 1960s – Batch Processing – Not interactive, fixed lifetime
- Late-1960s – 1970s – Time-sharing – Interactive, arbitrary lifetime
- Interactivity was a major advance
  - Led to PC revolution, which led to Internet revolution
- But batch processing never went away
  - The original hardware model persisted into the 1980s
  - Computational paradigm is still an important niche
Necessity is the mother of invention

- In Google's early days, hundreds of **big-data batch computations**
  - Data sets – crawled web documents, web request logs, etc.
  - Computations – inverted indices, page rank, query popularity, etc.
  - A critical part of the *indexing pipeline*
- Parallelism required to finish in a reasonable amount of time
- Each computation had its own program
- Hard parts:
  - Distribute data
  - Parallelize computations
  - Balance load
  - Tolerate errors (Google used cheap commodity hardware)
- Obscured simplicity of underlying computation
Invention

• Jeff Dean & Sanjay Ghemawat realized problems had common structure
• They wrote framework that captured structure & automated hard parts
  – Data parallelization, fault tolerance, data distribution, load balancing, etc.
• The basic abstraction:
  1. Perform the same computation on all key-value pairs in parallel (Map)
  2. Merge the results using a specified computation (Reduce)
• Concepts borrowed from functional programming
  – Functional nature of computation vastly simplified fault tolerance
• And MapReduce was born
MapReduce

A robust, scalable framework for distributed computation on replicated, partitioned data
Map from a functional perspective

- \( \text{map}(f, x[0...n-1]) \)
  - Apply the function \( f \) to each element of list \( x \)

- e.g., in Python:
  
  ```python
  def square(x): return x*x
  map(square, [1, 2, 3, 4]) would return [1, 4, 9, 16]
  ```

- Parallel map implementation is trivial
  - What is the work? What is the depth (span)?
Reduce from a functional perspective

• `reduce(f, x[0...n-1])`
  
  – Repeatedly apply binary function `f` to pairs of items in `x`, replacing the pair of items with the result until only one item remains
  
  – One sequential Python implementation:
    
    ```python
    def reduce(f, x):
      if len(x) == 1: return x[0]
      return reduce(f, [f(x[0],x[1])] + x[2:])
    ```
    
    – e.g., in Python:
    
    ```python
def add(x,y): return x+y
def reduce(add, [1,2,3,4])
  would return 10 as
  reduce(add, [1,2,3,4])
  reduce(add, [3,3,4])
  reduce(add, [6,4])
  reduce(add, [10]) -> 10
    ```
Reduce with an associative binary function

- If the function $\ell$ is associative, the order $\ell$ is applied does not affect the result

$$1 + ((2+3) + 4) \quad 1 + (2 + (3+4)) \quad (1+2) + (3+4)$$

- Parallel reduce implementation is also easy
  - What is the work? What is the depth (span)?
Distributed MapReduce

• Distributed MapReduce is similar to (but not the same as!):
  \[ \text{reduce}(f2, \text{map}(f1, x)) \]

• Key idea: “data-centric” architecture
  – Send function \( f1 \) directly to the data
    • Execute it concurrently
  – Then merge results with reduce
    • Also concurrently

```plaintext
Worker 1
{a-c: [2],
d-g: [3,4],
h-j: [3],
k-z: [1]}

Worker 2
{a-c: {alice: 90,
       bob: 42,
       cohen: 9}}

Worker 3
{d-g: {deb: 16},
h-j: {}}

Master
{k-z: {pete: 12,
       reif: 42}}
```
MapReduce with key/value pairs (Google style)

• Master
  – Assign tasks to workers
  – Ping workers to test for failures

• Map workers
  – Map for each key/value pair
  – Emit intermediate key/value pairs

• Reduce workers
  – Sort data by intermediate key and aggregate by key
  – Reduce for each key
MapReduce with key/value pairs (Google style)

- For each word on the Web, count the number of occurrences
  - For Map: key1 is a document name, value is its contents
  - For Reduce: key2 is a word, values is a list of the number of counts of that word

f1(String key1, String value):
  for each word w in value:
    EmitIntermediate(w, 1);

f2(String key2, Iterator values):
  int result = 0;
  for each v in values:
    result += v;
  Emit(key2, result);

Map: (key1, v1) → (key2, v2)*
Reduce: (key2, v2*) → (key3, v3)*
MapReduce: (key1, v1)* → (key3, v3)*

MapReduce: (docName, docText)* → (word, wordCount)*
MapReduce architectural details

- Usually integrated with distributed storage system
  - Map worker executes function on its share of the data
- Map output usually written to worker's local disk
  - Shuffle: reduce worker often pulls intermediate data from map worker's local disk
- Reduce output usually written back to distributed storage system
Handling server failures with MapReduce

- **Map worker failure:**
  - Re-map using replica of the storage system data

- **Reduce worker failure:**
  - New reduce worker can pull intermediate data from map worker's local disk, re-reduce

- **Master failure:**
  - Options:
    - Restart system using new master
    - Replicate master
The beauty of MapReduce

• Low communication costs (usually)
  – The shuffle (between map and reduce) can be expensive

• MapReduce can be iterated
  – Input to MapReduce:  key/value pairs in the distributed storage system
  – Output from MapReduce:  key/value pairs in the distributed storage system
MapReduce to count mutual friends

- For each pair of people in a social network, count mutual friends
  - For Map: key1 is a person, value is the list of their friends
  - For Reduce: key2 is ???, values is a list of ???

\[
\begin{align*}
\text{f1}(\text{String key1, String value}): & \quad \text{f2}(\text{String key2, Iterator values}): \\
\end{align*}
\]

MapReduce: \((\text{person, friends})* \rightarrow (\text{pair of people, count of mutual friends})*\)
MapReduce to count mutual friends

- For each pair of people in a social network, count mutual friends
  - For Map: `key1` is a person, `value` is the list of their friends
  - For Reduce: `key2` is a pair of people, `values` is a list of 1s, for each mutual friend that pair has

```java
f1(String key1, String value):
    for each pair of friends in value:
        EmitIntermediate(pair, 1);

f2(String key2, Iterator values):
    int result = 0;
    for each v in values:
        result += v;
    Emit(key2, result);
```

MapReduce: `(person, friends)* \rightarrow (pair of people, count of mutual friends)*`
MapReduce to count incoming links

- For each page on Web, count number of pages that link to it
  - For Map: key1 is a document name, value is the contents of that document
  - For Reduce: key2 is ???, values is a list of ???

\[
f_1(String\ \text{key1},\ String\ \text{value}):\quad f_2(String\ \text{key2},\ Iterator\ \text{values}):\]

MapReduce: \((doc\text{Name},\ doc\text{Text})* \rightarrow (doc\text{Name},\ \text{number of incoming links})*\)
MapReduce to count incoming links

- For each page on Web, count number of pages that link to it
  - For Map: `key1` is a document name, `value` is the contents of that document
  - For Reduce: `key2` is link, `values` is a list of 1s

```java
f1(String key1, String value):
  for each link in value:
    EmitIntermediate(link, 1)

f2(String key2, Iterator values):
  int result = 0;
  for each v in values:
    result += v;
  Emit(key2, result);
```

MapReduce: `(docName, docText)* \rightarrow (docName, number of incoming links)*`
MapReduce to create an inverted index

- For each page on the Web, list the pages that link to it
  - For Map: `key1` is a document name, `value` is the contents of that document
  - For Reduce: `key2` is `???`, `values` is a list of `???

```java
f1(String key1, String value):
    for each link in value:
        EmitIntermediate(link, key1)

f2(String key2, Iterator values):
    Emit(key2, values)
```

MapReduce: `(docName, docText)* → (docName, list of incoming links)*`
List the mutual friends

- For each pair of people in a social network, list mutual friends
  - For Map: key1 is a person, value is the list of their friends
  - For Reduce: key2 is ???, values is a list of ???

f1(String key1, String value):

f2(String key2, Iterator values):

MapReduce: (person, friends)* \rightarrow (pair of people, list of mutual friends)*
List the mutual friends

- For each pair of people in a social network, list mutual friends
  - For Map: key1 is a person, value is the list of their friends
  - For Reduce: key2 is a pair of people, values is a list of their mutual friends

\[
\text{MapReduce: } (\text{person, friends})^* \rightarrow (\text{pair of people, list of mutual friends})^*
\]
Count friends + friends of friends

- For each person in a social network, count their friends and friends of friends
  - For Map: key1 is a person, value is the list of their friends
  - For Reduce: key2 is ???, values is a list of ???

\[
f1(String\ key1,\ String\ value): \quad f2(String\ key2,\ Iterator\ values):\]

MapReduce: (person, friends)* \to (person, count of f + fof)*
Count friends + friends of friends

- For each person in a social network, count their friends and friends of friends
  - For Map: `key1` is a person, `value` is the list of their friends
  - For Reduce: `key2` is ???, `values` is a list of ???

\[
\text{f1(} \text{String key1, String value):} \\
\text{for each friend1 in value:} \\
\quad \text{EmitIntermediate(friend1, key1)} \\
\text{for each friend2 in value:} \\
\quad \text{EmitIntermediate(friend1, friend2);} \\
\]

\[
\text{f2(} \text{String key2, Iterator values):} \\
\quad \text{distinct_values = {}} \\
\text{for each v in values:} \\
\quad \text{if not v in distinct_values:} \\
\quad \quad \text{distinct_values.insert(v)} \\
\quad \text{Emit(key2, len(distinct_values))} \\
\]

MapReduce: \((\text{person, friends})^* \rightarrow (\text{person, count of f + fof})^*\)
Friends + friends of friends + friends of friends of friends

- For each person in a social network, count their friends and friends of friends and friends of friends of friends
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is ???, values is a list of ???

\[ f1(String\ key1, \ String\ value) : f2(String\ key2, \ Iterator\ values) : \]

MapReduce: (person, friends)* → (person, count of f + fof + fofof)*
Problem: How to reach distance 3 nodes?

• Solution: Iterative MapReduce
  – Use MapReduce to get distance 1 and distance 2 nodes
  – Feed results as input to a second MapReduce process

• Also consider:
  – Breadth-first search
  – PageRank
Dataflow processing

• High-level languages and systems for complex MapReduce-like processing
  – Google – FlumeJava, Millwheel
  – Yahoo – Pig, Hive
  – Microsoft – Dryad, Naiad
Postscript

• MapReduce paradigm has become pervasive over the past decade
  – Hadoop is best known implementation
• But MapReduce is no longer used in Google’s indexing pipeline!
• Why?
Summary

- MapReduce is a powerful distributed processing frameworks
- Makes it easy to do fault-tolerant web-scale data processing
- Hides all the hard parts – programming model is simple
- Paradigm has become pervasive
- You’ll get a chance to implement it in yourself Homework 6