What is Functional Programming?

Opinions differ, and it is difficult to give a precise definition, but generally speaking:

- Functional programming is style of programming in which the basic method of computation is the application of functions to arguments;
- A functional language is one that supports and encourages the functional style.

Example

Summing the integers 1 to 10 in Java:

```java
int total = 0;
for (int i = 1; i <= 10; ++i)
    total = total + i;
```

The computation method is variable assignment.

Example

Summing the integers 1 to 10 in Haskell:

```haskell
sum [1..10]
```

The computation method is function application.

Why is it Useful?

Again, there are many possible answers to this question, but generally speaking:

- The abstract nature of functional programming leads to considerably simpler programs;
- It also supports a number of powerful new ways to structure and reason about programs.
What is Hugs?

• An interpreter for Haskell, and the most widely used implementation of the language;

• An interactive system, which is well-suited for teaching and prototyping purposes;

• Hugs is freely available from: www.haskell.org/hugs

The Standard Prelude

When Hugs is started it first loads the library file Prelude.hs, and then repeatedly prompts the user for an expression to be evaluated.

For example:

> 2+3*4
14

> (2+3)*4
20

The standard prelude also provides many useful functions that operate on lists. For example:

> length [1,2,3,4]
4

> product [1,2,3,4]
24

> take 3 [1,2,3,4,5]
[1,2,3]

Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

\[ f(a,b) + c \cdot d \]

As previously, but in Haskell syntax.

In Haskell, function application is denoted using space, and multiplication is denoted using *.

\( f \ a \ b \ + \ c \* d \)

Apply the function \( f \) to \( a \) and \( b \), and add the result to the product of \( c \) and \( d \).

Functional Programming Review

• Functional operations do not modify data structures: They always create new ones

• Original data still exists in unmodified form

• Data flows are implicit in program design

• Order of operations does not matter
Functional Programming Review

fun foo(l: int list) =
  sum(l) + mul(l) + length(l)

• Order of sum() and mul(), etc does not matter
• They do not modify \( l \)

Functional Updates Do Not Modify Structures

fun append(x, lst) =
  let lst' = reverse lst in reverse (x :: lst')

The append() function above reverses a list, adds a new element to the front, and returns all of that, reversed, which appends an item.

But it never modifies \( lst! \)

Functions Can Be Used As Arguments

fun DoDouble(f, x) = f (f x)

It does not matter what \( f \) does to its argument; DoDouble() will do it twice.

A function is called higher-order if it takes a function as an argument or returns a function as a result

Map

map f lst: ('a->'b) -> ('a list) -> ('b list)
Creates a new list by applying \( f \) to each element of the input list; returns output in order.

Fold

fold f x0 lst: ('a*'b->'b)->'b->('a list)->'b
Moves across a list, applying \( f \) to each element plus an accumulator. \( f \) returns the next accumulator value, which is combined with the next element of the list

fold left vs. fold right

• Order of list elements can be significant
• Fold left moves left-to-right across the list
• Fold right moves from right-to-left

SML Implementation:

fun foldl f a [] = a
| foldl f a (x::xs) = foldl f (f(x, a)) xs

fun foldr f a [] = a
| foldr f a (x::xs) = f(x, (foldr f a xs))
Example

fun foo(l: int list) = sum(l) + mul(l) + length(l)

How can we implement this?

Example (Solved)

fun foo(l: int list) = sum(l) + mul(l) + length(l)

fun sum(lst) = foldl (fn (x,a)=>x+a) 0 lst
fun mul(lst) = foldl (fn (x,a)=>x*a) 1 lst
fun length(lst) = foldl (fn (x,a)=>1+a) 0 lst

map Implementation

fun map f [] = []
| map f (x::xs) = (f x) :: (map f xs)

• This implementation moves left-to-right across the list, mapping elements one at a time

• ... But does it need to?

Implicit Parallelism In map

• In a purely functional setting, elements of a list being computed by map cannot see the effects of the computations on other elements
• If order of application of f to elements in list is commutative, we can reorder or parallelize execution
• This is the “secret” that MapReduce exploits

References

• http://net.pku.edu.cn/~course/cs501/2008/resorce/haskell/

Outline

• Functional Programming Recap
• MapReduce Theory & Implementation
• MapReduce Algorithms
Motivation: Large Scale Data Processing
• Want to process lots of data (> 1 TB)
• Want to parallelize across hundreds/thousands of CPUs
• … Want to make this easy

MapReduce
• Automatic parallelization & distribution
• Fault-tolerant
• Provides status and monitoring tools
• Clean abstraction for programmers

Programming Model
• Borrows from functional programming
• Users implement interface of two functions:
  - map (in_key, in_value) ->
    (out_key, intermediate_value) list
  - reduce (out_key, intermediate_value list) ->
    out_value list

map
• Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).
• map() produces one or more intermediate values along with an output key from the input.

reduce
• After the map phase is over, all the intermediate values for a given output key are combined together into a list
• reduce() combines those intermediate values into one or more final values for that same output key
• (in practice, usually only one final value per key)
reduce
reduce(out_key, intermediate_value list) ->
out_value list

Parallelism
• map() functions run in parallel, creating
different intermediate values from different
input data sets
• reduce() functions also run in parallel,
each working on a different output key
• All values are processed independently
• Bottleneck: reduce phase can’t start until
map phase is completely finished.

Example: Count word occurrences
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");
reduce(String output_key, Iterator
  intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));

Example vs. Actual Source Code
• Example is written in pseudo-code
• Actual implementation is in C++, using a
MapReduce library
• Bindings for Python and Java exist via
interfaces
• True code is somewhat more involved
  (defines how the input key/values are
  divided up and accessed, etc.)

Locality
• Master program divides up tasks based on
  location of data: tries to have map() tasks
  on same machine as physical file data, or
  at least same rack
• map() task inputs are divided into 64 MB
  blocks: same size as Google File System
  chunks

Fault Tolerance
• Master detects worker failures
  – Re-executes completed & in-progress map() tasks
  – Re-executes in-progress reduce() tasks
• Master notices particular input key/values
  cause crashes in map(), and skips those
  values on re-execution.
  – Effect: Can work around bugs in third-party libraries!
Optimizations

• No reduce can start until map is complete:
  – A single slow disk controller can rate-limit the whole process
• Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish

Why is it safe to redundantly execute map tasks? Wouldn’t this mess up the total computation?

Optimizations

• “Combiner” functions can run on same machine as a mapper
• Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

Under what conditions is it sound to use a combiner?

The Example Again

```java
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");

reduce(String output_key, Iterator intermediate_values):
// output_key: a word
// output_values: a list of counts
int result = 0;
for each v in intermediate_values:
    result += ParseInt(v);
    Emit(AsString(result));
```

MapReduce Conclusions

• MapReduce has proven to be a useful abstraction
• Greatly simplifies large-scale computations at Google
• Functional programming paradigm can be applied to large-scale applications
• Fun to use: focus on problem, let library deal w/ messy details

References


Outline

• Functional Programming Recap
• MapReduce Theory & Implementation
• MapReduce Algorithms
## Algorithms for MapReduce

- Sorting
- Searching
- Indexing
- Classification
- TF-IDF
- Breadth-First Search / SSSP
- PageRank
- Clustering

## MapReduce Jobs

- Tend to be very short, code-wise
  - IdentityReducer is very common
- “Utility” jobs can be composed
- Represent a *data flow*, more so than a procedure

## Sort: Inputs

- A set of files, one value per line.
- Mapper key is file name, line number
- Mapper value is the contents of the line

## Sort Algorithm

- Takes advantage of reducer properties: (key, value) pairs are processed in order by key; reducers are themselves ordered
- Mapper: Identity function for value
  \[(k, v) \rightarrow (v, _)]
- Reducer: Identity function \((k', _) \rightarrow (k', \text{null})\)

## Sort: The Trick

- (key, value) pairs from mappers are sent to a particular reducer based on hash(key)
- Must pick the hash function for your data such that \(k_1 < k_2 \Rightarrow \text{hash}(k_1) < \text{hash}(k_2)\)

## Final Thoughts on Sort

- Used as a test of Hadoop's raw speed
- Essentially “IO drag race”
- Highlights utility of GFS
Search: Inputs

- A set of files containing lines of text
- A search pattern to find

- Mapper key is file name, line number
- Mapper value is the contents of the line
- Search pattern sent as special parameter

Search Algorithm

- Mapper:
  - Given (filename, some text) and “pattern”, if “text” matches “pattern” output (filename, _)

- Reducer:
  - Identity function

Search: An Optimization

- Once a file is found to be interesting, we only need to mark it that way once
- Use Combiner function to fold redundant (filename, _) pairs into a single one
  - Reduces network I/O

Indexing: Inputs

- A set of files containing lines of text

- Mapper key is file name, line number
- Mapper value is the contents of the line

Inverted Index Algorithm

- Mapper: For each word in (file, words), map to (word, file)

- Reducer: Identity function

Inverted Index: Data flow
### An Aside: Word Count

- Word count was described in codelab I
- Mapper for Word Count is (word, 1) for each word in input line
  - Strikingly similar to inverted index
  - Common theme: reuse/modify existing mappers

### Bayesian Classification

- Files containing classification instances are sent to mappers
- Map (filename, instance) → (instance, class)
- Identity Reducer

### Bayesian Classification

- Existing toolsets exist to perform Bayes classification on instance
  - E.g., WEKA, already in Java!
- Another example of discarding input key

### TF-IDF

- Term Frequency – Inverse Document Frequency
  - Relevant to text processing
  - Common web analysis algorithm

### The Algorithm, Formally

\[
\text{tf}_i = \frac{n_i}{\sum_k n_k}
\]
\[
\text{idf}_i = \log \frac{|D|}{|\{d : t_i \in d\}|}
\]
\[
\text{tfidf} = \text{tf} \cdot \text{idf}
\]

- \(D\): total number of documents in the corpus
- \(\{d : t_i \in d\}\): number of documents where the term \(t_i\) appears (that is \(n_i \neq 0\))

### Information We Need

- Number of times term X appears in a given document – word frequency
- Number of terms in each document – word count for a document
- Number of documents X appears in - Doc Frequency In Corpus
- Total number of documents
Job 1: Word Frequency in Doc
- Mapper
  - Input: (docname, contents)
  - Output: ((word, docname), 1)
- Reducer
  - Sums counts for word in document
  - Outputs ((word, docname), n)
- Combiner is same as Reducer

Job 2: Word Counts For Docs
- Mapper
  - Input: ((word, docname), n)
  - Output: (docname, (word, n))
- Reducer
  - Sums frequency of individual n's in same doc
  - Feeds original data through
  - Outputs ((word, docname), (n, N))

Job 3: Doc Frequency In Corpus
- Mapper
  - Input: ((word, docname), (n, N))
  - Output: (word, (docname, n, N, 1))
- Reducer
  - Sums counts for word in corpus
  - Outputs ((word, docname), (n, N, m))

Job 4: Calculate TF-IDF
- Mapper
  - Input: ((word, docname), (n, N, m))
  - Assume D is known (or, easy MR to find it)
  - Output ((word, docname), TF*IDF)
- Reducer
  - Just the identity function

Final Thoughts on TF-IDF
- Several small jobs add up to full algorithm
- Lots of code reuse possible
  - Stock classes exist for aggregation, identity
- Jobs 3 and 4 can really be done at once in same reducer, saving a write/read cycle

BFS: Motivating Concepts
- Performing computation on a graph data structure requires processing at each node
- Each node contains node-specific data as well as links (edges) to other nodes
- Computation must traverse the graph and perform the computation step
  - How do we traverse a graph in MapReduce?
  - How do we represent the graph for this?
Breadth-First Search

- Breadth-First Search is an iterated algorithm over graphs
- Frontier advances from origin by one level with each pass

Breadth-First Search & MapReduce (1/2)

- Problem:
  - This doesn’t “fit” into MapReduce
- Solution:
  - Iterated passes through MapReduce – map some nodes, result includes additional nodes which are fed into successive MapReduce passes

Breadth-First Search & MapReduce (2/2)

- Problem:
  - Sending the entire graph to a map task (or hundreds/thousands of map tasks) involves an enormous amount of memory
- Solution:
  - Carefully consider how we represent graphs

Graph Representations

- The most straightforward representation of graphs uses references from each node to its neighbors

Direct References

- Structure is inherent to object
- Iteration requires linked list “threaded through” graph
- Requires common view of shared memory (synchronization!)
- Not easily serializable

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & 1 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 0 \\
4 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Adjacency Matrices

- Another classic graph representation.
  - \( M[i][j] = '1' \) implies a link from node \( i \) to \( j \).
  - Naturally encapsulates iteration over nodes
Adjacency Matrices: Sparse Representation

- Adjacency matrix for most large graphs (e.g., the web) will be overwhelmingly full of zeros.
- Each row of the graph is absurdly long.
- Sparse matrices only include non-zero elements.

Sparse Matrix Representation

1: (3, 1), (18, 1), (200, 1)
2: (6, 1), (12, 1), (80, 1), (400, 1)
3: (1, 1), (14, 1)
...

1: 3, 18, 200
2: 6, 12, 80, 400
3: 1, 14
...

Finding the Shortest Path

- A common graph search application is finding the shortest path from a start node to one or more target nodes.
- Commonly done on a single machine with Dijkstra’s Algorithm.
- Can we use BFS to find the shortest path via MapReduce?

Finding the Shortest Path: Intuition

- We can define the solution to this problem inductively:
  - \( \text{DistanceTo}(\text{startNode}) = 0 \)
  - For all nodes \( n \) directly reachable from \( \text{startNode} \), \( \text{DistanceTo}(n) = 1 \)
  - For all nodes \( n \) reachable from some other set of nodes \( S \), \( \text{DistanceTo}(n) = 1 + \min(\text{DistanceTo}(m), m \in S) \)

From Intuition to Algorithm

- A map task receives a node \( n \) as a key, and \((D, \text{points-to})\) as its value
  - \( D \) is the distance to the node from the start
  - \( \text{points-to} \) is a list of nodes reachable from \( n \)
  - \( \forall p \in \text{points-to}, \text{emit}(p, D+1) \)
- Reduce task gathers possible distances to a given \( p \) and selects the minimum one

What This Gives Us

- This MapReduce task can advance the known frontier by one hop.
- To perform the whole BFS, a non-MapReduce component then feeds the output of this step back into the MapReduce task for another iteration.
  - Problem: Where’d the points-to list go?
  - Solution: Mapper emits \((n, \text{points-to})\) as well.
Blow-up and Termination

- This algorithm starts from one node
- Subsequent iterations include many more nodes of the graph as frontier advances
- Does this ever terminate?
  - Yes! Eventually, routes between nodes will stop being discovered and no better distances will be found. When distance is the same, we stop
  - Mapper should emit \((n, D)\) to ensure that "current distance" is carried into the reducer

Adding weights

- Weighted-edge shortest path is more useful than cost==1 approach
- Simple change: points-to list in map task includes a weight 'w' for each pointed-to node
  - emit \((p, D+w)\) instead of \((p, D+1)\) for each node p
  - Works for positive-weighted graph

Comparison to Dijkstra

- Dijkstra's algorithm is more efficient because at any step it only pursues edges from the minimum-cost path inside the frontier
- MapReduce version explores all paths in parallel; not as efficient overall, but the architecture is more scalable
- Equivalent to Dijkstra for weight=1 case

PageRank: Random Walks Over The Web

- If a user starts at a random web page and surfs by clicking links and randomly entering new URLs, what is the probability that s/he will arrive at a given page?
- The PageRank of a page captures this notion
  - More "popular" or "worthwhile" pages get a higher rank

PageRank: Visually

PageRank: Formula

Given page A, and pages T1 through Tn linking to A, PageRank is defined as:

\[
PR(A) = (1-d) + d \left( \frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_n)}{C(T_n)} \right)
\]

\(C(P)\) is the cardinality (out-degree) of page P
\(d\) is the damping ("random URL") factor
PageRank: Intuition

- Calculation is iterative: PR_{i+1} is based on PR_i
- Each page distributes its PR_i to all pages it links to. Linkees add up their awarded rank fragments to find their PR_{i+1}
- d is a tunable parameter (usually = 0.85) encapsulating the “random jump factor”

\[ PR(A) = (1-d) + d \left( \frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_n)}{C(T_n)} \right) \]

PageRank: First Implementation

- Create two tables ‘current’ and ‘next’ holding the PageRank for each page. Seed ‘current’ with initial PR values
- Iterate over all pages in the graph, distributing PR from ‘current’ into ‘next’ of linkees
- current := next; next := fresh_table();
- Go back to iteration step or end if converged

Distribution of the Algorithm

- Key insights allowing parallelization:
  - The ‘next’ table depends on ‘current’, but not on any other rows of ‘next’
  - Individual rows of the adjacency matrix can be processed in parallel
  - Sparse matrix rows are relatively small

Distribution of the Algorithm

- Consequences of insights:
  - We can map each row of ‘current’ to a list of PageRank “fragments” to assign to linkees
  - These fragments can be reduced into a single PageRank value for a page by summing
  - Graph representation can be even more compact; since each element is simply 0 or 1, only transmit column numbers where it’s 1

Phase 1: Parse HTML

- Map task takes (URL, page content) pairs and maps them to (URL, (PR_{init}, list-of-urls))
  - PR_{init} is the “seed” PageRank for URL
  - list-of-urls contains all pages pointed to by URL
- Reduce task is just the identity function
Phase 2: PageRank Distribution

- Map task takes (URL, (cur_rank, url_list))
  - For each u in url_list, emit (u, cur_rank/url_list)
  - Emit (URL, url_list) to carry the points-to list along through iterations

\[ PR(A) = (1-d) + d \left( \frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_n)}{C(T_n)} \right) \]

Phase 2: PageRank Distribution

- Reduce task gets (URL, url_list) and many (URL, val) values
  - Sum vals and fix up with d
  - Emit (URL, (new_rank, url_list))

\[ PR(A) = (1-d) + d \left( \frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_n)}{C(T_n)} \right) \]

Finishing up...

- A non-parallelizable component determines whether convergence has been achieved (Fixed number of iterations? Comparison of key values?)
- If so, write out the PageRank lists - done!
- Otherwise, feed output of Phase 2 into another iteration

PageRank Conclusions

- MapReduce isn't the greatest at iterated computation, but still helps run the "heavy lifting"
- Key element in parallelization is independent PageRank computations in a given step
- Parallelization requires thinking about minimum data partitions to transmit (e.g., compact representations of graph rows)
  - Even the implementation shown today doesn't actually scale to the whole Internet; but it works for intermediate-sized graphs

Clustering

- What is clustering?

Google News

- They didn’t pick all 3,400,217 related articles by hand…
- Or Amazon.com
- Or Netflix…
Other less glamorous things...

- Hospital Records
- Scientific Imaging
  - Related genes, related stars, related sequences
- Market Research
  - Segmenting markets, product positioning
- Social Network Analysis
- Data mining
- Image segmentation...

The Distance Measure

- How the similarity of two elements in a set is determined, e.g.
  - Euclidean Distance
  - Manhattan Distance
  - Inner Product Space
  - Maximum Norm
  - Or any metric you define over the space...

Types of Algorithms

- Hierarchical Clustering vs.
- Partitional Clustering

Hierarchical Clustering

- Builds or breaks up a hierarchy of clusters.

Partitional Clustering

- Partitions set into all clusters simultaneously.
K-Means Clustering

• Simple Partitional Clustering
• Choose the number of clusters, k
• Choose k points to be cluster centers
• Then…

\[
\text{iterate} \{
\begin{align*}
\text{Compute distance from all points to all k-centers} \\
\text{Assign each point to the nearest k-center} \\
\text{Compute the average of all points assigned to all specific k-centers} \\
\text{Replace the k-centers with the new averages}
\end{align*}
\}
\]

But!

• The complexity is pretty high: \( k \times n \times O(\text{distance metric}) \times \text{num (iterations)} \)
• Moreover, it can be necessary to send tons of data to each Mapper Node. Depending on your bandwidth and memory available, this could be impossible.

Furthermore

• There are three big ways a data set can be large:
  - There are a large number of elements in the set.
  - Each element can have many features.
  - There can be many clusters to discover
• Conclusion – Clustering can be huge, even when you distribute it.

Canopy Clustering

• Preliminary step to help parallelize computation.
• Clusters data into overlapping Canopies using super cheap distance metric.
• Efficient
• Accurate

\[
\text{Canopy Clustering} \begin{align*}
\text{While there are unmarked points} \{ \\
\text{pick a point which is not strongly marked} \text{ call it a canopy center} \\
\text{mark all points within some threshold of it as in it’s canopy} \\
\text{strongly mark all points within some stronger threshold}
\end{align*}
\]
After the canopy clustering...

- Resume hierarchical or partitional clustering as usual.
- Treat objects in separate clusters as being at infinite distances.

MapReduce Implementation:

- Problem – Efficiently partition a large data set (say... movies with user ratings!) into a fixed number of clusters using Canopy Clustering, K-Means Clustering, and a Euclidean distance measure.

The Distance Metric

- The Canopy Metric ($$
- The K-Means Metric ($$$)

Steps!

- Get Data into a form you can use (MR)
- Picking Canopy Centers (MR)
- Assign Data Points to Canopies (MR)
- Pick K-Means Cluster Centers
- K-Means algorithm (MR)
  – Iterate!

Data Massage

- This isn’t interesting, but it has to be done.

Selecting Canopy Centers
Elbow Criterion

- Choose a number of clusters s.t. adding a cluster doesn’t add interesting information.
- Rule of thumb to determine what number of Clusters should be chosen.
- Initial assignment of cluster seeds has bearing on final model performance.
- Often required to run clustering several times to get maximal performance.
Clustering Conclusions

• Clustering is slick
• And it can be done super efficiently
• And in lots of different ways

Overall Conclusions

• Lots of high level algorithms
• Lots of deep connections to low-level systems
• Clean abstraction layer for programmers between the two