Lecture 24
MapReduce
This Class So Far

- *Concurrent* programming in Java
- Exploiting *parallelism* to improve performance
- *Distributed* programming in Java using RMI
- Next topic: *MapReduce*
  - A “programming model” for processing large data sets in parallel on a cluster
  - Developed by Google researchers in early 2000s
  - Key features
    - Conceptual simplicity
    - Scalability and fault-tolerance of operations
MapReduce, Conceptually

- Input data consists of key/value pairs
  - E.g. key could be student name: “Testudo Terrapin”
  - Value could be course being taken, e.g. “CMSC 433”
- MapReduce developer specifies
  - “map” function to produce intermediate set of (possibly) new-key, new-value pairs
  - “reduce” function to convert intermediate data into final result
MapReduce is based on concepts from *function programming*

- “map” in functional languages (e.g. OCaml!) converts a function over values to a function mapping lists to lists
  - Given list, (map f) applies f to each element in the list
  - The list of results is then returned
- “fold” takes a seed / function value as input, returns a function mapping lists to single values as output
  - Actually, two versions: “left” and “right”
  - Point of both is to convert list to single value

So?

- Functional languages do not modify variables
- Mapping can be computed in parallel!
- MapReduce uses a variant of “fold”; details later
Functional Map

• Suppose f is a function
• Then (map f) is a new function on lists:

\[ [x_1; x_2; \ldots ; x_n] \]

\[ \downarrow f \quad \downarrow f \quad \downarrow f \]

\[ [f(x_1); f(x_2); \ldots ; f(x_n)] \]
Map Examples in OCaml

```ocaml
# let add1 x = x+1;;
val add1 : int -> int = <fun>
# let g = List.map add1;;
val g : int list -> int list = <fun>
# g [1;2;3];;
- : int list = [2; 3; 4]
# let double x = [x;x];;
val double : 'a -> 'a list = <fun>
# let h = List.map double;;
val h : '_a list -> '_a list list = <fun>
# h [1;2;3];;
- : int list list = [[1; 1]; [2; 2]; [3; 3]]
```
Functional Fold (Left)

• Suppose $f$ is a *binary* function, $s$ is a value.
• Then $(\text{fold\_left } f \ s)$ is a function that collapses lists to a single value.

$$(\text{fold\_left } f \ s) \ [x_1; x_2; \ldots \ x_n] =$$

$$f ( \ldots f ( f(s, x_1), x_2 ) \ldots, x_n)$$

• E.g. if $f(x,y) = x+y$, $s = 0$, then

$$(\text{fold\_left } f \ 0) \ [1;2;3] = ((0+1) + 2) + 3 = 6$$
Fold (left) Examples in OCaml

```ocaml
# let sum x y = x+y;;
val sum : int -> int -> int = <fun>
# let h = List.fold_left sum 0;;
val h : int list -> int = <fun>
# h [1;2;3];;
- : int = 6
# let prefix tl hd = hd::tl;;
val prefix : 'a list -> 'a -> 'a list = <fun>
# let k = List.fold_left prefix [];;
val k : '_a list -> '_a list = <fun>
# k [1;2;3];;
- : int list = [3; 2; 1]
```
MapReduce, Logically

• Assumption: input data for MapReduce applications consists of collections of (key, value) pairs

• A MapReduce application contains:
  – A “mapper function” converting single (key, value) pairs to lists of (key2, value2) pairs
  – A “reducer function” converting pairs of form (key2, value2 list) to a list of values

• The MapReduce framework does the following
  – Apply “mapper” to the input data
  – Glue together the resulting lists into a single list of (key2, value2) pairs
  – Rearrange this list into a list (key2, value2 list) pairs, where each distinct key2 appears once
  – Applying “reducer” to each element in the new list
  – Return the aggregate results
MapReduce in OCaml

- MapReduce can be implemented in OCAML
  - Auxiliary functions
    - flatten: convert list of lists into a single list by gluing them together
    - collect: convert list of (key2, value2) lists into list of (key2, value2 list) list
  - The “mapReduce” function takes a mapper, reducer, produces a new “end-to-end” function
  - See code in mapReduce.ml

- This implementation has no concurrency!
  - It only demonstrates functionality of MapReduce
  - Distributed implementations have to provide the same functionality!
MapReduce: flatten

- Recall: in OCAML “List.append” (also written as “@”) glues two lists together in order
  \( \text{List.append } ([0;1], [2;3]) = [0;1] @ [2;3] = [0;1;2;3] \)
- flatten generalizes this to “lists of lists”: 
  \( \text{flatten } [ [0;1]; [2;3] ] = [0;1;2;3] \)
- flatten \([ l_1; l_2; \ldots l_n ]\) can be thought of as:
  \( ( \ldots ((([] @ l_1) @ l_2) @ \ldots) @ l_n \)
- So, \( \text{flatten} = \text{fold_left} \text{ List.append } [] \)!
MapReduce: collect

• What does collect do?
  – Reads output of “mapper”, i.e. list of lists of (key2, value2) pairs
  – Produces lists of (key2, value2 list) pairs

• How to do this?
  – Flatten output of mapper to obtain list of (key2, value2) pairs
  – For each pair in this list, insert into “structure under construction”, which is initially []
  – Another application of List.fold_left!

```ocaml
# let collect mapOut = List.fold_left insert [] (flatten mapOut);;
val collect : ('_a * '_b) list list -> ('_a * '_b list) list = <fun>
```
MapReduce: insert

• Job of insert
  – Given list of (key2, value2 list), (key2,value2) pair
  – Return new (key2, value2 list) with value inserted appropriately

• Code

```ml
let rec insert l (k, v) =
  match l with
  | [] -> [(k, [v])]
  | (k',l')::tl ->
    if (k' = k)
    then (k', v::l')::tl
    else (k', l')::(insert tl (k,v))
```
MapReduce in OCaml

• Code

```ocaml
let mapReduce mapFun reduceFun data =
  let mapResult = List.map mapFun data in
  let collectResult = collect mapResult in
  let reduceResult = List.map reduceFun collectResult in
  flatten reduceResult
```

• What does a developer provide?
  – mapFun
  – reduceFun

• Implementation of mapReduce takes care of everything else!
To Install OCaml

• http://caml.inria.fr/ocaml/release.en.html