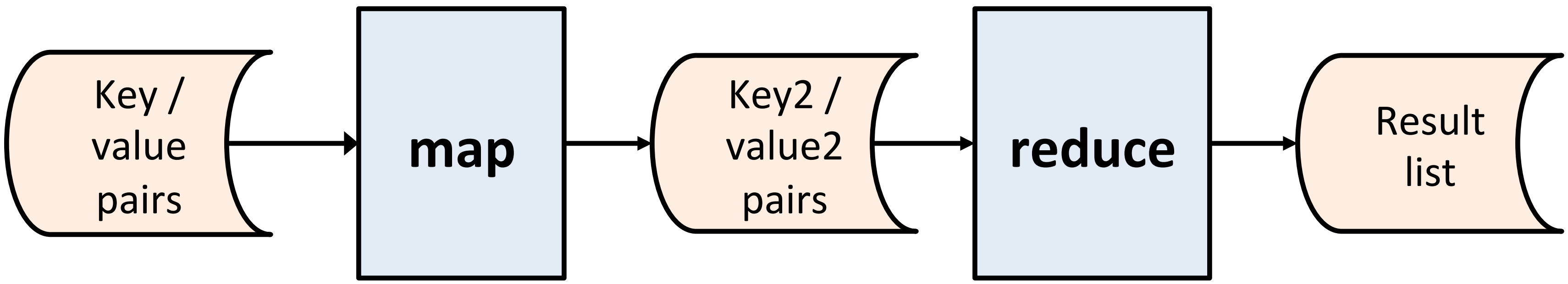
CMSC 433 Fall 2014
Section 0101
Michael Hicks
(most slides due to Rance Cleaveland)

Lecture 21
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 a URL: “www.cs.umd.edu”
  - Value could be the .html code in the file associated with the URL
- 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

What?

- Think of data processed by MapReduce as “tables”
  - The table has two columns: one for keys, the other for values
  - Each key/value pair in the data set corresponds to a row in the table
- So:
  - “map” converts input table into a new, intermediate table
  - “reduce” constructs a new table that aggregates data in the intermediate table

Key	Value
key_1	$value_1$
key_2	$value_2$
…	
key_n	$value_n$

Example: Word Counting

- Suppose we want to give a MapReduce application giving an occurrence count for each word in a list of files
  - Input table: file name / file contents pairs (both strings, the second being much longer!)
  - Final table produced by reduce: word / int pairs, where each int is the # of occurrences of the word in the documents
- How do we do this using MapReduce?

Word Counting: map

map converts individual row (file name, file contents) into collection of (word, "1") rows

File name (key)	Contents (value)
constitution.txt	"We the people …"
decl_ind.txt	"When, in the course…"
…	

map

Word (key2)	Count (value2)
"When"	1
"in"	1
"the"	1
"course"	1
…	

Word Counting: reduce

reduce takes all rows with a given word (key2) and sums the counts (value2), yielding (at most!) one row in output table

Word (key2)	Count (value2)
…	
"the"	1
"the"	1
…	
"the"	1
…	

reduce

Word (key2)	Count (value2)
…	
"the"	15
…	

Other Applications of MapReduce

- grep (search)
- Count of URL reference frequency ("pagerank")
- "Web-link graph reversal": compute all URLs with a link to each of a given list of URLs
- sort
- "Inverted index": given list of documents, produce output giving, for each word, the documents it appears in
- Used by 1000s of organizations around the world, including Amazon, Google, Yahoo, …

Foundations of MapReduce

- MapReduce concepts from *functional 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
    - In Lisp, "fold" referred to as "reduce"
- So?
  - Functional languages do not (normally) 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:

$[x_1;$ $\quad x_2; \dots ;$ $\quad x_n]$

$\downarrow f \qquad \downarrow f \qquad \downarrow f \qquad$ (map f)

$[f(x_1);$ $\quad f(x_2); \dots ;$ $\quad f(x_n)]$

- The $f(x_i)$ can be computed in parallel!
  - The x_i do not share state
  - f cannot modify its arguments

Map Examples in 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 (fold_left f s) is a function that "iteratively applies" f over lists to produce a single value

  (fold_left f s) [x_1; x_2; … x_n] =

  f (… f (f(s, x_1), x_2) …, x_n)
- E.g. if f(x,y) = x+y, s = 0, then

  (fold_left f 0) [1;2;3] = ((0+1) + 2) + 3 = 6

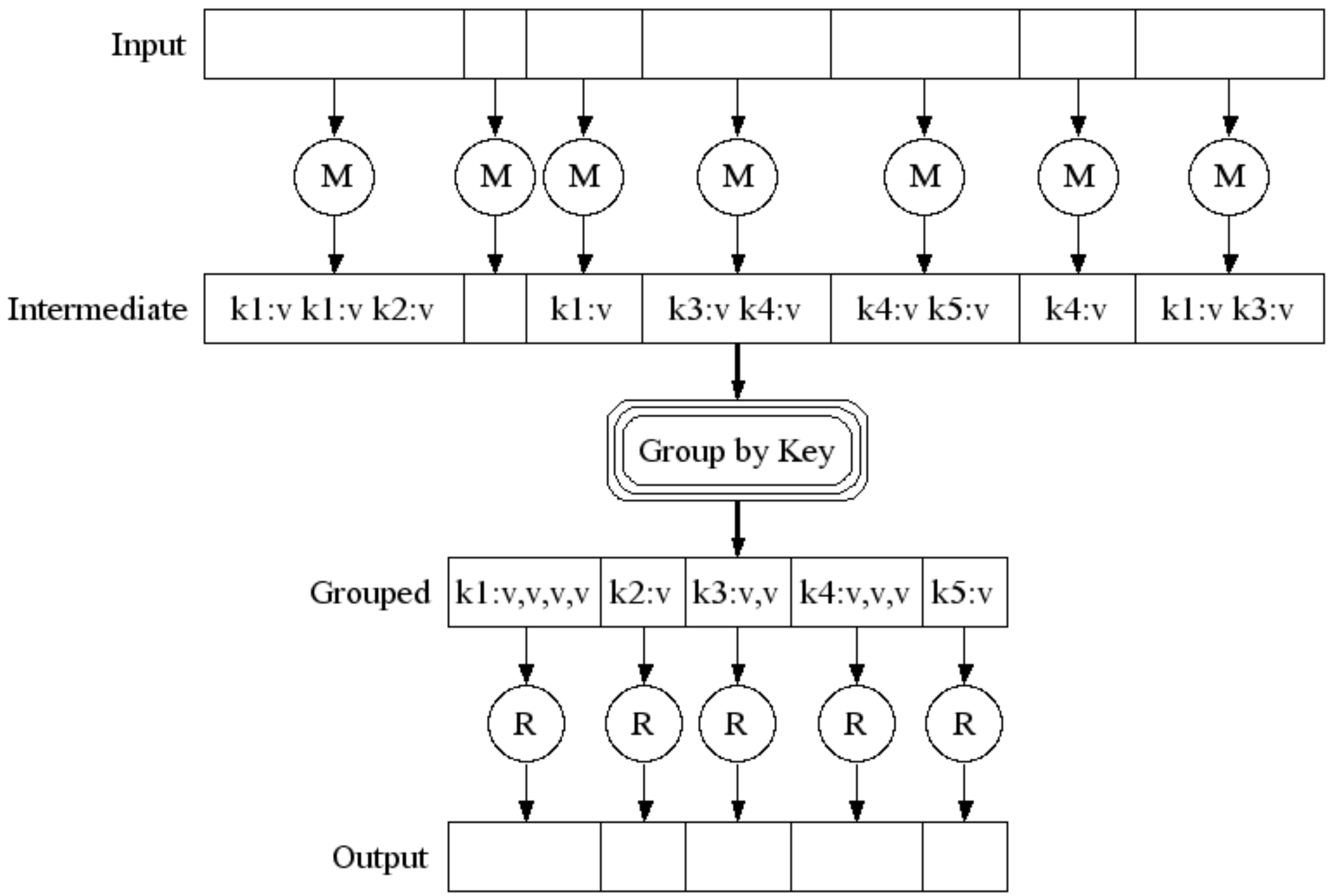
Fold (left) Examples in OCaml

```
# 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]
```

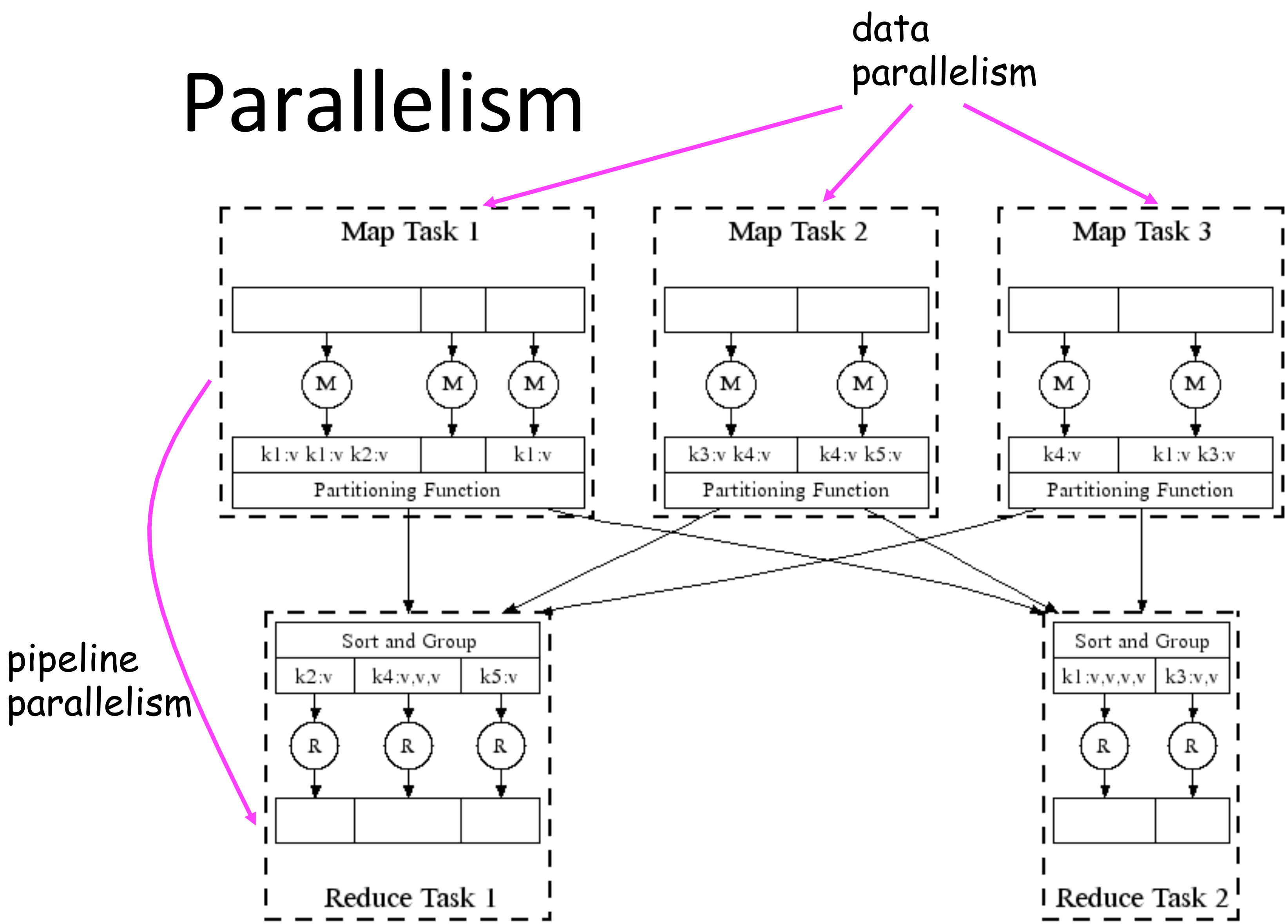
MapReduce, Logically

- Assumption: input data for MapReduce applications consists of lists of (key, value) pairs (i.e. tables)
- A MapReduce application contains:
  - A "mapper function" converting single (key, value) pairs (i.e. single rows in the old table) to lists of (key2, value2) pairs (i.e. multiple rows in the new table)
  - A "reducer function" converting pairs of form (key2, value2 list) to a list of values (i.e. reducer aggregates data associated to key2 in the intermediate table)
- 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
  - Group elements of this list into (key2, value2 list) pairs, where each distinct key2 appears once
  - Applying "reducer" to each (key2, value2 list) element of the new list
  - Return the list of results

MapReduce, Visually



Parallelism



Key: no implicit dependencies between map or reduce tasks

Fault Tolerance

- Handle worker failures via re-execution
  - Detect failure via periodic heartbeats
- Re-execute in-progress **reduce** tasks and in-progress and completed **map** tasks
  - Can do this easily since inputs are stored on the file system
- Key: Map and Reduce are functional
  - So they can always be reexecuted to produce the same answer

Optimizations

- Perform redundant computations on idle machines
  - Whichever one finishes "wins"
- Exploit locality: send tasks to the data, not the other way around
- Use *combiners* to reduce data transfer
  - Called at mapper before sending results to reducer
  - Implements the reducer interface
    - but input/output key and values match mapper
  - Function must be commutative and associative (why?)

Programming Model is the Key

- Simple control makes dependencies evident
  - Can automate scheduling of tasks and optimization
    - Map, reduce for different keys, embarassingly parallel
    - Pipeline between mappers, reducers evident
- **map** and **reduce** are pure functions
  - Can rerun them to get the same answer
    - in the case of failure, or
    - to use idle resources toward faster completion
  - No worry about data races, deadlocks, etc. since there is no shared state

MapReduce in OCaml

- MapReduce can be implemented in OCAML
  - Auxiliary functions
    - *flatten*: convert list of lists into a single list by gluing them together
    - *groupby*: 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

  List.append ([0;1], [2;3]) = [0;1] @ [2;3] = [0;1;2;3]

- flatten generalizes this to "lists of lists":

  flatten [[0;1]; [2;3]] = [0;1;2;3]

- flatten [l_1; l_2; … l_n] can be thought of as:

  (… (([] @ l_1) @ l_2) @ …) @ l_n

- So, flatten = fold_left List.append [] !

MapReduce: groupby

- What does groupby 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!

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

MapReduce: insert

- Job of insert
  - Given list of (key2, value2 list) pairs, (key2,value2) pair
  - Return new (key2, value2 list) list with value inserted appropriately
- Code

```
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

```
let mapReduce mapFun reduceFun data =
  let mapResult = List.map mapFun data in
  let groupedResult = groupby mapResult in
  let reduceResult = List.map reduceFun groupedResult in
    flatten reduceResult
```

- What does a developer provide?
  – mapFun : ‘k * ‘v -> ‘k2 * ‘v2
  – reduceFun: ‘k2 * ‘v2 list -> ‘v3 list
- Implementation of mapReduce takes care of everything else!

Word Counting in OCaml MapReduce

- Input data has type (string * string) list, where first string (key) is file name, second (value) is file contents
- Result has type (string * int) list, where string (key2) is word, int (value) is # of occurrences of word in files
- Mapper

```
let wcMapFun (fname, fcontents) =
  List.map (fun str -> (str, 1)) (stringToWordList
fcontents)
```

- Reducer

```
let wcReduceFun (word, numList) = [(word, sumList
numList)]
```

- Application

```
let wordCounter = mapReduce wcMapFun wcReduceFun
```

- Details in mapReduceWordCount.ml

MapReduce in Erlang

- File phofs.erl
  - Parallel mappers, but only single reducer
- `mapreduce(MapFun, ReduceFun, L)`
  - Spawns process to call MapFun on each element of L; this function will send (k,v) pairs back
  - Calls group_by to gather responses into a dictionary mapping each key k to a list of values vs
  - Calls ReduceFun on each element of dictionary to produce final list of results

Word Counting in Erlang

```
wc_dir(Dir) ->
    F1 = fun generate_words/2,
    F2 = fun count_words/3,
    Files = … text file names …,
    L1 = phofs:mapreduce(F1, F2, Files),
    reverse(sort(L1)).
```