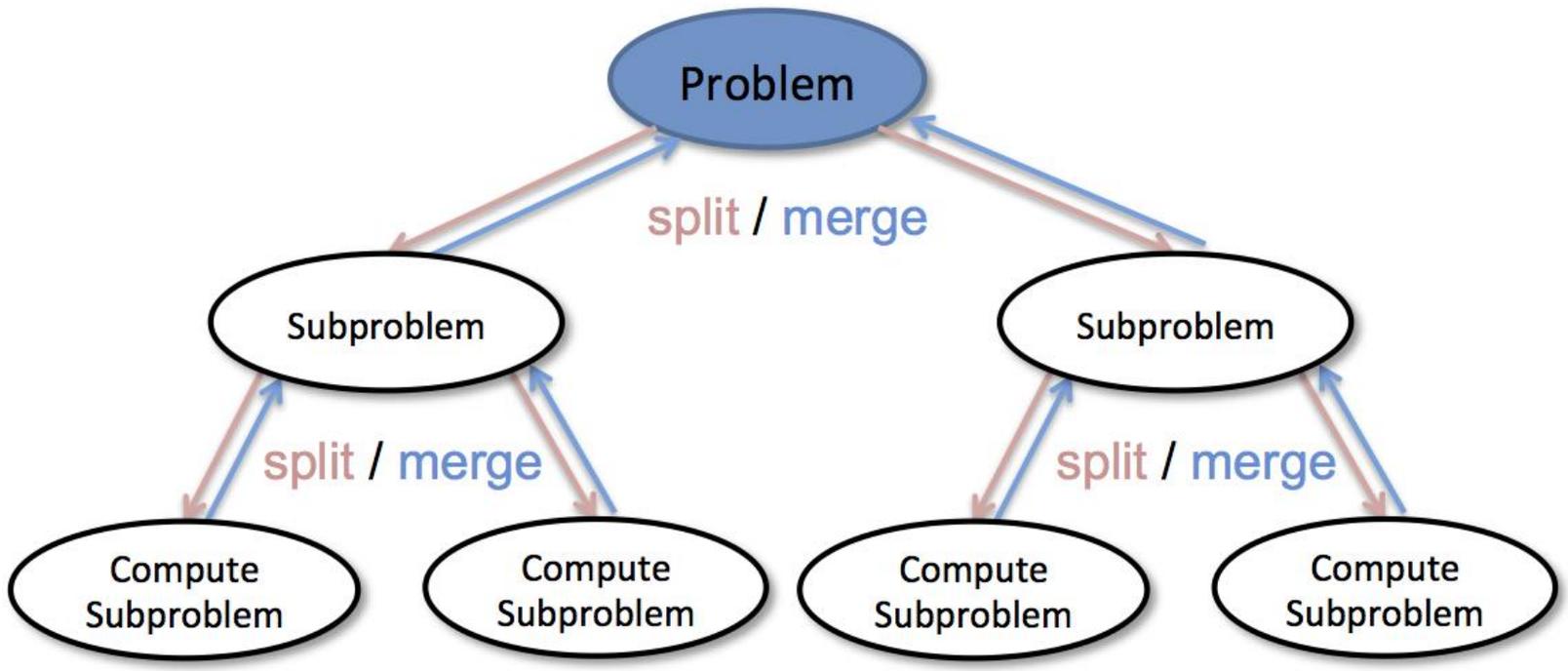
DTS205TC High Performance Computing

Lecture 10 MapReduce

Di Zhang, Spring 2024

Divide-and-conquer



- A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.
- *Example: binary search, merge sort

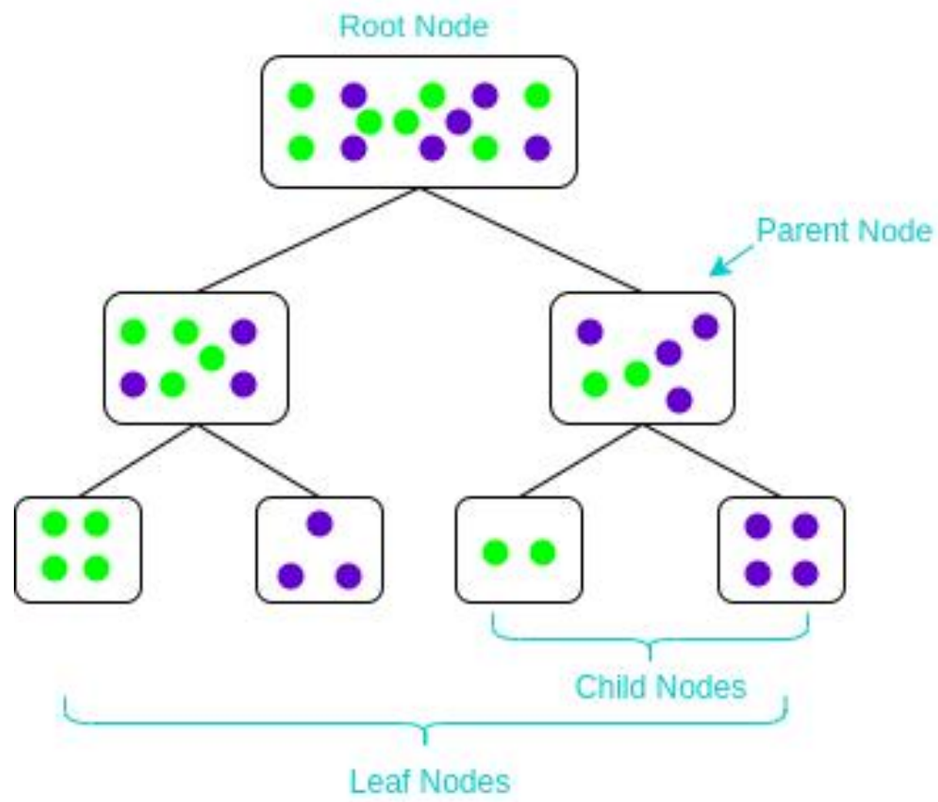
A simplest example


```python
# print string char by char

def divide_conquer(s):

    if len(s) == 1:
        print(s)
    else:
        divide_conquer(s[:len(s)//2]) # print first half
        divide_conquer(s[len(s)//2:]) # print second half

divide_conquer('abcdefg')
```

```
a
b
c
d
e
f
g

Process finished with exit code 0
```

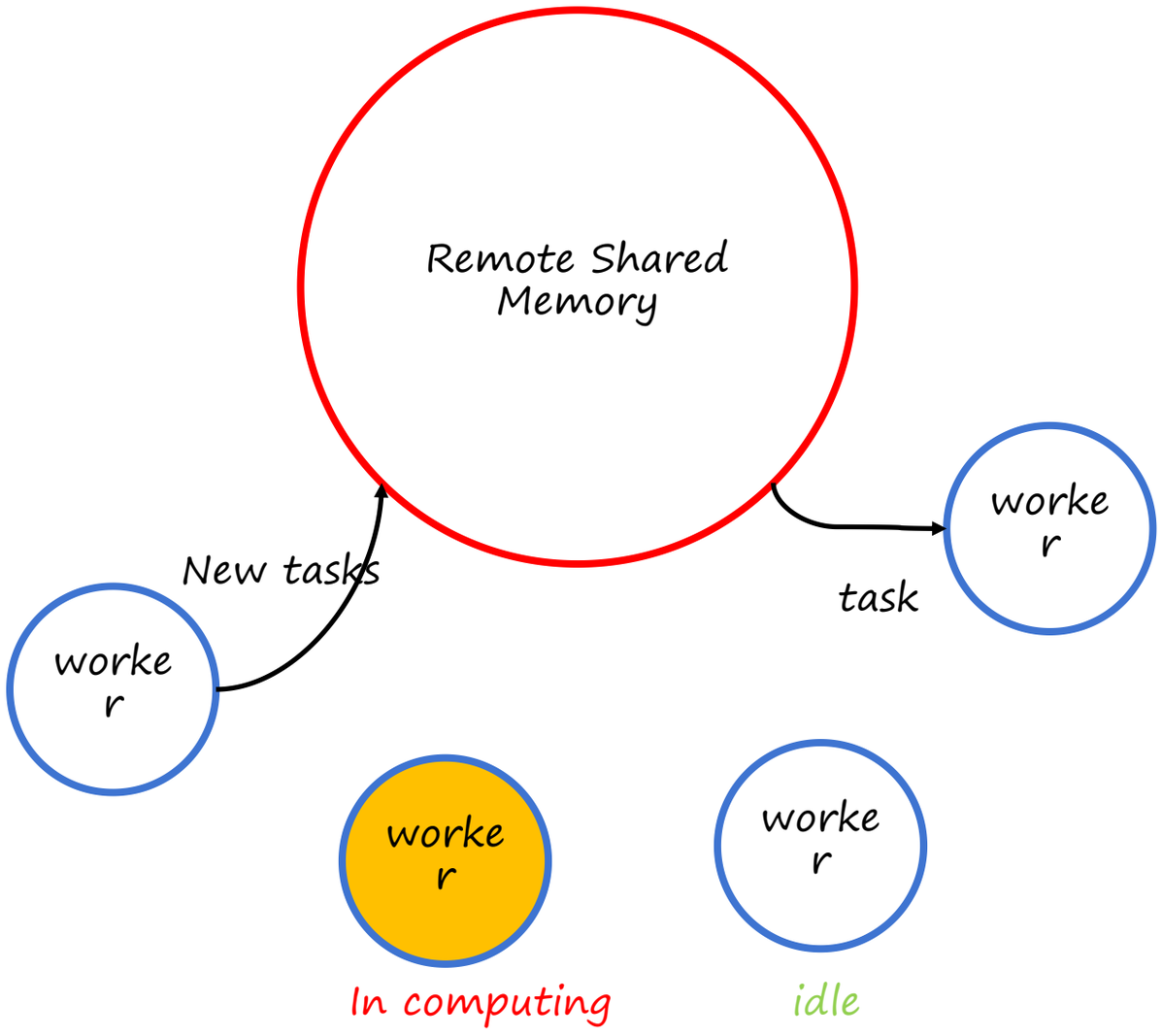
- Key elements of Divide-and-conquer:
  - Divisible problem:
    - Print the string
  - Split：
    - Divide the string in half
  - Conditions for Termination:
    - Whether the length is 1
  - Termination action:
    - print

Example: Decision Tree



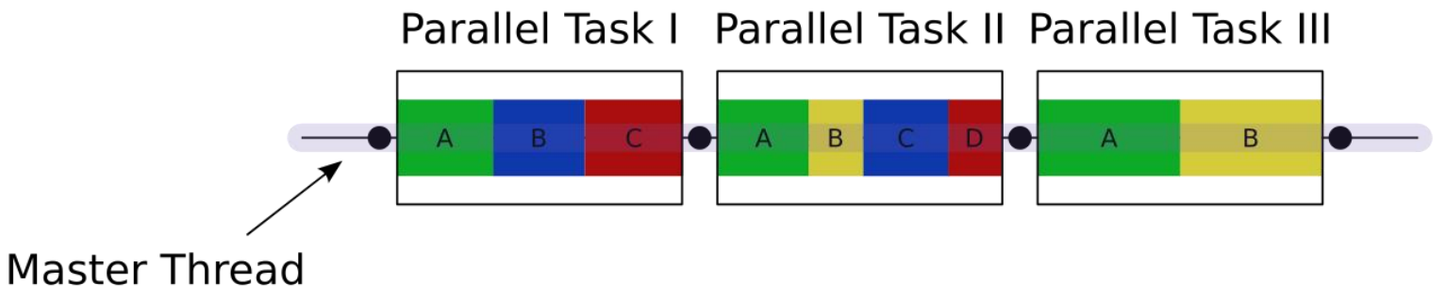
- Key elements:
  - Divisible problem:
    - Build decision boundaries to 'purify' data
  - Split：
    - By information gain, the sample is divided into two piles
  - Conditions for Termination:
    - Whether it contains only one class

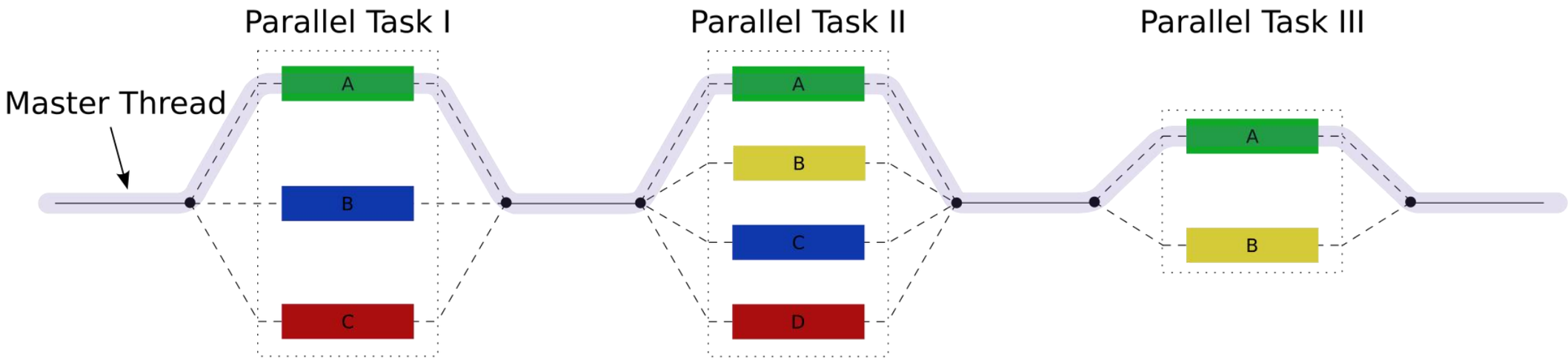
Parallelization



- Use workpool to implement dynamic assignment of tasks in Divide-and-conquer
  Unlike before, this is that each work is not only a consumer of the task, but also a producer; i.e., the number of tasks can be increased and is not a fixed value (in Lec. 6)

Fork-Join

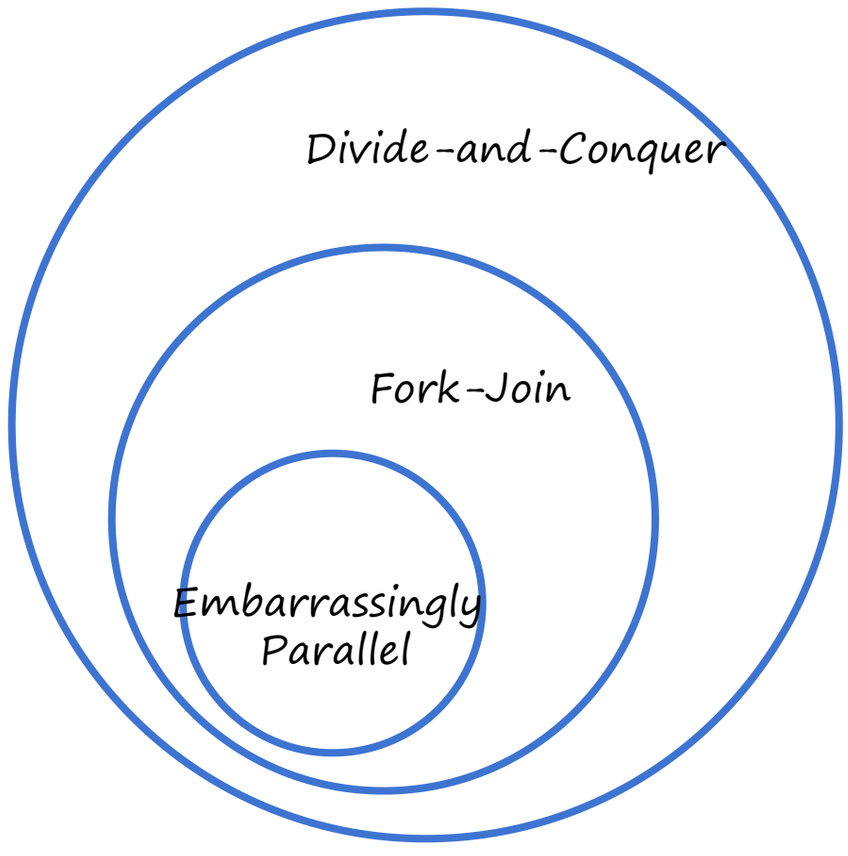




- Fork-Join executes branches in parallel at Fork points in the program, to "join" (merge) at a subsequent point and resume sequential execution
- It is a simplified and iterative version of Divide-and-conquer
- Fork-Join is the main model of parallel execution in the OpenMP

Compared to Embarrassingly Parallel

- Embarrassingly Parallel
  - Only with Fork, no Join!
- The expressive capabilities of the three models are compared as below. They are each other's subsets and supersets.



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
- Lists are primitive data types

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)
```

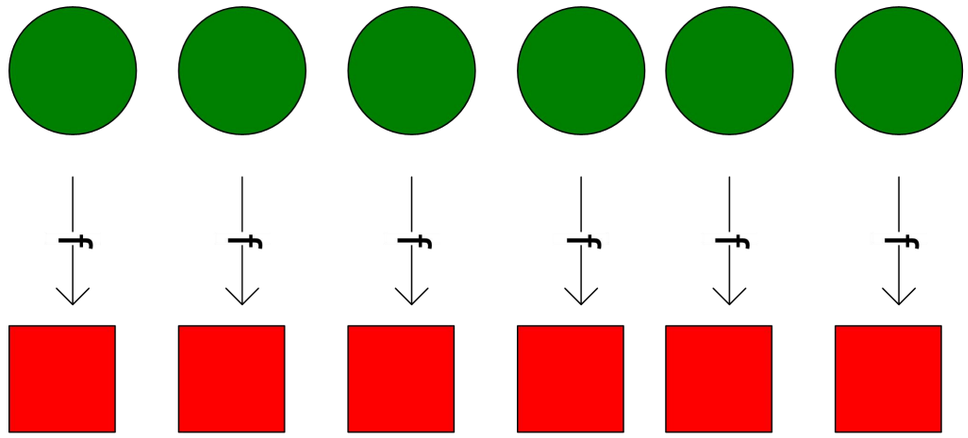
It does not matter what f does to its argument; DoDouble() will do it twice.

What is the type of this function?

Map

map f lst: ('a->'b) -> ('a list) -> ('b list)

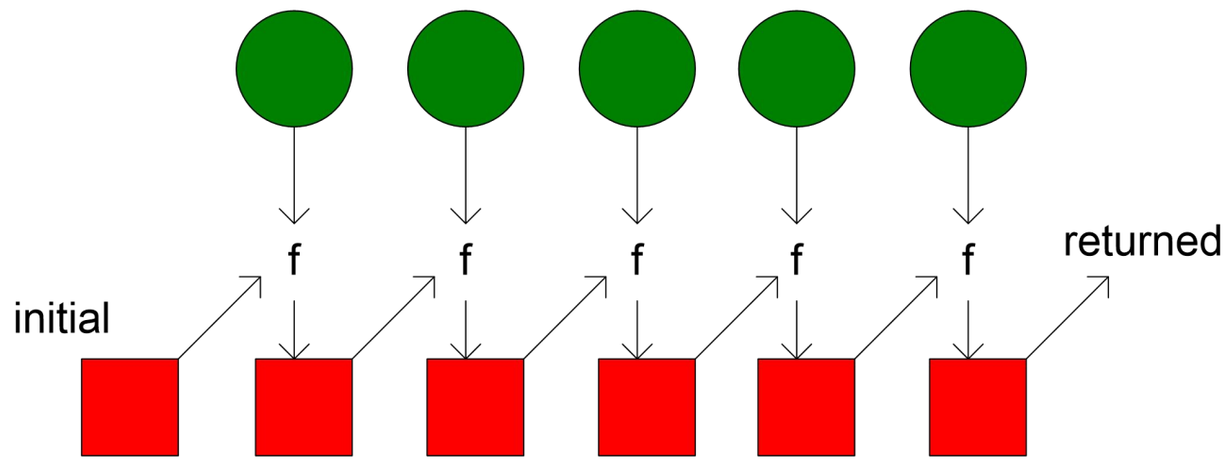
Creates a new list by applying f to each element of the input list; returns output in order.



Fold

fold f x_0 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
```

A More Complicated Fold Problem

- Given a list of numbers, how can we generate a list of partial sums?

e.g.: [1, 4, 8, 3, 7, 9]

[0, 1, 5, 13, 16, 23, 32]

A More Complicated Map Problem

- Given a list of words, can we: reverse the letters in each word, and reverse the whole list, so it all comes out backwards?

["my", "happy", "cat"] -> ["tac", "yppah", "ym"]

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

Why Look at Functional Programming?

Xi'an Jiaotong-Liverpool University
西交利物浦大學

- Let's assume a long list of records: imagine if...
  - We can parallelize map operations
  - We have a mechanism for bringing map results back together in the fold operation
- That's MapReduce!
- Observations:
  - No limit to map parallelization since maps are independent
  - We can reorder folding if the fold function is commutative and associative

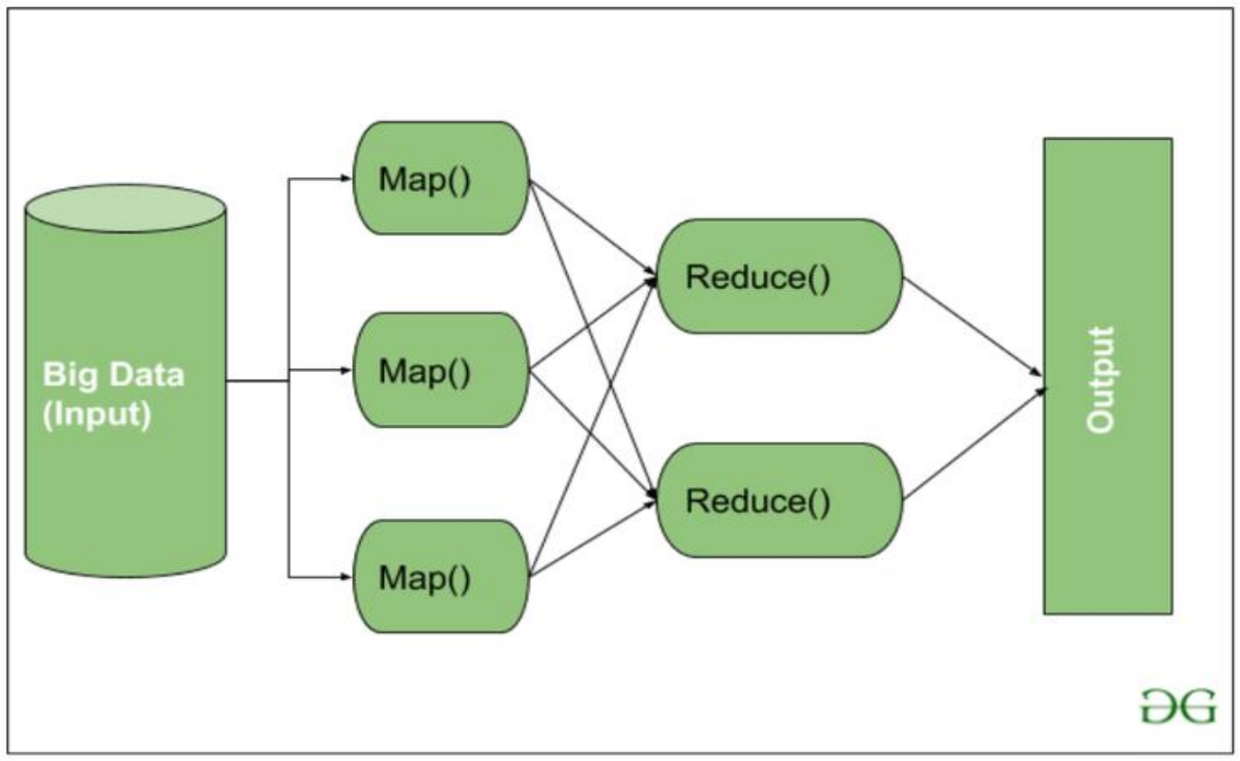
MapReduce

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

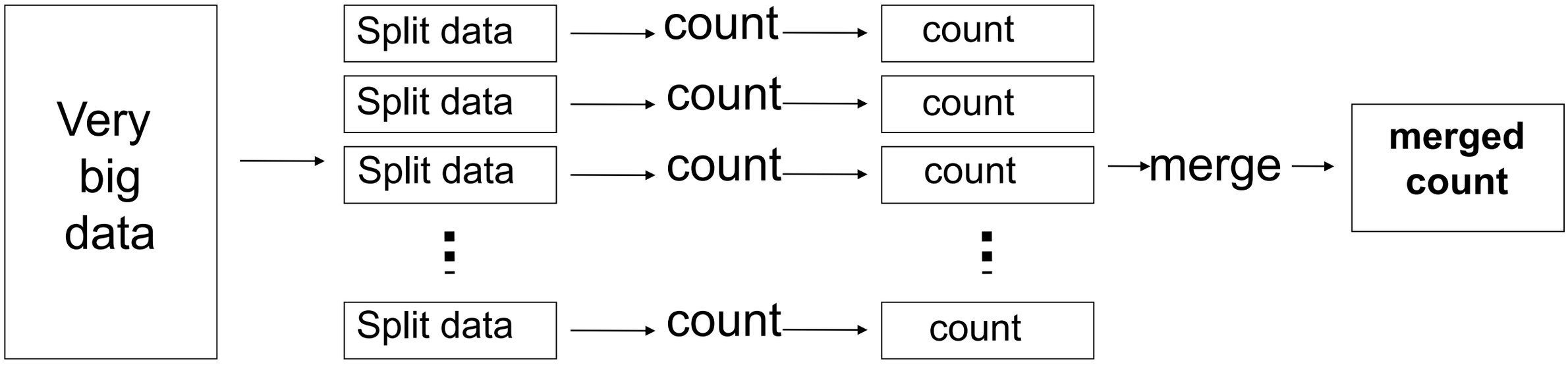
- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as (key,value) pairs: e.g., (filename, offset).
- map() produces one or more *intermediate* values along with an output key from the input.

Reduce

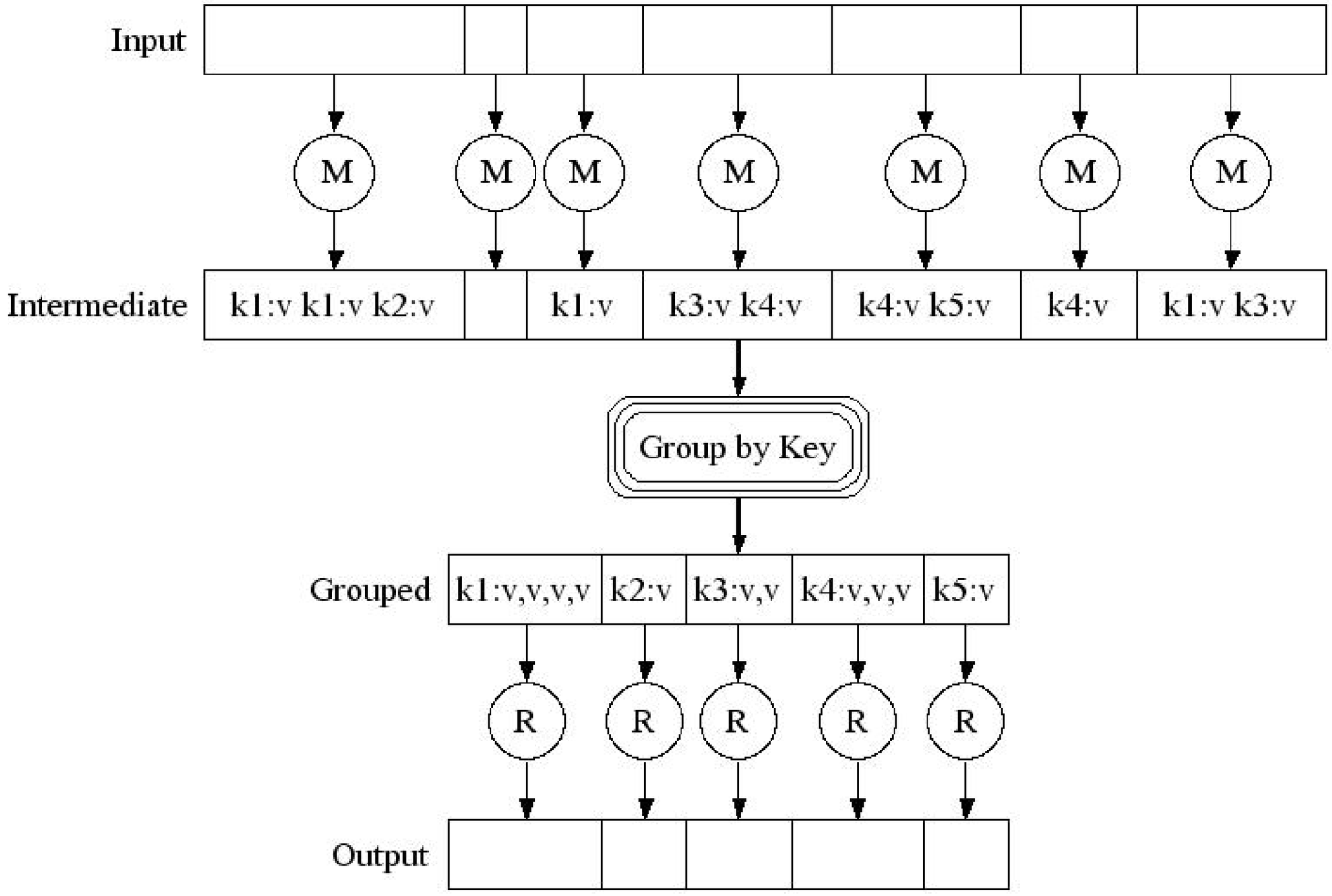
- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)

Distributed Word Count

Xi'an Jiaotong-Liverpool University 西交利物浦大学



Partitioning Function



Example Word Count

Xi'an Jiaotong-Liverpool University 西交利物浦大學

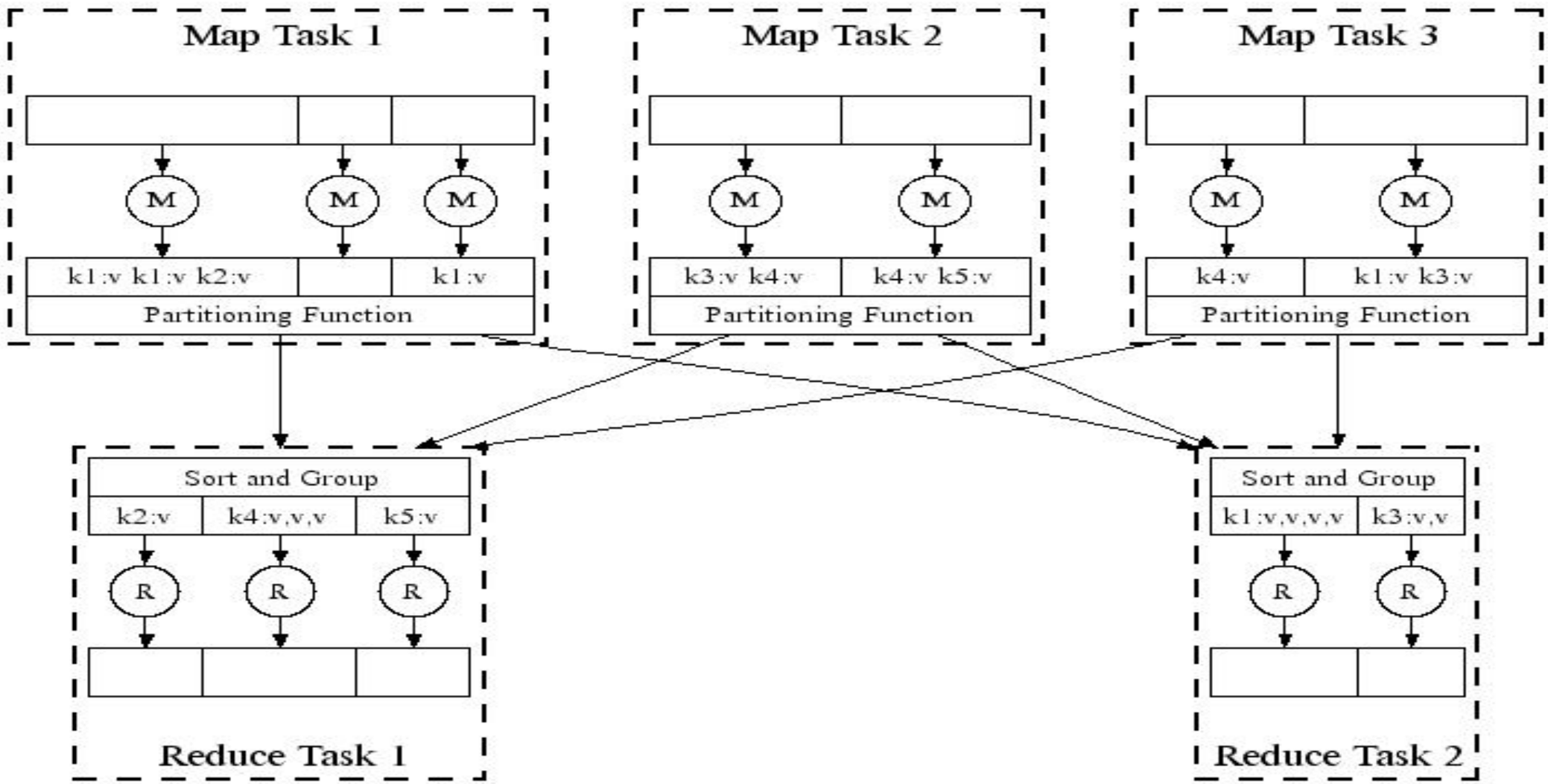
- Map

```
public static class MapClass extends MapReduceBase
implements Mapper {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(WritableComparable key, Writable value,
  OutputCollector output, Reporter reporter)
  throws IOException {
    String line = ((Text)value).toString();
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      output.collect(word, one);
    }
  }
}
```

Partitioning Function (2)

- Default : `hash(key) mod R`
- Guarantee:
  - Relatively well-balanced partitions
  - Ordering guarantee within partition

MapReduce in Parallel: Example



MapReduce

- Distributed Grep
  - Map:

    ```
    if match(value,pattern) emit(value,1)
    ```

  - Reduce:

    ```
    emit(key,sum(value*))
    ```

- Distributed Word Count
  - Map:

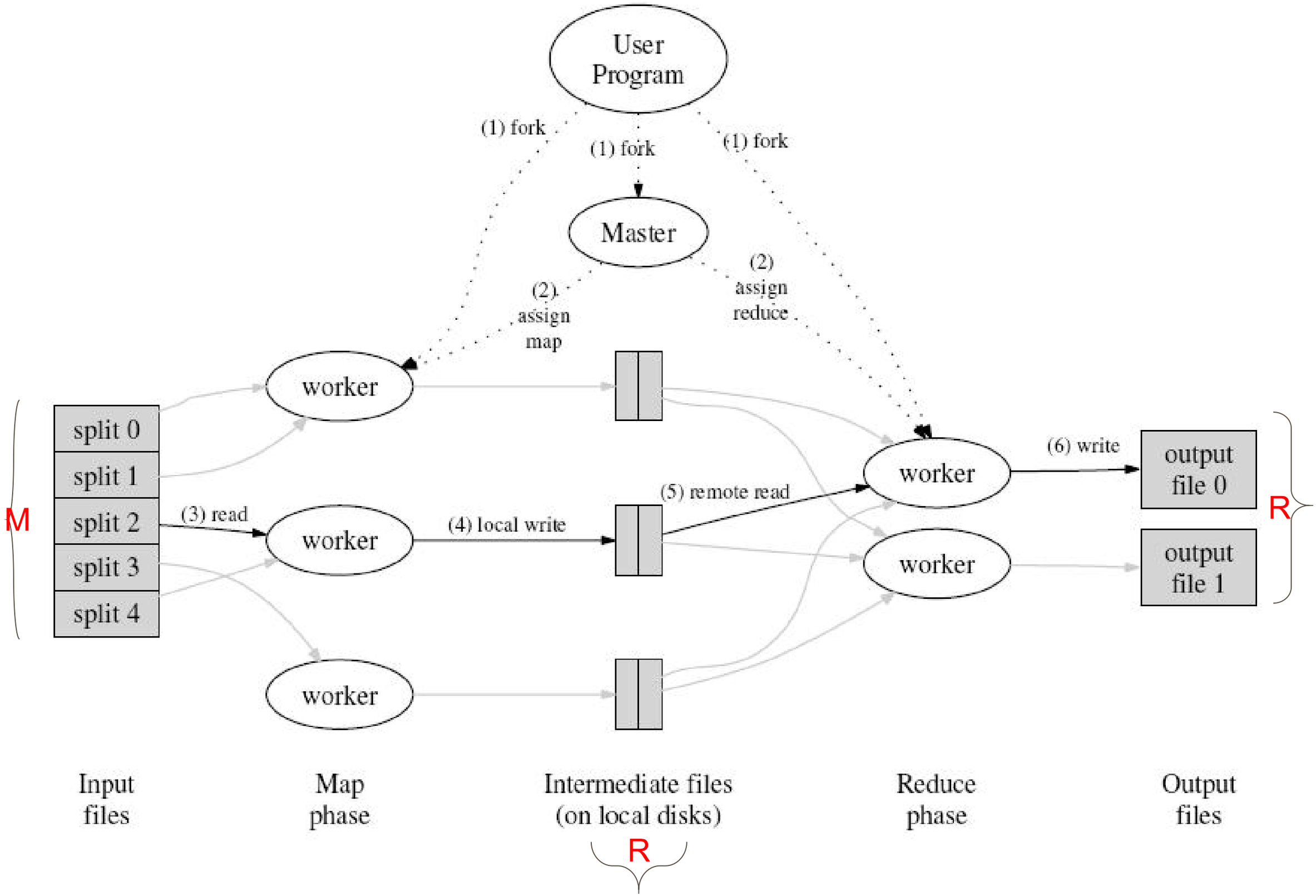
    ```
    for all w in value do emit(w,1)
    ```

  - Reduce:

    ```
    emit(key,sum(value*))
    ```

Implementation Overview

- Typical cluster
  - 100's/1000s of 2-CPU x86 machines, 2-4 GB of memory
  - Storage is on local disks
  - GFS: distributed file system manages data
  - Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machine


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
M
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
R
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files
R

Execution

- How is this distributed?
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. Now partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!
- Bottleneck: reduce phase can't start until map phase is completely finished.

Locality

- Master program divvies 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

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

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

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