



Lecture 32

Distributed Computing and MapReduce

CS 61C, Fall 2024 @ UC Berkeley

Slides credit: Dan Garcia, Justin Yokota, Peyrin Kao

Slides template credit: Josh Hug, Lisa Yan

Scope Changes

Lecture 32, CS 61C, Fall 2024

Scope Changes

Coordination Game

- Demo
- Debrief

Distributed Computing

- Definition
- Challenges

Examples

- Manager-Worker Framework
- MapReduce

A brief history:

- This lecture was introduced in 2023 as **Process-Level Parallelism (PLP)**.
- It used to have a lot of syntax. Look up **OpenMPI** if you're curious.

Changes for this semester:

- We're blending PLP (2023) and MapReduce (a long-time special topics lecture).
- It's now more conceptual: More about a way of thinking, than specific syntax.

Exam scope for Fall 2024:

- You should know the basic concepts behind distributed computing.
- You don't need to know the specifics of MapReduce (it's just an example).
- The first half of today will be lightly tested on exams (1–2 true/false questions).

See recording for the live demo.
No slides in this section.

Coordination Game Demo

Lecture 32, CS 61C, Fall 2024

Scope Changes

Coordination Game

- **Demo**
- Debrief

Distributed Computing

- Definition
- Challenges

Examples

- Manager-Worker Framework
- MapReduce

Slides will be added after lecture.

Coordination Game Debrief

Lecture 32, CS 61C, Fall 2024

Scope Changes

Coordination Game

- Demo
- **Debrief**

Distributed Computing

- Definition
- Challenges

Examples

- Manager-Worker Framework
- MapReduce

Coordination Game Debrief

Communicating – didn't work.

Lag time is a killer – when you say something, there's a delay before others hear it.

Also, not everyone hears what you say.

Multiple people trying to do the same thing.

Some people are trying to sabotage.

Not everyone is following the same plan.

Someone initially volunteered to be the boss.

Distributed Computing: Definition

Lecture 32, CS 61C, Fall 2024

Scope Changes

Coordination Game

- Demo
- Debrief

Distributed Computing

- **Definition**
- Challenges

Examples

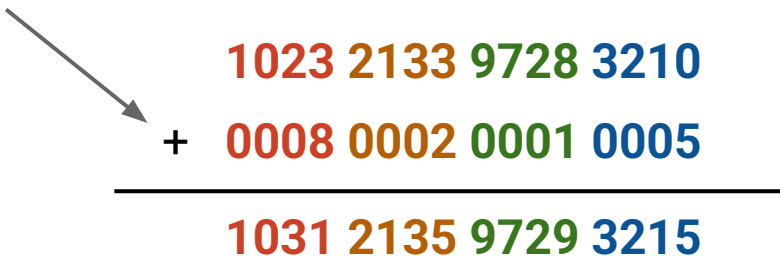
- Manager-Worker Framework
- MapReduce

Review: Types of Parallelism So Far

SIMD (Single Instruction, Multiple Data):

Use one instruction to operate on multiple pieces of data at the same time.

Use a single addition operation to
add four pairs of numbers.



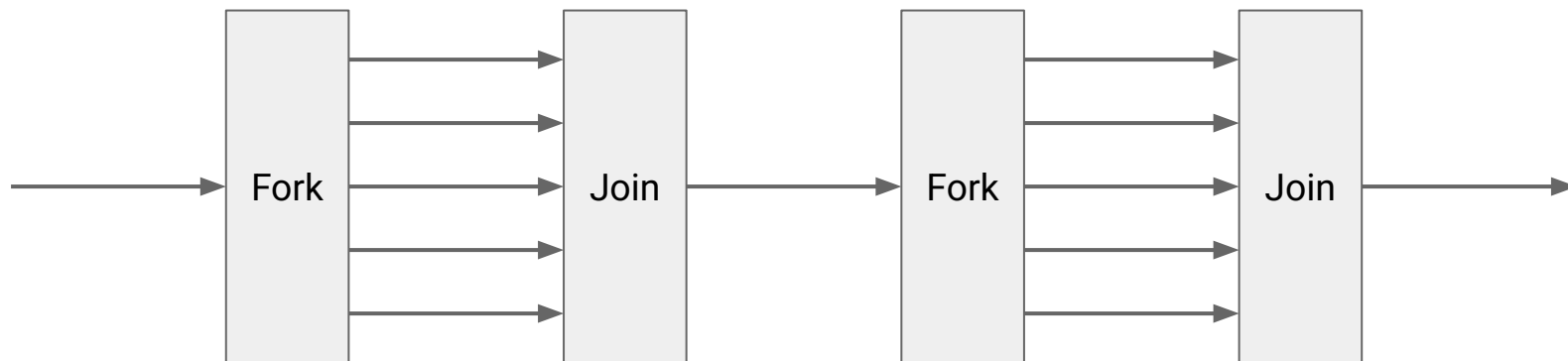
A diagram illustrating SIMD (Single Instruction, Multiple Data) parallelism. It shows a single addition operation (+) being applied to four pairs of numbers simultaneously. The numbers are color-coded: red for the first pair, orange for the second, green for the third, and blue for the fourth. A grey arrow points from the text 'Use a single addition operation to add four pairs of numbers.' to the '+' sign.

	1023	2133	9728	3210
+	0008	0002	0001	0005
<hr/>				
	1031	2135	9729	3215

Review: Types of Parallelism So Far

Thread-Level Parallelism (TLP):

One program runs multiple threads simultaneously.



Today's new type of parallelism: **Distributed computing**.

- Everything we've seen so far is parallelizing a single program.
- Today: Many different programs work together to achieve a common goal.

Analogy:

- One person multi-tasking = parallelizing a single program.
- Many people working together = distributed computing.

To scale up distributed programming: Just add more computers.

- Lots of cheap, commodity computers can work together to solve hard problems.

Distributed Computing: Challenges

Lecture 32, CS 61C, Fall 2024

Scope Changes

Coordination Game

- Demo
- Debrief

Distributed Computing

- Definition
- **Challenges**

Examples

- Manager-Worker Framework
- MapReduce

Challenges of Distributed Computing

Concurrency is hard.

- Separate programs usually don't share memory.
- Sharing state between different programs is hard.
- Hard to use locks.

Handling failure is hard.

- Single program: If a thread crashes, the whole program crashes.
- Distributed computing: If one program crashes, the others keep going.
- Can create "zombie processes" that keep running after everyone else is finished.
Consumes resources until we restart the system.

Challenges of Distributed Computing

Communication is hard.

- Programs coordinate by sending messages between each other.
- Messages can take time to transmit (e.g. over the Internet).
It takes time to start communication, transmit bytes of data, access memory, etc.
- What if Program A sends a message to Program B...
...but Program B isn't expecting to receive a message?

Goal: Split problem into independent sub-tasks, and minimize communication between programs.

Example: Manager-Worker Framework

Lecture 32, CS 61C, Fall 2024

Scope Changes

Coordination Game

- Demo
- Debrief

Distributed Computing

- Definition
- Challenges

Examples

- **Manager-Worker Framework**
- MapReduce

Manager-Worker Framework: Motivation

Suppose we have 20 independent tasks, and 4 programs.

Naive approach: Give 5 tasks to each program.

Program 0:

- Task 1
- Task 2
- Task 3
- Task 4
- Task 5

Program 1:

- Task 6
- Task 7
- Task 8
- Task 9
- Task 10

Program 2:

- Task 11
- Task 12
- Task 13
- Task 14
- Task 15

Program 3:

- Task 16
- Task 17
- Task 18
- Task 19
- Task 20

Problem: Might not load-balance well.

- What if Task 17 takes 5x as long as every other task?
- What if the tasks were sorted by how long they take?

Need some way to dynamically assign work, while minimizing communication.

Program 0 is the lone **manager**, whose job is assigning work to the other processes.

- Manager itself doesn't do actual work.

Manager pseudocode:

1. Setup.

2. While there's still work to do:

- Wait for a worker to request more work.
- Find the next task to do.
- Tell that worker what task to do.

3. Repeat once per worker:

- Wait for a worker to request more work.
- Tell that worker we're all done.

4. Teardown.

All other programs are workers, who receive work from the manager.

Worker pseudocode:

1. Setup.
2. `done=False`.
3. While `done==False`:
 - Send manager a request for work.
 - Receive message from manager.
 - If message is work: Do the work.
 - If message is "all done": Set `done=True`.
4. Teardown.

Manager-Worker Framework: Design Considerations

Transmitting messages can be slow/expensive, so messages should be short.

- Example: Manager sends a filename of a file with the task instructions.
- Example: Manager just sends the ID of the task to do.
- Example: Manager sending `-1` means "all done."
- Don't forget to document your message protocol for users!

Manager is "wasted" not doing any work, but that's a good idea.

- If manager is stuck on a hard task, that would stall the workers.
- The manager coordinating everyone else minimizes concurrency issues.
Example: We don't need to worry about two programs doing the same task.

Manager-Worker Framework: Design Considerations

Assumption so far: Tasks are independent, and can be done in any order.

What if tasks have dependencies?

- Manager maintains queue of tasks that can be done right now.
- If queue is empty, and worker requests a task, tell the worker to come back later.

Need to write two versions of code: manager code, and worker code.

Not in exam scope in Fall 2024.

Example: MapReduce

Lecture 32, CS 61C, Fall 2024

Scope Changes

Coordination Game

- Demo
- Debrief

Distributed Computing

- Definition
- Challenges

Examples

- Manager-Worker Framework
- **MapReduce**

What is MapReduce?

Simple data-parallel programming model designed for scalability and fault-tolerance.

Pioneered by Google.

- Google processes >25 petabytes of data per day!

Open-source Hadoop project.

- Used at Yahoo!, Facebook, Amazon, etc.



What is MapReduce Used For?

At Google:

- Index construction for Google Search.
- Article clustering for Google News.
- Statistical machine translation.
- For computing multi-layer street maps.

At Yahoo!:

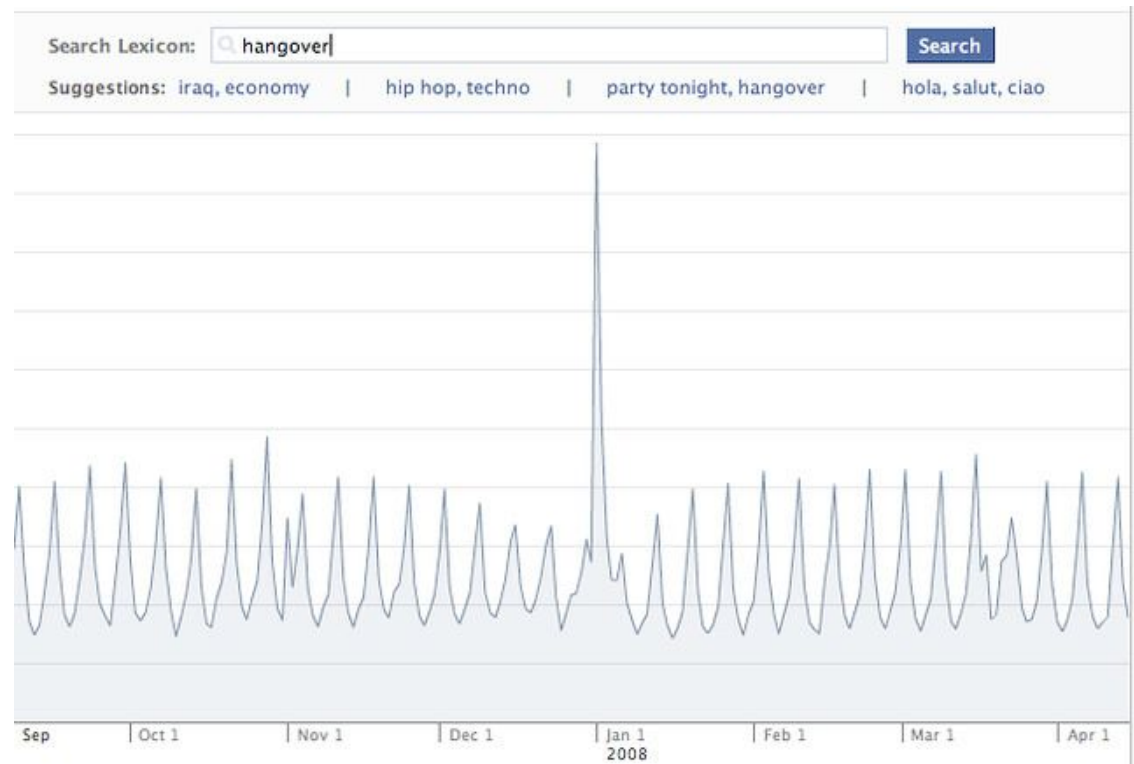
- "Web map" powering Yahoo! search.
- Spam detection for Yahoo! Mail.

At Facebook:

- Data mining.
- Ad optimization.
- Spam detection.

Example: Facebook Lexicon

Tracking usage of the term "hangover" across time would require reading tons of Facebook posts and collecting word counts.



Scalability to large data volumes:

- 1000s of machines, 10000s of disks.

Cost-efficiency:

- Commodity machines (cheap, but unreliable).
- Commodity network.
- Automatic fault-tolerance via re-execution (fewer administrators).
- Easy, fun to use (fewer programmers).

Jeffrey Dean and Sanjay Ghemawat, "[MapReduce: Simplified Data Processing on Large Clusters](#)," 6th USENIX Symposium on Operating Systems Design and Implementation, 2004.

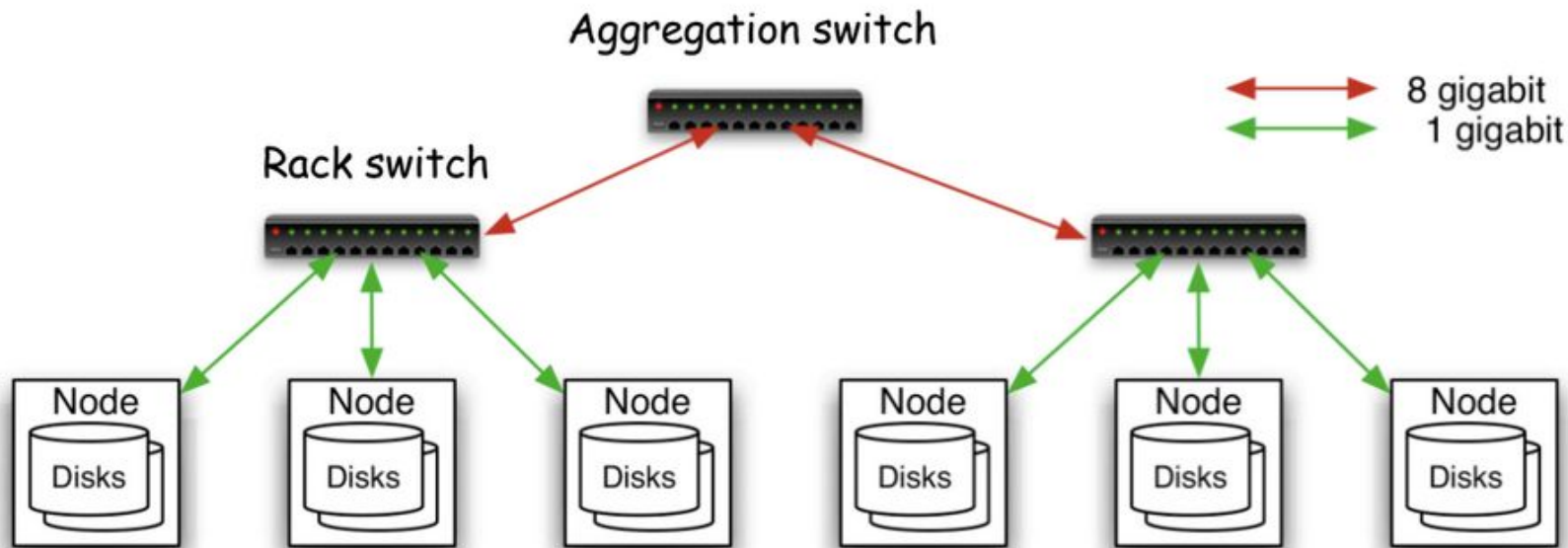
- Optional reading. A digestible CS paper at the 61C level!

Typical Hadoop Cluster

40 nodes/rack, 1000-4000 nodes in cluster.

1 Gbps bandwidth within rack, 8 Gbps out of rack.

Node specs (Yahoo terasort): 8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)



MapReduce in CS 10 and CS 61A

The MapReduce paradigm can be implemented in several different languages:

Snap! (CS 10 programming language)

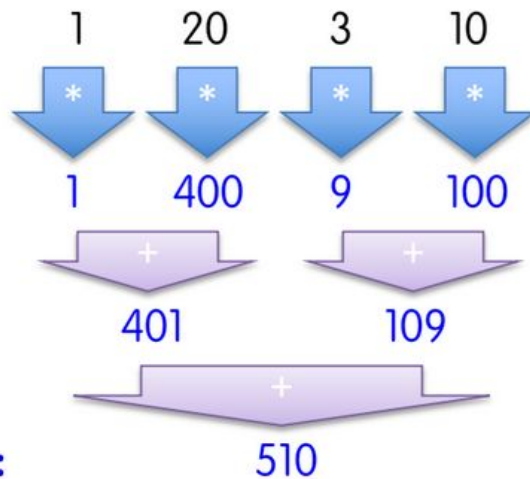


Scheme (CS 61A programming language)

```
> (reduce +  
      (map square '(1 20 3 10))  
      510)
```

MapReduce consists of two steps: Map, then Reduce.

Input:



Note:
only
two
data
types!

Output:

510

MapReduce in CS 10 and CS 61A

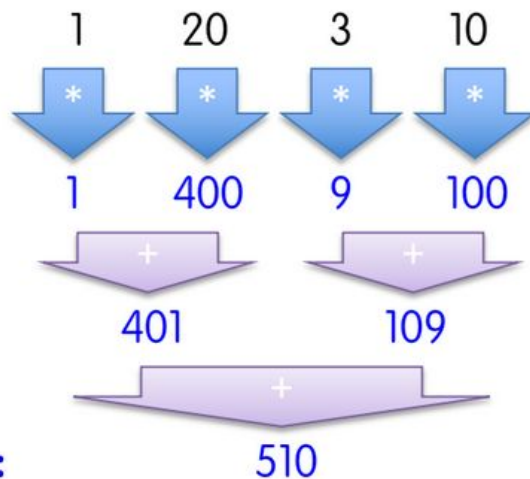
The MapReduce paradigm can be implemented in several different languages:

Python (CS 61A programming language)

```
>>> from functools import reduce
>>> def plus(x,y): return x+y
>>> def square(x): return x*x
>>> reduce(plus,
           map(square, (1,20,3,10)))
510
```

MapReduce consists of two steps: Map, then Reduce.

Input:



Note:
only
two
data
types!

Output:

510

MapReduce Programming Model

Input and output: each a set of key/value pairs.

Programmer specifies two functions:

`map (in_key, in_value) → list(intermediate_key, intermediate_value)`

- Processes input key/value pair.
- Slices data into "shards" or "splits"; distributed to workers.
- Produces set of intermediate pairs.

`reduce (intermediate_key, list(intermediate_value)) → list(out_value)`

- Combines all intermediate values for a particular key.
- Produces a set of merged output values (usually just one).

MapReduce Example: Word Count

"Mapper" nodes are responsible for the map function.

```
// "I do I learn" → ("I",1), ("do",1), ("I",1), ("learn",1)
map(String input_key, String input_value):
    // input_key   : document name (or line of text)
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");
```

MapReduce Example: Word Count

"Reducer" nodes are responsible for the reduce function.

```
// ("I",[1,1]) → ("I",2)
```

```
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:
```

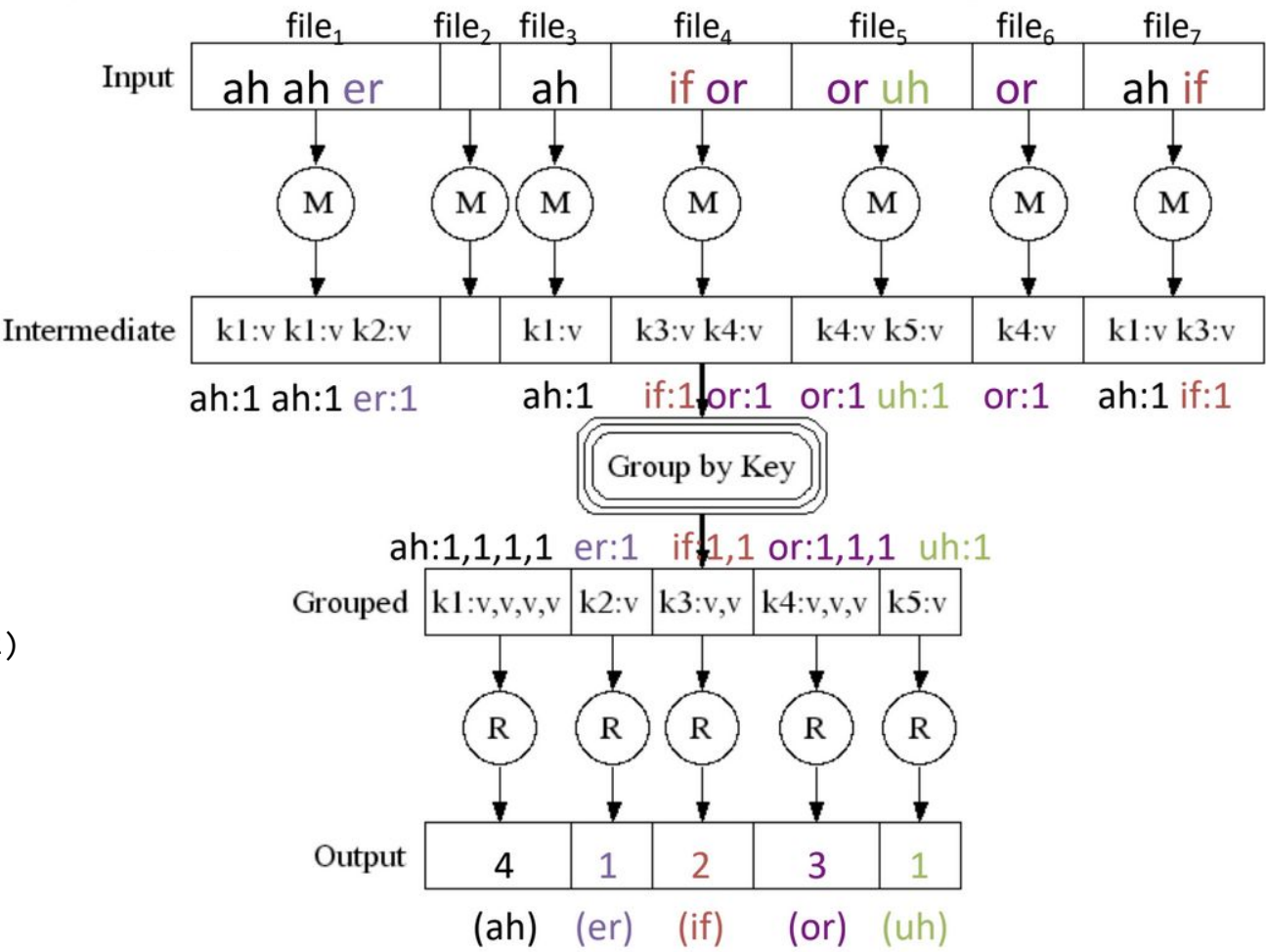
```
        result += ParseInt(v);
```

```
    Emit(AsString(result));
```

Data is stored on a distributed file system (DFS).

MapReduce WordCount Diagram

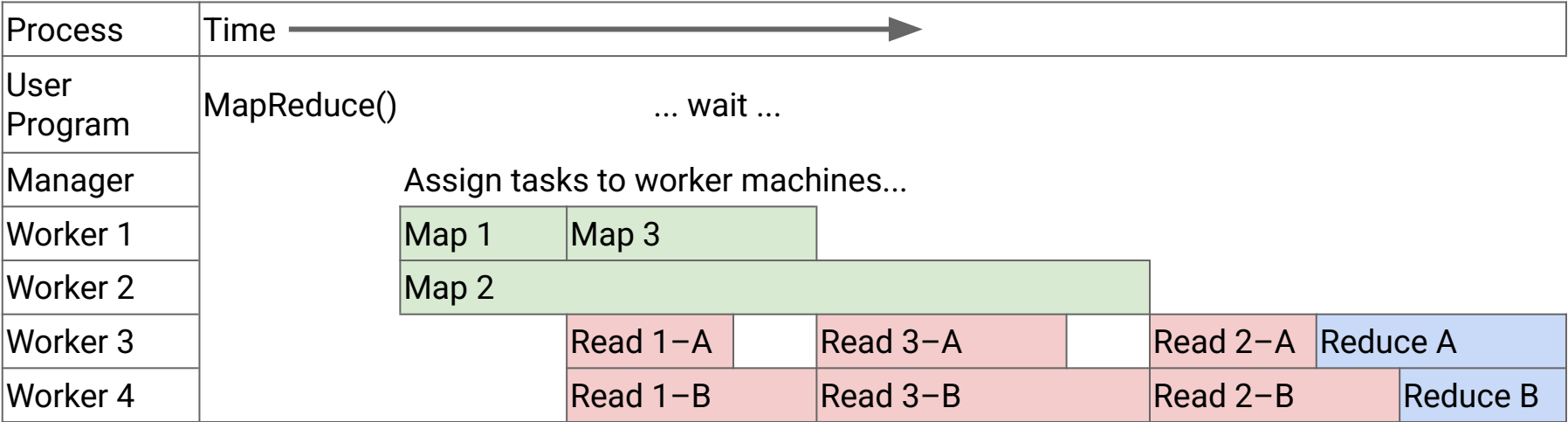
Map:
"I do I learn" →
("I",1), ("do",1),
("I",1), ("learn",1)



Reduce:
("I",[1,1]) → ("I",2)

MapReduce Processing Time Line

- Manager assigns Map + Reduce tasks to "worker" servers.
- As soon as a Map task finishes, worker can be assigned a new Map or Reduce task.
- Data shuffle begins as soon as a given Map finishes.
- Reduce task begins as soon as all data shuffles finish.
- To tolerate faults, reassign task if a worker server goes down.



MapReduce WordCount Java code

(You don't need to understand this code.)

```
public static void main(String[] args)
throws IOException {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(WCMap.class);
    conf.setCombinerClass(WCReduce.class);
    conf.setReducerClass(WCReduce.class);
    conf.setInputPath(new Path(args[0]));
    conf.setOutputPath(new Path(args[1]));
    JobClient.runJob(conf);
}
```

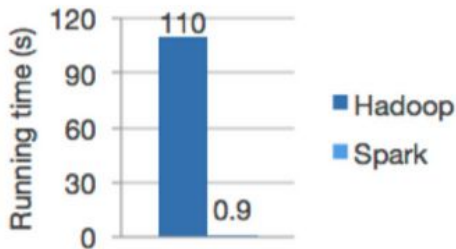
```
public class WCMap extends MapReduceBase implements Mapper {
    private static final IntWritable ONE = new IntWritable(1);
    public void map(WritableComparable key, Writable value,
                    OutputCollector output,
                    Reporter reporter) throws IOException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            output.collect(new Text(itr.next()), ONE);
        }
    }
}

public class WCReduce extends MapReduceBase implements Reducer {
    public void reduce(WritableComparable key, Iterator values,
                       OutputCollector output,
                       Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += ((IntWritable) values.next()).get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

Apache Spark is a fast and general engine for large-scale data processing.

Speed:

- Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
- Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.



Ease of Use:

- Write applications quickly in Java, Scala or Python.
- Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala and Python shells.



Word Count in Spark's Python API

```
file.flatMap(lambda line: line.split())  
      .map(lambda word: (word, 1))  
      .reduceByKey(lambda a, b: a + b)
```



One-line solution!

flatMap in Spark's Python API

```
>>> def neighbor(n):  
...     return [n-1,n,n+1]
```

```
>>> R = sc.parallelize(range(5))
```

```
>>> R.collect()
```

```
[0, 1, 2, 3, 4]
```

```
>>> R.map(neighbor).collect()
```

```
[[-1, 0, 1], [0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5]]
```

```
>>> R.flatMap(neighbor).collect()
```

```
[-1, 0, 1, 0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5]
```

Word Count in Spark's Python API

```
unix% cat file.txt
```

```
ah ah er
```

```
ah
```

```
if or
```

```
or uh
```

```
or
```

```
ah if
```

Word Count in Spark's Python API

```
>>> W = sc.textFile("file.txt")
```

```
>>> W.flatMap(lambda line: line.split()).collect()
```

```
['ah', 'ah', 'er', 'ah', 'if', 'or', 'or', 'uh', 'or', 'ah', 'if']
```

```
>>> W.flatMap(lambda line: line.split()).map(lambda word:  
(word,1)).collect()
```

```
[('ah', 1), ('ah', 1), ('er', 1), ('ah', 1), ('if', 1), ('or', 1),  
( 'or', 1), ('uh', 1), ('or', 1), ('ah', 1), ('if', 1)]
```

```
>>> W.flatMap(lambda line: line.split()).map(lambda word:  
(word,1)).reduceByKey(lambda a,b: a+b).collect()
```

```
[('er', 1), ('ah', 4), ('if', 2), ('or', 3), ('uh', 1)]
```

Parallel? Let's Check...

```
>>> def crunch(n):  
...     time.sleep(5) # to simulate number crunching  
...     return n*n  
...  
>>> crunch(10) # 5 seconds later  
100  
  
>>> list(map(crunch,range(4))) # 20 seconds later  
[0, 1, 4, 9]  
  
>>> R = sc.parallelize(range(4))  
>>> R.map(crunch).collect() # 5 seconds later  
[0, 1, 4, 9]
```

Summary: Distributed Computing

Distributed Computing: Many computers work together to achieve a common goal.

- Easy to scale up, e.g. with commodity computers.
- If one computer crashes, the others keep running.
- Communication is slow and hard. Ideally, subtasks should be independent.

Manager-Worker Framework:

- Manager assigns work, doesn't do any actual work.
- Workers are assigned work to do.

MapReduce:

- Gives programmers an abstraction for distributed computing.
- Map and Reduce operations performed in parallel on key-value pairs.
- Work is distributed between computers, using manager-worker framework.