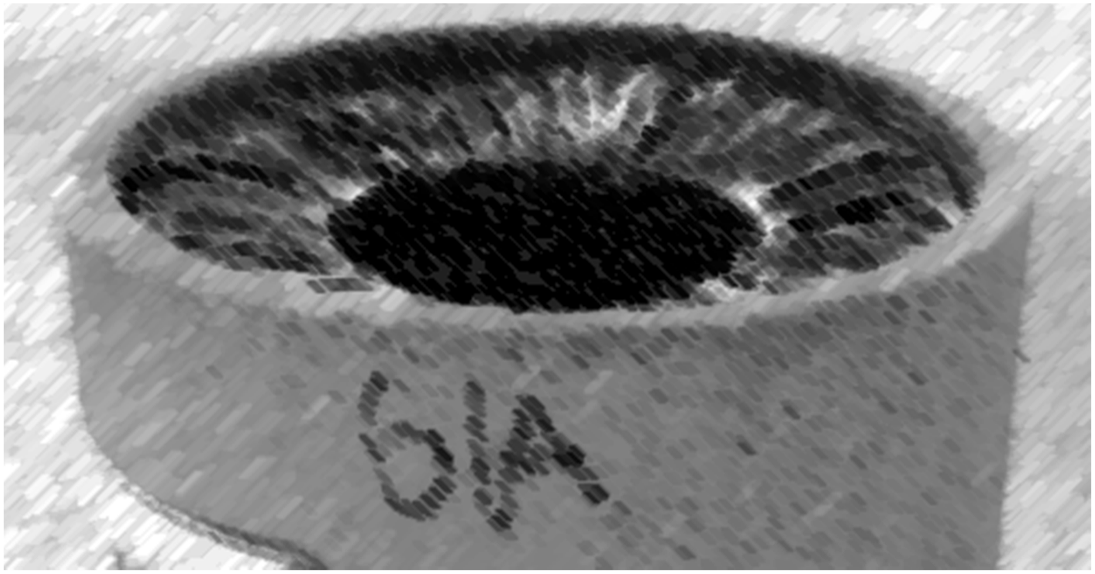


CS61A Lecture 41

Amir Kamil

UC Berkeley

April 26, 2013

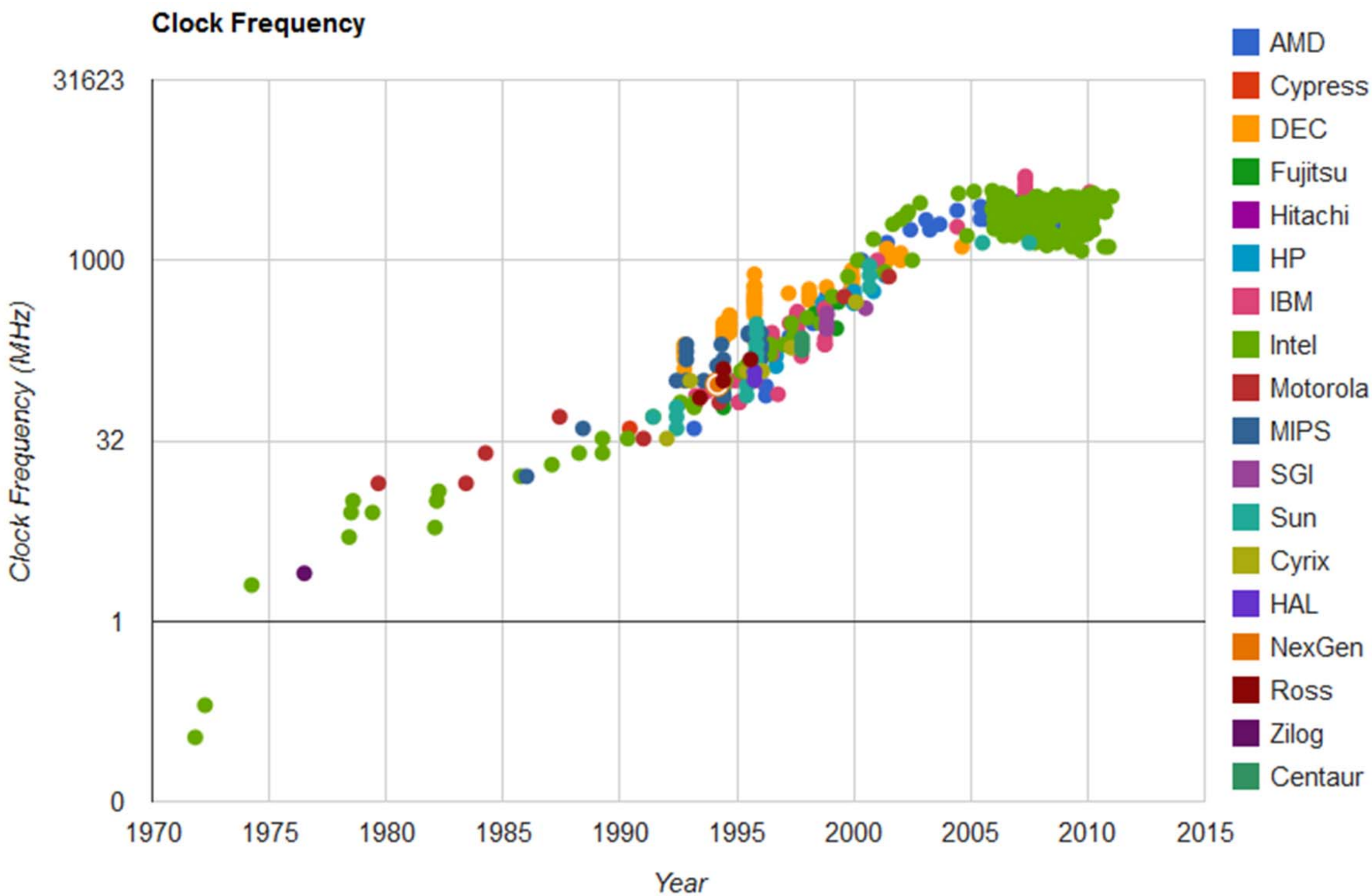
Announcements

- HW13 due Wednesday
- Scheme project due Monday
- Scheme contest deadline extended to Friday

CPU Performance

Performance of individual CPU cores has largely stagnated in recent years

Graph of CPU clock frequency, an important component in CPU performance:



http://cpudb.stanford.edu

Parallelism

Applications must be *parallelized* in order run faster

- Waiting for a faster CPU core is no longer an option

Parallelism is easy in functional programming:

- When a program contains only pure functions, call expressions can be evaluated in any order, lazily, and in parallel
- Referential transparency: a call expression can be replaced by its value (or *vice versa*) without changing the program

But not all problems can be solved efficiently using functional programming

Today: the easy case of parallelism, using only pure functions

- Specifically, we will look at *MapReduce*, a framework for such computations

Next time: the hard case, where shared data is required

MapReduce

MapReduce is a *framework* for batch processing of Big Data

What does that mean?

- **Framework**: A system used by programmers to build applications
- **Batch processing**: All the data is available at the outset, and results aren't used until processing completes
- **Big Data**: A buzzword used to describe data sets so large that they reveal facts about the world via statistical analysis

The MapReduce idea:

- Data sets are too big to be analyzed by one machine
- When using multiple machines, systems issues abound
- Pure functions enable an abstraction barrier between data processing logic and distributed system administration

Systems

Systems research enables the development of applications by defining and implementing abstractions:

- **Operating systems** provide a stable, consistent interface to unreliable, inconsistent hardware

- **Networks** provide a simple, robust data transfer interface to constantly evolving communications infrastructure

- **Databases** provide a declarative interface to software that stores and retrieves information efficiently

- **Distributed systems** provide a single-entity-level interface to a cluster of multiple machines

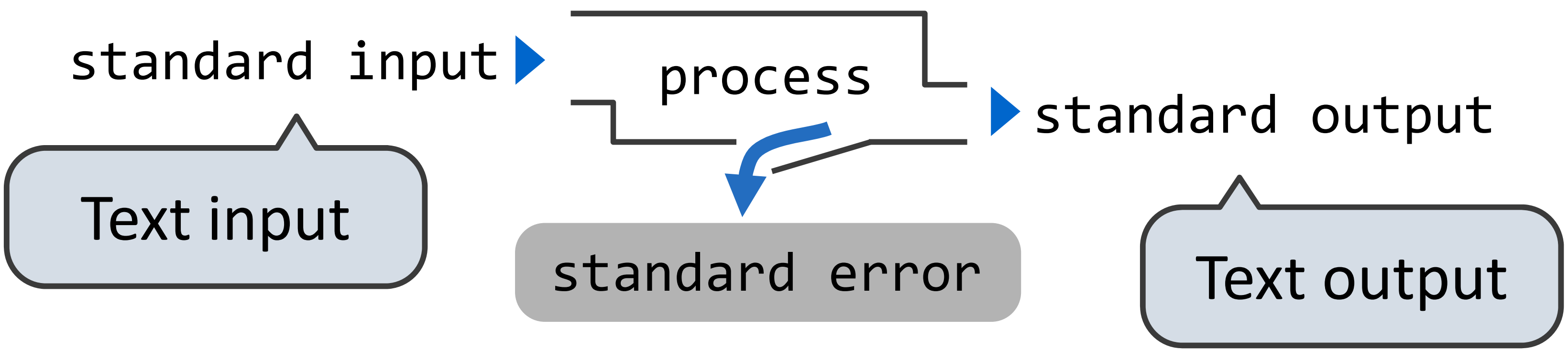
A unifying property of effective systems:

Hide *complexity*, but retain *flexibility*

The Unix Operating System

Essential features of the Unix operating system (and variants):

- **Portability**: The same operating system on different hardware
- **Multi-Tasking**: Many processes run concurrently on a machine
- **Plain Text**: Data is stored and shared in text format
- **Modularity**: Small tools are composed flexibly via pipes



The ***standard streams*** in a Unix-like operating system
are conceptually similar to Python iterators

Python Programs in a Unix Environment

The built-in `input` function reads a line from *standard input*

The built-in `print` function writes a line to *standard output*

The values `sys.stdin` and `sys.stdout` also provide access to the Unix *standard streams* as "files"

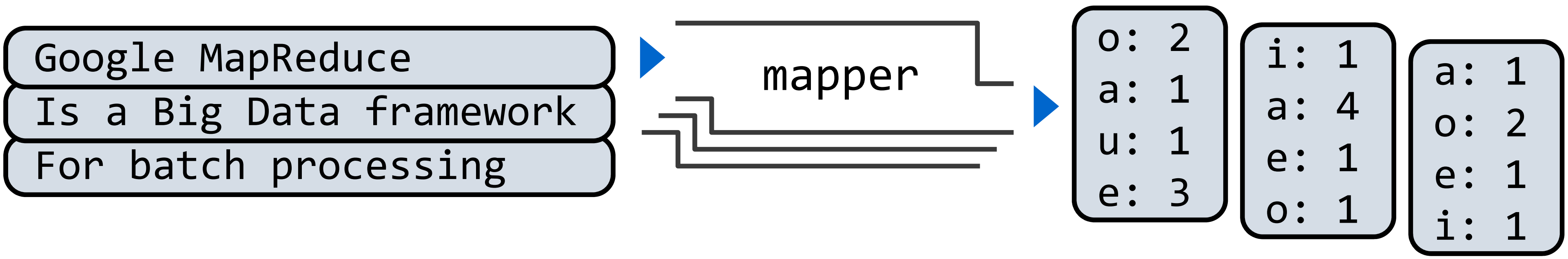
A Python "file" is an interface that supports iteration, read, and write methods

Using these "files" takes advantage of the operating system *standard stream* abstraction

MapReduce Evaluation Model

Map phase: Apply a *mapper* function to inputs, emitting a set of intermediate key-value pairs

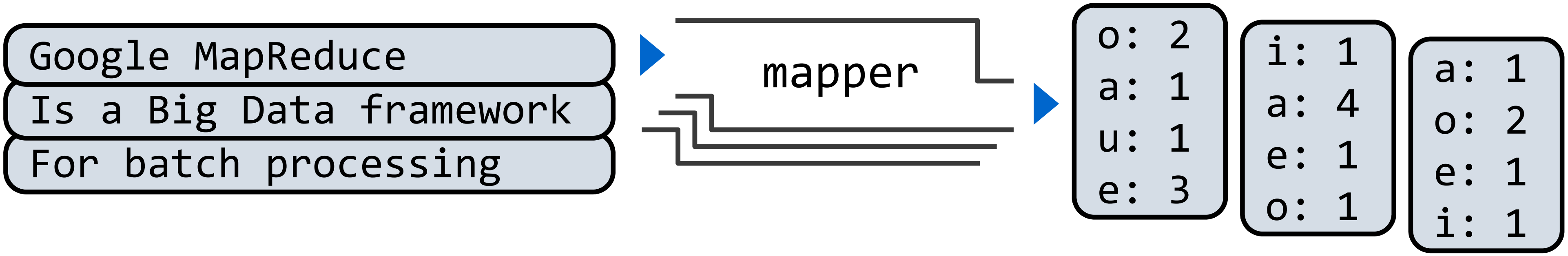
- The *mapper* takes an iterator over inputs, such as text lines
- The *mapper* yields zero or more key-value pairs per input



Reduce phase: For each intermediate key, apply a *reducer* function to accumulate all values associated with that key

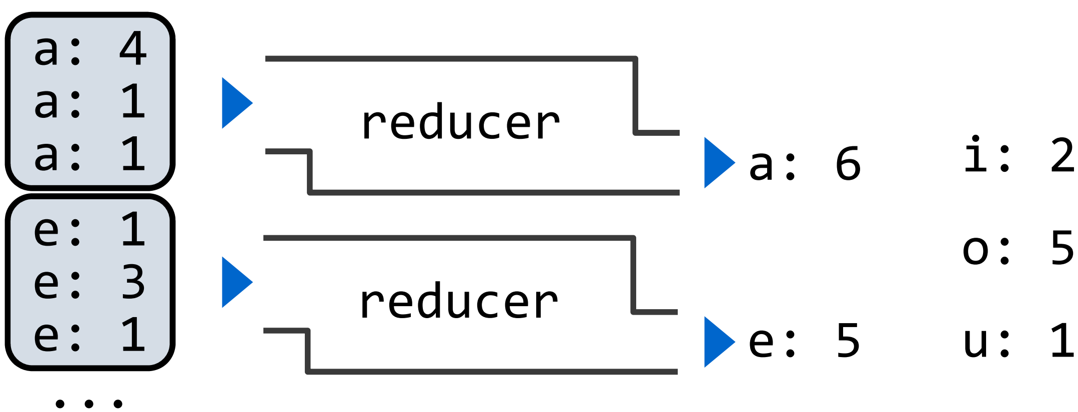
- The *reducer* takes an iterator over key-value pairs
- All pairs with a given key are consecutive
- The *reducer* yields 0 or more values, each associated with that intermediate key

MapReduce Evaluation Model

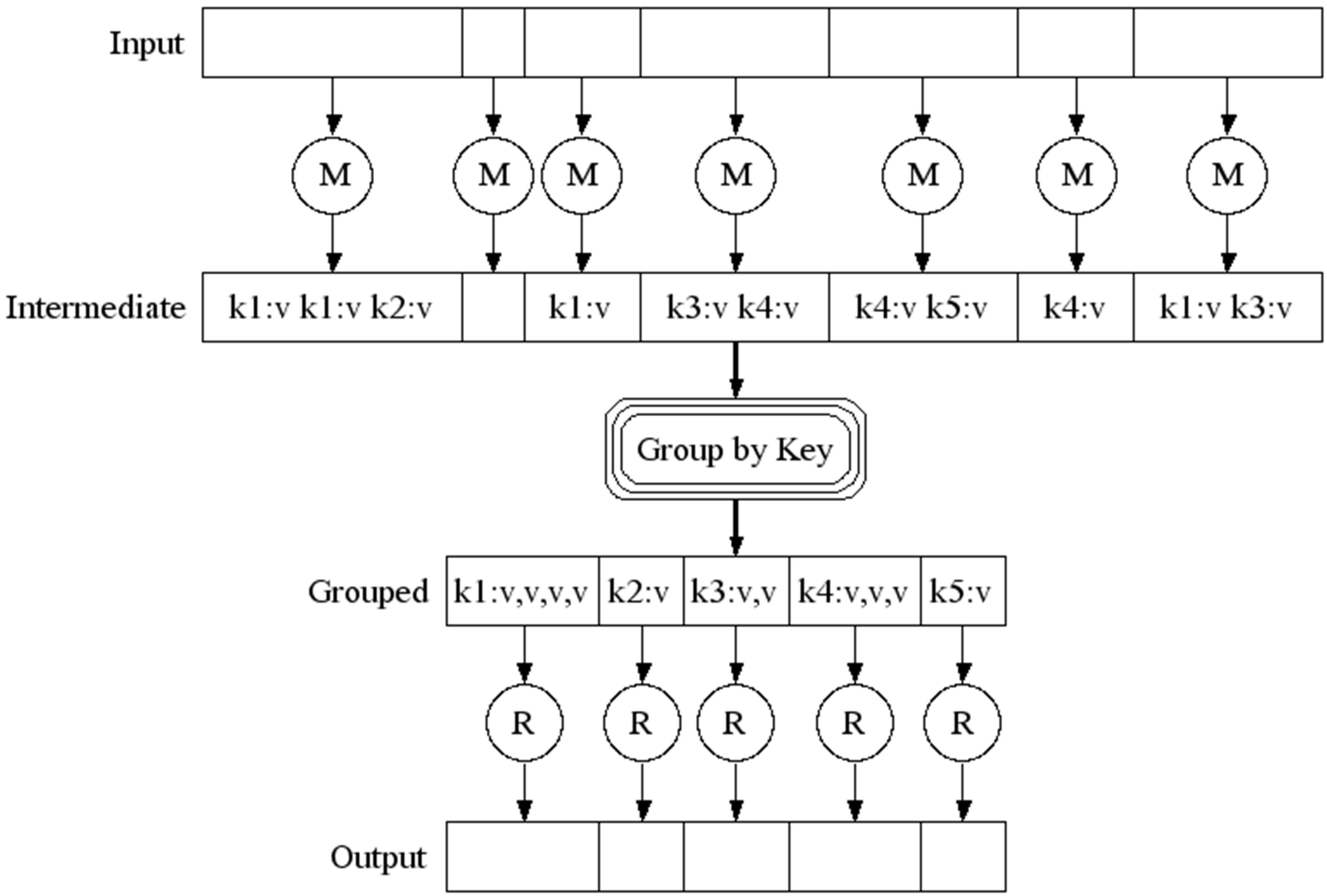


Reduce phase: For each intermediate key, apply a *reducer* function to accumulate all values associated with that key

- The *reducer* takes an iterator over key-value pairs
- All pairs with a given key are consecutive
- The *reducer* yields 0 or more values, each associated with that intermediate key

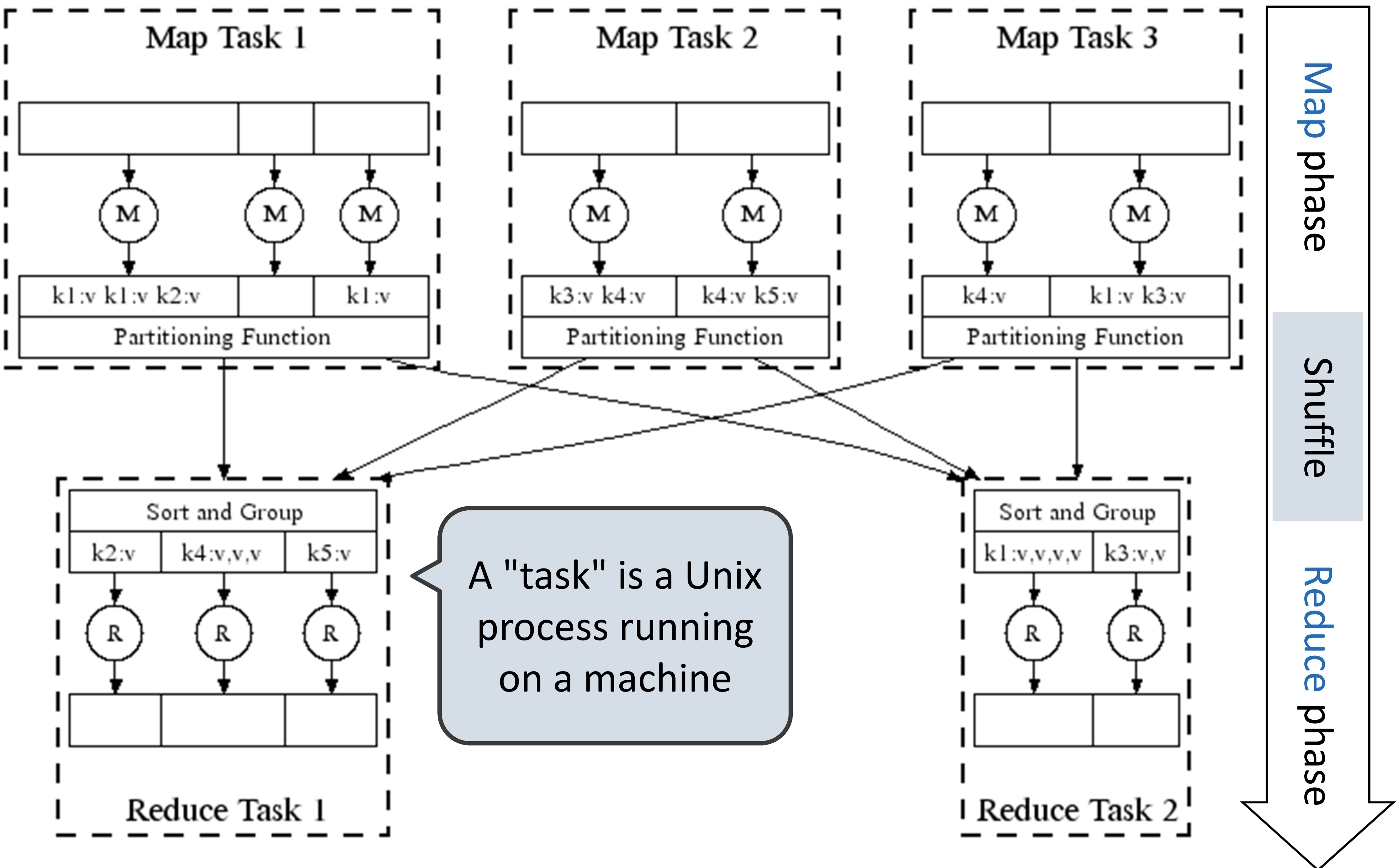


Above-the-Line: Execution Model



http://research.google.com/archive/mapreduce-osdi04-slides/index-auto-0007.html

Below-the-Line: Parallel Execution



http://research.google.com/archive/mapreduce-osdi04-slides/index-auto-0008.html

MapReduce Assumptions

Constraints on the *mapper* and reducer:

- The *mapper* must be equivalent to applying a deterministic pure function to each input independently
- The *reducer* must be equivalent to applying a deterministic pure function to the sequence of values for each key

Benefits of functional programming:

- When a program contains only pure functions, call expressions can be evaluated in any order, lazily, and in parallel
- Referential transparency: a call expression can be replaced by its value (or *vice versa*) without changing the program

In MapReduce, these functional programming ideas allow:

- Consistent results, however computation is partitioned
- Re-computation and caching of results, as needed

Python Example of a MapReduce Application

The *mapper* and *reducer* are both self-contained Python programs

- Read from *standard input* and write to *standard output*!

Mapper

```
#!/usr/bin/env python3

import sys
from ucb import main
from mapreduce import emit

def emit_vowels(line):
    for vowel in 'aeiou':
        count = line.count(vowel)
        if count > 0:
            emit(vowel, count)

for line in sys.stdin:
    emit_vowels(line)
```

Tell Unix: this is Python

The **`emit`** function outputs a key and value as a line of text to standard output

Mapper inputs are lines of text provided to standard input

Python Example of a MapReduce Application

The *mapper* and *reducer* are both self-contained Python programs

- Read from *standard input* and write to *standard output*!

Reducer

```
#!/usr/bin/env python3

import sys
from ucb import main
from mapreduce import emit, group_values_by_key

for key, value_iterator in group_values_by_key(sys.stdin):
    emit(key, sum(value_iterator))
```

Takes and returns iterators

Input: lines of text representing key-value pairs, grouped by key

Output: Iterator over (key, value_iterator) pairs that give all values for each key

What the MapReduce Framework Provides

Fault tolerance: A machine or hard drive might crash

- The MapReduce framework automatically re-runs failed tasks

Speed: Some machine might be slow because it's overloaded

- The framework can run multiple copies of a task and keep the result of the one that finishes first

Network locality: Data transfer is expensive

- The framework tries to schedule map tasks on the machines that hold the data to be processed

Monitoring: Will my job finish before dinner?!?

- The framework provides a web-based interface describing jobs