COEN6731 Distributed Software Systems

Week 7: MapReduce

Gengrui (Edward) Zhang, PhD
Web: gengruizhang.com

Today's outline

Distributed Computing

MPI

MapReduce

TP & AP

Transaction Processing

- Often called OLTP (Online Transaction Processing)

Analytical Processing

- Often called OLAP (Online Analytical Processing)

TP & AP

Transaction Processing

- Often called OLTP (Online Transaction Processing)

```
UPDATE accounts
SET balance = balance - 100
WHERE id = 123;
```

More “System” oriented

Analytical Processing

- Often called OLAP (Online Analytical Processing)

TP & AP

Transaction Processing

- Often called OLTP (Online Transaction Processing)

```
UPDATE accounts
SET balance = balance - 100
WHERE id = 123;
```

More “System” oriented

Analytical Processing

- Often called OLAP (Online Analytical Processing)

```
SELECT region, SUM(sales)
FROM orders
GROUP BY region;
```

More “Computing” oriented

TP runs the business,

while AP analyzes the business

A simple task: word count

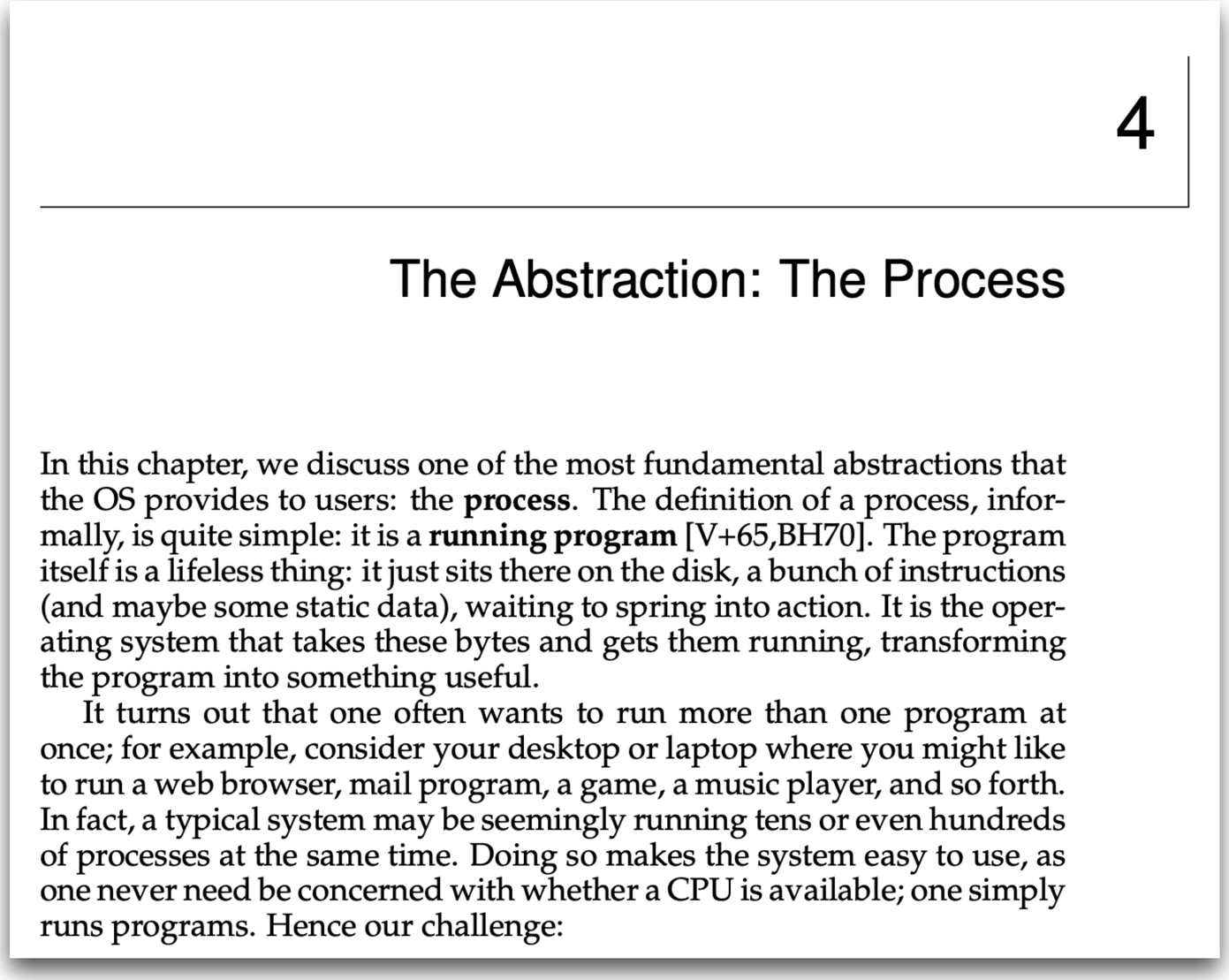
4

The Abstraction: The Process

In this chapter, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program** [V+65,BH70]. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where you might like to run a web browser, mail program, a game, a music player, and so forth. In fact, a typical system may be seemingly running tens or even hundreds of processes at the same time. Doing so makes the system easy to use, as one never need be concerned with whether a CPU is available; one simply runs programs. Hence our challenge:

A simple task: word count



4

The Abstraction: The Process

In this chapter, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program** [V+65,BH70]. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where you might like to run a web browser, mail program, a game, a music player, and so forth. In fact, a typical system may be seemingly running tens or even hundreds of processes at the same time. Doing so makes the system easy to use, as one never need be concerned with whether a CPU is available; one simply runs programs. Hence our challenge:

<key, value>

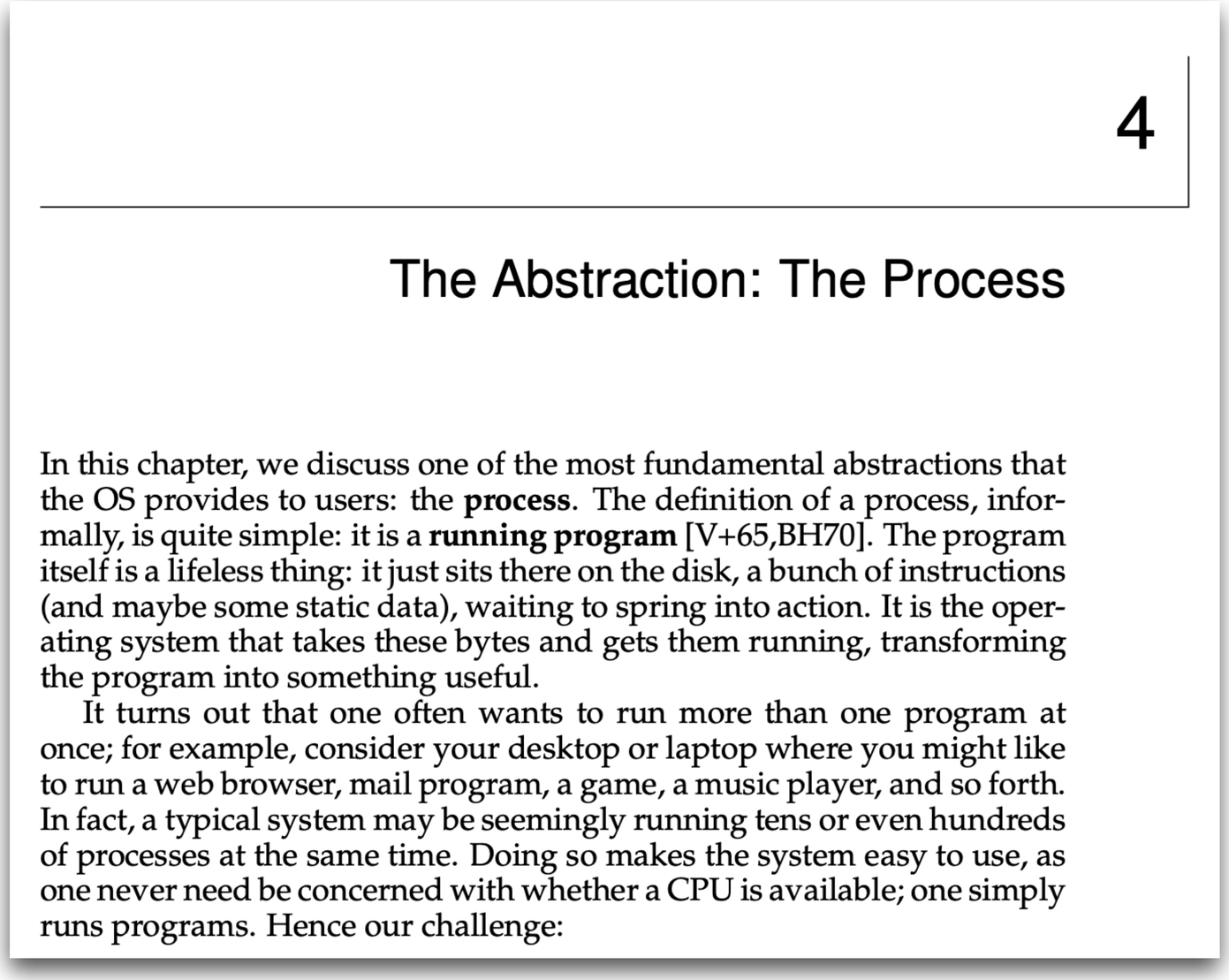
Word Occurrence

< “this”, 3 >

< “chapter”, 2 >

…

A simple task: word count



4

The Abstraction: The Process

In this chapter, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program** [V+65,BH70]. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where you might like to run a web browser, mail program, a game, a music player, and so forth. In fact, a typical system may be seemingly running tens or even hundreds of processes at the same time. Doing so makes the system easy to use, as one never need be concerned with whether a CPU is available; one simply runs programs. Hence our challenge:

<key, value>

Word Occurrence

< “this”, 3 >

< “chapter”, 2 >

…

How would you count words in 10TB of data across 100 machines?

Distributedly counting words

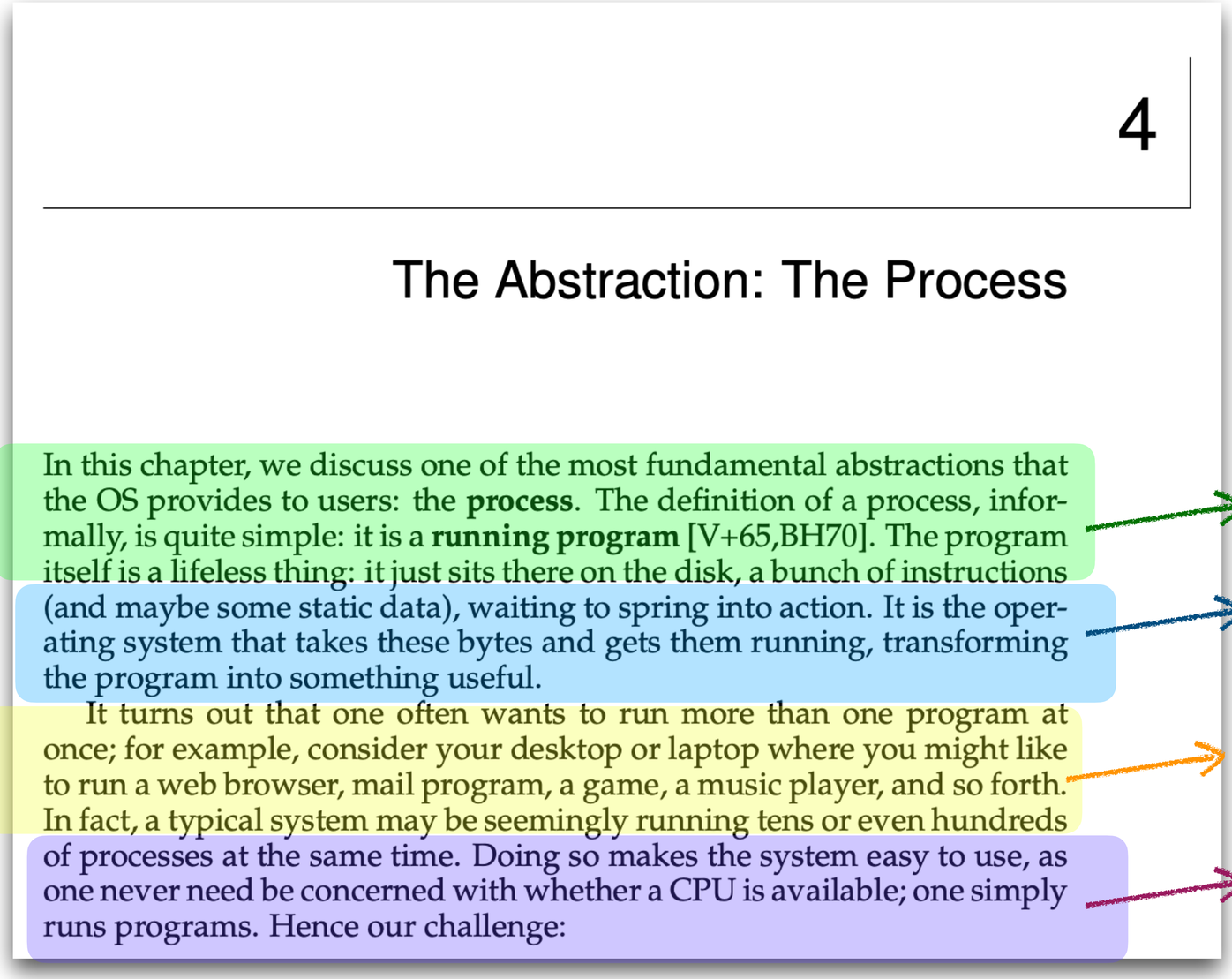
4

The Abstraction: The Process

In this chapter, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program** [V+65,BH70]. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where you might like to run a web browser, mail program, a game, a music player, and so forth. In fact, a typical system may be seemingly running tens or even hundreds of processes at the same time. Doing so makes the system easy to use, as one never need be concerned with whether a CPU is available; one simply runs programs. Hence our challenge:

Distributedly counting words



4

The Abstraction: The Process

In this chapter, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program** [V+65,BH70]. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where you might like to run a web browser, mail program, a game, a music player, and so forth. In fact, a typical system may be seemingly running tens or even hundreds of processes at the same time. Doing so makes the system easy to use, as one never need be concerned with whether a CPU is available; one simply runs programs. Hence our challenge:

Task 1

Task 2

Task 3

Task 4

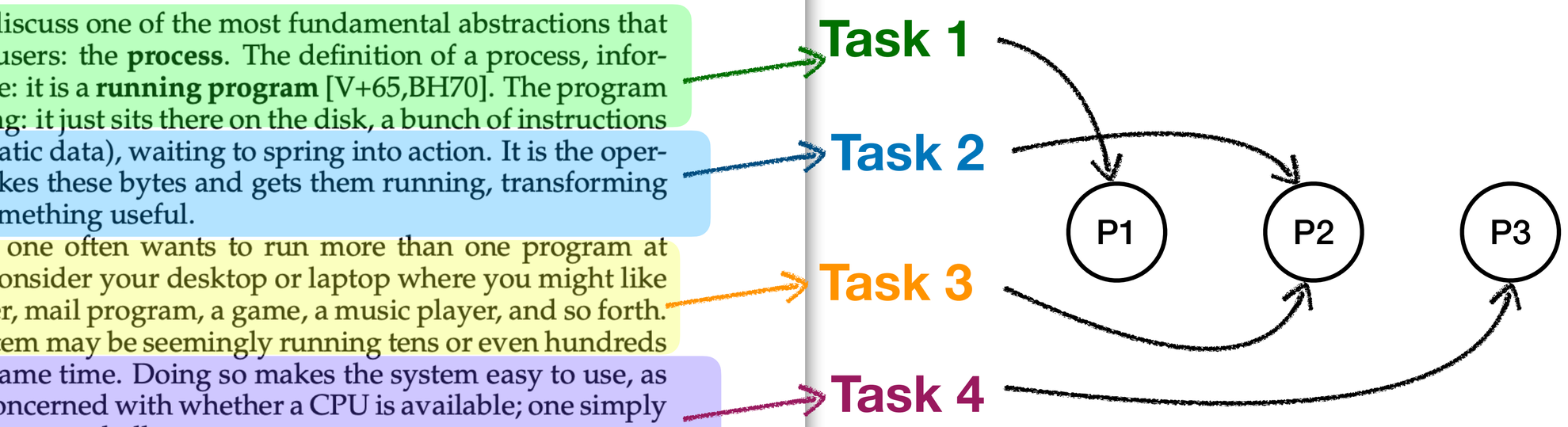
Distributedly counting words

4

The Abstraction: The Process

In this chapter, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program** [V+65,BH70]. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where you might like to run a web browser, mail program, a game, a music player, and so forth. In fact, a typical system may be seemingly running tens or even hundreds of processes at the same time. Doing so makes the system easy to use, as one never need be concerned with whether a CPU is available; one simply runs programs. Hence our challenge:



Distributedly counting words

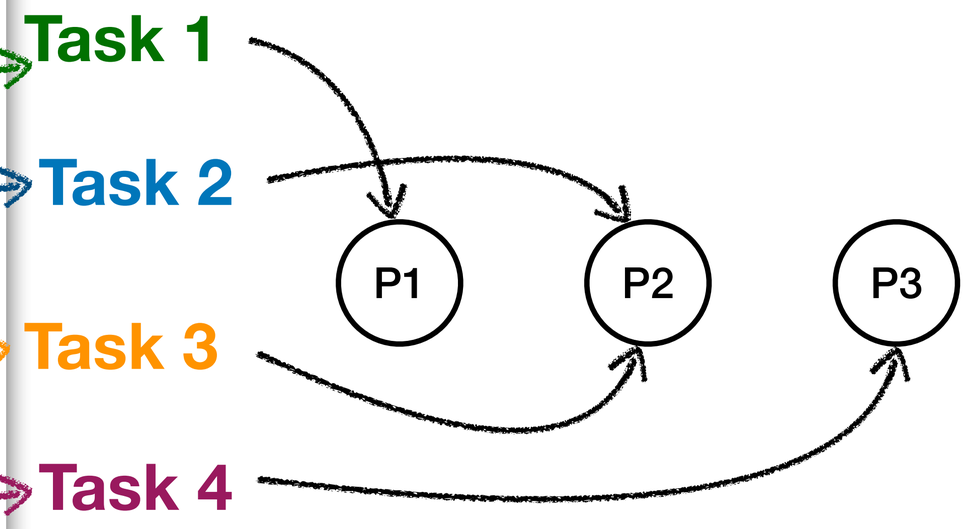
4

The Abstraction: The Process

In this chapter, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program** [V+65,BH70]. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where you might like to run a web browser, mail program, a game, a music player, and so forth. In fact, a typical system may be seemingly running tens or even hundreds of processes at the same time. Doing so makes the system easy to use, as one never need be concerned with whether a CPU is available; one simply runs programs. Hence our challenge:

- Data partition
- Parallel execution
- Fault tolerance
- Aggregation
- Scheduling



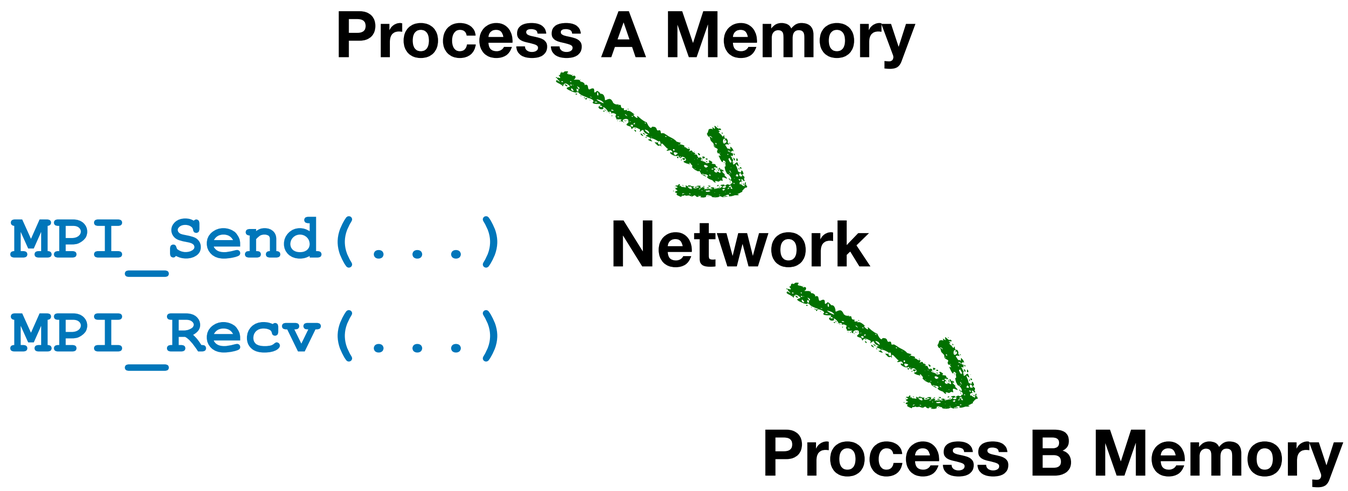
Message Passing Interface (MPI)

- MPI is a standard used for writing parallel processes and communicate with each other by sending messages
  - Open MPI
- Commonly used in High-Performance Computing (HPC)

- Can start multiple separate processes
- Each process has its own memory
- They communicate by sending and receiving messages
- Each process has a unique ID

Distributed Word Count

- Assume we have n processes, a master and $n-1$ workers
- We must manually handle:
  - Data partitioning
  - Sending data
  - Local computation
  - Global aggregation
  - Synchronization



MapReduce

- MapReduce: Simplified Data Processing on Large Clusters
  - Takes a set of input key/value pairs
  - Produces a set of output key/value pairs

MapReduce

- MapReduce: Simplified Data Processing on Large Clusters
  - Takes a set of input key/value pairs
  - Produces a set of output key/value pairs
- Two functions:
  - **Map**
    - Takes an input pair and produces a set of **intermediate** key/value pairs
    - it groups together all intermediate values associated with the same intermediate key (say K) and passes them to Reduce()
  - **Reduce**
    - Accepts an intermediate key K and a set of values for that key
    - Merges together these values to form a possibly smaller set of values

MapReduce Example

- Back to counting # of words

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce Example

Doc1: "hello world hello"

Doc2: "hello mapreduce"

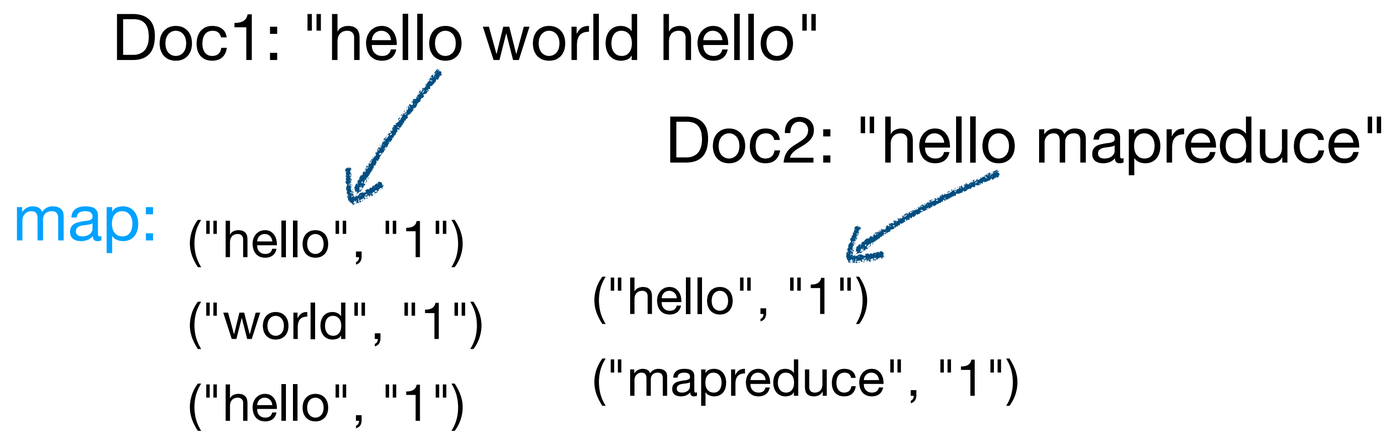
- Back to counting # of words

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce Example

- Back to counting # of words

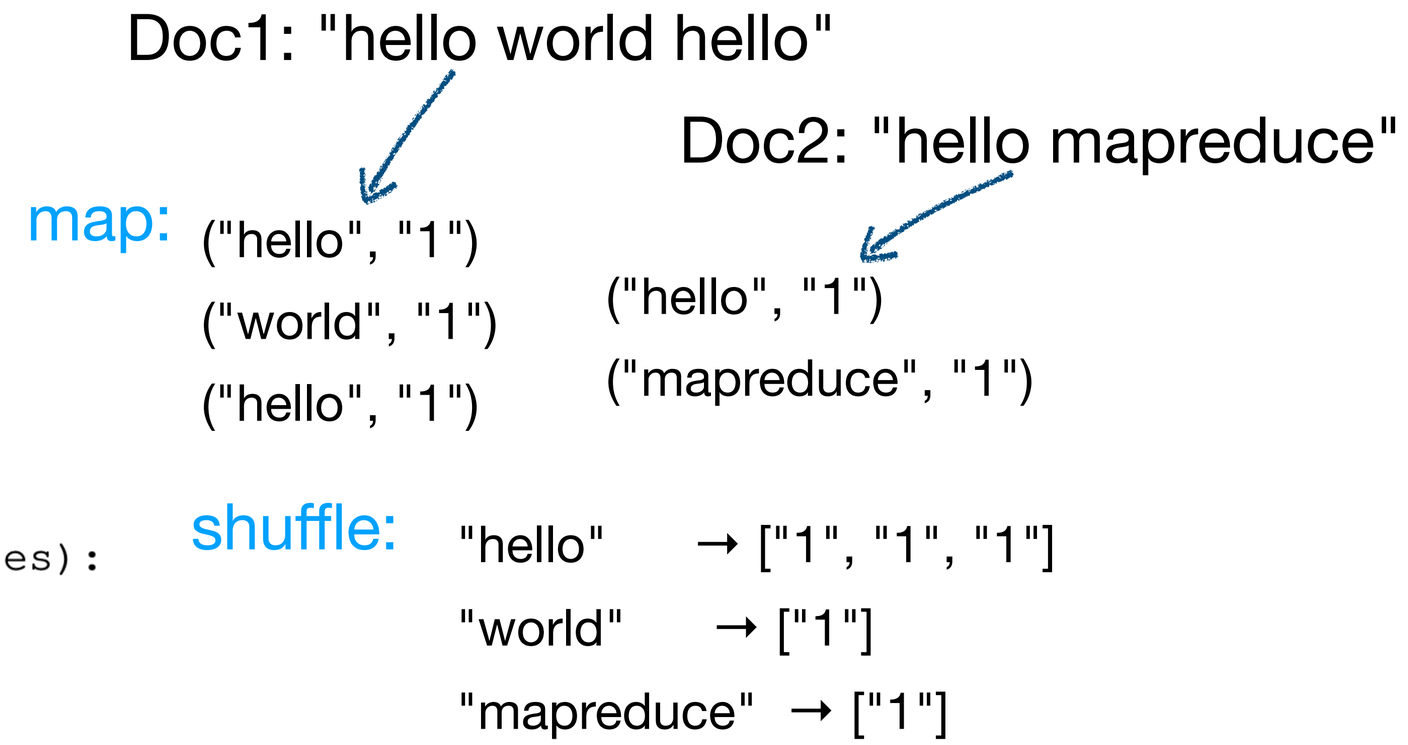
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```



MapReduce Example

- Back to counting # of words

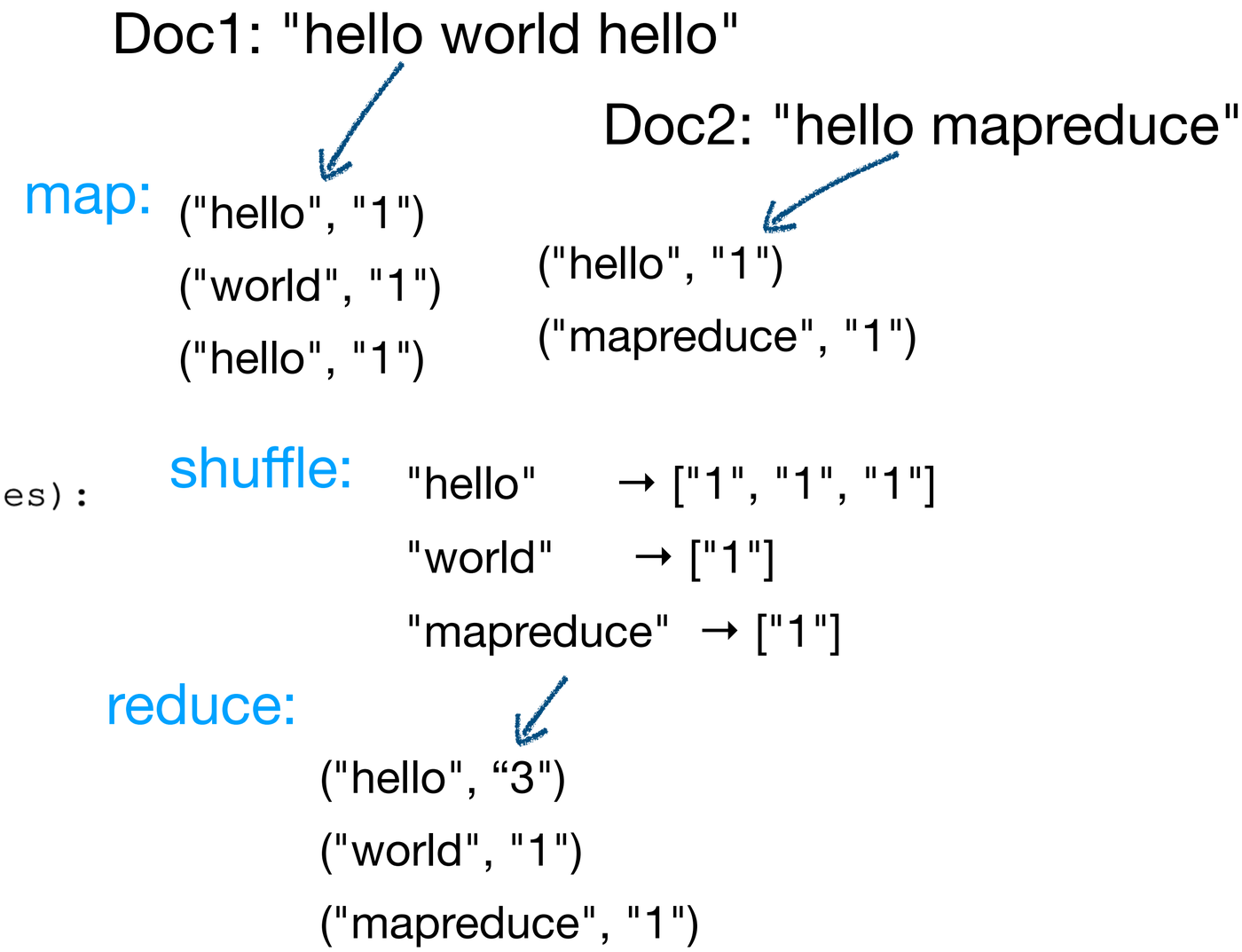
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```



MapReduce Example

- Back to counting # of words

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```



MapReduce Example

- Conceptually:

$$\begin{array}{lll} \textbf{map} & \texttt{(k1,v1)} & \rightarrow \texttt{list(k2,v2)} \\ \textbf{reduce} & \texttt{(k2,list(v2))} & \rightarrow \texttt{list(v2)} \end{array}$$

MapReduce Example

- Conceptually:

Original key and value

(e.g., k1: document name)

$$\begin{aligned} &\textbf{map} && (k1, v1) && \rightarrow list(k2, v2) \\ &\textbf{reduce} && (k2, list(v2)) && \rightarrow list(v2) \end{aligned}$$

MapReduce Example

- Conceptually:

Original key and value (e.g., k1: document name)

Intermediate key and value (e.g., k2: word)

$$\begin{aligned} \text{map} &\quad (k1, v1) &&\rightarrow \text{list}(k2, v2) \\ \text{reduce} &\quad (k2, \text{list}(v2)) &&\rightarrow \text{list}(v2) \end{aligned}$$

MapReduce Example

- Conceptually:

Original key and value

(e.g., k1: document name)

Intermediate key and value

(e.g., k2: word)

$$\begin{aligned} &\text{map} && (k1, v1) && \rightarrow \text{list}(k2, v2) \\ &\text{reduce} && (k2, \text{list}(v2)) && \rightarrow \text{list}(v2) \end{aligned}$$

A key and all values associated with that key

MapReduce Example

- Conceptually:

Original key and value

(e.g., k1: document name)

Intermediate key and value

(e.g., k2: word)

map (k1,v1) → list(k2,v2)

reduce (k2,list(v2)) → list(v2)

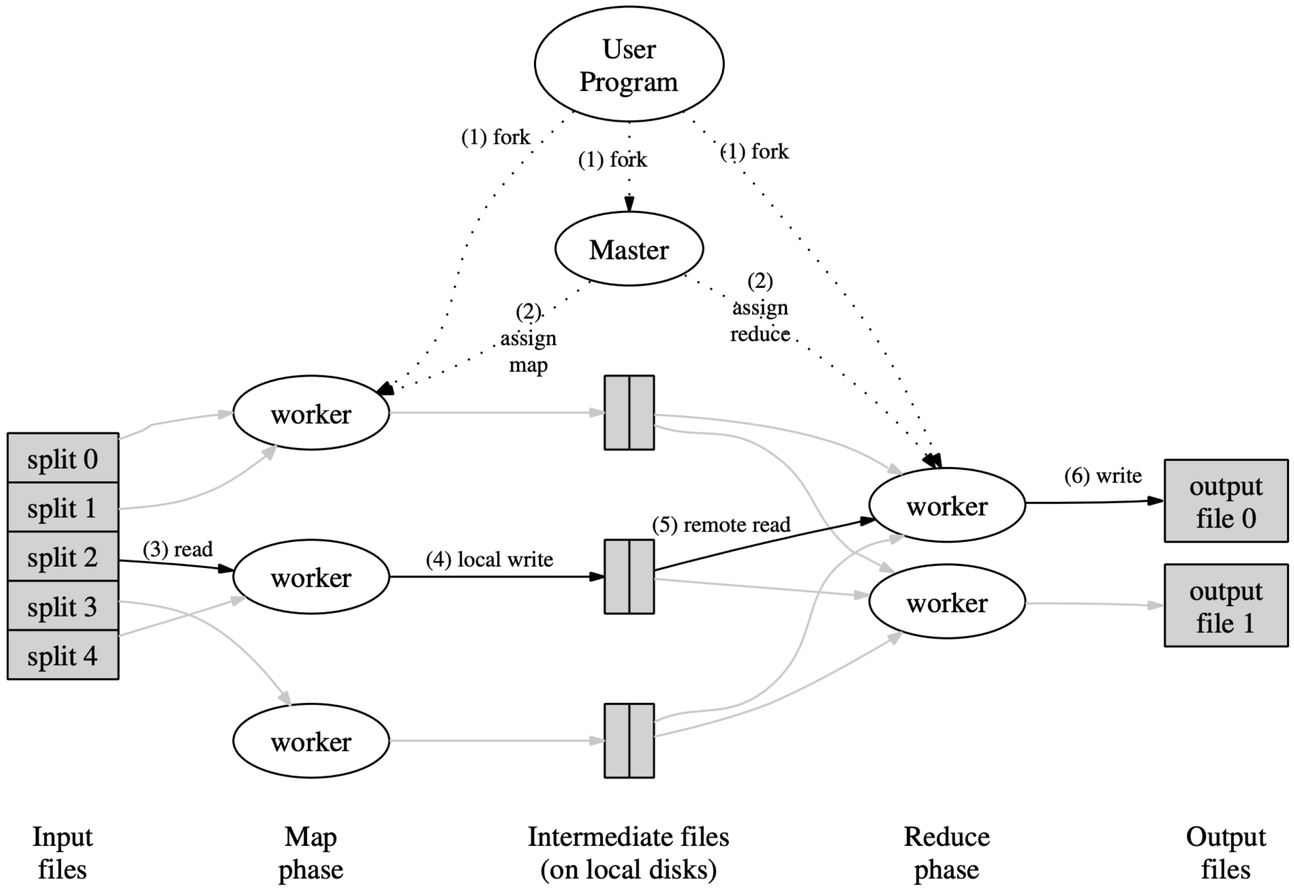
A key and all values associated with that key

map: transform records

reduce: aggregate by key

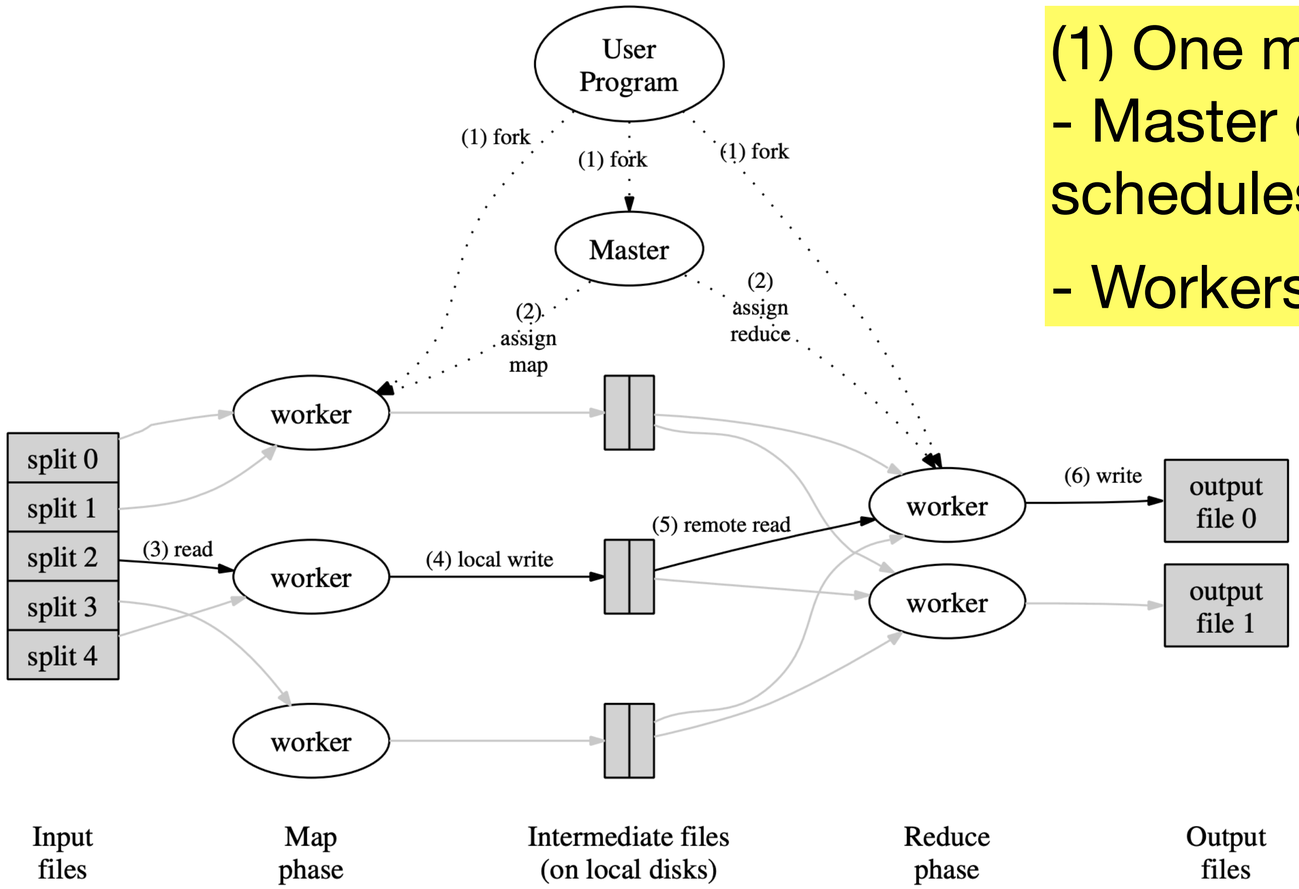
MapReduce

- MapReduce: Simplified Data Processing on Large Clusters



MapReduce

- MapReduce: Simplified Data Processing on Large Clusters

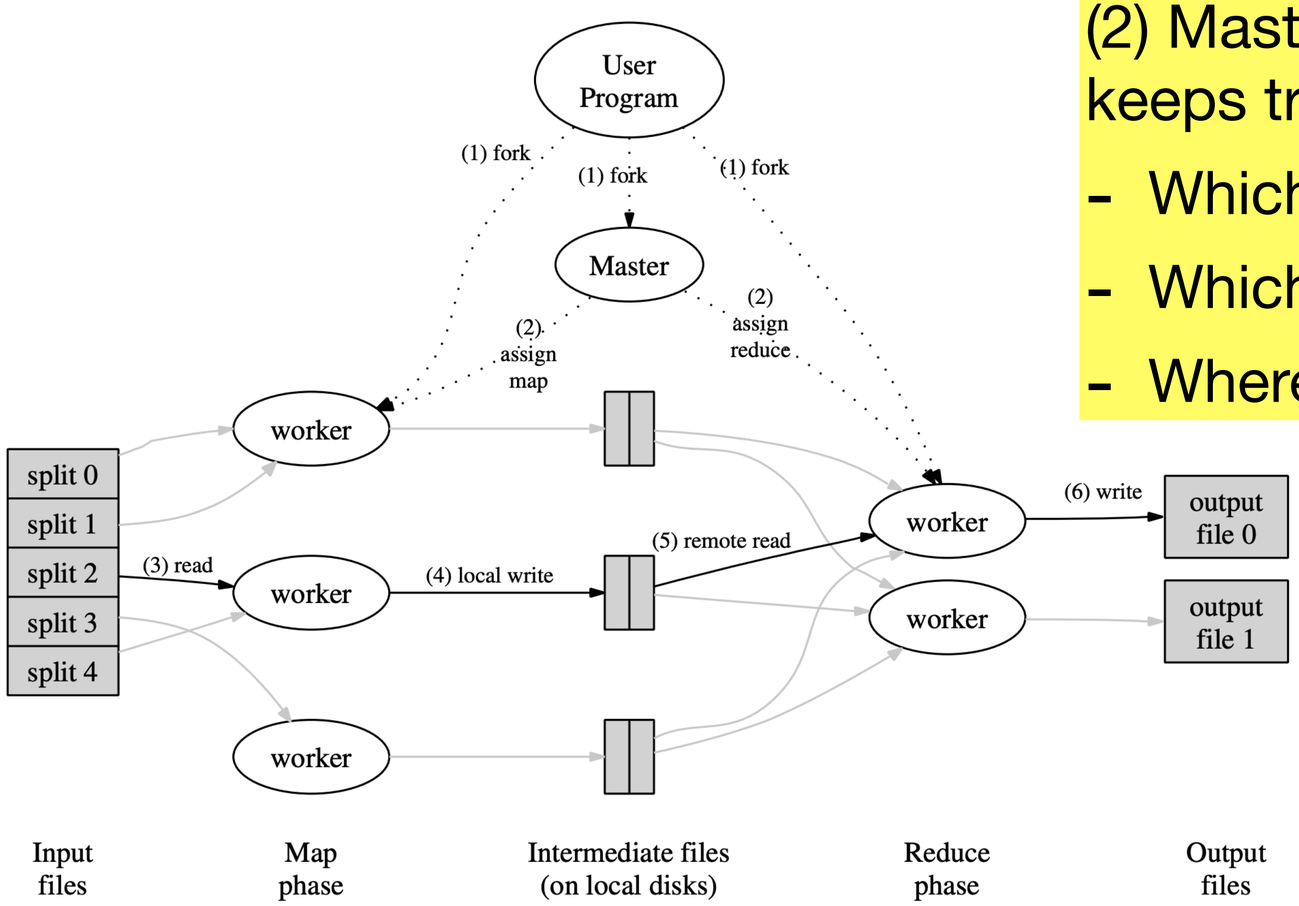


(1) One master; many workers.
- Master does NOT process data; it only schedules

- Workers execute map or reduce tasks

MapReduce

- MapReduce: Simplified Data Processing on Large Clusters

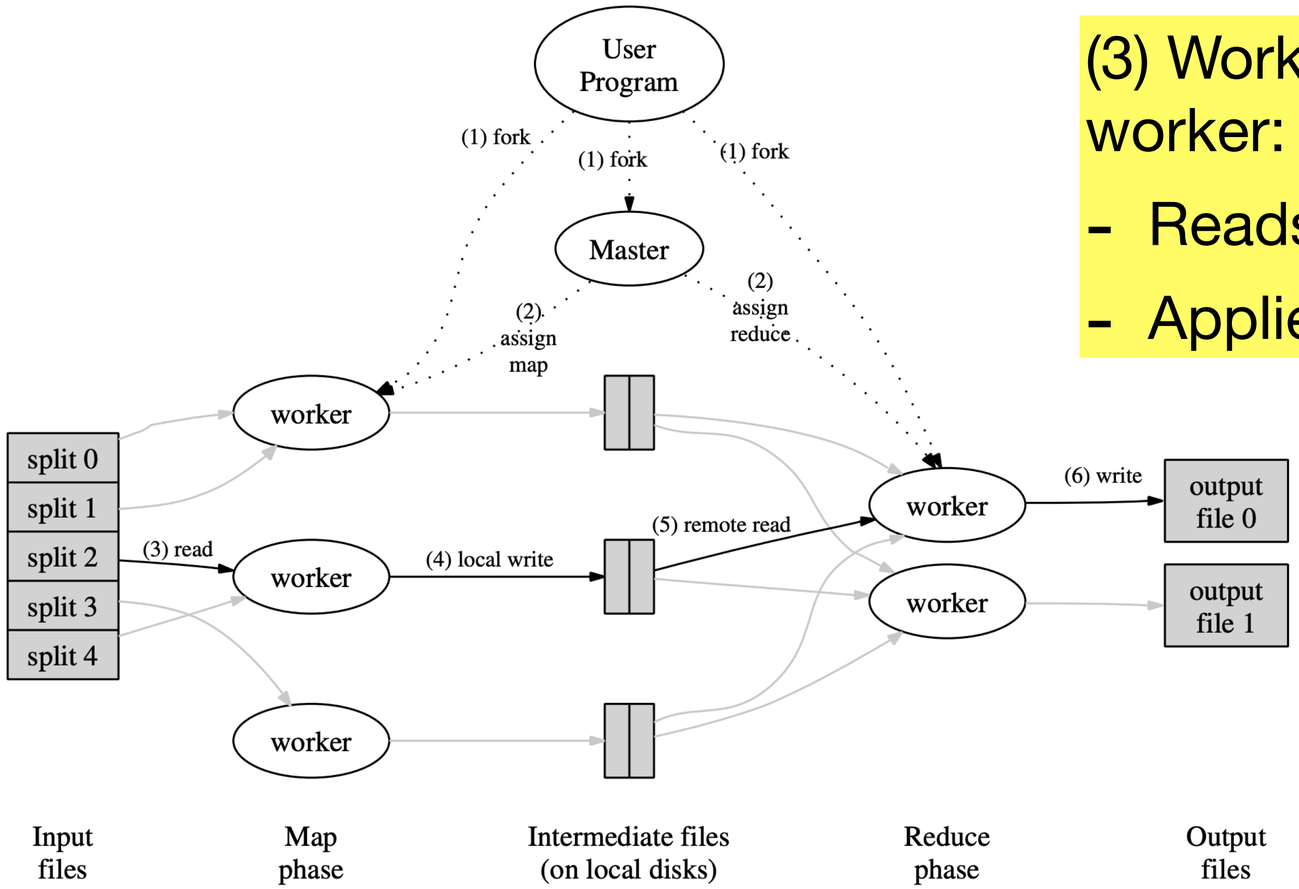


(2) Master assigns map/reduce tasks. It keeps track of:

- Which splits are done
- Which workers are idle
- Where intermediate files are located

MapReduce

- MapReduce: Simplified Data Processing on Large Clusters

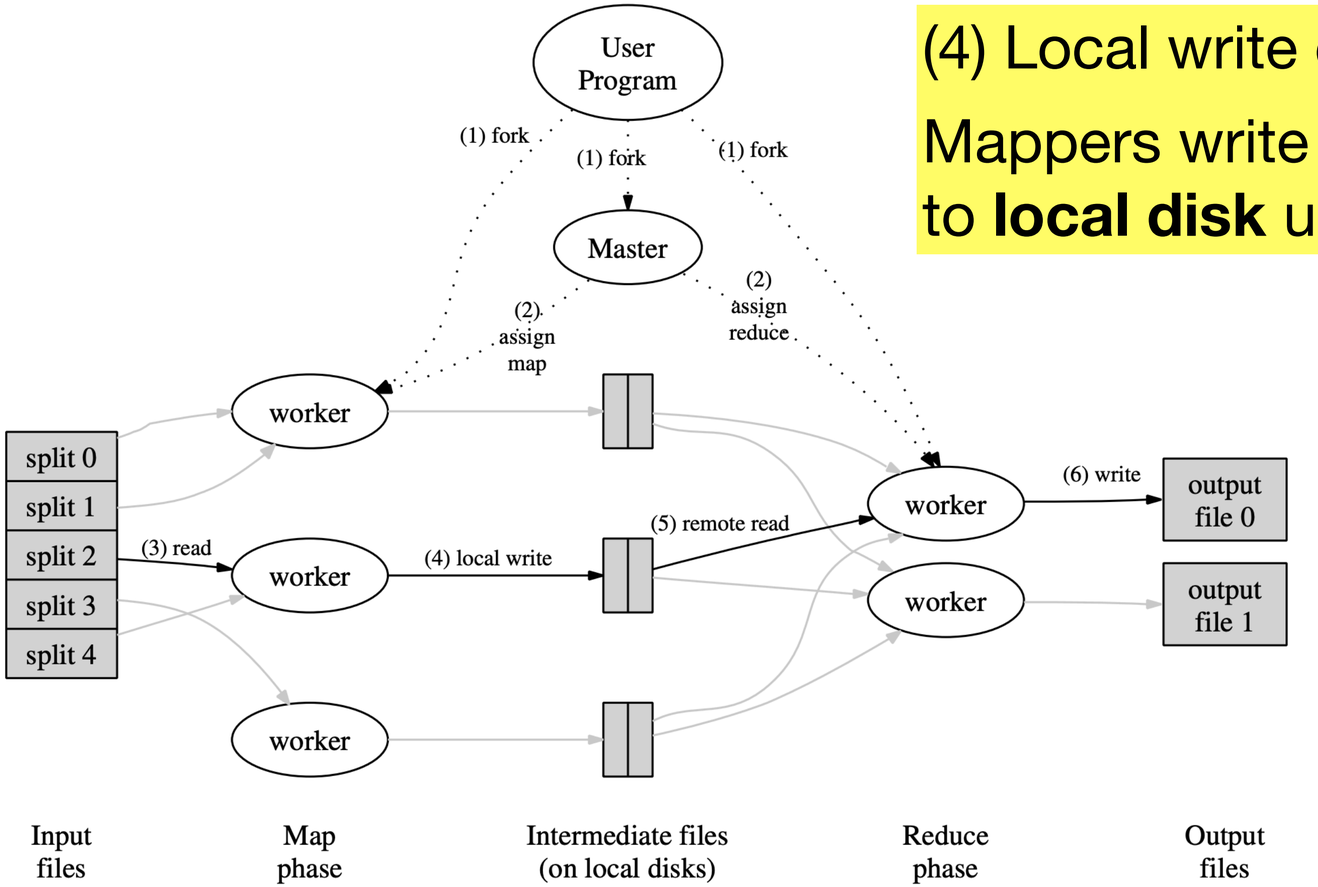


(3) Workers read input splits. Each map worker:

- Reads its assigned input split
- Applies the map() function

MapReduce

- MapReduce: Simplified Data Processing on Large Clusters



(4) Local write of intermediate data

Mappers write intermediate results to **local disk** until reducers fetch it

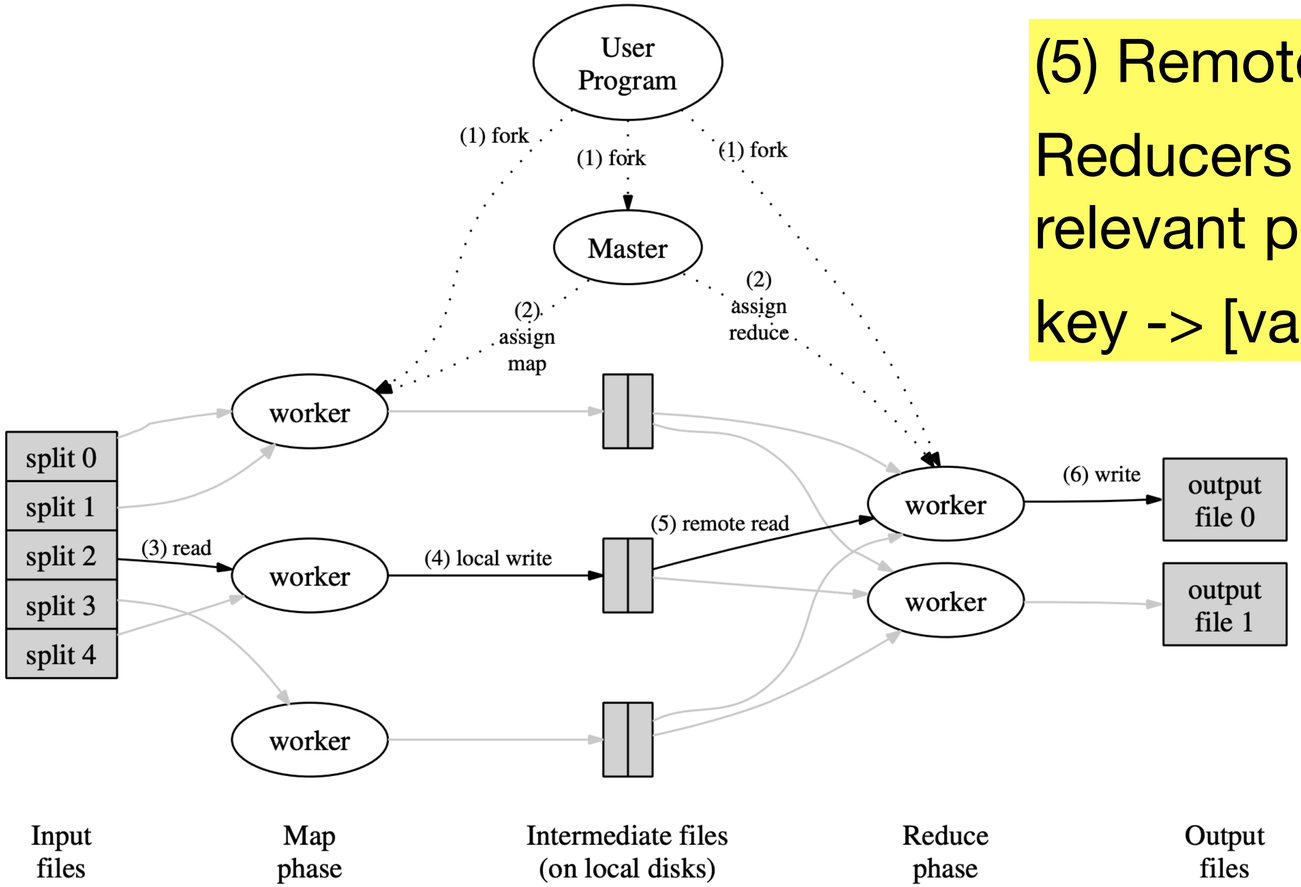
Hidden process: **Shuffle Phase**

Reducers need all values for a key, which requires:

- Sorting
- Partitioning
- Network transfer

MapReduce

- MapReduce: Simplified Data Processing on Large Clusters



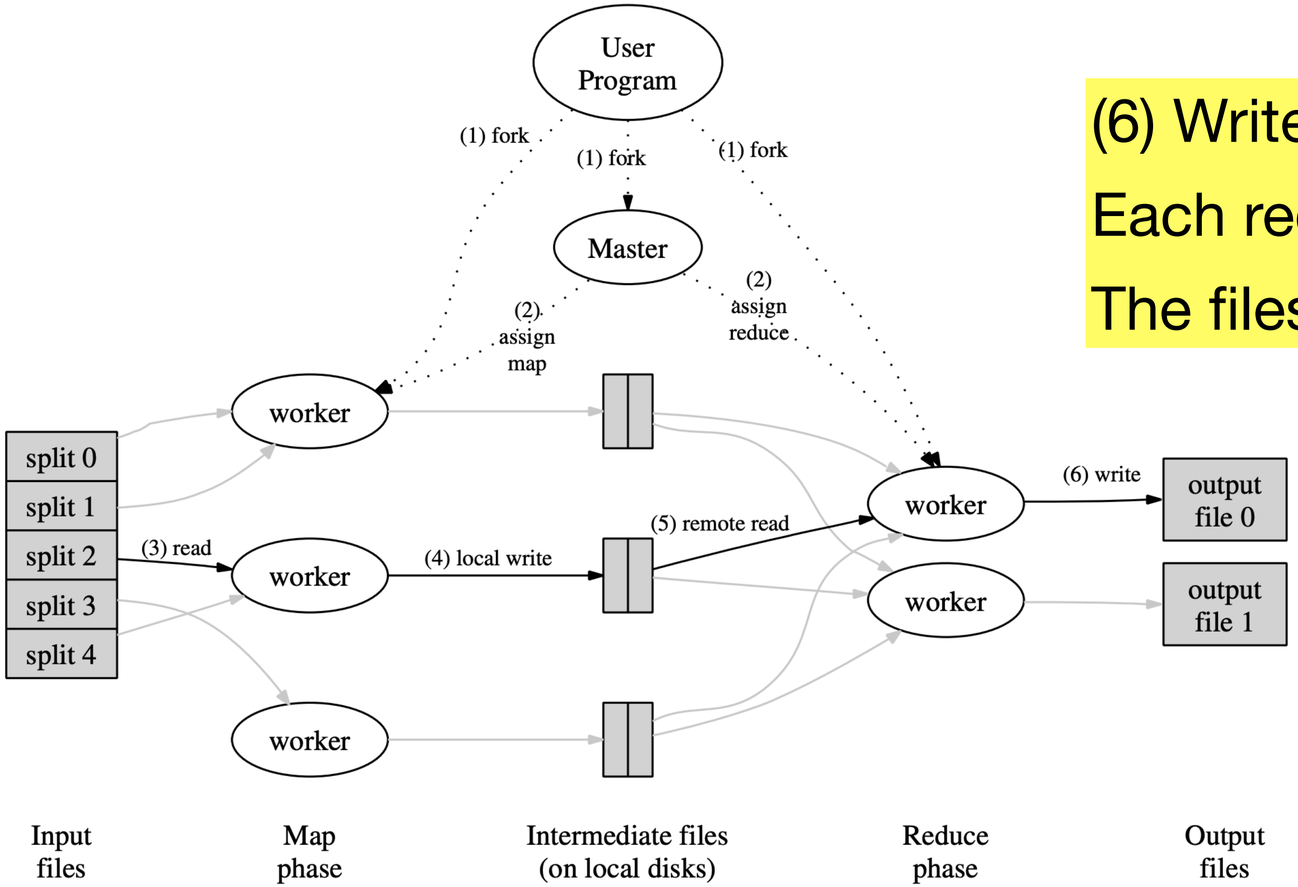
(5) Remote read by reduce workers

Reducers contact each mapper and pull relevant partition of intermediate data (shuffle)

key -> [values]

MapReduce

- MapReduce: Simplified Data Processing on Large Clusters



(6) Write final output

Each reducer produces one output file.

The files are written to distributed storage

Why MapReduce is powerful?

Input → Map → Local Disk → Shuffle → Reduce → Output

- Map tasks are independent
- Reduce tasks are independent
- Failures only require re-running small tasks
- No shared memory required

MPI vs. MapReduce

MPI: low-level message passing for tightly-coupled HPC systems.

MapReduce: high-level data-parallel abstraction for large unreliable clusters.

Aspect	MPI	MapReduce
Level	Low-level	High-level
Communication	Explicit	Implicit
Data partitioning	Manual	Automatic
Fault tolerance	None (default)	Built-in
Latency	Low	High
Target Environment	Supercomputers (Parallel computing)	Commodity clusters (Distributed computing)

K-means in MapReduce

- Goal of K-means
  - Given points $x_i \in R^d$ and K clusters, iterate:
    - Assignment: assign each point to nearest centroid
    - Update: recompute each centroid as mean of its assigned points

    Repeat until centroids stop changing (or max iterations)

In MapReduce, one MapReduce job = one iteration

K-means Mapper and Reducer

Input records:
K1: point ID,
V1: point vector (e.g., (x, y))

K2: centroid ID,
V2: point vector (e.g., (x, y)

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

K-means Mapper and Reducer

Input records:
K1: point ID,
V1: point vector (e.g., (x, y))

K2: centroid ID,
V2: point vector (e.g., (x, y)

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

$$reduce(k_2, list(v_2)) \rightarrow list(v_3)$$

New centroid

In other words:

$$map(pointID, point) \Rightarrow (clusterID, point)$$

$$reduce(clusterID, [p_1, p_2, \dots]) \Rightarrow (clusterID, newCentroid)$$

Step 1: Driver

- Initialize centroids $C^{(0)}$ (random or k-means++)
- For iteration $t = 0,1,2...$
  - Run MapReduce job with centroids $C^{(t)}$ on mappers
  - Reducers output new centroids $C^{(t+1)}$
  - If convergence: $||C^{(t+1)} - C^{(t)}||$ small enough
    - Stop

Step 1: Driver

- Initialize centroids $C^{(0)}$ (random or k-means++)
- For iteration $t = 0,1,2...$
  - Run MapReduce job with centroids $C^{(t)}$ on mappers
  - Reducers output new centroids $C^{(t+1)}$
  - If convergence: $||C^{(t+1)} - C^{(t)}||$ small enough
    - Stop

Mappers need to know new centroids:

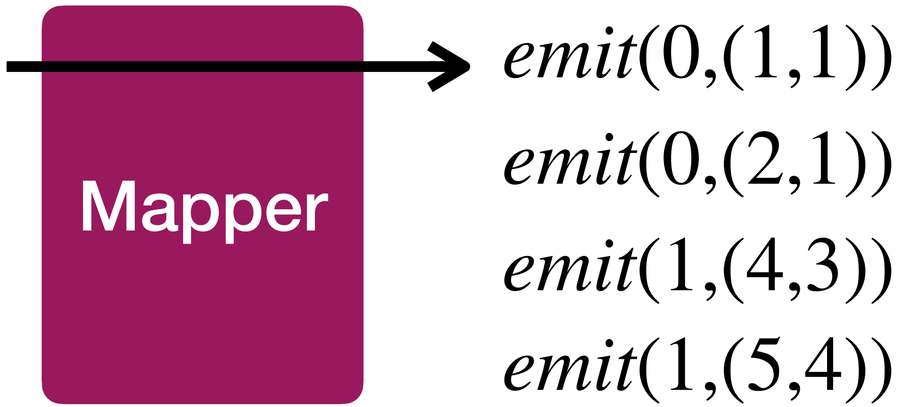
Distributed cache/broadcasts etc

Example K = 2

- Points:
  - P1 = (1, 1)
  - P2 = (2, 1)
  - P3 = (4, 3)
  - P4 = (5, 4)
- Initial centroids:
  - C0 = (1, 1)
  - C1 = (5, 4)

Example K = 2

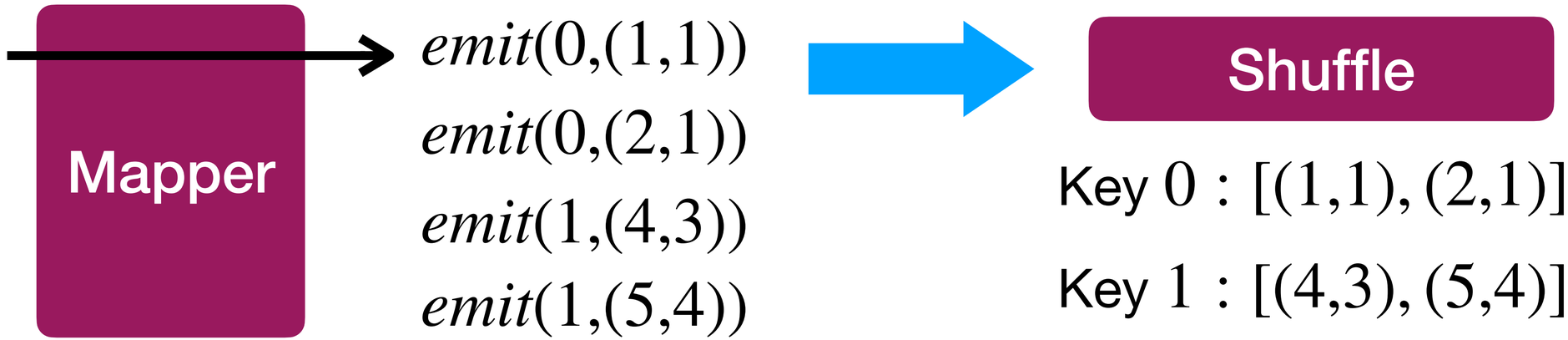
- Points:
  - P1 = (1, 1)
  - P2 = (2, 1)
  - P3 = (4, 3)
  - P4 = (5, 4)
- Initial centroids:
  - C0 = (1, 1)
  - C1 = (5, 4)



$emit(0,(1,1))$
$emit(0,(2,1))$
$emit(1,(4,3))$
$emit(1,(5,4))$

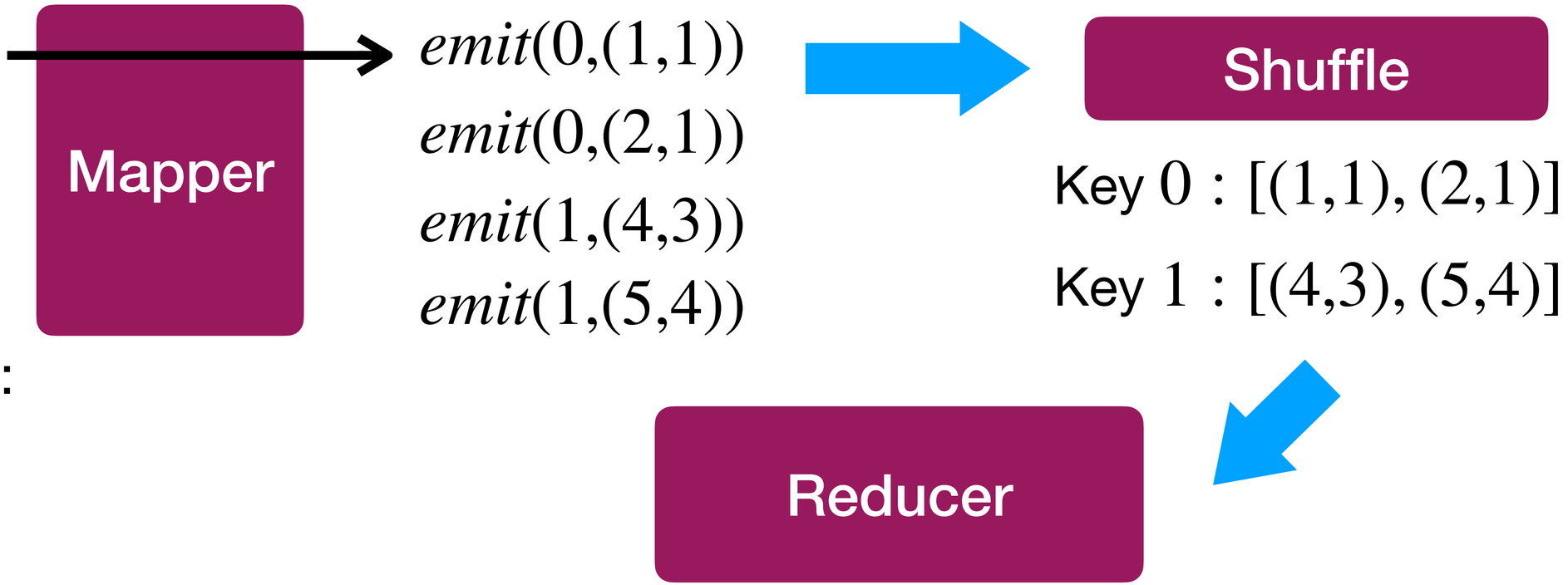
Example K = 2

- Points:
  - P1 = (1, 1)
  - P2 = (2, 1)
  - P3 = (4, 3)
  - P4 = (5, 4)
- Initial centroids:
  - C0 = (1, 1)
  - C1 = (5, 4)



Example K = 2

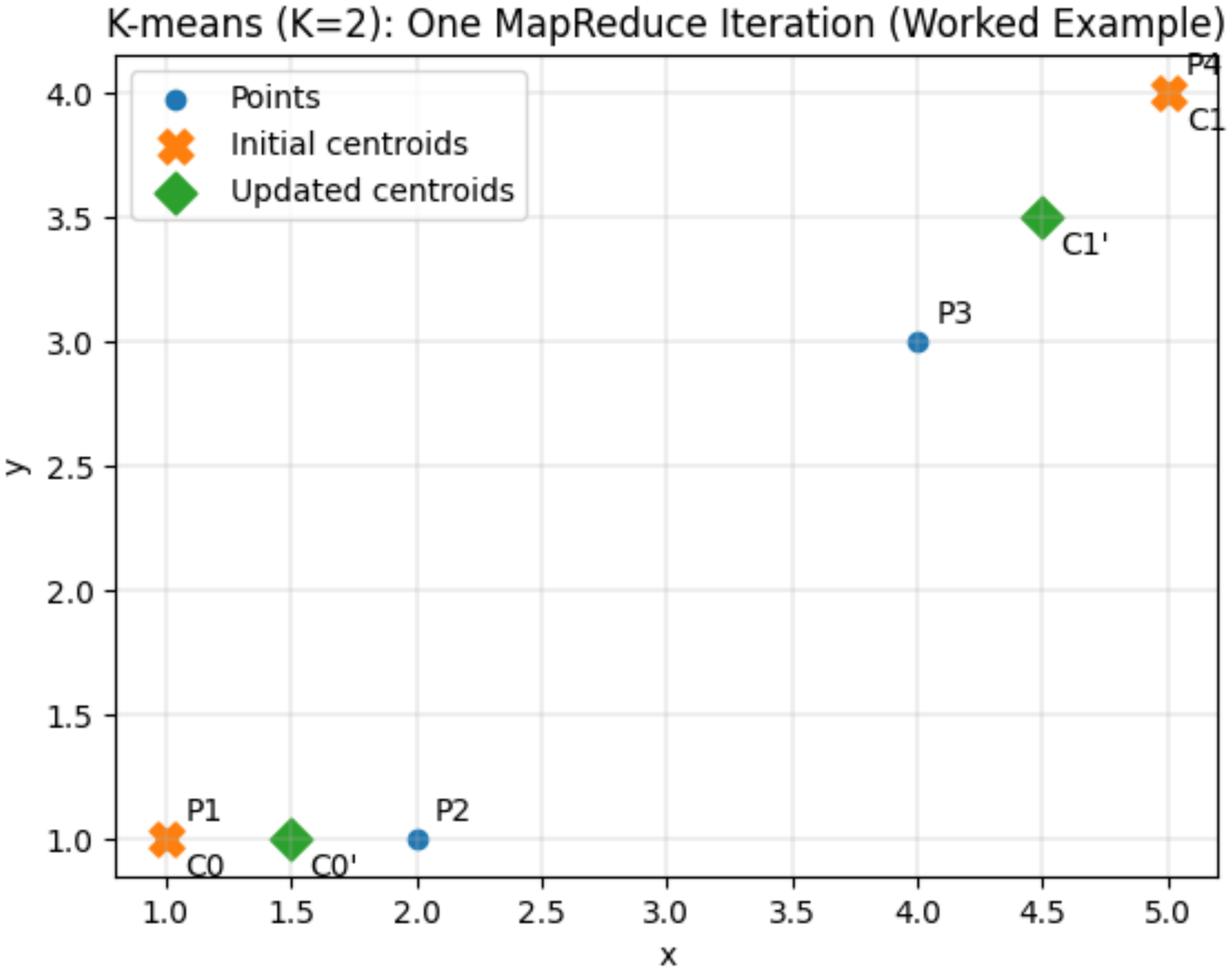
- Points:
  - P1 = (1, 1)
  - P2 = (2, 1)
  - P3 = (4, 3)
  - P4 = (5, 4)
- Initial centroids:
  - C0 = (1, 1)
  - C1 = (5, 4)



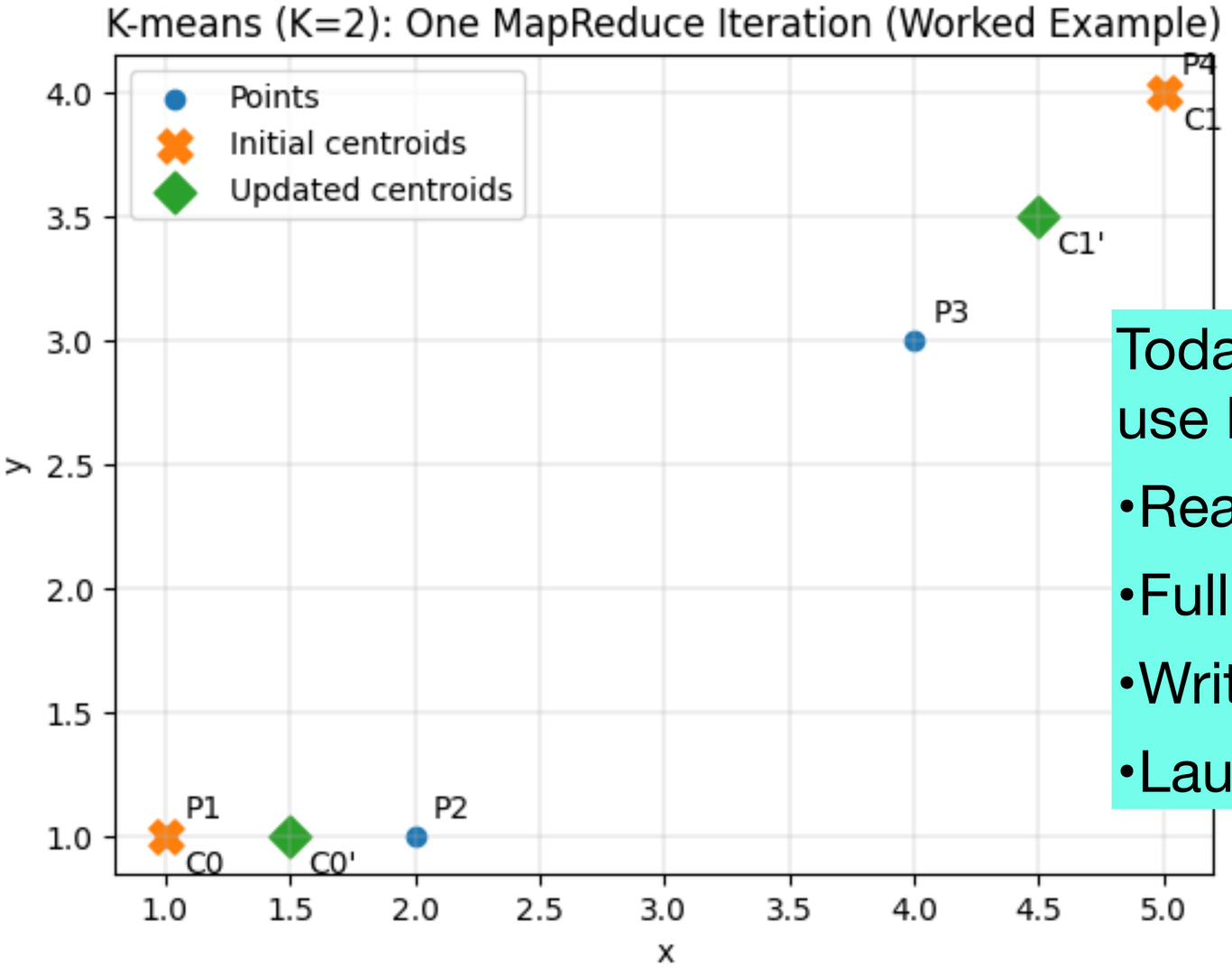
$$\mu_{k(0)} = \left(\frac{3}{2}, \frac{2}{2}\right) = (1.5, 1.0) \longrightarrow emit(0,(1.5,1))$$

$$\mu_{k(1)} = \left(\frac{9}{2}, \frac{7}{2}\right) = (4.5, 3.5) \longrightarrow emit(1,(4.5,3.5))$$

Plot of the example



Plot of the example



Today's real-world applications rarely use MapReduce for K-means:

- Reading entire dataset
- Full shuffle
- Writing intermediate results
- Launching a new job

Worksheet