09-2024

code URJC

Sistemas Distribuidos de Procesamiento de Datos I

Tema 2: Arquitecturas de procesamiento de datos

Tema 2.2 MapReduce

Universidad Rey Juan Carlos

Michel Maes
michel.maes@urjc.es

Atribuciones

Parte del material de este tema se ha realizado a partir del material proporcionado por el profesor
Alberto Sánchez Campos (alberto.sanchez@urjc.es)

MapReduce

code URJC

- Modelos de programación paralela
- MapReduce
- MapReduce en local
- MapReduce distribuido
- Limitaciones de MapReduce

MapReduce

code URJC

- **Modelos de programación paralela**
- MapReduce
- MapReduce en local
- MapReduce distribuido
- Limitaciones de MapReduce

Modelos de programación paralela

- Modelos de programación:
  - **Memoria compartida**: cada procesador tiene acceso a un conjunto de memoria compartida
  - **Memoria distribuida (Paso de mensajes):** Procesos colocados en diferentes máquinas se comunican con otros procesos enviando y recibiendo mensajes
- Permiten el desarrollo de paralelismo de tareas y datos

Modelos de programación paralela

code URJC

- La mayoría de los esfuerzos en paralelización están dentro de las siguientes categorías:
  - Paralelizados usando directivas de compilación tal como OpenMP.
  - Paralelizados usando librerías de paso de mensaje como MPI.
  - Paralelizados usando arquitecturas masivamente paralelas, como CUDA en GPUs

Modelos de programación paralela

code URJC

- **OpenMP – Memoria compartida**
  - OpenMP es un estándar para programación en memoria compartida
  - OpenMP provee un conjunto estándar de directivas, rutinas de librerías run-time
  - No necesita modificar el código secuencial solamente indicarle mediante #pragma la realización de operaciones paralelas

    ```
    #pragma omp parallel for
    ```

  - Orientado a paralelizar códigos bajo el modelo de memoria compartida.

Modelos de programación paralela

code URJC

- OpenMP – Memoria compartida

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def calc_pi():
    num_steps = 100000
    step = 1.0 / num_steps

    the_sum = 0.0
    with openmp("parallel for reduction(+:the_sum) schedule(static)"):
        for j in range(num_steps):
            c = step
            x = ((j-1) - 0.5) * step
            the_sum += 4.0 / (1.0 + x * x)

    pi = step * the_sum
    return pi

print("pi =", calc_pi())
```

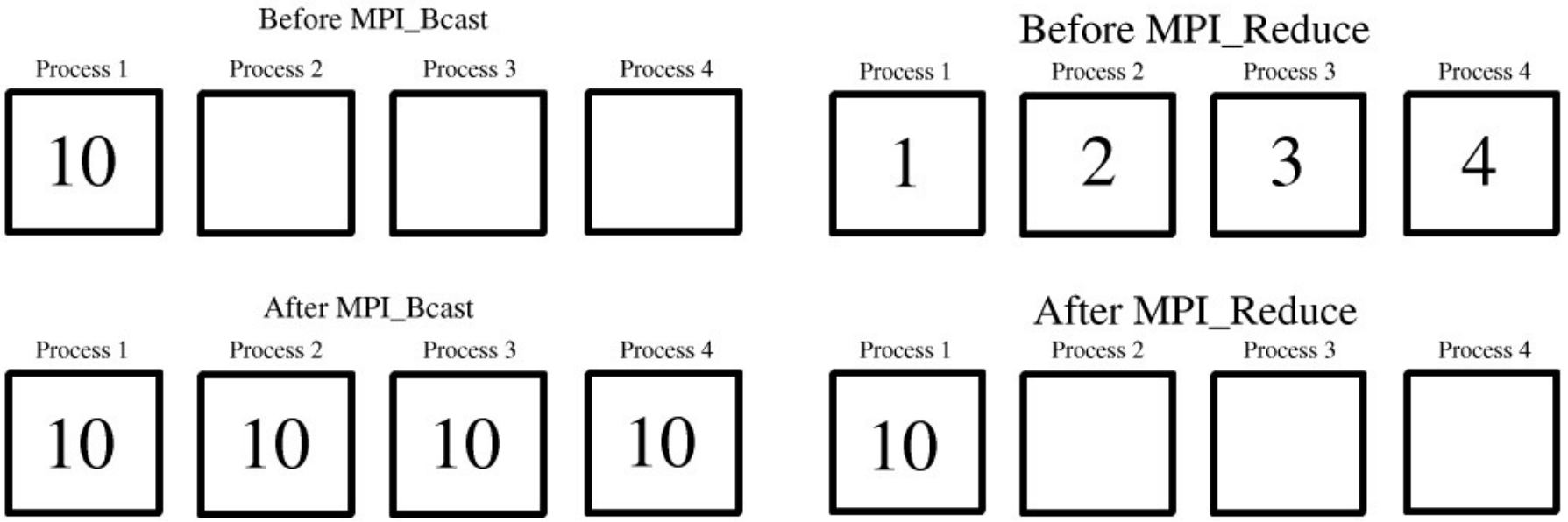
Modelos de programación paralela

code URJC

- **MPI – Paso de Mensajes**
  - MPI es el estándar para paso de mensajes
  - Orientado a memoria distribuida
  - No es un lenguaje, sino una biblioteca que se añade a lenguajes ya existentes (C, C++, Fortran, Python)
  - Muy versátil
    - Diferentes opciones (síncrono, asíncrono, individual, colectivo, …)
  - Colección de funciones que ocultan detalles de bajo nivel tanto HW como SW

Modelos de programación paralela

- **MPI – Paso de Mensajes**
  - En general, requieren de envío (MPI_Send) y recepción (MPI_Receive) de mensajes
  - Tiene operaciones globales para hacer broadcast y recepción múltiple (incluso reduce de información)



Modelos de programación paralela

code URJC

- **MPI vs. OpenMP**
  - Cada una tiene sus ventajas y desventajas:
    - OpenMP
      - Conceptualmente sencillo.
      - Memoria compartida.
    - MPI
      - General, y universal.
      - Útil en sistemas distribuidos. Mayor escalabilidad
      - Control muy fino sobre las comunicaciones.
      - Obliga al programador a operar en un relativo bajo nivel de abstracción.

MapReduce

code URJC

- Modelos de programación paralela
- **MapReduce**
- MapReduce en local
- MapReduce distribuido
- Limitaciones de MapReduce

MapReduce

code URJC

- Orientado al paralelismo de datos: las operaciones son ejecutadas en paralelo sobre colecciones de datos
  - Orientado a memoria distribuida
  - Grado de paralelismo muy alto
  - Fácil de distribuir
  - Muy potente, pero no es una solución fácilmente generalizable.
  - Oculta la dificultad de escribir código paralelo vs MPI
- El propio sistema se encarga del equilibrio de carga, etc.
- Muchos problemas de análisis de datos pueden ser formulados de esta manera

MapReduce

code URJC

- MapReduce surge en un escenario de paralelismo de datos anterior al Big data y hasta cloud
  - Anterior a este movimiento (primeros resultados en 2003)
  - MPI ya provee funciones Reduce
  - Divide y vencerás: no es una idea nueva
- Servicio eficiente y escalable para la gestión de grandes volúmenes de datos
  - Idea: mover el procesamiento a los nodos con los datos en vez de al revés.
  - Puede funcionar en cualquier entorno (en comparación con el procesamiento con CUDA que requiere arquitectura específica GPU).

MapReduce

code URJC

- Objetivo: hacer posible el procesado de grandes volúmenes de datos.
- Se basa en dos funciones básicas: map y reduce
  - map:
    - Los datos de entrada del problema se dividen en bloques (sub-problemas), que se distribuyen entre los nodos. Cada nodo procesa su sub-problema.
    - **map(f, X) = [f(x1), f(x2), f(x3), ...]**: aplica la función f a todos los valores de X para crear una nueva lista de valores
  - reduce:
    - Los resultados de cada sub-problema se combinan en un resultado final.
    - **reduce(f, X) = f(x1, f(x2, f(x3, ...)))**: Acumula de forma recursiva los valores de X, aplicando la función f para crear un nuevo valor

MapReduce

- Ejemplo:
  - Sumar un número a una lista de números
    - Aplicar una función a cada dato de forma independiente.
    - Lleva a cabo una función de los valores individuales de un conjunto de datos para crear una nueva lista de valores

      entrada: [10 15 20 25 30 35 40]
      función: x+1 (mapper)
      salida: [11 16 21 26 31 36 41]
  - Suma una lista de números
    - Se quiere generar un resultado 'único' a partir de un conjunto de datos.
    - Combina valores de un conjunto de datos para crear un nuevo valor

      entrada: [10 15 20 25 30 35 40]
      función: suma (reducer)
      salida: 175

MapReduce

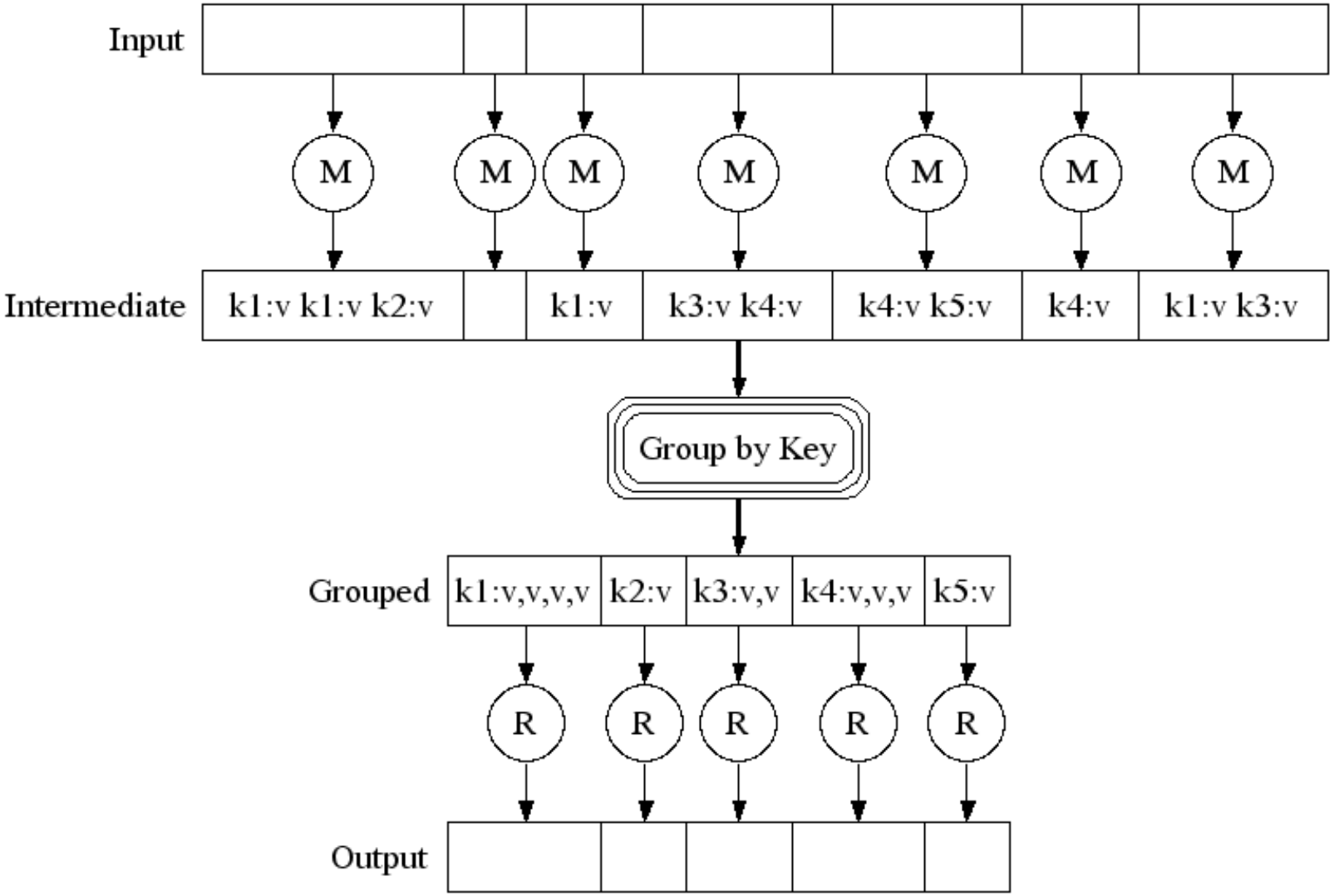
- Los **mapper** son claramente paralelizables. Se ejecutan sobre cada elemento de forma individual.
- En lugar de procesar un conjunto de elementos atómicos digamos que nuestros mappers procesan pares (k1,v1) y generan otros pares (k2, v2).

MapReduce

- Los **reducers** son dificilmente paralelizables
- Digamos que los reducers procesan pares de la forma (k2, [v2, v2, v2...]) y generan pares (k3, v3).

- Así conseguimos varios reducers, uno por cada clave k2, lo cual es paralelizable.
- Como los mappers generan varios valores k podemos organizarlos de la siguiente forma

MapReduce


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

MapReduce

code URJC

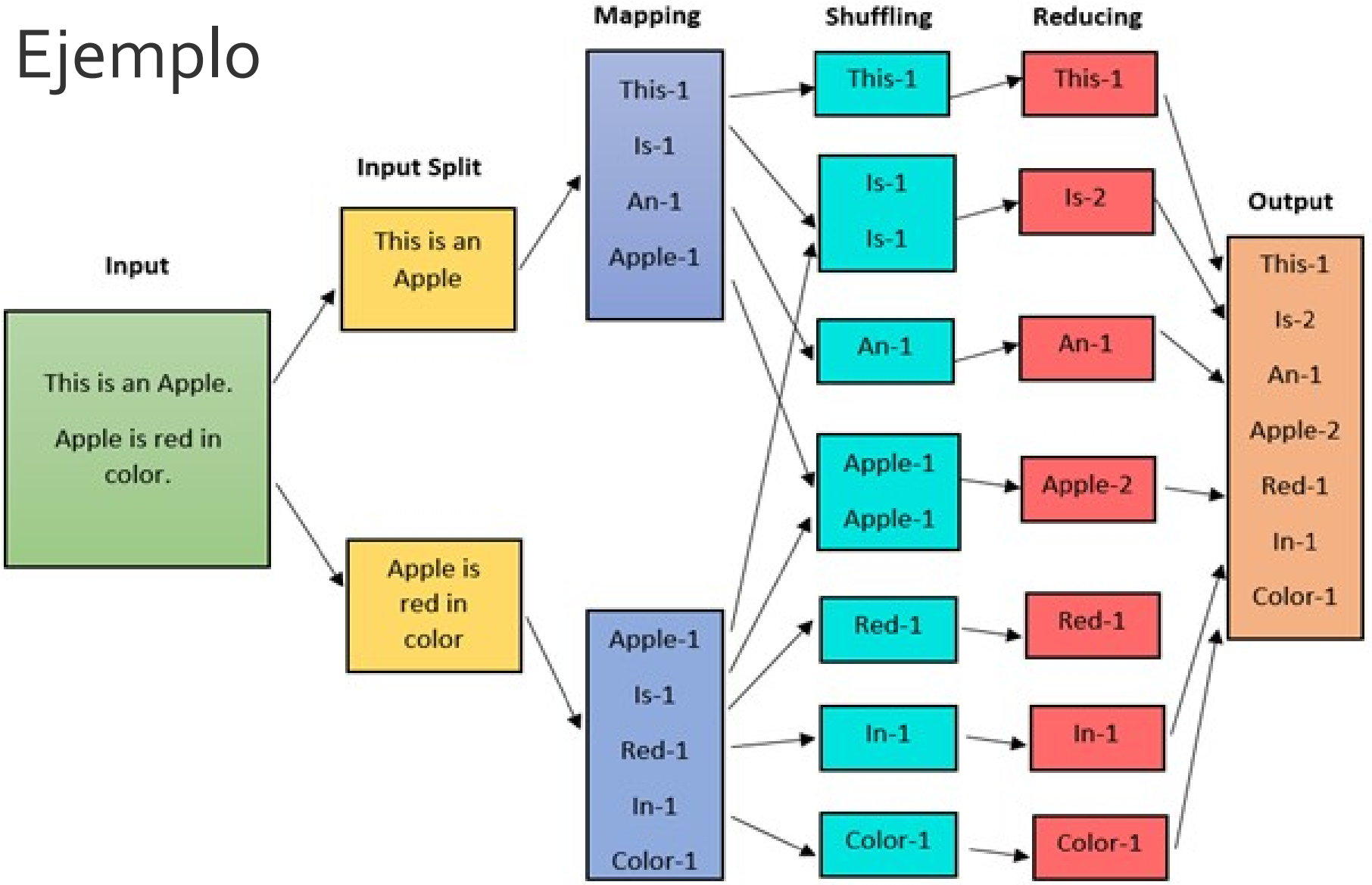
- Puede ser complejo de diseñar.
  - Requiere un cambio en la forma de pensar y nuevos patrones de diseño.
- El resultado final es más claro y conciso que con otros modelos de computación distribuida.
- No hay que preocuparse de:
  - Concurrencia.
  - Recuperación en caso de error.
  - Garantizar la escalabilidad del proceso.

MapReduce

- Los cinco pasos de MapReduce:
  1) Se prepara la entrada del Map(): Partición y diseminación de los datos.
  2) Se ejecuta Map() en cada nodo (mapper). La salida de cada sub-problema se identifica con una clave intermedia.
  3) Se baraja y ordena (shuffle and sort) la salida de cada Map(), usando la clave intermedia.
  4) Ejecuta Reduce() en cada nodo. Cada reducer procesa los datos asociados a una clave intermedia.
  5) Se agrupa la salida de los reducers como resultado final.

MapReduce

- Ejemplo



MapReduce

code URJC

- Modelos de programación paralela
- MapReduce
- **MapReduce en local**
- MapReduce distribuido
- Limitaciones de MapReduce

MapReduce en local

code URJC

- Podemos aprender sobre cómo programar utilizando MapReduce sin necesidad de un clúster distribuido
- Nos ayudará a afianzar los conceptos y modelar los problemas
- Utilizaremos Python para desarrollar los mappers y los reducers

MapReduce en local

- Veremos 3 formas de ejecutar MapReduce en local:
  - Con una librería Python propia sencilla
    - Nos ayudará a entender MapReduce
  - Con scripts de bash/Python
    - Nos ayudará a entender cómo paralelizar MapReduce
  - Con la librería MRJob
    - Solución avanzada que se puede lanzar en local o en un clúster sin realizar modificaciones

MapReduce en local

code URJC

Programación MapReduce (Libreria propia)

- La forma en que normalmente se ejecuta un MapReduce es escribir un programa Java con tres partes .
  - Un método main que configura el trabajo:
    - Establece las clases Mapper y Reducer
    - Establece la forma de particionado
    - Establece otras configuraciones Hadoop
  - A Mapper class
    - Toma entradas <K,V> y devuelve salidas <K,V>
  - A Reducer class
    - Toma entradas <K, Iterator [V]>, y devuelve salidas <K,V>

MapReduce en local

code URJC

Programación MapReduce (Libreria propia)

- Escribiremos una librería Python que implemente el modelo de programación MapReduce fielmente
- Se ejecutará en su totalidad en una sola máquina sinimplicar computación paralela

MapReduce en local

code URJC

Programación MapReduce (Libreria propia)

- Dividido en 4 partes:
  - Se crea un objeto MapReduce que se utiliza para pasar datos entre map y reduce
  - La función mapper tokeniza cada entrada y emite un par clave-valor.
  - La función reduce resume la lista de ocurrencias y emite un recuento en la forma clave-valor.
  - Se carga el archivo de entrada y se ejecuta la consulta MapReduce mostrando el resultado por stdout

MapReduce en local

code URJC

- Librería MapReduce

wordcount_0/MapReduce.py

```
class MapReduce:
    def __init__(self):
        self.intermediate = {}
        self.results = []

    def emit_intermediate(self, key, value):
        self.intermediate.setdefault(key, [])
        self.intermediate[key].append(value)

    def emit(self, value):
        self.results.append(value)

    def execute(self, data, mapper, reducer):
        for line in data.readlines():
            mapper(line)

        for key in self.intermediate:
            reducer(key, self.intermediate[key])

        for result in self.results:
            print(result)
```

MapReduce en local

- Ejemplo wordcount

wordcount_o/wordcount.py

```
import MapReduce
import sys
# 1. Se crea un objeto MapReduce que se utiliza para pasar datos entre map y reduce
mr = MapReduce.MapReduce()
# 2. La función mapper tokeniza cada entrada y emite un par clave-valor.
def mapper(line):
    words = line.split()
    for word in words:
        mr.emit_intermediate(word, 1)
# 3. La función reduce resume la lista de ocurrencias y emite un recuento en la
forma clave-valor.
def reducer(key, list_of_values):
    count = sum(list_of_values)
    mr.emit((key, count))
# 4. Se carga el archivo de entrada y se ejecuta la consulta MapReduce mostrando el
resultado por stdout
if __name__ == '__main__':
    inputdata = open(sys.argv[1])
    mr.execute(inputdata, mapper, reducer)
```

MapReduce en local

code URJC

- Ejemplo wordcount

```
$ python wordcount.py data.txt
```

```
('word', 24)
('count', 11)
('from', 2)
('Wikipedia', 1)
('the', 38)
('free', 1)
('encyclopedia', 1)
('is', 19)
('number', 3)
('of', 25)
...
```

MapReduce en local

- En el ejemplo anterior hemos realizado un programa MapReduce que permite, dado un fichero de datos con múltiples lineas de texto, realizar un recuento de las ocurrencias de cada palabra
- El **mapper** se ocupa de partir las lineas, devolvidendo una tupla *(word, 1)*
- El **reducer** suma las ocurrencias de cada palabra, devolviendo una tupla *(word,total_count)*

```
$ python wordcount.py data.txt
```

MapReduce en local

- Ejemplos para aplicar MapReduce
  - Nucleótidos
  - Índice invertido
  - Amistad
  - Multiplicación de matrices

MapReduce en local

code URJC

- Todos usan la misma librería MapReduce

```
import json

class MapReduce:
      def __init__(self):
            self.intermediate = {}
            self.result = []

      def emit_intermediate(self, key, value):
            self.intermediate.setdefault(key, [])
            self.intermediate[key].append(value)

      def emit(self, value):
            self.result.append(value)

      def execute(self, data, mapper, reducer):
            for line in data:
                  record = json.loads(line)
                  mapper(record)

            for key in self.intermediate:
                  reducer(key, self.intermediate[key])

            jenc = json.JSONEncoder()
            for item in self.result:
                  print(jenc.encode(item))
```

MapReduce en local

code URJC

Ejemplo: Nucleótidos

- Considere un conjunto de pares clave-valor, donde cada clave es un id. de secuencia y cada valor es una cadena de nucleótidos, por ejemplo, GCTTCCGAAATGCTCGAA …. Escribir una consulta MapReduce para quedarse con los primeros 10 caracteres de cada cadena de nucleótidos sin duplicados.
- Mapper:
  - La entrada es una lista de 2 elementos: [ID_secuencia, nucleótidos]
  - ID_secuencia: Identificador único formateado como una cadena
  - Nucleótidos: Secuencia de nucleótidos formateados como una cadena
- Reduce:
  - La salida de la función debe ser los nucleótidos únicos recortados.
- Datos: **dna.json**

MapReduce en local

code URJC

Ejemplo: Índice invertido

- Dado un conjunto de documentos, un índice invertido es un diccionario donde cada palabra se asocia con una lista de los identificadores de documentos en los que aparece dicha palabra.
- Mapper:
  - La entrada es una lista de 2 elementos: [ID_documento, texto]
  - ID_documento : identificador de documento formateado como una cadena
  - texto : el texto del documento formateado como una cadena. Puede tener palabras diferentes o elementos de puntuación por lo que cualquier entrada debe ser considerada como válida (sólo hay que utilizar value.split ())
- Reduce:
  - El resultado debe ser una lista de pares (palabra, lista de ID de documento)
  - Palabra: cadena que representa a cada palabra encontrada
  - lista de ID de documentos.
- Datos: **books.json**

MapReduce en local

code URJC

Ejemplo: Amistad

- La relación de amistad es a menudo simétrica, lo que significa que si yo soy tu amigo, tu eres mi amigo. Implementar un algoritmo de MapReduce para comprobar si esta propiedad se mantiene generando una lista de todas las relaciones no simétricas.
- Mapper:
  - La entrada es una lista 2 elemento : [Persona A, Persona B]
  - Persona A: Nombre de una persona con formato como una cadena
  - Persona B: Nombre de uno de los amigos formateado como una cadena. Esto implica que Persona B es amiga de Persona A, pero no implica la relación inversa.
- Reduce:
  - El resultado debe ser una lista de pares (persona, amigo) y (amigo, persona) para cada amistad asimétrica.
  - Solo existirá una de ellas (persona, amigo) o (amigo, persona) de salida en la entrada. Esto indica asimetría
- Datos: **friends.json**

MapReduce en local

code URJC

Ejemplo: Multiplicación de matrices

- A partir de dos matrices A y B en un formato de matriz dispersa, donde cada registro es de la forma i, j, valor, diseñar un algoritmo de MapReduce para calcular la multiplicación de matrices : A x B
- Mapper:
  - La entrada son las valores de la matriz con formato de tuplas. Cada tupla tendrá el formato [matriz, i, j, valor] , donde la matriz es una cadena y i, j, y el valor son enteros .
  - El primer campo, matriz, es una cadena que identifica a qué matriz se refiere. Este campo tiene dos valores posibles :
  - 'a' indica que el registro es de la matriz A
  - ' b' indica que el registro es de matriz B
- Reduce:
  - La salida de la función serán elementos de la matriz formateadas como tuplas. Cada tupla tendrá el formato (i, j , valor) donde cada elemento es un número entero.
- Datos: **matrix.json**

MapReduce en local

code URJC

MapReduce con bash

- Procesamiento en batch clásico
- Definiremos un mapper y un reducer en Python
  - Serán muy parecidos a los que utilizaremos en un futuro para realizar procesamiento distribuido
- Utilizaremos comandos de bash para conectar todo el proceso

MapReduce en local

code URJC

MapReduce con bash

- Python puede controlar fácilmente los siguientes requisitos
  - Usar el módulo sys para leer desde STDIN y
  - Usar print para imprimir a STDOUT.
  - La tarea restante consiste simplemente en dar formato a los datos con un carácter de tabulación (\t) entre la clave y el valor.
  - Map y reduce podrán ser archivos de texto separados (p. ej. mapper.py y reducer.py)

MapReduce en local

code URJC

- Ejemplo wordcount (Mapper)

wordcount_1/mapper.py

```
#!/usr/bin/env python
import sys
# Read each line from stdin
for line in sys.stdin:
   # Get the words in each line
   words = line.split()
# Generate the count for each word
for word in words:
   # Write the key-value pair to stdout to be
   # processed by the reducer.
   # The key is anything before the first tab
   # character and the value is anything after the
   # first tab character.
   print("{0}\t{1}".format(word, 1))
```

MapReduce en local

- Ejemplo wordcount (Reducer)

wordcount_1/reducer.py

```
#!/usr/bin/env python
import sys
curr_word = None
curr_count = 0
# Process each key-value pair from the mapper
for line in sys.stdin:
      # Get the key and value from the current line
      word, count = line.split('\t')
      # Convert the count to an int
      count = int(count)
      # If the current word is the same as the previous word,
      # increment its count, otherwise print the words count
      # to stdout
      if word == curr_word:
            curr_count += count
      else:
            # Write word and its number of occurrences as a
            # key-value pair to stdout
            if curr_word:
                  print('{0}\t{1}'.format(curr_word, curr_count))
            curr_word = word
            curr_count = count

# Output the count for the last word
if curr_word == word:
      print('{0}\t{1}'.format(curr_word, curr_count))
```

MapReduce en local

MapReduce con bash

- Una vez definidos el mapper y el reducer podemos concatenarlos con pipes

```
$ cat data.txt | python mapper.py | sort -t 1 | python reducer.py
```

- Los mismos mapper y reducer pueden ser utilizados en un clúster Hadoop
- Volveremos a ver esté ejemplo más adelante

MapReduce en local

MapReduce con bash

- Se puede mejorar el código anterior
- Para ello utilizaremos iteradores y generadores en Python

MapReduce en local

- Ejemplo wordcount (Mapper mejorado)

wordcount_2/mapper.py

```python
#!/usr/bin/env python
"""A more advanced Mapper, using Python iterators and generators."""
import sys

def read_input(file):
    for line in file:
        # split the line into words
        yield line.split()

def main(separator='\t'):
    # input comes from STDIN (standard input)
    data = read_input(sys.stdin)
    for words in data:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        # tab-delimited; the trivial word count is 1
        for word in words:
            print('%s%s%d' % (word, separator, 1))

if __name__ == "__main__":
    main()
```

MapReduce en local

code URJC

- Ejemplo wordcount (Reducer mejorado)

wordcount_2/reducer.py

```
#!/usr/bin/env python
"""A more advanced Reducer, using Python iterators and generators."""
from itertools import groupby
from operator import itemgetter
import sys

def read_mapper_output(file, separator='\t'):
      for line in file:
            yield line.rstrip().split(separator, 1)

def main(separator='\t'):
      # input comes from STDIN (standard input)
      data = read_mapper_output(sys.stdin, separator=separator)
      # groupby groups multiple word-count pairs by word,
      and creates an iterator that returns consecutive keys and their group:
      # current_word - string containing a word (the key)
      # group - iterator yielding all ["<current_word>", "<count>"] items
      for current_word, group in groupby(data, itemgetter(0)):
            try:
                  total_count = sum(int(count) for current_word, count in group)
                  print("%s%s%d" % (current_word, separator, total_count))
            except ValueError:
                  #count was not a number
                  pass

if __name__ == "__main__":
      main()
```

MapReduce en local

code URJC

MapReduce con bash

- Ejercicio nucleotidos
  - Modela el problema de los nucleotidos para resolverlo con MapReduce
  - Tip: Necesitarás utilizar la librería json de Python para procesar las lineas

```
$ cat dna.json | python mapper.py | sort -t 1 | python reducer.py
```

  - Solución: **dna_1/**

MapReduce en local

code URJC

MapReduce con MRJob

- Librería para ejecutar trabajos MapReduce en Python
- Facilita el envío de trabajos escritos en Python a Hadoop y la ejecución de cada uno de sus pasos individuales
- Misma ejecución
  - En local
  - En cluster Hadoop
  - En AWS EMR
  - En Google Cloud DataProc
- Facilita testeo antes de llevarlo a Hadoop

https://pypi.org/project/mrjob/

MapReduce en local

MapReduce con MRJob

- Instalación

```
$ pip install mrjob
```

- Configurable para cloud en /etc/mrjob.conf o ~/.mrjob.conf
- Permite ejecutar también código de Spark

MapReduce en local

code URJC

MapReduce con MRJob

- Un trabajo se define por una clase que hereda de MRJob.
- Esta clase contiene los métodos que definen un paso de un trabajo MapReduce:
  - **mapper()**. Toma una clave y un valor como argumentos y crea (yield) tantos pares de clave valor como necesite
  - **combiner()**. Para optimizaciones en envío a reducer.
  - **reducer()**. Toma una clave y un iterador de valores y produce pares de clave-valor.
- Todas las funciones son opcionales, aunque debe existir al menos una implementada.

MapReduce en local

code URJC

MapReduce con MRJob – Ejemplo 1

wordcount_3/MRWordCount.py

```
from mrjob.job import MRJob

class MRWordCount(MRJob):
    def mapper(self, _, line):
        for word in line.split():
        yield(word, 1)

    def reducer(self, word, counts):
        yield(word, sum(counts))

if __name__ == '__main__':
    MRWordCount.run()
```

```
$ python MRWordCount.py -r inline data.txt
```

MapReduce en local

MapReduce con MRJob – Ejemplo 2

wordcount_3/MRWordCount.py

```
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

```
$ python MRWordFrequencyCount.py -r inline data.txt
```

MapReduce en local

code URJC

MapReduce con MRJob

- Opciones de ejecución

```
$ python <program.py> [arguments…] -r <runner>
```

  - -r inline. Modo especial para debug en un único nodo (ejecución por defecto)
  - -r local. Simulación de Hadoop con varios procesos en local
  - -r hadoop. En un cluster Hadoop. Los archivos de entrada pueden provenir de HDFS
  - -r emr. En un entorno AWS con EMR. Los archivos de entrada y de salida pueden estar en S3
- Más configuración en: https://pythonhosted.org/mrjob/guides/configs-all-runners.html
  - --file ficheros: Lista de ficheros a subir al mismo directorio donde el trabajo ejecuta
  - --no-output (no muestra la salida por pantalla, es decir no envía los datos al cliente, sino que los deja en el output). Importante si son muchos datos

MapReduce en local

code URJC

MapReduce con MRJob

- MRJob permite definir varios pasos:
  - La función *steps(self)* permite definir un conjunto de pasos MRStep
  - Por cada MRStep hay que definir las operaciones que realiza
    - mapper
    - combiner
    - reducer

MapReduce en local

code URJC

wordcount_3/MRMostUsedWord.py

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")

class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)
        ]

    def mapper_get_words(self, _, line):
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        yield None, (sum(counts), word)

    def reducer_find_max_word(self, _, word_count_pairs):
        yield max(word_count_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()
```

```
$ python MRWordFrequencyCount.py -r inline data.txt
```

MapReduce

code URJC

- Modelos de programación paralela
- MapReduce
- MapReduce en local
- **MapReduce distribuido**
- Limitaciones de MapReduce

MapReduce distribuido

code URJC

Almacenamiento de resultados

- Se utiliza HDFS (Hadoop Distributed File System) para almacenar los resultados y obtener los datos de entrada.
  - Pensado para ejecución de aplicaciones MapReduce.
  - Escalable y tolerante a fallos.
  - Muy orientado a tareas MapReduce:
    - Localización de los datos en nodos donde se ejecutan los Map asociados.

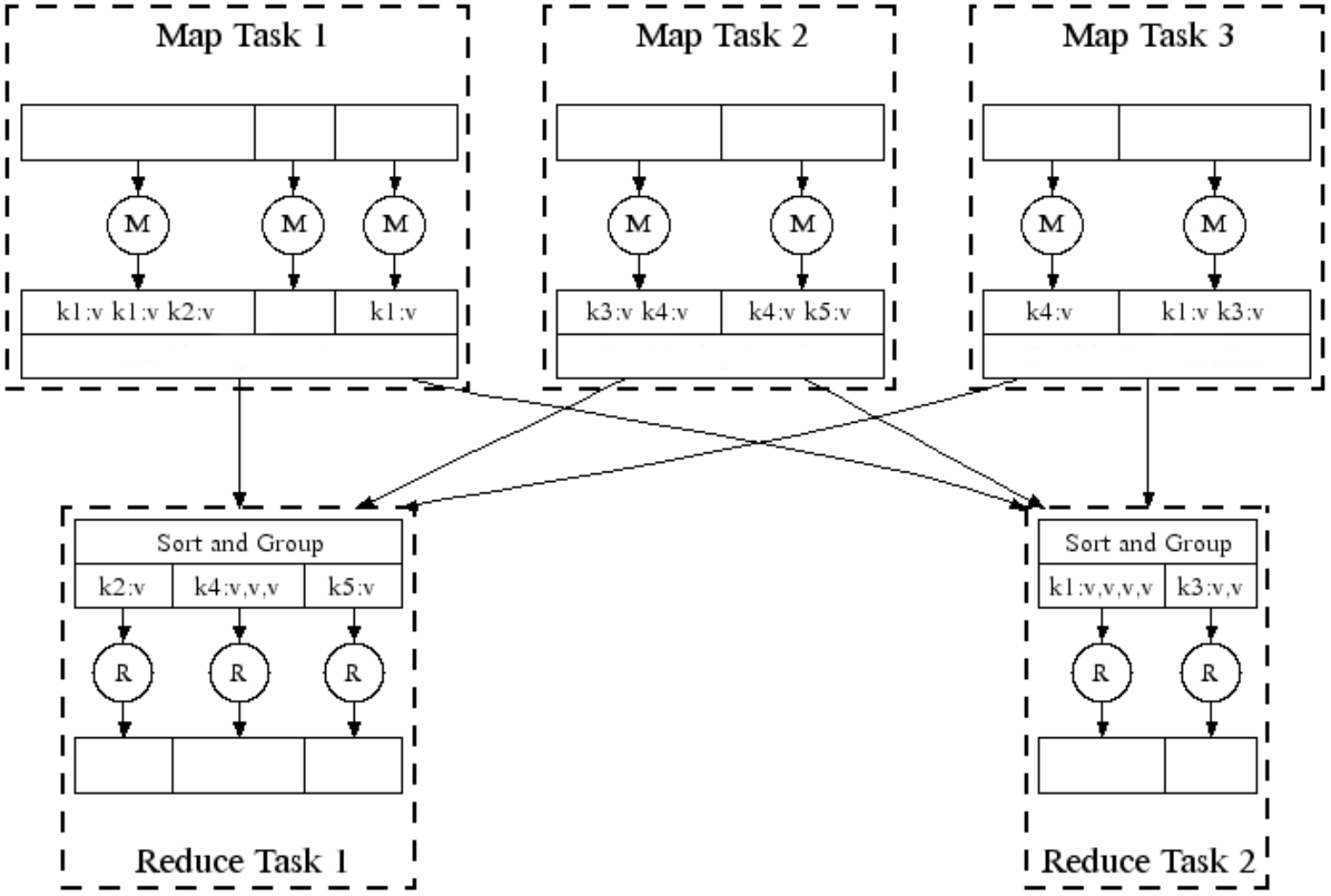
MapReduce distribuido

code URJC

- No hay dependencia entre dos cualesquiera de la misma tarea.
  - Para hacer su trabajo un mapper no necesita saber nada acerca de otro mapper, y del mismo modo pasa en los reducer.
  - Las diferentes etapas pueden distribuirse fácilmente a un número arbitrario de máquinas.
  - No hay dependencias inherentes entre las tareas que les exigen estar en la misma máquina.

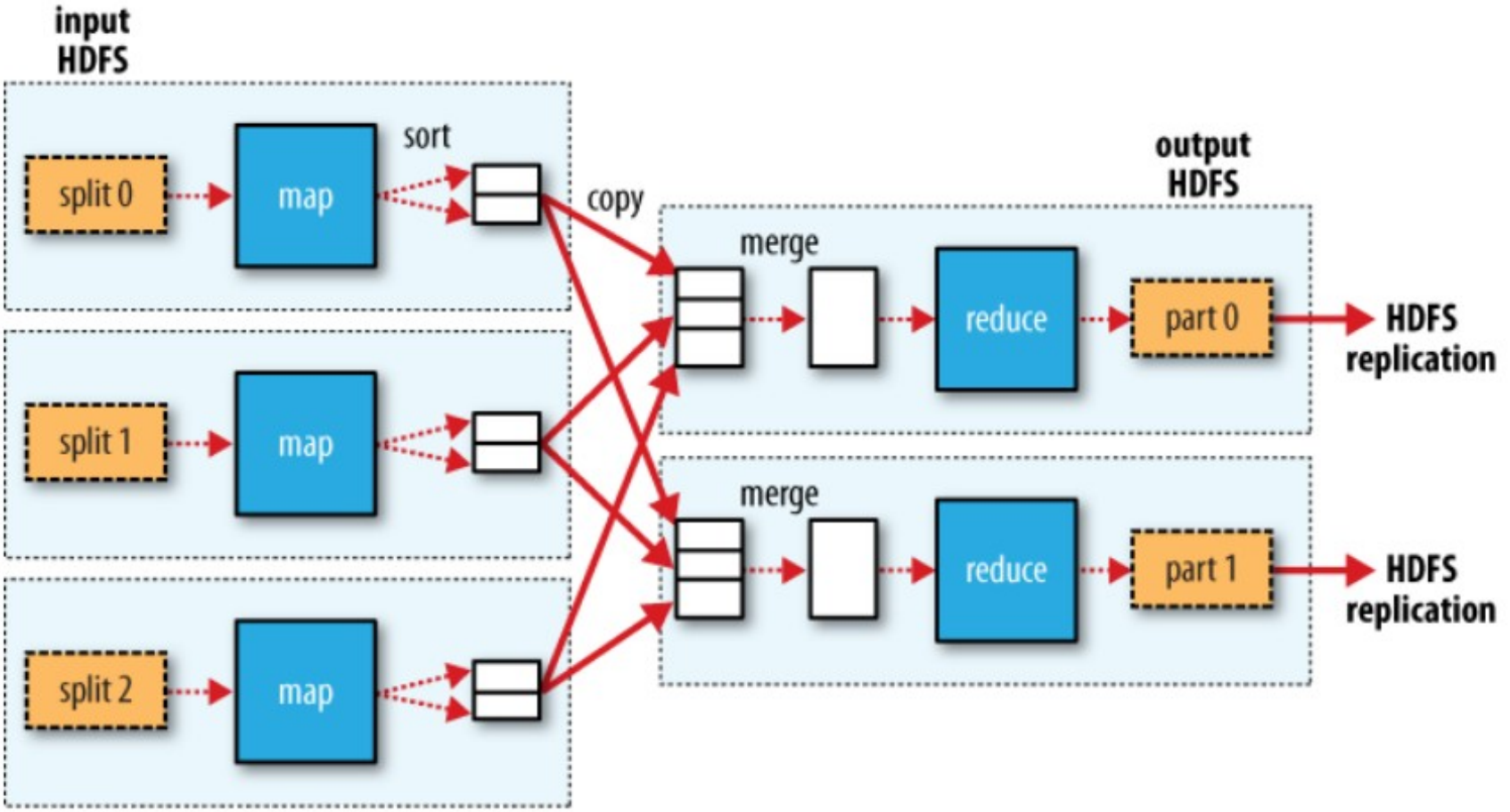
MapReduce distribuido

- Ejemplo de ejecución paralela



MapReduce distribuido

- Ejemplo de ejecución paralela



MapReduce distribuido

code URJC

- La información tiene que moverse a través de la red entre los nodos. Leer los datos a través de la red es lento.
  - En la medida de lo posible ejecutar las funciones Map y Reduce en la misma máquina donde se encuentran los datos o al menos lo más cerca posible.
- En cualquier caso, se puede producir un envío masivo de datos de mappers a reducers
  - Puede generar un problema en el rendimiento.

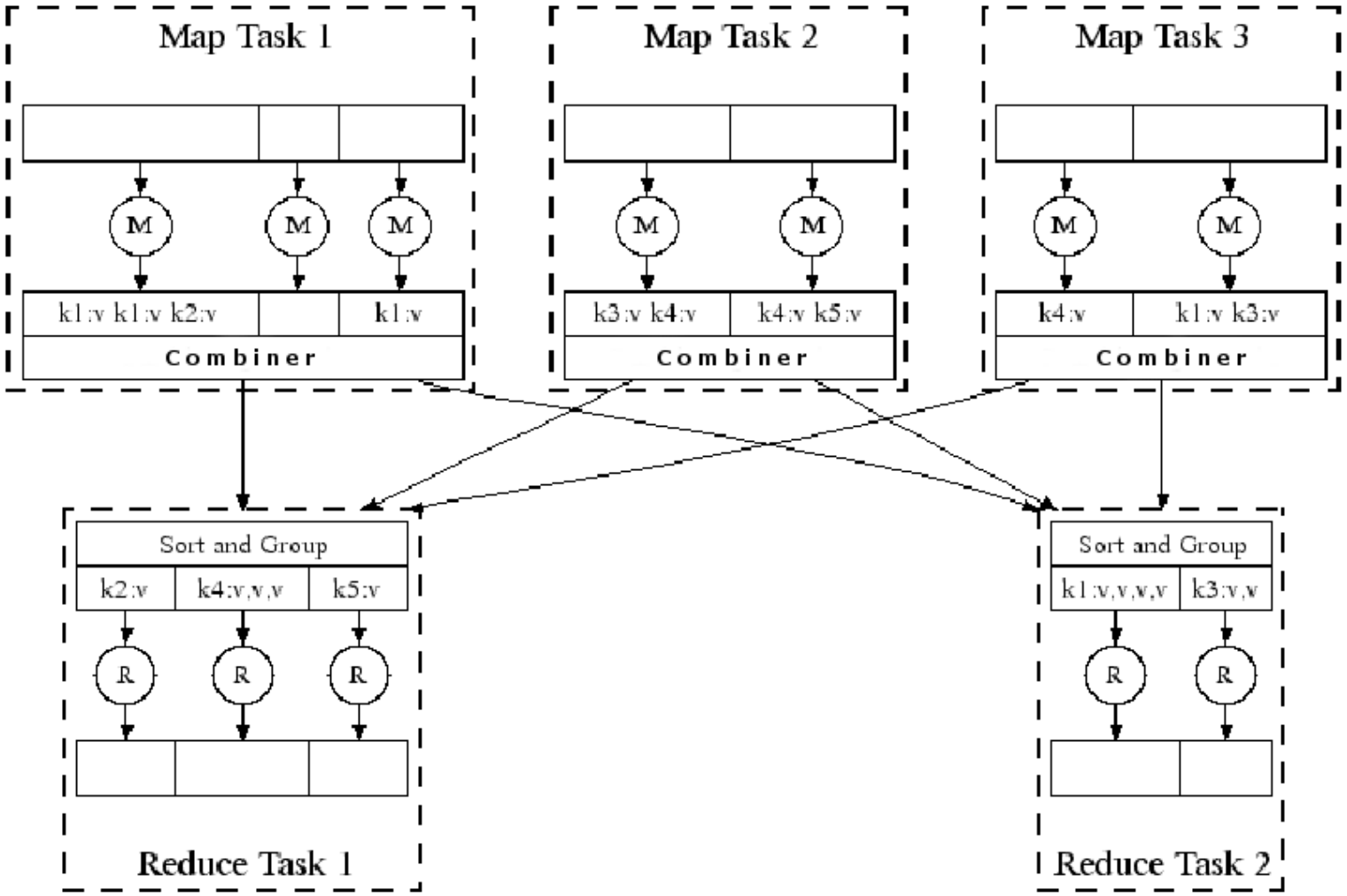
MapReduce distribuido

code URJC

- La salida del Mapper genera una gran cantidad de datos intermedios
  - Estos se tienen que transmitir por la red hacia los Reducer.
  - Si la cantidad de datos es excesivamente grande aquí se puede producir un cuello de botella.
  - Se puede implementar un Combiner, que se ejecuta a la salida de la fase Map de forma local a este antes de enviar los datos a través de la red.
- Añadir un combiner:
  - Cuidado que la operación que se realiza en el Combiner sea asociativa y conmutativa
  - Ahora la entrada del reducer no es la salida del mapper si no la del combiner

MapReduce distribuido

- Ejemplo de ejecución paralela (combiner)



MapReduce distribuido

- Los ejemplos que hemos visto en este tema pueden ser utilizamos de manera distribuida en un clúster de Hadoop
- En los siguientes temas veremos como desplegar un clúster
  - Simulando un clúster en local (Docker)
  - Utilizan un clúster real (Cloudera+Openstack)

MapReduce

code URJC

- Modelos de programación paralela
- MapReduce
- MapReduce en local
- MapReduce distribuido
- **Limitaciones de MapReduce**

Limitaciones de MapReduce

code URJC

- MapReduce es una tecnología base para muchas iniciativas de Big Data.
- Sin embargo, presenta varias limitaciones importantes, que han motivado el desarrollo de alternativas más nuevas y poderosas.
  - Modelo de programación restrictiva
  - Gran dependencia de las operaciones locales de lectura/escritura en disco
  - Desconocimiento de la estructura interna de datos