# CS 61C: Great Ideas in Computer Architecture (Machine Structures)
# *MapReduce, Spark, and HDFS*

Instructors:

Nicholas Weaver & Vladimir Stojanovic

http://inst.eecs.berkeley.edu/~cs61c/

# New-School Machine Structures (It's a bit more complicated!)

*Software* *Hardware*

- Parallel Requests
  - Assigned to computer
  - e.g., Search "cats"
- Parallel Threads
  - Assigned to core
  - e.g., Lookup, Ads
- Parallel Instructions
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions
- Parallel Data
  - >1 data item @ one time
  - e.g., Deep Learning for image classification
- Hardware descriptions
  - All gates @ one time
- Programming Languages

*Harness Parallelism & Achieve High Performance*



# Data-Level Parallelism (DLP)

- SIMD
  - Supports data-level parallelism in a single machine
  - Additional instructions & hardware

  e.g. Matrix multiplication in memory

- DLP on WSC
  - Supports data-level parallelism across multiple machines
  - MapReduce & scalable file systems

  e.g. Training CNNs with images across multiple disks

# What is MapReduce?

- Simple data-parallel ***programming model*** and ***implementation*** for processing large dataset
- Users specify the computation in terms of
  - a ***map*** function, and
  - a ***reduce*** function
- Underlying runtime system
  - Automatically ***parallelize*** the computation across large scale clusters of machines.
  - ***Handles*** machine ***failure***
  - ***Schedule*** inter-machine communication to make efficient use of the networks

Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *6th USENIX Symposium on Operating Systems Design and Implementation, 2004*. (optional reading, linked on course homepage – a digestible CS paper at the 61C level)

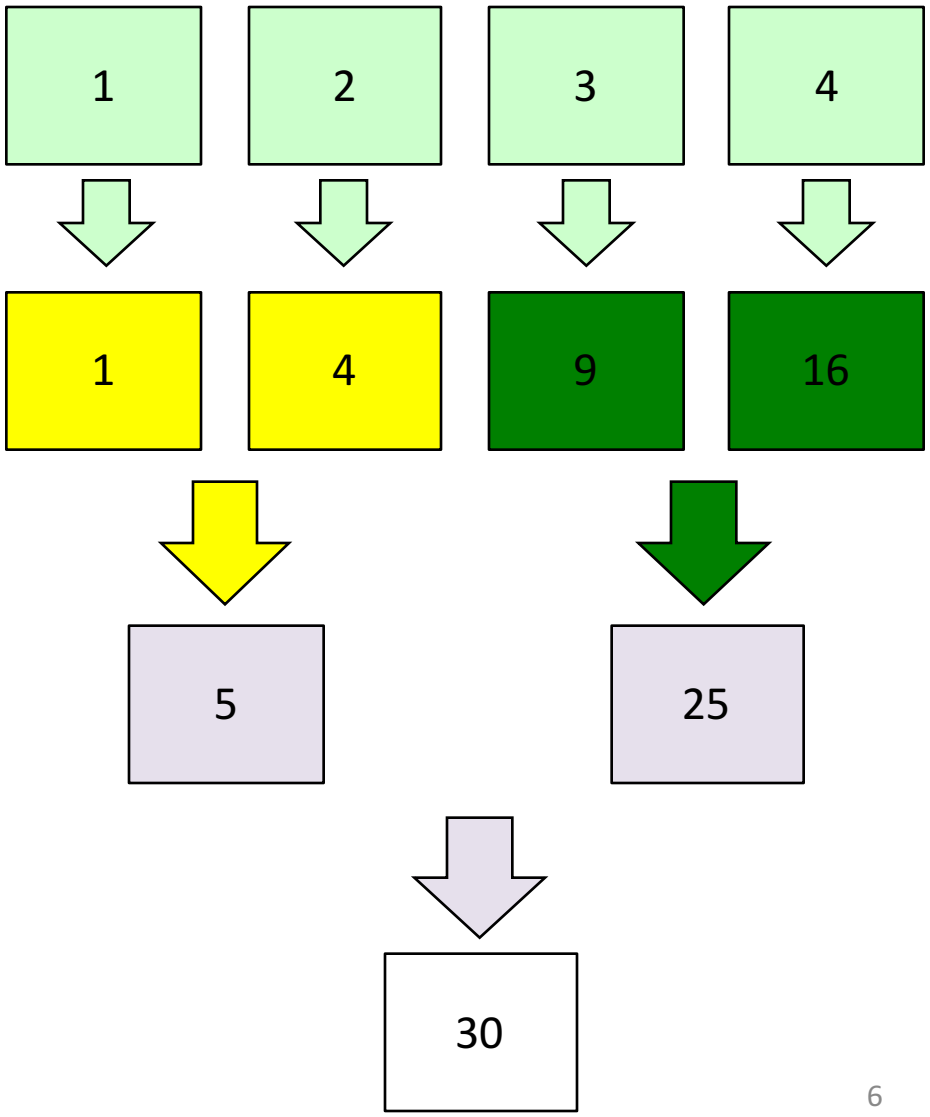
# What is MapReduce used for?

- At Google:
  - Index construction for Google Search
  - Article clustering for Google News
  - Statistical machine translation
  - For computing multi-layers street maps
- At Yahoo!:
  - "Web map" powering Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

# Inspiration: Map & Reduce Functions, ex: Python

Calculate : $\sum_{n=1}^{4} n^2$

```
A = [1, 2, 3, 4]
def square(x):
    return x * x
def sum(x, y):
    return x + y
reduce(sum,
    map(square, A))
```



# MapReduce Programming Model

- ***Map***: `(in_key, in_value) → list(interm_key, interm_val)`

```
map(in_key, in_val):
   // DO WORK HERE
   emit(interm_key,interm_val)
```

  - Slice data into "shards" or "splits" and distribute to workers
  - Compute set of intermediate key/value pairs

- ***Reduce***: `(interm_key, list(interm_value)) → list(out_value)`

```
reduce(interm_key, list(interm_val)):
   // DO WORK HERE
   emit(out_key, out_val)
```

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

# MapReduce Word Count Example

***Task of counting the number of occurrences of each word in a large collection of documents.***

User-written Map function reads the document data and parses out the words. For each word, it writes the (key, value) pair of (word, 1). That is, the word is treated as the intermediate key and the associated value of 1 means that we saw the word once.

***Map*** phase: (doc name, doc contents) → list(word, count)

```
// “I do I learn” → [(“I”,1),(“do”,1),(“I”,1),(“learn”,1)]
```

```
map(key, value):
  for each word w in value:
    emit(w, 1)
```

# MapReduce Word Count Example

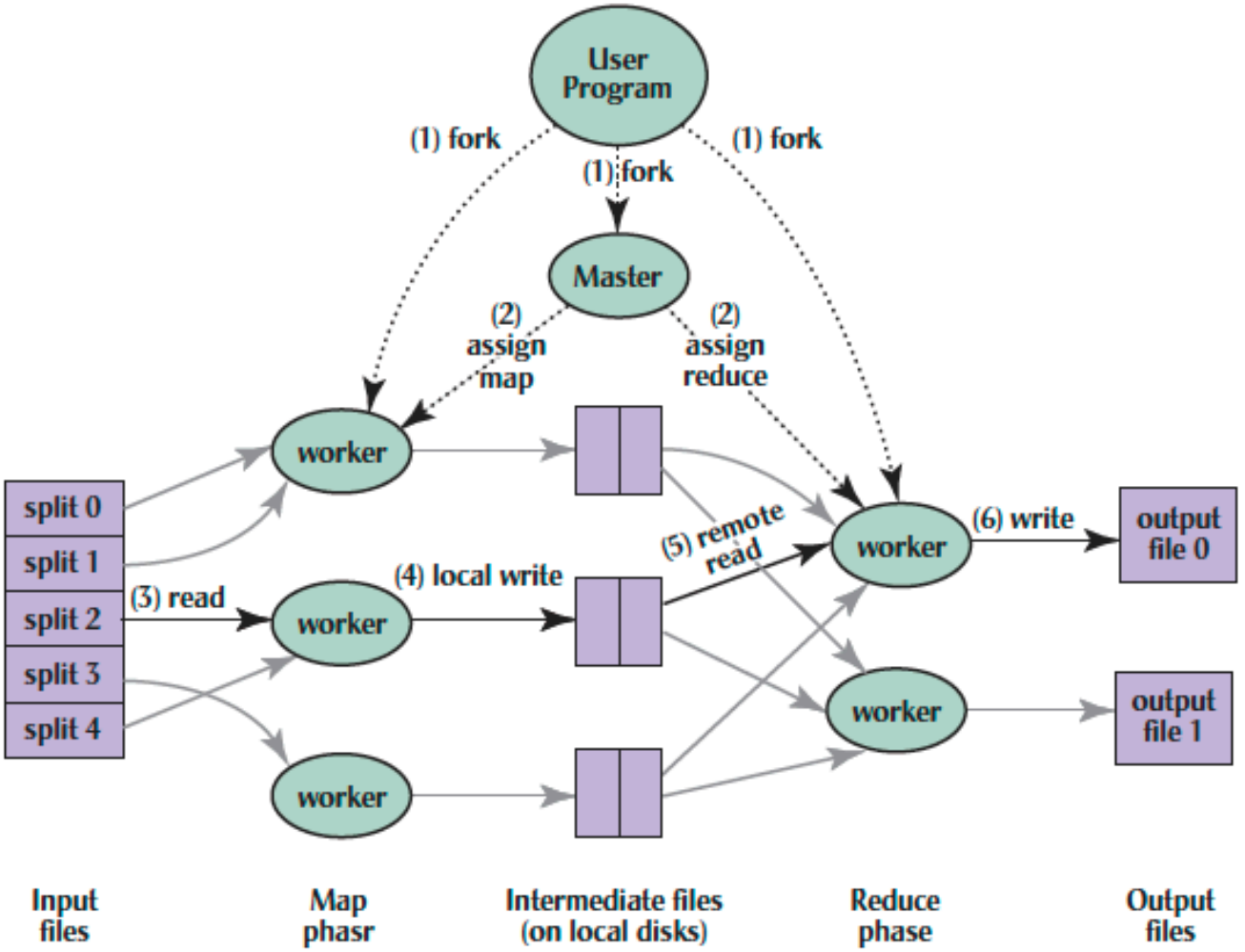
*Task of counting the number of occurrences of each word in a large collection of documents.*

The intermediate data is then sorted by MapReduce by keys and the user's Reduce function is called for each unique key. In this case, Reduce is called with a list of a "1" for each occurrence of the word that was parsed from the document. The function adds them up to generate a total word count for that word.

***Reduce*** phase: (word, list(counts)) → (word, count_sum)

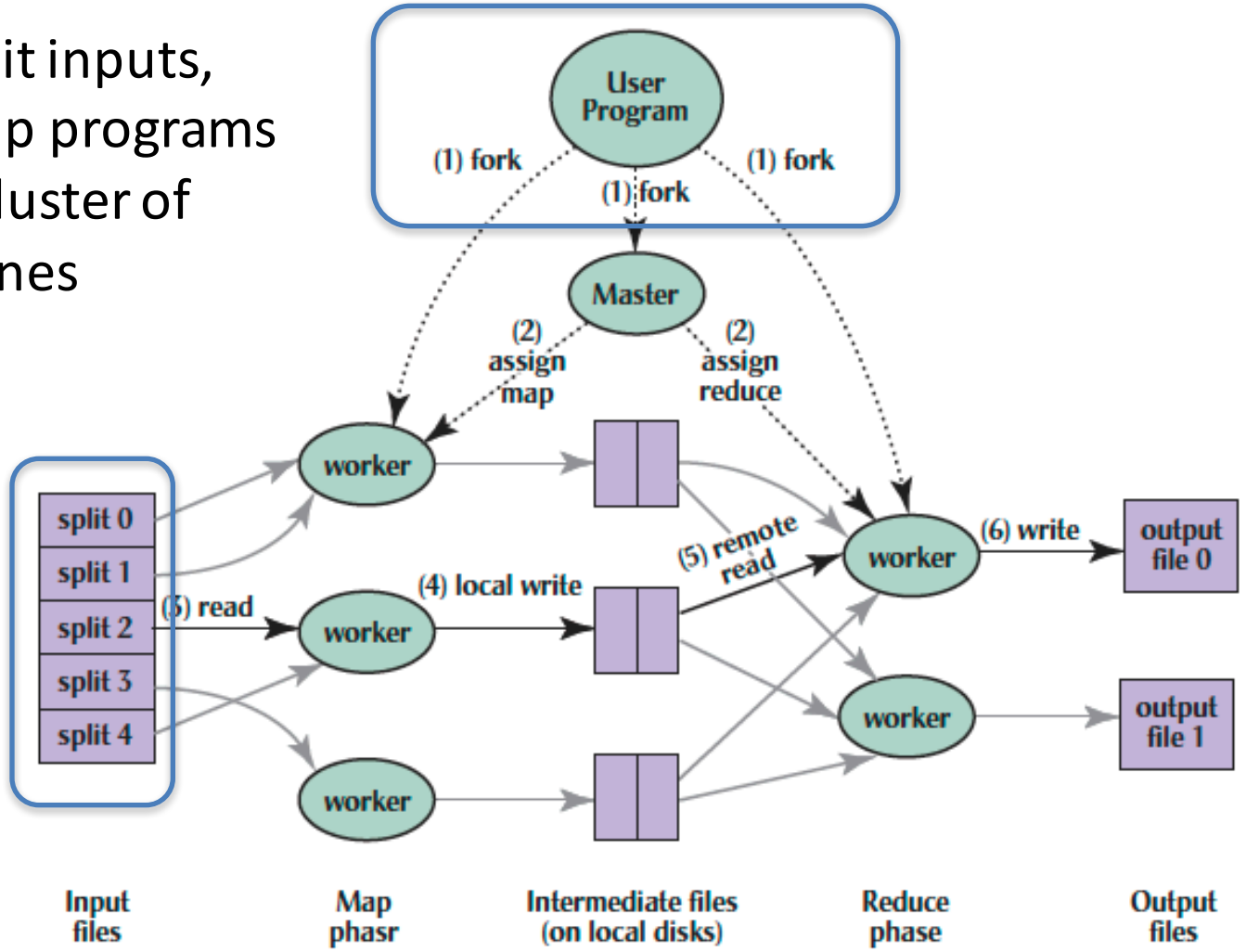
```
// ("I", [1,1]) → ("I",2)
reduce(key, values):
  result = 0
  for each v in values:
    result += v
  emit(key, result)
```

# MapReduce Implementation



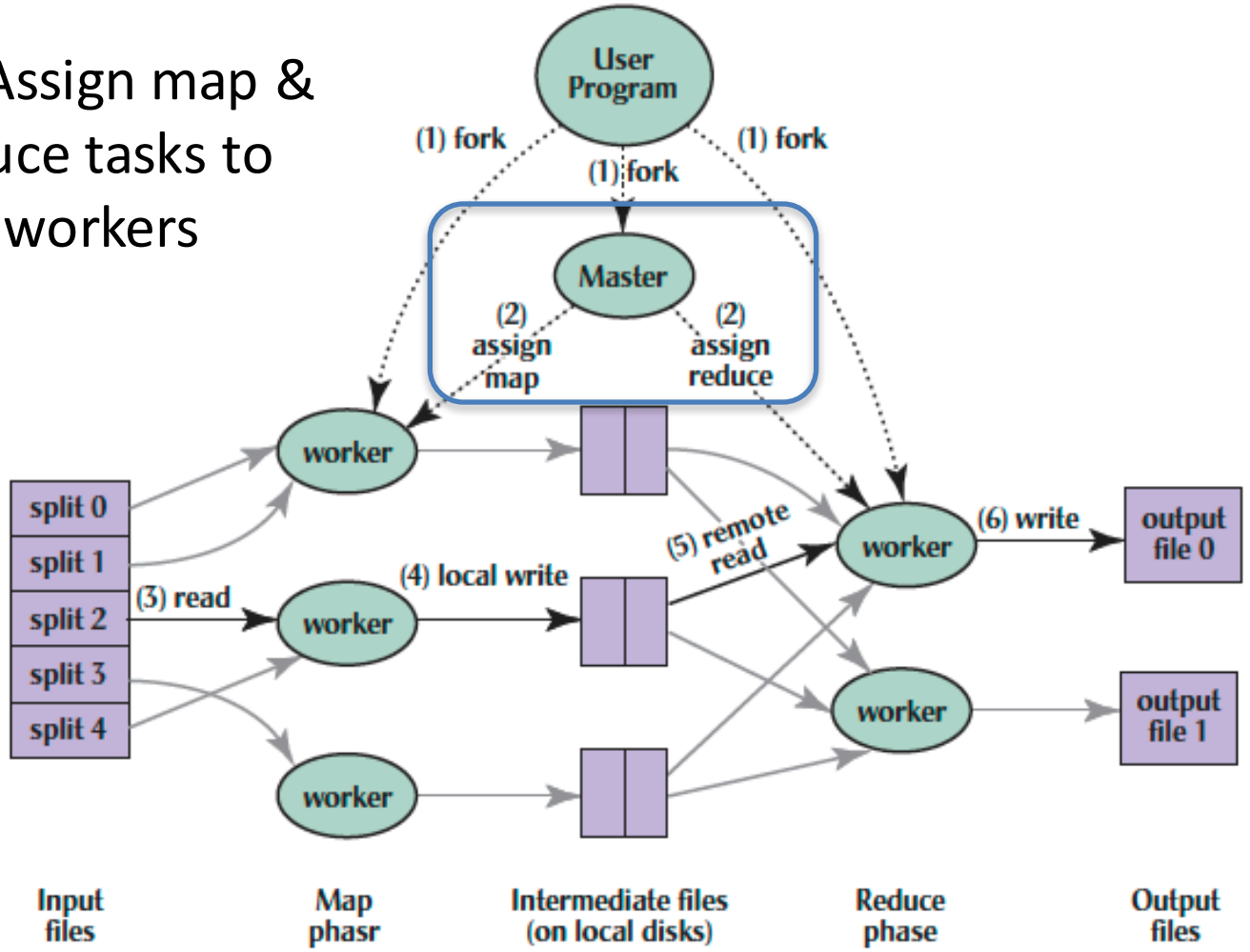
# MapReduce Execution

(1) Split inputs, start up programs on a cluster of machines



# MapReduce Execution

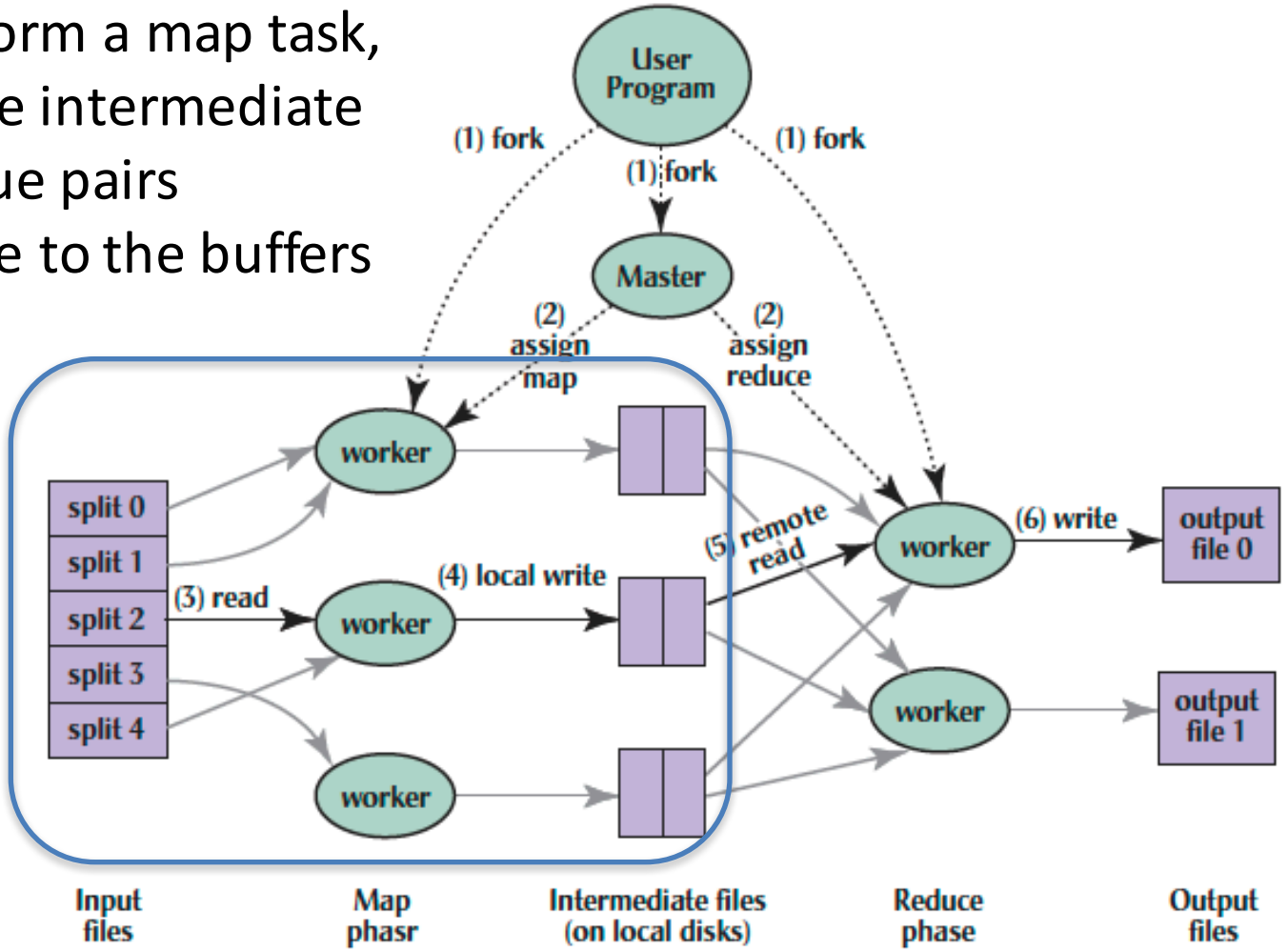
(2) Assign map & reduce tasks to idle workers



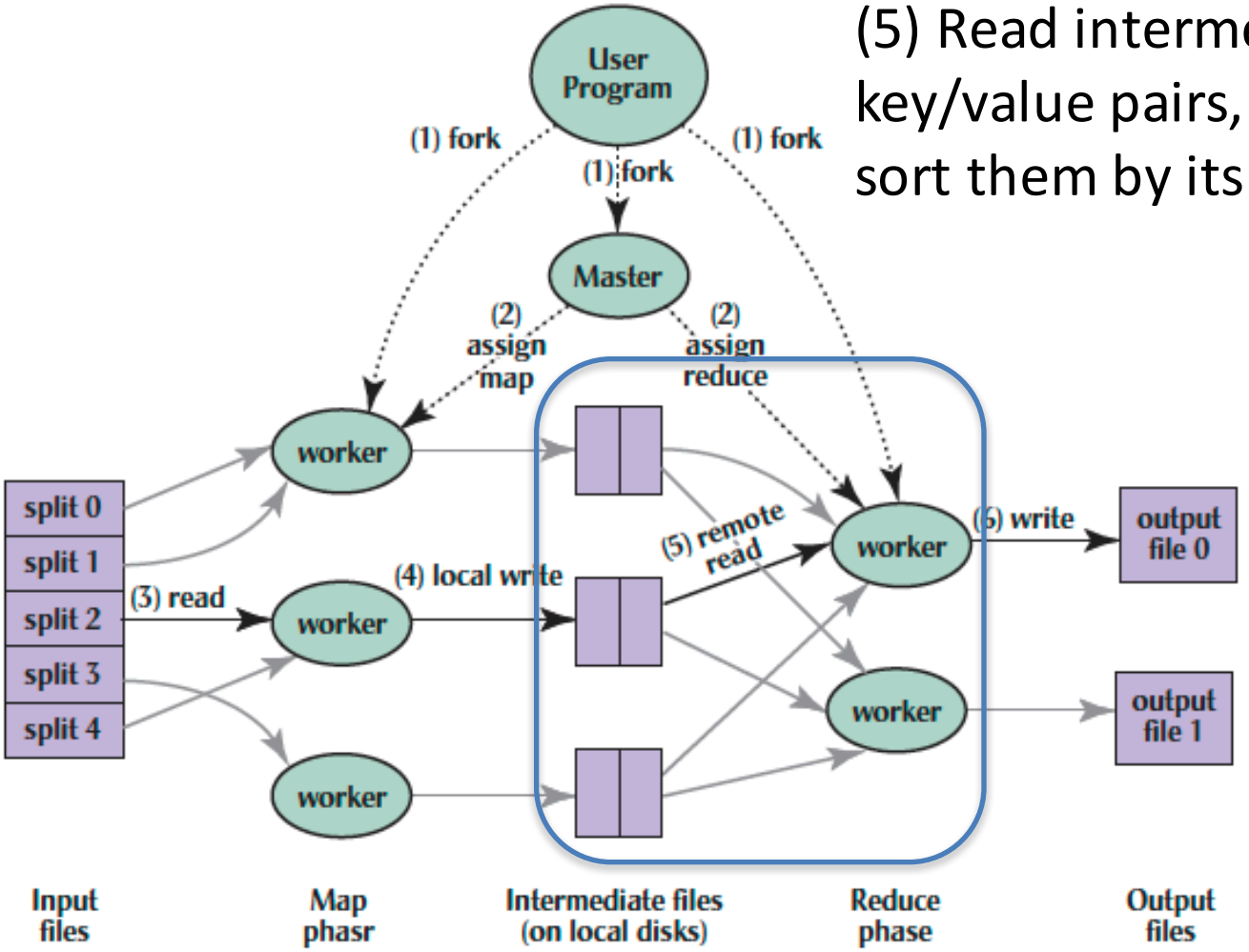
# MapReduce Execution

(3) Perform a map task, generate intermediate key/value pairs

(4) Write to the buffers

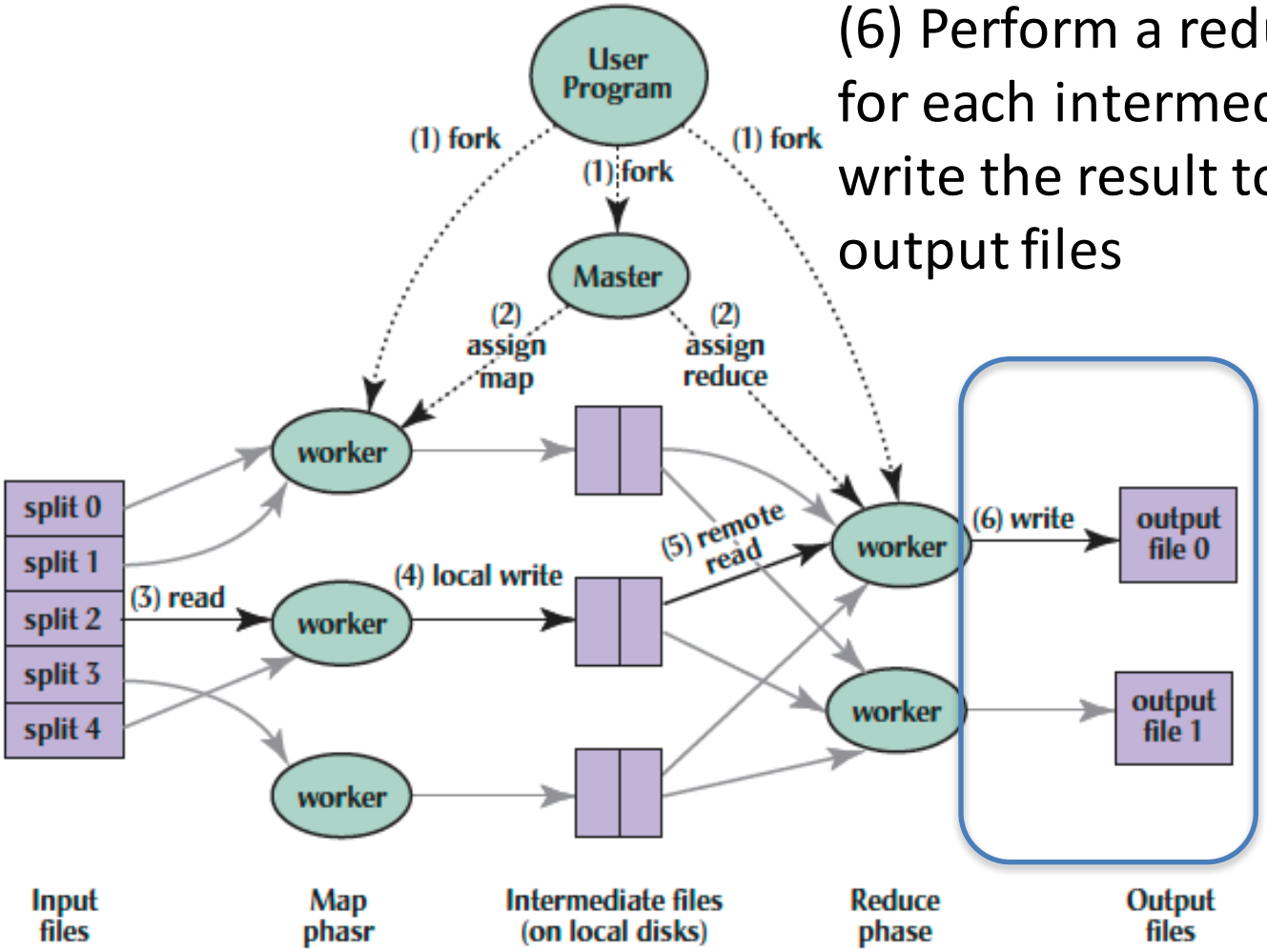


# MapReduce Execution




(5) Read intermediate key/value pairs, sort them by its key.

# MapReduce Execution




(6) Perform a reduce task for each intermediate key, write the result to the output files

# Big Data Framework: Hadoop & Spark

- Apache Hadoop
  - Open-source MapReduce Framework
  - Hadoop Distributed File System (HDFS)
  - MapReduce Java APIs


- Apache Spark
  - Fast and general engine for large-scale data processing.
  - Originally developed in the AMP lab at UC Berkeley
  - Running on HDFS
  - Provides Java, Scala, Python APIs for
    - Database
    - Machine learning
    - Graph algorithm

# WordCount in Hadoop's Java API

```
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));
    }
}
```

# Word Count in Spark's Python API

```
// RDD: primary abstraction of a distributed collection of items
file = sc.textFile("hdfs://…")
// Two kinds of operations:
// Actions: RDD → Value
// Transformations: RDD → RDD
// e.g. flatMap, Map, reduceByKey
file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
```

# The Hadoop Distributed File System

- The magic of Hadoop is not just in parallelizing the computation…
  - But building a (mostly) ***robust,*** (mostly) ***distributed file system***
- Model is to take a large group of machines and provide a robust, replicated filesystem
  - Disks and machines can arbitrarily crash, with some key exceptions
    - Done by replication: Usually at least 3x
  - Can also localize replicas
    - EG, one copy in each rack

# HDFS Blocks

- Files are broken into fixed-sized blocks
  - Commonly 128 MB!
- Small-element latency is awful…
  - It takes the same amount of time to read 1B as it does 128 MB!
  - But that is a sensible decision:
    - A typical spinning disk already biases towards accessing large blocks: 200+ MB/s bandwidth, 5+ ms latency
    - HDFS needs to store "metadata" in memory
  - HDFS is designed for high ***throughput*** operations
- Any block is replicated across multiple separate ***DataNodes***
  - Usually at least 3x replication

# HDFS NameNode

- The ***NameNode*** tracks the filesystem metadata
  - For each file, what blocks on which DataNodes
- The ***NameNode*** is both a potential bottleneck and point of failure
  - Need lots of memory to keep all the filesystem metadata in RAM
    - Since that is latency-bound
  - Requests all go to the NameNode
    - Single point of contention
  - NameNode fails and the system goes down!

# HDFS's Single Points of Failure…

- The NameNode itself
  - The backup gives fail-over, but that is not the same
  - NameNode, unlike DataNodes, are often not trivially replaceable
    - Since the NameNode often requires more memory
- Often: the switch
  - Need multiple redundant networks in a rack if you need to survive switch failures
- Often: the power
  - Need systems with multiple power supplies and redundant power if you need to survive power failures
- But it's a tradeoff:
  - HDFS is not ***supposed*** to be "high availability", but "inexpensive and big and (mostly) reliable":
    - '5-9s of availability' aka 'operating 99.999% of the time' allows for just 5 minutes downtime in a year!

# Summary

- Warehouse Scale Computers
  - New class of computers
  - Scalability, energy efficiency, high failure rate
- Request-level parallelism

  e.g. Web Search
- Data-level parallelism on a large dataset
  - MapReduce
  - Hadoop, Spark