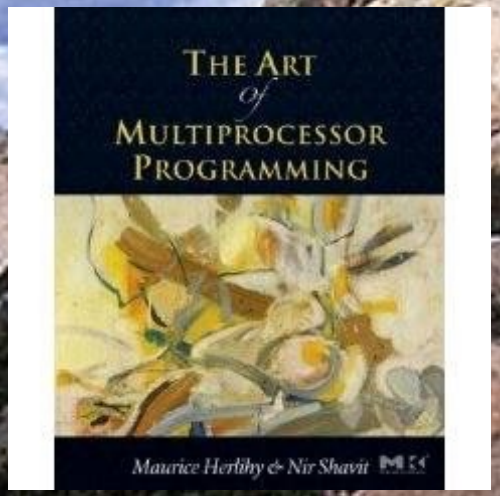
Data Parallelism from a Multicore Perspective

Hydra
Summer 2021



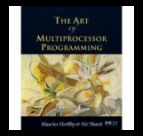
Maurice Herlihy
Brown University

Did You Ever Wonder …

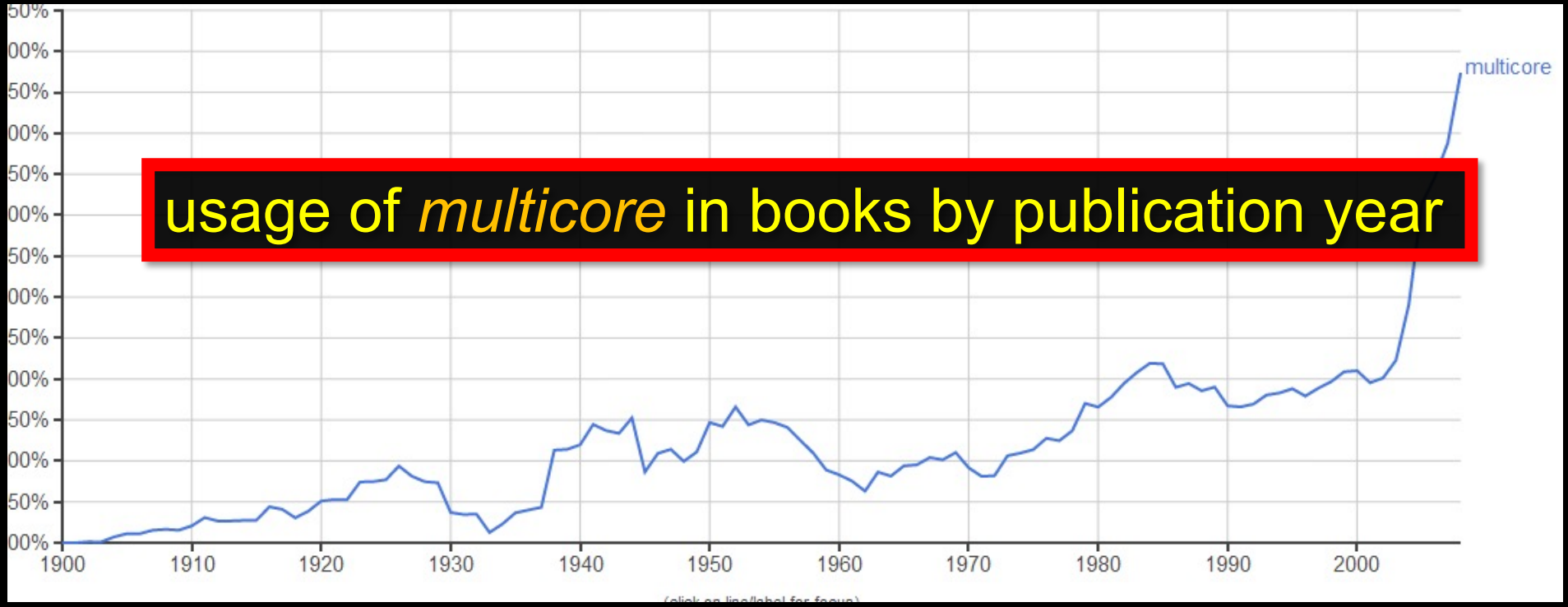
When did the term *multicore* become popular?

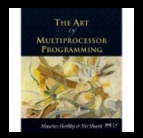
"A multi-core processor is a single computing component with two or more independent actual central processing units, which are the units that read and execute program instructions."

wikipedia

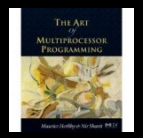


Let's Ask Google Ngram!

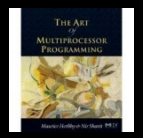




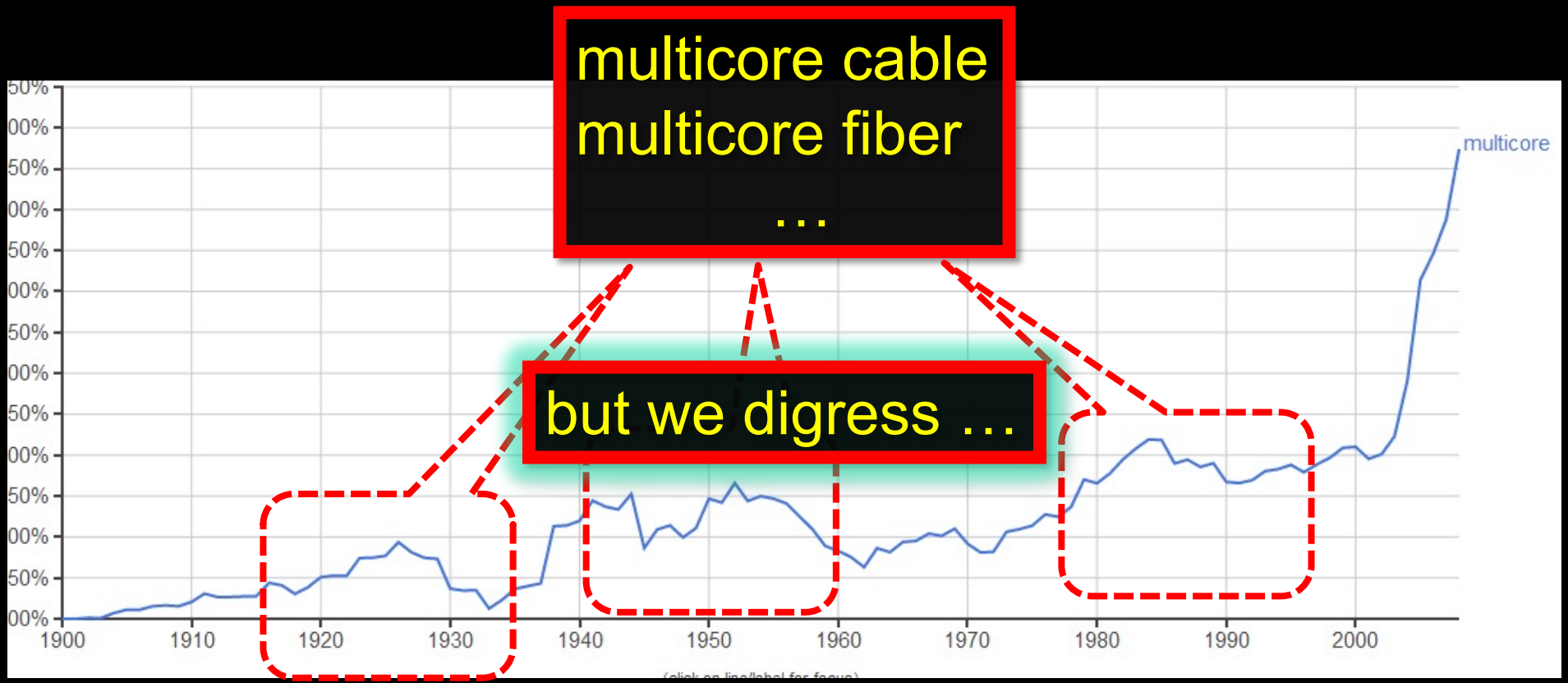
Let's Ask Google Ngram!

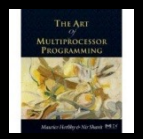


Let's Ask Google Ngram!

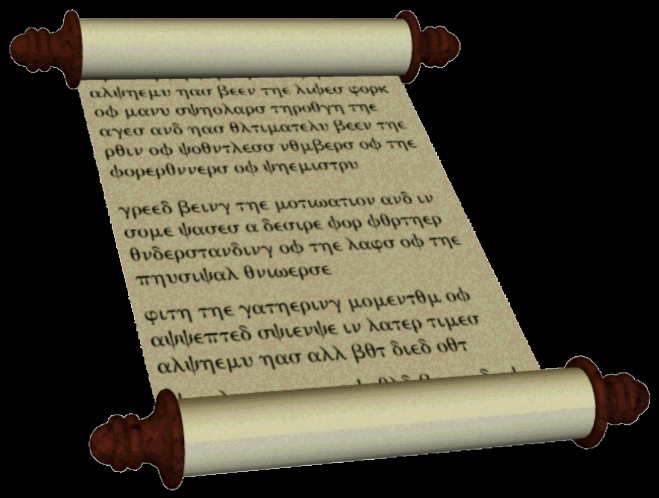


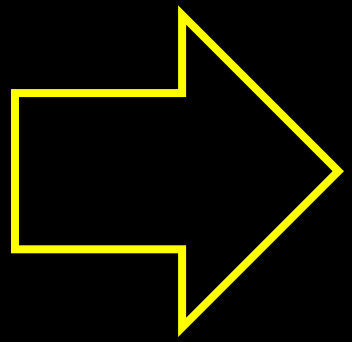
Let's Ask Google Ngram!





WordCount





alpha → 8

bravo → 3

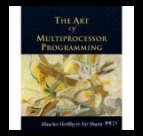
charlie → 9

…

zulu → 1

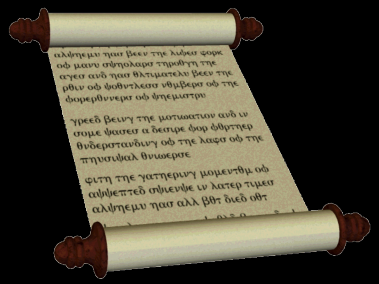
easy to do sequentially …

what about in parallel?

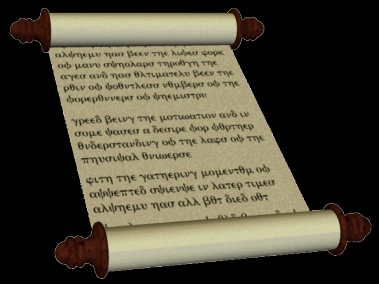


MapReduce

split text among *mapping* threads

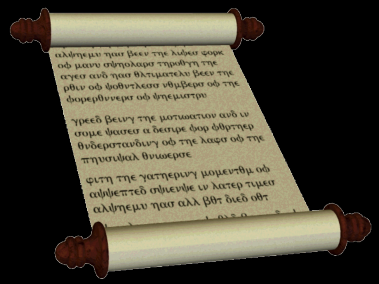


chapter 1

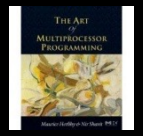


chapter 2

...



chapter *k*

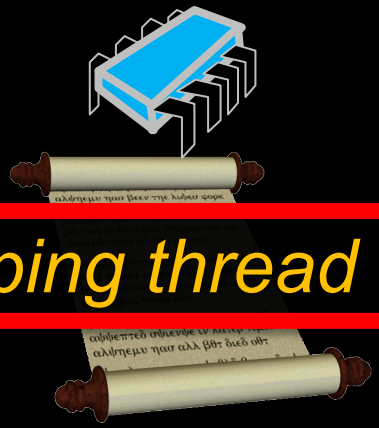


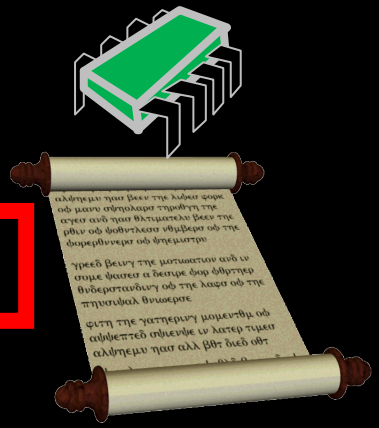
Map Phase

must count words!

must count words!

must count words!





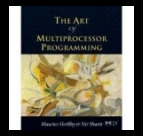
a *mapping thread* per chapter

…

chapter 1

chapter 2

chapter *k*



Map Phase

alpha → 9
juliet → 2,
alpha → 1
tango → 4

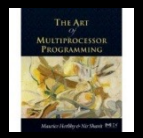
each mapper thread produces a *stream* …

of *key-value* pairs …

key: word

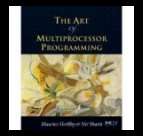
value: local count

chapter 1



Mapper Class

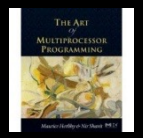
```
abstract class Mapper<IN, K, V>
    extends RecursiveTask<Map<K, V>> {
  IN input;
  public void setInput(IN anInput) {
    input = anInput;
  }
}
```



Mapper Class

```
abstract class Mapper<IN, K, V>
    extends RecursiveTask<Map<K, V>> {
  IN input;
  public void setInput(IN anInput) {
    input = anInput;
  }
}
```

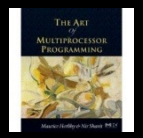
input: document fragment



Mapper Class

```
abstract class Mapper<IN, K, V>
    extends RecursiveTask<Map<K, V>> {
  IN input;
  public void setInput(IN anInput) {
    input = anInput;
  }
}
```

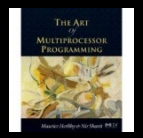
key: individual word



Mapper Class

```
abstract class Mapper<IN, K, V>
    extends RecursiveTask<Map<K, V>> {
  IN input;
  public void setInput(IN anInput) {
    input = anInput;
  }
}
```

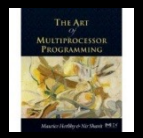
value: local count



Mapper Class

```
abstract class Mapper<IN, K, V>
    extends RecursiveTask<Map<K, V>> {
  IN input;
  public void setInput(IN anInput) {
    input = anInput;
  }
}
```

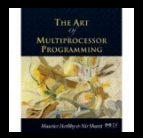
a task that runs in parallel with other tasks



Mapper Class

```
abstract class Mapper<IN, K, V>
    extends RecursiveTask<Map<K, V>> {
  IN input;
  public void setInput(IN anInput) {
    input = anInput;
  }
}
```

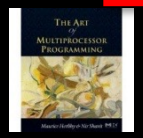
produces a map: word → count



Mapper Class

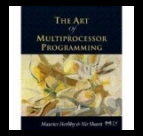
```
abstract class Mapper<IN, K, V>
    extends RecursiveTask<Map<K, V>> {
  IN input;
  public void setInput(IN anInput) {
     input = anInput;
  }
}
```

initialize input: which document fragment?



WordCount Mapper

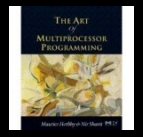
```
class WordCountMapper extends
  mapreduce.Mapper<
      List<String>, String, Long
    > {
  …
  }
```



WordCount Mapper

```
class WordCountMapper extends
   mapreduce.Mapper<
        List<String>, String, Long
     > {
   …
}
```

document fragment is list of words

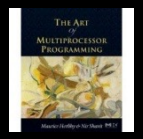


WordCount Mapper

```
class WordCountMapper extends
   mapreduce.Mapper<
        List<String>, String, Long
    > {
   …
```

document fragment is list of words

map each word …



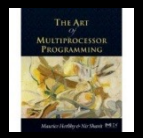
WordCount Mapper

```
class WordCountMapper extends
   mapreduce.Mapper<
        List<String>, String, Long
    > {
   …
```

document fragment is list of words

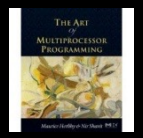
map each word …

to its count in the fragment



WordCount Mapper

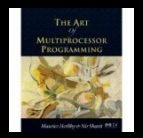
```
Map<String,Long> compute() {
  Map<String,Long> map = new HashMap<>();
      for (String word : input) {
        map.merge(word,
                  1L,
                  (x, y) -> x + y);
      }
      return map;
    }
  }
```



WordCount Mapper

```
Map<String,Long> compute() {
  Map<String,Long> map = new HashMap<>();
    for (String word : input) {
      map.merge(word,
                1L,
                (x, y) -> x + y);
    }
  }
```

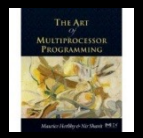
the `compute()` method constructs the local word count



WordCount Mapper

```
Map<String,Long> compute() {
  Map<String,Long> map = new HashMap<>();
    for (String word : input) {
      map.merge(word,
                1L,
                (x, y) -> x + y);
    }
    return map;
  }
}
```

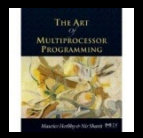
create a map to hold the output



WordCount Mapper

```
Map<String,Long> compute() {
  Map<String,Long> map = new HashMap<>();
    for (String word : input) {
      map.merge(word,
                1L,
                (x, y) -> x + y);
    }
    return map;
  }
}
```

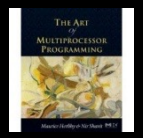
examine each word in the document fragment



WordCount Mapper

```
Map<String,Long> compute() {
  Map<String,Long> map = new HashMap<>();
      for (String word : input) {
        map.merge(word,
                  1L,
                  (x, y) -> x + y);
      }
      return map;
    }
  }
```

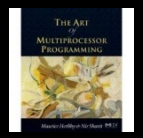
increment that word's count in the map



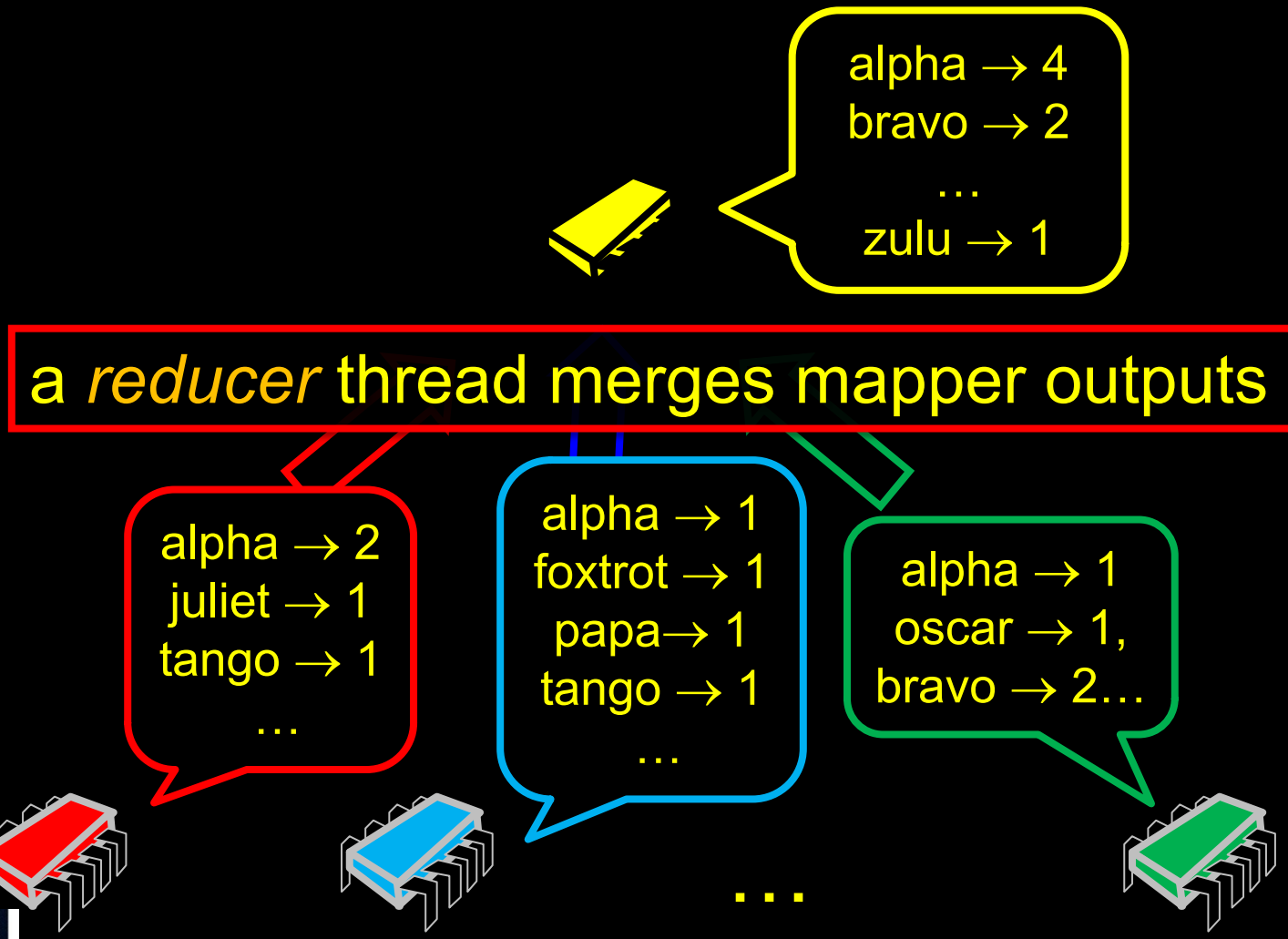
WordCount Mapper

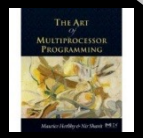
```
Map<String,Long> compute() {
  Map<String,Long> map = new HashMap<>();
      for (String word : input) {
        map.merge(word,
                  1L,
                  (x, y) -> x + y);
      }
      return map;
    }
  }
```

when the local count is complete, return the map

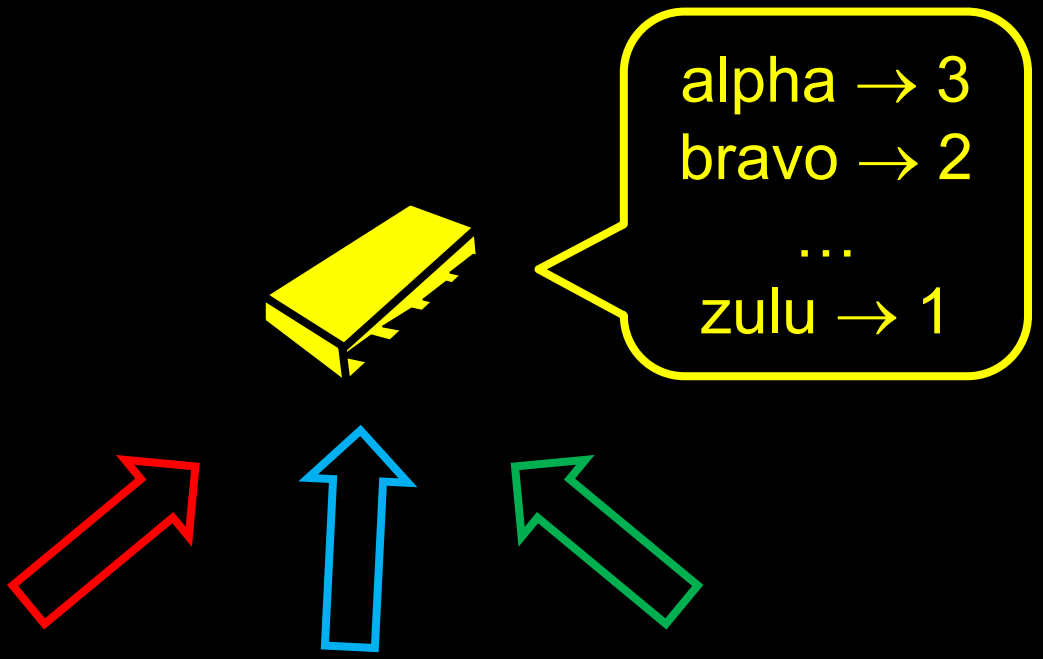


Reduce Phase





Reduce Phase

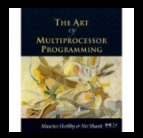


the reducer task produces a *stream* …

of *key-value* pairs …

key: word

value: word count



Reducer Class

```
abstract class Reducer<K, V, OUT>
    extends RecursiveTask<OUT> {
  K key;
  List<V> valueList;
  public void setInput(
    K aKey,
    List<V> aList) {
    key = aKey;
    valueList = aList;
  }
}
```

Reducer Class

```
abstract class Reducer<K, V, OUT>
    extends RecursiveTask<OUT> {
  K key;
  List<V> valueList;
  public void setInput(
    K aKey,
    List<V> aList) {
    key = aKey;
    valueList = aList;
  }
}
```

each reducer is given a single key (word)

Reducer Class

```
abstract class Reducer<K, V, OUT>
    extends RecursiveTask<OUT> {
  K key;
  List<V> valueList;
  public void setInput(
    K aKey,
    List<V> aList) {
    key = aKey;
    valueLis
  }
}
```

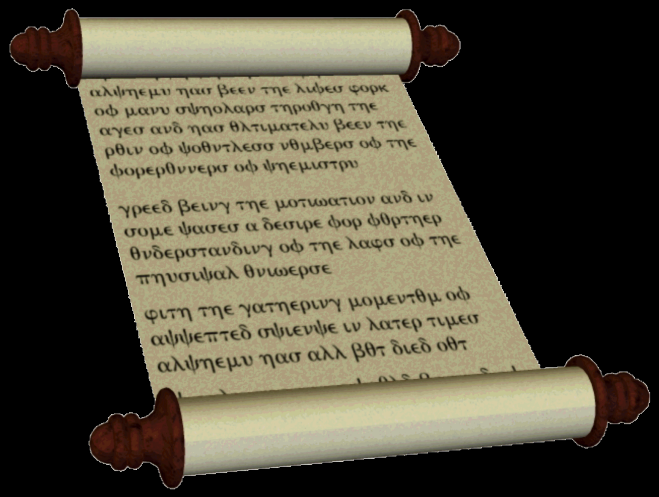
and a list of associated values
(word count per fragment)

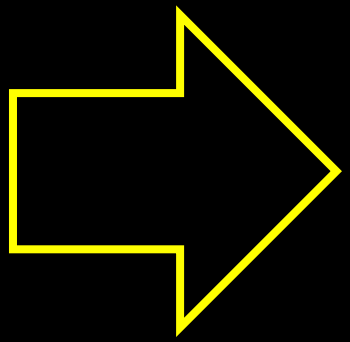
Reducer Class

```
abstract class Reducer<K, V, OUT>
    extends RecursiveTask<OUT> {
  K key;
  List<V> valueList;
  public void setInput(
    K aKey,
    List<V> aList) {
    key = aKey;
    valu
  }
}
```

It produces a single summary value (the total count for that word)

WordCount

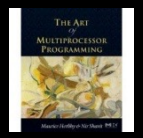




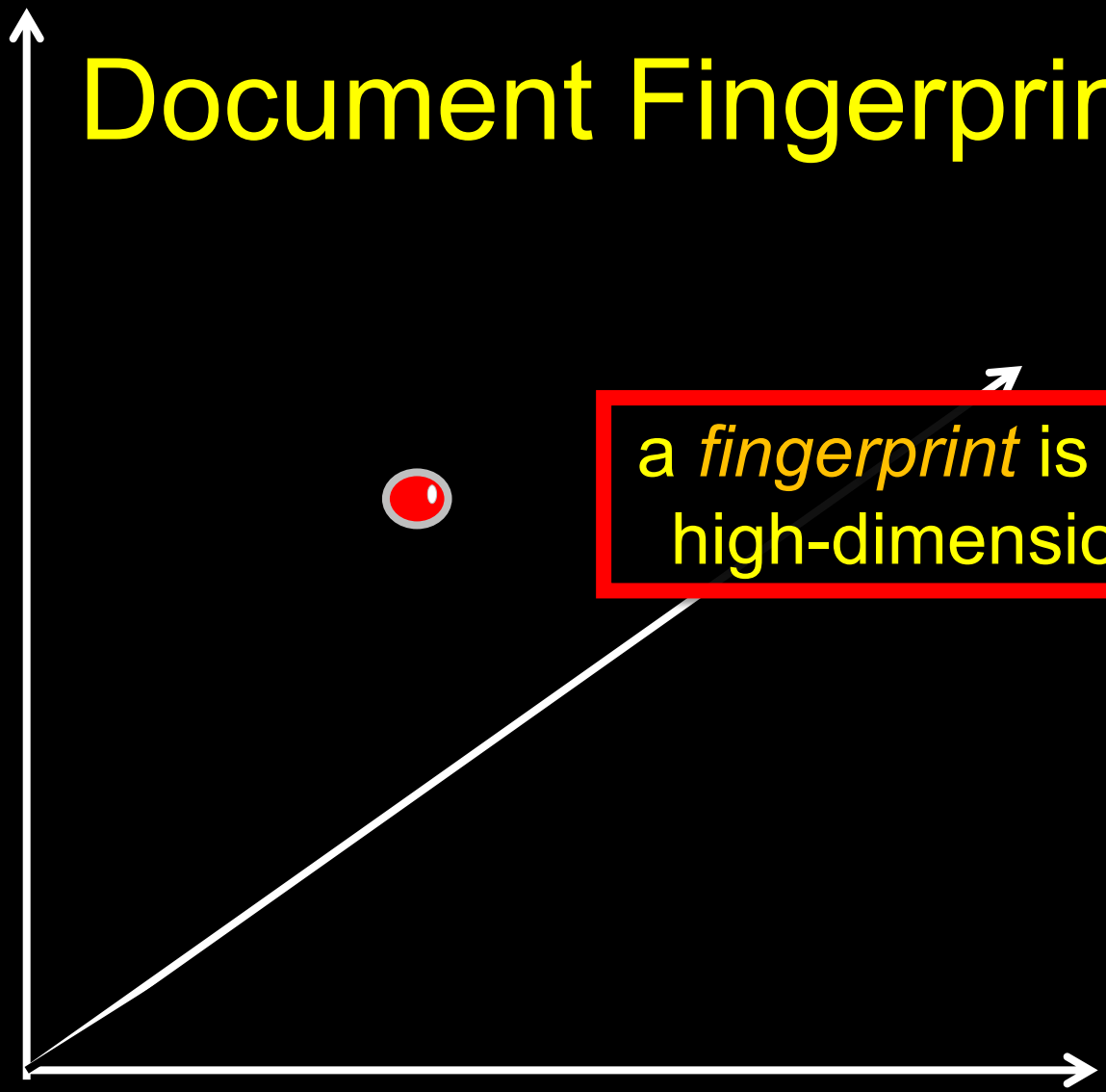
0.037
0.002
0.045
…
0.000

normalizing document wordcount

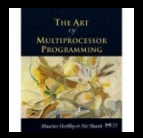
gives a *fingerprint* vector



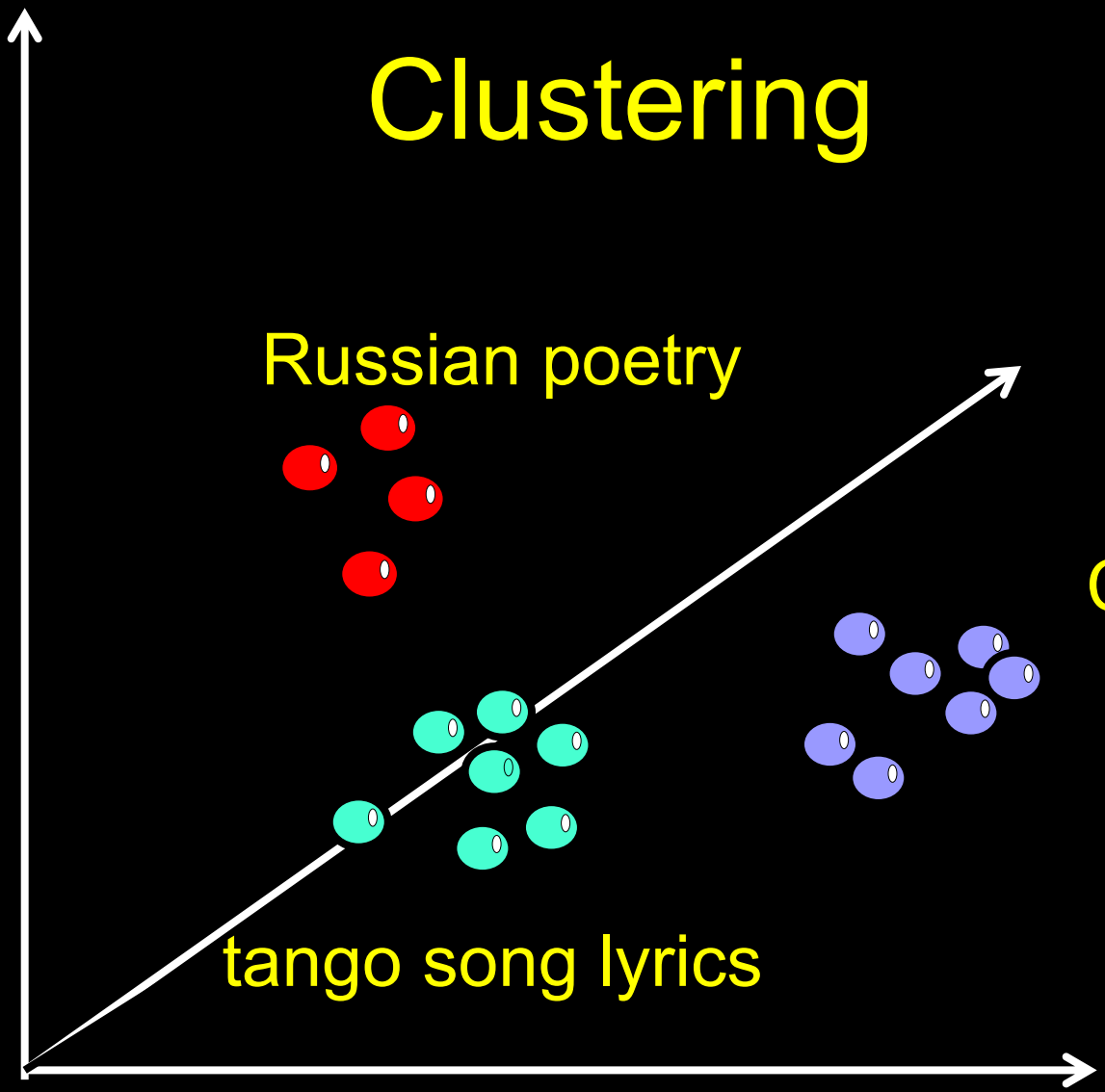
Document Fingerprint

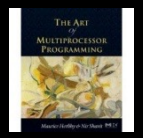


a *fingerprint* is a point in a high-dimensional space



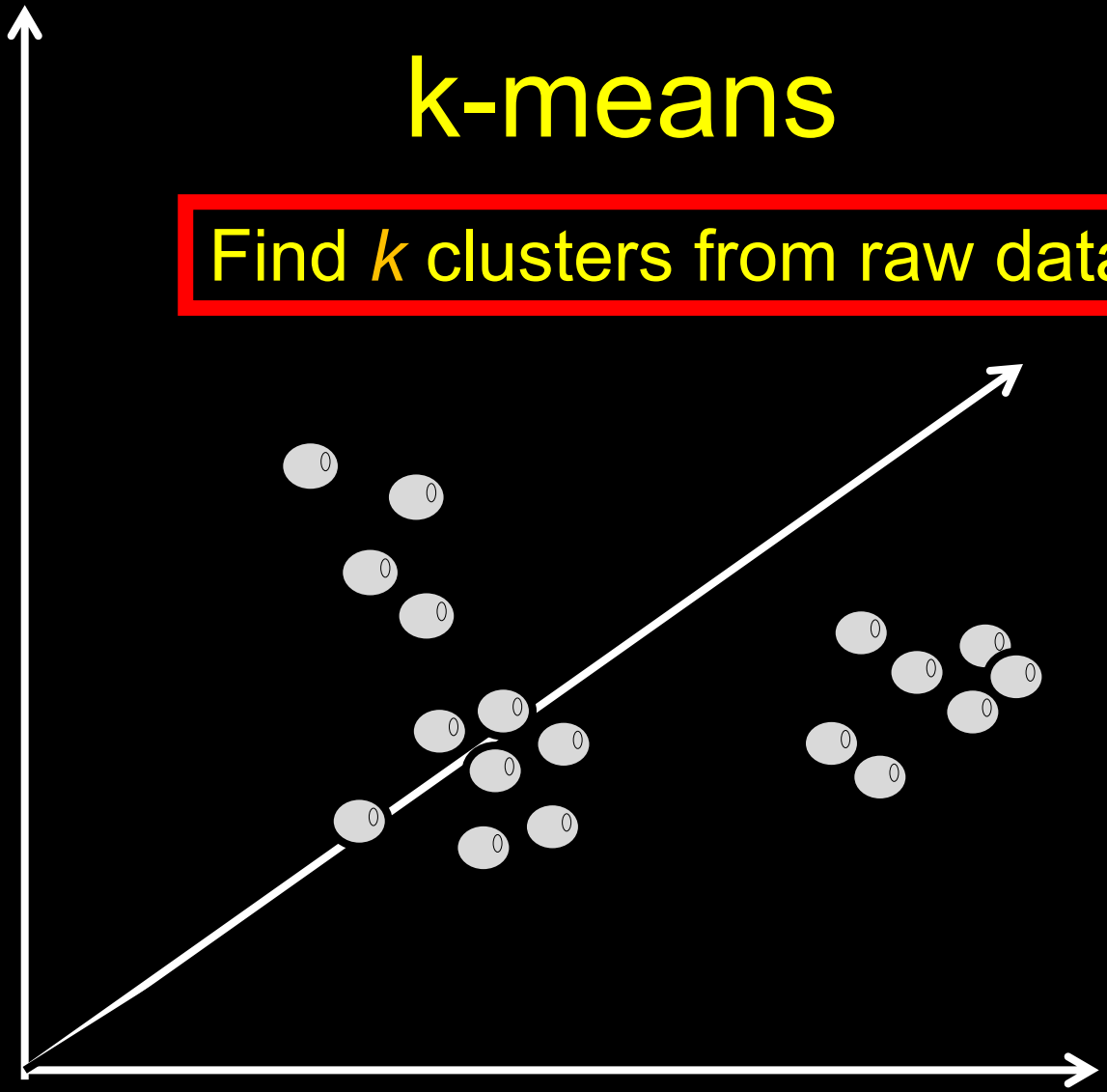
Clustering

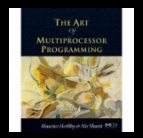

Russian poetry
Computer Science papers
tango song lyrics


The Art of Multiprocessor Programming

k-means

Find *k* clusters from raw data



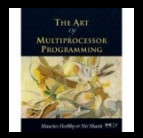


k-means

Find *k* clusters from raw data

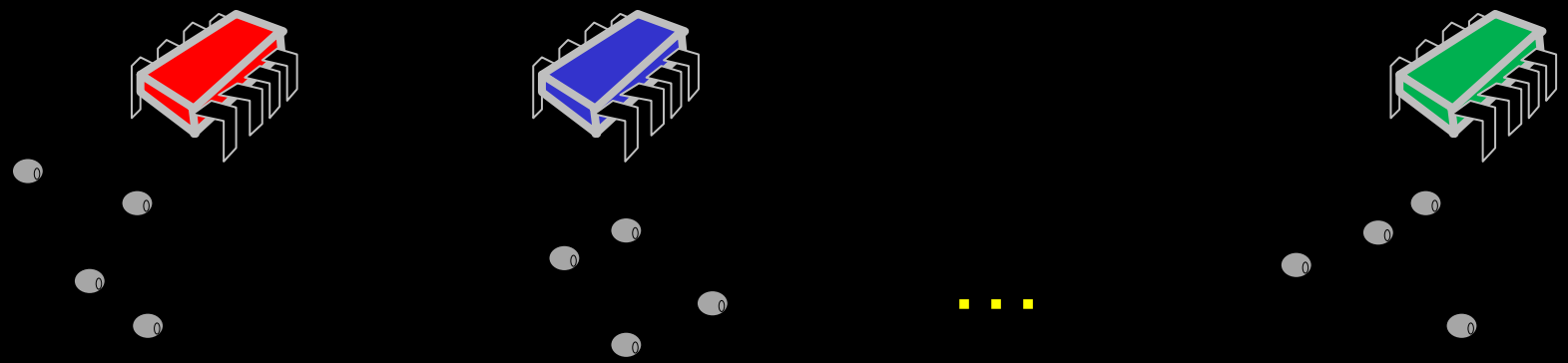
each vector closer to those in same cluster …

than in different clusters.

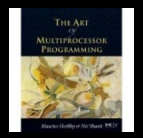


MapReduce

split points among *mapping* threads

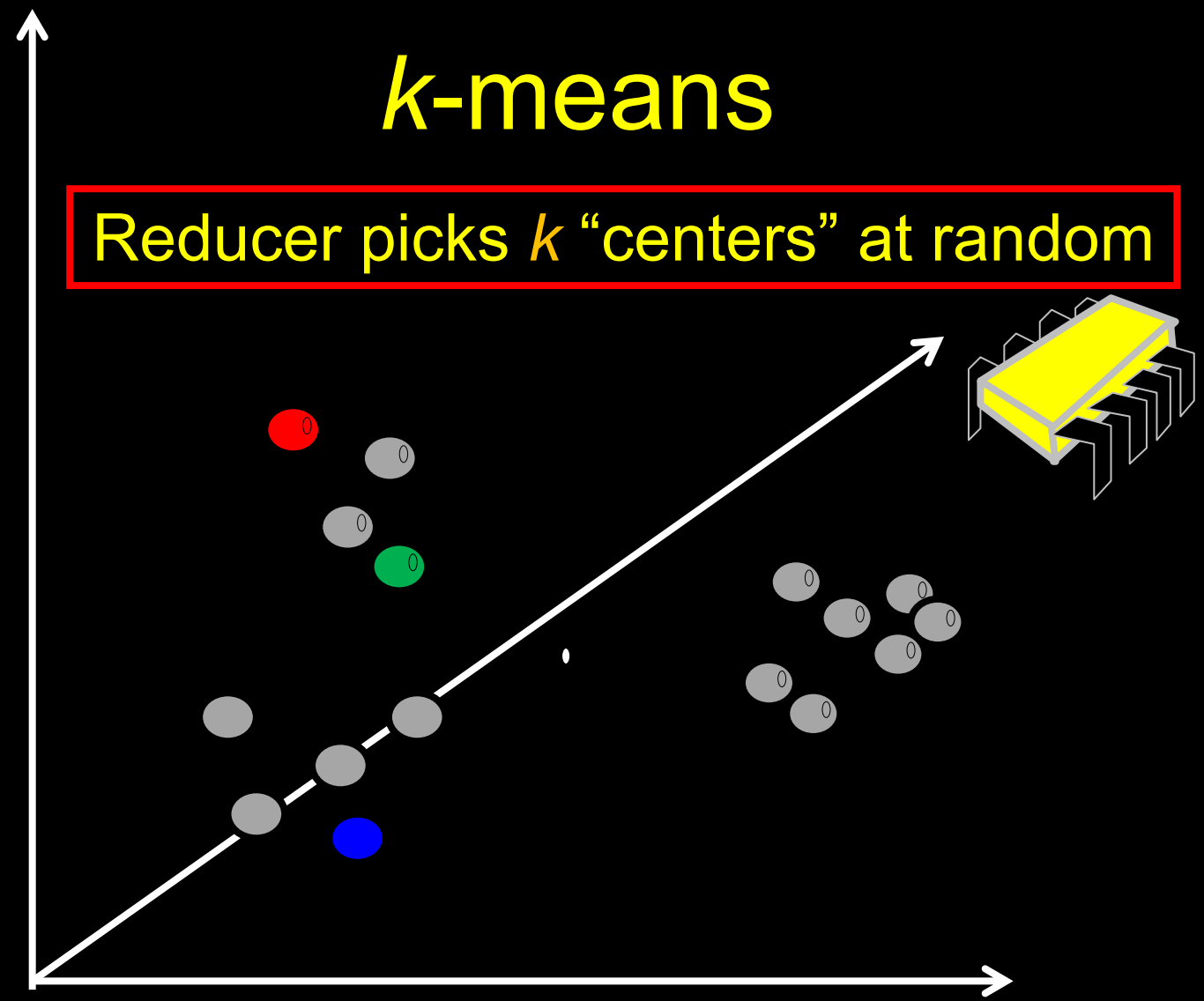


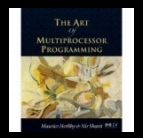
thread 1 thread 2 … thread *k*



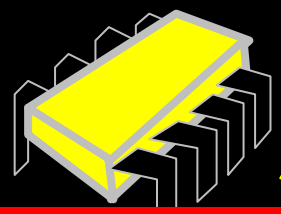
k-means

Reducer picks *k* “centers” at random





Reduce Phase

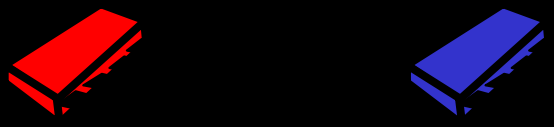


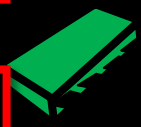
$0 \rightarrow c_0$
$1 \rightarrow c_1$
$2 \rightarrow c_2$

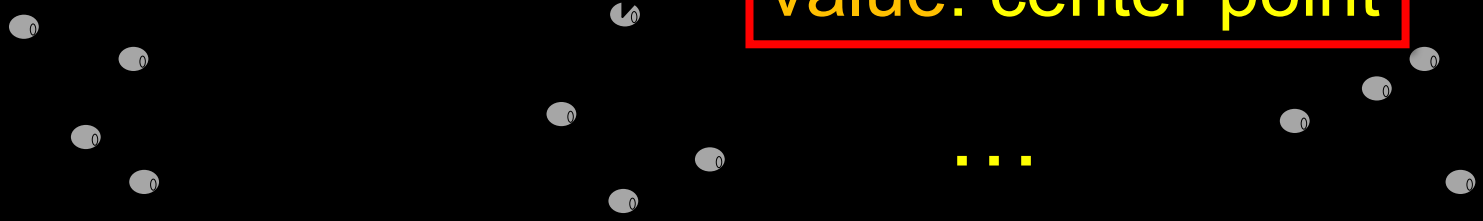
reducer sends *key-value* pair to mappers

key: cluster number

value: center point



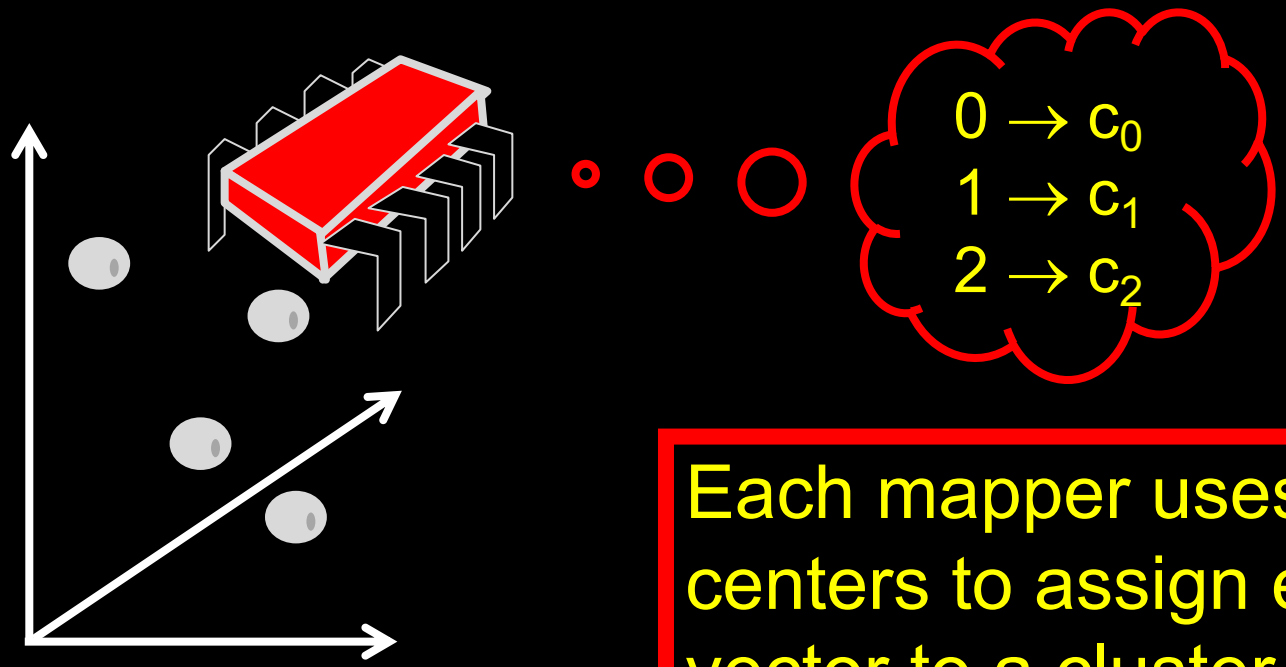




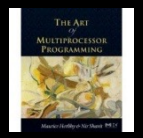
…

thread 1 thread 2 thread *k*

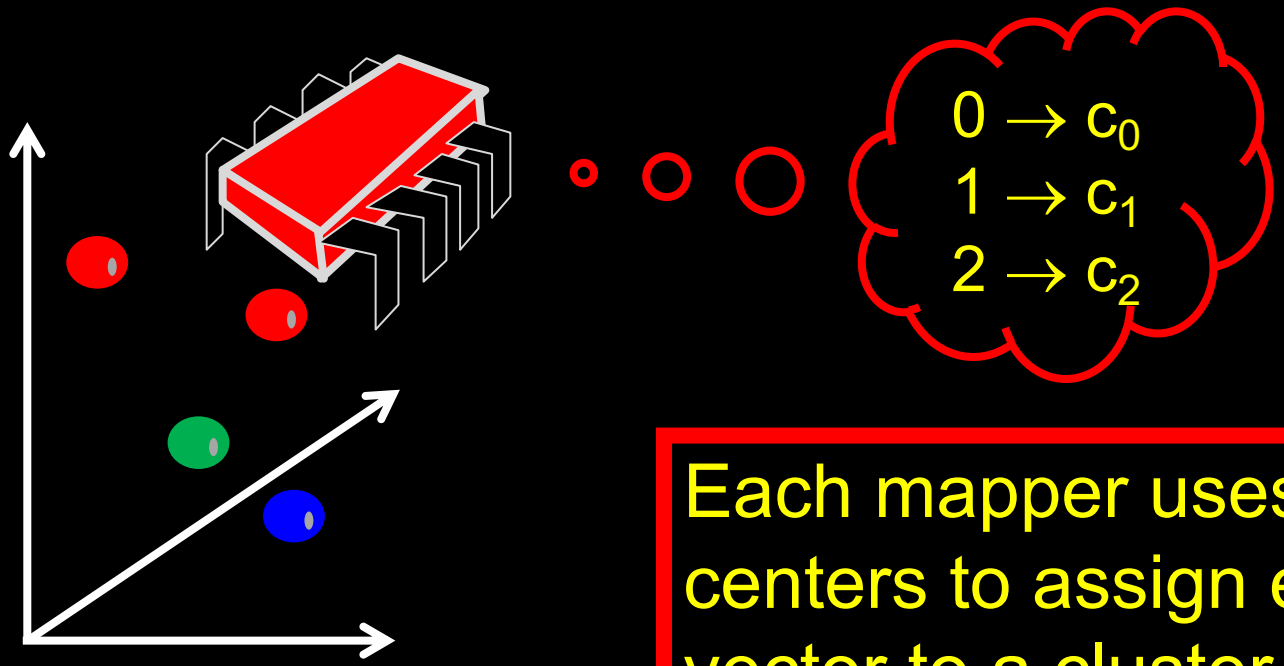
Mappers



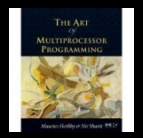
Each mapper uses centers to assign each vector to a cluster



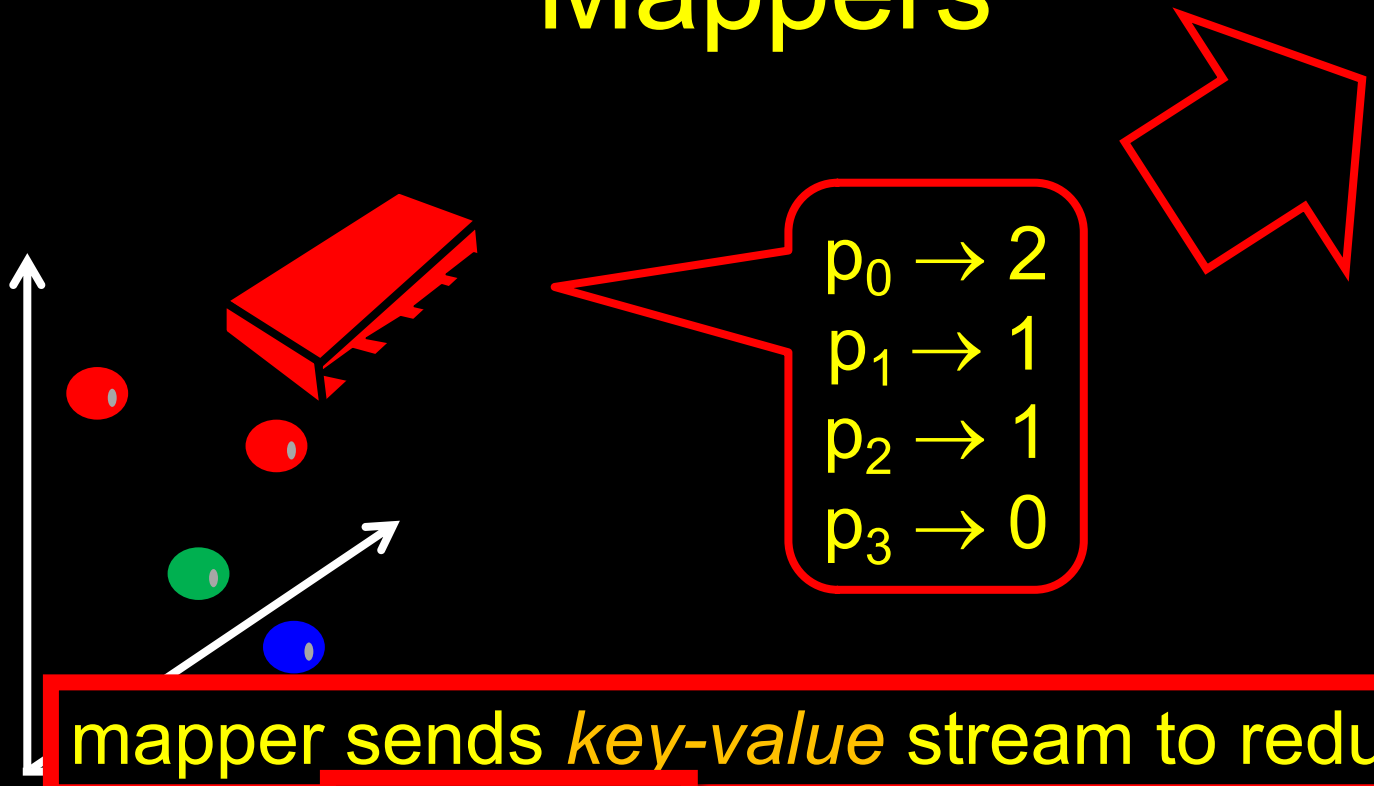
Mappers



Each mapper uses centers to assign each vector to a cluster



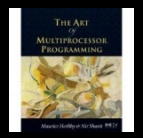
Mappers



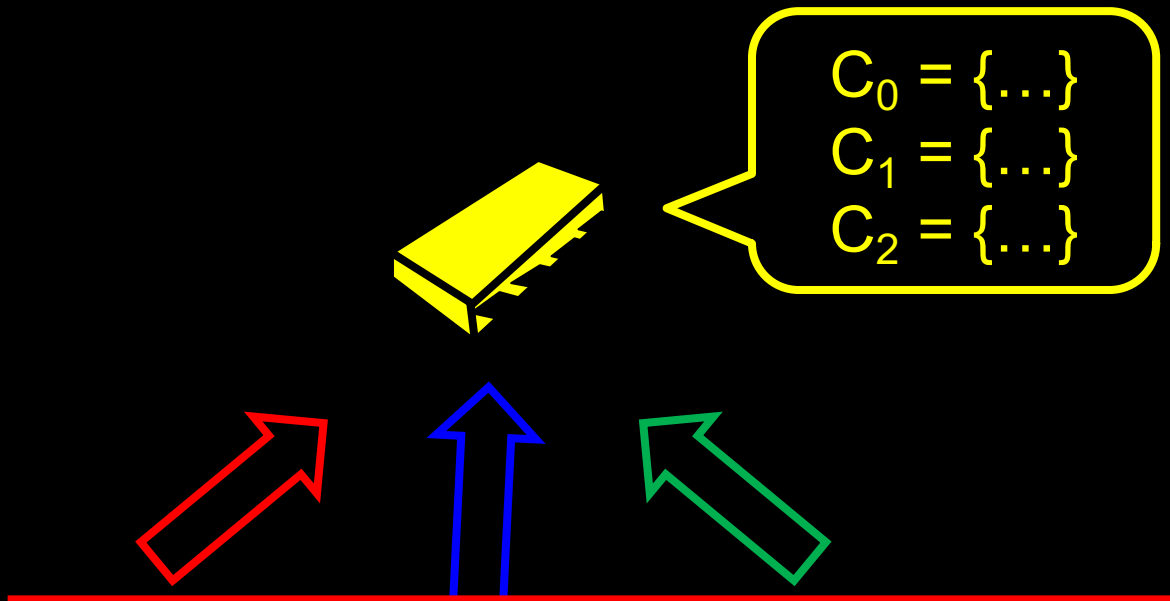
mapper sends *key-value* stream to reducer

key: point

value: cluster ID

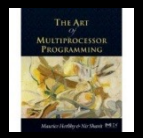


Back at the Reducer

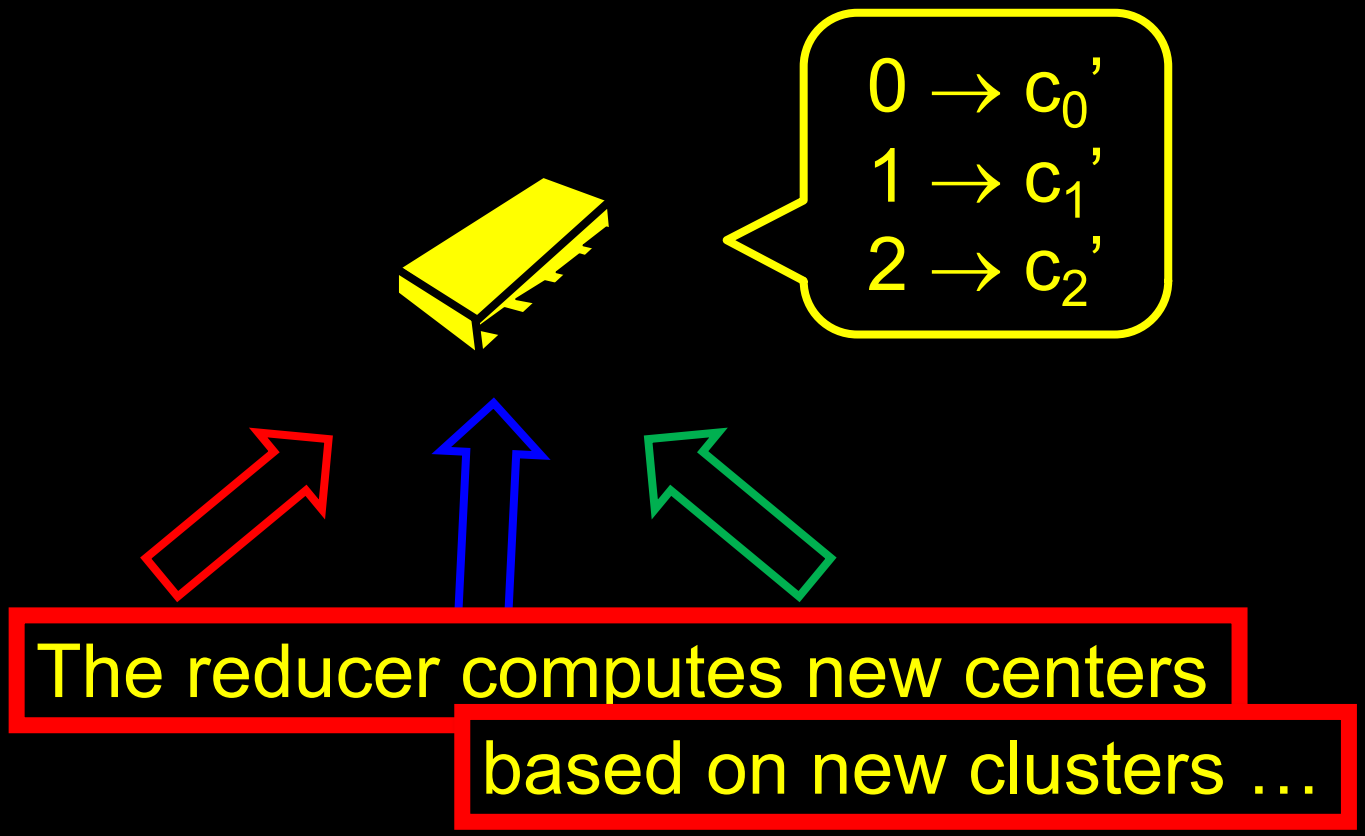


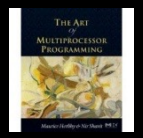
The reducer merges the streams ….

and assembles clusters …

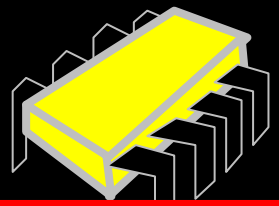


Back at the Reducer





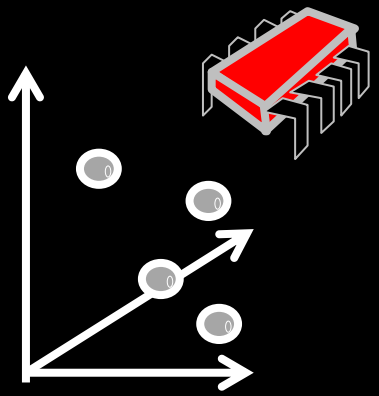
Once is Not Enough



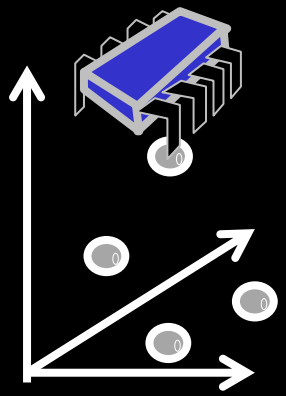
$0 \rightarrow c_0'$
$1 \rightarrow c_1'$
$2 \rightarrow c_2'$

reducer sends *new centers* to mappers

process ends when centers become stable

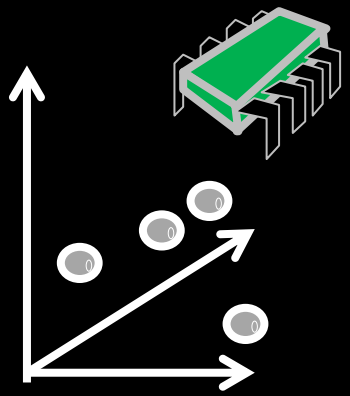


thread 1



thread 2

…



thread *k*

To Recapitulate

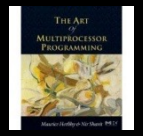
We saw two problems …

wordcount & *k-means* …

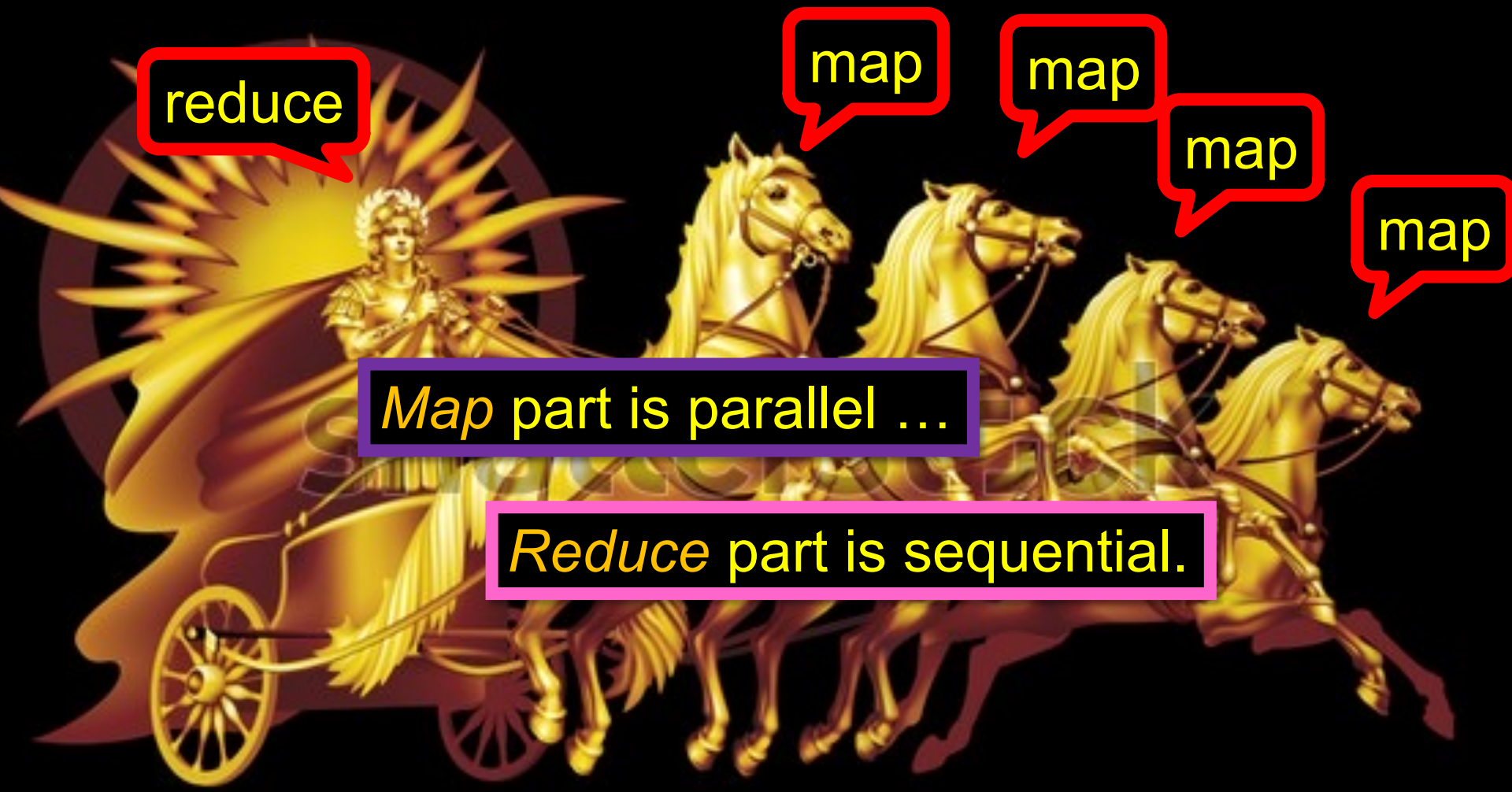
with similar parallel solutions

Map part is parallel …

Reduce part is sequential.



Big Picture

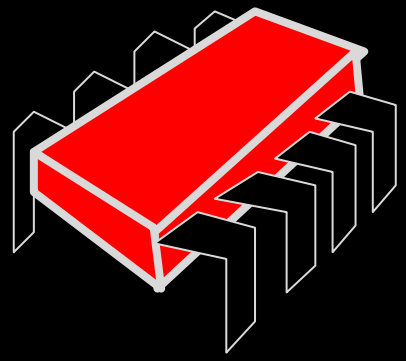


Map part is parallel …

Reduce part is sequential.

Map Function

(k_1, v_1)



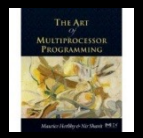
$list(k_2, v_2)$

doc, contents

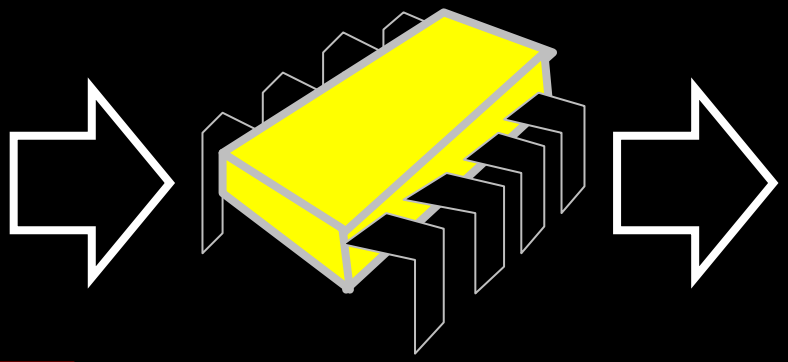
word, count

cluster ID, center

point, cluster ID



Reduce Function

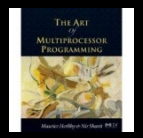
$(k_2, \text{list}(v_2))$  $\text{list}(v_2)$

word, list of counts

count

cluster ID, list of points

new cluster center



Example

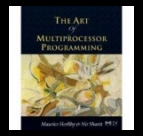
Distributed Grep

Map:

line of document

Reduce:

copy line to display



Example

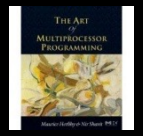
URL Access Frequency

Map:

(URL, local count)

Reduce:

(URL, total count)



Example

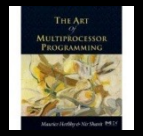
Reverse web link graph

Map:

(target link, source page)

Reduce:

(target link, list of source pages)



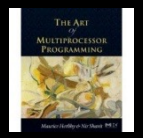
Other Examples

histogram

matrix multiplication

PageRank

Betweenness centrality



Distributed MapReduce

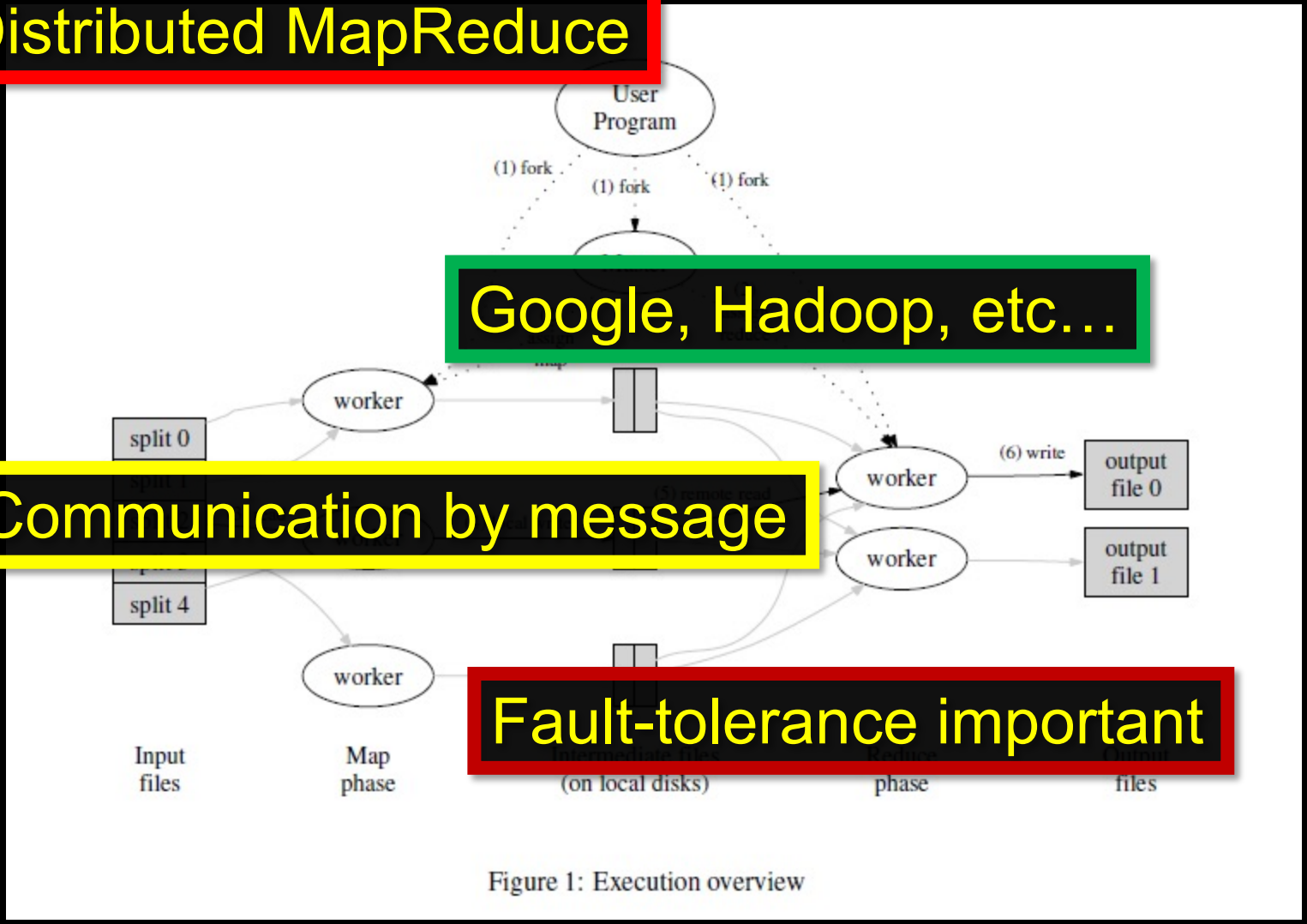
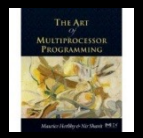


Figure 1: Execution overview

Google, Hadoop, etc…

Communication by message

Fault-tolerance important

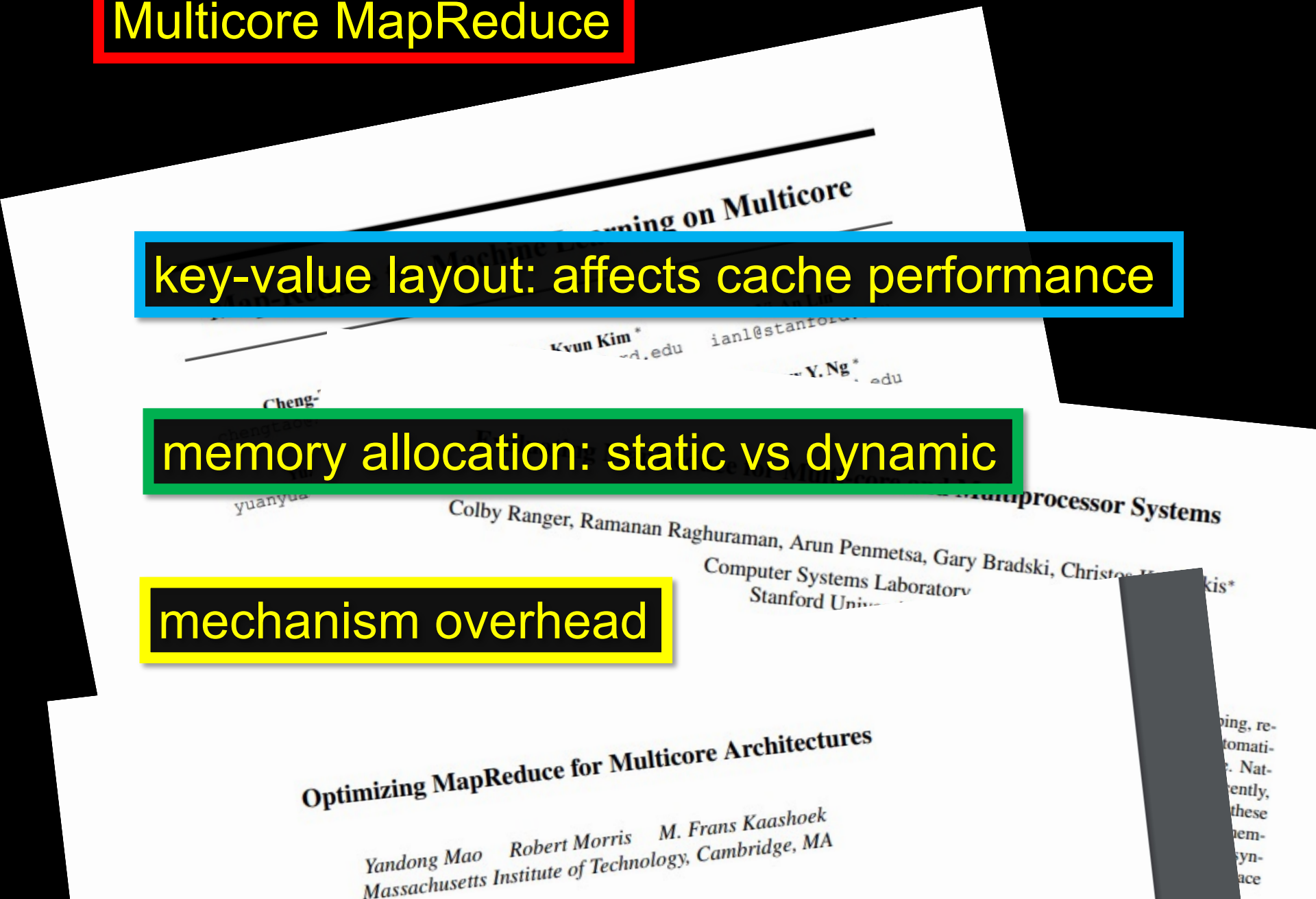


Multicore MapReduce

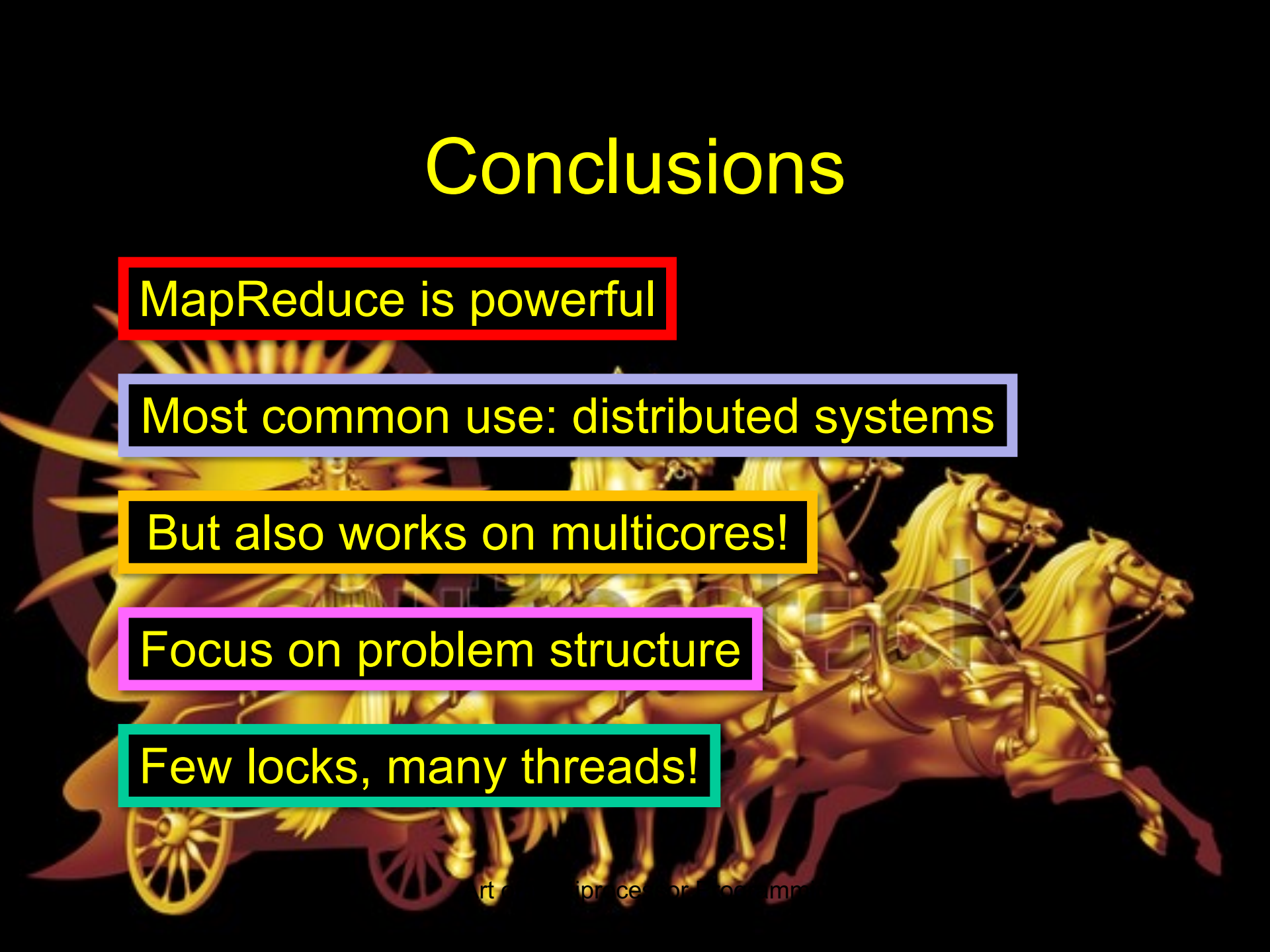
key-value layout: affects cache performance

memory allocation: static vs dynamic

mechanism overhead



Conclusions



MapReduce is powerful

Most common use: distributed systems

But also works on multicores!

Focus on problem structure

Few locks, many threads!