Concurrency: Multi-core Programming & Data Processing

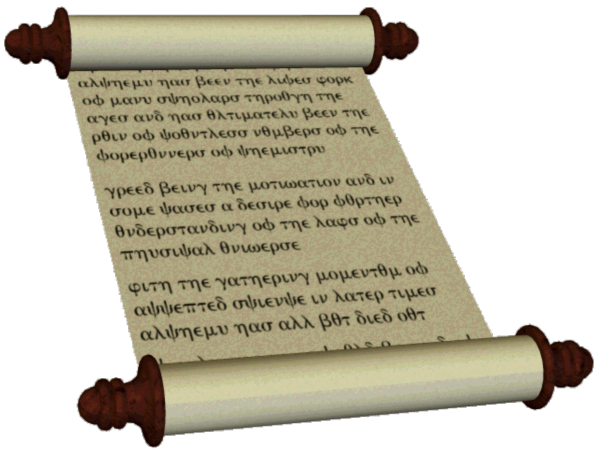
Data Parallelism

Prof. P. Felber
Pascal.Felber@unine.ch
http://iiun.unine.ch/
Institut d'informatique
Université de Neuchâtel

*Based on slides by **Maurice Herlihy** and **Nir Shavit***

"WordCount"

- Count then number of occurrences of words in a text



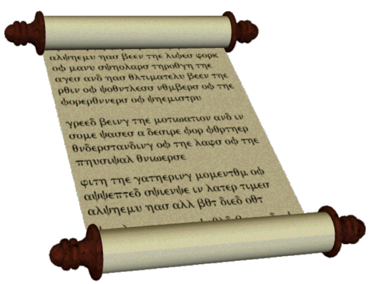
$$\begin{pmatrix} \text{alpha} \rightarrow 8 \\ \text{bravo} \rightarrow 3 \\ \text{charlie} \rightarrow 9 \\ \ldots \\ \text{zulu} \rightarrow 1 \end{pmatrix}$$

Easy to do sequentially...

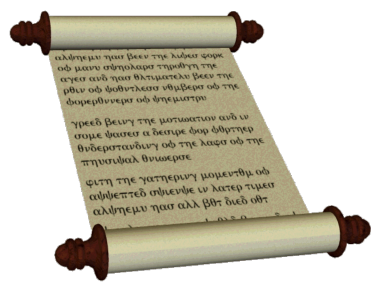
What about in parallel?

MapReduce

Split text among mapping threads

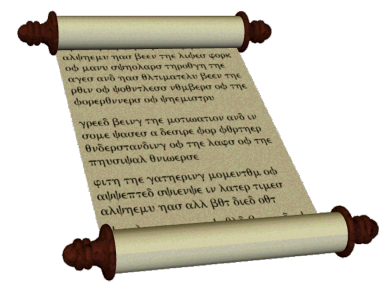


Chapter 1



Chapter 2

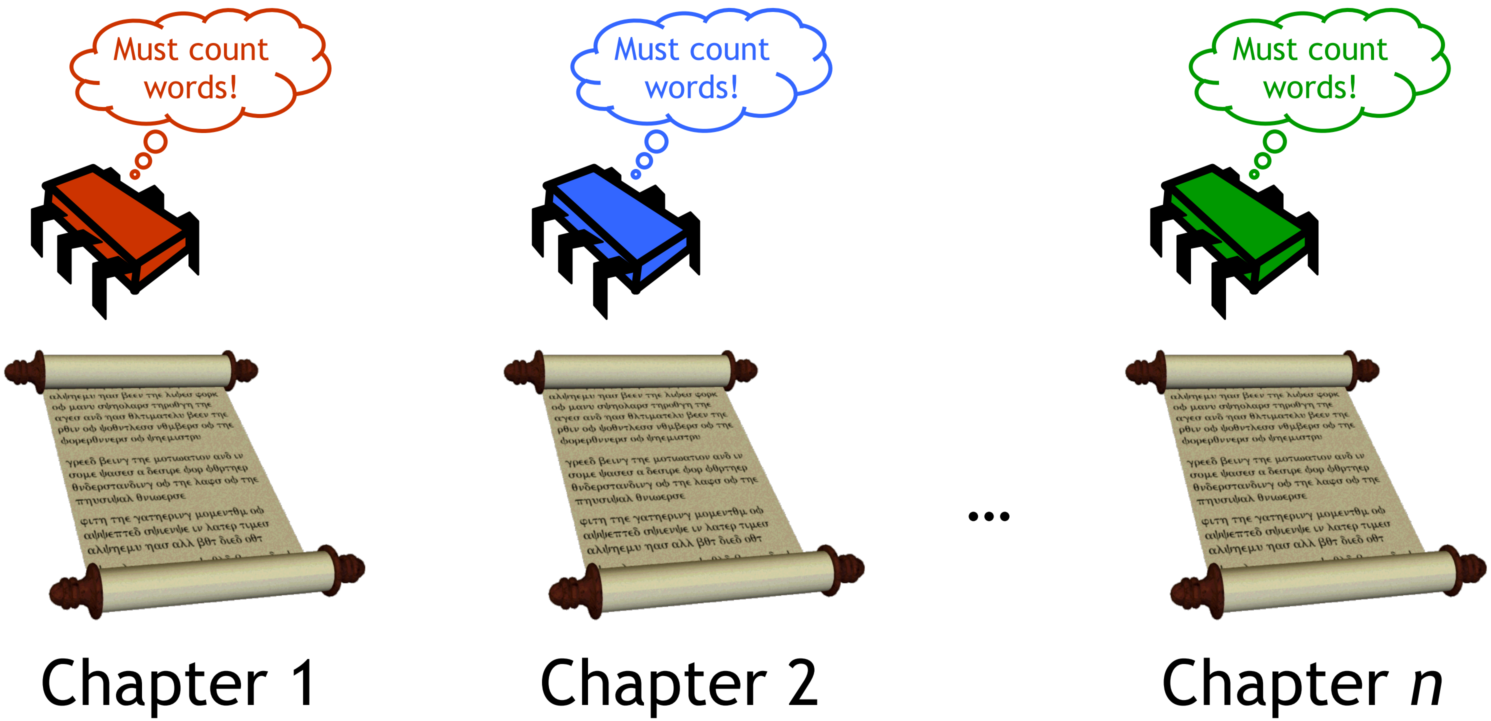
…



Chapter *n*

Map Phase

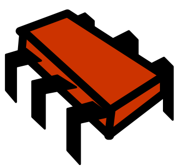
One mapping thread per chapter

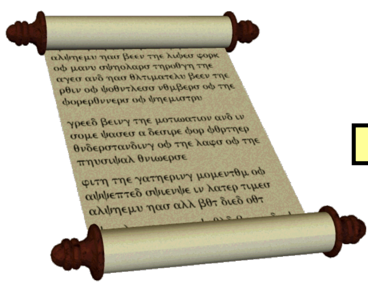


Map Phase

Each mapper produces a stream of key-value pairs

```
  key  : word
value  : local count
```





Chapter 1

$$\left(\begin{array}{l} \text{alpha} \rightarrow 3 \\ \text{juliet} \rightarrow 2 \\ \text{alpha} \rightarrow 1 \\ \qquad\quad \ldots \end{array} \right)$$

Mapper Class

Input: document fragment   Key: individual word   Value: local count

```
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

A task that runs in parallel with other tasks

Initialize input: which document fragment?

WordCount Mapper

Document fragment is list of words

Map each word…

…to its count in the fragment

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

Construct local word count

Create map to hold output

Examine each word in the document fragment

Increment that word's count in the map

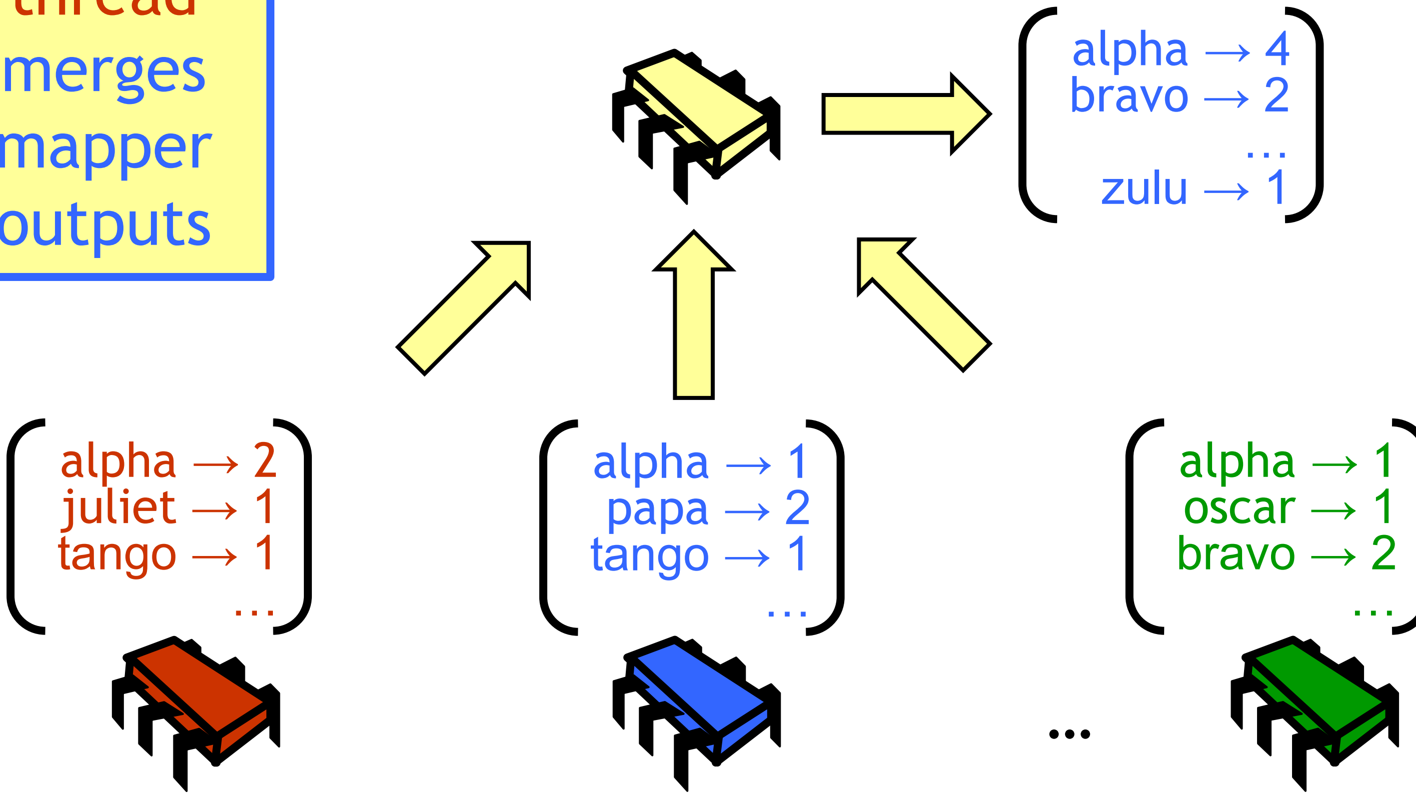
When the local count is complete, return the map

Reduce Phase

One reducer thread merges mapper outputs

The reducer produces a stream of key-value pairs

```
key   : word
value : word count
```



Reducer Class

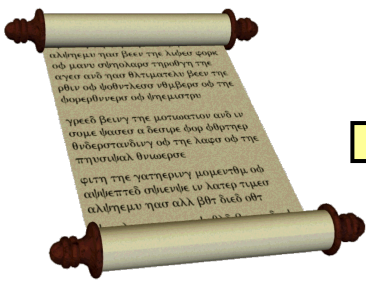
Each reducer is given a single key (word)…

```
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;
  }
}
```

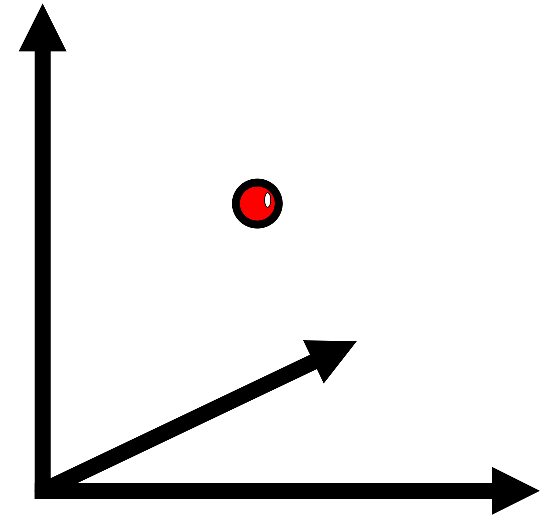
It produces a single summary value (total count for that word)

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

WordCount



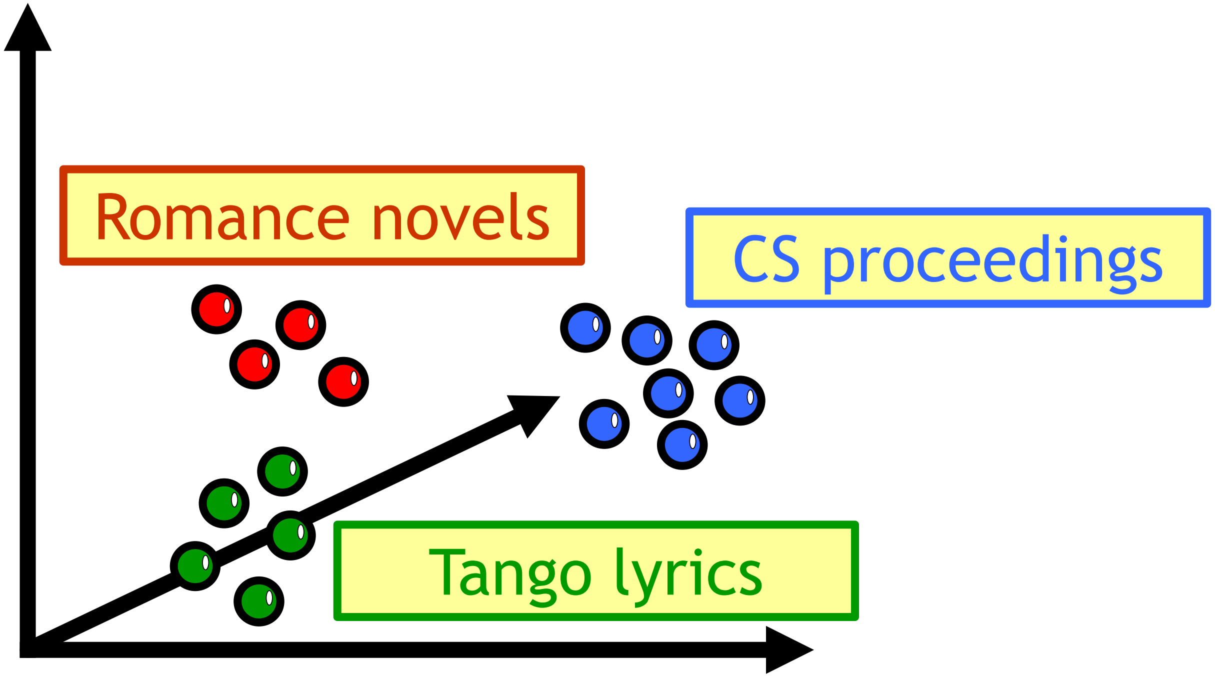
$$\begin{pmatrix} 0.037 \\ 0.002 \\ 0.045 \\ \ldots \\ 0.000 \end{pmatrix}$$



Normalizing document word count gives a fingerprint vector

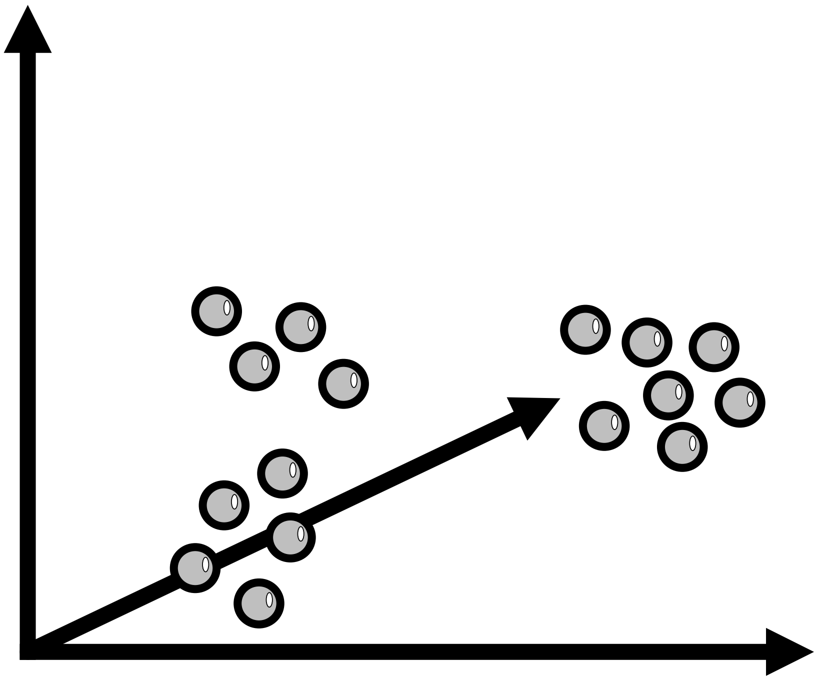
A fingerprint is a point in a high-dimensional space

Clustering



Similar documents have their fingerprints close to each other

K-Means

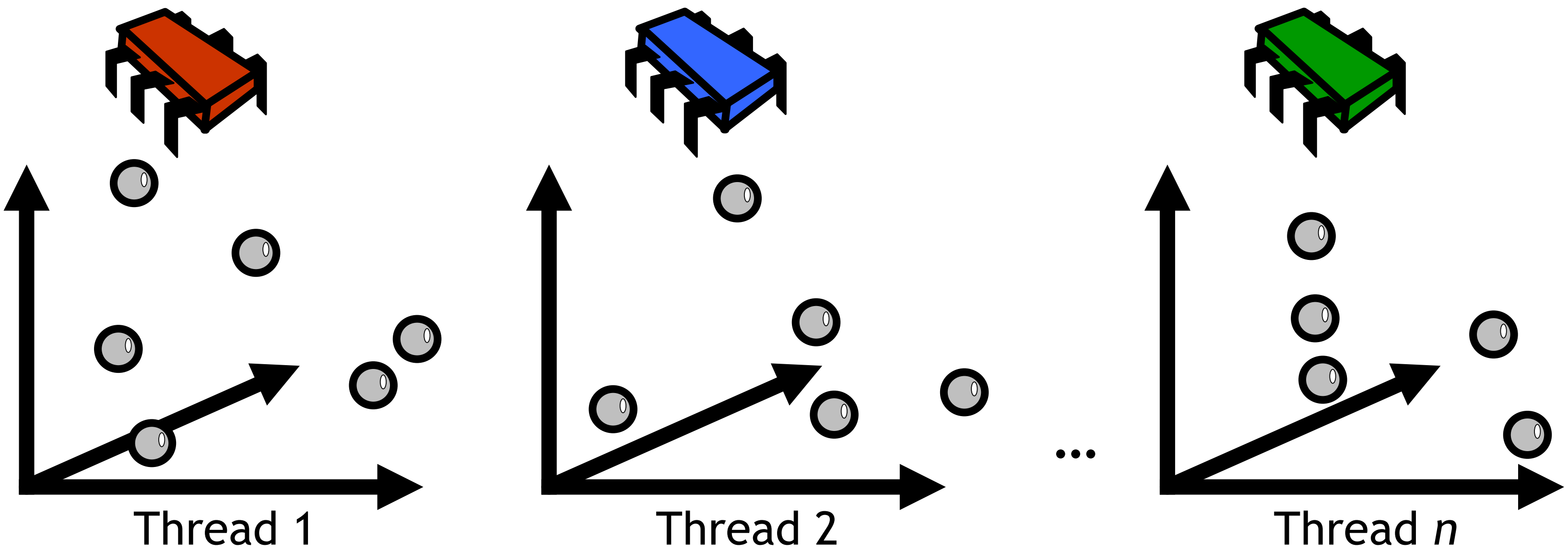


Find *k* clusters from raw data

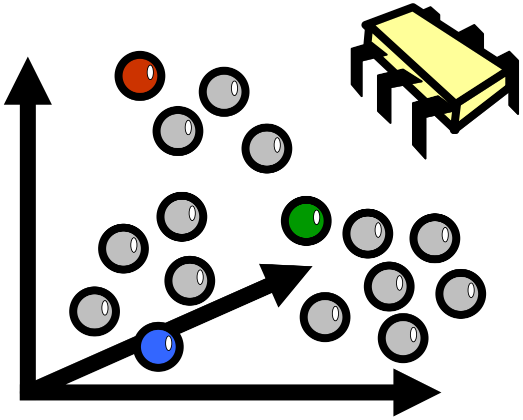
Each vector closer to those in same cluster than in different clusters!

MapReduce

Split points among mapping threads



MapReduce

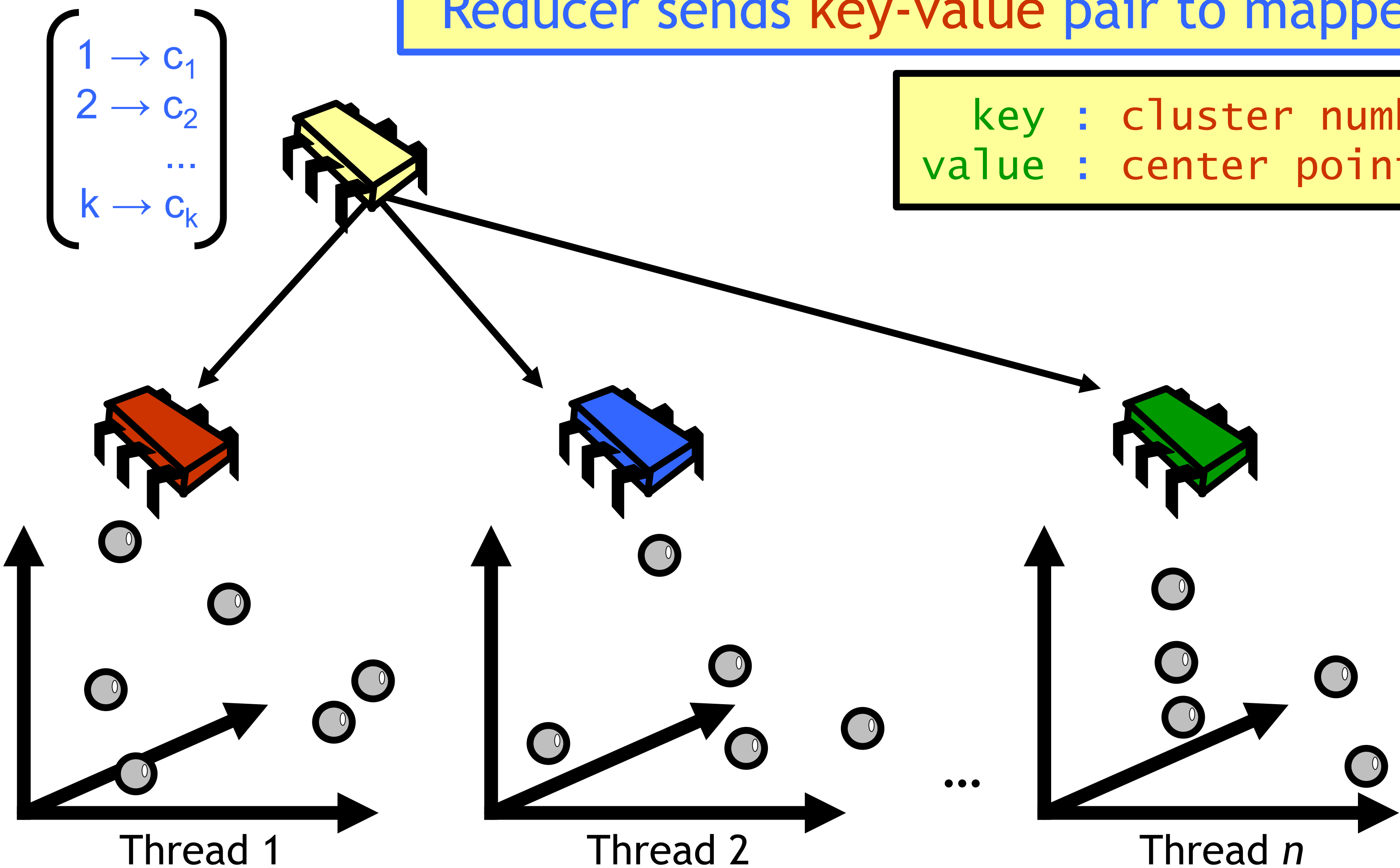


Reducer picks *k* "centers" at random

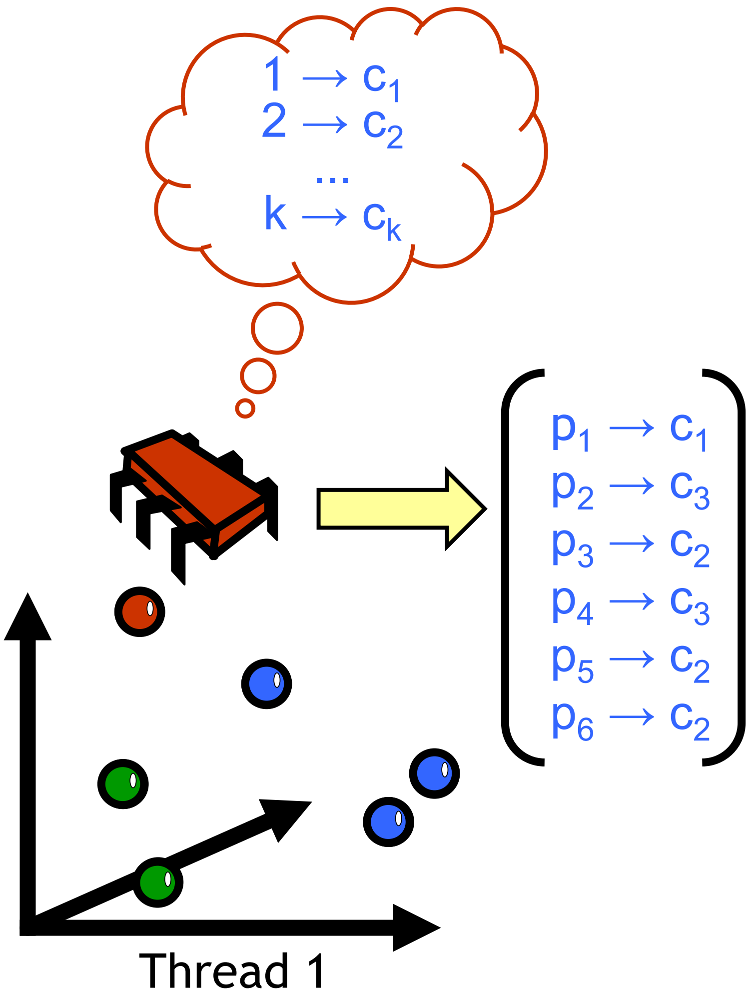
Reducer

Reducer sends key-value pair to mappers

```
  key : cluster number
value : center point
```



Mappers



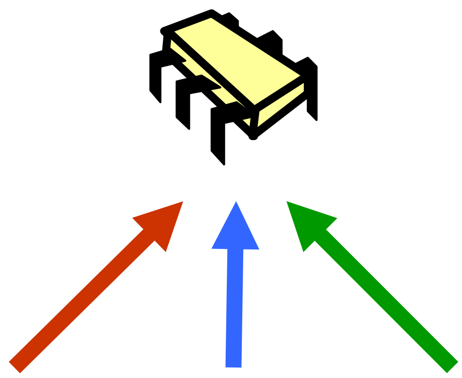
Each mapper uses centers to assign each vector to a cluster

Mapper sends **key-value** stream to reducer

```
  key : point
value : cluster number
```

Back at Reducer

$$\begin{pmatrix} C_1 = \{\ldots\} \\ C_2 = \{\ldots\} \\ C_3 = \{\ldots\} \\ \ldots \end{pmatrix}$$



$$\begin{pmatrix} 1 \rightarrow c_1' \\ 2 \rightarrow c_2' \\ 3 \rightarrow c_3' \\ \ldots \end{pmatrix}$$

The reducer merges the streams and assembles clusters

The reducer computes new centers based on new clusters

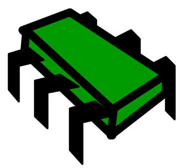
Once is not enough: reducer sends new centers to mappers

The process ends when centers become stable

To Recapitulate

- We saw two problems…
  - Word count
  - K-means
- …with similar solutions…
  - Map part is parallel
  - Reduce part is sequential
- …that can applied to many other problems

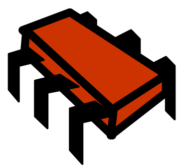
Map Function

(k_1, v_1) ⇒  ⇒ $\text{list}(k_2, v_2)$

`(doc, contents)` `(word, count)`

`(cluster#, center)` `(point, cluster#)`

Reduce Function

$(k_2, \text{list}(v_2))$ ⇒  ⇒ v_3

`(word, counts-list)`	`count`
`(cluster#, points-list)`	`new-cluster-center`

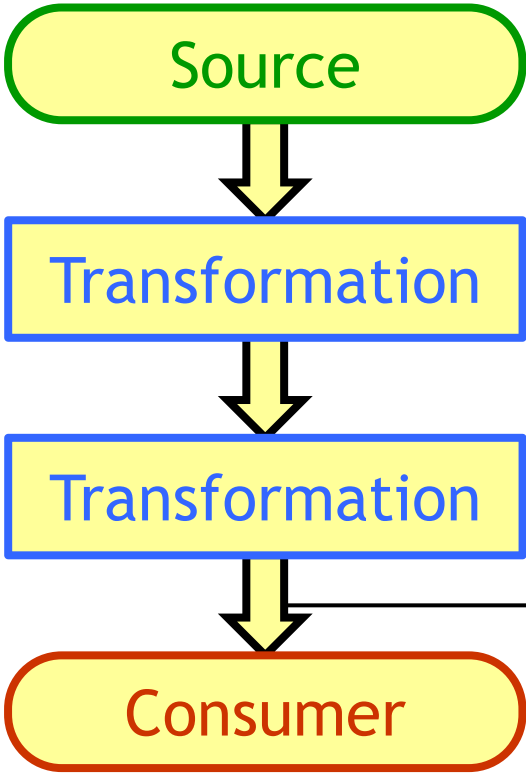
Examples

- Distributed grep
  - **Map:** line of document
  - **Reduce:** copy line to display
- URL access frequency
  - **Map:** (URL, local count)
  - **Reduce:** (URL, total count)
- Reverse Web link graph
  - **Map:** (target link, source page)
  - **Reduce:** (target link, list of source pages)
- Page rank, matrix multiplication, histogram…

Summary

- MapReduce is a generic solution for many problems
  - Map performs filtering and sorting
  - Reduce performs a summary operation
- Can be applied to different architectures
  - Distributed MapReduce on clusters
  - Multicore MapReduce with shared memory

Streams



Sequence of transformations (sometimes in parallel)

Transformations given by mathematical functions

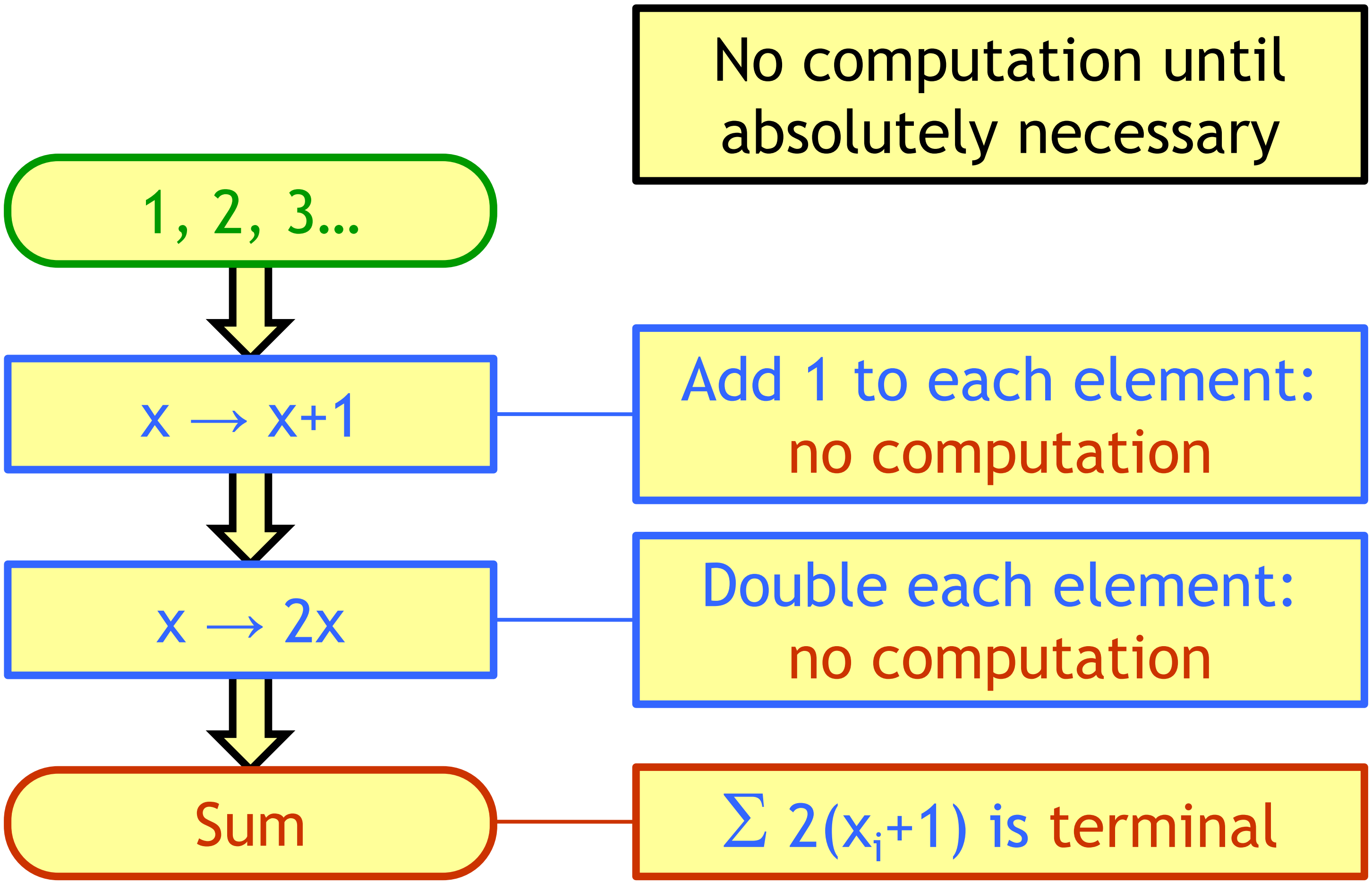
Transformations create new streams (no modifications or side effects)

Functional Programming

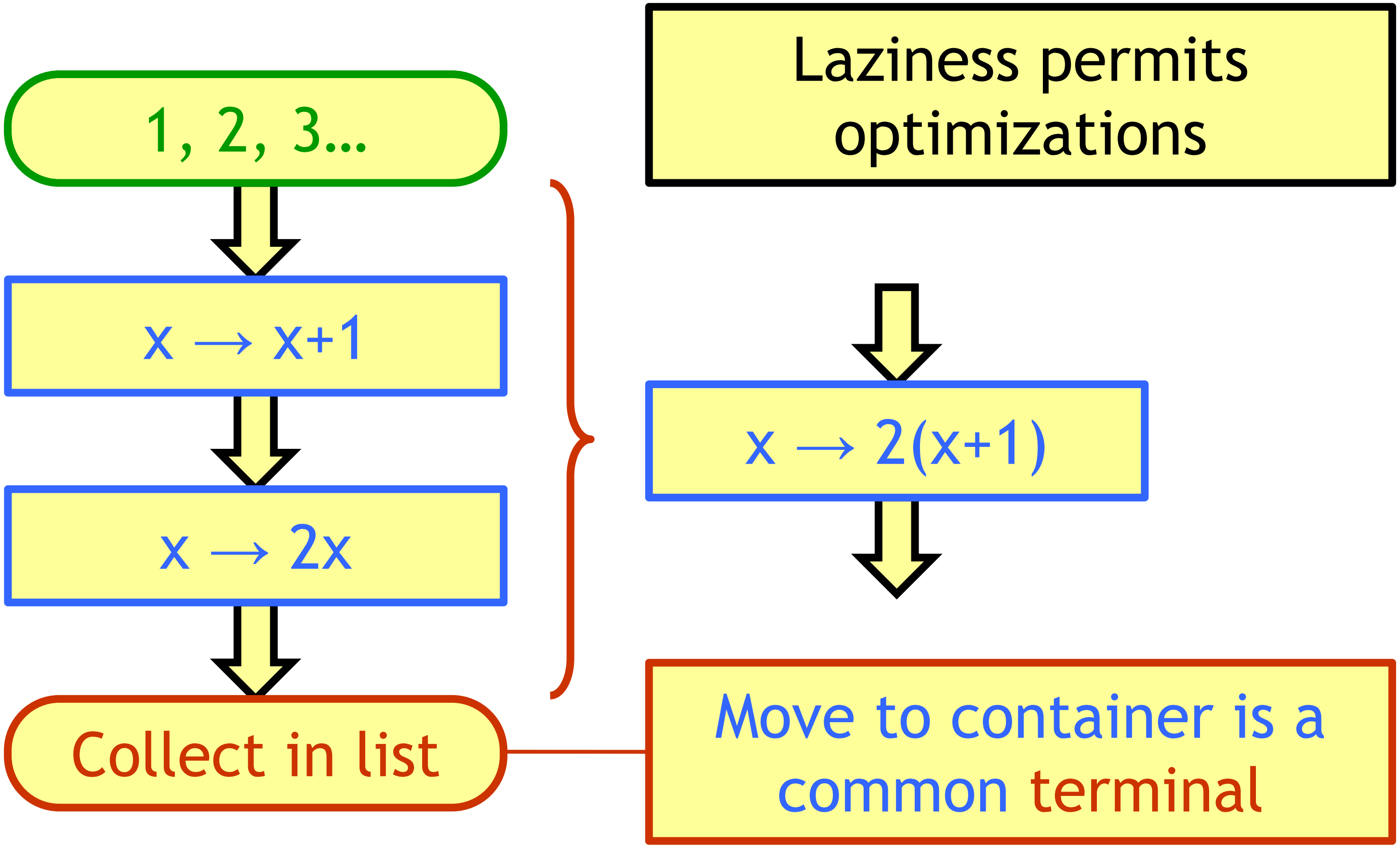
- Functions map old state to new state
  - Old state never changes
- No complex side effects
- Elegant, easier proof of correctness

- Isn’t it too inefficient to be practical?

Laziness


No computation until absolutely necessary
1, 2, 3…
$x \rightarrow x+1$
Add 1 to each element: no computation
$x \rightarrow 2x$
Double each element: no computation
Sum
$\sum 2(x_i+1)$ is terminal

Laziness


1, 2, 3…
x → x+1
x → 2x
Collect in list
Laziness permits optimizations
x → 2(x+1)
Move to container is a common terminal

Laziness

- Laziness permits infinite streams

```
Stream<Integer> fib = new FibStream();
```

1 1 2 3 5 8 13 21 34…

Unbounded Random Stream

Unbounded stream of double-precision random numbers

```
Stream<Double> randomDoubleStream() {
  return Stream.generate(
    () -> random.nextDouble()
  );
}
```

Stream that generates new elements on the fly

Function to call when generating new element: Java lambda expression (anonymous method)

No loops
No conditionals
No mutable objects

WordCount

Put each word from the document into a list

```
List<String> readFile(String fileName) {
    …
    return reader
      .lines()
      .map(String::toLowerCase)
      .flatMap(s -> pattern.splitAsStream(s))
      .collect(Collectors.toList());
}
```

Open the file, create a `FileReader`

Turn the `FileReader` into a stream of lines, each line a string

(1)

(2)

(3)

(1) `map` creates a new stream by applying a function to each stream element (here: convert to lower case)

(2) `flatMap` replaces one stream element with multiple stream elements (here: split line into words)

(3) `collect` (terminal operation) puts stream elements in a container (here: in a list)

How a stream program looks: each line creates a new stream

No loops
No conditionals
No mutable objects

WordCount

We have a list, now let's count words!

```
Map<String,Long> map = text
    .stream()
    .collect(
      Collectors.groupingBy(
       Function.identity(),
       Collectors.counting()));
```

Start with list of words

Turn list into a stream

Put stream into a container (Map): word → count

Each element's key is that element

Each element's value is the number of times it appears

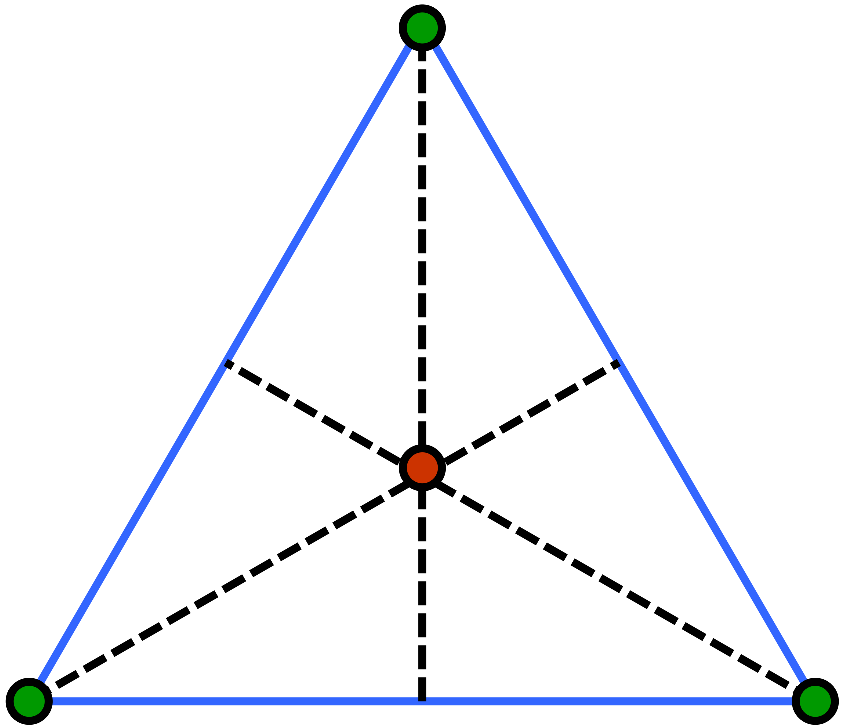
K-Means

```
class Point {
  Point(double x, double y) {…}
  Point plus(Point other) {…}
  Point scale(double x) {…}
  static Point barycenter(
    List<Point> cluster
    ) {…}
}
```

Stream-based!

Barycenter

The barycenter of a set of points is their center of mass



K-Means Barycenter

```
Point barycenter(List<Point> cluster) {
  double numPoints = cluster.size();
  Optional<Point> sum = cluster
                        .stream()
                        .reduce(Point::plus);
  return sum.get()
            .scale(1 / numPoints);
}
```

Cluster size

Turn list into stream

Reduce

```
stream.reduce(+):
```

Reduce is a terminal operation

$() \longrightarrow \varnothing$

$(a) \longrightarrow a$

$(a,b) \longrightarrow a+b$

$(a,b,c) \longrightarrow (a+b)+c$

etc.

K-Means Barycenter

```
Point barycenter(List<Point> cluster) {
  double numPoints = cluster.size();
  Optional<Point> sum = cluster
                      .stream()
                      .reduce(Point::plus);
  return sum.get()
            .scale(1 / numPoints);
}
```

Sum points in cluster

Optional because sum might be empty!

Extract sum and divide by number of points

K-Means

Read points from file

```
List<Point> points = readFile("cluster.dat");
centers = randomDistinctCenters(points);
double convergence = 1.0;
while (convergence > EPSILON) {
  Map<Integer, List<Point>> clusters
    = points.stream()
          .collect(Collectors.groupingBy(
            p -> closestCenter(centers, p)));
  …
}
```

Pick random centers

Keep going until centers are stable

Turn list of points into stream

Put each point in a map: key is closest center, value is list of points with that center

K-Means

```
while (convergence > EPSILON) {
  …
  Map<Integer, Point> newCenters = clusters
    .entrySet()
    .stream()
    .collect(
      Collectors.toMap(
        e -> e.getKey(),
        e -> Point.barycenter(e.getValue())
      ));
  convergence = distance(centers, newCenters);
  centers = newCenters;
}
```

Compute the new center (map: cluster# → center)

Turn map into a stream of pairs: (cluster#, point)

Turn stream into a map: cluster# → barycenter

New key is still cluster number

New value is the barycenter computed earlier

If centers have moved, start again with the new centers

Functional K-Means

- Many fewer lines of code
- Easier to read (really!)
- Easier to reason
- Easier to optimize

Parallelism?

So far streams are sequential

```
Arrays.asList("Arlington",
              "Berkeley",
              "Clarendon",
              "Dartmouth",
              "Exeter")
    .stream()
    .forEach(s -> printf("%s\n", s));
```

Make list of strings and turn them into a stream

`forEach` applies a method to each element (not functional)

Output

```
Arlington
Berkeley
Clarendon
Dartmouth
Exeter
```

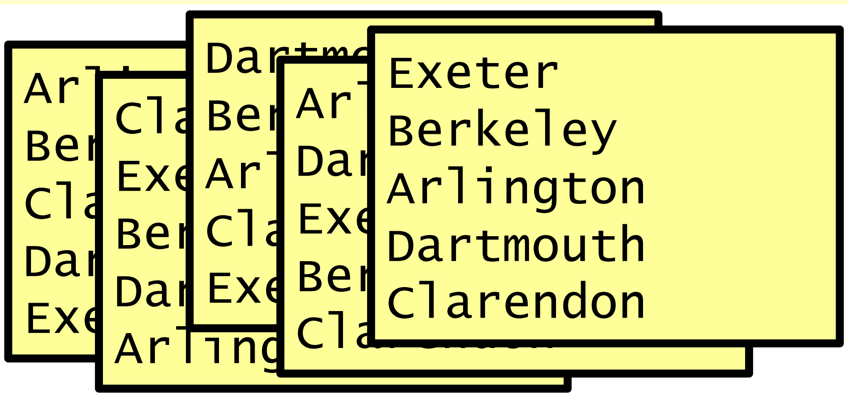
Parallel Streams

We can use parallel streams

```
Arrays.asList("Arlington",
              "Berkeley",
              "Clarendon",
              "Dartmouth",
              "Exeter")
    .parallelStream()
    .forEach(s -> printf("%s\n", s));
```

Turn list into parallel stream

Output



…

Parallel Streams

A sequential stream can be made parallel

```
Arrays.asList("Arlington",
              "Berkeley",
              "Clarendon",
              "Dartmouth",
              "Exeter")
    .stream()
    .parallel()
    .forEach(s -> printf("%s\n", s));
```

Can turn stream into a parallel stream

Pitfalls

```
list.stream().forEach(
   s -> list.add(0)
);
```

Lambda (function) must not modify source!

```
source.parallelStream()
   .forEach(
      s -> target.add(s));
```

Exception if target not thread-safe

Order added is non-deterministic

Summary

- Streams provide several benefits
  - Functional programming
  - Data parallelism
  - Compiler optimizations