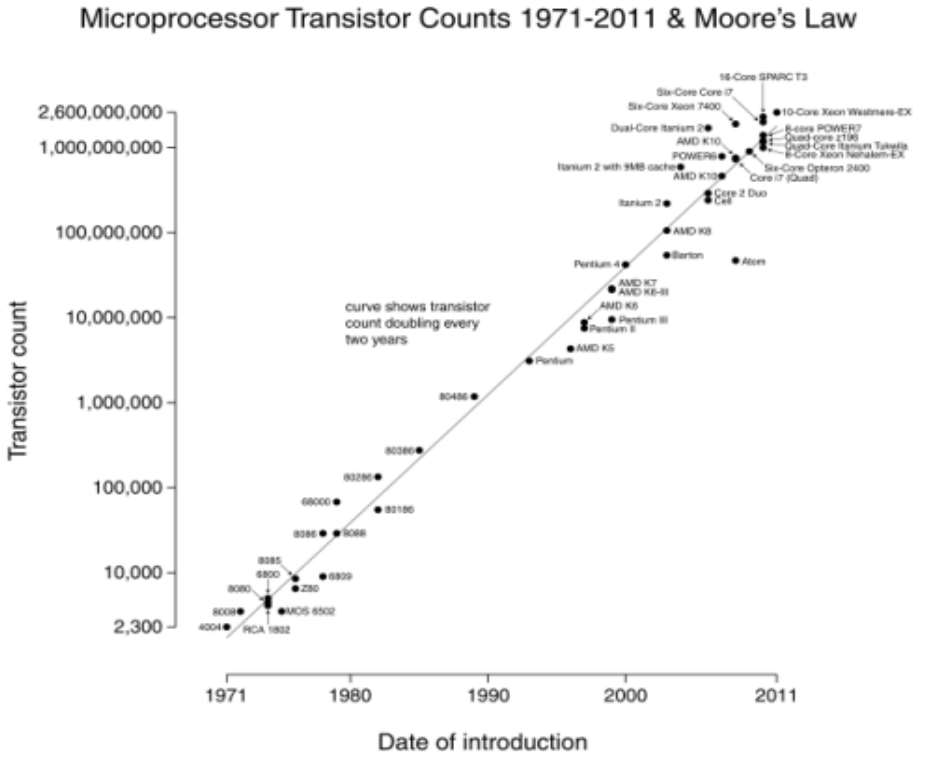
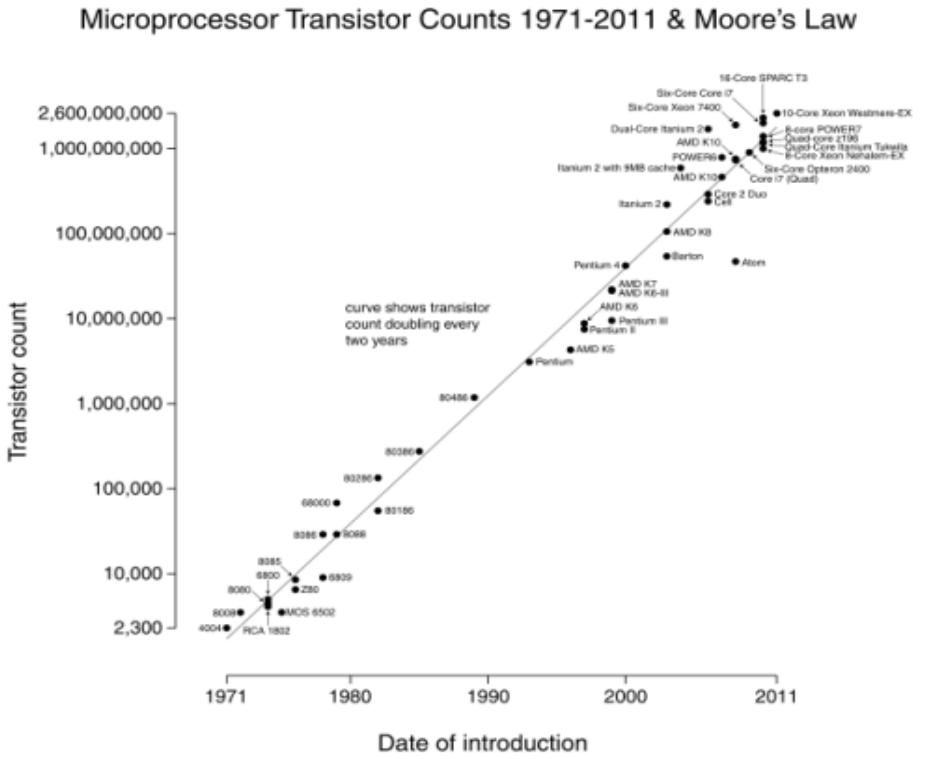
The Moore's law

The Moore's law



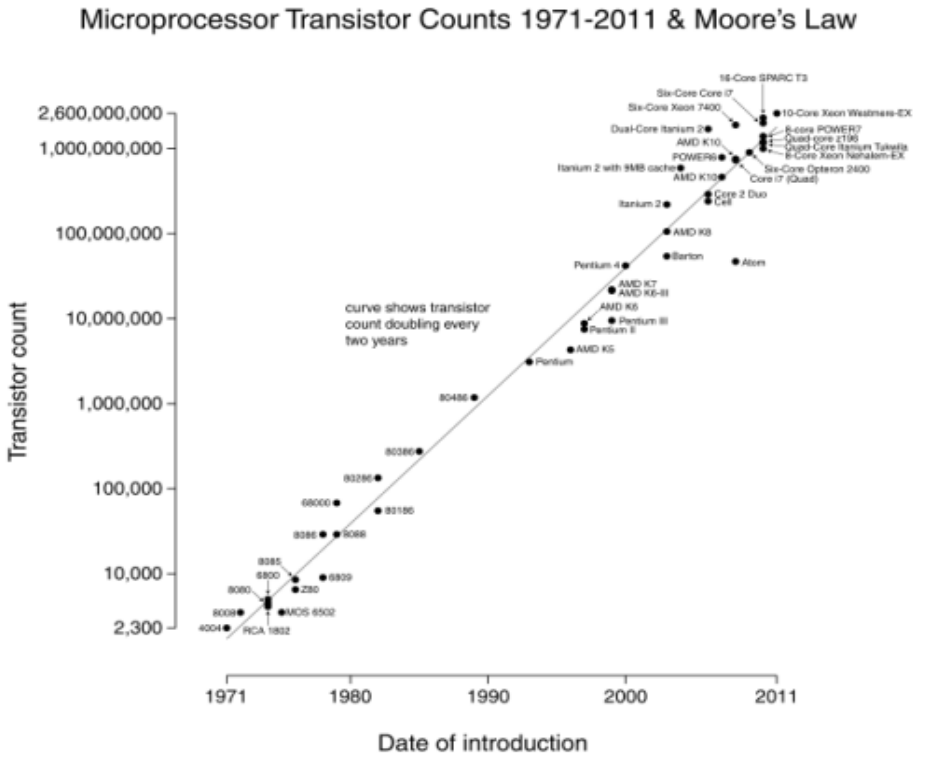
The Moore's law

- The number of transistors on a chip doubles approximately every two years



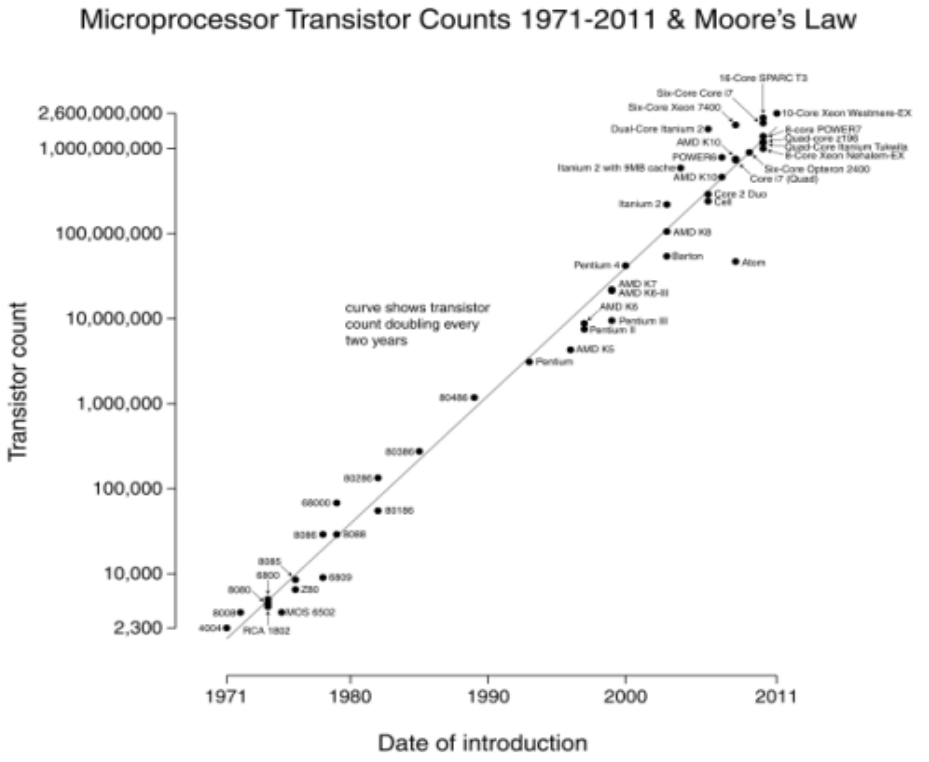
The Moore's law

- The number of transistors on a chip doubles approximately every two years
- Need more performance? Wait two years.



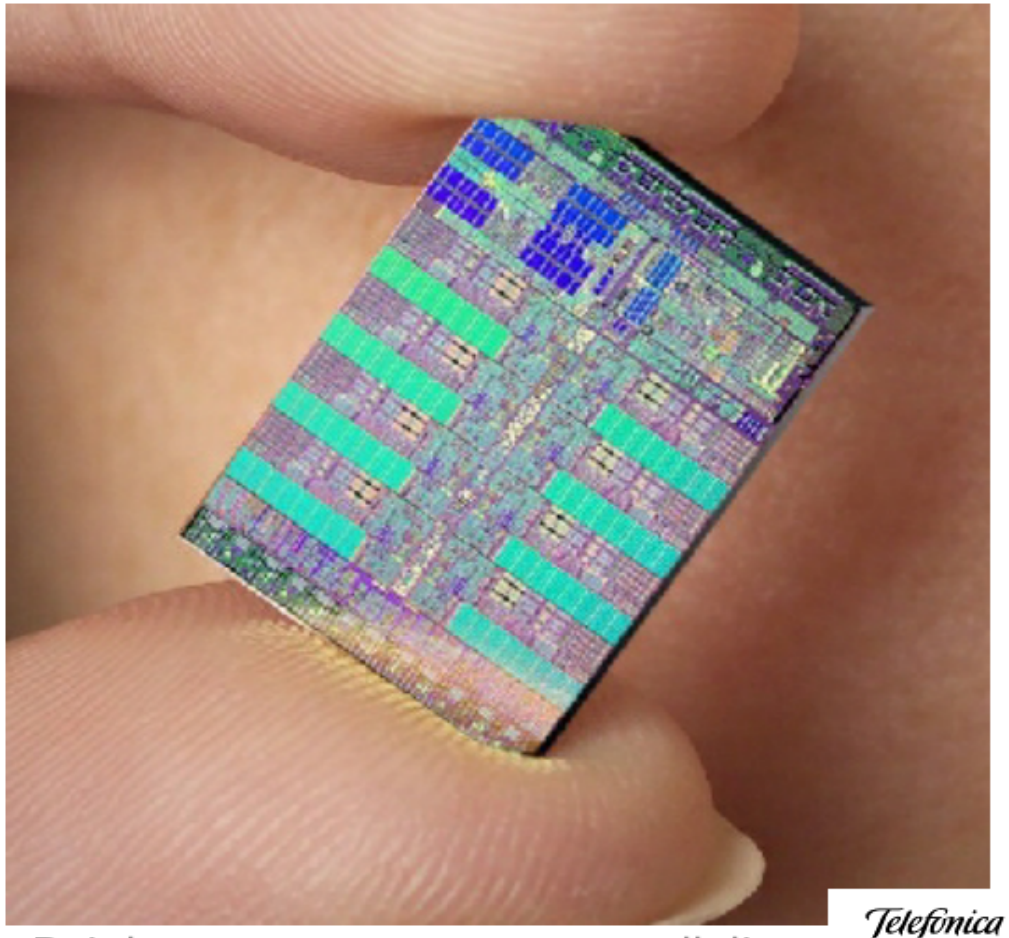
The Moore's law

- The number of transistors on a chip doubles approximately every two years
- Need more performance? Wait two years.
- Not anymore! We don't get faster processors, we get more of them



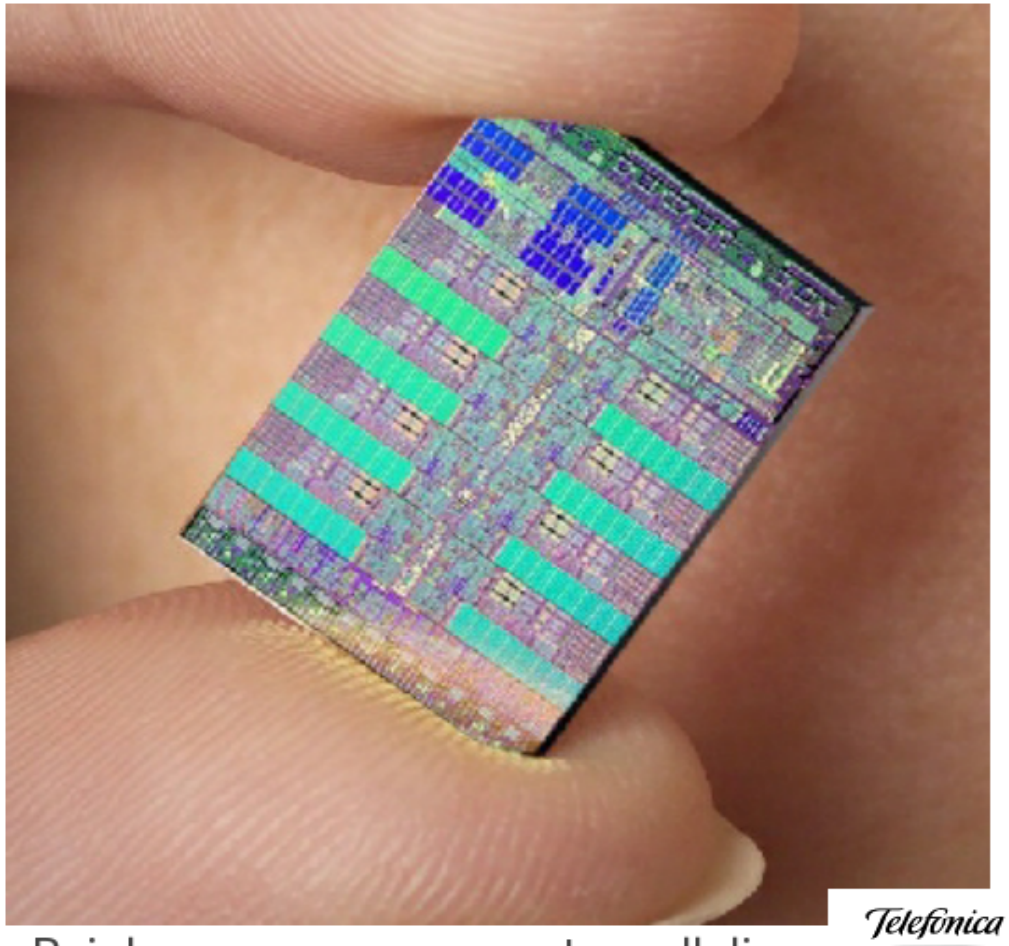
The Moore's law

- The number of transistors on a chip doubles approximately every two years
- Need more performance? Wait two years.
- Not anymore! We don't get faster processors, we get more of them



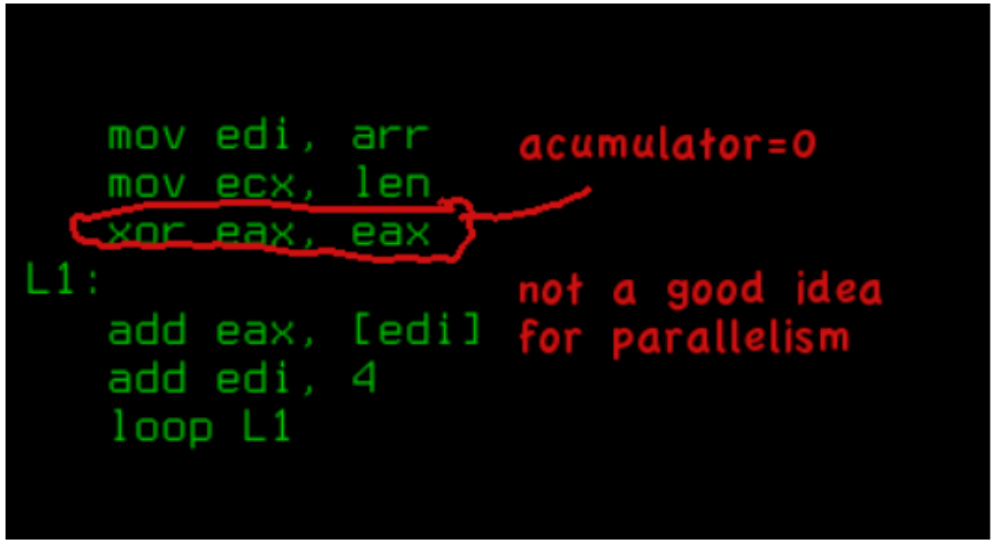
The Moore's law

- The number of transistors on a chip doubles approximately every two years
- Need more performance? Wait two years.
- Not anymore! We don't get faster processors, we get more of them
- But we have spent decades optimizing for one core (or few)



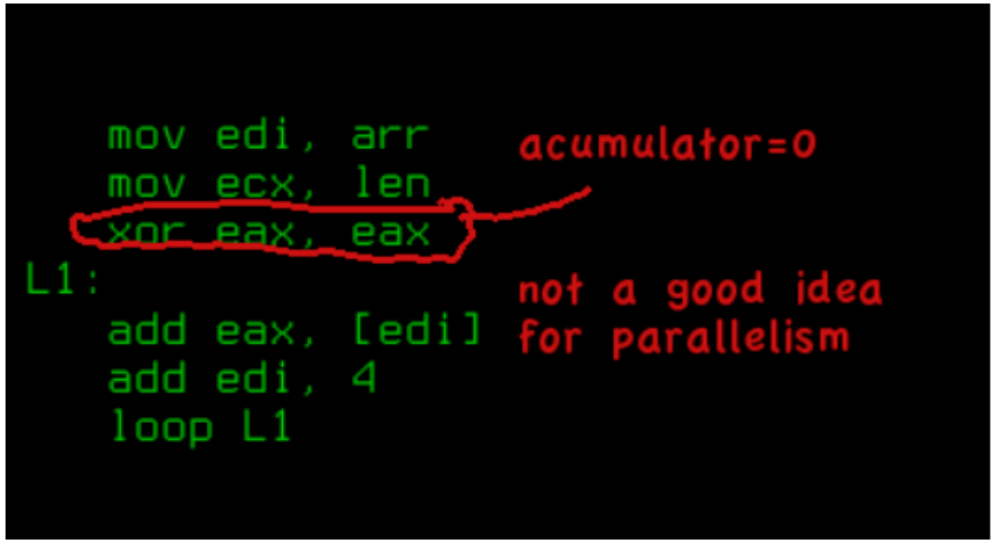
The Moore's law

- The number of transistors on a chip doubles approximately every two years
- Need more performance? Wait two years.
- Not anymore! We don't get faster processors, we get more of them
- But we have spent decades optimizing for one core (or few)



The Moore's law

- The number of transistors on a chip doubles approximately every two years
- Need more performance? Wait two years.
- Not anymore! We don't get faster processors, we get more of them
- But we have spent decades optimizing for one core (or few)
- What should we do?



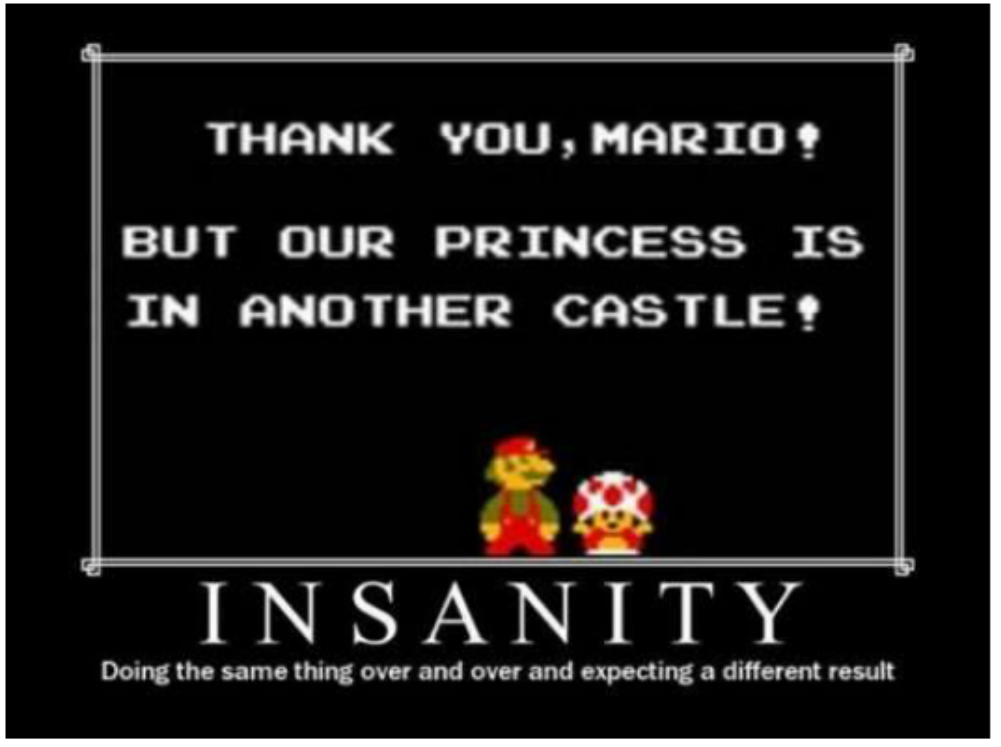
My two cents

My two cents


THANK YOU, MARIO!
BUT OUR PRINCESS IS
IN ANOTHER CASTLE!
INSANITY
Doing the same thing over and over and expecting a different result

My two cents

Referential transparency



My two cents

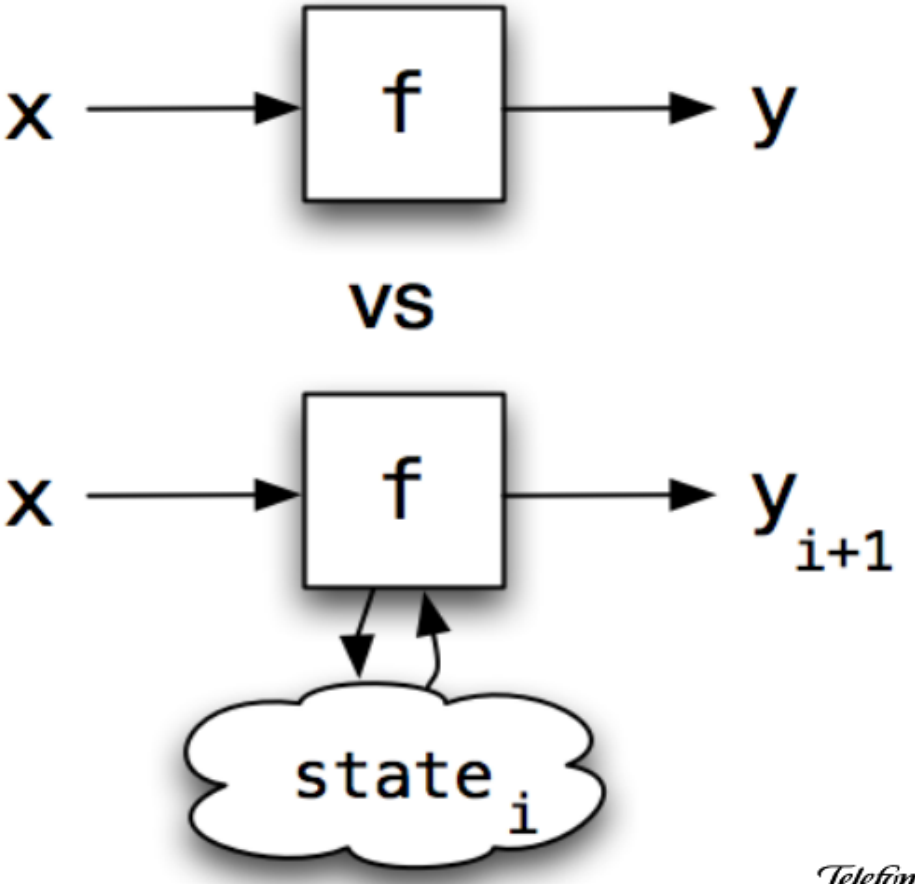
Referential transparency

- Transparent functions only depend on their inputs

My two cents

Referential transparency

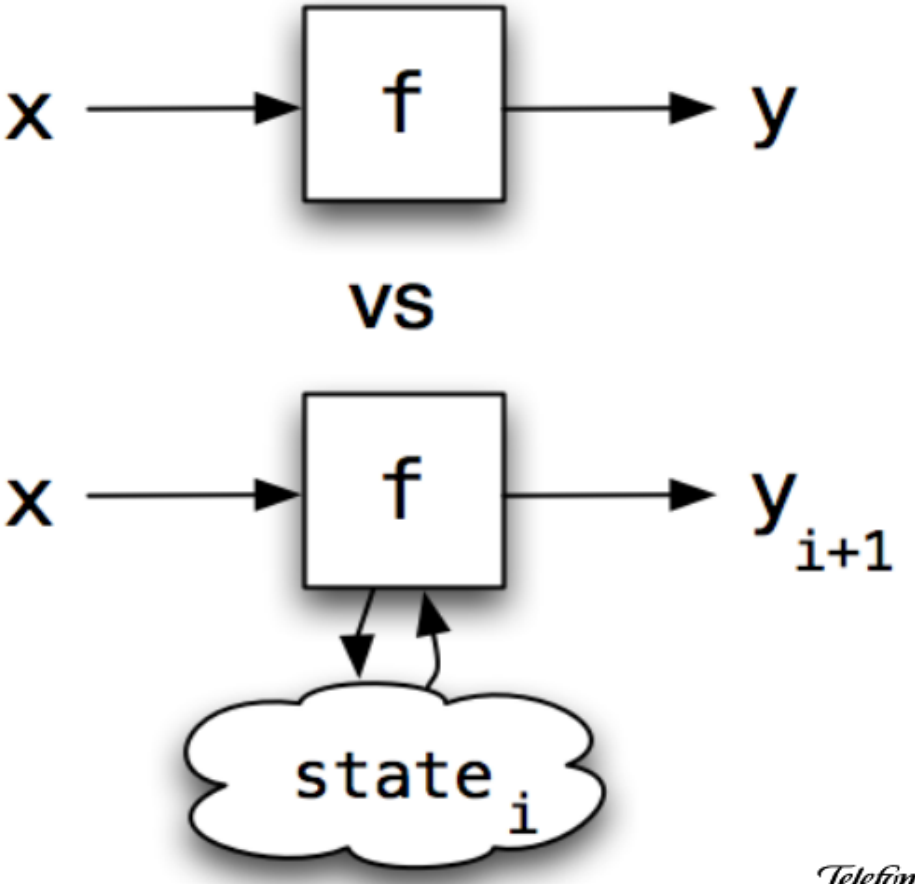
- Transparent functions only depend on their inputs



My two cents

Referential transparency

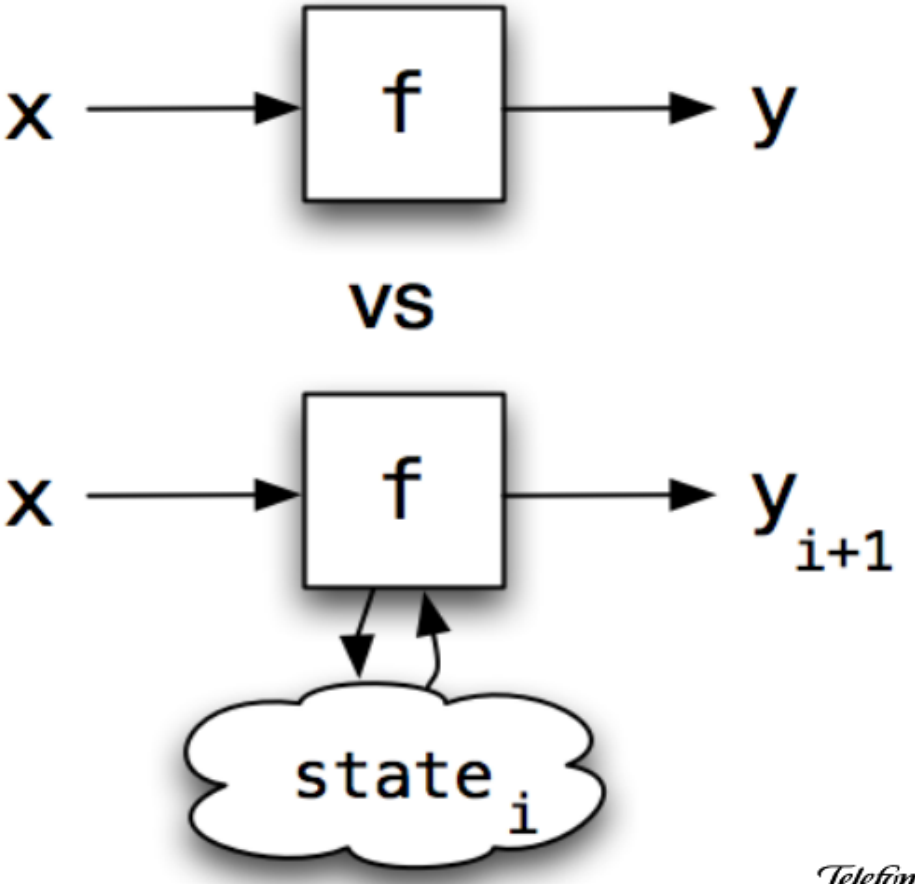
- Transparent functions only depend on their inputs
  - Not on any internal state



My two cents

Referential transparency

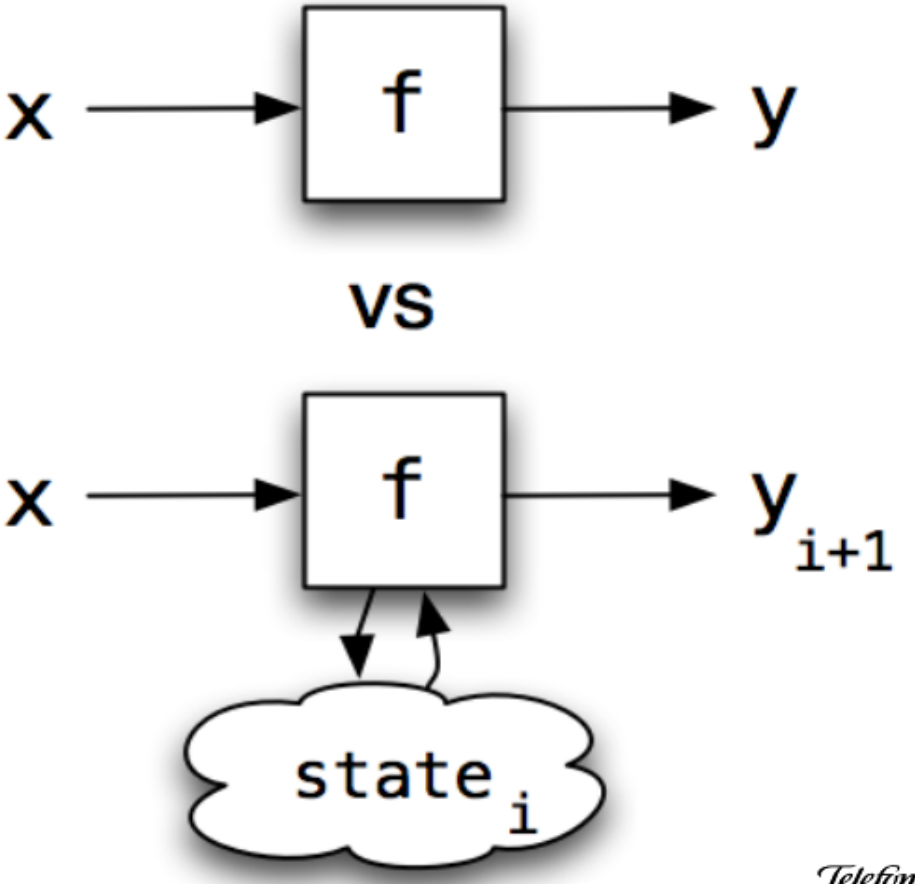
- Transparent functions only depend on their inputs
  - Not on any internal state
  - Not on when you call it



My two cents

Referential transparency

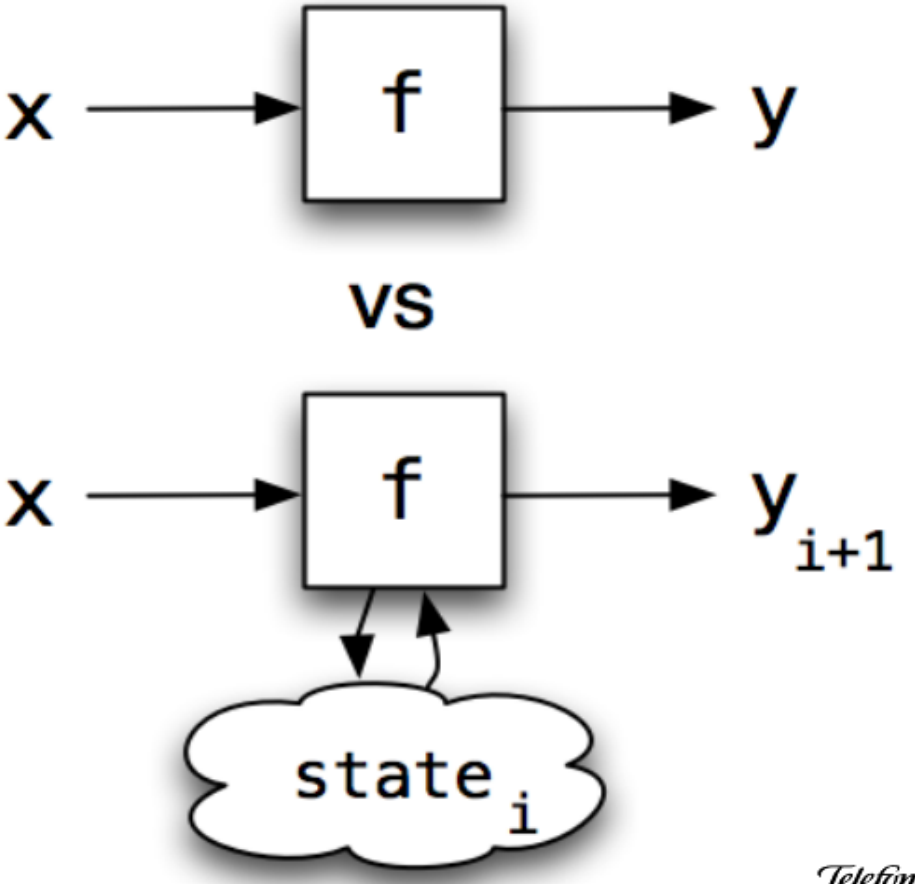
- Transparent functions only depend on their inputs
  - Not on any internal state
  - Not on when you call it
  - Not on the order in which you call it



My two cents

Referential transparency

- Transparent functions only depend on their inputs
  - Not on any internal state
  - Not on when you call it
  - Not on the order in which you call it
- Explicit data dependencies

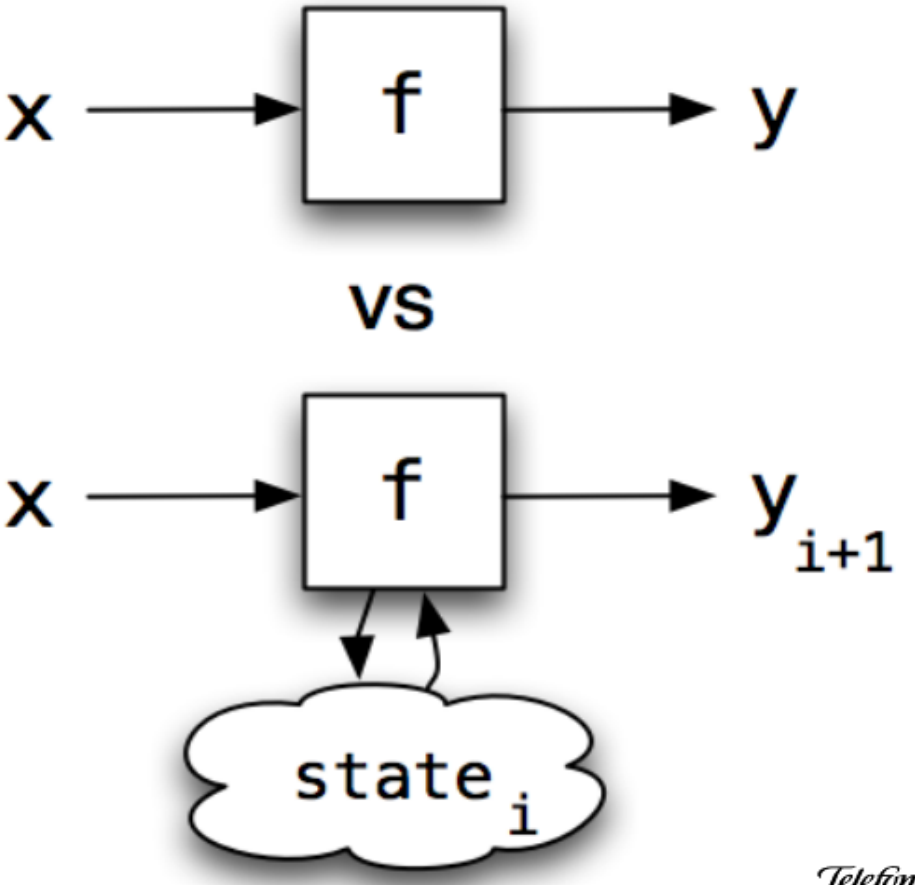


My two cents

Referential transparency

- Transparent functions only depend on their inputs
  - Not on any internal state
  - Not on when you call it
  - Not on the order in which you call it
- Explicit data dependencies

Be declarative



My two cents

Referential transparency

- Transparent functions only depend on their inputs
  - Not on any internal state
  - Not on when you call it
  - Not on the order in which you call it
- Explicit data dependencies

Be declarative

My two cents

Referential transparency

- Transparent functions only depend on their inputs
  - Not on any internal state
  - Not on when you call it
  - Not on the order in which you call it
- Explicit data dependencies

Be declarative

- What and not how

My two cents

Referential transparency

- Transparent functions only depend on their inputs
  - Not on any internal state
  - Not on when you call it
  - Not on the order in which you call it
- Explicit data dependencies

Be declarative

- What and not how
- Logic over flow control

My two cents

Referential transparency

- Transparent functions only depend on their inputs
  - Not on any internal state
  - Not on when you call it
  - Not on the order in which you call it
- Explicit data dependencies

Be declarative

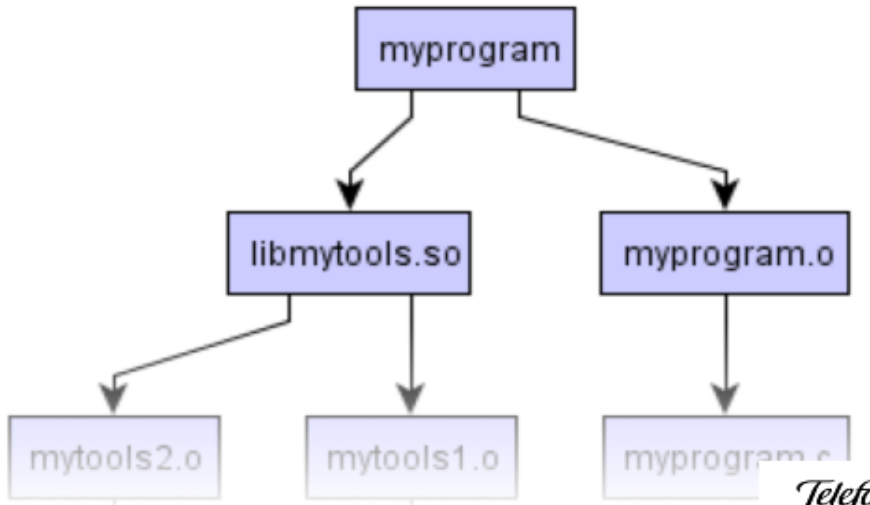
- What and not how
- Logic over flow control

```
myprogram: myprogram.o libmytools.so
    gcc -L. -lmylib -o $@ $^

myprogram.o: myprogram.c
    gcc -c -o $@ $<

libmytools.so: mytools1.o mytools2.o
    gcc -shared -o $@ $^

...
```



Painless concurrency and parallelism

My two cents

Referential transparency

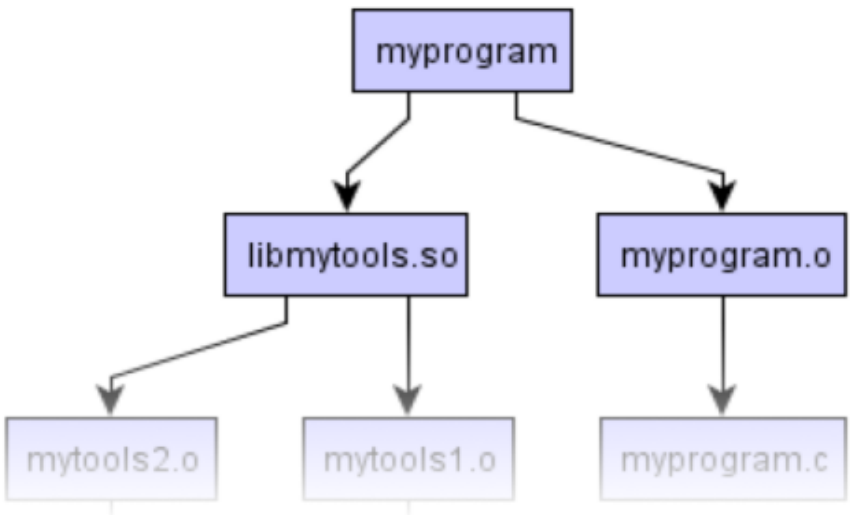
- Transparent functions only depend on their inputs
  - Not on any internal state
  - Not on when you call it
  - Not on the order in which you call it
- Explicit data dependencies

```
myprogram: myprogram.o libmytools.so
    gcc -L. -lmylib -o $@ $^

myprogram.o: myprogram.c
    gcc -c -o $@ $<

libmytools.so: mytools1.o mytools2.o
    gcc -shared -o $@ $^

...
```



Be declarative

- What and not how
- Logic over flow control
- Make room for reordering, parallelization and distribution

Streams

Streams

head rest

can be computed
on demand

Streams

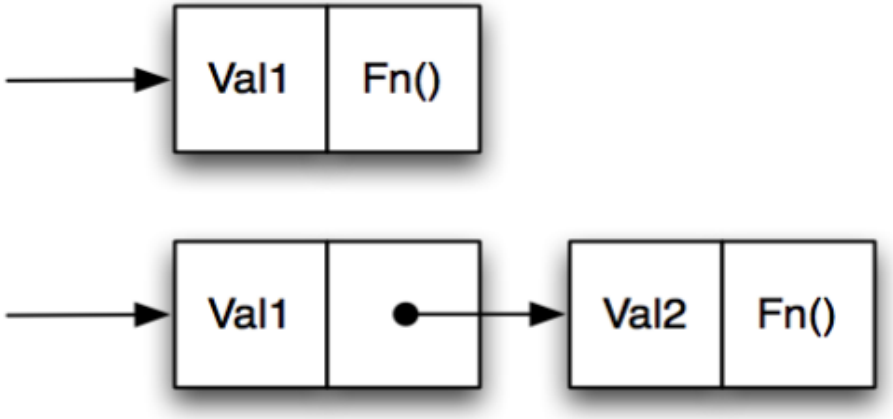
- Streams (or lazy sequences) abstracts from looping

head rest

can be computed
on demand

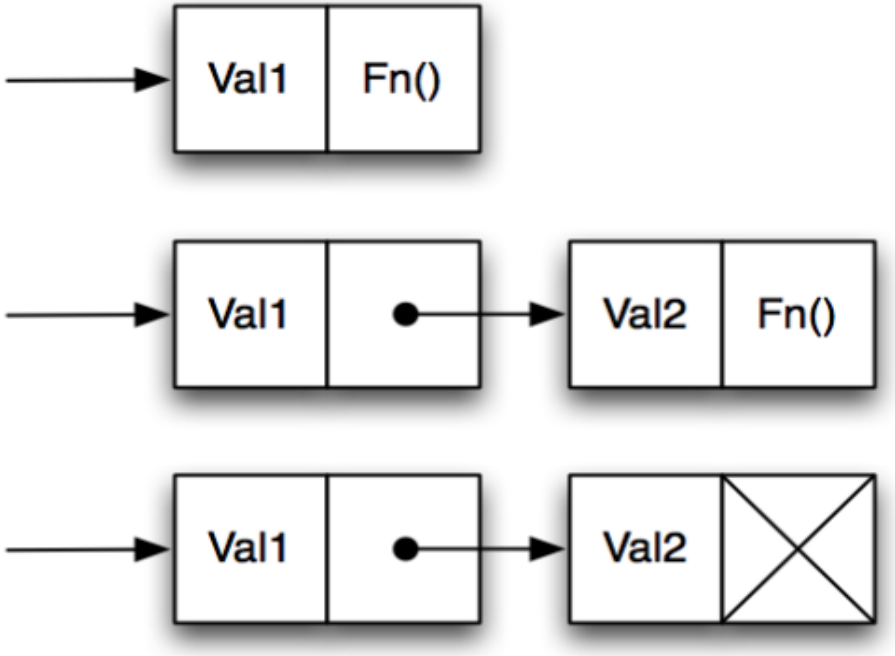
Streams

- Streams (or lazy sequences) abstracts from looping



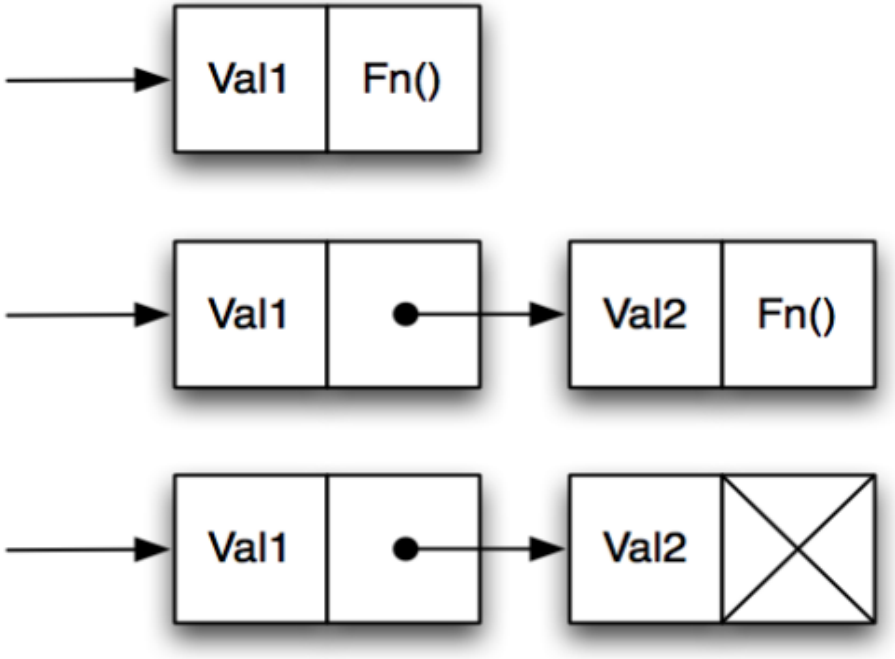
Streams

- Streams (or lazy sequences) abstracts from looping



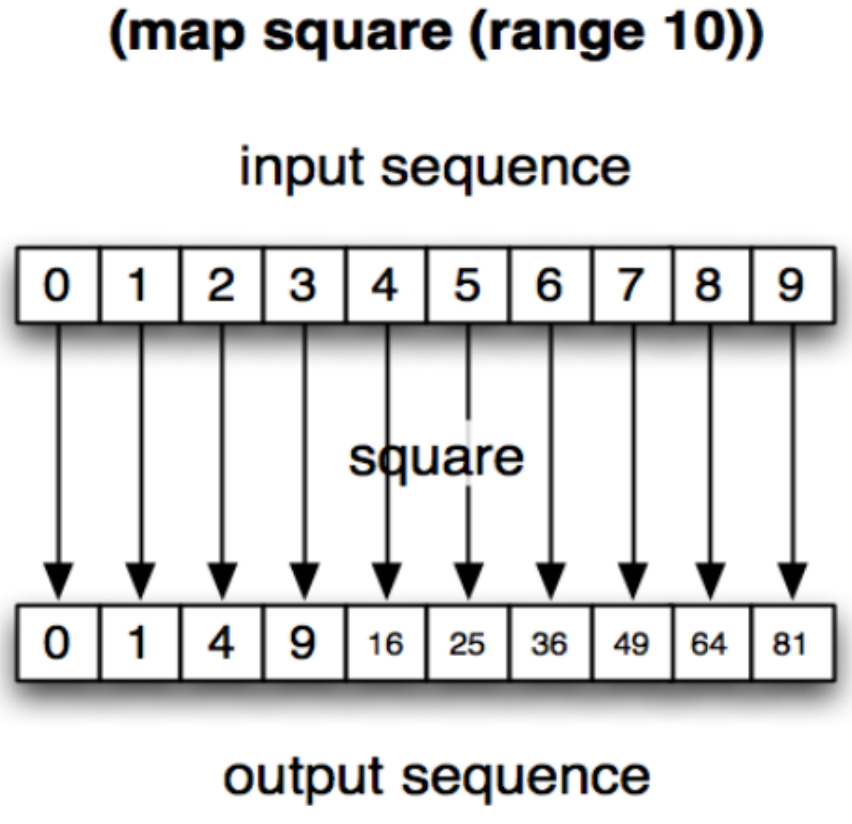
Streams

- Streams (or lazy sequences) abstracts from looping
- Stream operators are more declarative than loops



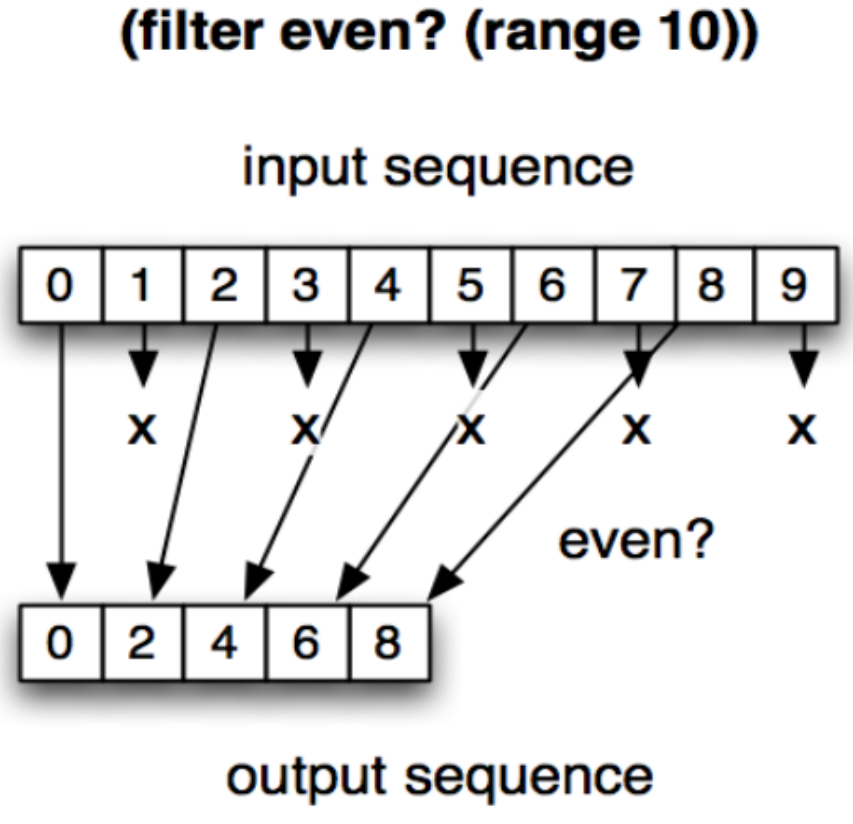
Streams

- Streams (or lazy sequences) abstracts from looping
- Stream operators are more declarative than loops
  - Map



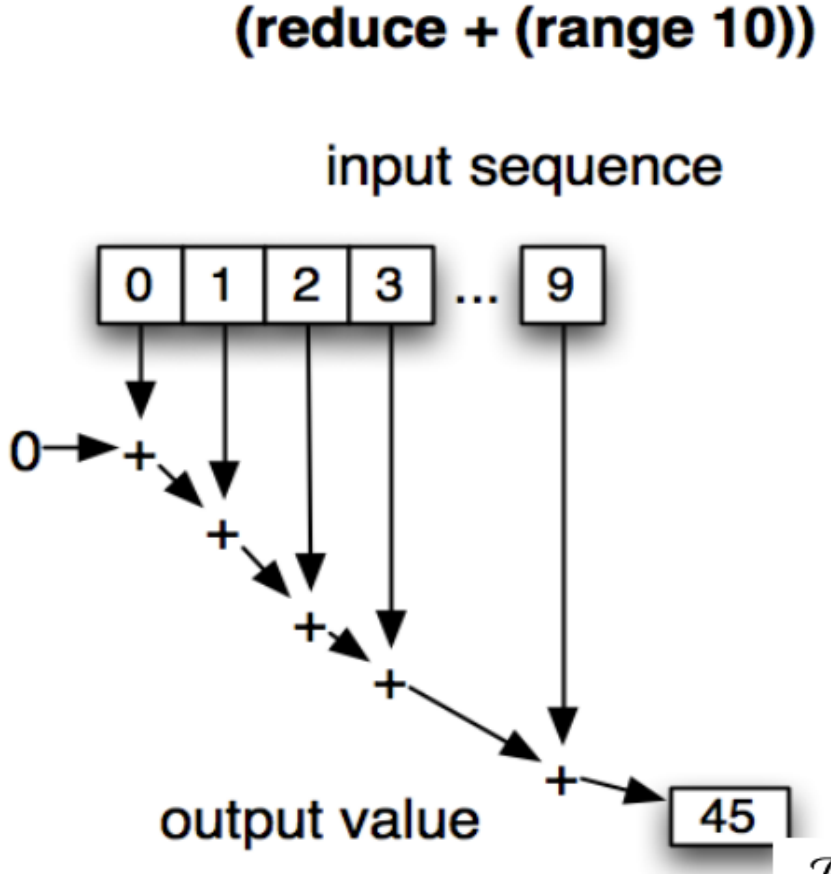
Streams

- Streams (or lazy sequences) abstracts from looping
- Stream operators are more declarative than loops
  - Map
  - Filter



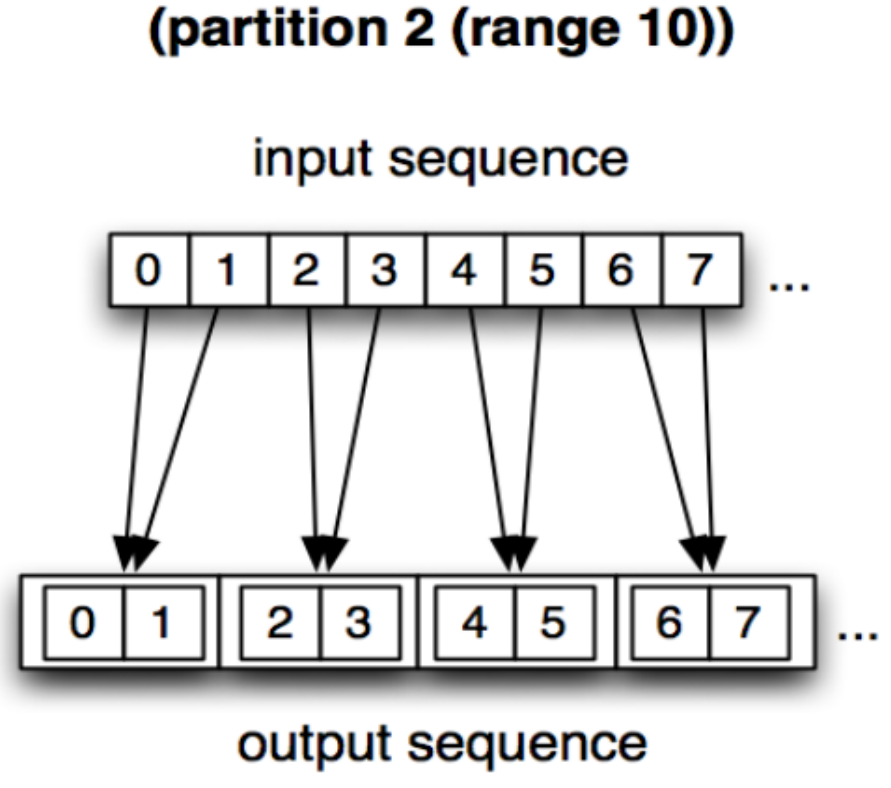
Streams

- Streams (or lazy sequences) abstracts from looping
- Stream operators are more declarative than loops
  - Map
  - Filter
  - Reduce



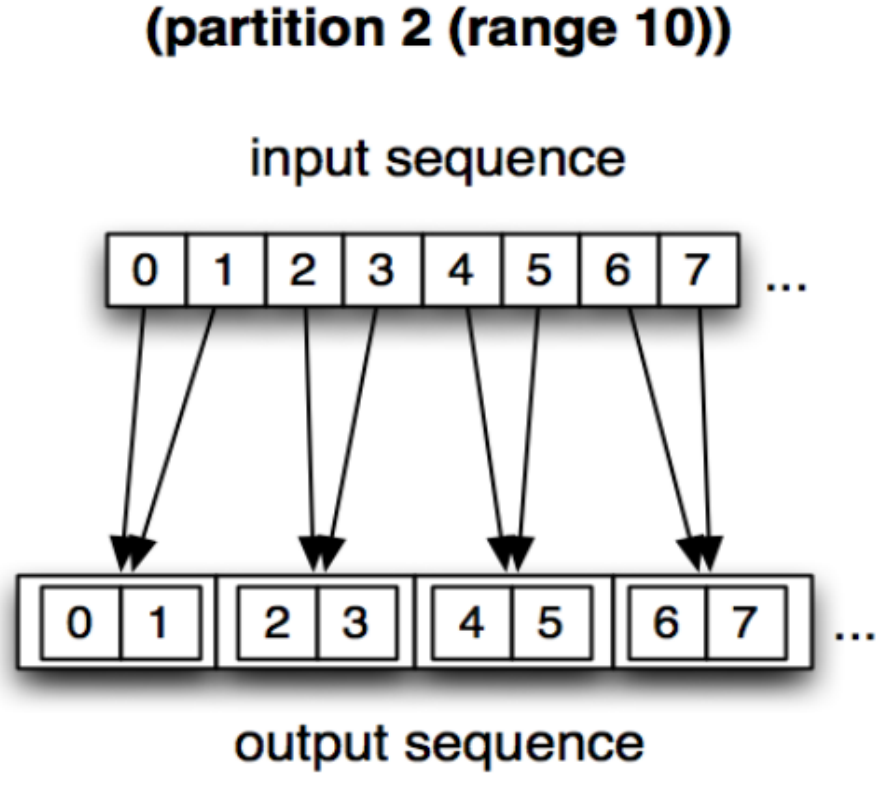
Streams

- Streams (or lazy sequences) abstracts from looping
- Stream operators are more declarative than loops
  - Map
  - Filter
  - Reduce
  - Partition



Streams

- Streams (or lazy sequences) abstracts from looping
- Stream operators are more declarative than loops
  - Map
  - Filter
  - Reduce
  - Partition
  - And so on...



Streams

- Streams (or lazy sequences) abstracts from looping
- Stream operators are more declarative than loops
  - Map
  - Filter
  - Reduce
  - Partition
  - And so on...
- It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures

Alan Perlis 192[illegible]
Painless concurrency and parallelism

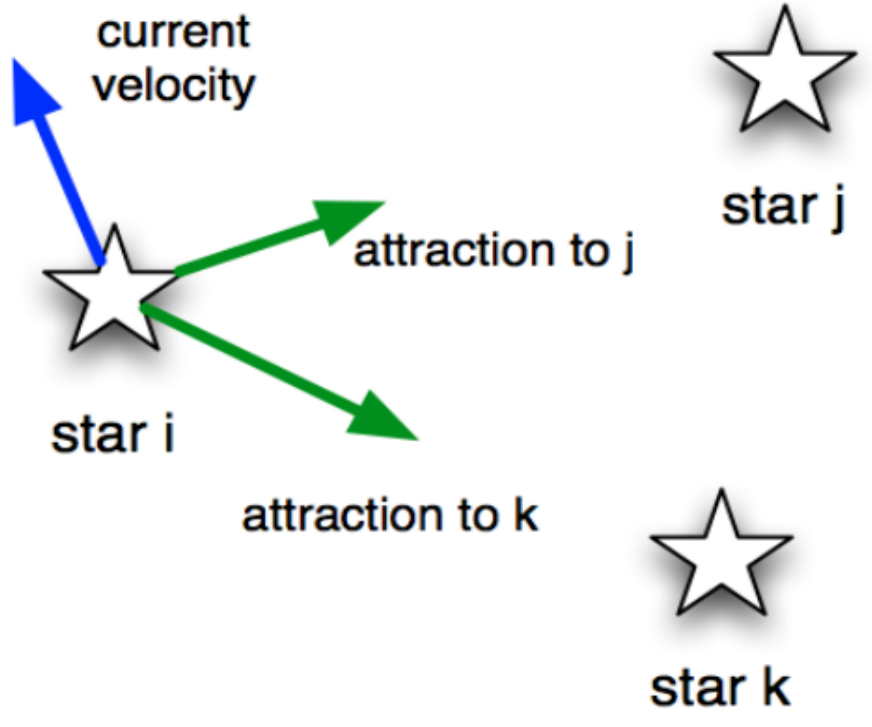
Galaxy simulation example

Galaxy simulation example

- Let's simulate a galaxy of n stars naïvely (O(n^2))

Galaxy simulation example

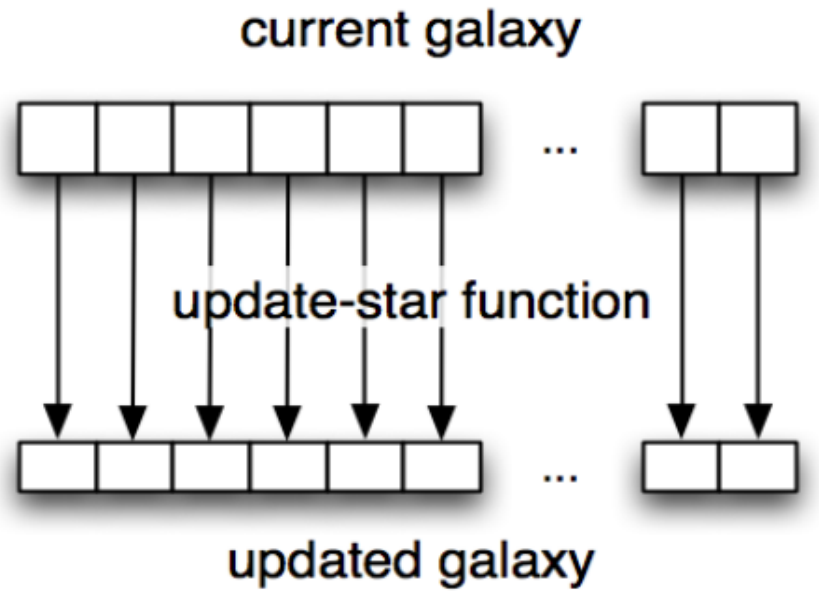
- Let's simulate a galaxy of n stars naïvely (O(n^2))



Galaxy simulation example

- Let's simulate a galaxy of n stars naïvely (O(n^2))

```
1  (defn update-galaxy [delta galaxy]
2    (vec
3      (map
4        (partial update-star delta galaxy)
5        galaxy)))
```

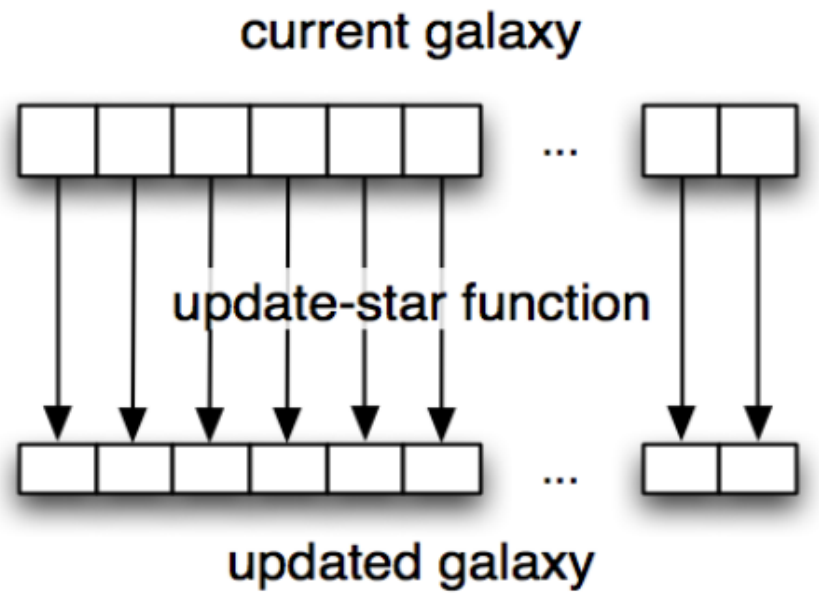


Galaxy simulation example

- Let's simulate a galaxy of n stars naïvely (O(n^2))

```
1  (defn update-galaxy [delta galaxy]
2    (vec
3      (map
4        (partial update-star delta galaxy)
5        galaxy)))
```

```
1  (defn simulate [updatef delta steps galaxy]
2    (->> galaxy
3      (iterate (partial updatef delta))
4      (drop steps)
5      first))
```

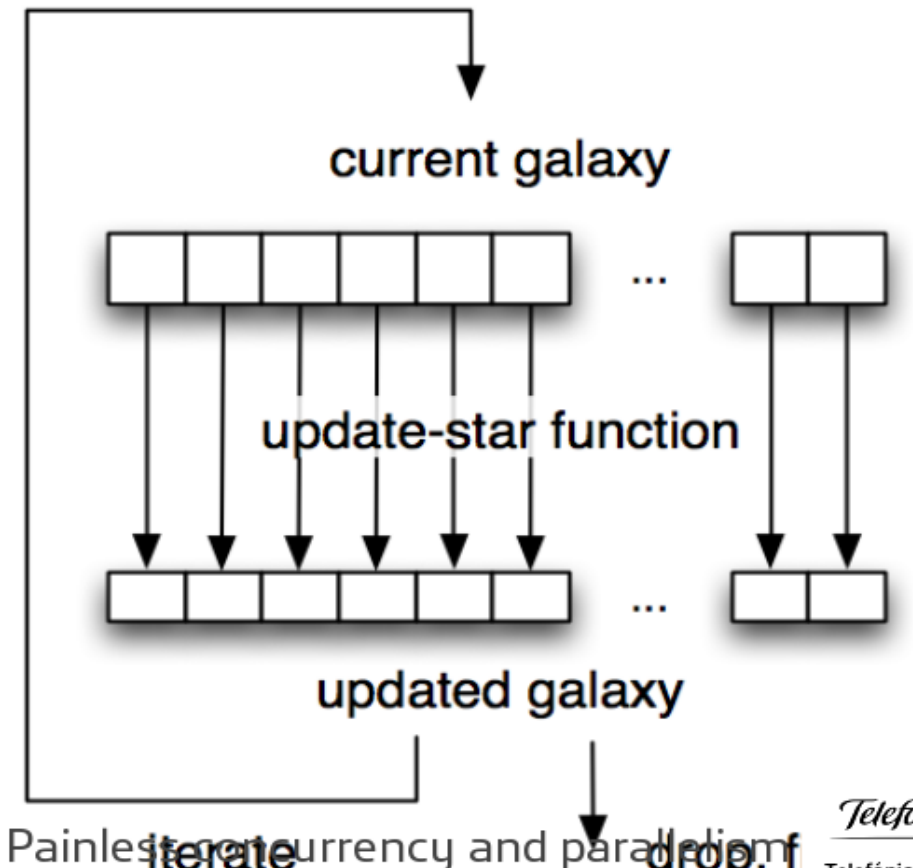


Galaxy simulation example

- Let's simulate a galaxy of n stars naïvely (O(n^2))

```
1  (defn update-galaxy [delta galaxy]
2    (vec
3      (map
4        (partial update-star delta galaxy)
5        galaxy)))
```

```
1  (defn simulate [updatef delta steps galaxy]
2    (->> galaxy
3      (iterate (partial updatef delta))
4      (drop steps)
5      first))
```



Painless concurrency and parallelism

Telefónica Digital

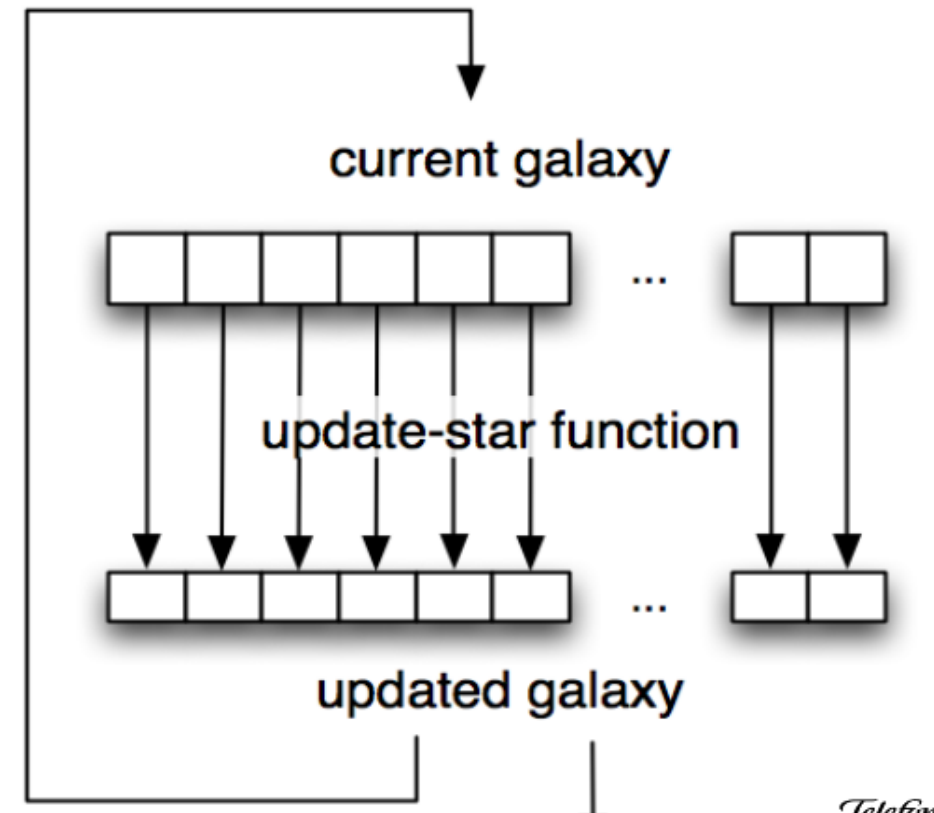
Galaxy simulation example

- Let's simulate a galaxy of n stars naïvely (O(n^2))

```
1  (defn update-galaxy [delta galaxy]
2    (vec
3      (map
4        (partial update-star delta galaxy)
5        galaxy)))
```

```
1  (defn simulate [updatef delta steps galaxy]
2    (->> galaxy
3      (iterate (partial updatef delta))
4      (drop steps)
5      first))
```

```
1  (simulate update-galaxy 0.1 100 galaxy)
```



Galaxy simulation example

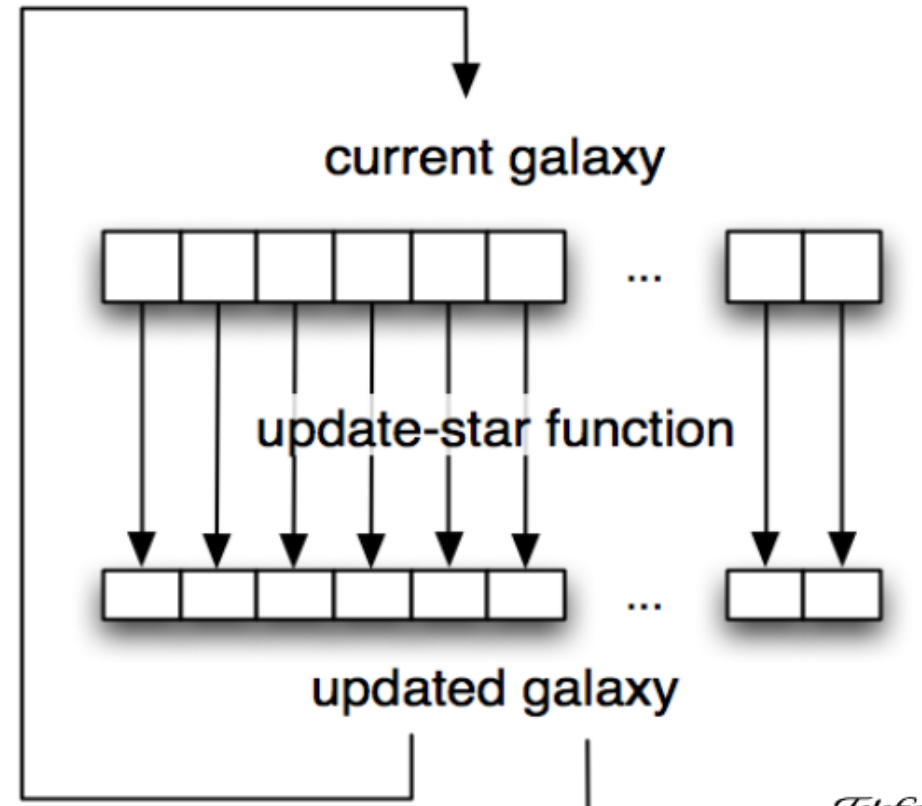
- Let's simulate a galaxy of n stars naïvely (O(n^2))

```
1  (defn update-galaxy [delta galaxy]
2    (vec
3      (map
4        (partial update-star delta galaxy)
5        galaxy)))
```

```
1  (defn simulate [updatef delta steps galaxy]
2    (->> galaxy
3      (iterate (partial updatef delta))
4      (drop steps)
5      first))
```

```
1  (simulate update-galaxy 0.1 100 galaxy)
```

- How difficult is parallelize the map part?



Painless concurrency and parallelism

Telefónica Digital

Galaxy simulation example

- Let's simulate a galaxy of n stars naïvely (O(n^2))

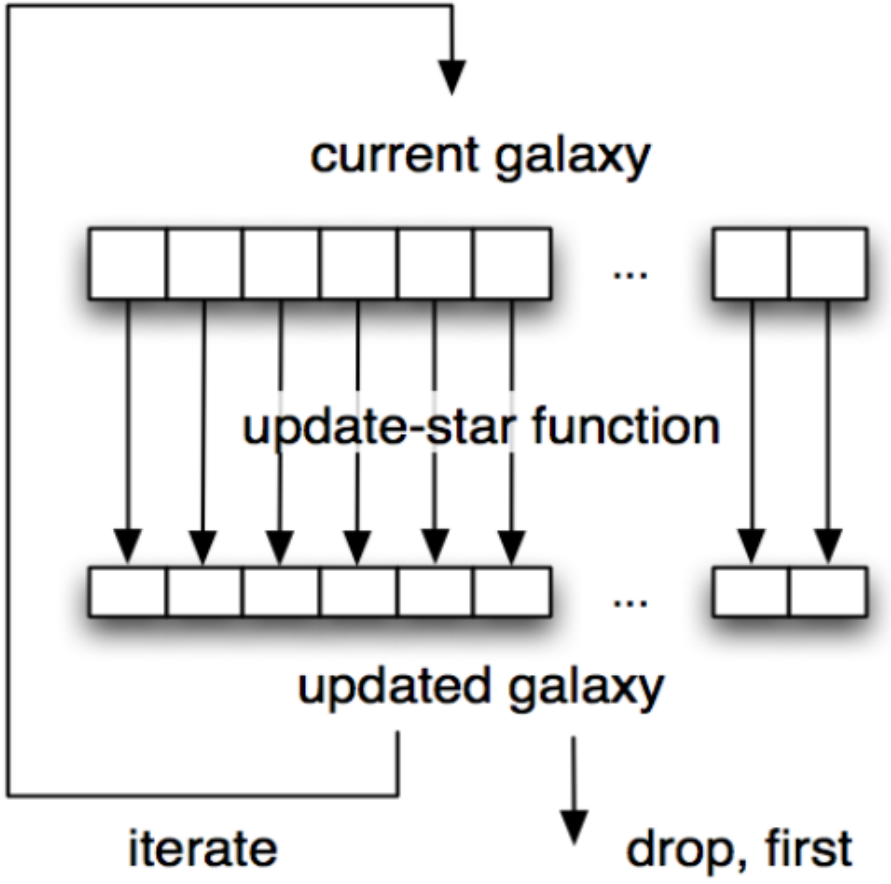
```clojure
(defn update-galaxy [delta galaxy]
  (vec
    (map
      (partial update-star delta galaxy)
      galaxy)))
```

```clojure
(defn simulate [updatef delta steps galaxy]
  (->> galaxy
    (iterate (partial updatef delta))
    (drop steps)
    first))
```

```clojure
(simulate update-galaxy 0.1 100 galaxy)
```

- How difficult is parallelize the map part?

```clojure
(defn update-galaxy-2 [delta galaxy]
  (vec
    (pmap
      (partial update-star delta galaxy)
      galaxy)))
```



Galaxy simulation example

- Let's simulate a galaxy of n stars naïvely (O(n^2))

```clojure
(defn update-galaxy [delta galaxy]
  (vec
    (map
      (partial update-star delta galaxy)
      galaxy)))
```

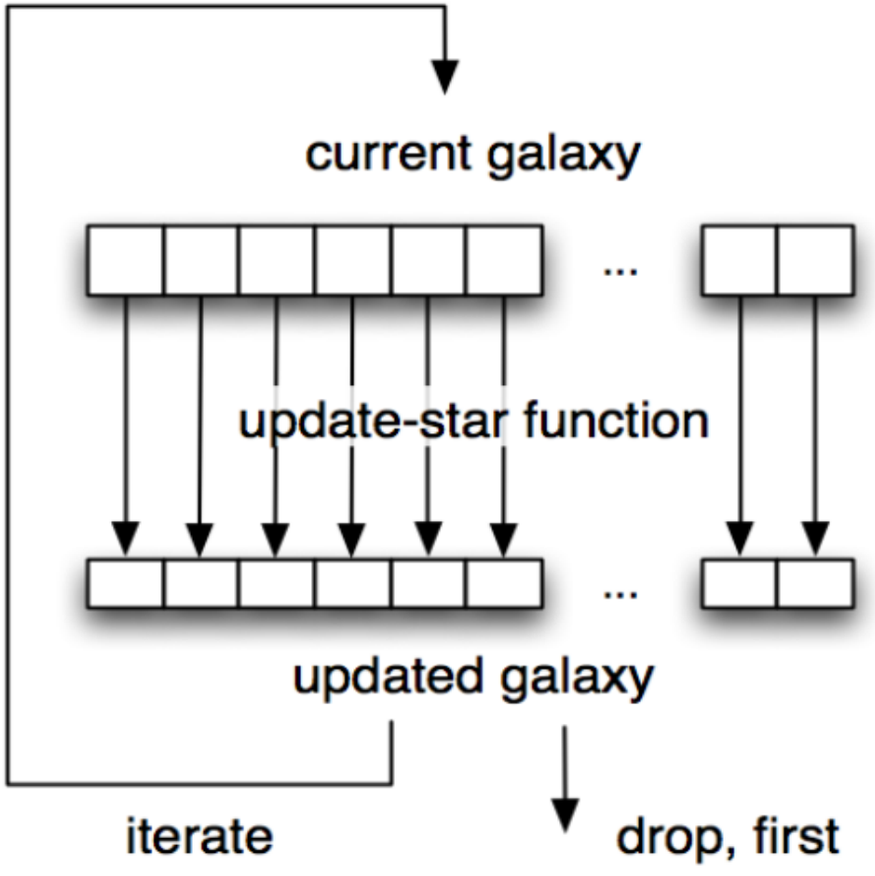
```clojure
(defn simulate [updatef delta steps galaxy]
  (->> galaxy
    (iterate (partial updatef delta))
    (drop steps)
    first))
```

```clojure
(simulate update-galaxy 0.1 100 galaxy)
```

- How difficult is parallelize the map part?

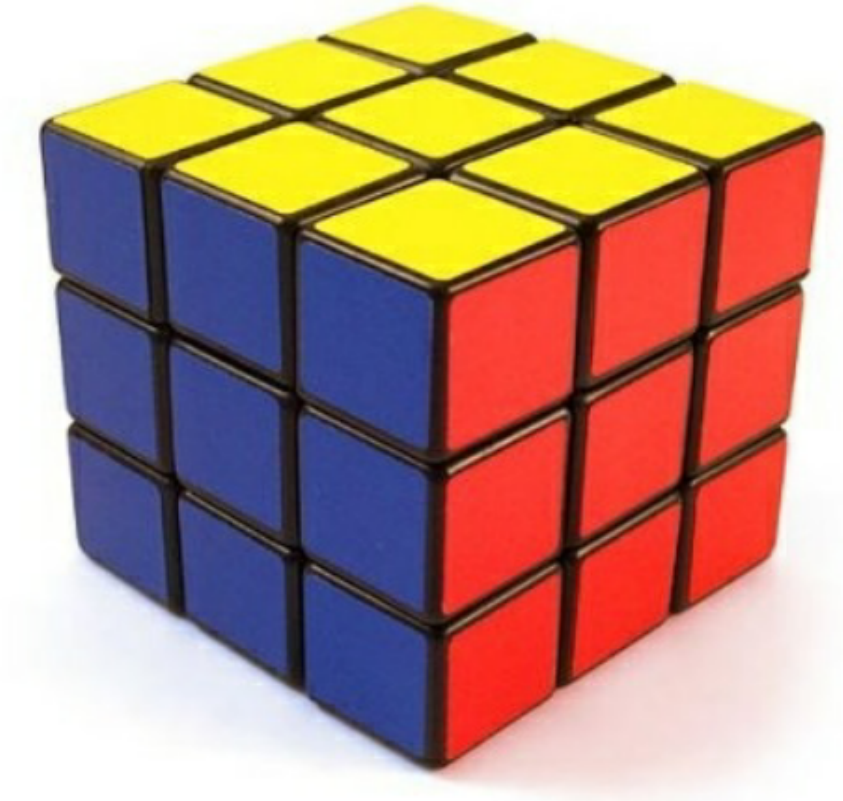
```clojure
(defn update-galaxy-2 [delta galaxy]
  (vec
    (pmap
      (partial update-star delta galaxy)
      galaxy)))
```

- ~50% speed up just by changing map by pmap



Problem solved?

Problem solved?

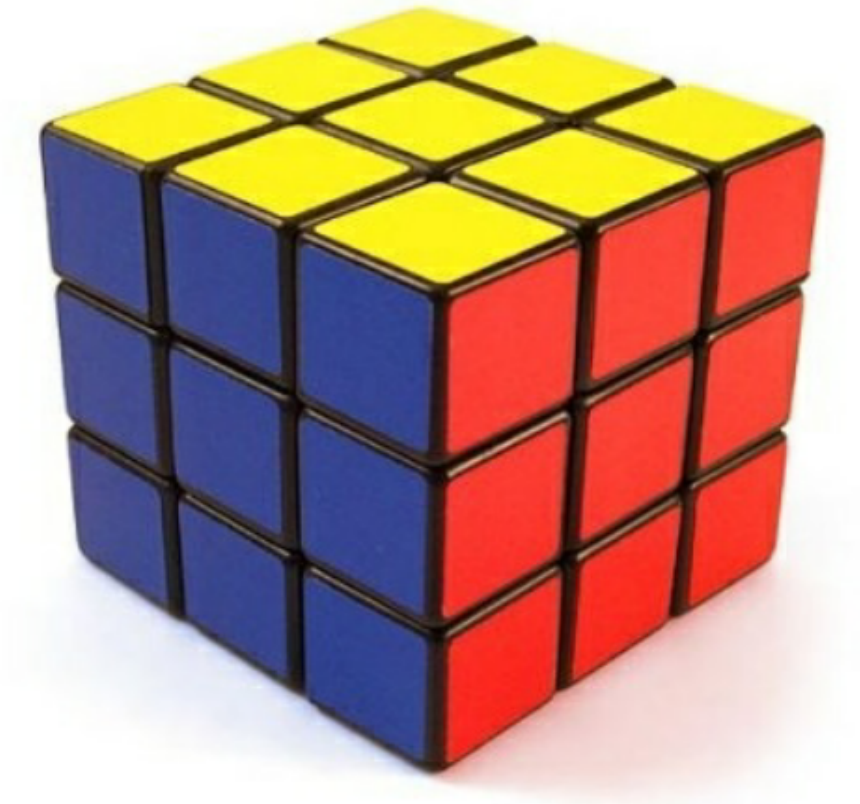


Ta-da!

Painless concurrency and parallelism

Problem solved?

- Let's look at other example: word counting

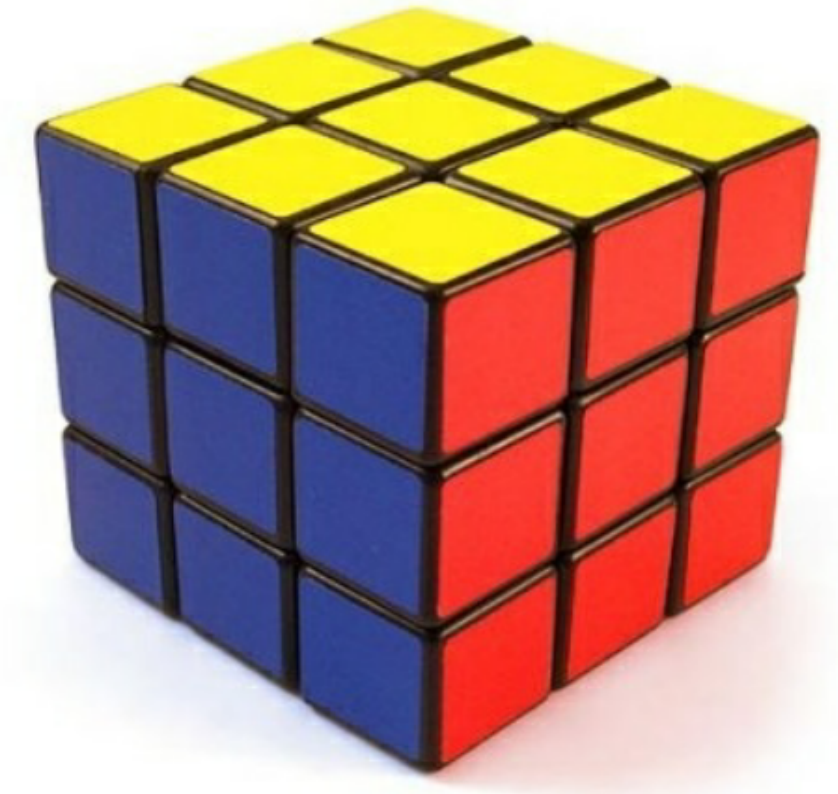


Ta-da!

Problem solved?

- Let's look at other example: word counting

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```



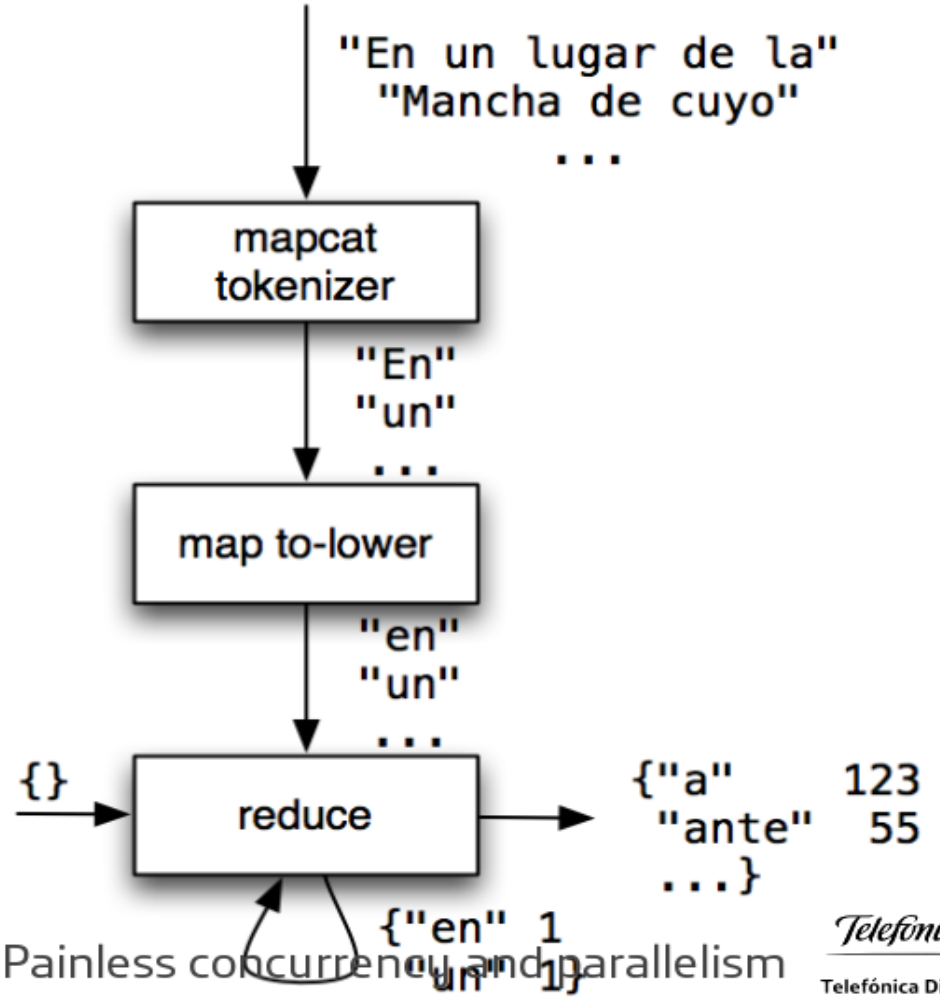
Ta-da!

Painless concurrency and parallelism

Problem solved?

- Let's look at other example: word counting

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

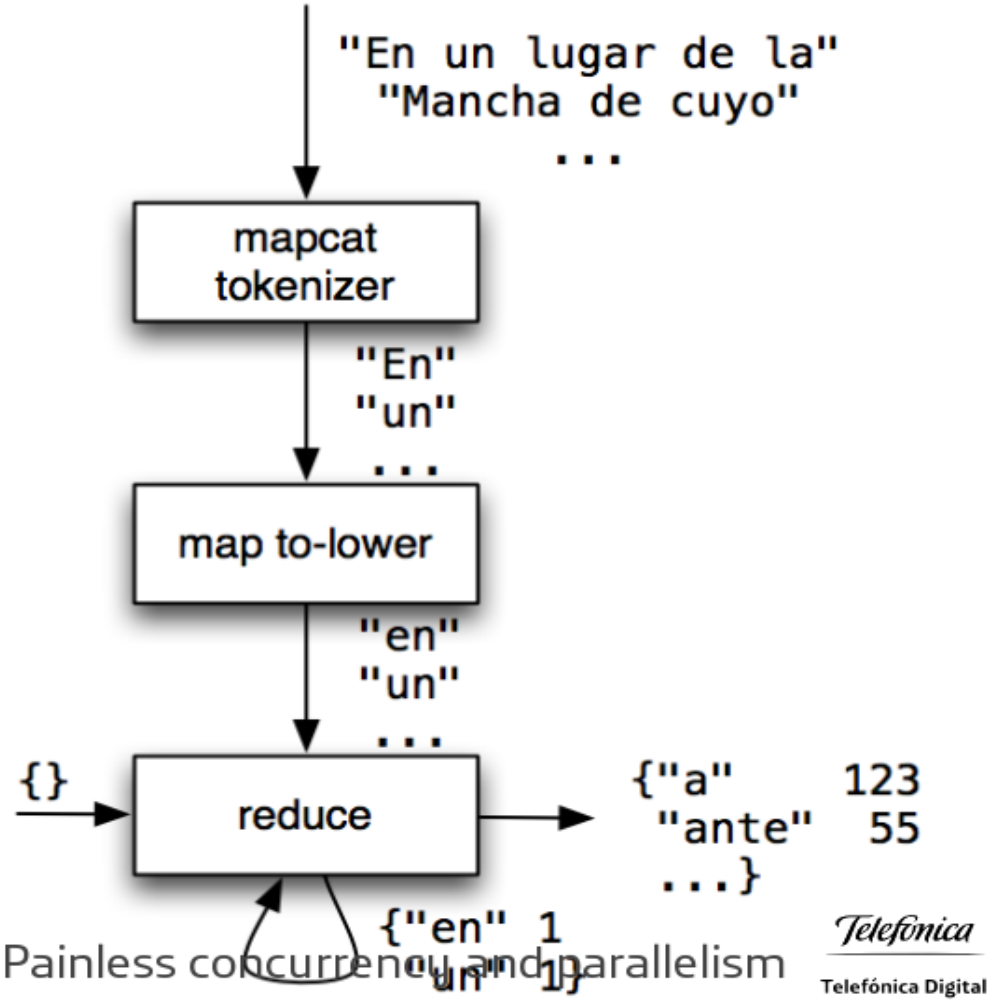


Problem solved?

- Let's look at other example: word counting

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

- pmap all over the place



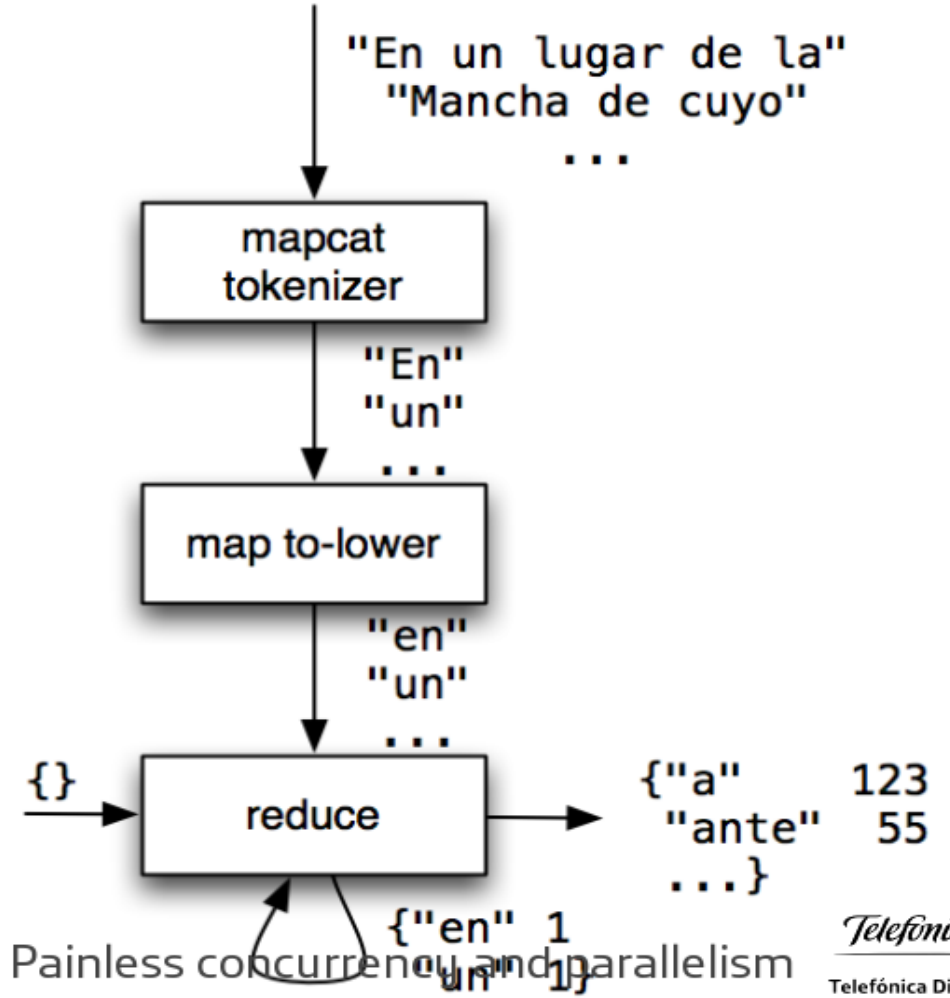
Problem solved?

- Let's look at other example: word counting

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

- pmap all over the place

```
1  (defn wordcount-pmap [lines]
2    (->> lines
3      (pmap (partial re-seq #"\w+"))
4      (apply concat)
5      (pmap to-lower)
6      (reduce inc-count {})))
```



Problem solved?

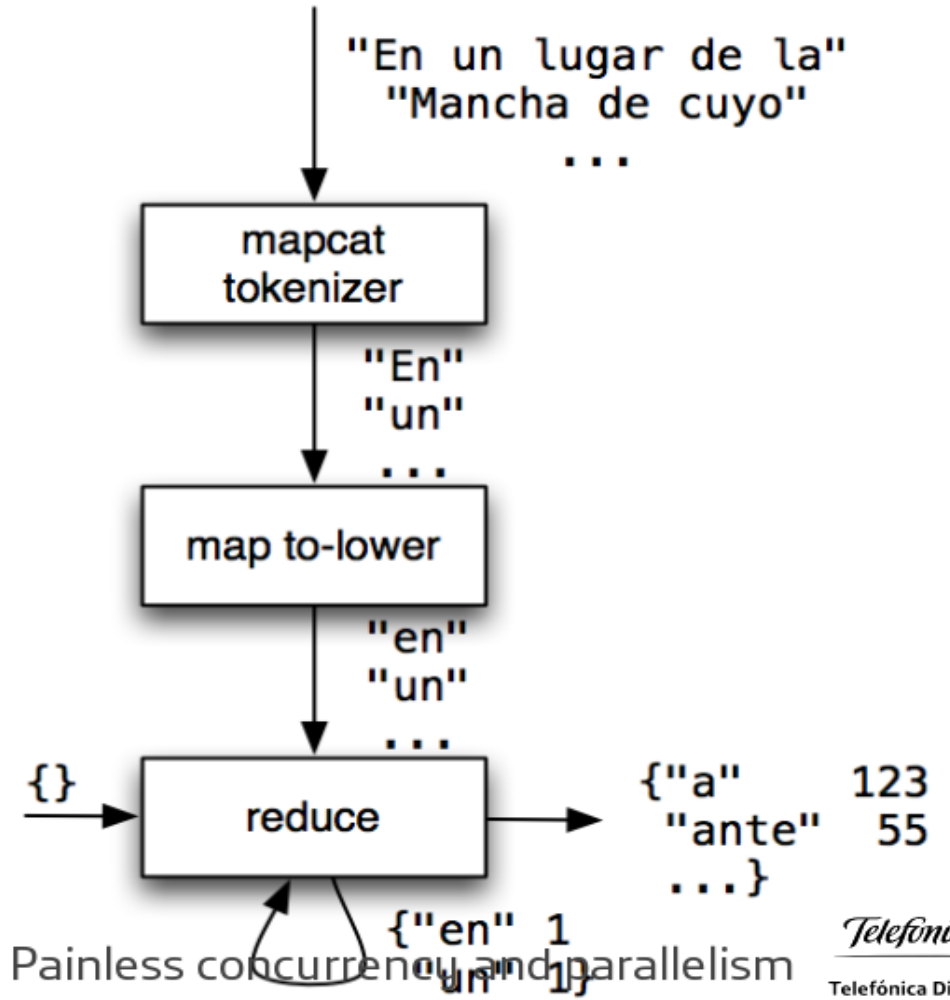
- Let's look at other example: word counting

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

- pmap all over the place

```
1  (defn wordcount-pmap [lines]
2    (->> lines
3      (pmap (partial re-seq #"\w+"))
4      (apply concat)
5      (pmap to-lower)
6      (reduce inc-count {})))
```

- Speed up of -450%! Synchronization time dominates.



Problem solved?

- Let's look at other example: word counting

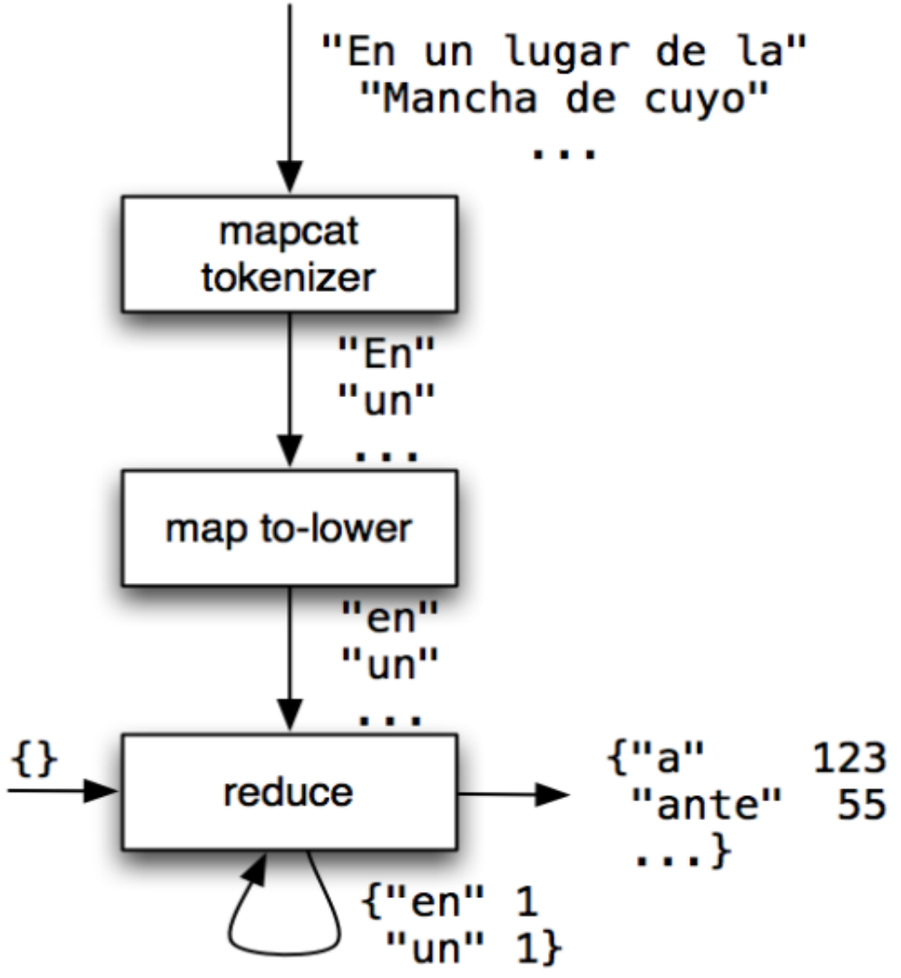
```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

- pmap all over the place

```
1  (defn wordcount-pmap [lines]
2    (->> lines
3      (pmap (partial re-seq #"\w+"))
4      (apply concat)
5      (pmap to-lower)
6      (reduce inc-count {})))
```

- Speed up of -450%! Synchronization time dominates.

```
1  (defn wordcount-buckets [lines]
2    (->> lines
3      (partition 512)
4      (pmap wordcount)
5      (reduce combine-sum)))
```



Problem solved?

- Let's look at other example: word counting

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

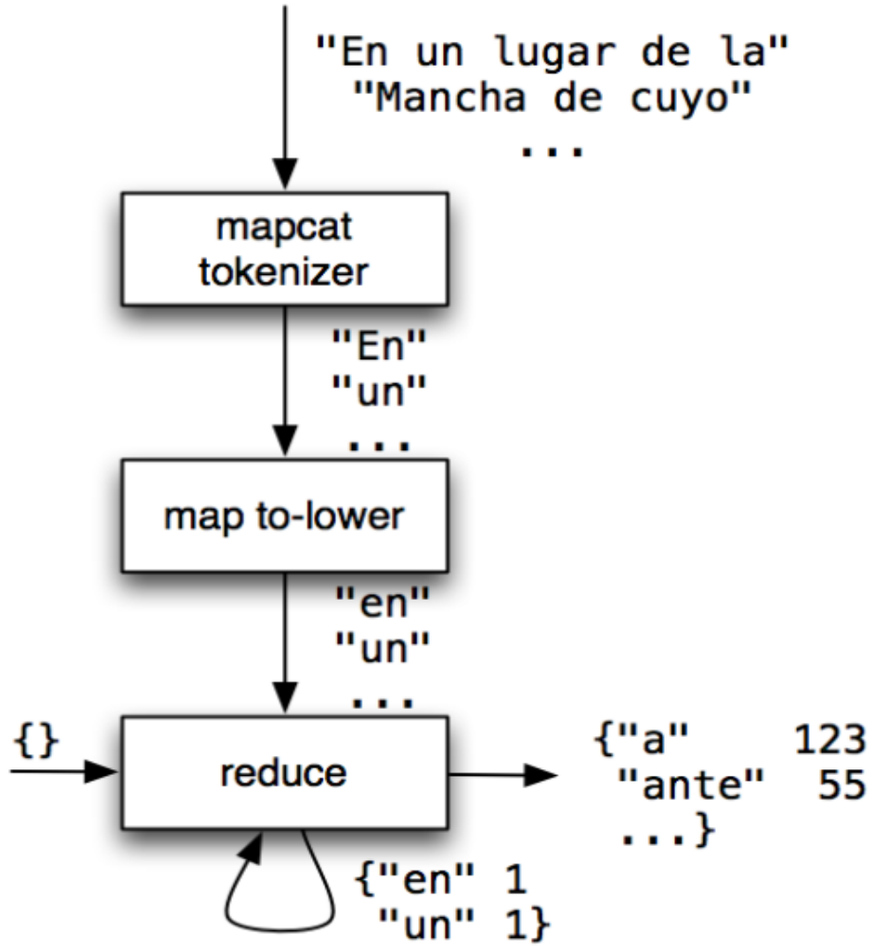
- pmap all over the place

```
1  (defn wordcount-pmap [lines]
2    (->> lines
3      (pmap (partial re-seq #"\w+"))
4      (apply concat)
5      (pmap to-lower)
6      (reduce inc-count {})))
```

- Speed up of -450%! Synchronization time dominates.

```
1  (defn wordcount-buckets [lines]
2    (->> lines
3      (partition 512)
4      (pmap wordcount)
5      (reduce combine-sum)))
```

- Speed up of 41%



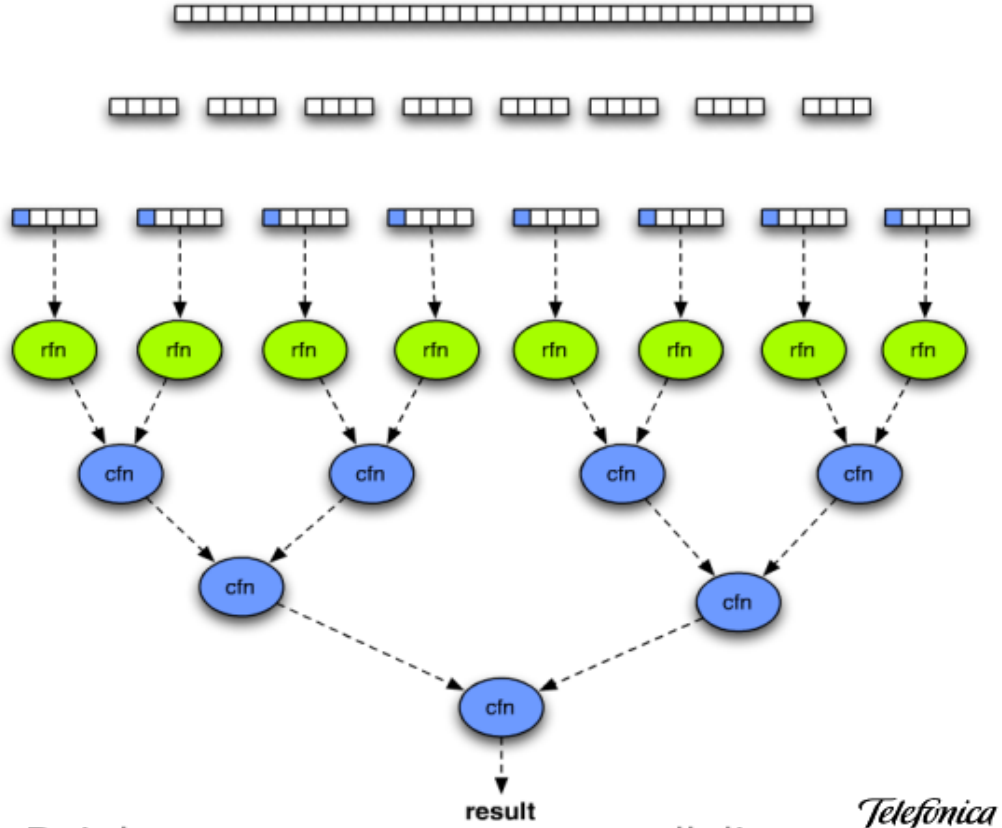
Reducers

Reducers

- The stream abstraction is intrinsically serial

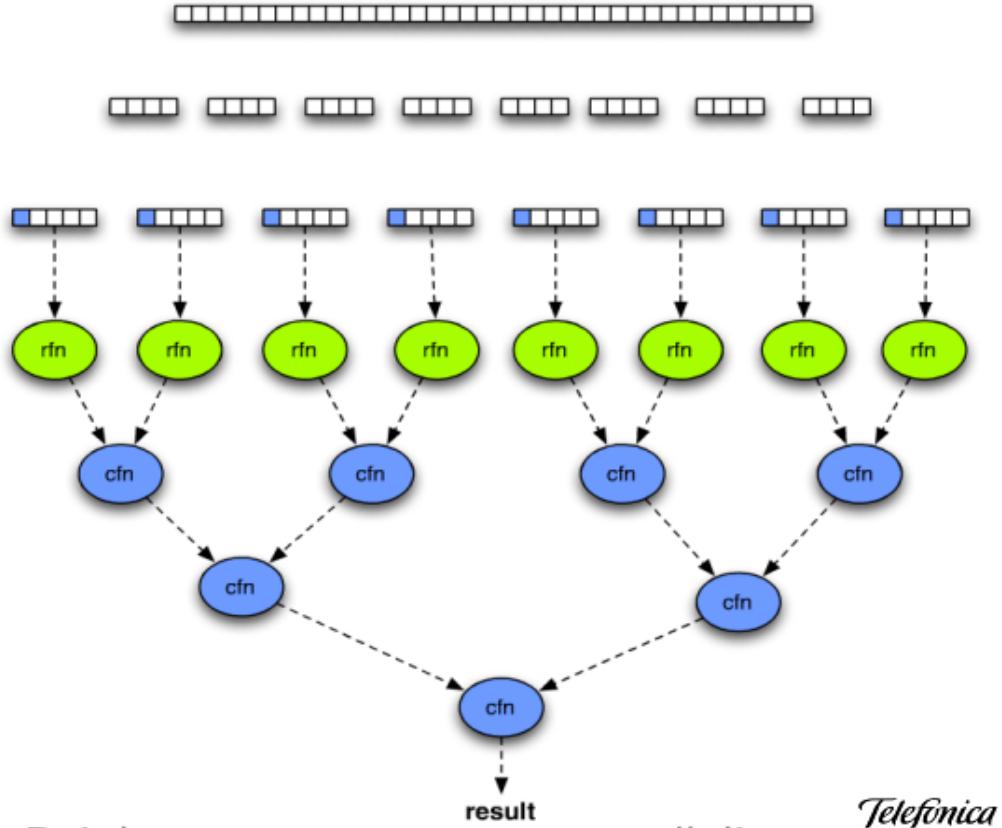
Reducers

- The stream abstraction is intrinsically serial



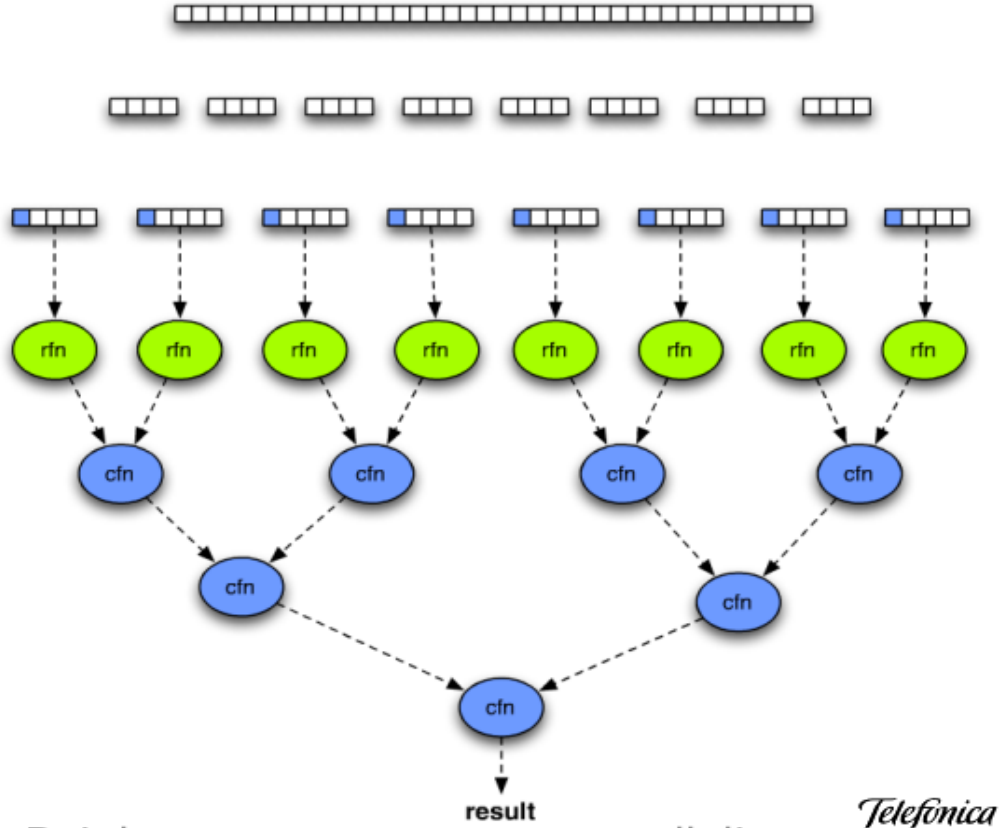
Reducers

- The stream abstraction is intrinsically serial
- The reduce-combine model unleashes parallelism



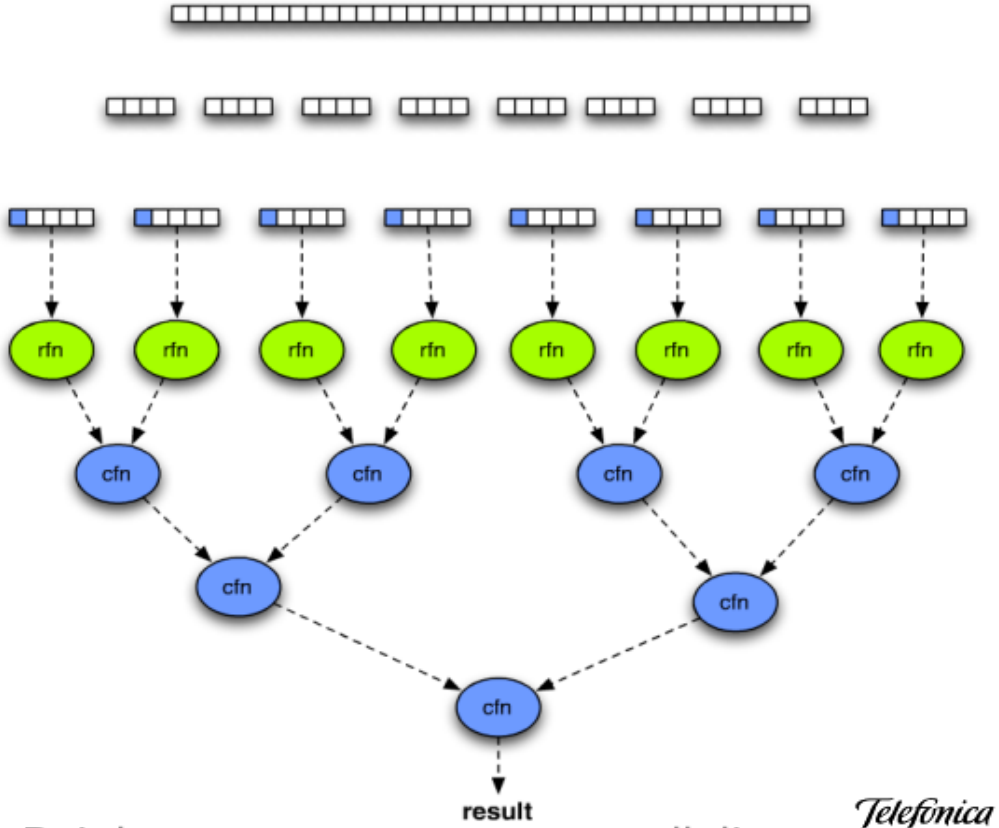
Reducers

- The stream abstraction is intrinsically serial
- The reduce-combine model unleashes parallelism
- Can we modify streams to use this model?



Reducers

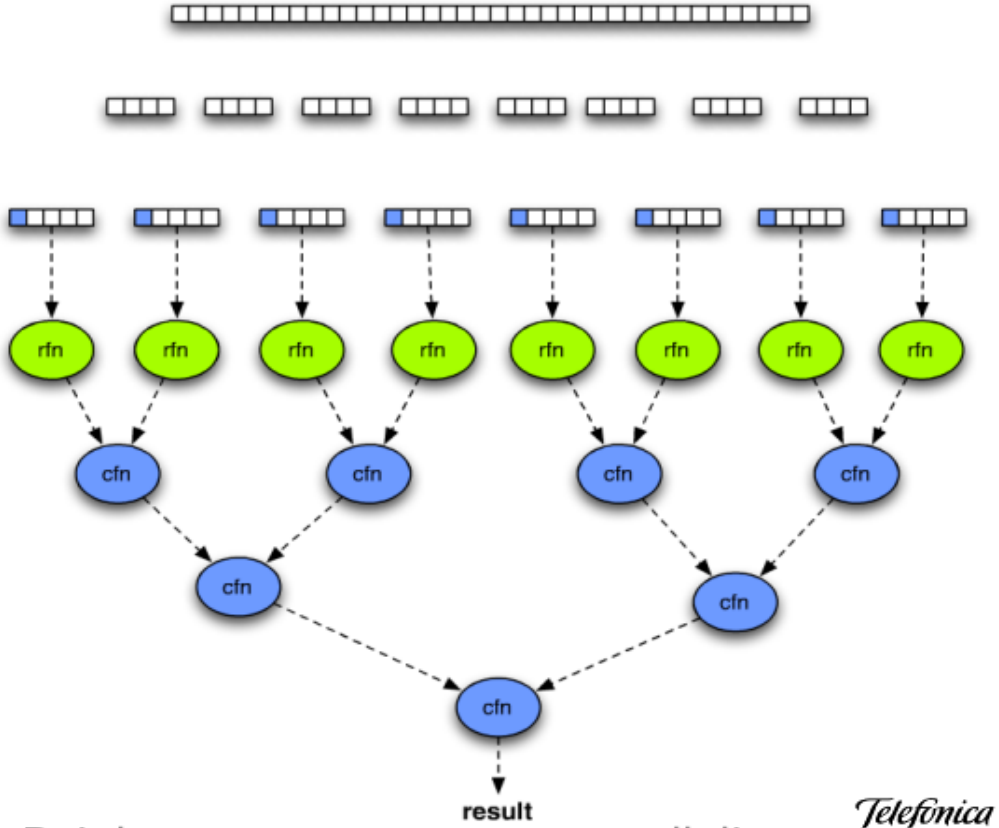
- The stream abstraction is intrinsically serial
- The reduce-combine model unleashes parallelism
- Can we modify streams to use this model?
  - Before



Reducers

- The stream abstraction is intrinsically serial
- The reduce-combine model unleashes parallelism
- Can we modify streams to use this model?
  - Before

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

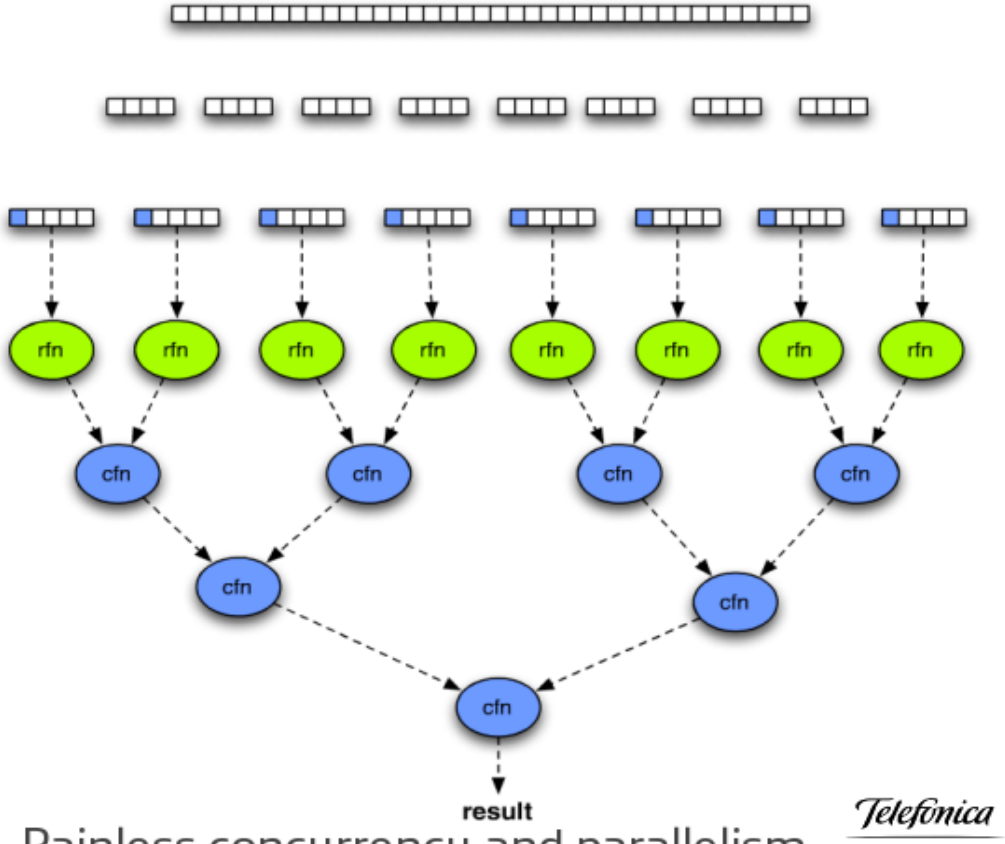


Reducers

- The stream abstraction is intrinsically serial
- The reduce-combine model unleashes parallelism
- Can we modify streams to use this model?
  - Before

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

  - After



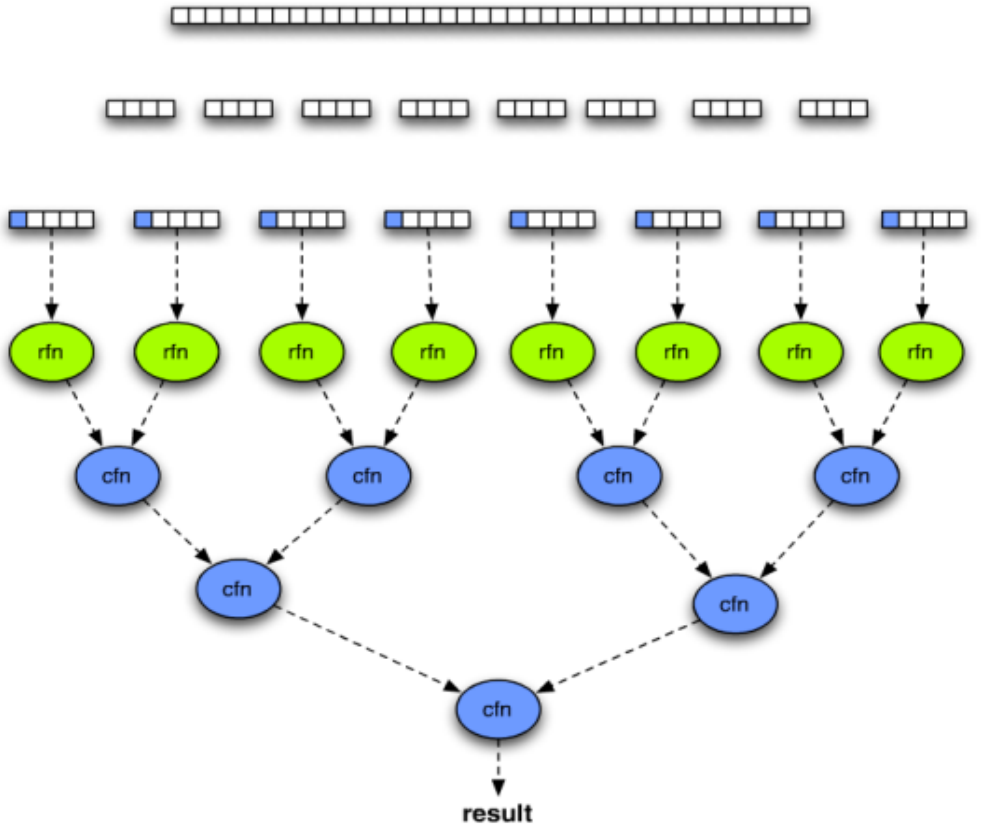
Reducers

- The stream abstraction is intrinsically serial
- The reduce-combine model unleashes parallelism
- Can we modify streams to use this model?
  - Before

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

  - After

```
1  (defn wordcount-reduce [lines]
2    (->> lines
3      (r/mapcat (partial re-seq #"\w+"))
4      (r/map to-lower)
5      (r/fold 8192 combine-sum inc-count)))
```



Reducers

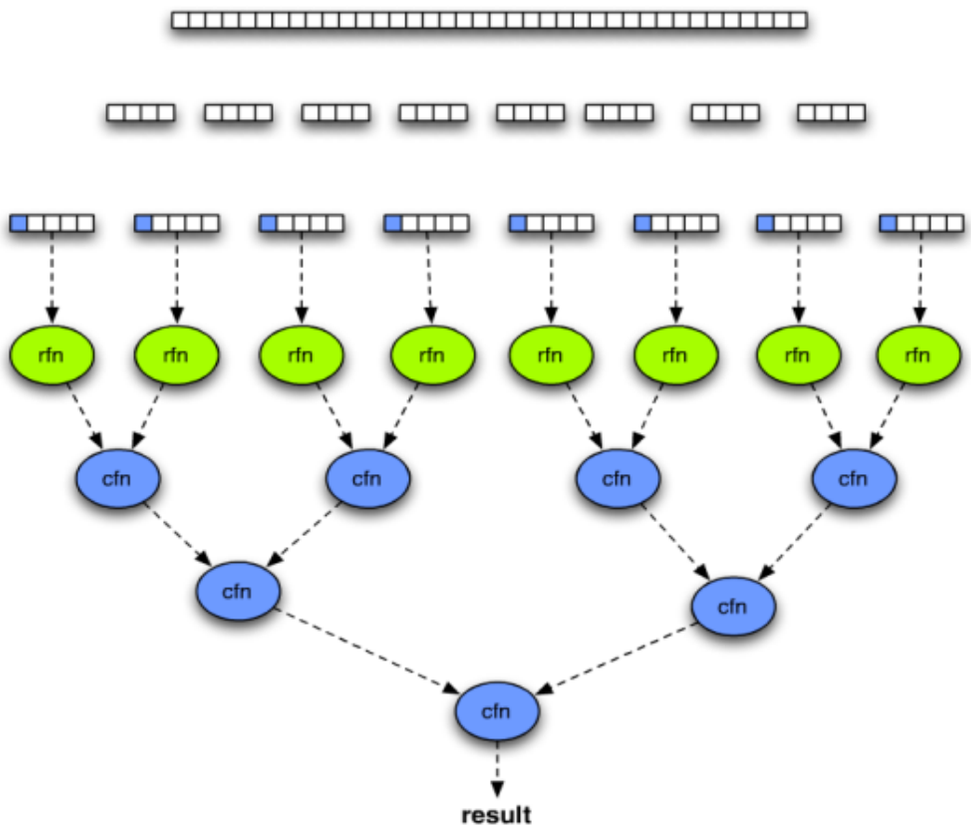
- The stream abstraction is intrinsically serial
- The reduce-combine model unleashes parallelism
- Can we modify streams to use this model?
  - Before

```
1  (defn wordcount [lines]
2    (->> lines
3      (mapcat (partial re-seq #"\w+"))
4      (map to-lower)
5      (reduce inc-count {})))
```

  - After

```
1  (defn wordcount-reduce [lines]
2    (->> lines
3      (r/mapcat (partial re-seq #"\w+"))
4      (r/map to-lower)
5      (r/fold 8192 combine-sum inc-count)))
```

- Similar speedup as bucketing



Compared times

Compared times

- Loop: 566 ms (+0.00%)

- Streams: 659 ms (+16.51%)

- Streams plenty of pmap's: 3154 ms (+457.57%)

- Streams bucketed: 334 ms (-41.00%)

- Reducers: 287 ms (-49.27%)

Conclusion

Leverage immutability, referential transparency and declarative models and stop suffering

Thanks for your time