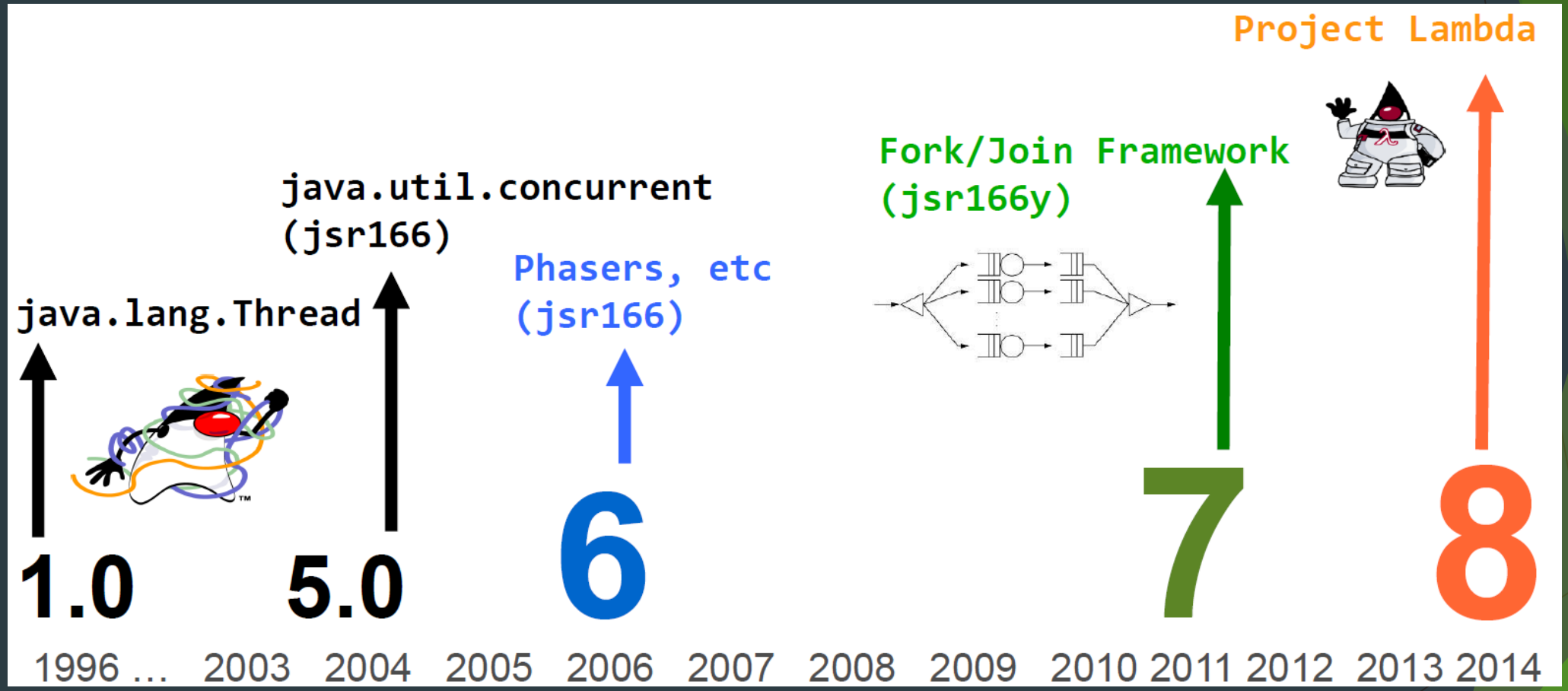
Java 8 and lambdas - did we expect too much?

Aleksander Radovan

About the author

- Java team lead at King ICT
- More than 12 years in Java
- OCP
- Senior Lecturer
- PhD student
- President of Education board at Croatian JUG (HUJAK)

Project Lambda



Simon Ritter: JDK 8 MOOC – Functional Programming in Java with Lambdas an Streams

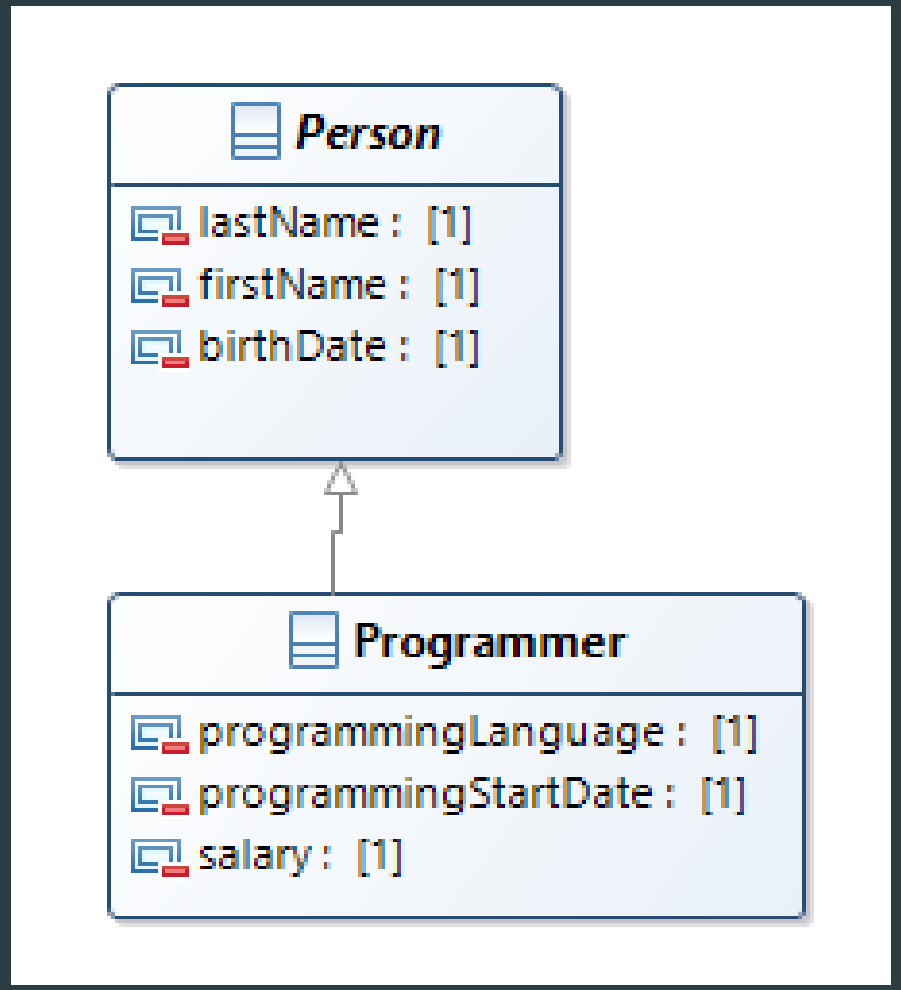
Lambda advantages

- Functional programming paradigm
- Easier to distribute processing of collections over multiple threads (CPU cores)
- Stream API
- Faster than conventional loops?

Benchmarking hardware and methodology

- CPU: Intel Core i7-4702MQ (4 cores)
- 32 GB RAM
- SSD Samsung 850 EVO 500 GB
- Eclipse Neon
- Java 8 update 101
- Generating objects with random values
- Comparing times needed to finish a specific tasks with iterators, loop, comparators, sequential and parallel streams
- Every measurement was conducted 10 times, the average values were taken into considerations

Domain model



Strategies

- Five strategies:
  - Using Iterator
  - Using foreach loop
  - Using Comparator
  - Using sequential stream
  - Using parallel stream

Iterator strategy implementation

```
Iterator<Programmer> iter = programmers.iterator();
Programmer youngestProgrammer = programmers.get(0);
iter.next();

while(iter.hasNext()) {
  Programmer newProgrammer = iter.next();
  if(newProgrammer.getBirthDate().isAfter(youngestProgrammer.getBirthDate())) {
      youngestProgrammer = newProgrammer;
  }
}
```

Foreach loop strategy implementation

```
Programmer youngestProgrammer = programmers.get(0);

for(Programmer newProgrammer : programmers.subList(1, programmers.size())) {
   if(newProgrammer.getBirthDate().isAfter(youngestProgrammer.getBirthDate())) {
      youngestProgrammer = newProgrammer;
   }
}
```

Comparator strategy implementation

```
Collections.sort(programmers, new Comparator<Programmer>() {
    @Override
    public int compare(Programmer p1, Programmer p2) {
        return p1.getBirthDate().compareTo(p2.getBirthDate());
    }
});

Programmer youngestProgrammer = programmers.get(programmers.size() - 1);
```

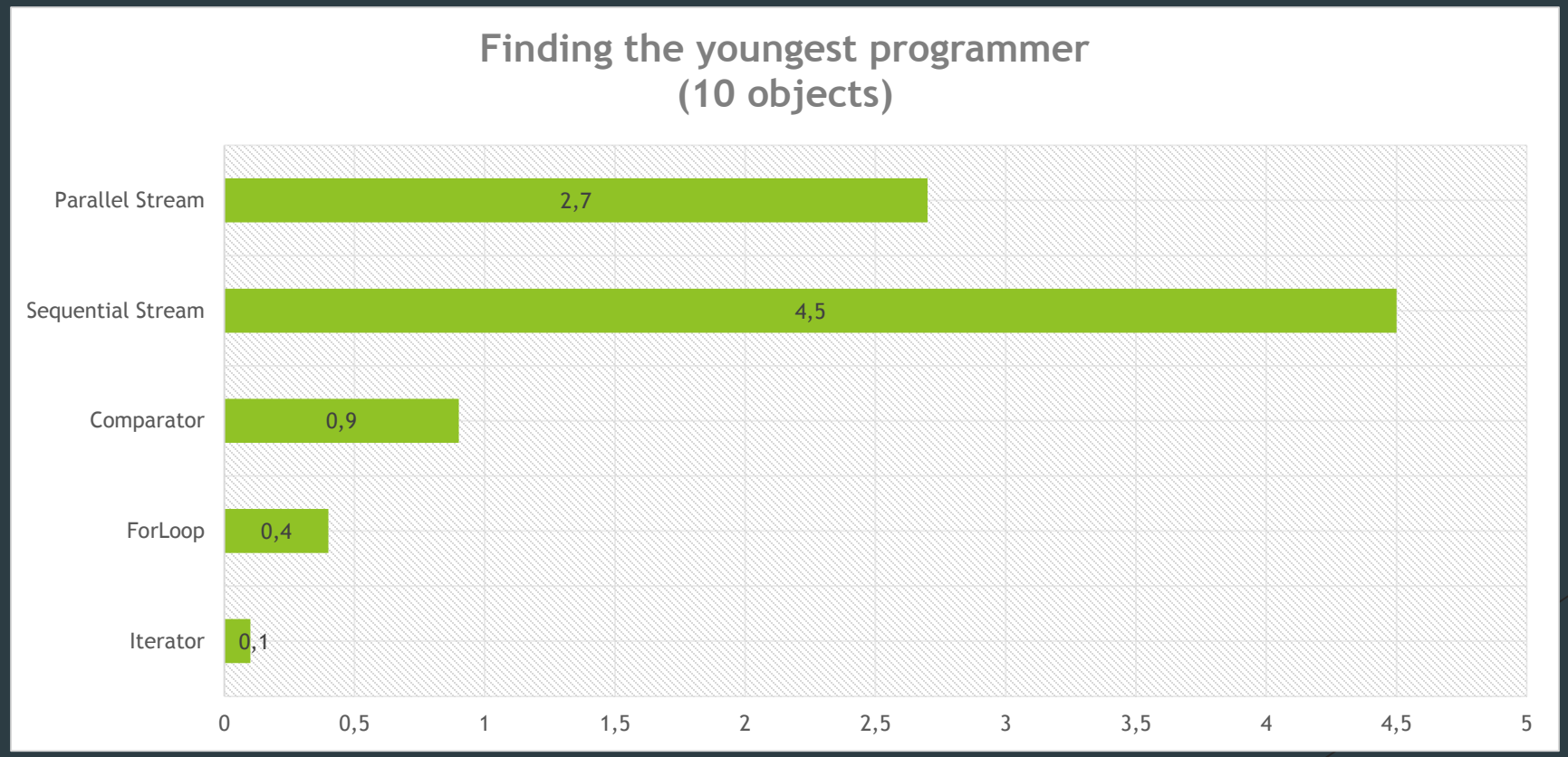
Stream API strategy implementations

```
Programmer youngestProgrammer =
   programmers.stream().max(Comparator.comparing(Programmer::getBirthDate)).get();
```

```
Programmer youngestProgrammer =
 programmers.parallelStream().max(Comparator.comparing(Programmer::getBirthDate)).get();
```

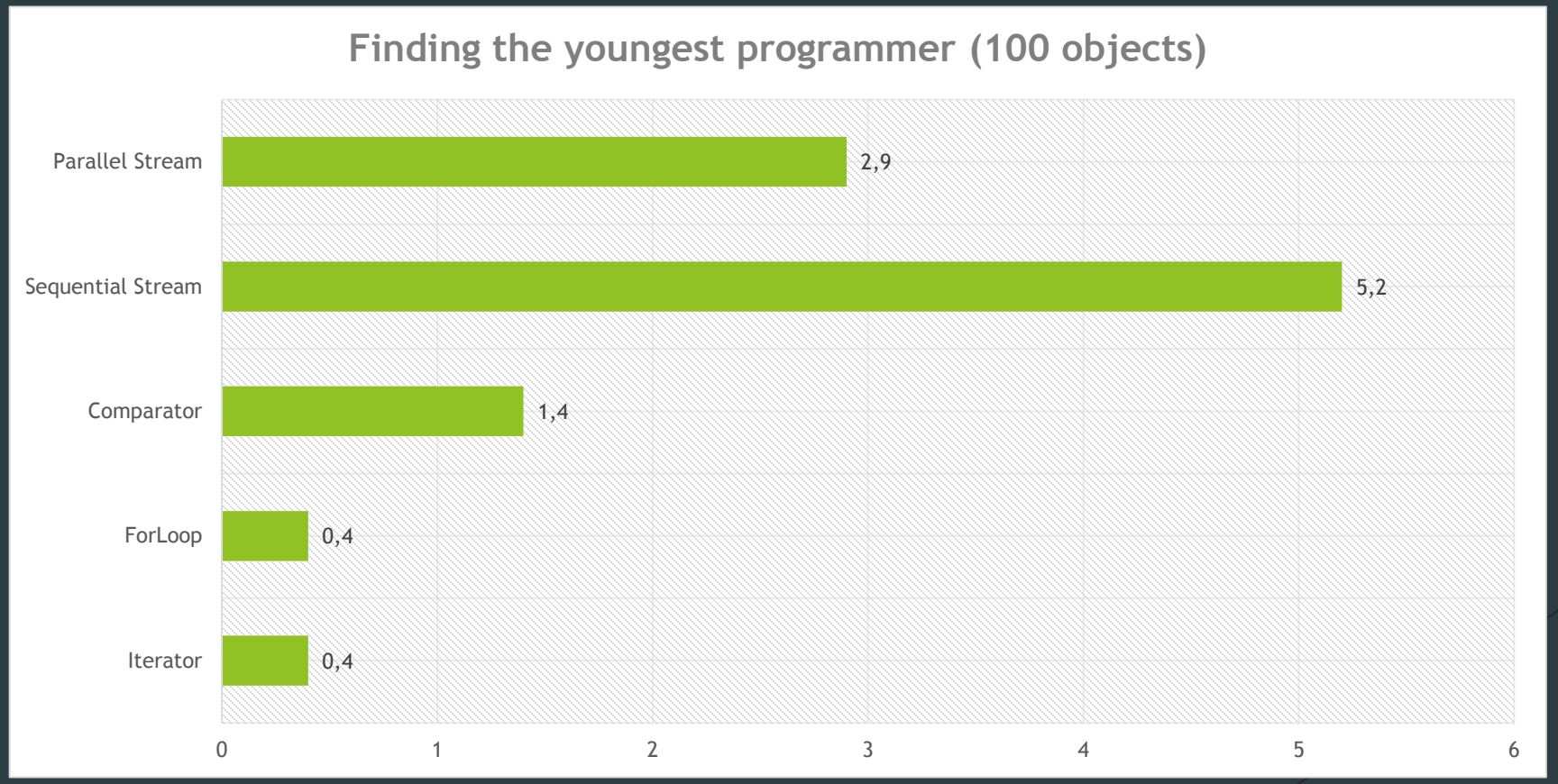
1. Case: finding the youngest programmer (10 objects)

- Results (ms):



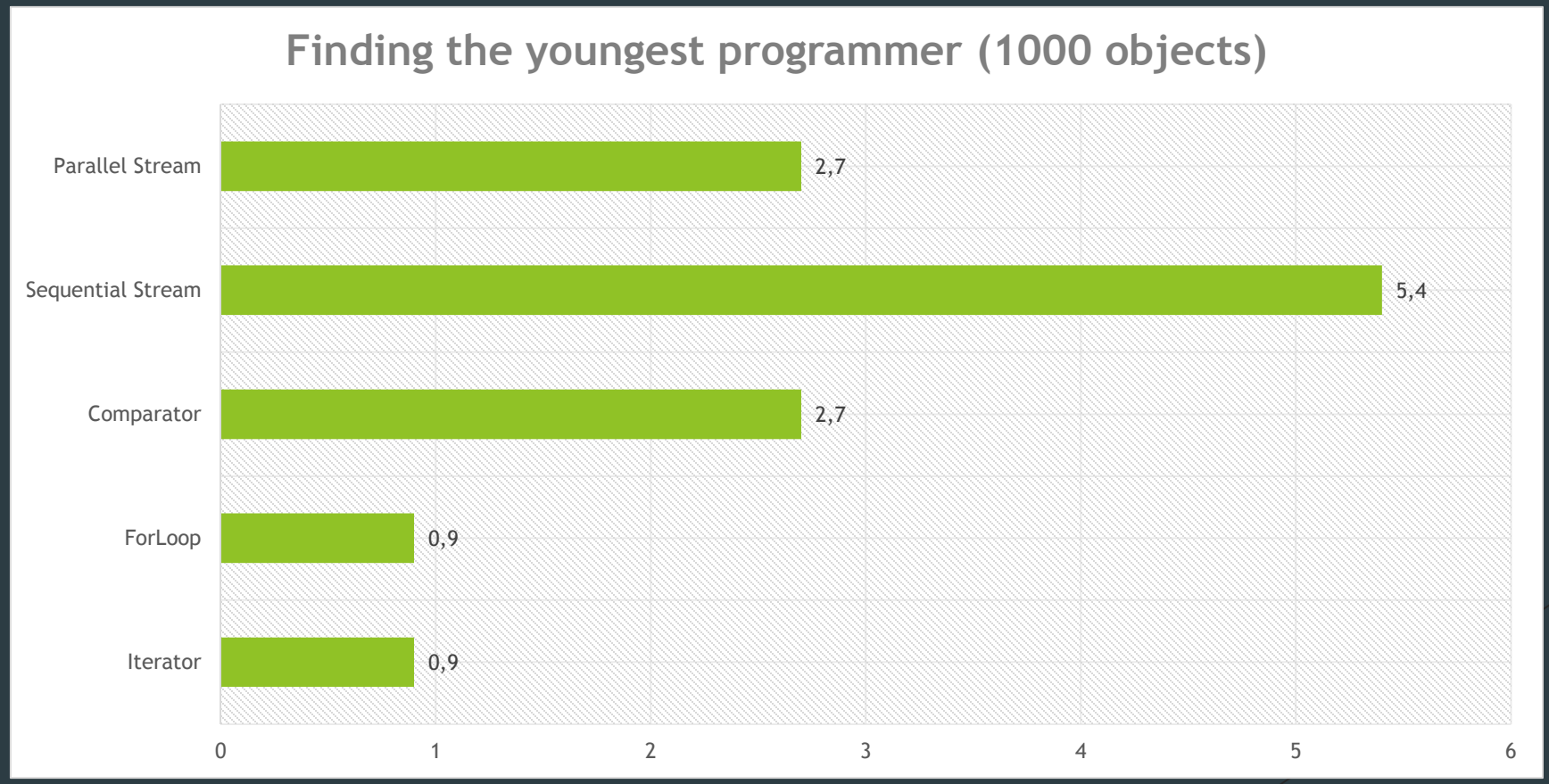
1. Case: finding the youngest programmer (100 objects)

- Results (ms):



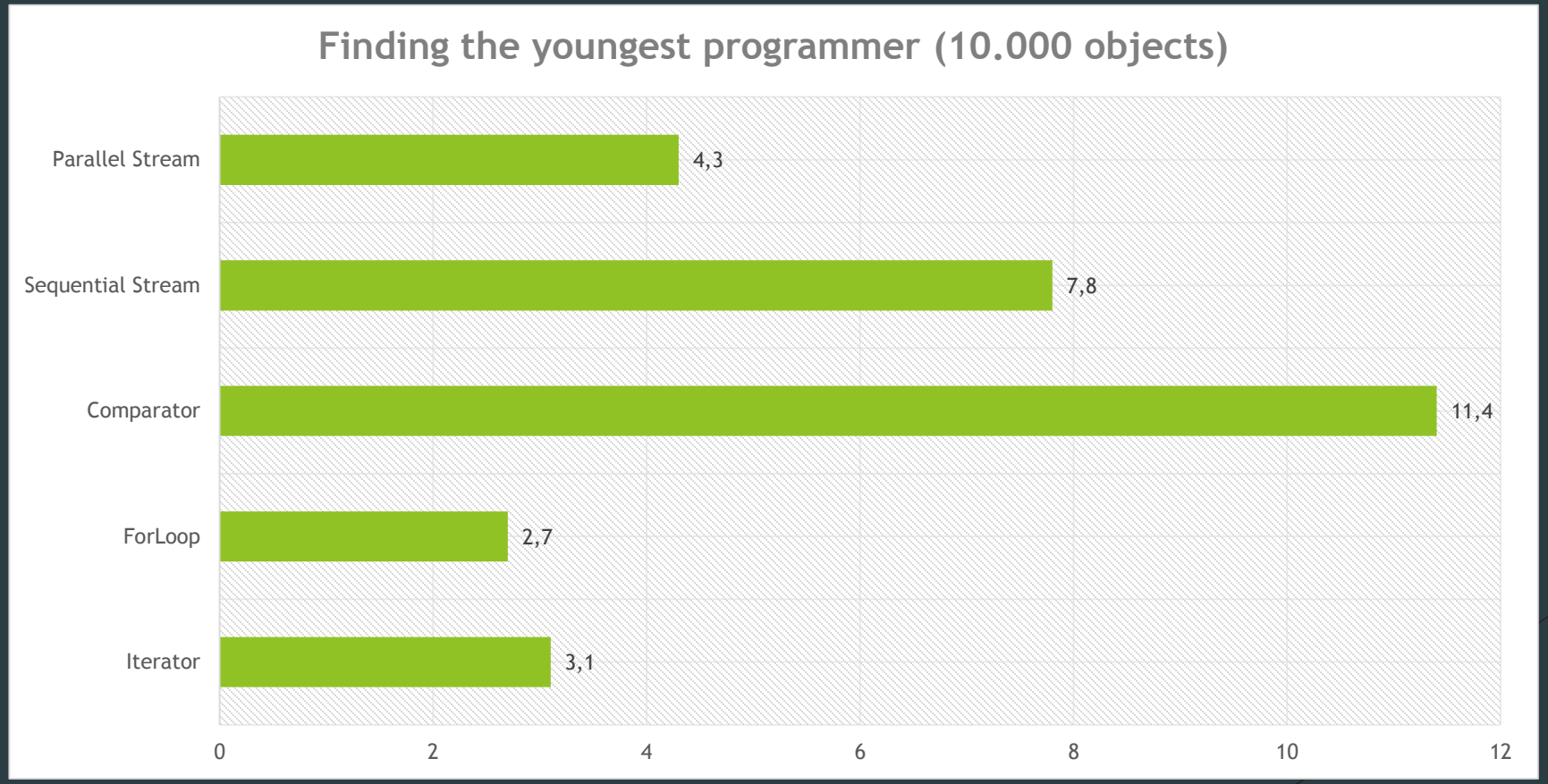
1. Case: finding the youngest programmer (1.000 objects)

- Results (ms):



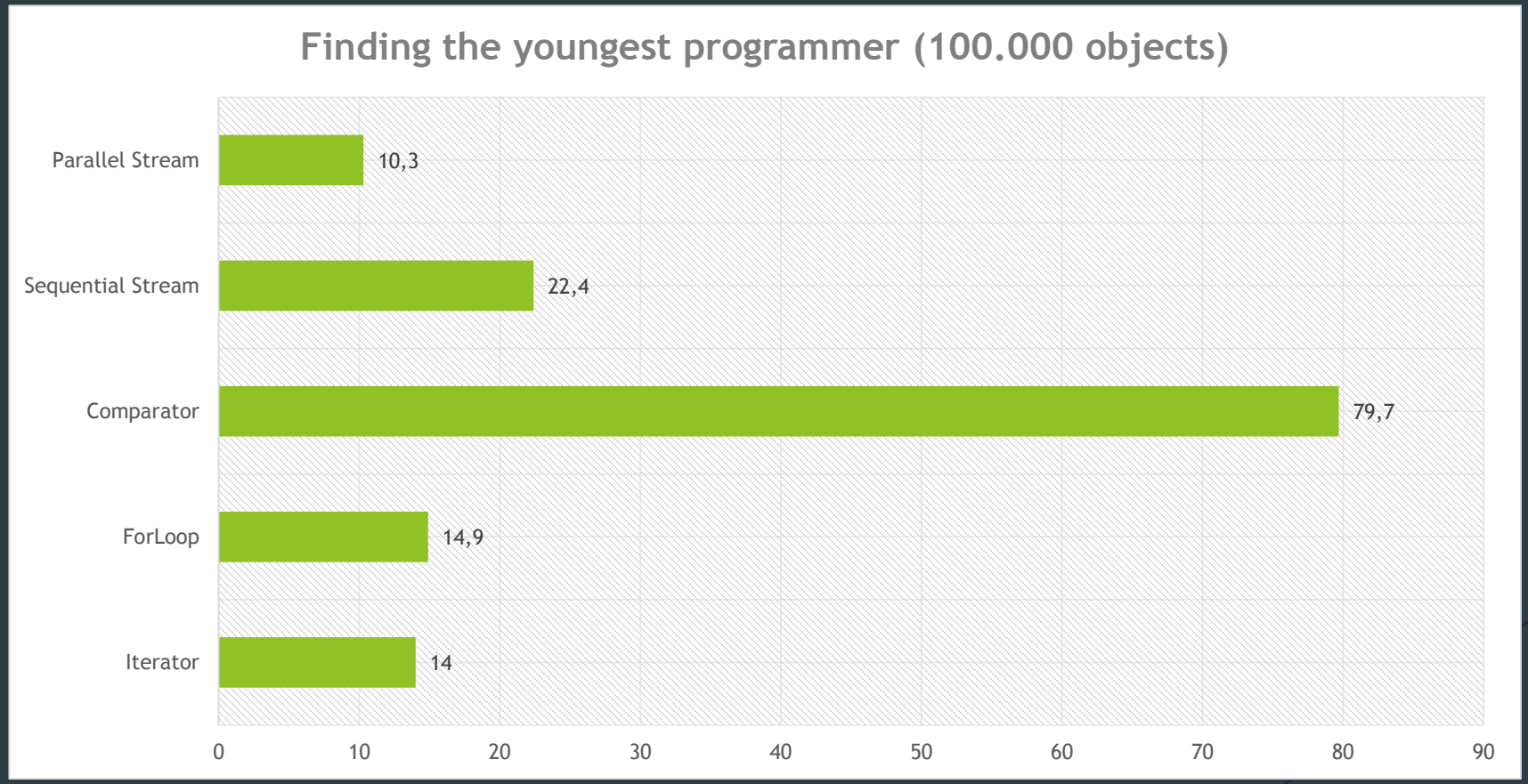
1. Case: finding the youngest programmer (10.000 objects)

- Results (ms):



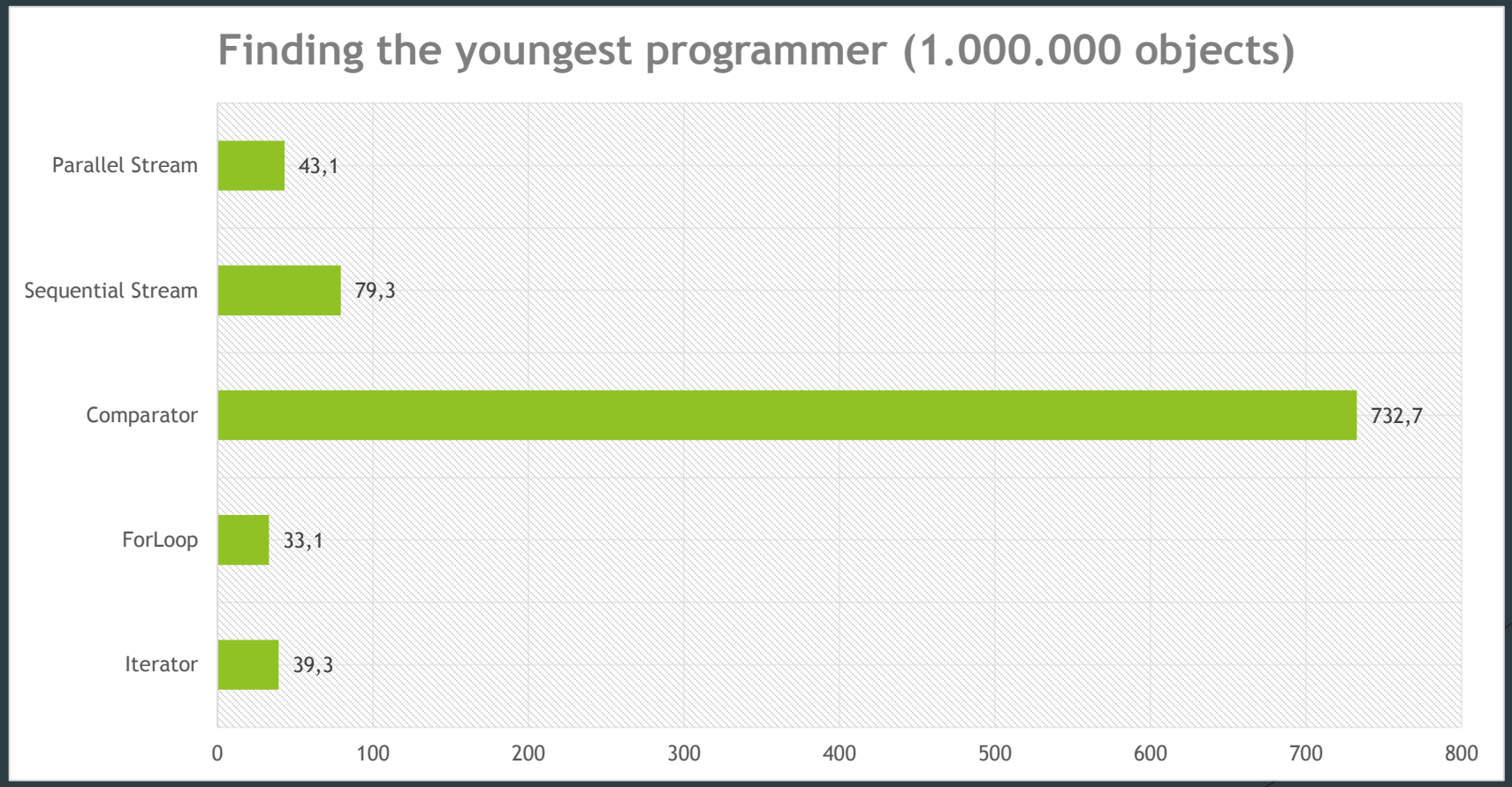
1. Case: finding the youngest programmer (100.000 objects)

- Results (ms):



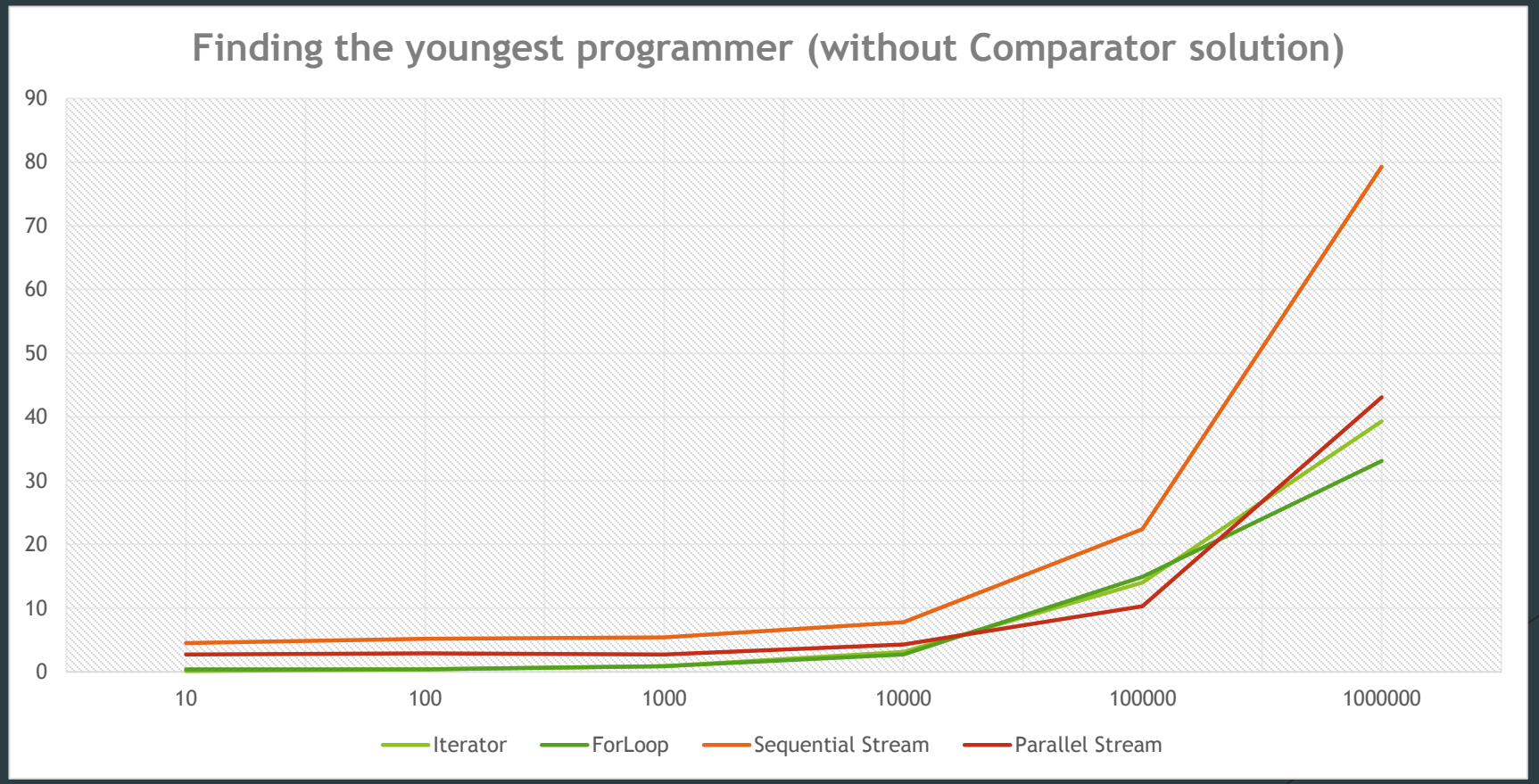
1. Case: finding the youngest programmer (1.000.000 objects)

- Results (ms):



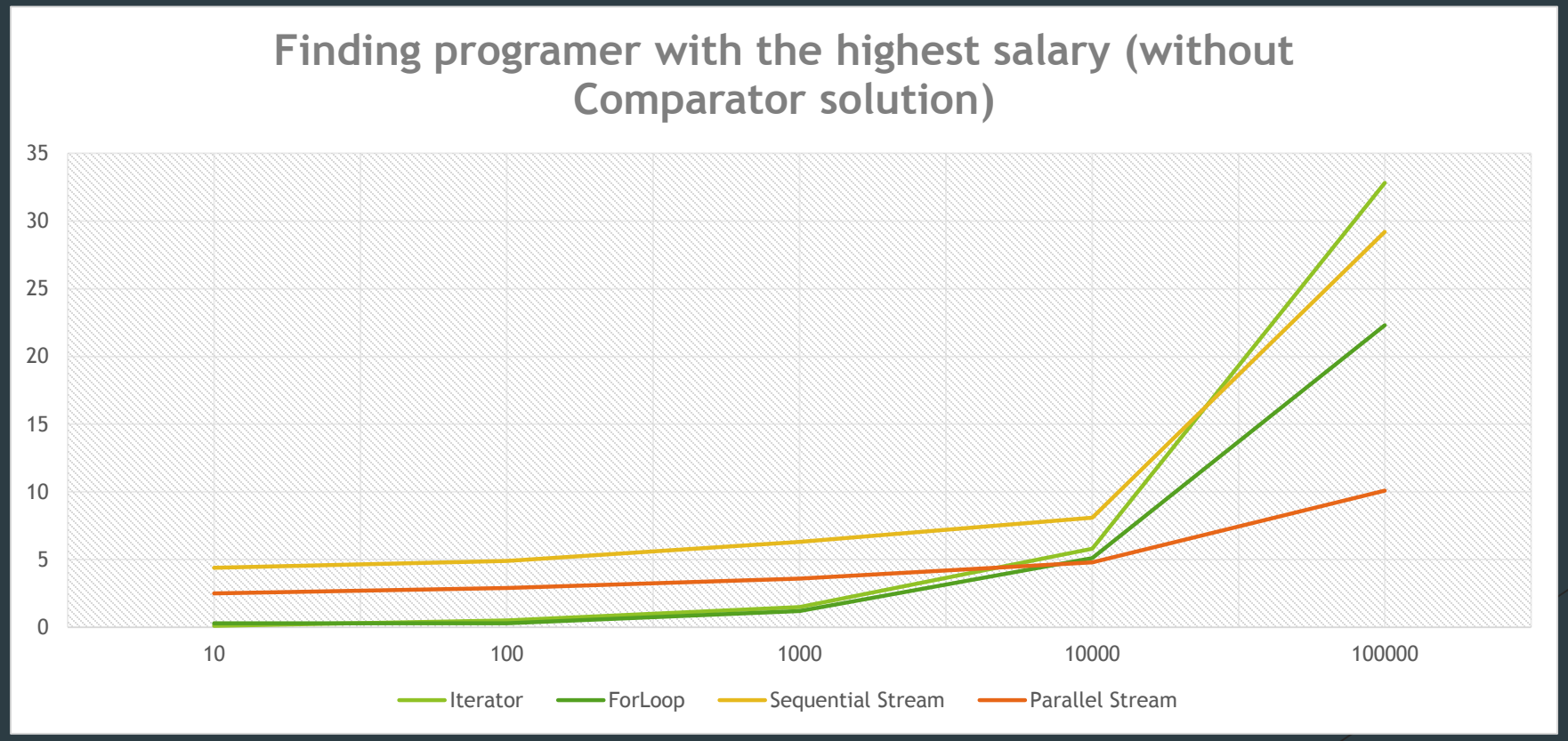
1. Case: finding the youngest programmer (summary)

- Results (ms):



2. Case: finding programmer with the highest salary (summary)

- Results (ms):



Filtering strategy implementations

```
Iterator<Programmer> iter = programmers.iterator();

while(iter.hasNext()) {
    Programmer newProgrammer = iter.next();
    if(newProgrammer.getLastName().contains(name)) {
        filteredProgrammers.add(newProgrammer);
    }
}


for(Programmer newProgrammer : programmers) {
    if(newProgrammer.getLastName().contains(name)) {
        filteredProgrammers.add(newProgrammer);
    }
}
```

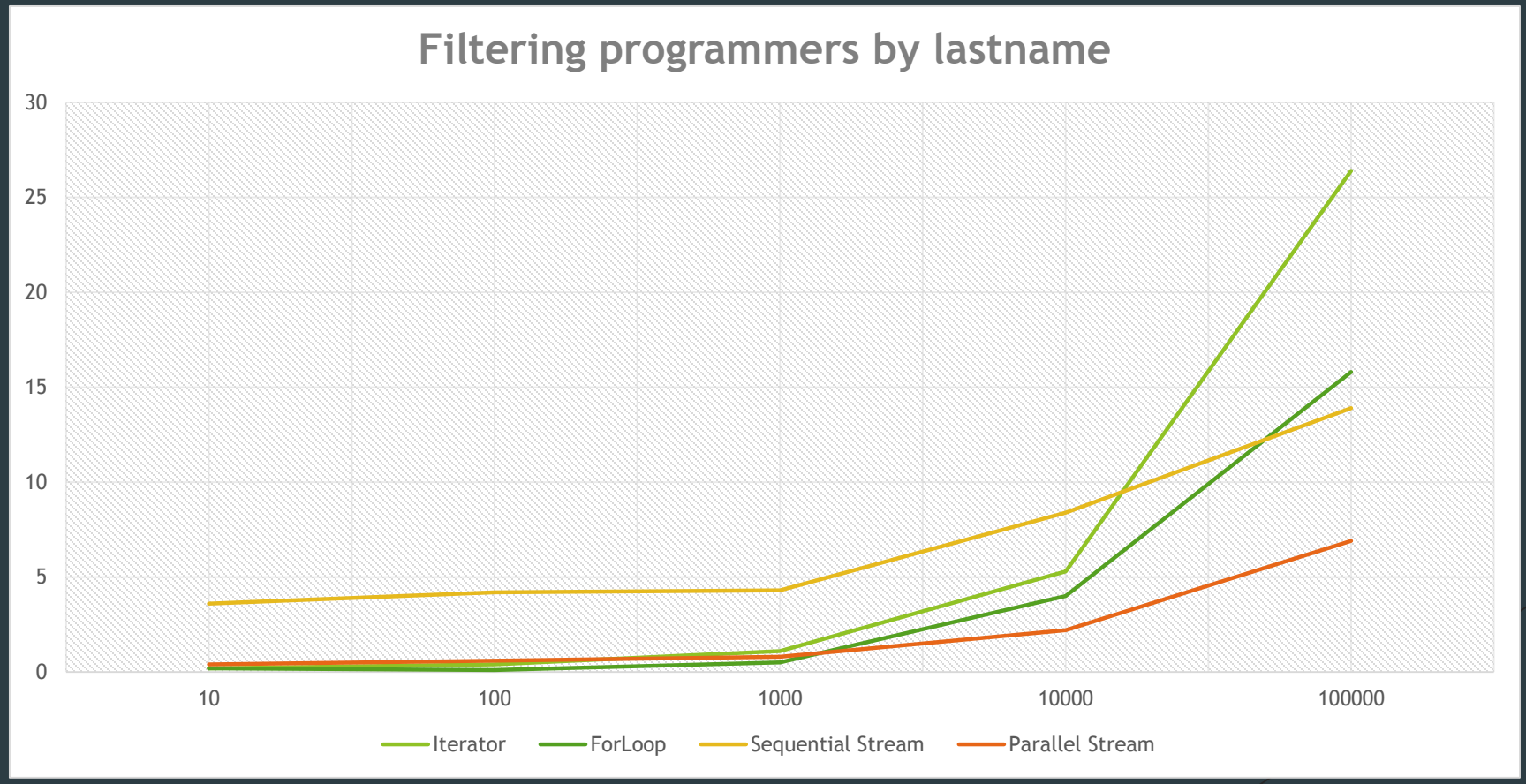
Filtering strategy implementations

```
List<Programmer> filteredProgrammers = programmers.stream().filter(p ->
   p.getLastName().contains(name)).collect(Collectors.toList());
```

```
List<Programmer> filteredProgrammers = programmers.parallelStream().filter(p ->
p.getLastName().contains(name)).collect(Collectors.toList());
```

3. Case: filtering programmers by lastname

- Results (ms):



Conclusions

- Stream API method not the best choice in every situation
- In case of arrays (instead of collections), the difference is more drastic
- With fewer objects (< 1000) the conventional (older) way is more appropriate
- The object type is also important (comparing BigDecimals, LocalDates…)
- The streams have certain amount of overhead
- But, „one-liner-programming” is fancy ☺

Thank you!