# Evaluating the Java Parallel ImageStreamGang Case Study

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

# Learning Objectives in this Part of the Lesson

- Understand purpose of the ImageStreamGang app
- Recognize patterns applied in the ImageStreamGang app
- Know how the structure of the ImageStreamGang app
- Visualize how Java parallel streams are applied to the ImageStreamGang app
- Learn how the parallel stream behaviors of ImageStreamGang are implemented
- Be aware of the pros & cons of the parallel streams solution

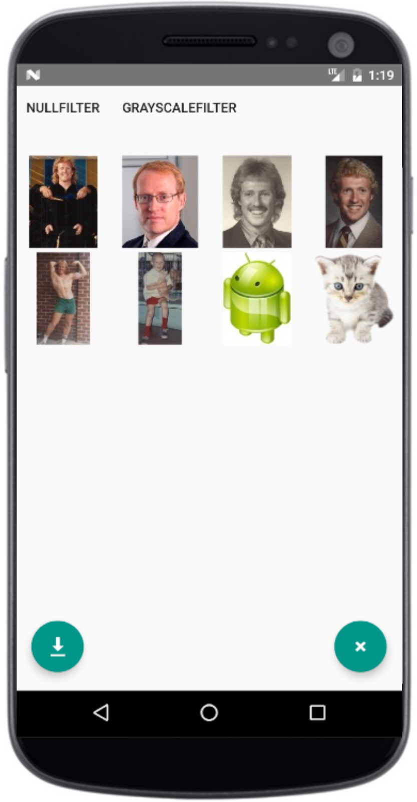
See [github.com/douglascraigschmidt/LiveLessons/blob/master/ImageStreamGang](https://github.com/douglascraigschmidt/LiveLessons/blob/master/ImageStreamGang)

# Pros of the Java Parallel Streams Solution

# Pros of the Java Parallel Streams Solution

- The parallel stream version is faster than the sequential streams version

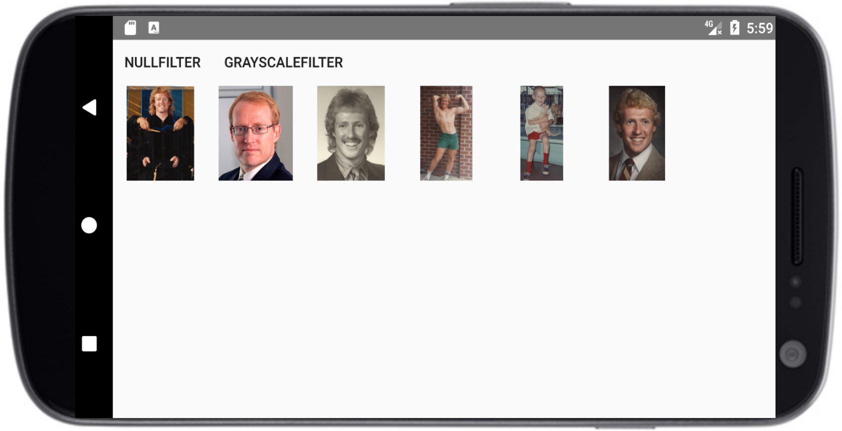
```
Starting ImageStreamGangTest
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 153 msecs
COMPLETABLE_FUTURES_1 executed in 251 msecs
PARALLEL_STREAM executed in 300 msecs
SEQUENTIAL_STREAM executed in 1026 msecs

Printing 4 results for input file 2 from fastest to slowest
PARALLEL_STREAM executed in 62 msecs
COMPLETABLE_FUTURES_1 executed in 68 msecs
COMPLETABLE_FUTURES_2 executed in 70 msecs
SEQUENTIAL_STREAM executed in 261 msecs
Ending ImageStreamGangTest
```

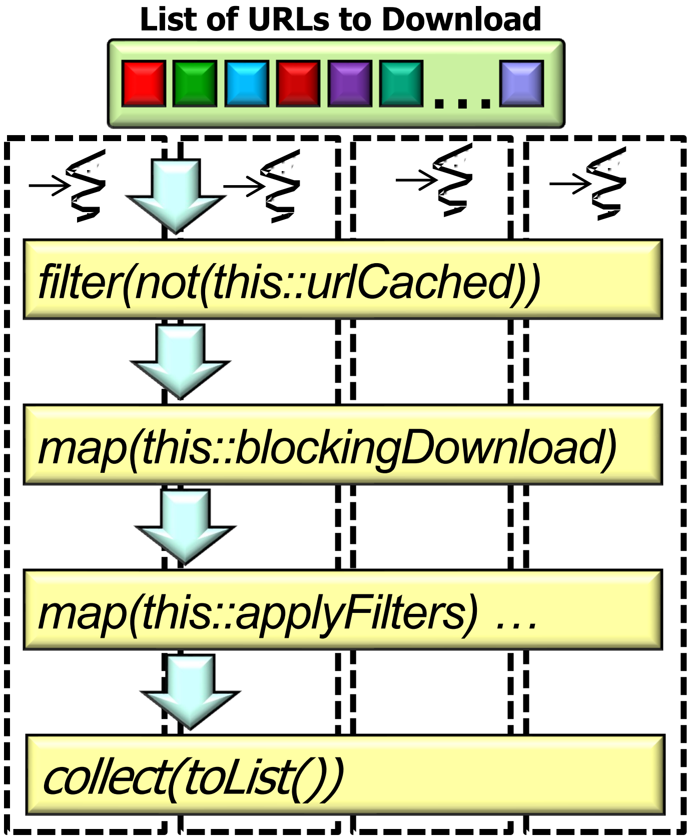


Tests conducted on a 2.4 GHz eight-core Lenovo P1 with 128 Gbytes of RAM

# Pros of the Java Parallel Streams Solution

- The parallel stream version is faster than the sequential streams version
  - e.g., images are downloaded & processed in parallel on multiple cores




List of URLs to Download

filter(not(this::urlCached))

map(this::blockingDownload)

map(this::applyFilters) …

collect(toList())

# Pros of the Java Parallel Streams Solution

- The solution is relatively straight forward to understand

```
void processStream() {
  List<Image> filteredImages =
  getInput()
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::blockingDownload)
    .map(this::applyFilters)
    .reduce(Stream::concat)
    .orElse(Stream.empty())
    .collect(toList());

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

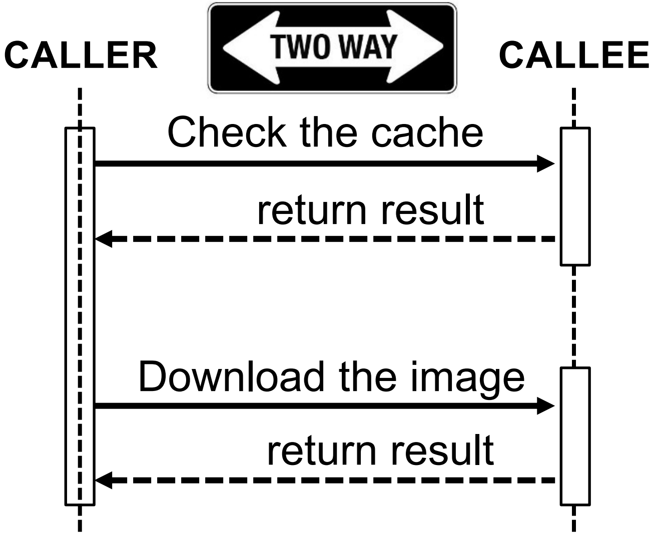
# Pros of the Java Parallel Streams Solution

- The solution is relatively straight forward to understand, e.g.
  - The behaviors map cleanly onto the domain intent

```
void processStream() {
  List<Image> filteredImages =
  getInput()
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::blockingDownload)
    .map(this::applyFilters)
    .reduce(Stream::concat)
    .orElse(Stream.empty())
    .collect(toList());

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

# Pros of the Java Parallel Streams Solution

- The solution is relatively straight forward to understand, e.g.
  - The behaviors map cleanly onto the domain intent
  - Behaviors are all synchronous



```java
void processStream() {
   List<Image> filteredImages =
   getInput()
      .parallelStream()
      .filter(not(this::urlCached))
      .map(this::blockingDownload)
      .map(this::applyFilters)
      .reduce(Stream::concat)
      .orElse(Stream.empty())
      .collect(toList());

   System.out.println(TAG
         + "Image(s) filtered = "
         + filteredImages.size());
}
```

See www.iro.umontreal.ca/~keller/Layla/remote.pdf

# Pros of the Java Parallel Streams Solution

- The solution is relatively straight forward to understand, e.g.
  - The behaviors map cleanly onto the domain intent
  - Behaviors are all synchronous
  - The flow of control can be read "linearly"
    - Parallel programming thus closely resembles sequential programming

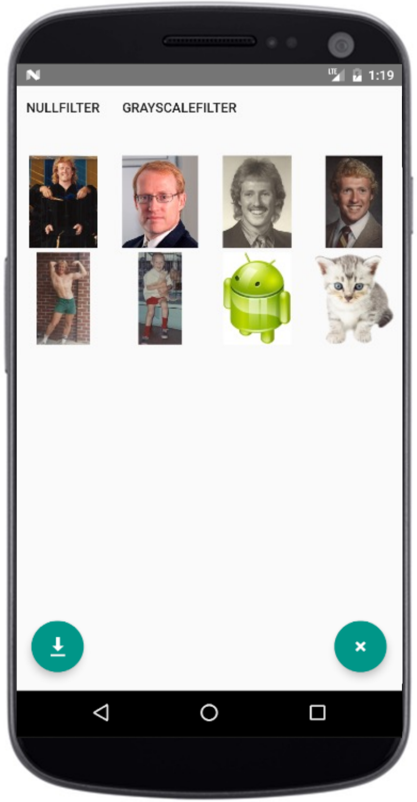
```
void processStream() {
  List<Image> filteredImages =
  getInput()
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::blockingDownload)
    .map(this::applyFilters)
    .reduce(Stream::concat)
    .orElse(Stream.empty())
    .collect(toList());

  System.out.println(TAG
        + "Image(s) filtered = "
        + filteredImages.size());
}
```

# Cons of the Java Parallel Streams Solution

# Cons of the Java Parallel Streams Solution

- Completable futures are sometimes faster than parallel streams

```
Starting ImageStreamGangTest
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 153 msecs
COMPLETABLE_FUTURES_1 executed in 251 msecs
PARALLEL_STREAM executed in 300 msecs
SEQUENTIAL_STREAM executed in 1026 msecs

Printing 4 results for input file 2 from fastest to slowest
PARALLEL_STREAM executed in 62 msecs
COMPLETABLE_FUTURES_1 executed in 68 msecs
COMPLETABLE_FUTURES_2 executed in 70 msecs
SEQUENTIAL_STREAM executed in 261 msecs
Ending ImageStreamGangTest
```



# Cons of the Java Parallel Streams Solution

- In general, there's a tradeoff between computing performance & programmer productivity when choosing amongst Java parallelism frameworks
  - i.e., completable futures are often more efficient & scalable than parallel streams, but are somewhat harder to program

Productivity
Performance

# End of Evaluating the Java Parallel ImageStreamGang Case Study