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 the structure/functionality of the ImageStreamGang app
- Visualize how Java parallel streams are applied to the ImageStreamGang app
- Learn how to implement parallel streams behaviors of ImageStreamGang
- Be aware of the pros & cons of the parallel streams solution

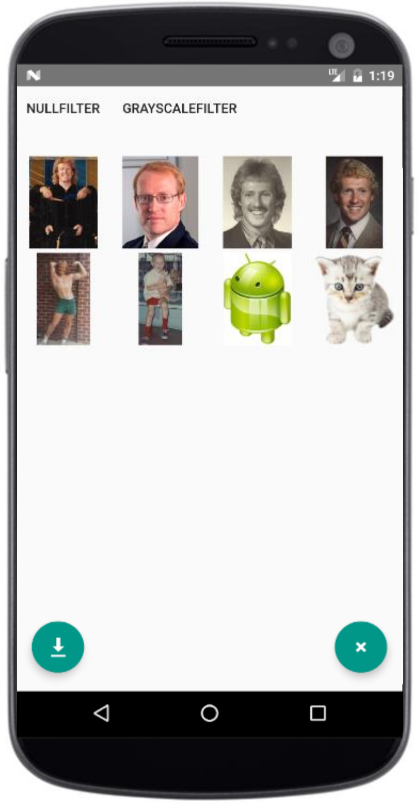
See github.com/douglascraigschmidt/LiveLessons/blob/master/ImageStreamGang

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

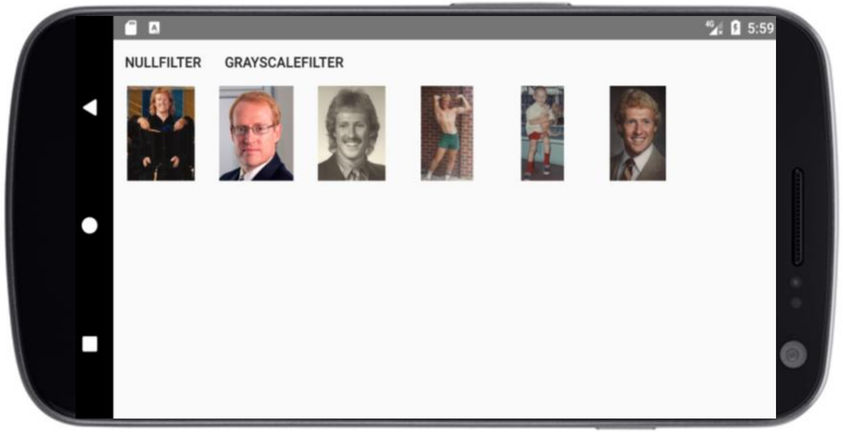
```
Starting ImageStreamGangTest
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 312 msecs
COMPLETABLE_FUTURES_2 executed in 335 msecs
PARALLEL_STREAM executed in 428 msecs
SEQUENTIAL_STREAM executed in 981 msecs

Printing 4 results for input file 2 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 82 msecs
COMPLETABLE_FUTURES_1 executed in 83 msecs
PARALLEL_STREAM executed in 102 msecs
SEQUENTIAL_STREAM executed in 251 msecs
Ending ImageStreamGangTest
```

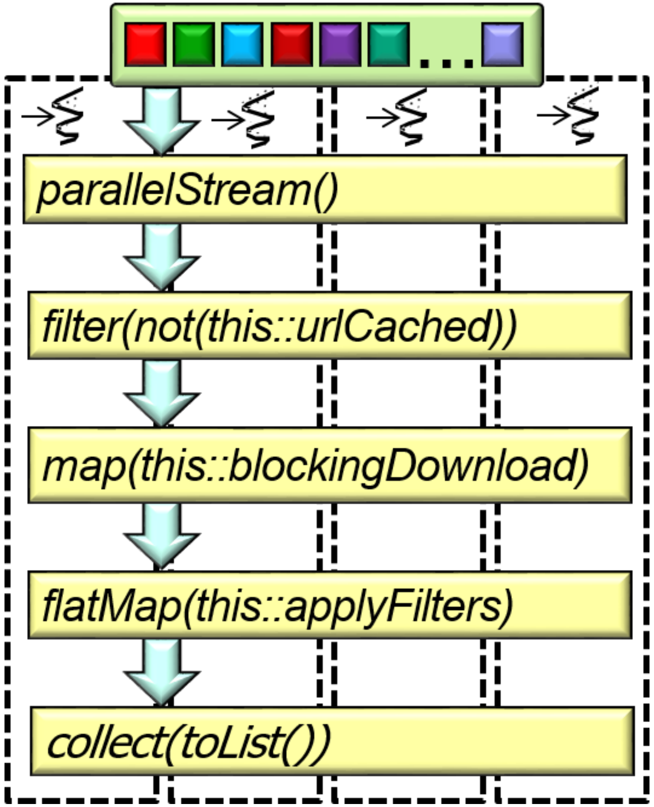


Six-core 2.6 Ghz Windows Intel computer with 64 GB RAM

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




parallelStream()
filter(not(this::urlCached))
map(this::blockingDownload)
flatMap(this::applyFilters)
collect(toList())

Pros of the Java Parallel Streams Solution

- The solution is relatively straight forward to understand

```
void processStream() {
  List<URL> urls = getInput();

  List<Image> filteredImages = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::blockingDownload)
    .flatMap(this::applyFilters)
    .collect(toList());

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

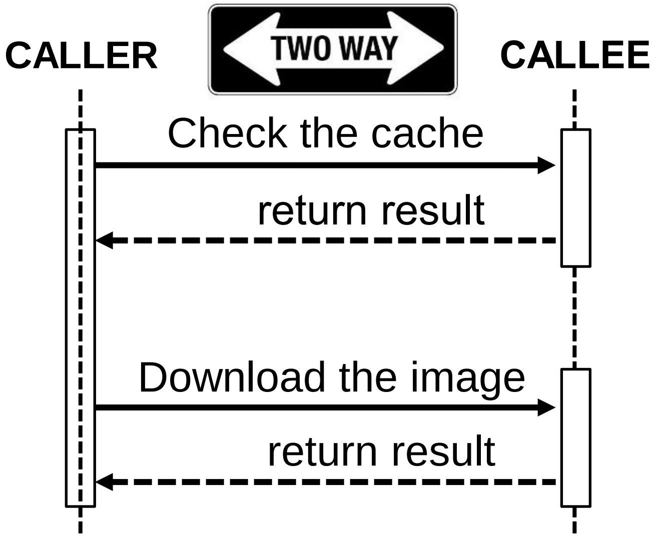
Pros of the Java Parallel Streams Solution

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

```
void processStream() {
  List<URL> urls = getInput();

  List<Image> filteredImages = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::blockingDownload)
    .flatMap(this::applyFilters)
    .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



```
void processStream() {
   List<URL> urls = getInput();

   List<Image> filteredImages = urls
     .parallelStream()
     .filter(not(this::urlCached))
     .map(this::blockingDownload)
     .flatMap(this::applyFilters)
     .collect(toList());

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

See

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

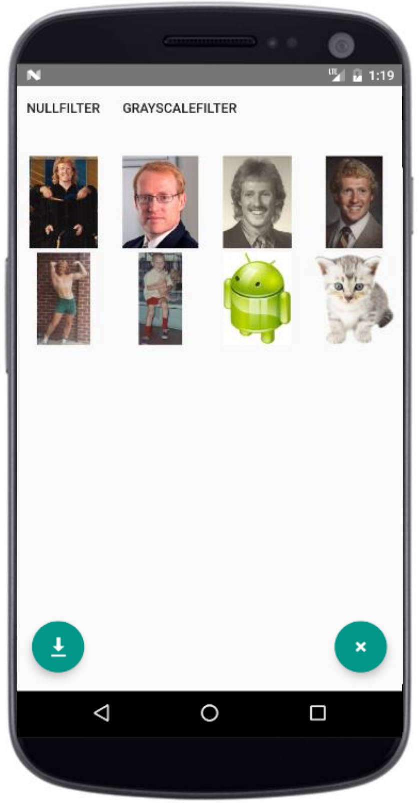
```
void processStream() {
  List<URL> urls = getInput();

  List<Image> filteredImages = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::blockingDownload)
    .flatMap(this::applyFilters)
    .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

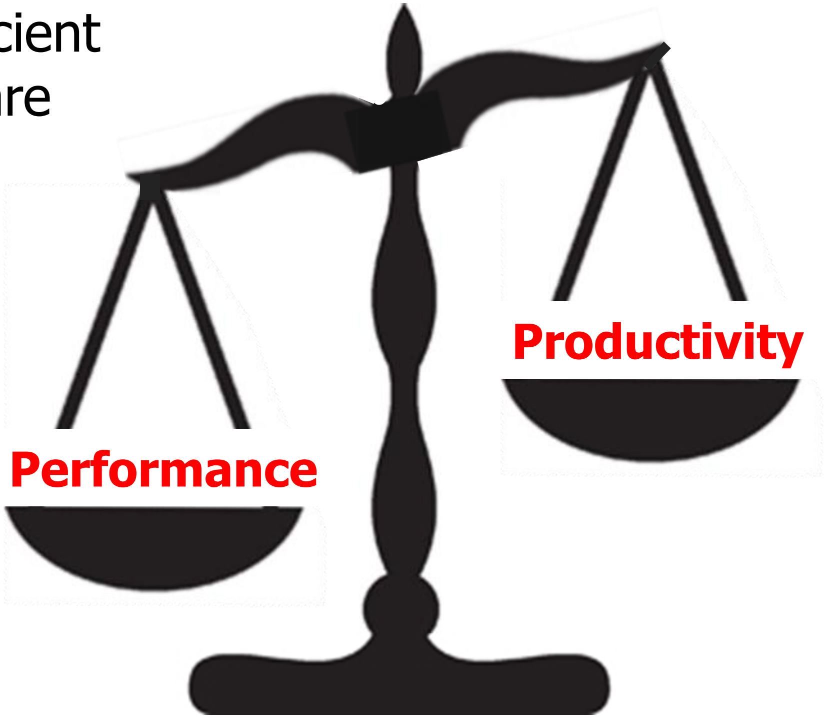
- The completable futures versions are faster than the parallel streams version

```
Starting ImageStreamGangTest
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 312 msecs
COMPLETABLE_FUTURES_2 executed in 335 msecs
PARALLEL_STREAM executed in 428 msecs
SEQUENTIAL_STREAM executed in 981 msecs

Printing 4 results for input file 2 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 82 msecs
COMPLETABLE_FUTURES_1 executed in 83 msecs
PARALLEL_STREAM executed in 102 msecs
SEQUENTIAL_STREAM executed in 251 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 more efficient & scalable than parallel streams, but are somewhat harder to program



End of Evaluating the Java Parallel ImageStreamGang Case Study