Java 8 Parallel ImageStreamGang Example (Part 3)

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

- Recognize the structure/functionality of the ImageStreamGang app
- Know how Java 8 parallel streams are applied to the ImageStreamGang app
- Understand the parallel streams implementation of ImageStreamGang

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

Learning Objectives in this Part of the Lesson

- Recognize the structure/functionality of the ImageStreamGang app
- Know how Java 8 parallel streams are applied to the ImageStreamGang app
- Understand the parallel streams implementation of ImageStreamGang
- Be aware of the pros & cons of the parallel streams solution

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

Implementing a Parallel Stream in ImageStreamGang

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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 imagestreamgang/streams/ImageStreamParallel.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

Get a list of URLs

```
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());
}
```

getInput() is defined by the underlying StreamGang framework

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

Convert a collection into a parallel stream

```
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());
}
```

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

Return an output stream consisting of the URLs in the input stream that are not already cached

```
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 docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#filter

Implementing a Parallel Stream in ImageStreamGang

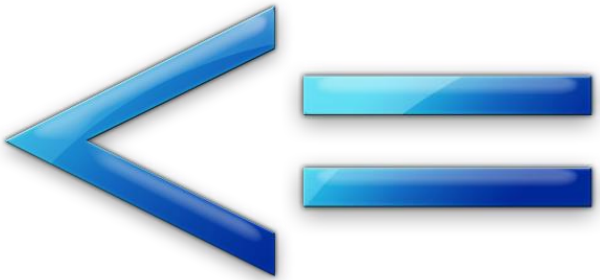
- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

Return an output stream consisting of the URLs in the input stream that are not already cached



# of output stream elements will be <= # of input stream elements

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

```
boolean urlCached(URL url) {
  return mFilters
    .stream()
    .filter(filter ->
       urlCached(url,
          filter.getName()))
    .count() > 0;
}
```

Determine whether this url has been downloaded to an image & had filters applied to it yet

See imagestreamgang/streams/ImageStreamGang.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

```
boolean urlCached(URL url,
          String filterName) {
  File file =
    new File(getPath(),
             filterName);

  File imageFile =
    new File(file,
          getNameForUrl(url));

  return imageFile.exists();
}
```

Check if a file with this name already exists

See imagestreamgang/streams/ImageStreamGang.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java


```
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());
}
```

There are clearly better ways of implementing an image cache!

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

Return an output stream consisting of the images that were downloaded from the URLs in the input stream

```
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 docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#map

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

Return an output stream consisting of the images that were downloaded from the URLs in the input stream

# of output stream elements must match the # of input stream elements

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

```
Image blockingDownload
                (URL url) {
  return BlockingTask
   .callInManagedBlocker
     (() ->
      downloadImage(url));
}
```

Downloads content from a url & converts it into an image

See imagestreamgang/streams/ImageStreamParallel.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

```
Image blockingDownload
                (URL url) {
  return BlockingTask
   .callInManagedBlocker
     (() ->
      downloadImage(url));
}
```

Uses a "managed blocker" to ensure sufficient threads are in the common fork-join pool

We covered BlockingTask.callInManagedBlocker() earlier in this course

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

```
Image blockingDownload
                (URL url) {
  return BlockingTask
   .callInManagedBlocker
     (() ->
      downloadImage(url));
}
```

I/O-bound tasks on an N-core CPU typically run best with N(1+WT/ST) threads (WT = wait time & ST = service time)*

See www.ibm.com/developerworks/library/j-jtp0730

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

Return an output stream containing the results of applying a list of filters to each image in the input stream & storing the results in the file system

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#flatMap

Implementing a Parallel Stream in ImageStreamGang

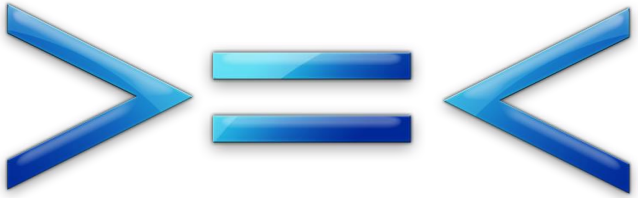
- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

Return an output stream containing the results of applying a list of filters to each image in the input stream & storing the results in the file system



# of output stream elements may differ from the # of input stream elements

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

```
Stream<Image> applyFilters
                    (Image image) {
  return mFilters
    .parallelStream()
    .map(filter ->
         makeFilterWithImage
          (filter,
           image).run())
}
```

Apply all filters to an image in parallel & store on the device

See imagestreamgang/streams/ImageStreamParallel.java

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

collect() is a "reduction" operation that combines elements into one result

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

Trigger all intermediate operations

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

Create a list containing all the filtered & stored images

Implementing a Parallel Stream in ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```
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());
}
```

Logs the # of images that were downloaded, filtered, & stored

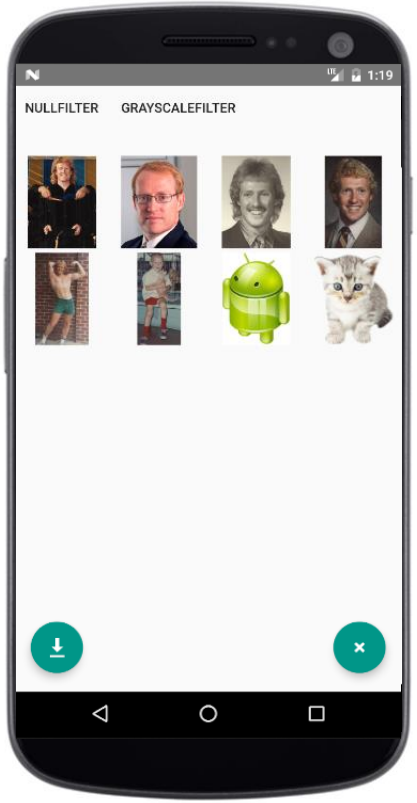
Pros of the Java 8 Parallel Streams Solution

Pros of the Java 8 Parallel Streams Solution

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

Starting ImageStreamGangTest
Printing results for input 1 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 276 msecs
COMPLETABLE_FUTURES_1 executed in 285 msecs
PARALLEL_STREAM executed in 383 msecs
SEQUENTIAL_STREAM executed in 1288 msecs

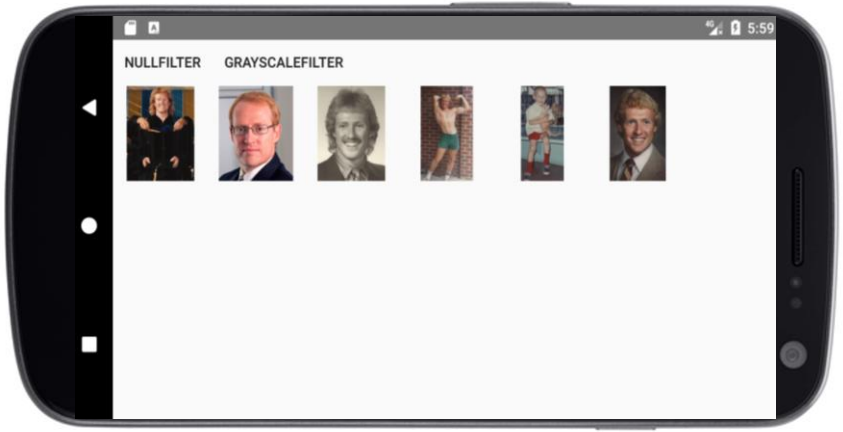
Printing results for input 2 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 137 msecs
COMPLETABLE_FUTURES_2 executed in 138 msecs
PARALLEL_STREAM executed in 170 msecs
SEQUENTIAL_STREAM executed in 393 msecs
Ending ImageStreamGangTest

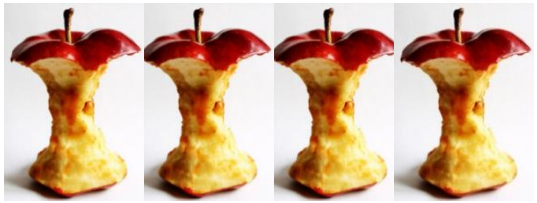


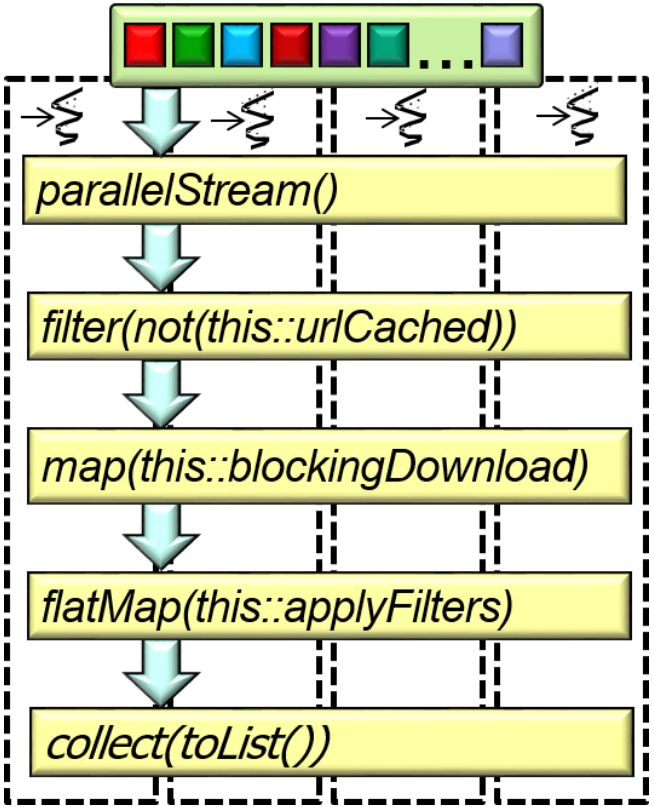
The performance speedup isn't quite linear on my quad-core computer

Pros of the Java 8 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







Pros of the Java 8 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 8 Parallel Streams Solution

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

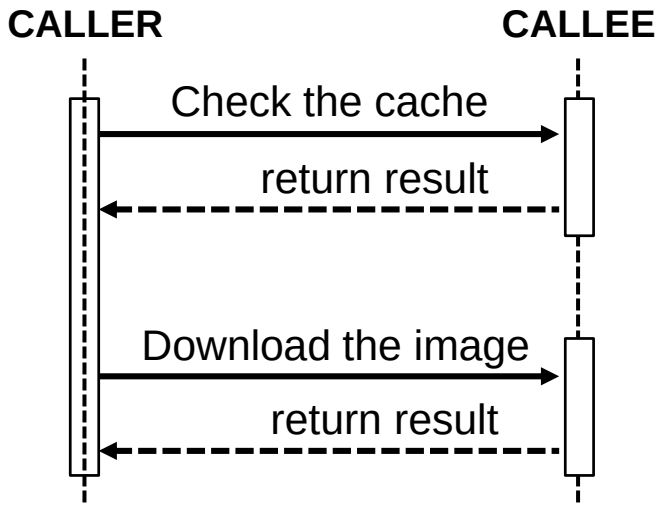

```
void processStream() {
   List<URL> urls = getInput();

   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 8 Parallel Streams Solution

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



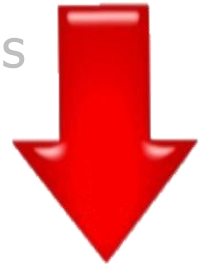

```
void processStream() {
   List<URL> urls = getInput();

   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());
}
```

Pros of the Java 8 Parallel Streams Solution

- The solution is relatively straight forward to understand, e.g.
  - The behaviors map cleanly onto the design intent
  - Behaviors are all synchronous
  - The flow of control can be read "linearly"



```
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());
}
```

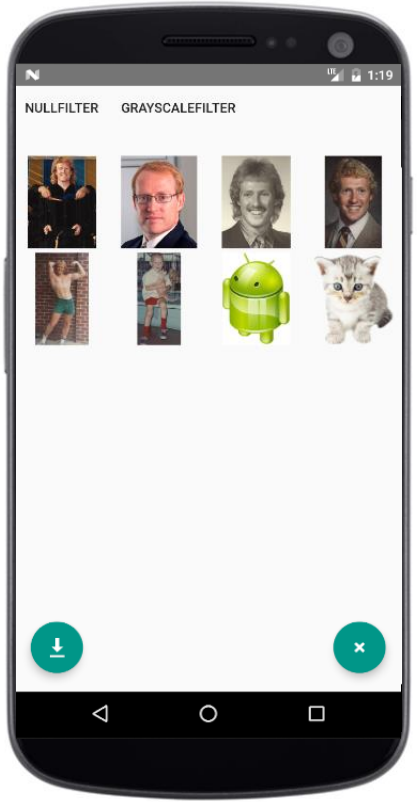
Cons of the Java 8 Parallel Streams Solution

Cons of the Java 8 Parallel Streams Solution

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

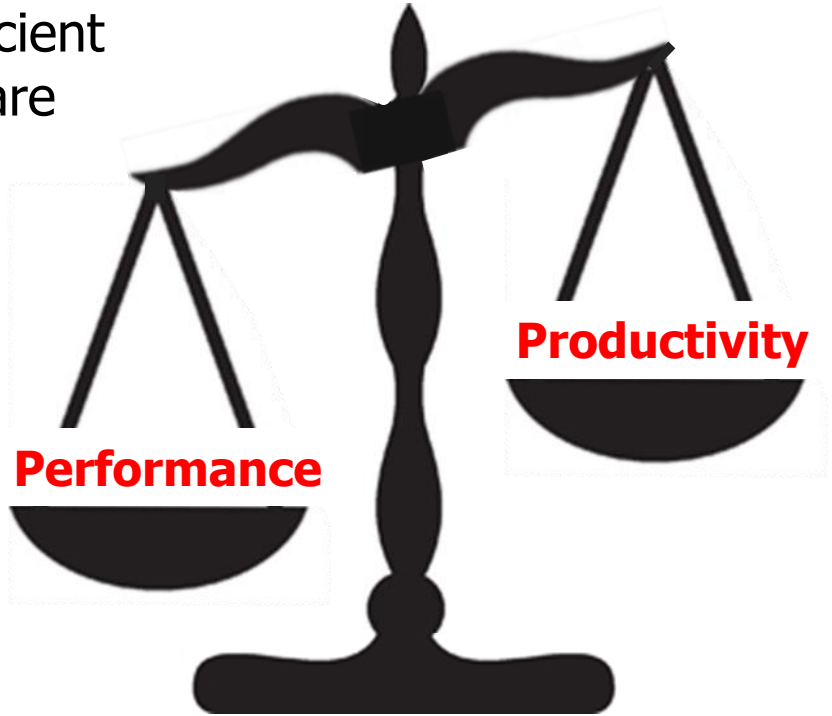
Starting ImageStreamGangTest
Printing results for input 1 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 276 msecs
COMPLETABLE_FUTURES_1 executed in 285 msecs
PARALLEL_STREAM executed in 383 msecs
SEQUENTIAL_STREAM executed in 1288 msecs

Printing results for input 2 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 137 msecs
COMPLETABLE_FUTURES_2 executed in 138 msecs
PARALLEL_STREAM executed in 170 msecs
SEQUENTIAL_STREAM executed in 393 msecs
Ending ImageStreamGangTest



Cons of the Java 8 Parallel Streams Solution

- In general, there's a tradeoff between computing performance & programmer productivity when choosing amongst Java 8 parallelism frameworks
  - i.e., completable futures are more efficient & scalable than parallel streams, but are somewhat harder to program



End of Java 8 Parallel ImageStreamGang Example (Part 3)