2.4 Lecture Summary

2 Functional Parallelism

2.4 Java Streams

Lecture Summary: In this lecture we learned about Java streams, and how they provide a functional approach to operating on collections of data. For example, the statement, "students.stream().forEach(s → System.out.println(s));", is a succinct way of specifying an action to be performed on each element s in the collection, *students*. An aggregate data query or data transformation can be specified by building a *stream pipeline* consisting of a *source* (typically by invoking the *.stream()* method on a data collection , a sequence of *intermediate operations* such as *map()* and *filter()*, and an optional *terminal operation* such as *forEach()* or *average()*. As an example, the following pipeline can be used to compute the average age of all active students using Java streams:

```
1  students.stream()
2      .filter(s -> s.getStatus() == Student.ACTIVE)
3      .mapToInt(a -> a.getAge())
4      .average();
```

From the viewpoint of this course, an important benefit of using Java streams when possible is that the pipeline can be made to execute in parallel by designating the source to be a *parallel stream*, i.e., by simply replacing *students.stream()* in the above code by *students.parallelStream()* or *Stream.of(students).parallel()*. This form of functional parallelism is a major convenience for the programmer, since they do not need to worry about explicitly allocating intermediate collections (e.g., a collection of all active students), or about ensuring that parallel accesses to data collections are properly synchronized.

Optional Reading:

1. Article on "Processing Data with Java SE 8 Streams"

2. Tutorial on specifying Aggregate Operations using Java streams

3. Documentation on java.util.stream.Collectors class for performing reductions on streams