2.1 Lecture Summary

---

2 Functional Parallelism

2.1 Future Tasks

Lecture Summary: In this lecture, we learned how to extend the concept of asynchronous tasks to *future tasks* and *future* objects (also known as *promise* objects). Future tasks are tasks with return values, and a future object is a "handle" for accessing a task's return value. There are two key operations that can be performed on a future object, *A*:

1. *Assignment* — *A* can be assigned a reference to a future object returned by a task of the form, *future* { ⟨ *task-with-return-value* ⟩ } (using pseudocode notation). The content of the future object is constrained to be *single assignment* (similar to a final variable in Java), and cannot be modified after the future task has returned.
2. *Blocking read* — the operation, *A.get()*, waits until the task associated with future object *A* has completed, and then propagates the task's return value as the value returned by *A.get()*. Any statement, S, executed after A.get() can be assured that the task associated with future object A must have completed before S starts execution.

These operations are carefully defined to avoid the possibility of a race condition on a task's return value, which is why futures are well suited for functional parallelism. In fact, one of the earliest use of futures for parallel computing was in an extension to Lisp known as MultiLisp.

Optional Reading:

1. Wikipedia article on Futures and promises.

2.2 Lecture Summary

---

2 Functional Parallelism

2.2 Creating Future Tasks in Java's Fork/Join Framework

Lecture Summary: In this lecture, we learned how to express future tasks in Java's Fork/Join (FJ) framework. Some key differences between future tasks and regular tasks in the FJ framework are as follows:

1. A future task extends the RecursiveTask class in the FJ framework, instead of RecursiveAction as in regular tasks.

2. The compute() method of a future task must have a non-void return type, whereas it has a void return type for regular tasks.

3. A method call like left.join() waits for the task referred to by object left in both cases, but also provides the task's return value in the case of future tasks.

Optional Reading:

1. Documentation on Java's RecursiveTask class

2.3 Lecture Summary

---

2 Functional Parallelism

2.3 Memoization

Lecture Summary: In this lecture, we learned the basic idea of "memoization", which is to remember results of function calls $f(x)$ as follows:

1. Create a data structure that stores the set $\{(x1, y1 = f(x1)), (x2, y2 = f(x2)), \ldots\}$ for each call $f(xi)$ that returns yi.
2. Perform look ups in that data structure when processing calls of the form $f(x')$ when x' equals one of the xi inputs for which $f(xi)$ has already been computed.

Memoization can be especially helpful for algorithms based on dynamic programming. In the lecture, we used Pascal's triangle as an illustrative example to motivate memoization.

The memoization pattern lends itself easily to parallelization using futures by modifying the memoized data structure to store $\{(x1, y1 = future(f(x1))), (x2, y2 = future(f(x2))), \ldots\}$. The lookup operation can then be replaced by a *get()* operation on the future value, if a future has already been created for the result of a given input.

Optional Reading:

1. Wikipedia article on Memoization.

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:

```
students.stream()
    .filter(s -> s.getStatus() == Student.ACTIVE)
    .mapToInt(a -> a.getAge())
    .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

2.5 Lecture Summary

---

2 Functional Parallelism

2.5 Determinism and Data Races

Lecture Summary: In this lecture, we studied the relationship between determinism and data races in parallel programs. A parallel program is said to be *functionally deterministic* if it always computes the same answer when given the same input, and *structurally deterministic* if it always computes the same computation graph, when given the same input. The presence of data races often leads to functional and/or structural nondeterminism because a parallel program with data races may exhibit different behaviors for the same input, depending on the relative scheduling and timing of memory accesses involved in a data race. In general, the absence of data races is not sufficient to guarantee determinism. However, all the parallel constructs introduced in this course ("Parallelism") were carefully selected to ensure the following *Determinism Property*:

If a parallel program is written using the constructs introduced in this course *and is guaranteed to never exhibit a data race*, then it must be both functionally and structurally deterministic.

Note that the determinism property states that all data-race-free parallel programs written using the constructs introduced in this course are guaranteed to be deterministic, but it does not imply that a

program with a data race must be functionally/structurally non-deterministic. Furthermore, there may be cases of “benign” nondeterminism for programs with data races in which different executions with the same input may generate different outputs, but all the outputs may be acceptable in the context of the application, e.g., different locations for a search pattern in a target string.

Optional Reading:

1. Wikipedia article on Race condition

Mini Project 2: Analyzing Student Statistics Using Java Parallel Streams

---

miniproject_2.zip

Project Goals & Outcomes

In lectures this week, you learned about a new programming construct added in Java 8: streams. Java streams offer a functional way to perform operations over Java Collections, and include basically useful operations such as filter, map, reduce, and scan.

Parallel Java streams automatically parallelize the functional operations described above. A parallel Java stream is created by simply applying the parallel() operation to an existing Java stream. Any stream operation applied to a Java parallel stream may be executed using multiple threads as long as its parallel execution does not break the semantics of the operation.

In Demo Video 2 of Week 2, Professor Sarkar showed an example of analyzing student data using both imperative loops and functional streams, and showed how the parallel stream version was not only faster to finish but also much less verbose and error-prone. It is recommended you review this demo video before starting this mini-project.

In this mini-project, we will explore in that direction further by implementing parallel stream versions of several other imperative data analysis programs. Throughout this assignment you should feel free to refer to any useful Javadocs.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

Once you have downloaded and unzipped the provided project files using the gray button labeled miniproject_2.zip at the top of this description, you should see the project source code at:

miniproject_2/src/main/java/edu/coursera/parallel/StudentAnalytics.java

and the project tests at

miniproject_2/src/test/java/edu/coursera/parallel/StudentAnalyticsTest.java.

Project Instructions

Your modifications should be done entirely inside of StudentAnalytics.java. You may not make any changes to the signatures of any public or protected methods inside of StudentAnalytics, or remove any of them. However, you are free to add any new methods you like. Any changes you make to StudentAnalyticsTest.java will be ignored in the final grading process.

Your main goals for this assignment are listed below. StudentAnalytics.java also contains helpful TODOs.

1. Implement StudentAnalytics.averageAgeOfEnrolledStudentsParallelStream to perform the same operations as averageAgeOfEnrolledStudentsImperative but using parallel streams.
2. Implement StudentAnalytics. mostCommonFirstNameOfInactiveStudentsParallelStream to perform the same operations as mostCommonFirstNameOfInactiveStudentsImperative but using parallel streams.
3. Implement StudentAnalytics. countNumberOfFailedStudentsOlderThan20ParallelStream to perform the same operations as countNumberOfFailedStudentsOlderThan20Imperative but using parallel streams. Note that any grade below a 65 is considered a failing grade for the purpose of this method.

You are free to refer to any online documentation that you find helpful. In particular, the documentation of the Java Stream class may be useful:

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Project Evaluation

Your assignment submission should consist of only the StudentAnalytics.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Your submission will be evaluated on Coursera's auto-grading system using 4 CPU cores. Note that the performance observed for tests on your local machine may differ from that on Coursera's auto-grading system, but that you will only be evaluated on the measured performance on Coursera. Also note that for all assignments in this course you are free to resubmit as many times as you like. See the Common Pitfalls page under Resources for more details. Please give it a few minutes to complete the grading. Once it has completed, you should see a score appear in the "Score" column of the "My submission" tab based on the following rubric:

- 10% – correctness of your averageAgeOfEnrolledStudentsParallelStream implementation
- 10% – performance of your averageAgeOfEnrolledStudentsParallelStream implementation on 4 cores
- 20% – correctness of your mostCommonFirstNameOfInactiveStudentsParallelStream implementation
- 20% – performance of your mostCommonFirstNameOfInactiveStudentsParallelStream implementation on 4 cores

- 20% – correctness of your countNumberOfFailedStudentsOlderThan20ParallelStream implementation
- 20% – performance of your countNumberOfFailedStudentsOlderThan20ParallelStream implementation on 4 cores