Lab 4: Java's ForkJoin Framework

Instructor: Mackale Joyner, Co-Instructor: Zoran Budimlić.

Course Wiki: http://comp322.rice.edu

Staff Email: comp322-staff@mailman.rice.edu

Goals for this lab

- Understand basics of Java's ForkJoin framework.
- Practice using Java's ForkJoin framework.

Downloads

As with previous labs, the provided template project is accessible through your private SVN repo at:

`https://svn.rice.edu/r/comp322/turnin/S19/NETID/lab_4`

For instructions on checking out this repo through IntelliJ or through the command-line, please see the Lab 1 handout. The below instructions will assume that you have already checked out the lab_4 folder, and that you have imported it as a Maven Project if you are using IntelliJ.

Note that for this lab we will be using standard Java features instead of HJlib. Therefore, there is no need to add the `-javaagent` command line argument in IntelliJ for running this lab's tests.

1 Summary of Java's ForkJoin framework

Java has a built-in ForkJoin framework which allows us to write parallel programs without the use of HJlib. For the purposes of this lab, you will only need to use and understand a few of the ForkJoin concepts:

- `ForkJoinPool`: a service that runs all the fork-join tasks in your program.
- `RecursiveTask<V>`: a fork-join task that returns a result of type `V`.
- `RecursiveAction`: a fork-join task that does not return a result.
- `invokeAll(Collection<T>)`: when passed a Collection of either `RecursiveTasks` or `RecursiveActions`, submits those tasks to the `ForkJoinPool` to run in parallel. Only returns once all parallel tasks have completed, similar to the `finish` construct in HJlib.
- `RecursiveTask.join()` waits for the completion of the task, then returns the result of the task's computation.
- `RecursiveAction.join()` waits for the completion of the task.

Additional examples of parallel programs using the ForkJoin framework are provided in the repository for reference (`ArraySum.java`, `ArrayDivide.java`, and `ArraySumFourWay.java`).

2 Parallelizing ReciprocalArraySum using Java's ForkJoin framework

In this lab, you will work on the Reciprocal Array Sum Problem, but use the standard Java ForkJoin framework instead of HJlib. You can refer to the Demonstration Video for Topic 1.1 for a refresher on Reciprocal Array Sum.

In this lab you will use "chunking", i.e. assigning the processing of multiple input data elements to a single task. While achieving the maximal theoretical parallelism of Reciprocal Array Sum would require assigning a single task to each input element, creating that many tasks on a real system would incur large overheads. Therefore, you are tasked below with chunking the processing of many input elements together in each task.

Your goals for this assignment are as follows:

1. Modify the `ReciprocalArraySumTask.compute()` method to implement the behavior of a single `ReciprocalArraySumTask`. This method should sequentially compute the reciprocal sum of values in the `input` array, from `startIndexInclusive` to `endIndexExclusive`. The sum should be saved in `value`.

   If your implementation of `compute()` is correct, both `testParSimple` tests in `ReciprocalArraySumPerformanceTest` should pass.

2. Modify the `ReciprocalArraySum.parManyTaskArraySum()` method to implement the reciprocal-array-sum computation in parallel using Java's ForkJoin framework, using a given number of tasks. You should first read and understand the provided implementation of `ReciprocalArraySum.parArraySum()`, which uses only two tasks. Base your code off of `parArraySum()`, since the ForkJoin contructs used are the same. Note that the `getChunkStartInclusive` and `getChunkEndExclusive` utility methods are provided for your convenience to help with calculating the region of the input array a certain task should process. If your implementation of `parManyTaskArraySum()` is correct, both `testParManyTask` tests should also now pass.

If you encounter an error saying `java.lang.OutOfMemoryError: Java heap space` on your local machine, you should try testing on the autograder. Your laptop may not have sufficient memory to run all the tests.

3 Parallelizing N-Queens using Java's ForkJoin framework

This week we will revisit the simple N-Queens problem (*i.e.,* how can we place N queens on an N $\times$ N chessboard so that no two queens can capture each other?) introduced in Lecture 7 and in Lab 3. You will modify the `NQueensForkJoin.compute()` method, which should use Java's ForkJoin Framework to parallelize the N-Queens computation. The sequential pieces of the code have been provided, and there are `TODO`s guiding you on where to place your edits. In IntelliJ, you can automatically find all TODOs by going to View > Tool Windows > TODO. You can test your implementation by running the tests in `NQueensForkJoinPerformanceTest.java`.

4 Demonstrating and submitting in your lab work

For this lab, you will need to demonstrate and submit your work before leaving, as follows.

1. Show your work to an instructor or TA to get credit for this lab. They will want to see your files submitted to Subversion in your web browser and the passing unit tests on your laptop or on the autograder.

2. Check that all the work for today's lab is in your `lab_4` directory by opening https://svn.rice.edu/r/comp322/turnin/S19/NETID/lab_4/ in your web browser and checking that your changes have appeared.