Lab 4: Java's ForkJoin Framework

Instructor: Vivek Sarkar, Co-Instructor: Mackale Joyner

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

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

Goals for this lab

- Understand how to use Java's ForkJoin framework.
- Understand the difference between RecursiveAction and RecursiveTask declarations.

Downloads

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

```
https://svn.rice.edu/r/comp322/turnin/S17/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 Parallelizing ReciprocalArraySum using RecursiveAction's in Java's ForkJoin framework

In Lab 1, you got your first hands-on experience with writing parallel code by optimizing the Reciprocal Array Sum problem using HJlib. In this lab, you will work on a similar problem but use the standard Java Fork-Join framework instead. You can refer to the Demonstration Video for Topic 1.1 for a refresher on Reciprocal Array Sum.

In this exercise 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, below you are tasked with chunking the processing of many input elements together in each task for a simple two-task implementation and a more general many-task implementation.

Your modifications should be made entirely inside of ReciprocalArraySum.java. You should not change the signatures of any public or protected methods inside of ReciprocalArraySum, but you can edit the method bodies and add any new methods that you choose. We will use our copy of ReciprocalArraySumTest.java in the final grading process, so do not change that file or any other file except ReciprocalArraySum.java.

Your main goals for this assignment are as follows:

1. Modify the `ReciprocalArraySum.parArraySum()` method to implement the reciprocal-array-sum computation in parallel using the Java Fork Join framework by partitioning the input array in half and computing the reciprocal array sum on the first and second half in parallel, before combining the

results. There are TODOs in the source file to guide you, and you are free to refer to the lectures and HJlib documentation. 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.

2. Modify the `ReciprocalArraySum.parManyTaskArraySum()` method to implement the reciprocal-array-sum computation in parallel using Java's Fork Join framework again, but using a given number of tasks (not just two). 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 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 tests.

2 Parallelizing N-Queens using RecursiveTask's in 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 last week in Lecture 7 and in Lab 3. Like in Lab 3, you will be expected to use the cutoff strategy with a given cutoff to implement NQueens efficiently. For a refresher on what the cutoff strategy and how it benefits real world performance, refer to Section 2.2 of Lab 3. You will edit the `NQueensForkJoin.java` file provided in your svn repository for this exercise. There are `TODO`s in this file guiding you on where to place your edits. In IntelliJ, you can automatically find all TODOs by going to View > Tool Windows > TODO.

The goal of this exercise is to create a parallel version of NQueens using RecursiveTask's in Java's ForkJoin framework, and observe its execution times. We have provided the `ArrayDivide.java`, `ArraySum.java` and `ArraySumFourWay.java` files as examples of how to use the Fork/Join framework in Java. You are not expected to need to execute these files. Your changes for this exercise should go in `NQueensForkJoinTask.compute` and `NQueensForkJoin.findQueens`.

3 A Note on Real Performance (Recap from Lab 3)

In this lab, we only measure real multi-threaded performance, running on your laptops and on the autograder. (Note that there is no support for abstract metrics in Java's ForkJoin framework.) Real performance is messier than abstract performance because it is affected by its environment. Many students will have varying hardware, varying software, and possibly other applications running at the same time using up most of the cores. Even your laptop's power manager can throttle the number of hardware cores your HJlib program gets to use, limiting your speedup. In future homeworks and labs, we will run on a Rice compute cluster to eliminate this issue, ensuring that there is no interference with your running multi-threaded programs. For tonight, if you do not see any speedup in real performance on some or all tests you can try either 1) closing down any other expensive applications that might be running, 2) plugging in your laptop, 3) double-checking your implementation of chunking and cutoffs to ensure you are not spawning massive numbers of tasks, or 4) using the autograder to automatically run your experiments on a compute cluster. Feel free to call over a TA for help, they can help judge if the problem is in the code or the environment. You will not be penalized if your code fails some of the performance tests due to environmental factors.

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/S17/NETID/lab_4/` in your web browser and checking that your changes have appeared.