Lab 2: Futures

Instructors: Mackale Joyner, Zoran Budimlić

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

Staff Email: comp322-staff@rice.edu

Goals for this lab

- Experiment with functional programming and futures, including the `future` API

Downloads

As with Lab 1, the provided template project is accessible through your private SVN repo at:

`https://svn.rice.edu/r/comp322/turnin/S18/NETID/lab_2`

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_2 folder, and that you have imported it as a Maven Project if you are using IntelliJ.

1 Getting Familiar with Futures

You can think of futures as an async with a return value. Like an async, the logic associated with a future takes place asynchronously, not necessarily when the future is created. Different from an async, however, is that future logic returns a value, which is of course only accessible after the future has completed. The value can be queried using the `HjFuture` object returned from a call to `future` by calling `get` on it. `HjFuture.get` will block the calling task until the corresponding future task has completed, and then return its value.

1.1 Checking a Binary Tree With Futures

In this exercise, you are given sequential code for a functional program that constructs a binary tree, and then traverses it using calls to `itemCheck()`. Your task is to convert the top-down traversal in `itemCheck()` to a correctly executing parallel HJ-lib program with futures. Once correctly implemented, your code should pass the provided `BinaryTreeCorrectnessTest`. Think back on your previous experience with functional programming, and your knowledge of futures and how they relate to your task.

1. Compile and run the original `BinaryTreeCorrectnessTest` program and note that it fails.
2. Now modify the provided `TreeNode` class under `src/main/java/edu/rice/comp322/` to perform the `itemCheck` traversal in parallel using futures. A correct parallel implementation should produce the same WORK but a different CPL, thereby passing the tests in `BinaryTreeCorrectnessTest`.

An autograder module for this lab is provided to help with testing your `TreeNode` solution.

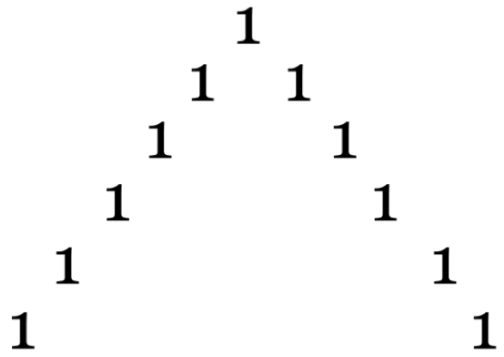

Figure 1: An initialized Pascal's triangle for N = 6.

1.2 Pascal's Triangle With Futures

Pascal's Triangle is a recursive algorithm that can be visualized as follows. In the initial step, we create a triangle of integers and initialize all border entries to one. We say that this triangle has `N` rows, and that each row has `K` columns (where `N` is fixed but `K` varies by row, where `K` for row `n` is `n + 1`). Rows and columns are numbered starting at zero. Figure 1 depicts an initialized Pascal's Triangle.

To fill in each empty cell of the triangle, we sum the values to its top left and top right. For example, to compute the the element for `n = 2` and `k = 1` we would sum the values stored at (1, 0) and (1, 1). Figure 2 depicts a Pascal's Triangle with element (2, 1) filled in.

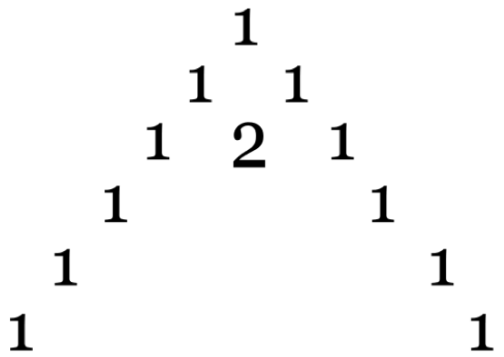

Figure 2: An example of filling in element (2, 1) for a Pascal's triangle with N = 6.

Applying this algorithm recursively by row would produce the complete triangle in Figure 3.

In this lab, you will need to edit `PascalsTriangle.java` to produce a correct parallel solution using futures. In `PascalsTriangle.java` you will find a reference sequential version which you can use `HjFuture` and the `future` API to parallelize. You must ensure that a call do `doWork(1)` is made for each addition of two parent nodes to calculate a child node's value. Running `PascalsTriangleCorrectnessTest.java` will verify the correctness and abstract performance of your solution.

An autograder module for this lab is provided to help with testing your `PascalsTriangle` solution.

Turning in your lab work

For each lab, you will need to turn in your work before leaving, as follows.

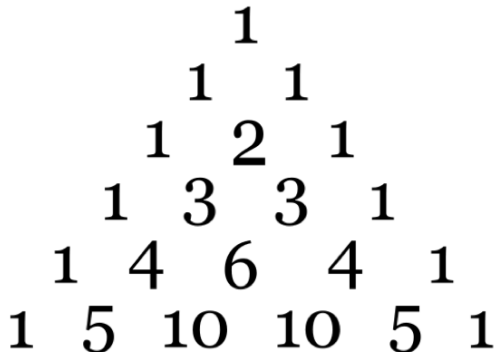

Figure 3: A complete Pascal's Triangle for N = 6.

1. Show your work (passing tests, preferably on the autograder) to an instructor or TA to get credit for this lab. Be prepared to explain the lab at a high level, as well as answer the following question for the Binary Tree and Pascal's Triangle programs:

   - What was your strategy in rewriting the provided sequential code as a parallel one? How did functional programming aid you in this pursuit?

2. Commit your code to SVN.