Lab 2: Futures and HJ-Viz

Instructors: Vivek Sarkar, Mackale Joyner

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
- Learn how to use HJ-Viz to visualize Computation Graphs (CGs) for small inputs

Downloads

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

`https://svn.rice.edu/r/comp322/turnin/S17/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. That 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.

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.

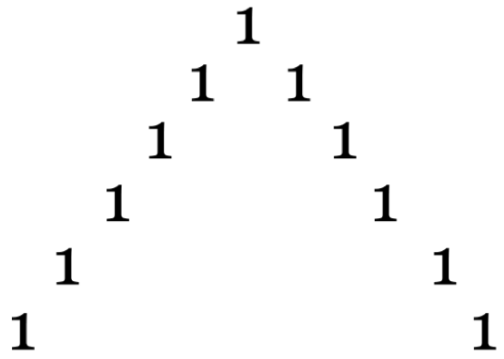

Figure 1: An initialized Pascal's triangle for N = 6.

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.

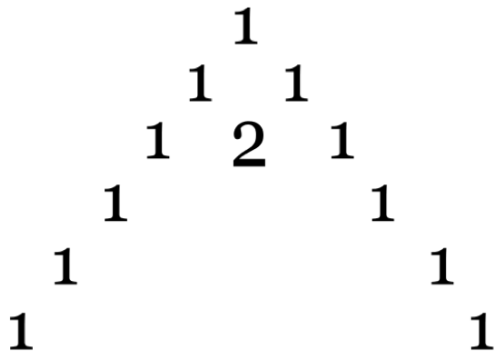

Figure 2: An example of filling in element (2, 1) for a Pascal's triangle with N = 6.

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.

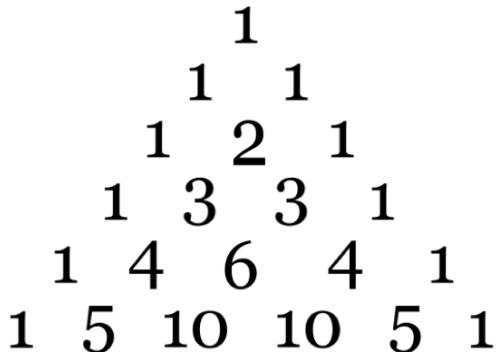

Figure 3: A complete Pascal's Triangle for N = 6.

2 Visualize Parallelism using HJ-Viz

2.1 Prerequisites

- Python 2.7 (http://python.org/) – check that you are using version 2.7 since HJ-Viz may not work with other versions of Python
- GraphViz (http://graphviz.org/)

If you're on Linux, we suggest using your package manager. Homebrew and macports on Mac have the software as well. For GraphViz:

- Windows: `http://graphviz.org/pub/graphviz/stable/windows/graphviz-2.38.msi`
- Mac: `http://www.graphviz.org/pub/graphviz/stable/macos/mountainlion/graphviz-2.36.0.pkg`

2.2 Assignment

HJ-Viz is a tool to generate interactive Computation Graphs (CGs) of parallel programs by analyzing event logs. Programmers can use the visualization of the CG by HJ-Viz to pinpoint potential sources of bugs and points of improvement for parallel performance by highlighting the critical path.

The goal of tonight's lab is to write a program whose output graph matches Figure 4 below, and to verify its correctness using this tool.

We have provided a skeleton file, `AsyncFinish.java` in your turnin repository as well.

- Use only finish and async constructs to write a program in `AsyncFinish.generateSameGraph` whose output through HJ-Viz matches Figure 4. This may require some trial and error. One possible first step is to leave the `generateSameGraph` method empty to see the graph generated by an empty program, and then work from there.
- On running `AsyncFinishTest` (provided under `src/test/java/edu/rice/comp322`) the code should generate an output.log file.
- You should have a provided HJ-Viz folder with all the required files to run HJ-Viz inside of lab_2:
  1. HJ-Viz is a Python program. Execute 'python main.py <path/to/the/log/file >', where logfile is the path to the .log file produced in the previous step.
  2. On Windows, you should be able to drag and drop the log file onto main.py in the file explorer.

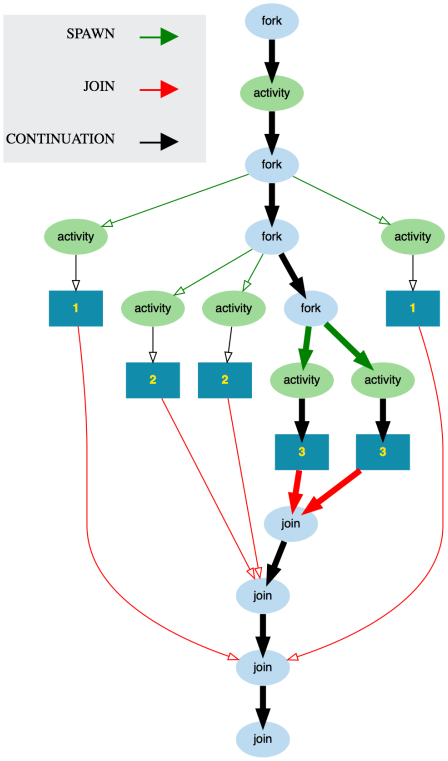

Figure 4: Sample output from HJ-Viz. `fork` nodes indicate the start of a finish scope. `join` nodes correspond to the end of a finish scope. `activity` nodes refer to the start of an async task. Square nodes labeled by a number x indicate calls to `doWork(`x`)`. Green, red, and black edges represent spawn, join, and continue edges. Bold edges belong to one or more critical paths.

3. The graph output is then produced in output.html in the same directory as output.log, and is viewable in any web browser. Since HJ-Viz is just a visualization tool, there is no autograder testing support provided for this part of the lab.

   When you view output.html, you can look out for the following features of HJ-Viz:

   (a) Click on the Graph tab (without checking the overlay box for now) to see the *computation graph* (this should be the default view when you open the output.html file). Note that all critical paths in the graph are indicated via thick arrows.

   (b) Click on the Legend tab to see a *legend* indicating the different kinds of edges according to their colors. Thus far, we have only learned about continue, spawn, and join edges; we will learn about the other edges later in the semester.

   (c) (Optional) Click on the Chart tab to see the *parallelism profile* of the computation graph. For a specified number of maximum processors, the chart shows time on the horizontal axis and the number of processors that can be used at a specific point in time on the vertical axis.

4. If you would like to run HJ-Viz on any other HJlib program, you just need to insert the calls to "`HjSystemProperty.eventLogging.setProperty(true);`" and
   " `ModuleEventLogging.dumpEventLog("output.log");`" as shown in `AsyncFinishTest.java`. Keep in mind that large program executions can generate very large computation graphs.

Turning in your lab work

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

1. Show your work to an instructor or TA to get credit for this lab. Be prepared to explain the lab at a high level, to show a correct computation graph from HJviz, 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.