Lab 8: Java Threads

Instructor: Vivek Sarkar, Co-Instructor: Shams Imam

Resource Summary

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

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

Important tips and links:

edX site : `https://edge.edx.org/courses/RiceX/COMP322/1T2014R`

Piazza site : `https://piazza.com/rice/spring2015/comp322/home`

Java 8 Download : `https://jdk8.java.net/download.html`

Maven Download : `http://maven.apache.org/download.cgi`

IntelliJ IDEA : `http://www.jetbrains.com/idea/download/`

HJ-lib Jar File : `http://www.cs.rice.edu/~vs3/hjlib/code/maven-repo/edu/rice/hjlib-cooperative/0.1.5-SNAPSHOT/hjlib-cooperative-0.1.5-SNAPSHOT.jar`

HJ-lib API Documentation : `https://wiki.rice.edu/confluence/display/PARPROG/API+Documentation`

HelloWorld Project : `https://wiki.rice.edu/confluence/pages/viewpage.action?pageId=14433124`

1 Lab Goal

In today's lab you will practice using Java Threads.

The Maven project for this lab is located in the following svn repository:

- `https://svn.rice.edu/r/comp322/turnin/S16/`*NETID*`/lab_8`

Use the subversion command-line client to checkout the project into appropriate directories locally. For example, you can use the following commands from a shell:

```
$ cd ~/comp322
$ svn checkout https://svn.rice.edu/r/comp322/turnin/S16/NETID/lab_8
```

In today's lab, you need to use NOTS to run performance tests. If you need any guidance on how to submit jobs on NOTS manually or through the autograder, please refer to earlier labs or ask a member of the teaching staff.

2 Conversion to Java Threads: N-Queens

1. The `NQueensSeq.java` program is a sequential solution to the N-Queens problem. The `NQueensHjLib.java` program has been provided to you as an example parallel solution to the N-Queens problem that uses HJlib. This version uses finish, async and finish accumulators.

2. Your task is to use the `NQueensThreads.java` file as a template for a pure Java parallel implementation of NQueens using Java threads concepts introduced in Lecture 23. You may not use any constructs from the HJlib.

   You will likely find it helpful to collect thread references in a data structure like an array after the threads are started, so that you have access to the references to perform calls to join(). For simplicity, you can include joins within each call to `nqueensKernel()`. This is correct, but more restrictive than the finish/async structure for the given code (which will require a more complicated data structure like a ConcurrentLinkedQueue to collect all the thread references).

3. When implementing your `NQueensThreads` solution you will likely find it useful for performance to use the cutoff strategy. Refer to the provided HJlib solution for hints on where to place this cutoff in the Java Threads version.

4. To test your solution, you are provided with a NQueensPerformanceTest class that validates the correctness and performance of your solution 12×12 and 14×14 boards. To complete this portion of the lab, you should submit these performance tests to NOTS by either modifying the provided myjob.slurm template and submitting manually, or through the autograder. The NQueensPerformanceTest class will also compare sequential performance to the provided HJlib version. Our experiments show the following performance with the reference solutions:

   (a) `NQueensSeq` for board size of 14: 31,000 ms on average.
   (b) `NQueensThreads` for board size of 14: 4,600 ms on average.
   (c) `NQueensHjLib` with input argument just 14: 4,200 ms on average.

3 Conversion to Java threads: Spanning Tree

1. The `SpanningTreeSeq.java` program is an example sequential solution to the spanning tree problem.

   The `SpanningTreeAtomicHjLib.java` program is a provided parallel solution to the minimum spanning tree problem. This version uses finish and async constructs along with an AtomicReference.

2. Your task is to convert SpanningTreeAtomicHjLib.java to a pure Java program. You should modify the provided `SpanningTreeAtomicThreads.java` file. Use Java threads instead of finish/async. (The AtomicReference calls can stay unchanged.) As before, you can include joins within each call to `compute()` for simplicity, or you can use a `ConcurrentLinkedQueue` for a more faithful simulation of a finish construct.

3. You have been provided with tests for your parallel spanning tree implementation in SpanningTreePerformanceTest. To complete this portion of the lab, you should submit these performance tests to NOTS by either modifying the provided myjob.slurm template and submitting manually, or through the autograder.

4. Compare the execution time of following versions of the spanning tree problem. Like with NQueens, you may choose to add cutoff threshold values for this program, so as to limit the number of Java threads that will be created (thereby reducing overhead). Below are some reference execution times:

   (a) `SpanningTreeAtomicThreads` with input arguments 50000 and 3000 and a cutoff depth of 5: 450 ms.
   (b) `SpanningTreeAtomicHjLib` with input arguments 50000 and 3000: 325 ms.

4 Programming Tips and Pitfalls for Java Threads

- Remember to call the *start()* method on any thread that you create. Otherwise, the thread's computation does not get executed.
- Since the *join()* method may potentially throw an InterruptedException, you will either need to enclose each call to *join()* within a *try-catch block*, or add a *throws InterruptedException* clause to the definition of the method that includes the call to *join()*.

5 Turning in your lab work

For this 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.
2. Check that all the work for today's lab is in the `lab_8` turnin directory. It's fine if you include more rather than fewer files — don't worry about cleaning up intermediate/temporary files.