Lab 9: Java Threads

Instructor: Vivek Sarkar

Resource Summary

Course wiki: https://wiki.rice.edu/confluence/display/PARPROG/COMP322

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

Coursera Login: visit http://rice.coursera.org and log in via Shibboleth

Clear Login: ssh *your-netid*@ssh.clear.rice.edu and then login with your password

Sugar Login: ssh *your-netid*@sugar.rice.edu and then login with your password

Linux Tutorial visit http://www.rcsg.rice.edu/tutorials/

IMPORTANT: Please refer to the tutorial on Linux and SUGAR from Lab 5, as needed. Also, if you edit files on a PC or laptop, be sure to transfer them to SUGAR before you compile and execute them (otherwise you may compile and execute a stale/old version on SUGAR).

As in past labs, create a text file named `lab_9_written.txt` *in the* `lab_9` *directory, and enter your timings and observations there.*

1 Convert to Java Threads - N-Queens

1. Download the `nqueens.hj` program from lab 5 that uses finish and async constructs along with AtomicInteger calls.
2. Convert it to a pure Java program by using Java threads instead of finish/async, using the concepts introduced in Lecture 24. (The AtomicInteger calls can stay unchanged.) For simplicity, you can include joins within each call to `nqueens_kernel()`. This is correct, but more restrictive than the finish/async structure for the given code. But it simplifies parallelization using Java threads.
3. Compile and run the program as follows to solve the N-Queens problem on a 12×12 board (default value).
   ```
   javac nqueens.java
   java nqueens
   ```
4. Compare the execution time of three versions of NQueens:
   (a) `java nqueens 14 5 0`
   This should correspond to the sequential execution of your Java program since the third argument (= 0) is the cutoff value.
   (b) `java nqueens 14`
   This is a parallel Java run with the default cutoff value of 3. Try experimenting with different values for `cutoff_value` if needed.
   (c) `hj nqueens 14`
   This is a parallel HJ run with the default cutoff value of 3. (It is recommended that you use separate directories for compiling the Java and HJ versions so as to avoid any possible interference among classfiles generated for both versions.)

2 Convert to Java threads - Spanning Tree

1. Download the `spanning_tree_atomic.hj` solution from lab 7 that uses finish and async constructs along with AtomicReference calls.
2. Convert it to a pure Java program by using Java threads instead of finish/async, using the concepts introduced in Lecture 24. (The AtomicReference calls can stay unchanged.) For simplicity, you can include joins within each call to `compute()`. This is correct, but more restrictive than the finish/async structure for the given code. But it simplifies parallelization using Java threads.
3. Compile and run the programs as follows with the default input size.
   `javac spanning_tree_atomic.java`
   `java spanning_tree_atomic`
4. Compare the execution time of three versions of the spanning tree example. You may choose to add cutoff threshold values for this program as was done for N-Queens, so as to limit the number of Java threads that will be created:
   (a) `java spanning_tree_atomic 50000 1000`
   This is a parallel Java run. If you add support for a `cutoff_value`, you can experiment with different cutoff values.
   (b) `hj spanning_tree_atomic 50000 1000`
   This is a parallel HJ run. If you used a cutoff value for the parallel Java run above, you should also add it for this HJ version. (It is recommended that you use separate directories for compiling the Java and HJ versions so as to avoid any possible interference among classfiles generated for both versions.)
   (c) For completeness, create a sequential version of the Java threads program in item 1, and evaluate its performance with the same parameters.

3 Programming Tips and Pitfalls for Java Threads

- Recall that any local variable from an outer scope that is accessed in an anonymous class (e.g., in the run() method) *must be declared final*.
- 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 include each call to *join()* in a *try-catch block*, or add a *throws InterruptedException* clause to the definition of the method that includes the call to *join()*.

4 Turning in your lab work

As in previous labs, you will need to complete a quiz on Coursera and turn in your work before leaving, as follows:

1. Visit rice.coursera.org, select "Fundamentals of Parallel Programming" course, and take the Lab 9 quiz.
2. Check that all the work for today's lab is in the `lab_9` directory. If not, make a copy of any missing files/folders there. It's fine if you include more rather than fewer files — don't worry about cleaning up intermediate/temporary files.

3. Before you leave, create a zip file of your work by changing to the parent directory for `lab_9/` and issuing the following command, "`zip -r lab_9.zip lab_9`".

4. Use the turn-in script to submit the contents of the `lab_9.zip` file as a new `lab_9` directory in your turnin repository as explained in Lab 1. You can always examine the most recent contents of your svn repository by visiting `https://svn.rice.edu/r/comp322/turnin/S13/`*your-netid*.