Lab 7: Isolated Statements, Atomic Variables

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_7_written.txt` *in the* `lab_7` *directory, and enter your timings and observations there.*

1 Parallelization using Isolated Statements

A parallelization strategy for the spanning tree algorithm was introduced this week in Lecture 19, along with an introduction to isolated statements. Recall the following constraints on isolated statements — an isolated statement may not contain any HJ statement that can perform a blocking operation e.g., finish, future get(), and phaser next/wait. In addition, a current limitation in the HJ implementation is that it does not support return statements within isolated.

Your task is to perform the following for the `spanning_tree_seq.hj` program provided for the lab. *As always, please use a SUGAR compute node (not the login node) for all performance evaluations:*

1. Compile the sequential `spanning_tree_seq.hj` program:
   hjc spanning_tree_seq.hj
2. Execute the program using two command line arguments, 100,000 (number of nodes in graph) and 1,000 (number of neighbors):
   hj -places 1:1 spanning_tree_seq 100000 1000
3. Parallelize this program by adding async, finish, and isolated constructs as described in Lecture 19. Call the parallelized version `spanning_tree_isolated.hj`
4. Compile the parallel `spanning_tree_isolated.hj` program:
   hjc spanning_tree_isolated.hj
5. Execute the program using two command line arguments, 100,000 (number of nodes in graph) and 1,000 (number of neighbors):
   hj -places 1:1 spanning_tree_isolated 100000 1000
6. Record the best of 5 execution times reported for `spanning_tree_seq.hj` and `spanning_tree_isolated.hj` in `lab_7_written.txt`. What speedup do you see?

2 Parallelization using Atomic Variables

Lecture 19 also introduced Java atomic variables. As discussed in the lecture, the operations that can be performed on atomic variables are limited to what is supported in the API, whereas isolated statements can be used to convert any general computation into critical sections.

Your task in this section is create a `spanning_tree_atomic.hj` program that replaces `isolated` in your `spanning_tree_isolated.hj` version by equivalent functionality using AtomicReference objects. In addition to the lecture slides, you can find a summary of AtomicReference operations at `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/AtomicReference.html`.

Compile and execute your program `spanning_tree_atomic.hj` program by repeating the steps from the previous section. Record the resulting performance in `lab_7_written.txt`.

3 Performance Evaluation of Sorted List program using Object-Based Isolated Statements

Object-based isolation was also introduced in Lecture 19, with the form `isolated` $(a, b, \ldots)$, where $a, b, \ldots$ is a list of object references. We have provided an example program in `SortedListExampleGbl.hj`. It includes a sorted-list data structure that supports parallel calls on the following methods — `lookup()`, `insert()`, `remove()`, and `sum()`. Note that the `lookup()` method does not contain an `isolated` statement, while the others do. This is assumed to be correct for this example, even though it can potentially create a data race. Also, the other three methods contain calls to a `dummy()` method that contains a synthetic loop with 100,000 arithmetic operations. This was done to simulate situations where the `insert()`, `remove()`, and `sum()` method calls may take more time than in this version and thereby amortize the overhead of object-based isolation.

The example program takes four command line parameters (with appropriate default values):

1. `nthreads`, the number of async tasks to be created that operated on the shared sorted-link data structure. The default value of `nthreads` is 1. It is recommended that you experiment with values up to 8 on a SUGAR compute node. Note that `nthreads` is an application parameter, and is not an HJ runtime parameter.
2. `maxValue`, the intended maximum size of the list. (The list is initialized to half this size.) The default value of `maxValue` is 2048.
3. `insertRemoveRate`, the fraction of operations that correspond to `insert()` and `remove()` method calls. The default value is 0.05, which indicates that 5% of the operations will be `insert()` and 5% will be `remove()`.
4. `sumRate`, the fraction of operations that correspond to `sum()` calls. The default value is 0.01, which indicates that 1% of the operations will be `sum()`.

The output of the program includes an aggregate operations/second throughput metric. This is a metric for which bigger values are better.

Your first task is to run the original `SortedListExampleGbl.hj` program and record the throughput obtained for `nthreads` = 8. Your second task is to replace each occurrence of `isolated` by an equivalent object-based isolated statement to improve parallelism, and observe what impact it has on the throughput performance for `nthreads` = 8. Record your observations in `lab_7_written.txt`.

NOTE: as in standard Java, the following warning message from the HJ compiler is an indication that you should use a type parameter when instantiating an instance of a generic class:

```
[warning] Use of a raw type could lead to unchecked operations
```

4 Turning in your lab work and quiz

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 7 quiz.
2. Check that all the work for today's lab is in the `lab_7` 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_7/` and issuing the following command, "`zip -r lab_7.zip lab_7`".
4. Use the turn-in script to submit the contents of the `lab_7.zip` file as a new `lab_7` 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*.