Lab 7: Atomic Variables and Isolated Statement

Instructor: Vivek Sarkar

1 Update your HJ/DrHJ Installation

The performance measurements for today's lab should be done on Sugar, and we've already updated the HJ installation there. (See Lab 4 handout on setup instructions to access the HJ installation in the COMP 322 userid on Sugar.)

However, if you're also working with a local installation, please update it from the HJ download page, `https://wiki.rice.edu/confluence/display/PARPROG/HJDownload`, to make sure that you have the latest updates and bug fixes.

2 Performance Evaluation of Isolated Statements and Atomic Variables

Atomic variables were introduced in Lecture 6 and isolated statements in Lecture 20. As discussed in these lectures, the operations that can be performed on atomic variables are limited to what is supported in the API, where as isolated statements can be used to convert any general computation into critical sections.

Recall the following constraints on isolated statements (Lecture 20) — 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_isolated.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 `spanning_tree_isolated.hj` program:
   hjc spanning_tree_isolated.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 1:8 spanning_tree_isolated 100000 1000
3. Search for `isolated` in `spanning_tree_isolated.hj` and replace it by equivalent functionality using AtomicReference objects. In addition to the slides for Lectures 6 and 20, you can find a summary of AtomicReference operations at `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/AtomicReference.html`.
4. Compile and execute the modified version of your program by repeating steps 1 and 2. Compare the resulting performance with the original `isolated` version.

3 Performance Evaluation of Object-Based Isolated Statements

Object-based isolation was also introduced in Lecture 20, with the form `isolated` $(a, b, \ldots)$, where $a, b, \ldots$ is a list of object references. As discussed in Lecture 20 and slide 9 of Lecture 21, there are many cases in which the overhead of object-based isolation may outweighs its benefits. The goal is the section is to study two examples, one for which standard isolation is better than object-based isolation, and another for the converse.

3.1 Spanning Tree example

Your task is to replace `isolated` by an equivalent object-based isolated statement in the `spanning_tree_isolated.hj` example from the previous section, with the goal of increasing parallelism. Observe how the performance of object-based isolation compares with the fully-isolated and AtomicReference versions studied in the previous section.

3.2 Sorted List example

We have provided another 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.

2. `maxValue`, the intended maximum value 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.

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
```