Lab 11: Speculative Task Parallelism and Eureka!

Instructor: Vivek Sarkar, Co-Instructor: Shams Imam

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

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

Importants tips and links

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

Piazza site : `https://piazza.com/rice/spring2016/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/`

HJlib Jar File : `https://github.com/habanero-maven/hjlib-maven-repo/raw/mvn-repo/edu/rice/hjlib-cooperative/0.1.9/hjlib-cooperative-0.1.9.jar`

HJlib API Documentation : `http://pasiphae.cs.rice.edu/`

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

1 Lab Goals

In today's lab you will use HJlib's eureka construct. The goals include:

- Familiarity with search and optimization style programs.
- Familiarity with the use of `Eureka`s in HJlib.
- Browsing HJlib Javadoc to write HJlib programs.

This lab can be downloaded from the following svn repository:

- `https://svn.rice.edu/r/comp322/turnin/S16/`*NETID*`/lab_11`

Use the subversion command-line client or IntelliJ to checkout the project into appropriate directories locally.

In today's lab, you **do not** need to use NOTS to run performance tests.

2 Introduction to Speculative Parallelism

In lecture earlier this week, you were introduced to Eureka-style speculative parallel programs. This lab is designed to familiarize yourself with writing speculatively task parallel programs. You will be required to write simple speculative programs to familiarize yourself with various Eureka constructs available in HJlib.

A wide range of problems, such as combinatorial optimization, constraint satisfaction problem, image matching, genetic sequence similarity, iterative optimization methods, can be reduced to tree or graph search problems. A pattern common to such algorithms to solve these problems is a *eureka* event, a point in the program which announces that a result has been found. Such an event curtails computation time by avoiding further exploration of a solution space or by causing the successful termination of the entire computation. Eureka-Style Computations include search and optimization problems that could benefit greatly from speculative parallelism. For example, in satisfiability problems, the first eureka event can trigger the termination of the entire computation as a proof of the existence has been found. On the other hand, in optimization problems, a eureka event declares (and updates) the currently best-known result and can prune the computation by causing the termination of specific tasks that cannot provide a better result.

3 Lab Tasks

The overall task in this lab is to perform pattern search in a 2D string array with various desired outcomes. In `PatternMatchEureka`, you are provided a few functions: `runSequential`, `runAsyncFinish`, `runExistentialEureka`, `runCountEureka`, `runMinimalIndexEureka`, and `runMaximalIndexEureka`. The lab provides you the sequential version to perform a search. For this lab, you need to write parallel programs that achieve the following:

(a) `runAsyncFinish`: `async-finish` version of the program that parallelizes the sequential program in `runSequential`. As a general example, Figure 1 shows a simple `async-finish` program performing a search.
(b) `runExistentialEureka`: `Eureka` version for the index of any matching element using `HjSearchEureka`. As a general example, Figure 2 shows a simple program performing a computation using search eureka.
(c) `runCountEureka`: `Eureka` version for the index of first `COUNT` matching elements using `HjCountEureka`.
(d) `runMinimalIndexEureka`: `Eureka` version for the minimum index of all matching element using `HjExtremaEureka`.
(e) `runMaximalIndexEureka`: `Eureka` version for the maximum index of all matching element using `HjExtremaEureka`.

Figure 1: Parallel search using just the async and finish constructs.

```
import edu.rice.hj.experimental.api.*
import static edu.rice.hj.experimental.ModuleZ.*

class ParallelAsyncFinishSearch {
  def doWork(matrix, goal) {
    val token = {null} // an array to store the element
    finish {
      for rowIndices in matrix.chunks()
        async {
          for r in rowIndices
            procRow(token, matrix(r), r, goal)
        }
    }
    return token[0]
  }
  def procRow(eu, array, r, goal) {
    for c in array.length()
      if (token[0] != null) {
        return
      }
      if goal.match(array(c))
        token[0] = [r, c]) // eureka event
  }
}
```

If solved correctly, all the eureka versions will have similar code, just the eureka instance will need to be changed. You may need to read up on the javadocs to figure out how to correctly initialize the various `Eureka` instances. The relevant Javadoc links for this lab are:

- Eureka classes: `http://www.cs.rice.edu/~vs3/hjlib/doc/index.html?edu/rice/hj/experimental/api/package-summary.html`

- Eureka factories, eureka-specific finish in ModuleZ: `http://www.cs.rice.edu/~vs3/hjlib/doc/index.html?edu/rice/hj/experimental/ModuleZ.html`

4 `Eureka` Construct and API

In this section, we introduce the eureka construct that is used by speculative tasks to trigger eureka events. A `Eureka` is a new construct that provides support for speculative parallelism in an `async-finish` setting. Once a `Eureka` construct has been *resolved* by reaching a stable value, it enables detection of a group of speculative tasks that can be terminated. The operations that can be performed on a `Eureka`, `eu`, are defined by the following interface:
(a) `offer`(`auxiliaryData`): Notifies `eu` that a eureka event has been triggered; additional information used to mutate the internal state of `eu` is available in `auxiliaryData`. Whether or not the event resolves `eu`, it can cause the task invoking this operation to terminate at a well-defined program point.
(b) `check`(`auxiliaryData`): This operation allows a speculative task to check whether it has become terminable as, e.g., `eu` has been resolved. If the task has become terminable, a call to `check` will cause the task to be terminated. By accepting an argument, `check` enables the caller to pass additional values that can be used to determine whether to terminate a task.
(c) `isResolved`(): Allows a speculative task to query whether `eu` has been resolved. This method returns a boolean value and never causes a task to be terminated.
(d) `get`(): If `eu` has been resolved, it returns the resolved value. Otherwise, a transitory value of `eu` is returned. One is guaranteed to receive the resolved value if this operation is invoked outside the `finish` scope on which `eu` was registered.

This interface can be used to implement `Eureka` patterns that include computations that produce both deterministic and non-deterministic results. These patterns include:
(a) **Search Eureka**: Search is a well-known pattern in EuSCs. Once the result is discovered, all parallel searching entities should ideally be terminated as quickly as possible to minimize doing redundant work. A `SearchEureka` construct is designed to be resolved by the first eureka event it processes, and it promptly terminates the task that triggered the event.
(b) **Count Eureka**: Another variant of a parallel search is where we wish to know the first K results that match a query. We wish to terminate the computation when at least K of the asynchronous computations have completed successfully. A `CountEureka` is initialized with a count K and is resolved after exactly K eureka events have been triggered. A call to `CountEureka.get()` returns a list of values of maximum length K instead of a single value. If none of the tasks triggered a eureka, then an empty list is returned. In general, a `SearchEureka` can be viewed as a `CountEureka` with a count of 1. **Note:** The `newCountEureka` factory method accepts two arguments, the first argument named `maxCount` represents K while the second argument named `spaceForResult` represents how many of the values to store as result. The common case, as is the case with this lab, is for `maxCount` to equal `spaceForResult`.
(c) **Optimization (Extrema) Eureka**: Many problems from artificial intelligence can be defined as combinatorial optimization problems. Subproblems are derived from the originally given problem through the addition of new constraints. The structure of the algorithm requires the ability to terminate individual subtrees of the search tree. An example is where we are interested in finding the lowest index of the goal item if it exists in the array. In our EuPM, the GUB is available in the `MinimaEureka` instance, `eu`, that a speculative task is registered on and can be retrieved by a call to `eu.get()`. Calls to `offer` and `check` pass the current known bounds or solution, respectively, as the argument. If the argument in the `offer` call is lower than the GUB, the GUB is updated in the `MinimaEureka` instance, otherwise the current task is terminated. Similarly, calls to `check` terminate a task if the argument is larger than the currently known GUB in `eu`.

5 Eureka Programming Model (EuPM)

In this section, we describe how parallelism is expressed via speculative tasks and termination of a single task, as well as a group of tasks, is supported in the EuPM. The EuPM is an extension of the task-parallel `async-finish` model where speculative tasks are created using the `async` keyword. A `finish` block can register on a `Eureka`, `eu`, with the following pseudocode syntax (the library API includes `eu` as a parameter to the `finish` API): `finish`(eu) ⟨*stmt*⟩. The `finish` construct simplifies the identification of the group of tasks that participate in a eureka-style synchronization on a particular `Eureka` instance.

All tasks having the same immediately enclosing finish (IEF) belong to the same group and inherit the registration on the `Eureka` instance, `eu`, from the finish scope. Finish scopes with different `Eureka` instance registrations can be nested allowing composability of different speculative computations. Similarly, multiple `finish` blocks can register on the same `Eureka` instance, `eu`, to represent that different speculative sub-computations are linked. When one of the speculative tasks resolves `eu` it makes other tasks from the same or different groups also registered on `eu` to become redundant and terminable. If none of the tasks trigger a eureka event that resolves the registered `eu`, the computation completes normally when all tasks inside each `finish` scope complete. Invalid calls to `check`/`offer` from a task not executing in a EuSC (i.e. `finish` scope not registered on a `Eureka`) results in a runtime error.

The EuPM specific operations that a task, `T`, can perform on a `Eureka`, `eu`, are defined as follows:
(a) **new**: Task `T` can create a new instance of the `Eureka` construct, `eu`, and obtain a handle to it. The reference `eu` can now be used to register on new `finish` scopes. The creator task can pass the reference of `eu` to other tasks.
(b) **registration**: `eu` can be explicitly registered on a `finish` scope. Note that the task that created `eu` cannot register on `eu`. A newly spawned task, `T`, implicitly registers on `eu` only if the IEF of `T` was explicitly registered on `eu`. Currently, we do not provide a mechanism for an `async` task to explicitly register on `eu`.

Figure 2: Parallel search using the Eureka model.

```
1  import edu.rice.hj.experimental.api.*
2  import static edu.rice.hj.experimental.ModuleZ.*
3
4  class ParallelEurekaSearch {
5    def doWork(matrix, goal) {
6      val eu = newSearchEureka([-1, -1])
7      finish (eu) { // eureka registration
8        for rowIndices in matrix.chunks()
9          async {
10           for r in rowIndices
11             procRow(eu, matrix(r), r, goal)
12         }
13     }
14   return eu.get()
15   }
16   def procRow(eu, array, r, goal) {
17     for c in array.length()
18       eu.check([r, c]) // termination check
19       if goal.match(array(c))
20         eu.offer([r, c]) // trigger eureka event
21   }
22 }
```

Figure 2 displays a parallel 2D array search program using `async` and `finish` constructs in the EuPM. We create the `SearchEureka` instance, `eu`, inside the factory method `eurekaFactory`. This instance, `eu`, is registered by the `finish` scope defined on line 7. Hence, all `async` tasks launched at line 9 are automatically registered on `eu` and belong to the same group. The tasks trigger the eureka event by invoking the `offer` method at line 20. There is no need for an explicit `return` statement in `procRow`, as `offer` on a `SearchEureka` causes the task to terminate. To enable cooperative termination, there are also calls to `check` (line 18) to check the state of the registered eureka implicitly. When `eu` has been resolved, `check` will cause the terminable tasks to terminate. Eventually, all tasks inside the `finish` block at line 7 will complete execution or be terminated, and the computation will proceed to line 14 and the result will be returned. Note that, the final answer in this example is nondeterministic, but there are no data races involved.

6 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.
2. Check that all the work for today's lab is in the `lab_11` turnin directory. It's fine if you include more rather than fewer files — don't worry about cleaning up intermediate/temporary files.