# COMP 322: Fundamentals of Parallel Programming

# Lecture 31: Eureka-style Speculative Task Parallelism

**Vivek Sarkar, Shams Imam**
**Department of Computer Science, Rice University**

**Contact email: vsarkar@rice.edu, shams.imam@twosigma.com**

**http://comp322.rice.edu/**

# Worksheet #30: Characterizing Solutions to the Dining Philosophers Problem

For the five solutions studied in today's lecture, indicate in the table below which of the following conditions are possible and why:

1. Deadlock: when all philosopher tasks are blocked (neither thinking nor eating)
2. Livelock: when all philosopher tasks are executing but ALL philosophers are starved (never get to eat)
3. Starvation: when one or more philosophers are starved (never get to eat)
4. Non-Concurrency: when more than one philosopher cannot eat at the same time, even when resources are available

| | Deadlock | Livelock | Starvation | Non-concurrency |
|---|---|---|---|---|
| Solution 1: synchronized | Yes (72/73) | No (68/73) | Yes (50/73) | Yes (22/73) |
| Solution 2: tryLock/unLock | No (73/73) | Yes (45/73) | Yes (67/73) | Yes (15/73) |
| Solution 3: isolated | No (71/73) | No (72/73) | Yes (26/73) | Yes (67/73) |
| Solution 4: object-based isolation | No (71/73) | No (67/73) | Yes (64/73) | No (64/73) |
| Solution 5: semaphores w/ FIFO queues | No (71/73) | No (71/73) | No (57/73) | No (71/73) |

# What is a "Eureka Style" Computation?

- Many optimization and search problems attempts to find a result with a certain property or cost
- Announce when a result has been found
  - An "aha!" moment – **Eureka** event
  - Can make rest of the computation unnecessary

==> Opportunities for "speculative parallelism", e.g., Parallel Search, Branch and Bound Optimization, Soft Real-Time Deadlines, Convergence Iterations, . . .


Image source: http://www.netstate.com/states/mottoes/images/ca_eureka.jpg

# Simple Example: Search in a 2-D Matrix

```
class AsyncFinishSearch {
  AtomicReference atomicRefFactory() {
    // [x, y] is pseudocode syntax for specifying an integer pair
    return new AtomicReference([-1, -1])
  }
  int[] doWork(matrix, goal) {
    val token = atomicRefFactory()
    finish (() -> {
      // How to break from a forasync loop?
      forasyncChunked (0, matrix.length - 1, (r) -> {
        procRow(matrix(r), r, goal, token)
      });
    });
    // return either [-1, -1] or valid index [i, j] matching goal
    return token.get()
  }
  void procRow(array, r, goal, token) {
    for (int c = 0; c < array.length(); c++)
      if goal.match(array(c)) // eureka!!!
        token.set([r, c])
        return
} }
```

# Challenges in Parallelizing a Eureka-Style Computation

- Detecting eureka events
  —need to pass token around as extra argument
- Terminating executing tasks after eureka
  —manual termination via cancellation tokens can be a burden
  —throwing an exception does not terminate other parallel tasks
  —"killing" a parallel task can lead to unpredictable results (depending on when the task was terminated)

# Example of Manual termination via Cancellation Tokens

- Manual periodic checks with `return`s
- User controls responsiveness

```
class AsyncFinishManualSearch {
  int[] doWork(matrix, goal) {
    val token = atomicRefFactory()
    finish (() -> {
      forasyncChunked (0, matrix.length - 1, (r) -> {
        if (token.get() != null)
          return
        procRow(matrix(r), r, goal, token)
      });
    });
    // [-1, -1] or valid index [i, j] matching goal
    return token.get()
  }
  void procRow(array, r, goal, token) {
    for (int c = 0; c < array.length(); c++)
      if (token.get() != null)
        return
      if goal.match(array(c)) // eureka!!!
        token.set([r, c])
        return
  } }
```

Repeated checks which are written manually

- Cumbersome to write
- Impossible to support inaccessible functions

# HJlib solution: the Eureka construct

1. `eureka = eurekaFactory() // create Eureka object`
2. `finish (eureka) S1  // register eureka w/ finish`
   - Multiple `finish`'es can register on same Eureka
   - Wait for all tasks to finish as before
     - Except that some tasks may terminate early when eureka is resolved
3. `async // task candidate for early termination`
   - Inherits eureka registrations from immediately-enclosing finish
4. `offer()`
   - Triggers eureka event on registered eureka
5. `check() // Like a "break" statement for a task`
   - Causes task to terminate if eureka resolved

# 2D Matrix Search using Eureka construct (Pseudocode)

```
class AsyncFinishEurekaSearch {
  HjEureka eurekaFactory() {
    return ...
  }
  int[] doWork(matrix, goal) {
    val eu = eurekaFactory()
    finish (eu, () -> { // eureka registration
      forasyncChunked (0, matrix.length - 1, (r) -> {
        procRow(matrix(r), r, goal)
      });
    });
    // return either [-1, -1] or valid index [i, j] matching goal
    return eu.get()
  }
  void procRow(array, r, goal) {
    for (int c = 0; c < array.length(); c++)
      check([r, c]) // cooperative termination check
      if goal.match(array(c)) // eureka!!!
        offer([r, c]) // trigger eureka event
  } }
```

# Eureka Variants (Pseudocode)

```
def eurekaFactory() {
  val initValue = [-1, -1]
  return new SearchEureka(initValue)
}
```

```
def eurekaFactory() {
  val K = 4
  return new CountEureka(K)
}
```

```
def eurekaFactory() {
  // comparator to compare indices
  val comparator = (a, b) -> {
    ((a.x - b.x) == 0) ? (a.y - b.y) : (a.x - b.x)
  }
  val initValue = [INFINITY, INFINITY]
  return new MinimaEureka(initValue, comparator)
}
```

```
def eurekaFactory() {
  val time = 4.seconds
  return new TimerEureka(time)
}
```

```
def eurekaFactory() {
  val units = 400
  return new EngineEureka(units)
}
```

# Binary Tree Search Example

**Inputs:**

- binary tree, T
- id for each node in T, in breadth-first order e.g., root.id = 0, root.left.id = 1, root.right.id = 2, …
- value for each node in T that is the search target

**Outputs:**

- calls to offer() resolve eureka
- calls to check() can lead to early termination
- final value of eureka contains id of a node with value == elemToSearch

```
HjSearchEureka<Integer> eureka = newSearchEureka(null);
finish(eureka, () -> {
    async(() -> {
        searchBody(eureka, rootNode, elemToSearch);
    });
});

private static void searchBody(
  HjSearchEureka<Integer> eureka, Node rootNode,
  int elemToSearch) throws SuspendableException {
    eureka.check(rootNode.id);
    if (rootNode.value == elemToSearch) {
        eureka.offer(rootNode.id);
    }
    if (rootNode.left != null) {
        async(() -> {
            searchBody(eureka, rootNode.left, elemToSearch);
        });
    }
    if (rootNode.right != null) {
        async(() -> {
            searchBody(eureka, rootNode.right, elemToSearch);
        });
    }
}
```

# Tree Min Index Search Example

**Inputs:**

- binary tree, T
- id for each node in T, in breadth-first order e.g., root.id = 0, root.left.id = 1, root.right.id = 2, …
- value for each node in T that is the search target

**Outputs:**

- calls to offer() update eureka with minimum id found so far (among those that match)
- calls to check() can lead to early termination if the argument is >= than current minimum in eureka
- final value of eureka contains minimum id of node with value == elemToSearch

```
HjExtremaEureka<Integer> eureka = newExtremaEureka(
  Integer.MAX_VALUE, (Integer i, Integer j) -> j.compareTo(i));
finish(eureka, () -> {
    async(() -> {
        minIndexSearchBody(eureka, rootNode, elemToSearch);
    });
});

private static void minIndexSearchBody(
  HjExtremaEureka<Integer> eureka, Node rootNode,
  int elemToSearch) throws SuspendableException {
    eureka.check(rootNode.id);
    if (rootNode.value == elemToSearch) {
        eureka.offer(rootNode.id);
    }
    if (rootNode.left != null) {
        async(() -> {
            minIndexSearchBody(eureka, rootNode.left, elemToSearch);
        });
    }
    if (rootNode.right != null) {
        async(() -> {
            minIndexSearchBody(eureka, rootNode.right, elemToSearch);
        });
    }
}
```

# AND-composition of Eurekas

```
class AsyncFinishEurekaDoubleSearch {
  int[] doWork(matrix, goal) {
    val eu1 = eurekaFactory()
    val eu2 = eurekaFactory()
    val eu = eurekaComposition(AND, eu1, eu2)
    finish (eu, () -> { // eureka registration
      forasyncChunked (0, matrix.length - 1, (r) -> {
        procRow(matrix(r), r, goal1, goal2)
      });
    });
    // return either [-1, -1] or valid index [i, j] matching goal
    return eu.get()
  }
  void procRow(array, r, goal) {
    for (int c = 0; c < array.length(); c++)
      val checkArg = [[r, c], [r, c]]
      check(checkArg) // cooperative termination check
      val loopElem = array(c)
      val res1 = g1.match(loopElem) ? [r, c] : null
      val res2 = g2.match(loopElem) ? [r, c] : null
      val foundIdx = [res1, res2] // pair of values for eu1 and eu2
      offer(foundIdx) // possible eureka event
} }
```