COMP 322: Fundamentals of Parallel Programming

Lecture 21: Eureka-style Speculative Task Parallelism

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

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

Worksheet #20 solution: Insertion of isolated for correctness

The goal of IsolatedPRNG is to implement a single Pseudo Random Number Generator object that can be shared by multiple tasks. Show the isolated construct(s) that you can insert in method nextSeed() to avoid data races in the sample main program (pseudocode is fine).

```
1.class IsolatedPRNG {
2. private int seed;
3. public int nextSeed() {
4.   int retVal;
5.   isolated {
6.    retVal = seed;
7.    seed = nextInt(retVal);
8.    }
9.   return retVal;
10.  });
11.} // nextSeed()
12.  . . .
13.} // IsolatedPRNG
```

```
main() { // Pseudocode
  // Initial seed = 1
  IsolatedPRNG r = new IsolatedPRNG(1);
  async(() -> { print r.nextSeed(); ... });
  async(() -> { print r.nextSeed(); ... });
} // main()
```

What might happen if line 5 and line 6 were enclosed in separate isolated statements?

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

```
1  class AsyncFinishSearch {
2    def atomicRefFactory() {
3      val initValue = [-1, -1]
4      return new AtomicRef(initValue)
5    }
6    def doComputation(matrix, goal) {
7      val token = atomicRefFactory()
8      finish
9        for rowIndices in matrix.chunks()
10         async
11           for (r in rowIndices)
12             processRow(matrix(r), r, goal, token)
13     // return either [-1, -1] or a valid index [i, j]
14     return token.get()
15   }
16   def processRow(rowData, r, goal, token) {
17     for (c in rowData.indices())
18       if goal.matches(rowData(c))  // eureka!!!
19         token.set([r, c])
20         return
21   }
```


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 impact parallel tasks
  - —"killing" a parallel task can lead to unpredictable results

Example of Manual termination via Cancellation Tokens

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

```
1   async
2     for (r in rowIndices)
3       if token.eureka()
4         return
5       processRow(matrix(r), r, goal, token)

7   def processRow(rowData, r, goal, token) {
8     for (c in rowData.indices())
9       if token.eureka()
10        return
11      if goal.matches(rowData(c))
12        token.set([r, c])
13  }
```

Repeated checks which are written manually

- Cumbersome to write
- Impossible to support inaccessible functions

HJlib solution: the Eureka construct

1. `eureka = eurekaFactory()`
2. `finish (eureka) S1`
   - 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`
   - Inherits eureka registrations from immediately-enclosing finish
4. `offer()`
   - Triggers eureka event on registered eureka
5. `check()`
   - Causes task to terminate if eureka resolved

2D Matrix Search using Eureka construct (Pseudocode)

```
1  class AsyncFinishEurekaSearch {
2    def eurekaFactory() {
3      ...
4    }
5    def doComputation(matrix, goal) {
6      val eu = eurekaFactory()
7      finish (eu) // eureka registration
8        for rowIndices in matrix.chunks()
9          async
10           for r in rowIndices
11             processRow(matrix(r), r, goal)
12     return eu.get()
13   }
14   def processRow(rowData, r, goal) {
15     for c in rowData.indices()
16       check([r, c]) // cooperative termination check
17       if goal.matches(rowData(c))
18         offer([r, c]) // trigger eureka event
19   } }
```


Eureka Variants

```
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

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

private static void searchBody(
  final HjSearchEureka<Integer> eureka, final Node rootNode,
  final 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 mimimum in eureka
- final value of eureka contains minimum id of node with value == elemToSearch

```
final 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(
  final HjExtremaEureka<Integer> eureka, final Node rootNode,
  final 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

```
1  class AsyncFinishEurekaDoubleSearch {
2    def doComputation(matrix, goal1, goal2) {
3      val eu1 = eurekaFactory()
4      val eu2 = eurekaFactory()
5      val eu = eurekaComposition(AND, eu1, eu2)
6      finish (eu) // eureka registration
7        for rowIndices in matrix.chunks()
8          async
9            for r in rowIndices
10             processRow(matrix(r), r, goal1, goal2)
11     // eu1.get() or eu2.get() to determine which goal was found
12     return eu.get()
13   }
14   def processRow(rowData, r, goal1, goal2) {
15     for c in rowData.indices()
16       val checkArg = [[r, c], [r, c]] // pair for eu1 and eu2
17       check(checkArg) // cooperative termination check
18       val foundIdx1 = goal1.matches(rowData(c)) ? [r, c] : [-1, -1]
19       val foundIdx2 = goal2.matches(rowData(c)) ? [r, c] : [-1, -1]
20       val foundIdx = [foundIdx1, foundIdx2] // pair for eu1 and eu2
21       offer(foundIdx) // possible eureka event
22   } }
```