COMP 322: Fundamentals of Parallel Programming

Lecture 2: Task Creation and Termination using Async & Finish

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

COMP 322 Lecture 2 12 January 2011
Acknowledgments for Today’s Lecture

• COMP 322 Lecture 2 handout
Habanero-Java (HJ) Language

• HJ is a new language developed in the Rice Habanero Multicore Software research project
  — Derived from IBM’s Java-based X10 v1.5 implementation in 2007
  — HJ is an extension of Java 1.4
    – Java 5 & 6 *language* features (generics, metadata, etc.) are currently not supported by the HJ front-end
    – However, Java 5 & 6 libraries and classes can be called from HJ programs
      • Just don’t call a method that performs a blocking operation because that will mess up the HJ scheduler!

• Four classes of parallel programming primitives in HJ:
  1. Dynamic task creation & termination: forall, async, finish, get
  2. Mutual exclusion and isolation: isolated
  3. Collective and point-to-point synchronization: phaser, next
  4. Locality control --- task and data distributions: places, here
HJ Compilation and Execution Environment

```
Foo.hj
    ▼
    |   
    v  
HJ compiler

hjc Foo.hj

Foo.class

hj --places m:n Foo

HJ Runtime Environment = JRE + HJ libraries + HJ Multithreaded Runtime
```

- **HJ source program** --- must contain a class named Foo with a `public static void main(String[] args)` method
- **HJ compiler** translates Foo.hj to Foo.class, and inserts calls to HJ runtime as needed
- **HJ runtime** allocates m*n worker threads across m “places” (default values: m = 1, n = 3)
- **HJ Abstract Performance Metrics** (optional, enabled by --perf=true option for hj command)

**Caveat:** this is a research prototype with known limitations. Please report bugs and suggestions to comp322-staff@mailman.rice.edu.
Async and Finish Statements for Task Creation and Termination (Recap)

**async** \( S \)

- Creates a new child task that executes statement \( S \)
- Parent task immediately continues to statement following the async

```java
//Task \( T_0 \) (Parent)
finish {   //Begin finish
    async
    STMT1;   //\( T_1 \) (Child)
    //Continuation
    STMT2;   //\( T_0 \)
}   //Continuation   //End finish
STMT3;   //\( T_0 \)
```

**finish** \( S \)

- Execute \( S \), but wait until *all* (transitively) spawned asyncs in \( S \)'s scope have terminated.
- Implicit *finish* between start and end of main program

```
//Task \( T_0 \) (Parent)
finish {   //Begin finish
    async
    STMT1;   //\( T_1 \) (Child)
    //Continuation
    STMT2;   //\( T_0 \)
}   //Continuation   //End finish
STMT3;   //\( T_0 \)
```
Async Example #1

// Example 1: execute iterations of a counted for loop in parallel
// (we will later see forall as a shorthand for this common case)
for (int i = 0; i < A.length; i++)
    // value of local variable i is copied on entry to async
    async { A[i] = B[i] + C[i]; }
Async Example #2

// Example 2: execute iterations of a while loop in parallel

p = first;
while ( p != null ) {
    // value of local variable p is copied on entry to async
    async { p.x = p.y + p.z; }
    p = p.next;
}
Async Example #3

// Example 3: Example 2 rewritten as a recursive method
static void process(T p) {
    if ( p != null ) {
        // value of local variable p is copied on entry to async
        async { p.x = p.y + p.z; }
        process(p.next);
    }
}

Async Example #4

// Example 4: execute method calls in parallel
async left_s = quickSort(left);
async right_s = quickSort(right);
Scheduling HJ tasks on processors in a parallel machine

- HJ runtime creates a small number of worker threads, typically one per core
- Workers push async's/continuations into a logical work queue
  - when an async operation is performed
  - when an end-finish operation is reached
- Workers pull task/continuation work item when they are idle
Continuations

- A continuation is one of two kinds of program points
  - The point in the parent task immediately following an async
  - The point immediately following an end-finish

- Continuations are also referred to as task-switching points
  - Program points at which a worker may switch execution between different tasks

```latex
finish { // F1
  async A1;
  finish { // F2
    async A3;
    async A4;
  }
  S5;
}
```

Continuations
Local Variables

- Java variables can be classified as *local* or *shared*
- A local variable is only visible in the scope in which it is defined
- A shared variable (static field, instance field, array element) can potentially be accessed anywhere
- Three rules for accessing local variables across tasks in HJ:

  // Rule 1: an inner async may access the value of any outer final local var
  final int i1 = 1; async { ... = i1; /* i1=1 */ }

  // Rule 2: an inner async may access the value of any outer local var
  int i2 = 2; // i2=2 is copied on entry into the async like a method param
  async { ... = i2; /* i2=2*/ }
  i2 = 3; // This assignment is not seen by the above async

  // Rule 3: an inner async is not permitted to modify an outer local var
  int i3; async { i3 = ...; /* ERROR */ }
Finish Statements

• Implicit finish statement in main() method
• Each async task has a unique Immediately Enclosing Finish (IEF)
• One possible approach to converting a sequential Java program to a parallel HJ program
  – Insert async’s at points where parallelism is desired
  – Then insert finish’s to ensure that the parallel version produces the same results as the sequential version
Finish Example #1

// Example 1: Sequential version
for (int i = 0; i < A.length; i++) A[i] = B[i] + C[i];
System.out.println(A[0]);

// Example 1: Incorrect parallel version
for (int i = 0; i < A.length; i++) async A[i] = B[i] + C[i];
System.out.println(A[0]);

// Example 1: Correct parallel version
finish for (int i = 0; i < A.length; i++) async A[i] = B[i] + C[i];
System.out.println(A[0]);
Finish Example #2

// Example 2: Sequential version
p = first;
while ( p != null ) {
    p.x = p.y + p.z;   p = p.next;
} System.out.println(first.x);

// Example 2: Incorrect parallel version
p = first;
while ( p != null ) {
    async { p.x = p.y + p.z; }
    p = p.next;
} System.out.println(first.x);
Finish Example #2 (contd)

// Example 2: Correct parallel version
p = first;
finish while ( p != null ) {
    async { p.x = p.y + p.z; }
    p = p.next;
}
System.out.println(first.x);
Which statements can potentially be executed in parallel with each other?

```javascript
finish { // F1
  // Part 1 of Task A0
  async {A1; async A2;}
  finish { // F2
    // Part 2 of Task A0
    async A3;
    async A4;
  }
  // Part 3 of Task A0
}
```

- Example: A2 can potentially execute in parallel with A3 and A4, but Part 3 of A0 cannot execute in parallel with A3 and A4
Async-Finish Exception Semantics

- Any exception thrown by an async is accumulated into a MultiException at its Immediately Enclosing Finish (IEF)
- Async creation can never result in an exception

```java
finish { // F1
    // Part 1 of Task A0
    async {A1; async A2;}
    try {
        finish { // F2
            // Part 2 of Task A0
            try { async A3; }
            catch (Exception e1) { }; // will not catch exception in A3
            async A4;
        }
    } catch (Exception e2) { }; // will catch exception in A3
    // Part 3 of Task A0 }
```