The JR Programming Language: Concurrent Programming in an Extended Java

THE JR PROGRAMMING LANGUAGE: CONCURRENT PROGRAMMING IN AN EXTENDED JAVA

RONALD A. OLSSON
Department of Computer Science
University of California, Davis

AARON W. KEEN
Department of Computer Science
California Polytechnic State University

Kluwer Academic Publishers
Boston/Dordrecht/London

Contents

Chapter 1

CONCURRENT EXECUTION

This document describes JR's concurrent invocation statement, whose design and implementation were finished just after the JR book was published. Note that this presentation is preliminary (and contains some notes to ourselves for later work); comments are welcome! At some point, this document will be merged into the JR book (most likely as a new section right after Section 4.3 "Static and Non-Static Processes").

The concurrent invocation statement also introduces an "additional source of asynchrony" that needs to be considered when dealing with exception handling. So, further changes will be made to Chapter 12 "Exceptions", especially Section 12.4. (See Reference [1], available on the JR webpage, for details.)

1.1 Concurrent Invocation Statement

The concurrent invocation statement provides another mechanism for creating processes dynamically. It consists of one or more concurrent commands separated by `[]` delimiters:

`co` *concurrent_command* `[]` *concurrent_command* `[]` ...

Each concurrent command consists of an invocation and optionally a block of postprocessing code. It thus has two forms:

invocation `;` or *invocation* `{` *block* `}`

The invocation part of a concurrent command is a call invocation, send invocation, or a simple assignment that calls a user-defined function. (JR does not provide a general cobegin statement, often described in operating systems textbooks and courses, in which concurrent commands can be arbitrary statement lists. Such a statement is difficult to implement because each statement list would need to be able to access the stack of the enclosing process.)

A concurrent command can be preceded by an optional quantifier to specify a collection of invocations, one for each combination of values of the bound variables. The scope of such a bound variable extends to the end of the concurrent command. The quantifiers have the same form as those seen earlier in this chapter.

The next subsection describes the meaning and gives examples of concurrent invocation statements without postprocessing code. The subsection following that describes and illustrates postprocessing code.

Concurrent Invocation Without Postprocessing Code

As a simple example, consider the following concurrent invocation statement:

```
co p(3); [] q(); [] a = r(x,y);
```

It consists of three invocations: one each of `p`, `q`, and `r`. The final invocation assigns the value that `r` returns to `a`.

Execution of a concurrent invocation statement first starts all invocations in parallel. A concurrent invocation statement with no postprocessing code terminates when all its invocations have completed. The above `co` therefore terminates when all three of its invocations have terminated.

As a more complete example, the following class uses processes to compute the partial sums of integers from 0 up to one less than the value of command-line argument `N`. The program initializes `sum` so that `sum[i]` is equal to `i`. When the for loop terminates, each `sum[i]` is the sum of the integers from 0 to `i`. The program uses what is called a parallel prefix algorithm: Start with distance `d` equal to 1. Then add `sum[i-d]` to `sum[i]` in parallel (for all `i` greater than `d`), double `d`, and repeat until `d` is at least `n`. To avoid interference between the processes, an extra array, `old`, is used for temporary storage of a copy of `sum`.

```
public class PartialSums{
  private static int N;
  private static int d; // distance
  private static int [] sum;
  private static int [] old; // copy of sum
  // for all i, set sum[i] to 0 + ... + i
  public static void main(String[] args) {
    N = Integer.parseInt(args[0]);
    // initialize sum
    sum = new int [N];
    for (int i = 0; i < N; i++) {
      sum[i] = i;
    }
    old = new int [N];
    // compute; double the distance each time
    for (d = 1; d < N; d = d+d) {
      co ((int i = 0; i < N; i++)) save(i);
```

```
      co ((int i = 0; i < N; i++)) update(i);
    }
    // print results
    for (int i = 0; i < N; i++) {
      System.out.println(i + " " + sum[i]);
    }
  }
  private static op void save(int i) {
    old[i] = sum[i];
  }
  private static op void update(int i) {
    if (i > d) sum[i] += old[i-d];
  }
}
```

Each iteration of the main compute loop uses two concurrent invocation statements. Each of these uses a quantifier to create `N` processes. The first group of processes executes `save` to copy old values of `sum` into `old`. The second group executes `update` to calculate new values for `sum`. Each of the concurrent invocation statements uses the bound variable `i` as an argument in its invocations. This is permitted since the scope of the quantifier variable extends through the concurrent command.

As another example, the following program implements a parallel version of the familiar quicksort algorithm:

```
public class Quicksort{
  public static void main(String[] args) {
    // read N and array a from command line
    int N = Integer.parseInt(args[0]);
    int [] a = new int [N];
    for (int i = 0; i < N; i++) {
      a[i] = Integer.parseInt(args[i+1]);
    }
    System.out.println("input:");
    writeArray(a);
    sort(a, 0, a.length-1);
    System.out.println("sorted:");
    writeArray(a);
  }
  private static void writeArray(int [] a) {
    for (int i = 0; i < a.length; i++) {
      System.out.println(a[i]);
    }
  }
  // sort a[lb:rb]
  // i.e., part of a between positions lb and rb (left and right bounds)
  private static op void sort(int [] a, int lb, int rb) {
    if (rb <= lb) return; // empty or 1-element array needs no sorting
    int pivot = a[lb];
```

```
    int lx = lb+1, rx = rb; // left and right indexes
    while (lx <= rx) {
      if (a[lx] <= pivot) {
        lx++;
      }
      else /* a[lx] > pivot */ {
        int temp = a[lx];  a[lx] = a[rx];  a[rx] = temp; // swap
        rx--;
      }
    }
    int temp = a[lb];  a[lb] = a[rx];  a[rx] = temp; // swap
    co sort(a, lb, rx-1);
    [] sort(a, lx, rb);
  }
}
```

The `sort` op-method first partitions argument array `a` into two parts based on the value of pivot element `a[lb]`. It then swaps the pivot with the rightmost value in the left partition. Finally, `sort` uses a concurrent invocation statement to recursively sort each of the partitions in parallel.

Concurrent Invocation With Postprocessing Code

Execution of a concurrent invocation statement that has postprocessing code is slightly more complicated than one that does not have such code. Again, all invocations are first started in parallel. Then, as each invocation terminates, the corresponding postprocessing block is executed, if there is one. Postprocessing blocks are executed one at a time, by the same process that initiated execution of the `co`; they are *not* executed concurrently and thus can change variables without requiring mutual exclusion. Execution of `co` terminates when all postprocessing blocks have terminated or when some postprocessing block executes a break statement.

As an example, consider the following program fragment:

```
int cnt = 0;
co ((int i = 0; i < a.length; i++; a[i] != 0)) p(i) {
  cnt++;
  System.out.println(cnt + " " + i);
}
```

It uses a quantifier to invoke `p` once for each value of `i` such that `a[i]` is non-zero. The postprocessing block counts the number of such invocations as they complete and outputs their indices. The use of the bound variable `i` in the postprocessing block is legal since the scope of the quantifier variables extends to the end of postprocessing block. Since the postprocessing blocks are executed one at a time, the update of `cnt` need not be protected by a critical section and the output will not be interleaved.

As another example, consider the following:

```
// read one copy of a replicated file,
// recording which response was received first
int whichOne;
co ((int i = 0; i < 4; i++)) fd[i].read(arguments) {
  whichOne = i;
  break;
}
```

It uses a quantifier to initiate four invocations. When any of the invocations terminates, the postprocessing code records, in `whichOne`, the index of that invocation and then executes a break statement. The effect of the `break` is to exit the entire `co` without waiting for the other invocations to complete.

If a postprocessing block exits before all invocations have terminated, the outstanding invocations are not canceled; they will still be serviced, but the invoker will not wait for them to complete nor get back results. Thus the `co` in the above example waits for just one of the four invocations it initiates to complete. The other three will presumably complete sometime, but the invoker is in no way affected by their completion; indeed, the invoker may no longer exist.

A postprocessing block in a concurrent invocation statement can also contain a continue statement. As in a for statement, execution of `continue` within a postprocessing block causes execution of that postprocessing block to terminate; the enclosing `co` then delays until another invocation terminates, or the `co` terminates if all invocations have terminated. ****

Examples: Voting Using The Concurrent Invocation Statement

**** either add as new subsection, using examples from Reference [1], or leave as exercises (first three below).

(if add, then add appropriate index entries and update roadmap at start of this section.)

Exercises

1.1 Write a concurrent invocation statement that polls `N` voters for yes or no votes and terminates when all responses have been received.

1.2 Write a concurrent invocation statement that polls `N` voters for yes or no votes and terminates when at least `N/2` responses have been received. Assume `N` is even.

1.3 Repeat the previous exercise, but terminate the concurrent invocation statement when a majority of identical responses have been received. Again assume `N` is even.

1.4 Recall the `PartialSums` class in Section 1.1. Consider replacing its two concurrent invocation statements by the single statement:

```
co ((int i = 0; i < N; i++)) save(i) { update(i); }
```

Would the program still be correct? Explain. Conjecture as to whether it would run faster or slower than before.

1.5 Recall the `PartialSums` class in Section 1.1. Consider replacing its two concurrent invocation statements by the single statement:

```
co ((int i = 0; i < N; i++)) saveAndUpdate(i);
```

The `saveAndUpdate` op-method combines the actions of the two op-methods `save` and `update`. Would the program still be correct? Explain. Conjecture as to whether it would run faster or slower than before.

1.6 Trace the execution of the quicksort program in the `Quicksort` class (see Section 1.1) on input "6 3 4 2 5 0 1". How many processes are created? What are the actual values of the indexes are passed to each instance of `sort`?

1.7 Consider rewriting the concurrent invocation statement that uses `cnt` (see Section 1.1) with the for statement:

```
for (int i = 0; i < a.length; i++) {
  if (a[i] != 0) {
    p(i);
    cnt++;
    System.out.println(cnt + " " + i);
  }
}
```

What differences, if any, are there between the executions of these two statements? Explain.

1.8 Assume `q` is serviced by an op-method. For each part below, what differences, if any, are there between the executions of the two statements?

(a)
```
co ((int i = 1; i <= 4; i++))  q(i);
for (int i = 1; i <= 4; i++) { q(i); }
```

(b)
```
co ((int i = 1; i <= 4; i++))  q(i);
for (int i = 1; i <= 4; i++) { send q(i); }
```

(c)
```
co ((int i = 1; i <= 4; i++))  send q(i);
for (int i = 1; i <= 4; i++) { send q(i); }
```

(d)

```
co ((int i = 1; i <= 4; i++))
  send q(i)  { System.out.println(i); }
for (int i = 1; i <= 4; i++) {
  send q(i);   System.out.println(i);
}
```

References

[1] H. N (Angela) Chan, E. Pauli, B. Y. Man, A. W. Keen, and R. A. Olsson. An exception handling mechanism for the concurrent invocation statement. In Jose C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, number 3648 in Lecture Notes in Computer Science, pages 699–709, Monte de Caparica, Portugal, August 2005. Springer–Verlag.

Index

****

**** also need update other index entries:

- list co under invocation statement
- add entry for co under quantifier entry