Refactoring for Concurrency in Java

Kyle Doren
University of Illinois at Urbana-Champaign
doren1@illinois.edu

Alexandria Shearer
Santa Clara University
ashearer@scu.edu

Motivations

Programmers forced to write parallel programs which is difficult, tedious, and error-prone.
We present two automated refactorings: Convert recursion to ForkJoinTask and Convert int to AtomicInteger which provide **thread safety**, **scalability**, and **better throughput**.
Tools help improve programmer productivity.

Tools will be present in a future official release of eclipse

Convert recursion to ForkJoinTask

Goal

A refactoring that converts a sequential divide-and-conquer recursive method to a parallel version using the ForkJoinTask framework (shipping with Java 7).

Example

```
public int fibonacci(int num) {
     if (num < 2)
          return num;
     else
          return fibonacci(num – 1)
               + fibonacci(num - 2);
}
```

```
public int fibonacci(int end) {
  int processorCount = availableProcessors();
  ForkJoinPool pool = new
     ForkJoinPool(processorCount);
  FibTask aFibTask = new FibTask(end);
  pool.invoke(aFibTask);
  return aFibTask.result;
}
protected void compute() {
A    if (num < 1000) {
          result = fibonacci_sequential(num);
          return;
     } else {
     FibTask task1 = new FibTask(num - 1);
     FibTask task2 = new FibTask(num - 2);
B    invokeAll(task1, task2);
     result = task1.result + task2.result;
     return;
     }
}
```

A. Sequential Threshold

- Determine when to use sequential algorithm
- Default value based on parameter analysis
- Less thread creation to keep overhead very low

B. Forking and joining tasks

- InvokeAll is syntatic sugar for forking the given tasks and then joining them and getting the result afterwards
- Easily scalable as can do multiple tasks with one call
- Simple to understand

Convert int to AtomicInteger

Goal

A refactoring that converts an `int` field to an `AtomicInteger`, thus performing all updates atomically (lock-free programming).

AtomicInteger API

Access	`int`	`AtomicInteger`
Read	`f`	`f.get()`
Write	`f = e`	`f.set(e)`
Cond. Write	`if (f==e) f=e1`	`f.compareAndSet(e,e1)`
Prefix Inc.	`++f`	`f.incrementAndGet()`
Postfix Inc.	`f++`	`f.getAndIncrement()`
Infix Add	`f = f + e`	`f.addAndGet(e)`
Add	`f += e`	`f.addAndGet(e)`
Prefix Dec.	`--f`	`f.decrementAndGet()`
Postfix Dec.	`f--`	`f.getAndDecrement()`
Infix Sub.	`f = f - e`	`f.addAndGet(-e)`
Subtract	`f -= e`	`f.addAndGet(-e)`

Removing a Synchronized Block or Modifier

Synchronized blocks ensures atomic execution through usage of locks. Replacing synchronization with more scalable atomic constructs enables more concurrency.

A synchronized block may be removed if and only if:

- There is only one call to the atomic API after refactoring
- There is an access to only one field
- There are no other side effects within the parameters to the atomic call, like method invocations or infix expressions

Example

```
class Counter {
  int count= 0;

  int synchronized inc() {
    return count++;
  }

  void synchronized mult() {

    count = count *2;
  }

  void synchronized check()
  {
    if (count == 5) {
      count = 3;
    }
  }
}
```

```
class Counter {
  AtomicInteger count
     = new AtomicInteger(0);

  int inc() {
    return count.getAndIncrement();
  }

  void synchronized mult() {
    //TODO non-atomic call
    count.set(count.get() * 2);
  }

  void check()
  {

    count.compareAndSet(5, 3);

  }
}
```


Acknowledgements

Thanks to our mentor, Dr. Danny Dig, one of his PhD students Cosmin Radoi, and program administrators Craig Zilles and Jill Peckham.

Passionate on Parallel
2012 REU Program

parallel COMPUTING INSTITUTE

NSF

This REU is co-funded by the ASSURE program of the Department of Defense in partnership with the National Science Foundation REU Site program under Award No. 1004311. Funding for this project was also provided by SRC.