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

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

```java
public int fibonacci(int num) {
    int processorCount = availableProcessors();
    ForkJoinPool pool = new ForkJoinPool(processorCount);
    ForkTask aFibTask = new ForkTask();
    pool.invoke(aFibTask);
    return aFibTask.result;
}
```

```java
protected void compute() {
    A if (num < 1000) {
        result = fibonacci_sequential(num);
        return;
    } else {
        ForkTask task1 = new ForkTask(num - 1);
        ForkTask task2 = new ForkTask(num - 2);
        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 syntactic 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

<table>
<thead>
<tr>
<th>Access</th>
<th>int</th>
<th>AtomicInteger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>f = x</td>
<td>f.get()</td>
</tr>
<tr>
<td>Write</td>
<td>f = x</td>
<td>f.set(x)</td>
</tr>
<tr>
<td>Cond. Write</td>
<td>if (x = x) f = x</td>
<td>f.compareAndSet(x, x)</td>
</tr>
<tr>
<td>add</td>
<td>f += x</td>
<td>f.addAndGet(x)</td>
</tr>
<tr>
<td>access</td>
<td>f + x</td>
<td>f.get() + x</td>
</tr>
<tr>
<td>Sub</td>
<td>f -= x</td>
<td>f.subtract(x)</td>
</tr>
<tr>
<td>Sub</td>
<td>f - x</td>
<td>f.get() - x</td>
</tr>
<tr>
<td>Sub</td>
<td>f += x</td>
<td>f.addAndGet(x)</td>
</tr>
</tbody>
</table>

Removing a Synchronized Block or Modifier

Synchronized blocks ensure 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

```java
class Counter {
    int count = 0;
    public void inc() {
        count++; //TODO non-atomic call
    }
}
```

```java
class Counter {
    AtomicInteger count = new AtomicInteger(0);
    public void inc() {
        count.incrementAndGet();
    }
}
```

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

This REU is co-funded by the ADVANCE program of the Department of Defense in partnership with the National Science Foundation REU Site program under Award No. 1049311. Funding for this project was also provided by SRC.

Passionate on Parallel
2012 REU Program