Refactoring Sequential Java Code for Concurrency via Concurrent Libraries

Danny Dig (MIT → UPCRC Illinois)

John Marrero (MIT)
Michael D. Ernst (MIT → U of Washington)

ICSE 2009
The Shift to Multicores Demands Work from Programmers

Users expect that new generations of computers run faster.

Programmers must find and exploit parallelism.

A major programming task: refactoring sequential apps for concurrency.
public class Counter {
    int value = 0;

    public int getCounter() {
        return value;
    }

    public void setCounter(int counter) {
        this.value = counter;
    }

    public int inc() {
        return ++value;
    }
}

Updating Shared Data Must Execute Atomically

read value
compute value + 1
store value
public class Counter {
    int value = 0;

    public int getCounter() {
        return value;
    }

    public void setCounter(int counter) {
        this.value = counter;
    }

    public synchronized int inc() {
        return ++value;
    }
}

Locking Has Too Much Overhead
public class Counter {
    int value = 0;

    synchronized int getCounter() {
        return value;
    }

    synchronized void setCounter(int counter) {
        this.value = counter;
    }

    public synchronized int inc() {
        return ++value;
    }
}
Refactoring for Concurrency: Goals

**Thread-safety**
- preserve invariants under multiple threads

**Scalability**
- performance improves with more parallel resources

Delegate the challenges to concurrent libraries:
- java.util.concurrent in Java 5
  - addresses both thread-safety and scalability

AtomicInteger from java.util.concurrent in the Counter example
Refactoring For Concurrency is Challenging

Manual refactoring to `java.util.concurrent` is:

- **Labor-intensive:** changes to many lines of code  
  (e.g., 1019 LOC changed in 6 open-source projects when converting to AtomicInteger and ConcurrentHashMap)

- **Error-prone:** the programmer can use the wrong APIs  
  (e.g., 4x misused incrementAndGet instead of getAndIncrement)

- **Omission-prone:** programmer can miss opportunities to use the new, efficient APIs  
  (e.g., 41x missed opportunities in the 6 open-source projects)

**Goal:** make concurrent libraries easy to use
Outline

Concurreencer, our interactive refactoring tool

Making programs **thread-safe**
- convert int field to AtomicInteger
- convert HashMap field to ConcurrentHashMap

Making programs **multi-threaded**
- convert recursive divide-and-conquer to task parallelism

Evaluation
AtomicInteger in java.util.concurrent

Lock-free programming on single integer variable

Update operations execute atomically

Uses efficient machine-level atomic instructions (Compare-and-Swap)

Offers both thread-safety and scalability
Convert int to AtomicInteger

Initialization

Read Access

Write Access

Prefix Expression
Transformations: Removing Synchronization Block

```java
class Counter 
{
    int value = 0;
    ...

    public synchronized int inc() {
        return ++value;
    }
}
```

```java
public class Counter {
    AtomicInteger value = new AtomicInteger(0);
    ...

    public int inc() {
        return value.incrementAndGet();
    }
}
```

Concurrencer removes the synchronization iff for all blocks:
- after conversion, the block contains exactly **one call** to the atomic API
- the block accesses **a single** field
Outline

Concurrencer, our interactive refactoring tool

Making programs thread-safe
- convert int field to AtomicInteger
- convert HashMap field to ConcurrentHashMap

Making programs multi-threaded
- convert recursive divide-and-conquer to task parallelism

Evaluation
“Put If Absent” Pattern Must Be Atomic

```java
HashMap<String, File> cache = new HashMap<String, File>();

public void service(Request req, Response res) {
    ...
    String uri = req.requestURI().toString();
    ...
    File resource = cache.get(uri);
    if (resource == null) {
        resource = new File(rootFolder, uri);
        cache.put(uri, resource);
    }
    ...
}
```
Locking the Entire Map Reduces Scalability

```java
HashMap<String, File> cache = new HashMap<String, File>();

public void service(Request req, Response res) {
    ...
    String uri = req.requestURI().toString();
    ...
    synchronized(lock){
        File resource = cache.get(uri);

        if (resource == null) {
            resource = new File(rootFolder, uri);
            cache.put(uri, resource);
        }
    }
    ...
}
```
ConcurrentHashMap in java.util.concurrent

Uses *fine-grained* locking (e.g., lock-striping)

\( \mathbb{N} \) locks, each guarding a subset of the hash buckets

Enables **all readers** to run concurrently

Enables a **limited number of writers** to update the map concurrently
New APIs in ConcurrentHashMap

ConcurrentHashMap provides three new update methods:
- putIfAbsent(key, value)
- replace(key, oldValue, newValue)
- remove(key, value)

Each update method:
- supersedes several calls to Map operations,
- but executes atomically
Concurrencer Replaces Update Operation with putIfAbsent()

```
HashMap cache;

String uri =
    req.requestURI().toString();
...

File resource = cache.get(uri);
if (resource == null) {
    resource = new File(rootFolder, uri);
    cache.put(uri, resource);
}
```

```
ConcurrentHashMap cache;

String uri =
    req.requestURI().toString();
...

    cache.putIfAbsent(uri,
    new File(rootFolder, uri);
```
Enabling program analysis for
Convert to ConcurrentHashMap

The creational code is always invoked before calling putIfAbsent

#1. Side-effects analysis
   - conservative analysis (MOD Analysis) warns the user about potential side-effects

#2. Read/write analysis determines whether to delete testValue
Outline

Concurrencer, our interactive refactoring tool

Making programs thread-safe
  - convert int field to AtomicInteger
  - convert HashMap field to ConcurrentHashMap

Making programs multi-threaded
  - convert recursive divide-and-conquer to task parallelism

Evaluation
Challenge: How to Keep All Cores Busy

Parallelize computationally intensive problems (fine-grained parallelism)

Many computationally intensive problems take the form of divide-and-conquer

Classic examples: mergesort, quicksort, search, matrix / image processing algorithms

Sequential divide-and-conquer are good candidates for parallelization when tasks are completely independent

- operate on different parts of the data
- solve different subproblems
Sequential and Parallel Divide-and-Conquer

```c
solve (Problem problem) {
    if (problem.size <= BASE_CASE)
        solve problem directly
    else {
        split problem into tasks
        solve each task
        compose result from subresults
    }
}
```
ForkJoinTask Framework in Java 7

Main class ForkJoinTask (a lightweight thread-like entity)

- `fork()` spawns a new task
- `join()` waits for task to complete
- `forkJoin()` syntactic sugar for spawn/wait
- `compute()` encapsulates the task's computation

Framework contains a work-stealing scheduler with good load balancing [Lea'00]
Concurrencer Parallelizes MergeSort

reimplement original method
subclass RecursiveAction
fields for input/output
task constructor
implement compute()
replace basecase with SeqThr
create parallel tasks
forkJoin the parallel tasks
fetch results from tasks

copy original sort method
for use in the sequential case
Outline

Concurrencer, our interactive refactoring tool

Making programs thread-safe
- convert `int` field to `AtomicInteger`
- convert `HashMap` field to `ConcurrentHashMap`

Making programs multi-threaded
- convert recursive divide-and-conquer to task parallelism

Evaluation
Research Questions

Q1: Is Concurrencer useful? Does it save programmer effort?

Q2: Is the refactored code correct? How does manually-refactored code compare with code refactored with Concurrencer?

Q3: What is the speed-up of the parallelized code?
Case-study Evaluation

Case-study 1:
- 6 open-source projects using AtomicInteger or ConcurrentHashMap
- used Concurrenancer to refactor the same fields as the developers did
- evaluates usefulness and correctness

Case-study 2:
- used Concurrenancer to refactor 6 divide-and-conquer algorithms
- evaluates usefulness, correctness and speed-up
# Q1: Is Concurrencer Useful?

<table>
<thead>
<tr>
<th>refactoring</th>
<th>project</th>
<th># of refactorings</th>
<th>LOC changed</th>
<th>LOC Concurrencer can handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert int field to AtomicInteger</td>
<td>MINA, Tomcat, Struts, GlassFish, JaxLib, Zimbra</td>
<td>64</td>
<td>401</td>
<td>100.00%</td>
</tr>
<tr>
<td>Convert HashMap field to ConcurrentHashMap</td>
<td>MINA, Tomcat, Struts, GlassFish, JaxLib, Zimbra</td>
<td>77</td>
<td>618</td>
<td>91.70%</td>
</tr>
<tr>
<td>Convert recursion to FJTask</td>
<td>mergeSort, fibonacci, maxSumConsecutive, matrixMultiply, quickSort, maxTreeDepth</td>
<td>6</td>
<td>302</td>
<td>100.00%</td>
</tr>
</tbody>
</table>
Q2: Is the Refactored Code Correct?

1. Thread-safety: omission of atomic methods

<table>
<thead>
<tr>
<th></th>
<th>putIfAbsent(key, value)</th>
<th>remove(key, value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>potential usages</td>
<td>73</td>
<td>10</td>
</tr>
<tr>
<td>human omissions</td>
<td>33</td>
<td>8</td>
</tr>
<tr>
<td>Concurreencer omissions</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

2. Incorrect values: errors in using atomic methods

Open-source developers misused `getAndIncrement` instead of `incrementAndGet` 4 times

- can result in off-by-one values

Concurreencer used the correct method
Q3: What is the Speedup of the Parallelized Algorithms?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speedup 2 Cores</th>
<th>Speedup 4 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>mergeSort</td>
<td>1.98x</td>
<td>3.47x</td>
</tr>
<tr>
<td>maxTreeDepth</td>
<td>1.55x</td>
<td>2.38x</td>
</tr>
<tr>
<td>maxSumConsecutive</td>
<td>1.78x</td>
<td>3.16x</td>
</tr>
<tr>
<td>quickSort</td>
<td>1.84x</td>
<td>3.12x</td>
</tr>
<tr>
<td>fibonacci</td>
<td>1.94x</td>
<td>3.82x</td>
</tr>
<tr>
<td>matrixMultiply</td>
<td>1.95x</td>
<td>3.77x</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>1.84x</strong></td>
<td><strong>3.28x</strong></td>
</tr>
</tbody>
</table>
Conclusions

Introducing concurrency is hard

Convert “introduce concurrency” into “introduce parallel library”
- still tedious, error- and omission-prone

Concurrencer is more effective than manual refactoring

http://refactoring.info/tools/Concurrencer

Future work:
- support more refactorings, e.g., convert Array to ParallelArray
BACK UP slides
## Convert int to AtomicInteger - transformations -

<table>
<thead>
<tr>
<th>Access</th>
<th>int</th>
<th>AtomicInteger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>f</td>
<td>f.get()</td>
</tr>
<tr>
<td>Write</td>
<td>f = e</td>
<td>f.set(e)</td>
</tr>
<tr>
<td>Cond. Write</td>
<td>if (f==e) f=e₁</td>
<td>f.compareAndSet(e,e₁)</td>
</tr>
<tr>
<td>Prefix Inc.</td>
<td>++f</td>
<td>f.incrementAndGet()</td>
</tr>
<tr>
<td>Postfix Inc.</td>
<td>f++</td>
<td>f.getAndIncrement()</td>
</tr>
<tr>
<td>Infix Add</td>
<td>f = f + e</td>
<td>f.addAndGet(e)</td>
</tr>
<tr>
<td>Add</td>
<td>f += e</td>
<td>f.addAndGet(e)</td>
</tr>
<tr>
<td>Prefix Dec.</td>
<td>--f</td>
<td>f.decrementAndGet()</td>
</tr>
<tr>
<td>Postfix Dec.</td>
<td>f--</td>
<td>f.getAndDecrement()</td>
</tr>
<tr>
<td>Infix Sub.</td>
<td>f = f - e</td>
<td>f.addAndGet(-e)</td>
</tr>
<tr>
<td>Subtract</td>
<td>f -= e</td>
<td>f.addAndGet(-e)</td>
</tr>
</tbody>
</table>
Two consecutive `inc()` return same value

```java
public int inc() {
    return ++value;
}
```

Thread A

- Value = 9
- Value + 1 = 10
- Value = 10

Thread B

- Value = 9
- Value + 1 = 10
- Value = 10
Outline

Concurrencer, our interactive transformation tool
- convert int field to AtomicInteger

- convert HashMap field to ConcurrentHashMap

- convert recursive divide-and-conquer to ForkJoin parallelism

Empirical evaluation
Basic Patterns that Concurrencer Replaces with
\texttt{map.putIfAbsent(key, value)}

(i) \hspace{1em} \textbf{if} (!\texttt{map.containsKey(key)})
\hspace{1em} \texttt{map.put(key, value)};

(ii) \hspace{1em} \texttt{boolean keyExists = map.containsKey(key);}
\hspace{1em} \textbf{if} (!\texttt{keyExists})
\hspace{2em} \texttt{map.put(key, value)};

(iii) \hspace{1em} \textbf{if} (\texttt{map.get(key)} == \texttt{null})
\hspace{2em} \texttt{map.put(key, value)};

(iv) \hspace{1em} \texttt{Object testValue = map.get(key);}
\hspace{2em} \textbf{if} (testValue == \texttt{null})
\hspace{3em} \texttt{map.put(key, value)};
private ConcurrentHashMap<String, Component> components;

public void addComponent(String subdomain)
    throws ComponentException {
    Component existingComponent = components.get(subdomain);
    if (existingComponent != null) {
        throw new ComponentException("Domain already taken");
    }
    components.put(subdomain, component);
Read/write analysis for `putIfAbsent()` with creational code

```java
public void service(Request req, Response res) {
    ...
    File resource = cache.get(uri);
    if (resource == null) {
        for (int i; i < uri.length; i++) {
            ... initialization code
        }
        resource = new File(rootFolder, uri);
        cache.put(uri, resource);
    }

    print(resource);
}
```

```java
public void service(Request req, Response res) {
    ...
    File resource = cache.get(uri);
    File newResource = createResource();
    if (cache.putIfAbsent(uri, newResource) == null) {
        resource = newResource;
    }

    print(resource);
}
```

```java
File createResource() {
    for (int i; i < uri.length; i++) {
        ... initialization code
    }
    resource = new File(rootFolder, uri);
    return resource;
}
```
Using `putIfAbsent()` with creational code

```java
public void service(Request req, Response res) {
    ... 
    File resource = cache.get(uri);
    if (resource == null) {
        for (int i; i < uri.length; i++) {
            ... initialization code
        }
        resource = new File(rootFolder, uri);
        cache.put(uri, resource);
    }
}
```

```java
public void service(Request req, Response res) {
    ... 
    cache.putIfAbsent(uri, createResource());
}
```

```java
File createResource() {
    for (int i; i < uri.length; i++) {
        ... initialization code
    }
    resource = new File(rootFolder, uri);
    return resource;
}
```
Code Patterns: `remove()` and `replace()`

```java
if(hm.containsKey("a_key"))
    hm.remove("a_key");

hm.remove("a_key");
...

if(hm.containsKey("a_key"))
    hm.put("a_key", "a_value");

hm.replace("a_key", "a_value");
...
```
Enabling program analysis for
Convert to ConcurrentHashMap

#1. Read/write analysis determines whether to delete testValue:

parameters:

Statements: BEFORE_PUT, AFTER_PUT
variables: testValue, newValue

1 if !isReadIn(AFTER_PUT, testValue) then
2 delete Variable(testValue);
3 else
4 // testValue is read later, do not delete it
5 if isWrittenIn(BEFORE_PUT, testValue)
6 \text{return}(putIfAbsent()) == success then
7 testValue ← newValue
Outline

Concurrencer, our interactive transformation tool
- convert int field to AtomicInteger

- convert HashMap field to ConcurrentHashMap

- convert recursive divide-and-conquer to ForkJoin parallelism

Empirical evaluation

Interactive, first-class program transformations
Outline

Concurrencer, our interactive transformation tool
  - convert int field to AtomicInteger

  - convert HashMap field to ConcurrentHashMap

  - convert recursive divide-and-conquer to ForkJoin parallelism

Empirical Evaluation
Outline

Concurrencer, our extension to Eclipse's refactoring engine

- convert int field to AtomicInteger

- convert HashMap field to ConcurrentHashMap

- convert recursive divide-and-conquer to Fork/Join parallelism

Empirical Evaluation
Example MergeSort with Fork/Join Framework

class MergeSort extends RecursiveAction {
    int[] toSort;
    int[] result;   // sorted array

    MergeSort(int[] toSort){
        ...
    }

    protected void compute() {
        if (toSort.length < Sequential_Threshold) {
            result = seqMergeSort(toSort);
        } else {
            MergeSort leftTask = new MergeSort(left);
            MergeSort rightTask = new MergeSort(right);
            forkJoin(leftTask, rightTask);
            result = merge(leftTask.result, rightTask.result);
        }
    }

    private int[] seqMergeSort(int[] toSort) {
        if (toSort.length == 1)
            return toSort;
        else { // left = 1st half ; right = 2nd half
            seqMergeSort(left); seqMergeSort(right);
            return merge(left, right);
        }
    }
}
Transformations for ExtractFJTask

```java
class MergeSort extends RecursiveAction {
    int[] toSort;
    int[] result; // sorted array
    MergeSort(int[] listToSort) {
        ...
    }

    protected void compute() {
        if (toSort.length < Sequential_Threshold) {
            result = seqMergeSort(toSort);
        } else {
            MergeSort leftTask = new MergeSort(left);
            MergeSort rightTask = new MergeSort(right);
            forkJoin(leftTask, rightTask);
            result = merge(leftTask.result, rightTask.result);
        }
    }

    private int[] seqMergeSort(int[] toSort) {
        if (toSort.length == 1)
            return toSort;
        else {
            seqMergeSort(left); seqMergeSort(right);
            return merge(left, right);
        }
    }
}
```

Subclass FJTask
- fields for args, result
- constructor
Transformations for ExtractFJTask

class MergeSort extends RecursiveAction {
    int[] toSort;
    int[] result;  // sorted array
    MergeSort(int[] listToSort) {
        ...
    }

    protected void compute() {
        if (toSort.length < Sequential_Threshold) {
            result = seqMergeSort(toSort);
        } else {
            MergeSort leftTask = new MergeSort(left);
            MergeSort rightTask = new MergeSort(right);
            forkJoin(leftTask, rightTask);
            result = merge(leftTask.result, rightTask.result);
        }
    }

    private int[] seqMergeSort(int[] toSort) {
        if (toSort.length == 1)
            return toSort;
        else {
            seqMergeSort(left); seqMergeSort(right);
            return merge(left, right);
        }
    }
}
Transformations for ExtractFJTask:
Reimplement the original sort method

```java
public int[] sort(int[] toSort) {
    ForkJoinExecutor pool =
        new ForkJoinPool(Runtime.getRuntime().availableProcessors());

    MergeSort sortObj = new MergeSort(toSort);

    pool.invoke(sortObj);

    return sortObj.result;
}
```
Computation Tree for MergeSort
Fork/Join Framework in Java 7

The nature of fork/join tasks:
- tasks are CPU-bound
- tasks only need to synchronize across subtasks, thus need efficient scheduling
- many tasks (e.g., millions)

Threads are not a good fit for this kind of computation
- heavyweight: overhead (creating, scheduling, destroying) might outperform useful computation

Fork/Join tasks are lightweight:
- start a pool of worker threads (= # of CPUs)
- map many tasks to few worker threads
- effective scheduling based on work-stealing
Fork/Join Framework in Java 7

Scheduling based on work-stealing (a-la Cilk)

- Each worker thread maintains a scheduling DEQUE
- Subtasks forked from tasks in a worker thread are pushed on the same dequeue
- Worker threads process their own deques in LIFO order
- When idle, worker threads steal tasks from other workers in FIFO order

Advantages:

- low contention for the DEQUE
- stealing from the tail ensures getting larger chunks of work, thus stealing becomes infrequent
Example Fibonacci with Fork/Join Parallelism

class Fibonacci {
    int number;
    int result;

    Fibonacci(int n) {
        number = n;
    }

    protected void compute() {
        if (number < Sequential_Threshold) {
            result = seqFibonacci(number);
        } else {

            INVOKE_IN_PARALLEL {
                Fibonacci f1 = new Fibonacci(number-1);
                Fibonacci f2 = new Fibonacci(number-2);
            }
            result = f1.result + f2.result;
        }
    }

    private int seqFibonacci(int number) {
        if (number < 2)
            return number;
        return seqFibonacci(number - 1) + seqFibonacci(number - 2);
    }
}
Computing max value from an array

class ComputeMax extends RecursiveAction{
    int max;
    int[] array;
    private int start;
    private int end;

    public ComputeMax(int[] randomArray, int i, int length) {
        this.array = randomArray;
        this.start = i;
        this.end = length;
    }

    protected void compute() {
        if (end - start < 500)
            computeMaxSequentially();
        else {
            int midrange = (end - start) / 2;
            ComputeMax left = new ComputeMax(array, start, start+midrange);
            ComputeMax right = new ComputeMax(array, start + midrange, end);
            forkJoin(left, right);
            max = Math.max(left.max, right.max);
        }
    }

    public void computeMaxSequentially() {
        max = Integer.MIN_VALUE;
        for (int i = start; i < end; i++) {
            max = Math.max(max, array[i]);
        }
    }
}
Fork/Join Transformations

1. Create a task class which extends one of the subclasses of FJTask
   - fields to hold arguments and result
   - constructor which initializes the arguments
   - define `compute()`

2. Implementing `compute()`
   - replace the original base case with threshold check
   - create subtasks, fork them in parallel, join each one of them
   - combine results

3. Replace the call to the original method with one that creates the task pool