

Refactoring Sequential Java Code for Concurrency via Concurrent Libraries

Danny Dig (UIUC)

**John Marrero (MIT)
Michael D. Ernst (MIT)**

UPCRC seminar – Nov 14th

Challenges Posed by the Shift to Multicores

The computing hardware industry shifted to multicores

Application programmers are required to find and exploit parallelism to gain performance improvements

A **major** programming task: **refactoring** sequential apps for concurrency

Challenges

- thread-safety (i.e., app runs correctly from multiple threads)
- scalability (i.e., performance improves with more parallel resources)

Java Library Support for Concurrency

Java 5 introduces `java.util.concurrent` (`j.u.c.`)

Thread-safety is built-in

- `Atomic*` classes (e.g., `AtomicInteger`) offer atomic update APIs
- data structures (e.g., `ConcurrentHashMap`)

Scalability support

- `Atomic*` classes are lock-free
- data structures are optimized for scalability
- `ForkJoinTask` framework for fine-grained parallelism (Java 7)

Need Tool Support for Converting to `j.u.c.`

Java programmer needs to refactor existing code

Manual refactoring to `j.u.c.` is:

- **Tedious:** updates to several lines of code
(e.g., 1019 LOC changed in 6 open-source projects)
- **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)



Outline

Concurrancer, 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

Motivational Example: Need Atomic Execution

```
public class Counter {  
    int value = 0;  
  
    public int getCounter() {  
        return value;  
    }  
  
    public void setCounter(int counter) {  
        this.value = counter;  
    }  
  
    public int inc() {  
        return ++value;  
    }  
}
```

→ get value
do value + 1
set value

Locking Has Too Much Overhead

```
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;  
    }  
}
```

AtomicInteger

Lock-free programming on **single** variables

Update operations execute atomically without blocking

Uses efficient machine-level atomic instructions (*Compare-and-Swap*) available on most modern processors

Offers both **thread-safety** and **scalability**

Convert int to AtomicInteger

Changes to be performed

- Counter.java - testConcurrency/src/p
- Update Imports
- Counter

Counter.java

Original Source	Refactored Source
<pre>public class Counter { private int value = 0; public int getCounter() { return value; } public void setCounter(int counter) { value = counter; } public int inc() { return ++value; } }</pre>	<pre>public class Counter { private AtomicInteger value = new AtomicInteger(0); public int getCounter() { return value.get(); } public void setCounter(int counter) { value.set(counter); } public int inc() { return value.incrementAndGet(); } }</pre>

Initialization

Read Access

Write Access

Prefix Expression

Preview > OK Cancel

Transformations: Removing Synchronization Block

```
public class Counter {  
    int value = 0;  
    ...  
    public synchronized int inc() {  
        return ++value;  
    }  
}
```

```
public class Counter {  
    AtomicInteger value = new AtomicInteger(0);  
    ...  
    public int inc() {  
        return value.incrementAndGet();  
    }  
}
```

Concurrenter removes the synchronization block iff:

- after conversion, the block contains exactly one call to the atomic API
- the block contains updates to one single field

Outline

Concurrancer, our interactive transformation tool

- convert `int` field to `AtomicInteger`

- convert `HashMap` field to `ConcurrentHashMap`

- convert recursive divide-and-conquer to `ForkJoin` parallelism

Empirical Evaluation

About interactive program transformations

“Put If Absent” Map Update Needs to Execute Atomically

```
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 Drastically Reduces Scalability

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



ConcurrentHashMap

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

Number of locks, each guarding a part of the hash buckets

Enables **all readers** to run concurrently

Enables a **limited number of writers** to update the map
concurrently

Added APIs in ConcurrentHashMap

`ConcurrentHashMap` implements `Map`

Three new API methods in `ConcurrentHashMap`:

- `putIfAbsent(key, value)`
- `replace(key, oldValue, newValue)`
- `remove(key, value)`

Each update API method:

- supersedes several calls to `Map` operations,
- but execute atomically

Transformations to Convert to ConcurrentHashMap

Initialization

Map Updates

Transformations: replace update operation with putIfAbsent()

```
String uri =
    req.requestURI().toString();
...
File resource = cache.get(uri);

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

```
String uri =
    Req.requestURI().toString();
...

cache.putIfAbsent(uri,
    new File(rootFolder, uri));
```

Using putIfAbsent () with creational code

```
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);
    }
}
```

```
public void service(Request req,
                    Response res) {
    ...

    cache.putIfAbsent(uri,
                      createResource());

}

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

Enabling program analysis

#1. Data-flow analysis to determine whether:

- the created value is read after `map.put` (if so, need to store in temporary variable)
- the variable which holds the old value is written in the conditional code and read afterwards (if so, write to it if `putIfAbsent` is successful)

#2. Side-effects analysis

- before calling `putIfAbsent`, the created value must be available, thus the creational code is always executed
- the creational code might have side-effects
- conservative analysis warns the user about potential side-effects

Outline

Concurrancer, our interactive transformation tool

- convert `int` field to `AtomicInteger`
- convert `HashMap` field to `ConcurrentHashMap`
- convert recursive divide-and-conquer to `ForkJoin` parallelism

Empirical Evaluation

About interactive program transformations

Challenge: How to Keep All Cores Busy

Identify computationally intensive problems and parallelize them (fine-grained parallelism)

Many computationally intensive problems take the form of recursive 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

Recursive Divide-and-Conquer

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

```
solve (Problem problem) {  
  if (problem.size <= SEQ_THRESHOLD )  
    solve problem sequentially  
  else {  
    split problem into tasks  
    IN Parallel (fork){  
      solve each task  
    } wait for all tasks (join)  
    compose result from subresults  
  }  
}
```

Example MergeSort with Fork/Join Parallelism

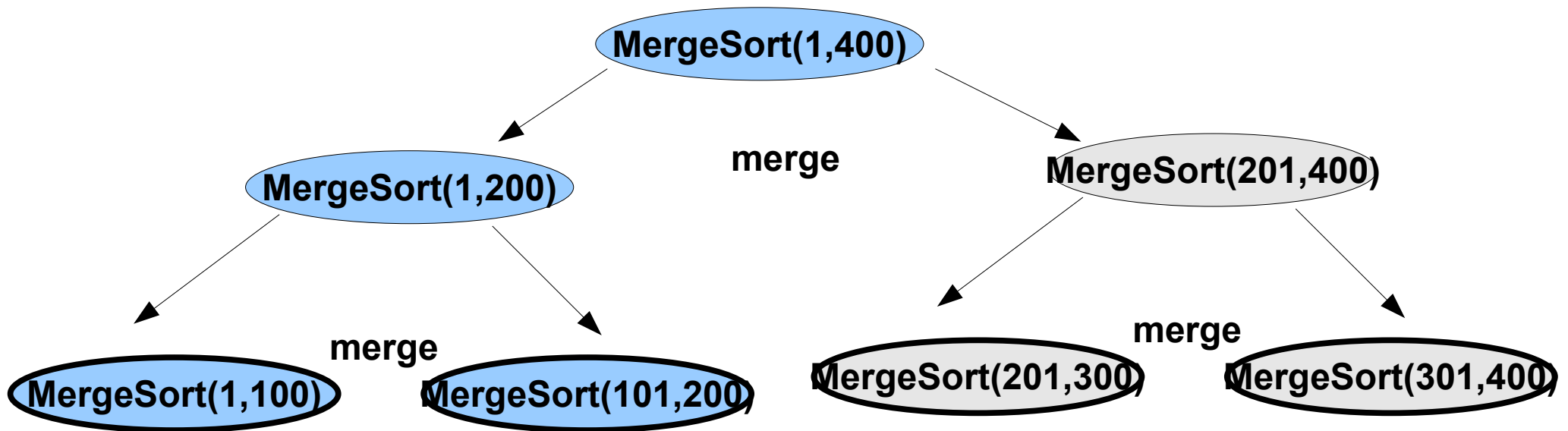
```
class MergeSort {
    int[] toSort;
    int[] result;    // sorted array

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

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

Computation Tree for MergeSort



Fork/Join Framework in Java 7

Main class ForkJoinTask (a lightweight thread-like entity)

- `fork()`
- `join()`
- `forkJoin()`
- `isDone()`
- `compute()`

Subclasses: RecursiveTask, RecursiveAction

ForkJoinExecutor is the entry class for starting ForkJoinTasks

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 {
            seqMergeSort(left); seqMergeSort(right);
            return merge(left, right);
        }
    }
}
```

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);
        }
    }
}
```

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);
        }
    }
}
```

- Implement compute ()
- replace base case with SequentialThreshold
 - fork, join subtasks
 - combine results

Transformations for ExtractFJTask

Reimplement the original sort method

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

    MergeSort sortObj = new MergeSort(toSort);

    pool.invoke(sortObj);

    return sortObj.result;
}
```

Outline

Concurrancer, our interactive transformation tool

- convert `int` field to `AtomicInteger`
- convert `HashMap` field to `ConcurrentHashMap`
- convert recursive divide-and-conquer to `ForkJoin` parallelism

Empirical Evaluation

About interactive program transformations



Empirical Evaluation

Q1: Is Concurrencyer useful? Does it save programmer effort when converting to j.u.c.?

Q2: How does manually-refactored code compare with code refactored with our tool in terms of using the correct APIs and identifying all opportunities?

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

Experiment 1:

- 6 open-source projects using `AtomicInteger` fields or `ConcurrentHashMap` fields
- used Concurrencyer to refactor the **same fields** as the developers did and answer Q1 and Q2

Experiment 2:

- used Concurrencyer to refactor 6 divide-and-conquer algorithms and answer Q1 and Q3



Q1: Is Concurrency useful?

refactoring	project	# of refactorings	LOC changed	LOC Concurrency can handle
Convert int field to AtomicInteger	MINA, Tomcat, Struts, GlassFish, JaxLib, Zimbra	64	401	401
Convert HashMap field to ConcurrentHashMap	MINA, Tomcat, Struts, GlassFish, JaxLib, Zimbra	77	618	567
Convert recursion to FJTask	mergeSort, fibonacci, maxSumConsecutive, matrixMultiply, quickSort, maxTreeDepth	6	302	302

Q2: How does manually and automated refactored code compare?

1. Omissions to use atomic APIs

<code>putIfAbsent(key, value)</code>			<code>remove(key, value)</code>		
potential usages	human omissions	Concurrencer omissions	potential usages	human omissions	Concurrencer omissions
73	33	10	10	8	0

2. Errors in using atomic APIs

Open-source developers 4 times erroneously used `getAndIncrement` instead of `incrementAndGet`

- can result in off-by-one values if the field is read in the same statement

 Concurrencer used the correct APIs

Q3: What is the speedup of the parallelized algorithms?

	speedup 2 cores	speedup 4 cores
mergeSort	1.98x	3.47x
maxTreeDepth	1.55x	2.38x
maxSumConsecutive	1.78x	3.16x
quickSort	1.84x	3.12x
fibonacci	1.94x	3.82x
matrixMultiply	1.95x	3.77x
Average	1.84x	3.28x

Outline

Concurrancer, our interactive transformation tool

- convert `int` field to `AtomicInteger`
- convert `HashMap` field to `ConcurrentHashMap`
- convert recursive divide-and-conquer to `ForkJoin` parallelism

Empirical Evaluation

About interactive program transformations

Interactive, First-class Program Transformations

Why interactive?

- because the human knows the problem domain
- human does the creative job, the tool does the tedious job

Why first-class?

Record transformations in the IDE, distribute the patch to others, replay transformations



Explicit documentation for parallelization



Interactive, First-class Program Transformations

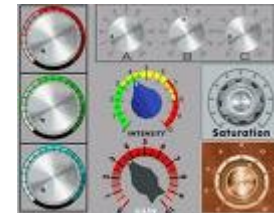
The IDE can provide two views of the same code



Compose transformations, assemble them in different combinations



Transformations can provide explicit “knobs” for autotuners



Same transformation can have several platform-specific implementations



Conclusions and Future Work

Refactoring sequential code to concurrency is non trivial

- refactoring via concurrent libraries is still tedious and prone to human errors and omissions

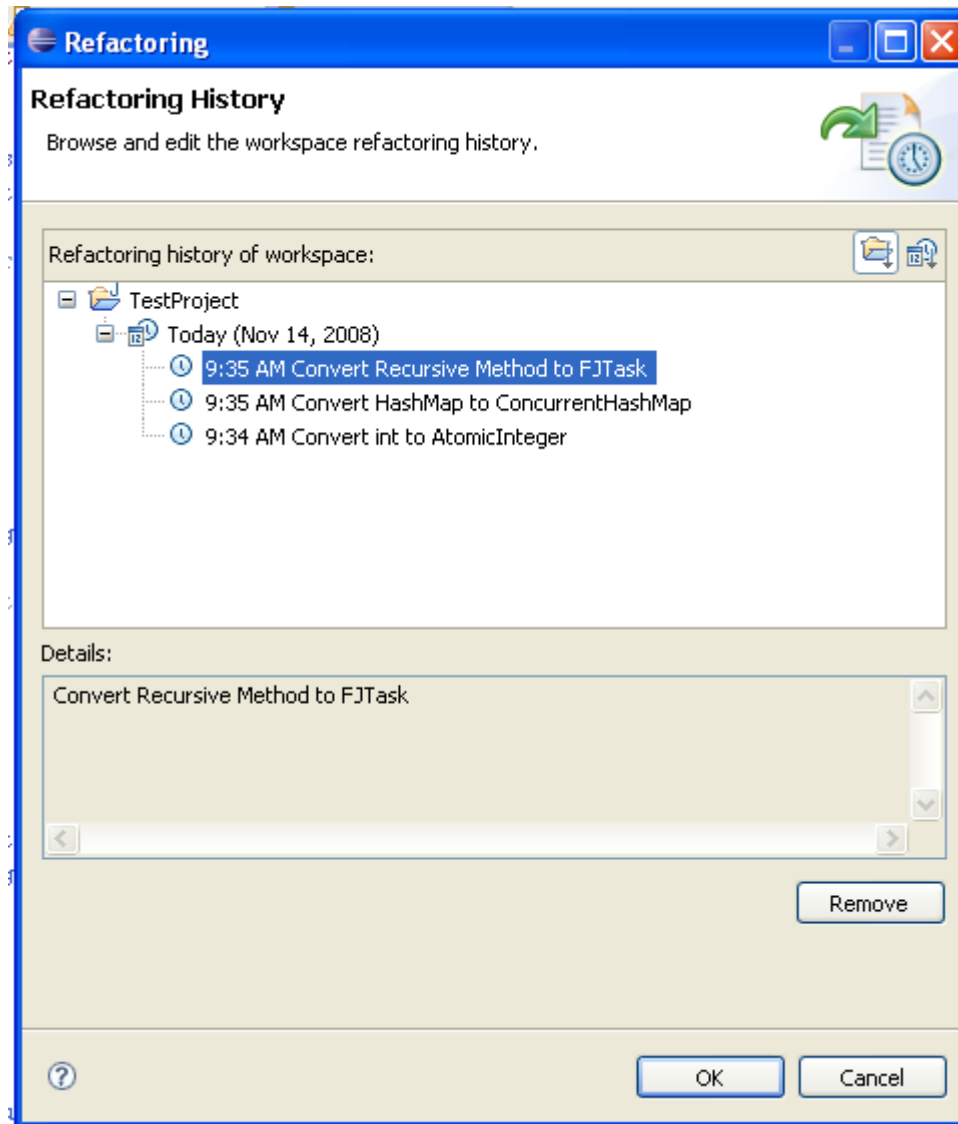
Concurrancer, an interactive refactoring tool is more effective than human developers

<http://refactoring.info/tools/Concurrancer>

Future work:

- support other `Atomic*` and `Collection` classes
- convert `Array` to `ParallelArray`
- extract a snapshot of code into a `Task` (e.g., in Learning-based Java)
- convert Java to Deterministic Parallel Java (DPJ)

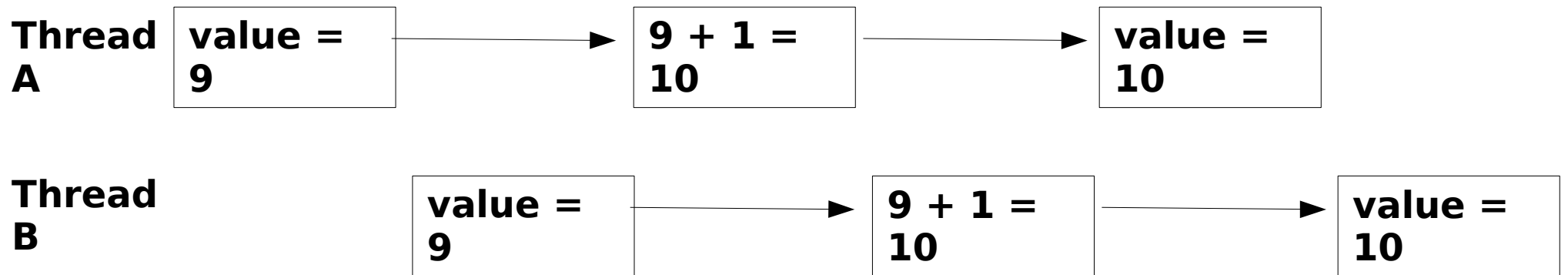
BACK UP slides



Two consecutive `inc()` return same value

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

get value
do value + 1
set value



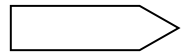
Transformations: Update Expressions

`value++`

`++value`

`value = value + 2;`

`value = value - 2;`



`value.getAndIncrement()`

`value.incrementAndGet()`

`value.addAndGet(2)`

`value.addAndGet(-2)`

Code Patterns matching `putIfAbsent()`

(i)

```
if (!map.containsKey(key)) {  
    map.put(key, value);  
}
```

(ii)

```
boolean keyExists = map.containsKey(key);  
if (!keyExists) {  
    map.put(key, value);  
}
```

(iii)

```
if (map.get(key) == null) {  
    map.put(key, value);  
}
```

(iv)

```
Object result = map.get(key);  
if (result == null) {  
    map.put(key, value);  
}
```

`map.putIfAbsent(key, value);`

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 dequeues 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) 46  
    }  
}
```

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**