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

UPCRC Illinois
Universal Parallel Computing
Research Center

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

Updating Shared Data Must Execute Atomically

```
public class Counter {
    int value = 0;

    public int getCounter() {
        return value;
    }

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

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

```
read value
compute value + 1
store value
```

UPCRC Illinois

Universal Parallel Computing Research Center

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;
    }
}
```

Locking is Error-Prone

```
  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

UPCRC Illinois
Universal Parallel Computing
Research Center

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

UPCRC Illinois
Universal Parallel Computing Research Center

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

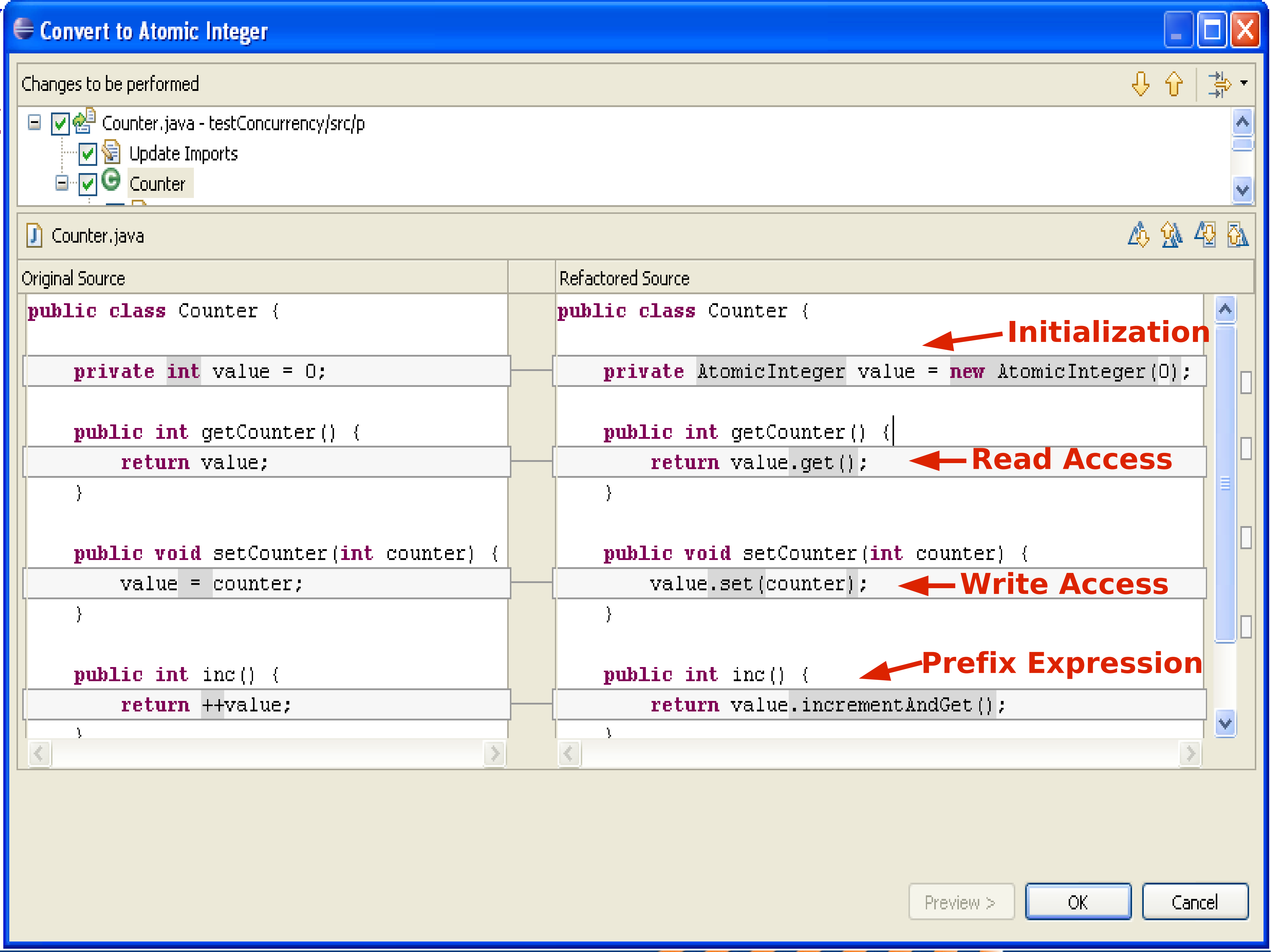
`AtomicInteger` in java.util.concurrent

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



Original Source

```
public class Counter {

    private int value = 0;

    public int getCounter() {
        return value;
    }

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

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

Refactored Source

```
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();
    }
```

Initialization

Read Access

Write Access

Prefix Expression

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();
  }
}
```

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

```
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

```
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)

`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`

UPCRC Illinois
Universal Parallel Computing Research Center

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

```
solve (Problem problem) {
  if (problem.size <= BASE_CASE )
      solve problem directly
  else {
      split problem into tasks

      solve each task

      compose result from subresults
  }
}
```

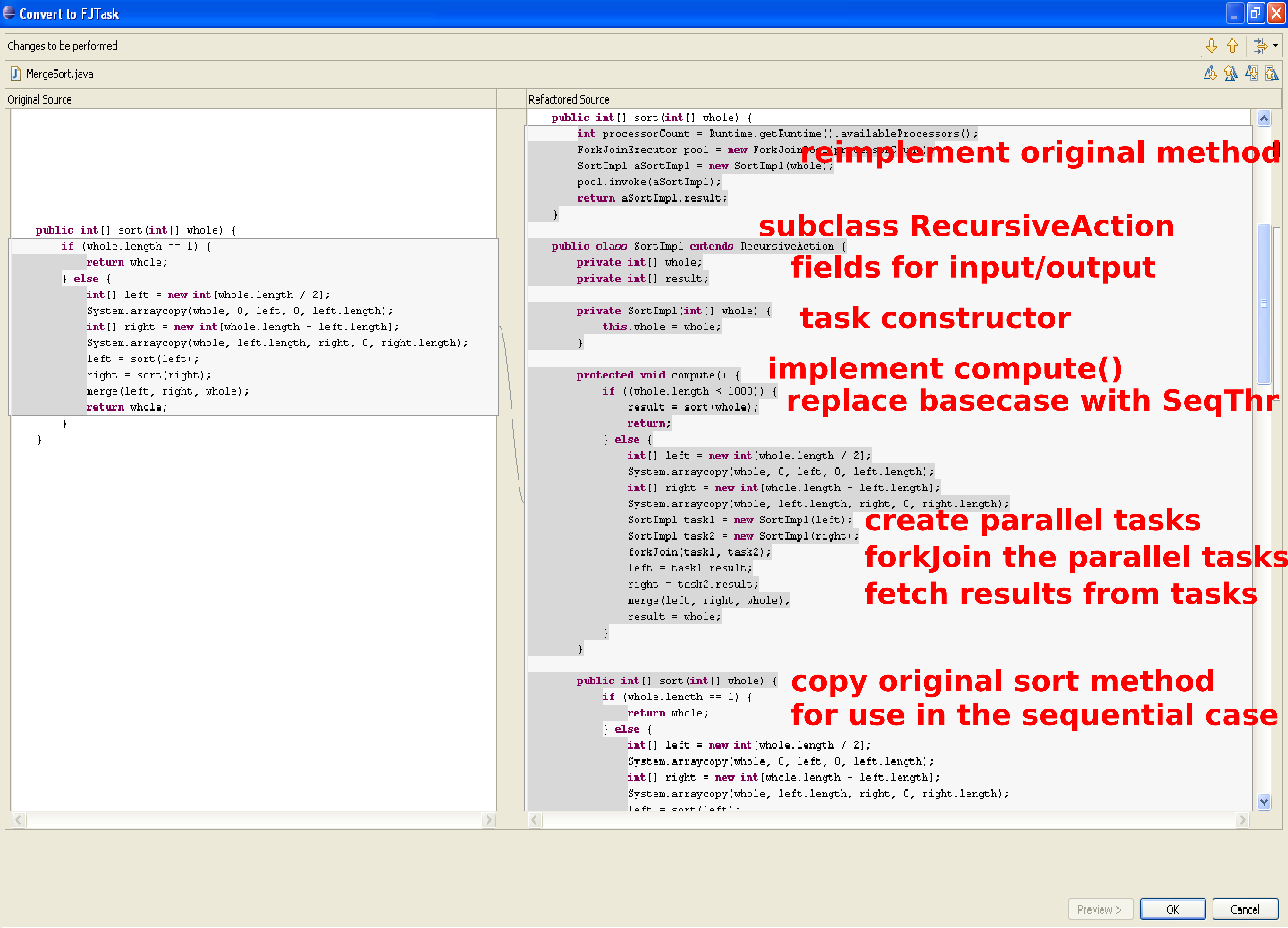
```
solve (Problem problem) {
  if (problem.size <= SEQ_THRESHOLD )
      solve problem sequentially
  else {
      split problem into tasks
      In Parallel (fork){
          solve each task
      } wait for all tasks (join)
      compose result from subresults
  }
}
```

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]



Original Source

```java
    public int[] sort(int[] whole) {
        if (whole.length == 1) {
            return whole;
        } else {
            int[] left = new int[whole.length / 2];
            System.arraycopy(whole, 0, left, 0, left.length);
            int[] right = new int[whole.length - left.length];
            System.arraycopy(whole, left.length, right, 0, right.length);
            left = sort(left);
            right = sort(right);
            merge(left, right, whole);
            return whole;
        }
    }
```

Refactored Source

```java
    public int[] sort(int[] whole) {
        int processorCount = Runtime.getRuntime().availableProcessors();
        ForkJoinExecutor pool = new ForkJoinPool(processorCount);
        SortImpl aSortImpl = new SortImpl(whole);
        pool.invoke(aSortImpl);
        return aSortImpl.result;
    }

    public class SortImpl extends RecursiveAction {
        private int[] whole;
        private int[] result;

        private SortImpl(int[] whole) {
            this.whole = whole;
        }

        protected void compute() {
            if ((whole.length < 1000)) {
                result = sort(whole);
                return;
            } else {
                int[] left = new int[whole.length / 2];
                System.arraycopy(whole, 0, left, 0, left.length);
                int[] right = new int[whole.length - left.length];
                System.arraycopy(whole, left.length, right, 0, right.length);
                SortImpl task1 = new SortImpl(left);
                SortImpl task2 = new SortImpl(right);
                forkJoin(task1, task2);
                left = task1.result;
                right = task2.result;
                merge(left, right, whole);
                result = whole;
            }
        }

        public int[] sort(int[] whole) {
            if (whole.length == 1) {
                return whole;
            } else {
                int[] left = new int[whole.length / 2];
                System.arraycopy(whole, 0, left, 0, left.length);
                int[] right = new int[whole.length - left.length];
                System.arraycopy(whole, left.length, right, 0, right.length);
                left = sort(left);
```

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

UPCRC Illinois
Universal Parallel Computing Research Center

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 Concurrencer to refactor the same fields as the developers did
- evaluates usefulness and correctness

Case-study 2:

- used Concurrencer to refactor 6 divide-and-conquer algorithms
- evaluates usefulness, correctness and speed-up

UPCRC Illinois
Universal Parallel Computing Research Center

Q1: Is Concurrencer Useful?

refactoring	project	# of refactorings	LOC changed	LOC Concurrencer can handle
Convert int field to `AtomicInteger`	MINA, Tomcat, Struts, GlassFish, JaxLib, Zimbra	64	401	100.00%
Convert `HashMap` field to `ConcurrentHashMap`	MINA, Tomcat, Struts, GlassFish, JaxLib, Zimbra	77	618	91.70%
Convert recursion to `FJTask`	mergeSort, fibonacci, maxSumConsecutive, matrixMultiply, quickSort, maxTreeDepth	6	302	100.00%

Q2: Is the Refactored Code Correct?

1. Thread-safety: omission of atomic methods

`putIfAbsent(key, value)`			`remove(key, value)`		
potential usages	human omissions	Concurrencer omissions	potential usages	human omissions	Concurrencer omissions
73	33	10	10	8	0

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

Concurrencer used the correct method

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

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

UPCRC Illinois
Universal Parallel Computing Research Center

BACK UP slides

Convert int to AtomicInteger - transformations -

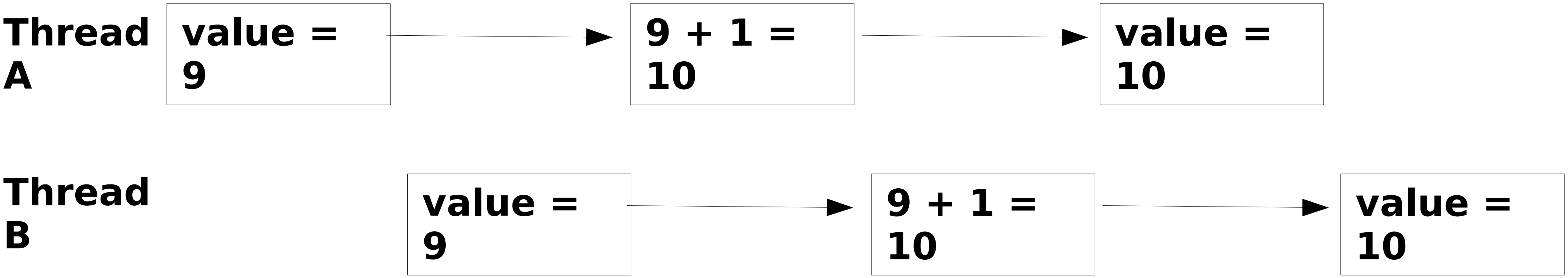
Access	**int**	**AtomicInteger**
Read	`f`	`f.get()`
Write	`f = e`	`f.set(e)`
Cond. Write	`if (f==e) f=`e_1	`f.compareAndSet(e,`e_1`)`
Prefix Inc.	`++f`	`f.incrementAndGet()`
Postfix Inc.	`f++`	`f.getAndIncrement()`
Infix Add	`f = f + e`	`f.addAndGet(e)`
Add	`f += e`	`f.addAndGet(e)`
Prefix Dec.	`--f`	`f.decrementAndGet()`
Postfix Dec.	`f--`	`f.getAndDecrement()`
Infix Sub.	`f = f - e`	`f.addAndGet(-e)`
Subtract	`f -= e`	`f.addAndGet(-e)`

UPCRC Illinois
Universal Parallel Computing
Research Center

Two consecutive `inc()` return same value

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

```
get value
do value + 1
set value
```



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

UPCRC Illinois
Universal Parallel Computing Research Center

Basic Patterns that Concurrencer Replaces with `map.putIfAbsent(key, value)`

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

uPCRC Illinois
Universal Parallel Computing
Research Center

`putIfAbsent` Pattern not Currently Handled

```
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

```
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);
}
```

```
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);
}

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

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;
}
```

Code Patterns: `remove()` and `replace()`

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

```
...
```

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

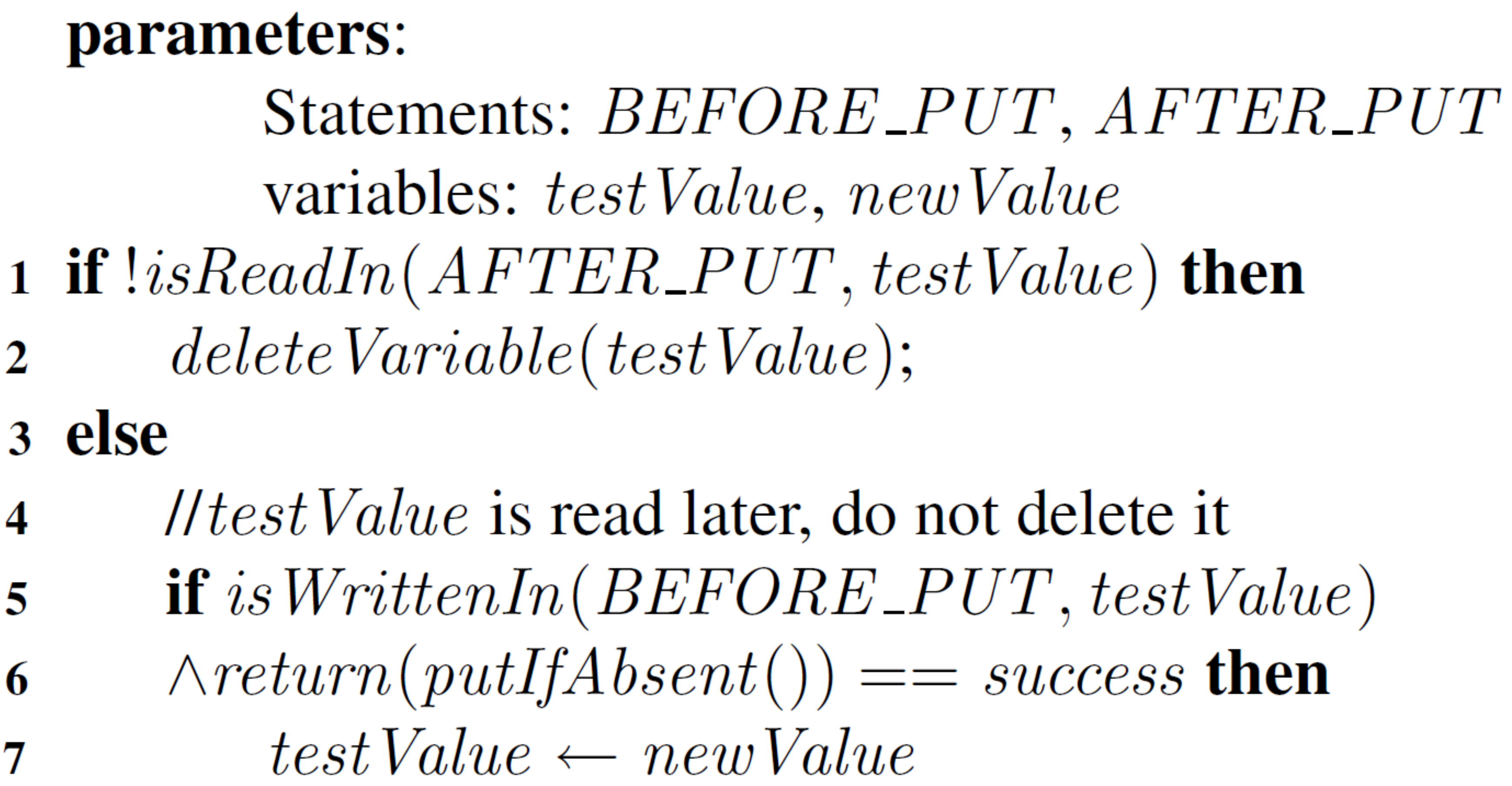
```
...
```

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

```
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: $testValue$, $newValue$
if !$isReadIn(AFTER\_PUT, testValue)$ then
    $deleteVariable(testValue)$;
else
    //$testValue$ is read later, do not delete it
    if $isWrittenIn(BEFORE\_PUT, testValue)$
    $\land return(putIfAbsent()) == success$ then
        $testValue \leftarrow newValue$
```

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);
      }
   }
```

UPCRC Illinois
Universal Parallel Computing Research Center

Transformations for ExtractFJTask

Subclass `FJTask`
- fields for args, result
- constructor

```
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);
       }
   }
```

UPCRC Illinois
Universal Parallel Computing Research Center

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

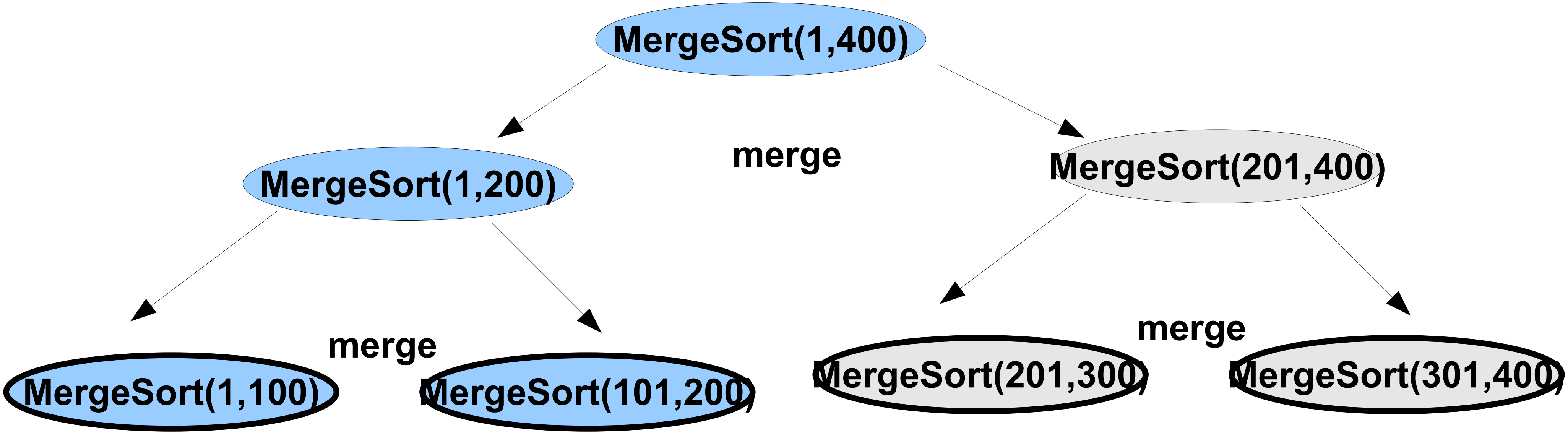
```
public int[] sort(int[] toSort){
  ForkJoinExecutor pool =
         new ForkJoinPool(Runtime.getRuntime().availableProcessors());

  MergeSort sortObj = new MergeSort(toSort);

  pool.invoke(sortObj);

  return sortObj.result;
}
```

Computation Tree for MergeSort



MergeSort(1,400)

merge

MergeSort(1,200)

MergeSort(201,400)

merge

MergeSort(1,100)

MergeSort(101,200)

merge

MergeSort(201,300)

MergeSort(301,400)

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