ReLooper: Refactoring for Loop Parallelism in Java

Danny Dig
University of Illinois
dig@cs.uiuc.edu

Mihai Tarce
Politehnica University of Timisoara
mihai.tarce@cs.upt.ro

Cosmin Radoi
Politehnica University of Timisoara
cosmin.radoi@cs.upt.ro

Marius Minea
Politehnica University of Timisoara
marius@cs.upt.ro

Ralph Johnson
University of Illinois
johnson@cs.uiuc.edu

Abstract

In the multicore era, sequential programs need to be refactored for parallelism. The next version of Java provides `ParallelArray`, an array datastructure that supports parallel operations over the array elements. For example, one can `apply` a procedure to each element, or `reduce` all elements to a new element in parallel. Refactoring an array to a `ParallelArray` requires (i) analyzing whether the loop iterations are safe for parallel execution, and (ii) replacing loops with the equivalent parallel operations. When done manually, these tasks are non-trivial and time-consuming. This demo presents ReLooper, an Eclipse-based refactoring tool, that performs these tasks automatically. Preliminary experience with refactoring real programs shows that ReLooper is useful.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Parallel Programming; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Program Editors

General Terms Algorithms, Design

Keywords Refactoring, program analysis, program transformation, parallelism and concurrency

1. Introduction

In the multicore era, unless programmers refactor the existing sequential programs for parallelism, they will not benefit from the underlying parallel processors. Refactoring for parallelism is non-trivial, because the refactored code needs to satisfy two conflicting goals: it needs to be *thread-safe* (i.e., run correctly when executed under multiple threads) and *scalable* (i.e., performance continues to improve when adding more cores).

The key to scaling performance is to use fine-grained parallelism. Java will include the `ParallelArray` framework [1], a special kind of array that provides fine-grained parallel operations. For example, one can `apply` a procedure to the elements of an array, `map` elements to new elements, or `reduce` all elements into a single value like a sum. The framework efficiently executes these parallel operations by splitting the computations on array elements among a pool of worker threads, and relying on a runtime library to balance the work among the processors in the system.

To refactor an existing array into a `ParallelArray`, the programmer constructs it by using factory methods (e.g., by copying elements from other arrays). Then the programmer identifies the loops that iterate over all the array elements and she analyzes each loop to infer its intent (e.g., the loop reduces all elements to a value). Next, she replaces the loop body with a call to the equivalent *parallel operation* (e.g., `reduce`). The parallel operation takes an *element operator* as an argument and executes it on each element. Since Java does not support anonymous functions (i.e., lambda expressions), the programmer needs to encapsulate the operator inside an anonymous class, by subclassing one of the 132 operator classes, and override the `op` method.

In addition, since `ParallelArray` assumes that all parallel computations do not interfere with each other, it runs them without any synchronization. It is the programmer's responsibility to verify that indeed the loop iterations do not have conflicting memory accesses. This analysis and code rewriting is non-trivial, and time-consuming.

We have implemented a refactoring tool, ReLooper, that automates the safety analysis and the rewriting of code. ReLooper is integrated with Eclipse's refactoring engine, so it offers all the convenient features of a refactoring engine: previewing the changes, preserving the formatting, undoing changes, etc. To use ReLooper, the programmer selects an array and chooses ConvertToParallelArray from the refactoring menu.

2. The Refactoring Tool

Figure 1 shows a preview of the changes that ReLooper applies to a small program that works with an array of `Complex` numbers. A complex number has the form `a + bi` where `a` is the real part, and `b` is the imaginary part. The first loop in method `ComplexTest.test()` initializes the array elements using the factory method `createRandom()`. The second loop iterates over all the array elements and computes the square of each complex number. The third loop adds all the numbers and stores the result in the `sum` variable.

Transformations. ReLooper changes the *type declaration* of `numbers` into a `ParallelArray` of `Complex` objects. Then it replaces the code that allocates storage for the array with code that creates a `ParallelArray` with the same capacity, and specifies the base element type and the pool of worker threads that will be used at runtime (`defaultExecutor()` arranges to use most of the processors available).

Copyright is held by the author/owner(s).
OOPSLA '09 October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-768-4/09/10

RIGHTSLINK

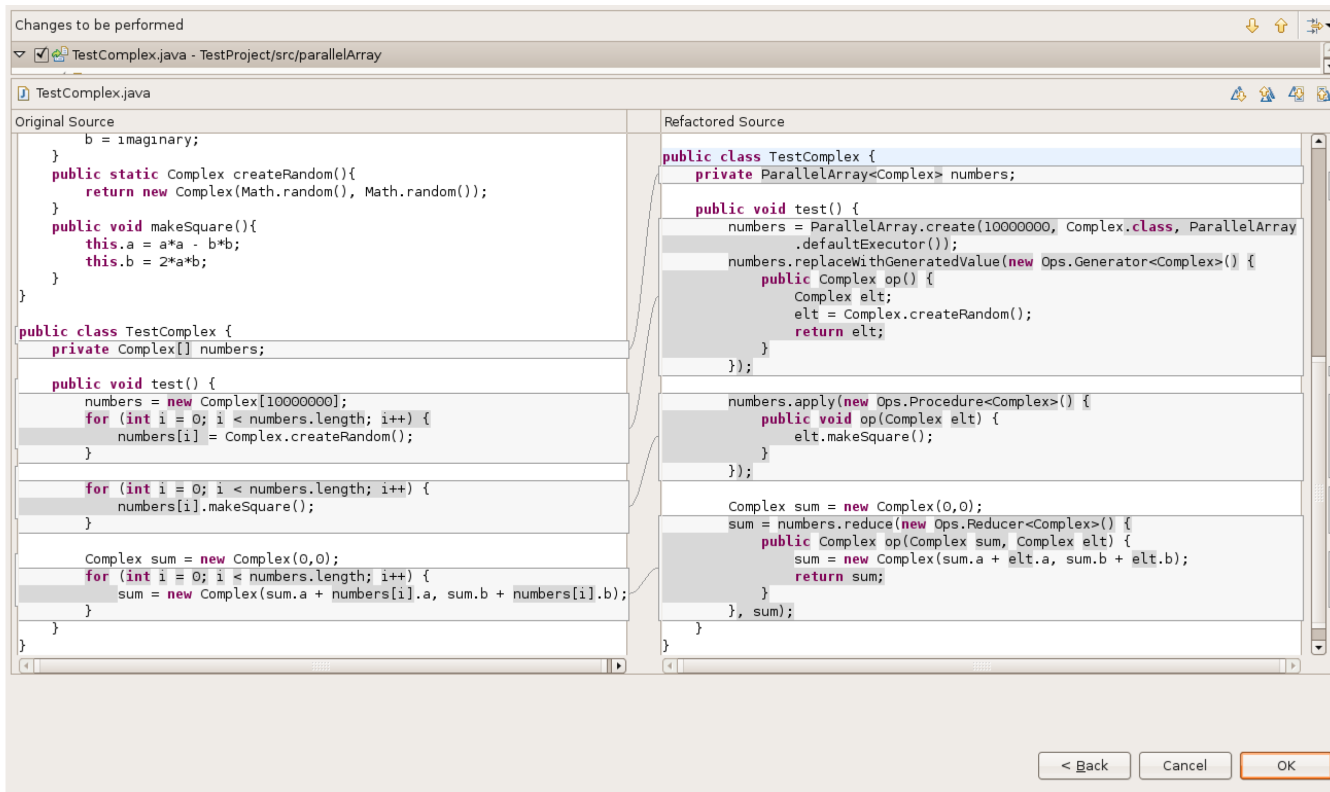

Figure 1. Using ReLooper to convert an array of `Complex` numbers to a `ParallelArray`. The screenshot shows a preview of the changes, with the original code on the left and the refactored code on the right.

For each loop that iterates over the array elements, ReLooper infers the intent of the loop and replaces it with the equivalent parallel operation from `ParallelArray`. In our example, the first loop initializes the array elements, so ReLooper replaces it by invoking the `replaceWithGeneratedValue` operation and passes an operator implemented as an anonymous `Generator` class. ReLooper overrides the `op()` method to create objects like in the original code. ReLooper correctly replaces the last two loops with the appropriate operations and generates the anonymous classes that encapsulate the operators.

Preconditions. ReLooper performs the following program analyses to determine whether the refactoring can be applied safely. First, it checks that a loop iterates over *all* elements of the array, i.e., from the first element to the last, without skipping elements.

Second, the analysis determines that there are no loop-carried dependencies between iterations, i.e., each iteration processes only one element, and the variables in one iteration do not depend on values coming from other iterations. Even though the `sum` variable is a loop-carried dependency, the analysis allows it because this dependency is eliminated when `sum` becomes the *accumulator* variable for the `reduce` operation (internally, the reduction creates fresh `sum` variables, and accumulates them in a final step).

Third, the analysis determines whether the loop iterations have conflicting memory accesses. As one step of this check, our analysis determines that the array elements are *unique*, i.e., the array does not contain duplicate objects. Processing duplicate objects in parallel could introduce data races. To check the uniqueness invariant, the analysis builds upon a context-sensitive, flow-insensitive, demand-driven pointer analysis [2] implemented in WALA [3]. Our custom analysis determines that elements created in different iterations are indeed unique. For example, it determines that different calls to `createRandom()` return unique objects, and that subsequent loops maintain the uniqueness invariant.

3. Conclusions

Refactoring tools can help programmers retrofit parallelism into sequential code. This demo presents ReLooper, a tool for parallelizing loops over arrays. Our preliminary experience with refactoring two large NLP applications and other medium-size applications shows that ReLooper is useful: on average, the refactoring finishes in less than 30 seconds, and the results are correct. As we are currently adding more analysis to check that updates to shared state among loop iterations are not conflicting, the main challenge remains to keep the refactoring both precise and fast enough to be used in an interactive mode.

ReLooper can be downloaded from its homepage:
`http://refactoring.info/tools/ReLooper`

4. Acknowledgments

This work is partially funded by Intel and Microsoft through the UPCRC Illinois, and a DOE grant ER25752. Cosmin and Mihai did a part of this work as undergraduate summer interns at the Information Trust Institute at the University of Illinois.

References

[1] D. Lea. ParallelArray package extra166y. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`, 2009.

[2] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proceedings of OOPSLA*, 2005.

[3] WALA: T. J. Watson Libraries for Analysis. `http://wala.sf.net`.

RIGHTSLINK