A Refactoring Approach to Parallelism

Danny Dig, University of Illinois at Urbana-Champaign

// In the multicore era, a major task for programmers will be application parallelization. This article outlines how refactoring tools can help with this task and describes a toolset for improving program performance, safety, and scalability. //

FOR DECADES, PROGRAMMERS HAVE RELIED on Moore's law to improve application performance. But with the advent of multicore chips, they must exploit parallel processing to realize the same improvement or to enable new, previously unavailable applications and services.

The most common way to parallelize a program is to do it incrementally, one piece at a time. Each small step is a refactoring, a revision to the code that preserves the program's behavior. Programmers prefer this approach because it maintains a working, deployable version of the program throughout the process. However, the refactoring approach is tedious because it means changing many lines of code, is error-prone, and requires programmers to ensure that parallel operations don't interfere with each other. For example, parallelizing several loops using Java's Parallel Array data structure can entail an average of 10 changes per loop, plus additional time ensuring that the parallel iterations didn't update shared objects or files, which could lead to data races.

In the next decade, refactoring tools will transform the process of retrofitting parallelism, just as they transformed sequential programming in the past decade. But unlike sequential refactoring, refactoring for parallelism is likely to make the code more complex, more expensive to maintain, and less portable. I present my vision on how refactoring tools, along with smart integrated development environments (IDEs) and performance tools, can further improve programmer productivity (by improving parallel code's readability and maintenance), program performance, and program portability. I also describe the current incarnation of this vision in the form of a refactoring toolset developed by my research group.

A Vision for Interactive Refactoring Tools

Researchers have proposed several tools for reducing the programmer's burden when converting existing sequential programs to parallel programs. They come in two distinct flavors: fully automatic tools (for example, automatic parallelizing compilers[1–4]) and interactive tools (such as refactoring tools[5–12]). The fundamental difference between these tools is the programmer's role. (For more on previous work relating to parallelism, see the "Other Refactoring Tools for Parallelism" sidebar.)

Automatic versus Interactive

When an automatic tool works, it gives great results. Unfortunately, without access to a programmer's domain knowledge, such a compiler has limited applicability. To date, commercial compilers have been successful at parallelizing small, straightforward kernel loops but not at introducing meaningful parallelism in large, irregular, nonscientific applications. Even though compilers have improved, programmers still parallelize most of the code by hand.

Interactive tools, by contrast, take a completely different approach, putting the programmer in the driver's seat. As

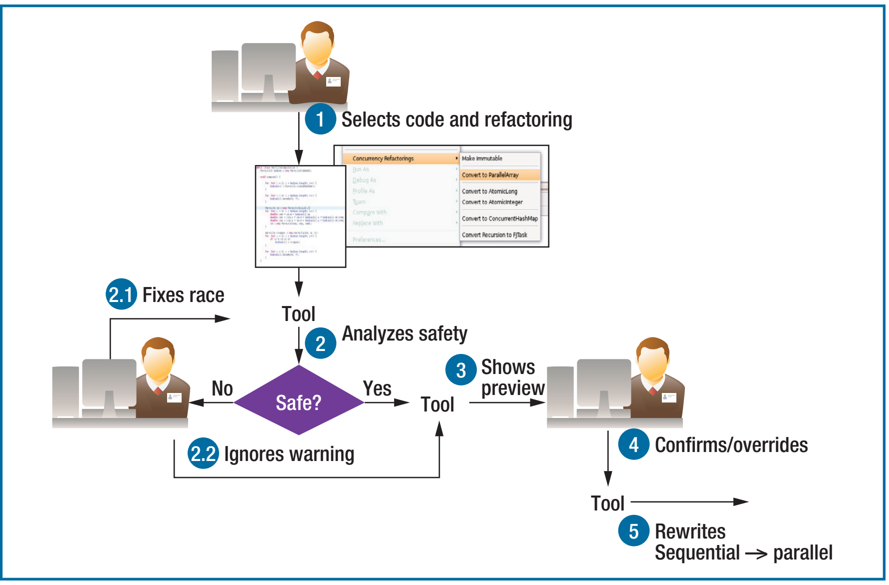


FIGURE 1. An interactive refactoring tool. There are three points of interaction: the programmer selects a code snapshot and targets it for parallelism with a transformation, the programmer then validates whether the safety warnings raised by the tool are genuine, and finally, the programmer confirms the edits that the tool applies.

the expert in the problem domain, the programmer understands the concepts amenable to parallelism as well as the existing program invariants that must be preserved, the data and control flow relationships between parts of the program, and the current algorithms and data structures.

Thus, the interactive approach combines the strengths of both the programmer (domain knowledge, seeing the big picture) and the computerized tool (fast search, remember, and compute). The programmer does the creative part: selecting code and targeting it with a transformation. The tool does the tedious job of checking thread safety (by traversing through many functions and aliased variables) and modifying the program. When the tool can't apply a transformation, it provides feedback within the visual interface of an IDE such as Eclipse or VisualStudio, thus allowing the programmer to pinpoint the problematic code.

In the past decade, interactive refactoring tools have revolutionized how programmers approach sequential programming. Without these tools, they often overdesigned their software because it was expensive to change the design once implemented. Refactoring tools let programmers continuously explore the design space of large code bases while preserving existing behavior. Modern IDEs incorporate refactoring in their top menu and often compete on the basis of refactoring support.

When it comes to parallel programming, interactive tools likewise enable programmers to safely and efficiently explore the space of performance optimizations and parallel constructs while preserving existing functionality. Tests of our current refactoring toolset for improving thread safety, throughput, and scalability support this prediction. Our toolset reduces the burden of analyzing and modifying code, is fast enough to be used interactively, and correctly applies transformations that open source developers often overlook.

A refactoring toolset for parallelism has several points of interaction with the programmer, who is ultimately responsible for identifying all shared data or compute-intensive code and targeting it with the appropriate refactoring. As illustrated in Figure 1, the programmer first selects some code and a target refactoring. The tool then analyzes the transformation's safety. By default, the refactoring tool applies the changes only when its analysis determines that it's safe to do so. If some of the preconditions aren't met, the tool raises warnings and highlights the problematic code. The programmer can decide to cancel the refactoring, fix the code and rerun the refactoring, or proceed despite the warnings.

Our growing toolset[7–9] of refactorings for parallelism lets programmers explore the parallelism space along three axes: thread safety, throughput, and scalability. The experience with replicating refactoring scenarios performed by open source developers shows that automation is useful. It also shows that we need to go further.

In the past, refactoring was traditionally associated with improving the code's structure, thus making the code more readable and reusable, even across different platforms. With refactorings for parallelism, the new code is likely to be less readable. Note how the parallel code in Figure 2 (on the right-hand side) hides the original code's intent, thus increasing its complexity and decreasing the productivity of the programmers who need to maintain it. The new code is also less portable because it's fine-tuned for a particular platform.

I envision smart IDEs that treat refactorings for performance intelligently, in ways that improve both the readability and portability of the parallel code. Furthermore, refactoring tools could also suggest transformations that achieve maximum program performance. For this, they will need to work in tandem with other tools (such as compilers and performance profilers).

Improving Programmer Productivity

When refactoring sequential programs, programmers usually throw away the old code and keep the new. But when refactoring for performance, they want to keep both and be able to navigate back and forth between the two forms.

A smart IDE that treats refactorings as first-class program transformations can automatically record them as the programmer applies them. Subsequently, these transformations can serve as explicit documentation about how a piece of code evolved, making the program easier to understand. Advanced refactoring engines like Eclipse already provide this recording capability.

The IDE can also provide two views of the same code: sequential and parallel. The programmer would use the sequential view to understand the program, fix bugs, or add new features, and use the parallel view for performance debugging. Annotations to the code in the sequential view would indicate that a programmer has applied a performance refactoring. For the example in Figure 2, the refactoring could leave an @Parallel annotation in front of each loop. By asking the IDE to expand this annotation, the programmer could view the parallel code.

OTHER REFACTORING TOOLS FOR PARALLELISM

The earliest work on interactive tools for parallelization grew out of the Fortran community and targeted loop parallelization. Interactive tools like ParaScope[1] and SUIF Explorer[2] relied on the user to specify which loops to interchange, align, replicate, or expand. The tool computed and displayed to the programmer information such as dependence graphs, but this work addressed numerical computation on scalar arrays and didn't deal with the sharing through the heap prevalent in object-oriented programs.

Reentrancer[3] is a recent refactoring tool developed at IBM for making code reentrant. Reentrancer changes global data stored in static fields into thread-local data. The refactoring for reentrancy can be seen as an enabling refactoring—that is, it makes accesses to global data thread-safe. We have manually performed this refactoring several times when eliminating writes to the global shared objects discovered by our tool.[4]

Robert Fuhrer[5] proposes five concurrency refactorings for the X10 programming language for server computing on networked nodes with distributed memory. X10 introduces several high-level parallel constructs (such as asynchronous tasks and clocks). The proposed set of refactorings converts sequential code to make use of these parallel constructs.

The Photran[6] project also plans to support several concurrency refactorings for high-performance computing in Fortran.

References

1. K. Kennedy, K.S. McKinley, and C.W. Tseng, "Interactive Parallel Programming Using the Parascope Editor," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 3, 1991, pp. 329–341.
2. S.-W. Liao et al., "Suif Explorer: An Interactive and Interprocedural Parallelizer," *Proc. 7th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, ACM Press, 1999, pp. 37–48.
3. J. Wloka, M. Sridharan, and F. Tip, "Refactoring for Reentrancy," *Proc. 7th Joint Meeting European Softare Eng. Conf. and the Int'l Symp. Foundations of Software Eng.* (ESEC/FSE), ACM Press, 2009, pp. 173–182.
4. D. Dig et al., *ReLooper: Refactoring for Loop Parallelism*, tech. report, Dept. Computer Science, Univ. Illinois at Urbana-Champaign, Sept. 2009; http://hdl.handle.net/2142/14536.
5. R. Fuhrer and V. Saraswat, "Concurrency Refactoring for x10," *Proc. 3rd ACM Workshop Refactoring Tools*, ACM Press, 2009.
6. M. Méndez et al., "A Catalog and Classification of Fortran Refactorings," *Proc. 11th Symp. Software Eng.* (ASSE 2010), 2010; www.fortranrefactoring.com.ar/papers/39jaiio-asse-20.pdf.

Improving Code Portability

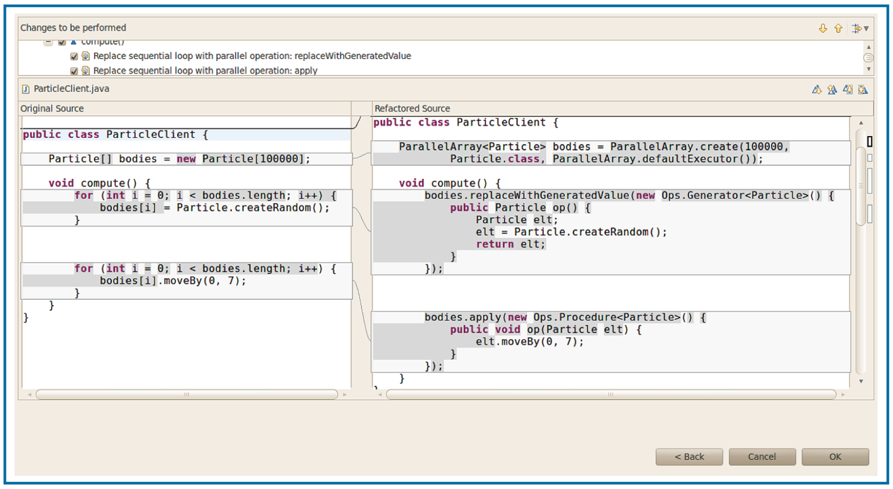
When programmers need to squeeze the last bit of performance out of their software, they often resort to platform-specific transformations that take into account hardware characteristics, such as the number of cores, available memory (its size and whether it's shared or distributed), cache line sizes, and so on.

With current methods, such platform-specific transformations are embedded deep within the code, making the code less portable. To migrate to a new platform, the programmer needs to first undo the platform-specific transformations, get the platform-independent code, and then apply new transformations.

Smart IDEs that understand explicit parallel transformations can make parallel code more portable because the same transformation can have several platform-specific implementations. For example, a programmer might refactor a loop for parallelism, with different transformations for running the code on a gaming console, a GPU, a general-purpose shared-memory computer, or a distributed system. In such a case, the refactoring tool would provide several alternative implementations of the same transformation. The programmer would maintain the portable code, which is annotated with transformations, and switch to the platform-specific view when needed.

Improving Performance

When deciding what to parallelize, programmers use their domain knowledge along with other tools to identify performance bottlenecks. Currently, there's a gap between performance tools and refactoring tools: after determining what to parallelize, a programmer still doesn't know which of the several potential refactorings would yield the best performance. A more focused interaction between refactoring tools and the other tools in the toolbox could fill up this gap. For example, refactoring tools could take feedback from performance tools such as hardware monitors or profilers. After running a program



Changes to be performed

compute()

☑ Replace sequential loop with parallel operation: replaceWithGeneratedValue

☑ Replace sequential loop with parallel operation: apply

ParticleClient.java

Original Source

```
public class ParticleClient {

    Particle[] bodies = new Particle[100000];

    void compute() {
        for (int i = 0; i < bodies.length; i++) {
            bodies[i] = Particle.createRandom();
        }


        for (int i = 0; i < bodies.length; i++) {
            bodies[i].moveBy(0, 7);
        }
    }
}
```

Refactored Source

```
public class ParticleClient {

    ParallelArray<Particle> bodies = ParallelArray.create(100000,
            Particle.class, ParallelArray.defaultExecutor());

    void compute() {
        bodies.replaceWithGeneratedValue(new Ops.Generator<Particle>() {
            public Particle op() {
                Particle elt;
                elt = Particle.createRandom();
                return elt;
            }
        });


        bodies.apply(new Ops.Procedure<Particle>() {
            public void op(Particle elt) {
                elt.moveBy(0, 7);
            }
        });
    }
```

FIGURE 2. The ParallelArray library. The preview shows the sequential code on the left-hand side; the right-hand side shows all the changes that need to be applied.

and detecting performance "smells" that indicate bottlenecks, a smart IDE could suggest several refactorings. The programmer in the loop could make informed decisions about which refactorings to apply. The runtime information would also complement the imprecise static analysis of refactoring tools.

Refactoring tools could also provide explicit knobs for other tools. For example, parallelizing a sequential divide-and-conquer algorithm requires the programmer to specify the cut-off threshold between the sequential and the parallel case. The programmer could provide an initial starting point, and the refactoring tool could hook into an autotuner to find the value that maximizes the performance. Even more radically, an autotuner could mix and match several refactorings and select the combination that yields the best performance.

Refactoring tools and compilers ought to complement rather than compete with each other. In cases when the compiler can't automatically parallelize a program, it could provide information that the refactoring tool and programmer can use to get the job done.

Our Refactoring Toolset for Parallelism

To turn this vision into reality, we first asked the question, "What parallelizing program transformations occur most often in practice?" To answer it, we conducted a quantitative and qualitative study[13] of five open source programs whose developers had parallelized them manually (two Eclipse plugins, JUnit, Apache Tomcat server, and Apache MINA library).

We found that parallelizing transformations aren't random but, rather, fall into four categories:

- those that improve thread safety (that is, the application behaves according to its specification even when executed under multiple threads);
- those that improve latency (that is, an application feels more responsive);
- those that improve throughput (that is, the program executes more computational tasks per unit of time); and
- those that improve scalability (that is, performance increases with the addition of more cores).

The industry trend is to attack the problem of introducing parallelism by using a parallel library or framework. For example, Microsoft provides the Task Parallel Library (TPL) for .NET, Intel provides Threading Building Blocks (TBB) for C++, and Java contains ForkJoinTask and ParallelArray. All these libraries have comparable features, and much of the complexity of writing parallel code (for example, balancing the computation load among the cores) is hidden within them. Libraries also provide highly scalable thread-safe collections (such as ConcurrentHashMap), plus lightweight tasks—thread-like entities but with much lower overhead for creation and management.

Our current refactoring toolset uses Java libraries and is implemented on top of Eclipse's refactoring engine. Thus, it offers all the practical features that programmers love: integration in an IDE, change previews, and undo. It currently automates six refactorings that fall into three categories: thread safety, throughput, and scalability. These refactorings often require transformations that span multiple nonadjacent program statements and require analyzing the program's control and data flow. Also, the refactoring tools must be able to analyze and detect shared objects in object-oriented programs that contain a web of heap-allocated objects interconnected to other objects through their fields.

Several researchers[13–15] recommend approaching the process by first making the code right (that is, thread-safe), then making it fast (that is, multi-threaded), and then making it scalable. I discuss our refactoring toolset in that order.

Refactorings for Thread Safety

To prepare or enable the program for parallel execution, the programmer must find the mutable data that will be shared. He or she can decide to synchronize accesses to such data, make it immutable, or eliminate the sharing. Our toolset supports two refactorings for synchronizing accesses: one[7] converts an int field to an AtomicInteger, a java.util.concurrent (j.u.c.) library class that provides atomic operations for field updates. The second converts a HashMap field into a ConcurrentHashMap, a thread-safe implementation for working with hashmaps.

An alternate way to make a whole

class thread-safe is to make it immutable. An immutable class, once properly constructed, is thread-safe by default and so can be shared among several threads with no need for synchronization.

Our refactoring toolset enables the programmer to convert a mutable class into an immutable one by making the class and all its fields final, so that they can't be assigned outside constructors and field initializers. The tool finds all mutator methods in the class—that is, methods that directly or indirectly mutate the internal state (as given by its fields)—and converts them into factory methods that return a new object whose state is the old state plus the mutation. Java programmers have seen such methods in immutable classes such as `String`, where the operations `replace(oldChar, newChar)` or `toUpperCase()` return a new `String` with some characters replaced.

Next, the tool finds the objects that either enter from outside (for example, as method parameters) and become part of the state or are already part of the state and escape (for example, through return statements). It clones these objects so that a client class holding a reference to these state objects can't mutate them. Finally, the tool updates the client code to use the class in an immutable fashion. For example, when the client invokes a factory method, the tool reassigns the reference to the immutable class to the object returned by the factory method (for example, a reference to `myString` is assigned to `myString.toUpperCase()`).

To evaluate the usefulness of automation, we ran this refactoring tool on 346 classes from known open source projects.[8] We also studied how open source developers refactor manually. The results showed that the refactoring is widely applicable and that several of the manual refactorings aren't correct (the code contains subtle mutations and non-cloned entering or escaping objects), whereas our tool is safer. Furthermore, refactoring with our tool is fast (2.3 seconds per class) and saves the programmer from having to analyze 84 methods and change on average 42 lines per refactored class.

Refactorings for Throughput

Once a program is thread-safe, multithreading can improve its performance. The programmer could manage a raw thread manually (that is, create, spawn, and wait for results) or use a lightweight task managed automatically by a framework (a programmer-friendlier construct). Our toolset supports two such refactorings. One[7] converts a sequential divide-and-conquer algorithm into an algorithm that solves the recursive subproblems in parallel using Java's `ForkJoinTask` framework.[14] Another parallelizes loops over arrays via ParallelArray,[14] a parallel library in Java. Using this library, the programmer can apply a procedure to each element or reduce all elements to a new element in parallel. The library balances the load among the cores it finds at runtime.

The refactoring changes the array's data type and supersedes loops over the array elements with the equivalent parallel operations from the ParallelArray library. In the example in Figure 2, the first loop replaces each element with another random element; thus, the tool supersedes the loop with the `replaceWithGeneratedValue` parallel operation. The second loop applies the `moveBy` function to each element; thus, the tool supersedes the loop with the `apply` parallel operation.

Each parallel operation takes as an argument an element operator (lambda function or a closure) and applies it on each element. Since Java doesn't support closures, the tool extracts the statements from the original loop and wraps them within the `op` method of an `Operator` class. The tool chooses the correct operator among a class hierarchy with 132 classes.

At the heart of the tool lies a dataflow analysis that identifies objects shared among loop iterations and detects writes to them. The analysis works with programs in both source code and byte code (for example, .jar-packaged libraries). When the analysis finds writes to a shared object, it presents the programmer with a "program slice" of code statements that refer to the object being shared and indicate the write access. These statements are hyperlinked to the original source code, which helps the developer find the problematic code.

For empirical evaluation, we used the tool to parallelize compute-intensive loops in seven real programs.[9] The results show that the analysis is fast (20 seconds/refactoring) and effective. It found several real races in the analyzed programs. Automation saves the programmer from analyzing 420 methods per refactoring. On average, the parallelized code was 2.75 times faster on a quad-core computer.

Refactorings for Scalability

You don't want to sacrifice thread safety and correctness in the name of performance, but a naive synchronization scheme can lead to serializing an application, thus drastically reducing its scalability. This usually happens

ABOUT THE AUTHOR

DANNY DIG is a research professor at the University of Illinois at Urbana-Champaign (UIUC), where he works on software evolution. His other research interests include automated refactoring, program analysis, and concurrency/parallelism. Dig has a PhD from UIUC , where his dissertation on automated software upgrades won the best PhD thesis award. Contact him via http://netfiles.uiuc.edu/dig/www.

when working with low-level synchronization constructs such as locks, the **goto** statements of parallel programming. Locks are tedious to work with and error-prone; too many slow down or deadlock a program, while too few lead to data races.

When possible, a better alternative is to use a highly scalable data structure provided by parallel libraries. Our toolset supports two such refactorings. One converts an **int** into an **AtomicInteger**, a lock-free data structure that uses compare-and-swap hardware instructions, and the other converts a **HashMap** field to a **ConcurrentHashMap**. If a class contains a **HashMap** field that is read/written in parallel, it must synchronize the accesses to the map. To accomplish this, the programmer can use a common lock or a synchronized wrapper over a **HashMap** (for example, **Collections.synchronizedMap(aMap)**). Both the synchronized and wrapped **HashMap** achieves its thread safety by protecting all accesses to the **map** with a common lock. This results in poor scalability, since multiple threads trying to access different parts of the map simultaneously wind up contending for the lock.

A better alternative is to refactor the **map** field into a **ConcurrentHashMap**, a thread-safe, highly scalable implementation for hash maps provided by the j.u.c. library. (All readers run in parallel, and a limited number of writers can run in parallel.) The refactoring replaces map updates with calls to **ConcurrentHashMap** APIs. For example, a common update operation is to first check whether a map contains a certain key, create the value object if it isn't present, and place the (key, value) in the map. The tool replaces such an updating pattern with a call to **putIfAbsent** of the **ConcurrentHashMap**, which executes the update atomically, without locking the entire map.

To evaluate our tool's usefulness, we used it to perform the same 77 refactorings that some open source developers performed manually.[7] The comparison shows that the manual refactorings were frequently incomplete. For example, in 33 out of 73 cases, the developers forgot to replace compound updates with the **putIfAbsent** API.

Building this refactoring toolset taught us several lessons. First, programmers often use parallel libraries, so refactoring tools need to support such libraries. Second, to keep the programmer engaged, refactoring tools need to finish their operations in less than 30 seconds, so they must use efficient, on-demand program analyses. Third, program analysis libraries and IDEs with excellent AST rewriting capabilities are essential for building refactoring tools. Fourth, once a program is parallel, it must stay maintainable, that is, remain readable and portable. Finally, refactoring tools must interact with other tools in the parallel toolbox.

Although the currently implemented refactorings are among the most commonly used in practice,[13] we need many more. We're constantly expanding the number of refactorings, inspired by the problems that industry practitioners face every day when they parallelize their programs. In addition, we plan to tackle the problems of readability, portability, and interactivity with other performance tools. Although our examples and refactorings use Java and Eclipse, they're representative for other object-oriented languages like C++ and C# that have similar shared-memory thread-based parallelism and libraries, and can also be accomplished in other environments.

Acknowledgments

This work is partially funded by Intel and Microsoft through the Universal Parallel Computing Research Center at the University of Illinois (UPCRC Illinois). I thank Paul Adamczyk, Nicholas Chen, Milos Gligoric, Ralph Johnson, Fredrik Kjolstad, Jeff Overbey, Cosmin Radoi, and the anonymous reviewers for providing valuable feedback on drafts of this article.

References

1. D.J. Kuck, "Automatic Program Restructuring for High-Speed Computation," *Proc. Conf. Analysing Problem Classes and Programming for Parallel Computing*, Springer, 1981, pp. 66–84.
2. F. Allen et al., "An Overview for the Ptran Analysis System for Multiprocessing," *J. Parallel and Distributed Computing*, vol. 5, no. 5, 1988, pp. 617–640.
3. R. Allen, D. Callahan, and K. Kennedy, "Automatic Decomposition of Scientific Programs for Parallel Execution," *Proc. 14th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*, ACM Press, 1987, pp. 63–76.
4. S.P. Amarasinghe et al., "An Overview of a Compiler for Scalable Parallel Machines," *Proc. 6th Int'l Workshop Languages and Compilers for Parallel Computing*, Springer, 1993, pp. 253–272.
5. K. Kennedy, K.S. McKinley, and C.W. Tseng, "Interactive Parallel Programming Using the Parascope Editor," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 3, 1991, pp. 329–341.
6. S.-W. Liao et al., "Suif Explorer: An Interactive and Interprocedural Parallelizer," *Proc. 7th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, ACM Press, 1999, pp. 37–48.
7. D. Dig, J. Marrero, and M.D. Ernst, "Refactoring Sequential Java Code for Concurrency via Concurrent Libraries," *Proc. 31st Int'l Conf. Software Eng.* (ICSE), IEEE Press, 2009, pp. 397–407.
8. F. Kjolstad et al., "Refactoring for Immutability," to appear in *Proc. 33rd Int'l Conf. Software Eng.* (ICSE), IEEE Press, 2011.
9. D. Dig et al., *ReLooper: Refactoring for Loop Parallelism*, tech. report, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, Sept. 2009; http://hdl.handle.net/2142/14536.
10. J. Wloka, M. Sridharan, and F. Tip, "Refactoring for Reentrancy," *Proc. 7th Joint Meeting European Soft. Eng Conf. and the Int'l. Symp. Foundations Software Eng.* (ESEC/FSE), ACM Press, 2009, pp. 173–182.
11. R. Fuhrer and V. Saraswat, "Concurrency Refactoring for x10," *Proc. 3rd ACM Workshop Refactoring Tools*, ACM Press, 2009.
12. M. Méndez et al., "A Catalog and Classification of Fortran Refactorings," *Proc. 11th Symp. Software Eng.* (ASSE 2010), 2010; www.fortranrefactoring.com.ar/papers/39jaiio-asse20.pdf.
13. D. Dig, J. Marrero, and M. D. Ernst, "How Do Programs Become more Concurrent? A Story of Program Transformations," tech. report, Computer Science and Artificial Intelligence Laboratory, MIT, Sept. 2008; http://hdl.handle.net/1721.1/42832.
14. D. Lea, *Concurrent Programming in Java*, Addison-Wesley, 2000.
15. B. Goetz et al., *Java Concurrency in Practice*, Addison-Wesley, 2006.