A Refactoring Approach to Parallelism

Danny Dig
Computer Science Department
University of Illinois at Urbana-Champaign
Email: dig@illinois.edu

***Abstract*—In the multicore era, a major programming task will be to make programs more parallel. This is tedious because it requires changing many lines of code, and it is error-prone and non-trivial because programmers need to ensure non-interference of parallel operations. Fortunately, refactoring tools can help reduce the analysis and transformation burden.**

We present our vision on how refactoring tools can improve programmer productivity, program performance, and program portability. We also present the current incarnation of this vision: a toolset that supports several refactorings for (i) making programs thread-safe, (ii) threading sequential programs for throughput, and (iii) improving scalability of parallel programs.

***Index Terms*—refactoring, parallelism, concurrency.**

I. Introduction

For decades, programmers relied on Moore's Law to improve the performance of their applications. With the advent of multicores, programmers are forced to exploit parallelism if they want to improve the performance of their applications, or when they want to enable new applications and services that were not possible before (e.g., enhanced user experience, better quality of service).

One approach for parallelizing a program is to rewrite it from scratch. However, the most common way is to parallelize a program *incrementally*, one piece at a time. Each small step can be seen as a behavior-preserving transformation, i.e., a refactoring. Programmers prefer this approach because it is safer: they prefer to maintain a working, deployable version of the program. Also, the incremental approach is more economical than rewriting.

However, the refactoring approach is still *tedious* because it requires changing many lines of code, is *error-prone* and is *non-trivial* because programmers need to ensure non-interference of parallel operations. For example, we parallelized several loops using Java's ParallelArray data-structure; this required an average of 10 changes per loop. We spent even more time ensuring that the parallel iterations do not update shared objects or files. Since the library assumes non-interference of parallel operations, it does not protect the data accesses, thus leading to data races.

To reduce the programmer's burden when converting sequential to parallel programs, several tools have been proposed. They come in two distinct flavors: (i) fully automatic tools (e.g., automatic parallelizing compilers [1]–[4]) and (ii) interactive tools (e.g., refactoring tools [5]–[12]). The fundamental difference between these tools is the role of the programmer.

Starting from a sequential program, a non-interactive tool creates a parallel program automatically, without any help from the programmer. When this works it gives great results. Unfortunately, without programmer's domain knowledge, the compiler has limited applicability. To date, the only compiler successes have been in programs involving dense matrix operations and stencil computations. Even though compilers have improved a lot, programmers still parallelize by hand most of the code.

Interactive tools take a completely different approach: sometimes, less automation is better! They let the programmer be in the driver's seat. The programmer is the expert on the problem domain, and so understands the domain concepts amenable to parallelism. The programmer also understands the the current sequential implementation: the program invariants that must be preserved during parallelization, along with the data- and control-flow relationships between parts of the program, and the algorithms and data structures used in the current implementation.

Thus, the interactive approach combines the strengths of the programmer (domain knowledge, seeing the big picture) and the computers (fast search, remember, and compute). The programmer does the creative part: selects code and targets it with a transformation. The tool does the tedious job: checks the safety (this involves searching in many files, by traversing through many functions and through aliased variables), and modifies the program. When the tool cannot apply a transformation, it provides information integrated within the visual interface of an IDE (e.g., Eclipse, VisualStudio), thus allowing a programmer to pinpoint the problematic code.

In the last decade of sequential programming, interactive refactoring tools have revolutionized how programmers approach software design. Without refactoring tools, programmers often over-designed, because it was expensive to change the design once it was implemented. Refactoring tools have enabled programmers to *continuously* evolve the design of large codebases, while preserving the existing behavior. Modern IDEs incorporate refactoring in their top menu, and often compete on the basis of refactoring support.

In the next decade of parallel programming, we envision that refactoring tools for retrofitting parallelism can be similarly transformative. Our current refactoring toolset for improving (i) thread-safety, (ii) throughput, and (iii) scalability seems to indicate so. Empirical evaluation shows that our toolset is useful: it reduces the burden of analyzing and modifying code, it is fast enough to be used interactively, and it correctly applies transformations that open-source developers overlooked.

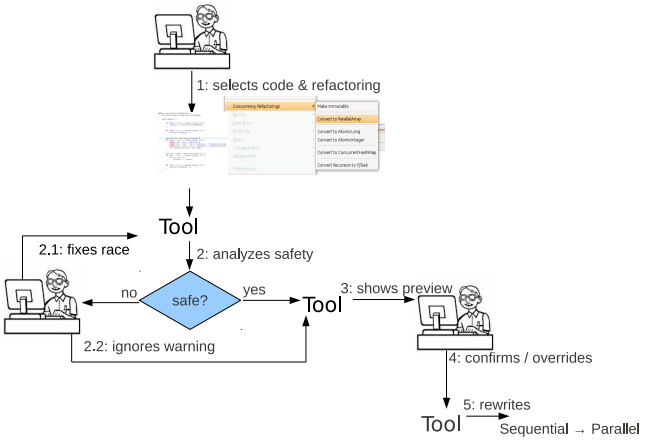

Fig. 1. The process of using a refactoring tool to parallelize code.

Like any other performance optimizations (and unlike sequential refactorings), refactorings for parallelism are likely to make the code more complex, more expensive to maintain, and less portable. We present our vision on how refactoring tools, along with smart IDEs and performance tools, can further improve programmer productivity (by improving the readability and maintenance of parallel code), program performance, and program portability.

II. A Vision for Refactoring Tools for Parallelism

A refactoring toolset for parallelism has several points of interaction with the programmer, shown in Fig. 1. The programmer selects some code and a target refactoring, and the tool analyzes the safety of the transformation. Ultimately, it is the programmer's responsibility to identify all shared data or compute-intensive code and target it with the appropriate refactorings. If some of the refactoring preconditions are not met, the tool raises warnings and highlights the problematic code. The programmer can decide to cancel the refactoring, fix the code, then re-run the refactoring, or he can decide to proceed against the warnings.

By default, the refactoring tool applies the changes only when its analysis determines that it is safe to do so. However, the programmer has the choice to ignore the warnings and apply the changes anyway.

Our growing toolset [7]–[9] of refactorings for parallelism uses the workflow described above (see more details in Section III). 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 has been traditionally associated with improving the structure of the code, thus making the code more *readable* and more *reusable*, even across different platforms. With refactorings for parallelism, the new code is likely to be less readable. Consider the refactoring for parallelizing a loop shown in Fig. 2. The parallel code (on the right-hand side) hides the intent of the original code, thus increasing the code complexity and decreasing the productivity of programmers who need to maintain it. Also, the new code is less portable, since it is fine-tuned for a particular platform.

We envision smart IDEs that treat refactorings for performance intelligently, thus improving both the readability and portability of the parallel code. Also, refactoring tools will need to work in tandem with other tools (e.g., compilers and performance monitors) to achieve maximum performance.

A. Improving Programmer Productivity

When refactoring in the sequential domain, the programmer would throw away the old code and keep the new code. When refactoring for performance, it is desirable 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 these transformations when they are applied by the programmer. Subsequently, these transformations can serve as explicit documentation about how a piece of code evolved, enhancing program understanding. Advanced refactoring engines like Eclipse already provide the recording capability.

The IDE can also provide two views of the same code: a sequential and a parallel view. The programmer would use the sequential view for program understanding, for fixing bugs in the original program, or for adding new features. The programmer would use the parallel view for performance debugging. The code in the sequential view could be lightly annotated to indicate that programmer has applied a performance refactoring. For the example in Fig. 2, the refactoring could leave an `@Parallel` annotation in front of each loop. By asking the IDE to expand this annotation, the programmer would view the parallel code.

B. Improving Code Portability

When programmers need to squeeze the last bit of performance out of their programs, they often resort to transformations that are platform-specific. For example, transformations take into account hardware characteristics like the number of cores, the memory (size, shared vs. distributed), cache line sizes, etc.

Currently, the platform-specific transformations are deeply embedded within the code, thus making the code less portable. To migrate to a new platform, the programmer needs to first undo the platform-specific code, get the platform-independent code, then apply new transformations.

Smart IDEs that understand explicit parallel transformations can help the portability of parallel code: same transformation can have several platform specific implementations. For example, the programmer refactors a loop for parallelism, but the specific transformation depends on whether the code runs on a gaming console, a general-purpose shared-memory computer, or a distributed system. The refactoring tool would provide several alternative implementations of the same transformation. The programmer maintains the portable code, which is annotated with transformations, not mixed with platform-specific code. He can switch to the platform-specific view when needed.

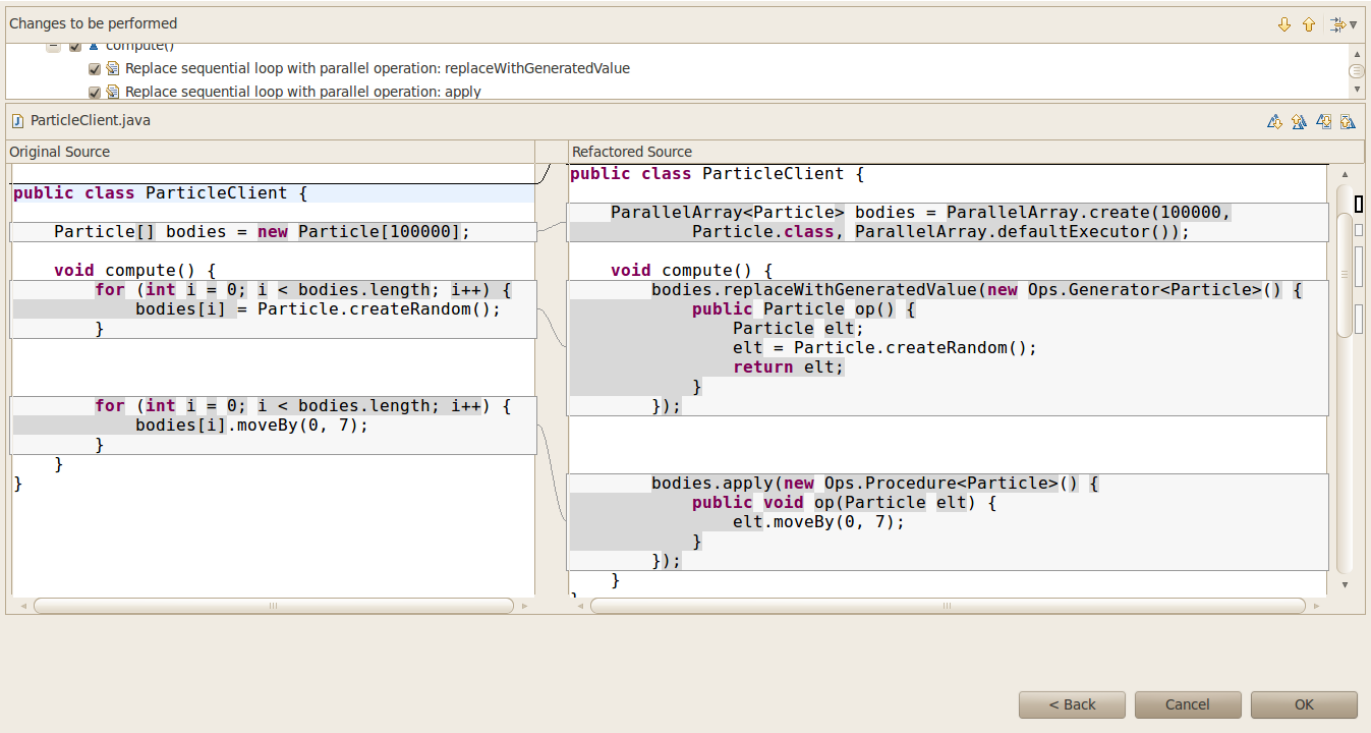

Fig. 2. Using a refactoring tool to parallelize loops using the `ParallelArray` library. The preview shows the sequential code on the left-hand side. The right-hand side shows all the changes that need to be applied.

C. Improving Performance

When deciding what to parallelize, the programmer uses her domain knowledge and he also uses other tools to identify performance bottlenecks. Currently, there is a gap between these tools. There needs to be more focused interactions between refactoring tools and the other tools in the toolbox.

Refactoring tools can take feedback from performance tools like hardware monitors or profilers. After running a program and detecting *performance smells*, performance tools can suggest several refactorings. The programmer in the loop can make informed decisions about which refactorings to apply. The runtime information can also help with the imprecision of the static analysis used in refactoring tools.

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

Refactoring tools and compilers need not compete, but they ought to complement each other. In cases when the compiler cannot automatically parallelize a program, it can provide information and leave this for the refactoring tool who can use feedback from the programmer to get the job done.

III. Our Current Refactoring Toolset for Parallelism

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

We found that parallelizing transformations are not random, but they fell into four categories: transformations that improve the *latency* (i.e., an application feels more responsive), transformations that improve the *throughput* (i.e., more computational tasks executed per unit of time), transformations that improve the *scalability* (i.e., the performance scales up when adding more cores), and transformations that improve *thread-safety* (i.e., application behaves according to its specification even when executed under multiple threads).

The industry trend is to convert the hard problem of introducing parallelism into the problem of 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). Much of the complexity of writing parallel code (e.g., balancing the computation load among the cores) is hidden in the library. Libraries also provide highly scalable, thread-safe collections (e.g., `ConcurrentHashMap`) and lightweight tasks, thread-like entities but with much lower overhead for creation and management.

Our current refactoring toolset uses the 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, previewing changes, and undo.

When parallelizing a sequential program, a programmer needs to (i) make the code thread-safe by protecting accesses to mutable shared data, (ii) make the code run on multiple threads of execution, and (iii) make the performance scalable when adding more cores. Several authors [13]–[15] advocate to first make the code right (i.e., thread-safe), then make it fast (i.e., multi-threaded), then make it scalable.

Our growing toolset currently automates six refactorings, that fall into three categories. Refactorings for thread-safety make a program thread-safe but do not introduce multi-threading yet. Refactorings for throughput add multi-threading. Refactorings for scalability replace existing data structures with highly scalable ones.

These refactorings often require transformations that span multiple, non-adjacent, 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 OO programs that contain a web of heap-allocated objects interconnected to other objects through their fields.

A. Refactorings for Thread-Safety

Before introducing multi-threading, the programmer needs to prepare or enable the program for parallel execution. This involves finding the *mutable* data that will be *shared* across parallel executions. The programmer can decide to (i) synchronize accesses to such data, or (ii) remove either its mutability or shared-ness. Our toolset supports two refactorings for synchronizing accesses to data: one refactoring [7] converts an `int` field to an `AtomicInteger`, a `j.u.c.` library class which provides atomic operations for field updates. Another refactoring converts a `HashMap` field into a `ConcurrentHashMap`, a thread-safe implementation for working with hashmaps. Below we present the refactoring for converting a mutable into an immutable class.

Make Class Immutable

One way to make a whole class thread-safe, is to make it immutable. An immutable class is thread-safe by default, because its state cannot be mutated once an object is properly constructed. Thus, an immutable class can be shared among several threads, with no need for synchronization.

Our refactoring enables the programmer to convert a mutable class into an immutable class. To do so, the tool makes the class and all its fields final, so that they cannot be assigned outside constructors and field initializers. The tool finds all mutator methods in the class, i.e., methods that directly or indirectly mutate the internal state (as given by its fields). The tool converts these mutator methods 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 like `String` where `replace(oldChar, newChar)` or `toUpperCase()` return a new `String` with some characters replaced.

Next, the tool finds the objects that are entering from outside (e.g., as method parameters) and become part of the state, or objects that are part of the state and are escaping (e.g., through return statements). It clones these objects, so that the class state can not be mutated by a client class who holds a reference to these state objects. Lastly, 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.

Our comparison with open-source classes that were manually refactored for immutability shows that the tool is much safer: it finds subtle mutations and entering/escaping objects that programmers overlooked. However, not all classes can be made immutable. For example, if a mutator method already returns an object, the tool cannot convert it into a factory method. Also, due to the extra overhead of copying state, using this refactoring is advisable only when mutations are not frequent. More details about this refactoring can be found in [8].

B. Refactorings for Throughput

Once a program is threadsafe, multi-threading can be used to improve its performance. The programmer could manage himself a raw thread (e.g., create, spawn, wait for results), or he could use a programmer-friendlier construct, a *lightweight task*, managed automatically by a framework. Our toolset supports two such refactorings. One refactoring [7] converts a sequential divide-and-conquer algorithm into an algorithm which solves the recursive subproblems in parallel using Java's `ForkJoinTask` framework [14]. Another refactoring parallelizes loops over arrays.

Parallelize Loop

This refactoring parallelizes loop iterations over an array via `ParallelArray` [14], a parallel library upcoming in Java. `ParallelArray` is an array data structure that supports parallel operations over the array elements. For example, one can `apply` a procedure to each element, or can `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 data type of the array, and it replaces loops over the array elements with the equivalent parallel operations from `ParallelArray`. Consider the example in Fig. 2. The first loop replaces each element with another random element, thus the tool invokes the `replaceWithGeneratedValue` parallel operation. The second loop applies the `moveBy` function to each element, thus the tool invokes the `apply` parallel operation.

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

At the heart of the tool lies a data-flow analysis that determines objects that are shared among loop iterations, and

detects writes to the shared objects. The analysis works with both programs in source code and in byte code (e.g., jar-packaged libraries). When the analysis finds writes to shared objects, it presents the user a stack of code statements that resulted in the objects being shared. These statements are hyper-linked to the original source code, thus helping the developer to find the problematic code.

Although we were able to refactor several real programs and the analysis was fast and effective, not all loops can be refactored. For example, a loop must (i) iterate over all the array elements, (ii) not contain blocking I/O calls, and (iii) not contain writes to shared objects. More information about the tool can be found in [9].

C. Refactorings for Scalability

One must not sacrifice thread-safety and correctness in the name of performance. However, a naive synchronization scheme can lead to serializing an application, thus drastically reducing its scalability. This usually happens when working with low-level synchronization constructs like locks. Locks are the goto statements of parallel programming: they are tedious to work with, and error prone. Too many locks 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 which uses compare-and-swap hardware instructions. Another refactoring converts a `HashMap` to `ConcurrentHashMap`.

Convert `HashMap` to `ConcurrentHashMap`

If a class contains a `HashMap` field that is read/written in parallel, it must synchronize the accesses to the map. The programmer can use a common lock, or can use a synchronized wrapper over a `HashMap` (e.g., `Collections.synchronizedMap(aMap)`). The synchronized `HashMap` achieves its thread-safety by protecting *all* accesses to the `map` with a *common* lock. This results in poor scalability when multiple threads try to access different parts of the map simultaneously, since they contend for the lock.

A better alternative is to refactor the `map` field into an `ConcurrentHashMap`, a thread-safe, highly scalable implementation for hash maps provided by the `j.u.c.` library (all readers run in parallel, 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 (i) first check whether a map contains a certain key, (ii) if not present, create the $value$ object, and (iii) place the $\langle key, value \rangle$ in the map. The tool replaces such an updating pattern with a call to `ConcurrentHashMap`'s `putIfAbsent` which executes the update *atomically*, without locking the entire map.

Comparison with 77 refactorings performed by open-source developers shows that they frequently performed this refactoring incorrectly, forgetting to replace some compound updates with the new atomic APIs in `ConcurrentHashMap`. However, this refactoring is not always applicable, for example when an application needs to lock the entire map for exclusive access (e.g., for a whole traversal). More details about this refactoring can be found in [7].

D. Lessons Learned

Building this refactoring toolset taught us several lessons. One, programmers often use parallel libraries, thus refactoring tools need to support such libraries. Two, to keep the programmer engaged, refactoring tools need to finish in less than thirty seconds. Thus, they must use efficient, on-demand program analyses. Three, program analysis libraries and IDEs with excellent AST rewriting capabilities are essential for building refactoring tools. Four, once a program is parallel, it must remain maintainable, i.e., readable and portable. Five, 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], one needs many more refactorings. We are constantly expanding the number of refactorings, inspired by the problems that industry practitioners face everyday when they parallelize their programs. Also, we will start tackling the problems of readability, portability, and interactiveness with other performance tools.

IV. Other Refactoring Tools for Parallelism

The earliest work on interactive tools for parallelization stemmed from the Fortran community, and targeted loop parallelization. Interactive tools like ParaScope [5] and SUIF Explorer [6] relied on the user to specify what loops to interchange, align, replicate, or expand. The tool computed and displayed to the programmer various information like dependence graphs. However, this work was done in the context of numerical computation on scalar arrays and did not deal with the sharing through the heap prevalent in object-oriented programs.

Reentrancer [10] 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, i.e., it makes accesses to global data thread-safe. We have manually performed this refactoring several times when eliminating writes to global shared objects pointed by our tool [9].

Fuhrer [11] 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 (e.g., asynchronous tasks, clocks). The proposed set of refactorings converts sequential code to make use of these parallel constructs.

The Photran [12] project also plans to support several concurrency refactorings for high-performance computing in Fortran.

V. Conclusion

Today's sequential programs are tomorrow's legacy programs, unless they are retrofitted for parallelism. Refactoring sequential programs for parallelism is time-consuming and

error-prone. It also leaves the code less readable and less portable. Fortunately, interactive refactoring tools can alleviate the burden of analyzing and transforming these programs. When combined with smart IDEs and other tools, future refactoring tools would tackle the problems of readability and portability as well.

Although our examples and refactorings are using Java and Eclipse, they are representative for other OO languages like C++ and C# and can also be accomplished in other environments.

ACKNOWLEDGMENT

This work is partially funded by Intel and Microsoft through the UPCRC Center at Illinois. The author would like to thank Paul Adamczyk, Nicholas Chen, Milos Gligoric, Ralph Johnson, Fredrik Kjolstad, Jeff Overbey, and Cosmin Radoi for providing valuable feedback on drafts of this paper.

REFERENCES

[1] D. J. Kuck, "Automatic program restructuring for high-speed computation," in *CONPAR 81: Conference on Analysing Problem Classes and Programming for Parallel Computing*, 1981, pp. 66–84.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferreant, "An overview for the ptran analysis system for multiprocessing," *J. Parallel Distrib. Comput.*, vol. 5, no. 5, pp. 617–640, 1988.

[3] R. Allen, D. Callahan, and K. Kennedy, "Automatic decomposition of scientific programs for parallel execution," in *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1987, pp. 63–76.

[4] S. P. Amarasinghe, J.-A. M. Anderson, M. S. Lam, and A. W. Lim, "An overview of a compiler for scalable parallel machines," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, 1993, pp. 253–272.

[5] K. Kennedy, K. S. McKinley, and C. W. Tseng, "Interactive parallel programming using the parascope editor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 3, pp. 329–341, 1991.

[6] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam, "Suif explorer: an interactive and interprocedural parallelizer," in *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1999, pp. 37–48.

[7] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in *31st International Conference on Software Engineering (ICSE)*, 2009, pp. 397–407.

[8] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Refactoring for immutability," UIUC, Tech. Rep. http://hdl.handle.net/2142/16399, June 2010.

[9] D. Dig, C. Radoi, M. Tarce, M. Minea, and R. Johnson, "Refactoring for loop parallelism," UIUC, Tech. Rep. http://hdl.handle.net/2142/14536, September 2009.

[10] J. Wloka, M. Sridharan, and F. Tip, "Refactoring for reentrancy," in *ESEC/SIGSOFT FSE*, 2009, pp. 173–182.

[11] R. Fuhrer and V. Saraswat, "Concurrency refactoring for x10," in *3rd ACM Workshop on Refactoring Tools*, 2009, pp. 1–4.

[12] M. Mndez, J. Overbey, A. Garrido, F. Tinetti, and R. Johnson, "A catalog and classification of fortran refactorings," in *11th Argentine Symposium on Software Engineering (ASSE 2010)*, 2010, pp. 1–10.

[13] D. Dig, J. Marrero, and M. D. Ernst, "How do programs become more concurrent? A story of program transformations." MIT, Tech. Rep. http://hdl.handle.net/1721.1/42832, September 2008.

[14] D. Lea, *Concurrent Programming in Java*. Addison-Wesley, 2000.

[15] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Addison-Wesley, 2006.