Parallel Programming Must Be Deterministic by Default

Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve and Marc Snir
University of Illinois at Urbana-Champaign
{bocchino,vadve,sadve,snir}@illinois.edu

Abstract

In today's widely used parallel programming models, subtle programming errors can lead to unintended nondeterministic behavior and hard to catch bugs. In contrast, we argue for a parallel programming model that is *deterministic by default*: deterministic behavior is *guaranteed* unless the programmer explicitly uses nondeterministic constructs. This goal is particularly challenging for modern object-oriented languages with expressive use of reference aliasing and updates to shared mutable state. We propose a broad research agenda in support of this goal, and we describe some of our own work to further that agenda.

1 Motivation

The general-purpose computing industry is at a major crossroads. Power constraints and design complexity are driving processor architects to place increasingly many compute cores on a chip. Commodity applications, developed by programmers with a wide range of skills, must harness this parallelism for increased performance.

Computations that use concurrency primarily for performance can be modeled as taking an input and computing some output. In such *transformational* computations, concurrency is not part of the problem specification. This is in contrast to event-based or *reactive* systems (such as web servers) where concurrency may be necessary for functionality (e.g., a distributed system) or correctness (e.g., bounded response times in interactive programs).

Many, though not all, transformational computations are *deterministic*: a given input is always expected to produce the same output. This category is vast; it includes almost all scientific computing, signal processing, encryption/decryption, sorting and searching, compiler and program analysis, and processor simulators.

Unfortunately, mainstream parallel programming models today provide no special assistance for programming deterministic algorithms. In fact, parallel applications today primarily use threads and shared memory, whether through libraries like pthreads and Intel's Threading Building Blocks (TBB), or multithreaded languages like Java and C#. Programs written in these models can be extremely difficult to understand and debug. First, the use of shared memory is uncontrolled: any access to a shared variable may result in an interaction with another thread. Second, there is no guarantee of determinism: the execution can follow one of a large number of interleavings of dependent memory accesses, and different interleavings can produce different results. A correctly written program may be deterministic, but correctness is difficult to check.

In a recent article [12], Lee eloquently argues that if we are to have any hope of simplifying parallel programming for the vast majority of programmers and applications, then parallel programming models must greatly constrain the possible interleavings of program executions. In particular, deterministic algorithms must be expressible in a style that *guarantees* determinism, and nondeterministic behaviors, where desired, must be requested explicitly. More generally, as we discuss in Section 2, enforcing deterministic semantics simplifies composing, porting, reasoning about, debugging, and testing parallel software.

Deterministic parallel programming models exist today. However, much of the prior work on determinism has been in a context that is not general enough (e.g., regular data parallel operators [15]) and/or requires a significant departure from mainstream programming styles (e.g., a pure functional style [13]). In contrast, many modern applications are developed in an object-oriented style with expressive use of features such as reference aliasing, imperative updates, dynamic method dispatch, and extensive reuse of sophisticated libraries and frameworks. Programmers are familiar with this style and productive in using it. We believe it is crucial for parallel languages to support this style.

In this paper, therefore, we argue that to simplify parallel programming, determinism must be brought to *mainstream object-oriented programming languages*. A broad research agenda will be required to achieve this goal:

How to guarantee determinism in a modern object-oriented language? For reasons discussed in Section 3, our philosophy is to provide *static* guarantees through a combination of a type system and (straightforward, intraprocedural) compiler analysis when possible, and to fall back on runtime checks (that either result in exceptions or cause the execution to roll back and retry) only when compile-time guarantees are infeasible. The key is to determine when concurrent tasks make conflicting accesses. The language can help provide this capability by enforcing structured parallel control flow (to make it easy to analyze which tasks can execute concurrently) and by providing type system mechanisms to convey explicitly what data can be accessed or updated by a specific task. These goals, and some techniques to achieve them, are discussed in Section 4.1.

How to provide sound guarantees when parts of the program either cannot be proved deterministic or have "harmless" nondeterminism? Libraries and frameworks written by expert programmers tend to be widely reused, carefully designed, and thoroughly tested. Such code may include components that are not deterministic in isolation, yet can be combined to provide deterministic results. For example, a sequence of commutative inserts to a concurrent search tree within a parallel loop can be executed in arbitrary order and yet give deterministic results, as long as no other operation (e.g., a find) is interleaved between those inserts. Languages should enable such libraries to express contracts that can be enforced by the compiler. The system can then ensure that a client application using the library is deterministic so long as the library implementation meets its specification. These goals are discussed in Section 4.2.

How to specify explicit nondeterminism when needed? A deterministic-by-default language may need to support transformational computations that permit more than one acceptable answer. If so, the language must achieve three goals. First, any nondeterministic behavior must be *explicit*, e.g., using nondeterministic control statements; hence the term "deterministic by default." Second, the nondeterminism should be carefully *controlled* so that programmers can reason about possible executions with relatively few interleavings. Third, nondeterministic code should be *isolated* from deterministic code so that the programmer can reason deterministically about the rest of the application. These goals are discussed in Section 5.

How to make it easier to develop and port programs to a deterministic-by-default language? The cost of porting code to a new language can be significantly reduced if the language is downward compatible with an existing, widely used language and supported by tools that allow incremental porting. A properly designed language will also have substantial, long-term productivity benefits. These opportunities are discussed in Section 6.

2 Benefits and Costs of Determinism

A parallel program is deterministic if, for a given input, every execution of the program produces identical *externally visible* output. Many transformational *parallel algorithms* have this basic property. Here we restrict our attention to nondeterminism introduced by concurrency, and we ignore nondeterminism that can also occur in sequential programs (e.g., calls to `gettimeofday`).

A parallel programming language is deterministic if every legal program in that language is deterministic. A deterministic parallel programming model (language or library) has significant advantages:

- A deterministic program can be understood without concern for execution interleavings, data races, or complex memory consistency models: the program behavior is completely defined by its sequential equivalent.
- Programmers can *reason* about programs, *debug* them during development, and *diagnose* error reports after deployment using the development techniques and tools currently used for sequential programs.
- Independent software vendors can *test* codes as they do for sequential programs, without being concerned about the need to cover multiple possible executions for each input. The same test suites developed for the sequential code can be used for the parallel code.
- Programmers can use an incremental parallelization strategy, progressively replacing sequential constructs with parallel constructs, while preserving program behavior.
- Two separately developed but deterministic parallel components should be far easier to compose than more general parallel code, because a deterministic component should have the same behavior regardless of the external context within which it is executed.

Deterministic semantics can also help with parallel performance. In particular, an explicitly parallel loop has *sequential semantics* with a *parallel performance model*: its performance will be what one would expect by assuming that parallel loop iterations do execute in parallel. In effect, both the semantic model and the performance model for such a program can be defined using obvious composition rules [6]. Further, deterministic programming models can enable programmers to spend more time on performance tuning (often the determining factor in performance for real-world software) and less time finding and eliminating insidious concurrency bugs.

Some methods of enforcing determinism can hinder performance, either by introducing high runtime overheads or by restricting expressiveness. We believe, however, that this problem is not fundamental. First, we can enforce determinism *without runtime checks* for a large class of programs (Section 4.1). Second, when we must relax determinism guarantees for high performance, we can often still provide a *library or framework interface* that appears deterministic to the user, with the nondeterministic behavior hidden inside the implementation (Section 4.2).

3 Guaranteeing Determinism

Practical alternatives to thread-based programming that work for object-oriented languages are starting to emerge for production use. The Cilk++ [1] language provides a task programming model that simplifies the expression of parallelism. Java's ForkJoinTask [2], Intel's Threading Building Blocks [17], and Microsoft's TPL [3] provide similar capabilities using libraries. Further, regular data parallel operations on arrays (update, filter, map, reduce) can be expressed elegantly and customized using emerging libraries and frameworks such as ParallelArray for Java [2] and TBB's templates for C++.

These emerging object-oriented languages and libraries are a valuable step; however, they lack any guarantees of determinism or even freedom from data races in the presence of shared references to mutable objects. Similarly, emerging research techniques like Galois [11] and Prometheus [5] provide deterministic semantics for "correctly written" programs, but provide no guarantee that those requirements are met at compile-time or runtime.

The fundamental challenge in guaranteeing deterministic parallel programming semantics is for the system to "see" and check the shared memory interactions between different parallel computations in the program. Broadly, there are four approaches for this kind of checking:

- **Language-based** approaches (discussed further in Section 4) use language extensions, usually in the type system, for compile-time enforcement of sharing constraints in a parallel program.

- **Compiler-based** approaches use parallelizing compiler technology (e.g., [10, 8]) to transform sequential programs (with or without annotations) into parallel form.

- **Software runtime** approaches (e.g., [18, 22]) use software runtime checks to detect, and possibly speculation to recover from, violations of deterministic semantics in the execution of a parallel program.

- **Hardware runtime** approaches (e.g., hardware-supported thread-level speculation (TLS) [19, 16]) use hardware support to achieve the same goals but with less overhead, at the cost of increased hardware complexity.

The four approaches involve different tradeoffs and can be combined in different ways into a composite solution. A language-based approach allows a high degree of programmer control, documents the available parallelism for future developers, and makes program behavior and performance characteristics explicit. Language information can also specify properties that hold at interface boundaries, enhancing modularity. A compiler approach can reduce the burden of writing parallel code, compared to a pure language approach. However, for all but very regular codes auto-parallelization is quite difficult; and even where successful it can be brittle (small code changes can destroy performance) and hard to understand.

A robust runtime can reduce or eliminate the need for the programmer or compiler to get the sharing patterns correct. However, runtime approaches can add overhead and make performance characteristics opaque. Further, runtime techniques are inherently input-dependent: one input may avoid determinism violations, while another one causes a violation.

Overall, explicitly parallel, language-based approaches are the only ones that provide the benefits of performance control, explicit interfaces, modularity, documentation, and compile-time enforcement. We therefore believe that such an approach is the most attractive in the long term. Such an approach can be combined with limited runtime software and hardware checking to enable greater expressivity, where needed.

4 Language Mechanisms for Determinism

It is particularly challenging to develop language mechanisms that provide deterministic semantics for general, object-oriented languages. We discuss our ideas for addressing these challenges below.

4.1 Effect Systems

We believe that an important part of the solution to controlling sharing is an object-oriented *effect system* [14, 9, 7] providing annotations that partition the heap and declare which parts of the heap are read and written by each task. An effect system could easily show, for example, that two distinct objects are being created at every recursive call of a divide and conquer pattern, so the subproblem computations do not interfere.

In the Deterministic Parallel Java (DPJ) project [4], we are developing a sophisticated effect system that partitions the heap into hierarchical *regions* and uses those regions to disambiguate accesses to distinct objects, as well as distinct parts of the same object, referred to through data structures such as sets, arrays, and trees [7]. A simple *local* type-checker (no whole-program analysis is required) can then ensure that there are no conflicting accesses to overlapping memory operations between concurrent tasks. In a correct DPJ program, nondeterminism cannot happen "by accident": any such behavior must be explicitly requested by the user, and a DPJ program with no such request has an "obvious" sequential equivalent.

When static checks do not work, either because the analysis is not possible or the annotation burden is not justified by the performance gains, we must fall back on runtime techniques. One approach is to use software speculation [22], with hardware support [16] to reduce overhead if it is available. An alternative approach is a fail-stop model that aborts the program if a deterministic violation occurs [18]. This approach gives a weaker guarantee, but it avoids the overhead of logging and rollback. Even in such cases, support for speculation will still be valuable, for two reasons: it can simplify initial porting of programs (see Section 6) and it can be used to express algorithms that are *inherently speculative*, where new tasks must be launched speculatively or the entire algorithm would become serial [11].

4.2 Encapsulating Complex Behaviors

In realistic programs, the guarantee of determinism may have to be weakened for parts of the program, for performance or expressivity. However, if we can encapsulate those parts behind an interface with suitable contracts, and guarantee that client code satisfies those contracts, then we can still provide sound guarantees for the rest of the program. This approach is attractive because the encapsulated code can often be placed in libraries and frameworks written by expert programmers skilled in low-level parallel programming and performance issues.

4.2.1 Local Nondeterminism

Algorithms that are deterministic *overall* may benefit from "locally nondeterministic" behaviors for higher performance. Some operations, such as associative reductions, have deterministic final results but require a schedule-dependent order of internal operations for high performance. Similarly, some sequences of operations that produce deterministic final behavior as seen by an external observer may have schedule-dependent internal representations (i.e., memory state). Examples include a sequence of insert operations on a set or on a splay tree, or computing the connected components of a graph.

Experienced programmers should be able to write such computations and encapsulate their code in libraries that have deterministic external behavior, with well-defined properties. E.g., programmers should be able to define *pure, associative* operators, as in languages like Fortress [20], which can then be used by a generic reduction or parallel prefix algorithm. The "pure and associative" requirement is a contract that can be checked by the compiler, possibly relying on effect annotations.

4.2.2 Unsoundness

In realistic applications, some parts of the program may in fact be deterministic (unlike the "locally nondeterministic" cases above), yet perform operations that cannot feasibly be proved sound by the type system or runtime checks. One example is a tree rebalancing. If the type system can guarantee that a data structure is a tree, then this guarantee can support sound parallel operations, such as a divide and conquer traversal that updates each subtree in parallel. However, rebalancing the tree in a way that retains the guarantee may be difficult, without imposing severe alias restrictions such as unique pointers. It is also difficult for a runtime to efficiently check that the tree structure is maintained after a rebalancing.

We believe a practical solution in such cases is to allow unsound operations, i.e., operations that may break the determinism guarantee, but to encapsulate those operations inside well-defined data structures and frameworks using traditional object-oriented encapsulation techniques (private and protected fields and inner classes) supplemented by effect analysis and/or alias control. The effect and alias restrictions can help keep track of what is happening when references in the rest of the code point to data inside an encapsulated structure [9]. Then the compiler can use the guarantees provided by the data structure or framework interface to provide sound guarantees for the rest of the program.

5 Explicit Nondeterminism

For some algorithms, nondeterministic behavior is considered acceptable, e.g., branch-and-bound search, graph clustering, and many graphics and media processing applications. In all these examples, the final result must meet some acceptance criterion, and multiple solutions that meet the criterion are permissible. This property can be exploited to write a simpler, or better performing, parallel algorithm. In contrast to encapsulated nondeterminism in the context of a deterministic program (Section 4.2.1), here the *visible program behavior* is nondeterministic.

We wish to express such algorithms while achieving the following goals. First, nondeterminism is explicitly expressed, e.g., using a nondeterministic control statement. Second, nondeterminism is carefully controlled, so that the programmer need reason about only relatively few program interleavings. Third, the nondeterministic part of the application should not compromise the ability to reason with determinism for the rest of the application.

We are investigating how to achieve these goals in DPJ. For example, suppose we have two parallel iteration constructs, one for iterating over independent objects (`foreach`), and another for iterating over objects that may overlap (`foreach_nd`, where "nd" stands for "nondeterministic"). `foreach` is deterministic: it guarantees that the result is as if the iterates were executed atomically and in sequential order. `foreach_nd` is nondeterministic and guarantees that the result is as if the iterates were executed atomically and in some arbitrary order. The `foreach_nd` construct achieves the three goals above as follows:

- It makes nondeterministic behavior explicit.

- It allows the programmer to understand the behavior of the loop by considering only the iteration order, and not arbitrary interleavings of memory access operations, or even arbitrary expressions or statements, in the iterates.

- It ensures that program units that do not contain a `foreach_nd` statement in their dynamic scope execute deterministically.

The Galois system [11] provides capabilities similar to our `foreach` and `foreach_nd`, except that it is possible to write incorrectly synchronized programs (for example, that have data races) in Galois. Our aim is to leverage the effect system described in Section 4.1 to *guarantee* the properties described above.

6 Usability

Common concerns with language-based solutions are the cost to rewrite existing programs and to learn new language features. We believe that (a) the costs tend to be exaggerated and the benefits underestimated; and (b) strong technical solutions can significantly reduce the costs. We discuss both points briefly in turn.

(1) First, note that we are proposing a small set of extensions to an established base language (such as Java or C#), *not* an entirely new language. This fact should mitigate the up-front cost of both learning the new features and writing code that uses the new features. Further, the extra effort to learn and use new language features is likely to be dwarfed by the effort required to write, port, tune, and test parallel code. A well-designed language that simplifies the latter tasks can more than justify the learning curve. Note also that because we are extending a base language, porting can be done *incrementally*, e.g., kernel by kernel.

(2) Although object-oriented effect notations require some extra effort from the programmer, such effort is not wasted. First, effect annotations on methods provide a compiler-checkable *interface* that allows sound, modular reasoning about program components, even in the absence of all the source code (such as for a library or framework). Thus, the annotations enhance modularity and composability. Second, the reasoning required to introduce regions and effects is exactly the reasoning required to understand the sharing patterns in the code. In fact, the region and effect mechanisms give programmers a concrete guide for how to carry out such reasoning.

(3) Non-trivial real-world applications are long-lived, and initial development or porting costs are usually a small fraction of long-term maintenance and enhancement costs. A language that simplifies testing and documents sharing patterns in the code reduces maintenance costs.

(4) Finally, we note that current threads-based languages have woefully inadequate shared memory models. The only memory model accepted today guarantees sequential consistency for data race-free programs, as for Java and (soon) C++. The difficulty lies in the semantics of data races. C++ does not provide any; this is untenable for a safe language. Java provides semantics that are complex and fragile. If we are to move towards safe parallel languages with tractable memory models, we *must prohibit data races for all allowed programs*. A type and effect system, as discussed in Section 4.1, could accomplish this goal with low runtime overhead.

Some *technical solutions* can further reduce the cost of using new language extensions:

- *Effect inference*: In DPJ, we are exploring how judicious use of effect inference (inferring region and effect annotations) can reduce the programming burden of type annotations for determinism [21].

- *Runtime checks*: The language can provide runtime checks, as described in Section 4.1, so that large programs can initially be ported without all the effect annotations needed for compile-time checking. The overheads of runtime checks can then be incrementally tuned away by introducing effect annotations where the benefits justify the effort.

- *Integrated development environment (IDE)*: An IDE can use sophisticated *interactive* compiler parallelization technology, combined with modern refactoring technology, to assist the initial porting process. Making porting a one-time effort allows such an environment to use powerful, but potentially slow, interprocedural parallelization techniques (the strengths of compilers); while making it interactive allows programmers to influence the process and avoid the problems of poor or brittle performance (the weaknesses).

7 Summary

We agree with Lee [12] that unrestricted use of threads is not an acceptable model for shared memory parallel programming. In this paper we have argued that appropriate language design can bring determinism in a much more flexible way to shared memory parallel computing, even in rich, imperative, object-oriented languages. The inevitable residuum of low-level shared memory code written by experts can be encapsulated into libraries and used by general programmers working within a safe, productive parallel programming environment. The up-front cost of using new language features will be more than outweighed by the benefits over the life of an application, and reduced through careful language design and appropriate interactive development tools. Achieving these goals will require a concerted research effort by the language, compiler, and architecture communities.

Acknowledgments

This work is funded by Microsoft and Intel through the Universal Parallel Computing Research Center (UPCRC) at the University of Illinois and by NSF grants 07-02724, 07-20772, 08-33128 and 08-33188.

References

[1] http://www.cilk.com/.

[2] http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/forkjoin/package-summary.html.

[3] http://research.microsoft.com/en-us/projects/tpl/.

[4] http://dpj.cs.uiuc.edu.

[5] M. D. Allen et al. Serialization sets: A dynamic dependence-based parallel execution model. *PPOPP*, 2009.

[6] G. E. Blelloch. Programming parallel algorithms. *CACM*, 1996.

[7] R. L. Bocchino, V. S. Adve, et al. A type and effect system for deterministic parallelism in object-oriented languages. Technical Report UIUCDCS-R-2009-3032, University of Illinois at Urbana-Champaign, 2009.

[8] M. J. Bridges et al. Revisiting the sequential programming model for the multicore era. *MICRO*, 2008.

[9] N. R. Cameron et al. Multiple ownership. *OOPSLA*, 2007.

[10] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[11] M. Kulkarni et al. Optimistic parallelism requires abstractions. *PLDI*, 2007.

[12] E. A. Lee. The problem with threads. *Computer*, 2006.

[13] H.-W. Loidl et al. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 2003.

[14] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. *POPL*, 1988.

[15] M. Metcalf and J. Reid. *Fortran 90 Explained*. Oxford University Press, 1992.

[16] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. *PPOPP*, 2003.

[17] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

[18] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *TOPLAS*, 1998.

[19] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. *HPCA*, 1998.

[20] Sun Microsystems, Inc. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Inc., 2008.

[21] M. Vakilian et al. Inferring method effect summaries for deterministic parallel java. Technical Report UIUCDCS-R-2009-3038, University of Illinois at Urbana-Champaign, 2009.

[22] A. Welc et al. Safe futures for Java. *OOPSLA*, 2005.