Parallel Programming Must Be Deterministic By Default

Robert Bocchino, Vikram Adve, Sarita Adve and Marc Snir

University of Illinois at Urbana-Champaign

{bocchino,vadve,sadve,snir}@illinois.edu

1. Motivation

The general-purpose computing industry is at a major crossroads. Power constraints and design complexity have pushed microprocessor designers to use multiple execution cores on a single die, with 4-16 cores being commonplace today, many tens of cores expected in the next 3-5 years, and some projections claiming hundreds of cores on a chip within a decade. Increases in application performance will depend on their ability to harness this parallelism. In the past, new software capabilities and technologies have been strongly driven by increases in processor performance (keeping up with increasing memory capacity and network bandwidth). In the future, such advances in mainstream applications will only occur if software developers are able to harness parallelism for higher performance.

Mainstream applications today primarily use threads programming for parallelism, whether through libraries like `pthreads` and Intel's Threading Building Blocks (TBB), or multithreaded languages like Java and C#. There is broad consensus, however, that programs written in current threads-based programming models can be extremely difficult to understand and debug. A root cause of the problem is that the execution of a shared-memory program can follow one of a large number of interleavings of dependent memory accesses, and some of these interleavings can potentially produce different results.

In contrast, a large class of computations is inherently deterministic; i.e., a given input for such a (correct) computation always produces the same output. Specifically, concurrency is used for broadly two purposes. For "reactive" computations like servers, interactive games, and embedded systems, concurrency is part of the problem specification; e.g., serving multiple external requests. This concurrency is usually necessary for *correctness*; e.g., to ensure bounded response times. On the other hand, for "transformative" computations, concurrency is not part of the problem specification and is not required for correctness. Here, concurrency is used solely for performance – to produce the output faster. Reactive computations may internally include transformative computations; e.g., a physics simulation in a game. Many, though not all, transformative algorithms are deterministic (a given input always produces the same output), and programming them with a fundamentally non-deterministic model unnecessarily introduces extraordinary complexity.

There is a rich literature discussing the merits of determinism and proposing deterministic models and languages. In a recent article [10], Lee eloquently argues that if we are to have any hope of simplifying parallel programming for the vast majority of mainstream 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 non-deterministic behaviors, where desired, must be requested explicitly.

Much of the prior work on determinism, however, has been in a context that is not general enough (e.g., regular data parallel operators [14]) and/or requires significant departure from mainstream programming styles (e.g., pure functional [12], dataflow [7], and actor [10] styles). In contrast, a large fraction of modern applications are developed in an object-oriented style with expressive use of features such as aliasing through references, imperative updates, dynamic method dispatch, and extensive reuse of sophisticated libraries and frameworks. Programmers are already familiar with this style, and there is a large existing code base written in languages such as Java, C++ and C#. Much of this code will need to be ported to multicore, as it is simply not feasible to rewrite it all from scratch in a new language. Thus, we believe it is crucial for parallel languages to support an object-oriented style, including the use of O-O libraries and frameworks. [1] In this paper, 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? Our philosophy is to provide static guarantees through a combination of a type system and (straightforward) compiler analysis when possible, and to fall back on runtime checks (that result in exceptions) only when compile time guarantees are infeasible. The key is to determine when concurrent tasks make conflicting accesses.

[1] A potential alternative to language mechanisms is automatic parallelization of sequential programs, either via a parallelizing compiler [8], or hardware-supported thread-level speculation (TLS) [6, 11], or both. Automatic parallelization can work well in some cases (especially very regular parallelism, such as SIMD), but it is not a complete solution. Automatically parallelized codes often have *fragile* performance (small changes can cause parallelization to fail); they are hard to tune for performance; expressing parallel algorithms in sequential notation is likely to be frustrating; and such codes do not document the available parallelism for future developers.

The language can enable this by restricting parallel control flow (and hence which tasks can execute concurrently) and by providing type system mechanisms to convey explicitly how a specific task or method shares data. These techniques are discussed in Section 3.2.

How to provide sound guarantees when parts of the program either cannot be proved deterministic or have "harmless" non-determinism? Libraries and frameworks are often written by expert programmers and widely reused. Such code will include components that cannot be proven deterministic but is often well designed and thoroughly tested. For example, commutative inserts to a concurrent search tree may be safe within a parallel loop as long as no other operation (e.g., a find) is interleaved between those inserts. Languages should enable such libraries to express enforceable contracts that can be checked by the compiler to 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 3.3.

How to specify explicit non-determinism when needed? There are transformational computations that may provide higher performance by permitting more than one acceptable answer (e.g., branch and bound search or graph clustering [3]). In these cases, the language must achieve three goals. First, any non-deterministic computations must be declared explicitly; hence the term "deterministic by default." Second, any such code must be guaranteed free of data races (ignoring any "trusted" code in low-level libraries of frameworks). Finally, such code must be encapsulated in a way that allows checking the deterministic properties of the rest of the application. These goals are discussed in Section 4.

How to make it easier to port programs to a deterministic-by-default language? The up-front cost of porting or developing code using new language features has substantial, long-term productivity benefits that are often not recognized. Furthermore, the up-front cost can be reduced using both language design techniques and sophisticated interactive porting tools. These opportunities are discussed in Section 5.

The following sections first elaborate on the benefits of determinism; we then discuss the above research issues, including alternative solutions, current approaches we are investigating, and problems that remain open for future work.

2. Benefits of Determinism

A parallel program is deterministic if, for a given input, every execution of the program produces identical, *externally visible* output. Although this is an idealization for reasons explained in Section 4, many transformative *algorithms* do have this basic property. A parallel programming language is deterministic if every legal program in that language is deterministic.

Consider a parallel program whose only parallel construct is a parallel loop, where the iterations are required to be independent. The outcome of such a program is the same as if the loop were executed sequentially; the parallel loop construct does not change the semantics of the program. We say that such a program has *sequential semantics* and a *parallel performance model*: a program that completes will have computed the same results that a sequential program would have computed; and its performance will be what one would expect by assuming that parallel loop iterates 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 [4].

Well-structured deterministic parallel constructs need not be limited to parallel loops: one can add data parallel operators, reduction operations, barrier synchronization, producer-consumer synchronization such as futures, etc. As Section 3 observes, one can even add imperative updates of more complex shared data structures like sets and trees.

More generally, a deterministic parallel programming model 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 patterns and tools currently used for sequential programs.
- ISVs can *test* codes as they do now, without being concerned about the need to cover multiple possible executions for each input.
- 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 (even if they share the same low-level parallel run-time system): with a well-defined programming language, a deterministic component should have the same behavior regardless of the external context within which it is executed (with some constraints on the behavior of that external context, as in Section 4).

3. Providing Deterministic Semantics

We next discuss our vision for providing deterministic parallel programming semantics using a combination of language mechanisms, compiler analysis, and runtime checks.

3.1 State of the Art

As discussed in Section 1, we need practical alternatives to `pthreads` or Java threads that work for object oriented languages. Some of these techniques are starting to emerge. Regular data parallel patterns on arrays (update, filter, map, reduce) can be expressed elegantly and customized using

emerging libraries and frameworks such as Parallel Array for Java [1], Microsoft's PLINQ for .NET, and Intel's Threading Building Blocks (TBB) for C++ [16]. For "task-parallel" computations such as divide and conquer recursion on arrays and trees, libraries such as FJTask for Java [1] and TBB for C++ [16] provide lightweight tasks that support these computations. However, where these templates do not "fit," programmers must still use low-level tasks. More frameworks, and perhaps even first-class language features, are needed to support a greater range of patterns.

3.2 The Need for Sharing Controls

While emerging data- or task-parallel libraries or frameworks are a valuable step, they lack any guaranteed mechanisms to prevent non-deterministic sharing or even data races. The host language needs to provide much stronger controls on sharing of data between parts of the program, to make sure that the contracts implied by the patterns are observed. It also needs to enable more flexible concurrency patterns, and to empower application teams to create new customized patterns. The fundamental challenge in checking contracts for all such patterns is that the compiler or runtime system needs to be able to "see" and check the dataflow interactions between different parallel computations in the program. In an imperative object-oriented language with aliasing through references and virtual function calls, identifying and analyzing such dataflow is generally extremely difficult. Therefore, we need to control or annotate sharing to make the dataflow more visible, eliminating hidden sharing that can cause surprises and races.

We believe that an important part of the solution is an object oriented *effect system* [13, 5] 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 [2], we are developing a sophisticated effect system that can disambiguate accesses to distinct parts of the same object, as well as distinct objects referred to through data structures such as sets, arrays, and trees. This capability allows a straightforward, *local* type-checker to check if two concurrent tasks might make conflicting accesses to overlapping memory locations. More precisely, it ensures that any two tasks making such conflicting accesses must be ordered explicitly with intervening synchronization. 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. We believe that some form of software speculation [20] is the right solution here. Hardware support for speculation [15] can reduce overhead if it is available. If the overhead becomes unacceptable, we can move to a fail-stop model that aborts the program if a deterministic violation is found [17]. 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 5) 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 [9].

3.3 Encapsulation

In realistic programs, the guarantee of determinism may have to be weakened for parts of the program, even if the overall program behaves deterministically. However, if we can encapsulate the nondeterministic part behind an interface, and guarantee that the interface specification is met, then we can still provide sound guarantees for the rest of the program. This approach is attractive because parallel libraries and frameworks are usually written by expert programmers skilled in low-level parallel programming and performance issues, while application code is likely to be written by a much larger class of programmers with typically less skill in parallel computing.

3.3.1 Encapsulating Nondeterminism

Algorithms that are deterministic *overall* may benefit from "locally non-deterministic" behaviors for higher performance. Operations like associative reductions have deterministic final results but in a high performance implementation, the internal order of operations can be schedule-dependent. Similarly, there are higher-level sequences of commutative operations that produce deterministic final behavior as seen by an external observer, but that may have schedule-dependent internal representations (i.e., memory state). Examples include a sequence of insert operations (or a sequence of remove operations) on a set or on a splay tree, or computing the connected components of a graph.

Programmers should be allowed to specify parts or all of these operations, though implementation and tuning of the most complex and nondeterministic parts should be left to experienced library programmers. All users should be able to define *pure, associative* operators, as in languages like Fortress [18], 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.

3.3.2 Encapsulating Unsoundness

In realistic applications, some parts of the program may be deterministic in fact 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 [5]. 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.

4. Supporting Non-deterministic Behaviors

For some algorithms, non-deterministic behavior is considered acceptable, e.g., branch-and-bound search, Delaunay mesh triangulation [9], and graph clustering. In all these examples, the final result must meet some acceptance criterion but multiple solutions that meet the criterion are permissible. Such behavior can be exploited to yield higher performance than a deterministic parallel algorithm.

There are two constraints that must (and can) be enforced on such code. First, the code should be guaranteed to be free of data races, i.e., non-determinism should strictly be due to timing variations, not incorrect synchronization. For example, *enforcing* atomicity on all shared variable accesses can provide such a guarantee. Second, and more challenging, non-deterministic code in an application should not compromise the ability to reason with determinism for the rest of the application, even though the overall observable behavior will obviously not be deterministic in general.

In DPJ, we are investigating how the latter constraint can be enforced when mixing non-deterministic with deterministic code. 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_sd*, where "sd" stands for "schedule dependent.") The *foreach* construct guarantees determinism (because the iterations are independent), while the *foreach_sd* construct does not guarantee determinism but does guarantee that the iterations are atomic with respect to one another.

Now note the following regarding *foreach_sd*:

- Non-deterministic operations are explicitly marked *sd*.
- Because of the atomicity requirement, the non-deterministic computations can produce timing-dependent results, but cannot have data races.
- The sharing controls ensure that we can still reason deterministically about *foreach* loops that do not enclose a *foreach_sd*. In particular, for a fixed initial memory state before entry into such a *foreach*, the behavior of the *foreach* is deterministic: there are no non-deterministic interactions between writes inside any *foreach_sd* and this *foreach*. If we do nest a *foreach_sd* inside a *foreach*, atomicity of the loop body in the former still guarantees that each of the outer *foreach* iterations is independent — though all behave non-deterministically.

This is an example of controlled nondeterminism: we allow explicit nondeterminism for performance, but we ensure that (1) it must be explicitly requested; and (2) the programmer can split the program syntactically into deterministic and nondeterministic parts, and reason about them separately.

5. A Case for a Language-Based Solution

One common concern with language-based solutions is that the programmer effort required can be onerous, whether to port large existing applications or to develop new ones. We believe that (a) with appropriate design, the benefits more than outweigh these costs in both situations; and (b) strong technical solutions can simplify both tasks. We discuss these briefly in turn.

Although significant initial effort is usually required to use an explicitly parallel language (or library), the benefits can outweigh the costs for several reasons, some of which are often widely missed:

(1) Perhaps most important, the extra effort to learn and use an explicitly parallel language *or* an explicitly parallel library or framework is likely to be dwarfed by the effort required to design or restructure data structures, control flow, and algorithms, and to test and tune the parallel code. A well-designed language that simplifies the latter tasks can more than justify the learning curve, e.g., as with generic programming in sequential programs.

(2) Although O-O effect systems may be conceptually more difficult for programmers to learn and use than APIs for an explicitly parallel library, this extra effort is not wasted. The reasoning required to introduce partitions and effects is exactly the reasoning required to understand the sharing patterns in the code. In fact, the partition 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. By investing the one-time effort to use explicitly parallel constructs (from a language or library), concurrency decisions and sharing patterns are documented explicitly in the code, which simplifies the task of future generations of developers.

(4) Finally, we note that current threads-based languages have woefully inadequate 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 semantics for data races. C++ does not provide any, which 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 would allow this – but determinism for such a language does not require much additional programmer burden, and would be far simpler to reason about.

Some *technical solutions* can further simplify porting existing code and writing new code in a parallel language:

- *Compatibility with a base language*: Adding type extensions to, and maintaining compatibility with, a base language (as DPJ does for Java) greatly simplifies adopting new language mechanisms, for several obvious reasons.
- *Effect inference*: In many cases, the programmer should only need to insert a subset of the partition and effect annotations, and have the rest inferred by the compiler [19].
- *Initial speculation*: The language should provide speculative parallel execution mechanisms, as described in Section 3.2 so that large programs can be initially ported using speculation, without all the effect annotations needed for compile-time checking. The overheads of speculation can then be incrementally tuned away by introducing effect annotations where the benefits justify the effort.
- *Interactive development environment (IDE)*: An IDE for the language 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, interprocedural parallelization techniques (the strengths of compilers) while making it interactive allows programmers to influence the process and avoid the problems of poor or unstable performance (the weaknesses).

6. Summary

Parallel computing is not going to be easier than sequential computing. It is going to be important to apply to this domain the best language design ideas that have been developed in the last decades, in order to increase programmer productivity. In particular, it is essential to use safe languages, and to extend safety to include suitable constraints on interaction between threads: languages that tolerate races must be banned. Additional constraints (regions, effects, interface contracts, etc.) will not only make parallel programming easier, but will also benefit software engineering, in general, by providing more analyzability, and hence facilitating testing and maintenance.

We agree with [10] that unrestricted use of threads in shared memory programming is not a good choice. We have argued in this paper 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 can be more than outweighed by the benefits over the life of an application, and reduced through careful language design and appropriate interactive development tools.

References

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

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

[3] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. Technical report, GA Tech., 2005.

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

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

[6] M. Cintra et al. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA*, 2000.

[7] W. M. Johnston et al. Advances in dataflow programming languages. *ACM Comp. Survs.*, 2004.

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

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

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

[11] W. Liu et al. POSH: a TLS compiler that exploits program structure. In *PPOPP*, 2006.

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

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

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

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

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

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

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

[19] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *J. Funct. Programming*, 1992.

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