Toward a Formal Semantic Framework for Deterministic Parallel Programming

Conference Paper · September 2011

DOI: 10.1007/978-3-642-24100-0_43 · Source: DBLP

Michael L. Scott
University of Rochester

DISC 2011

Toward a Formal Semantic Framework for Deterministic Parallel Programming[⋆]

Li Lu and Michael L. Scott

Computer Science Department, University of Rochester
Rochester, NY 14627-0226 USA
{llu, scott}@cs.rochester.edu

Abstract. Deterministic parallelism has become an increasingly attractive concept: a deterministic parallel program may be easier to construct, debug, understand, and maintain. However, there exist many different definitions of "determinism" for parallel programming. Many existing definitions have not yet been fully formalized, and the relationships among these definitions are still unclear. We argue that formalism is needed, and that history-based semantics—as used, for example, to define the Java and C++ memory models—provides a useful lens through which to view the notion of determinism. As a first step, we suggest several history-based definitions of determinism. We discuss some of their comparative advantages, prove containment relationships among them, and identify programming idioms that ensure them. We also propose directions for future work.

1 Introduction

Determinism, loosely defined, is increasingly touted as a way to simplify the design, verification, testing, and debugging of parallel programs—in effect, as a way to make it easier to understand what a parallel program does. Parallel functional languages have long enjoyed the benefits of determinism [18]. Recent workshops have brought together members of the architecture, programming languages, and systems communities to discuss determinism in more general languages and systems [1, 13]. Determinism has also featured prominently in recent workshops on pedagogy for concurrency [23, 28, 30].

At the very least, determinism suggests that a given parallel program—like a sequential program under most semantic models—should always produce the same output when run with the same input. We believe, however, that it needs to mean more than this—that runs of a deterministic program on a given input should not only produce the same output: they should produce it *in the same way*. By analogy to automata theory, a deterministic Turing machine doesn't just compute a single-valued function: it takes a uniquely determined action at every step along the way.

For real-world parallel programs, computing "in the same way" may be defined in many ways. Depending on context, we may expect that repeated runs of a deterministic

[⋆] This work was supported in part by the US National Science Foundation under grant CCR-0963759.

program will consume (more or less) the same amount of time and space; that they will display the same observable intermediate states to a debugger; that the behavior of distributed replicas will not diverge; that the number of code paths requiring separate testing will be linear in the number of threads (rather than exponential); or that the programmer will be able to straightforwardly predict the impact of source code changes on output or on time and space consumption.

History-based semantics has proven to be one of the most useful ways to model the behavior of parallel programs. Among other things, it has been used to explain the serializability of transactions [27], the linearizability of concurrent data structures [19], and the memory model that determines the values seen by reads in a language like Java [22] or C++ [11]. Memory models typically distinguish between *ordinary* and *synchronizing* accesses, and use these to build a cross-thread partial order among operations of the program as a whole. Recently we have proposed that the various sorts of synchronizing accesses be unified under the single notion of an *atomic action* [15, 29].

Informally, the parallel semantics of a given program on a given input is a set of *abstract executions*. Each execution comprises a set of *thread histories*, each of which in turn comprises a totally ordered sequence of *reads*, *writes*, and other *operations*—notably *external actions* like input and output. The history of a given thread is determined by the program text, the language's (separately specified, typically operational) sequential semantics, the program's input, and the values returned by reads (which may have been set by writes in other threads). An execution is said to be *sequentially consistent* if there exists a total order on reads and writes, consistent with program order in every thread, such that each read returns the value written by the most recent preceding write to the same location. Under relaxed memory models, a program is said to be *data-race free* if the model's partial order covers all pairs of conflicting operations.

An *implementation* maps source programs to sets of low-level *target executions* on some real or virtual machine. The implementation is correct only if, for every target execution, there exists a corresponding abstract execution that performs the same external actions, in the same order. (The extent to which shorter sequences of target-level operations must correspond to operations of the abstract execution is related to, but separate from, the subject of this paper; we do not address it further here.)

In the strictest sense of the word, a deterministic parallel program would be one whose semantics, on any given input, consists of only a single abstract execution, to which any legal target execution would have to correspond. In practice, this definition may prove too restrictive. Suppose, for example, that I have chosen, as a programmer, to "roll my own" shared allocator for objects of some heavily used data type, and that I am willing to ignore the possibility of running out of memory. Suppose further that my allocator keeps free blocks on a simple lock-free stack. Because they access a common top-of-stack pointer, allocation and deallocation operations must synchronize with one another, and will thus be ordered in any given execution. Since I presumably don't care what the order is, I may wish to allow arbitrary executions that differ only in the order realized, while still saying that my program is deterministic.

In general, we suggest, it makes sense to say that a program is deterministic if all of its abstract executions on a given input are *equivalent* in some well-defined sense. A *language* may be said to be deterministic if all its programs are deterministic. An *im-*

plementation may be said to be deterministic (for a given, not-necessarily-deterministic language) if all the target executions of a given program on a given input correspond to abstract executions that are mutually equivalent. For all these purposes, ***the definition of determinism amounts to an equivalence relation on abstract executions.***

We contend that history-based semantics provides a valuable lens through which to view determinism. By specifying semantics in terms of executions, we capture the notion of "computing in the same way"—not just computing the same result. We also accommodate programs (e.g., servers) that are not intended to terminate—executions need not be finite. By separating semantics (source-to-abstract-execution) from implementation (source-to-target-execution), we fix the level of abstraction at which determinism is expected, and, with an appropriate definition of "equivalence," we codify what determinism *means* at that level.

For examples like the memory allocator mentioned above, history-based semantics highlights the importance of language definition. If my favorite memory management mechanism were a built-in facility, with no implied ordering among allocation and deallocation operations of different objects, then a program containing uses of that facility might still have a single abstract execution. Other potential sources of nondeterminism that might be hidden inside the language definition include parallel iterators, bag-of-task work queues, and container data types (sets, bags, mappings). Whether *all* such sources can reasonably be shifted from semantics to implementation remains an open question (but we doubt it).

In a similar vein, an implementation may be deterministic for only a subset of some standard programming language—i.e., for a smaller language. MIT's Kendo system, for example, provides determinism for only those $\langle$program, input$\rangle$ pairs that are data-race free—a property the authors call *weak determinism* [26].

From an implementation perspective, history-based semantics differentiates between things that are required to be deterministic and things that an implementation might *choose* to make deterministic. This perspective draws a sharp distinction between projects like DPJ [10], Prometheus [3], and CnC [12], which can be seen as constraining the set of abstract executions, and projects like Kendo, DMP [16], CoreDet [5], and Grace [8], which can provide deterministic execution even for (some) pthread-ed programs in C. (Additional projects, such as Rerun [20], DeLorean [24], and DoublePlay [31], are intended to provide deterministic *replay* of a program whose initial run is more arbitrary.)

If we assume that an implementation is correct, history-based semantics identifies the set of executions that an application-level test harness might aspire to cover. For purposes of debugging, it also bounds the set of global states that might be visible at a breakpoint—namely, those that correspond to a consistent cut through the partial order of a legal abstract execution.

We believe the pursuit of deterministic parallel programming will benefit from careful formalization in history-based semantics. Toward that end, we present a basic system model in Section 2, followed by several possible definitions of equivalence for abstract executions in Section 3. We discuss the comparative advantages of these definitions in Section 4. We also prove containment relationships among the definitions, and identify programming idioms that ensure them. Many other definitions of equivalence are possible, and many additional questions seem worth pursuing in future work; we list a few of these in Section 5.

2 System Model

In a manner consistent with standard practice and with our recent work on atomicity-based semantics [15], we define an *execution* of a program P, written in a language $\mathcal{L}$, to be a 3-tuple $E_{P,\mathcal{L}} : (\mathit{OP}, <_p, <_s)$, where *OP* is a set of *operations*, and $<_p$ (*program order*) and $<_s$ (*synchronization order*) are irreflexive partial orders on *OP*. When we can do so without confusion, we omit P and $\mathcal{L}$ from our notation.

Each operation in *OP* takes one of six forms: (read, name, val, tid, uid), (write, name, val, tid, uid), (input, val, tid, uid), (output, val, tid, uid), (begin_atomic, tid, uid), or (end_atomic, tid, uid). In each of these, tid identifies the executing thread. Uid is an arbitrary unique identifier; it serves to make every operation distinct and to allow the set *OP* to contain multiple operations that are otherwise identical. In read and write operations, name identifies a program variable; in read, write, input, and output operations, val identifies a value read from a variable or from the program's input, or written to a variable or to the program's output. The domains from which thread ids, variable names, and values are chosen are defined by the semantics of $\mathcal{L}$. These domains are assumed to be countable, but not necessarily finite.

Program order, $<_p$, is a union of disjoint total orders, one per thread. Specifically, if o_1 = (…, t_1, u_1) and o_2 = (…, t_2, u_2) are distinct operations of the same execution (i.e., $u_1 \neq u_2$), then $(t_1 = t_2) \rightarrow (o_1 <_p o_2 \veebar o_2 <_p o_1)$ and $(t_1 \neq t_2) \rightarrow (o_1 \not<_p o_2 \wedge o_2 \not<_p o_1)$. (Here $\veebar$ is *exclusive or*.)

For any given thread t_i, $\mathit{OP}|_{t_i}$ or $E|_{t_i}$ represents t_i's *thread history*—its (totally ordered) sequence of operations. We use $\mathit{OP}|_s$ or $E|_s$ to represent an execution's *synchronization operations*: begin_atomic, end_atomic, input, and output. We use $\mathit{OP}|_e$ or $E|_e$ to represent the execution's *external operations*: input and output. Clearly $\mathit{OP}|_e \subset \mathit{OP}|_s$. (We assume that I/O races are always unacceptable.)

Synchronization order, $<_s$, is a total order on $\mathit{OP}|_s$. It does not relate reads and writes, but it is consistent with program order. That is, for o_1 = (…, t_1, u_1) and o_2 = (…, t_2, u_2), $(o_1 <_s o_2 \wedge t_1 = t_2) \rightarrow (o_1 <_p o_2)$.

For convenience, we define v_{in} and v_{out}, for a given execution E, to be the execution's *input* and *output vectors*—the (possibly infinite) sequences of values contained, in order of $<_s$, in E's input and output operations, respectively. For any given execution E, we use $\mathit{ext}(E)$ to represent the pair $\langle v_{\mathit{in}}, v_{\mathit{out}} \rangle$.

We use begin_atomic and end_atomic operations in our model to capture the synchronization operations of $\mathcal{L}$, whatever those may be—thread fork and join, lock acquire and release, monitor entry and exit, volatile variable read and write, etc. For this reason, we require that begin_atomic and end_atomic operations appear in disjoint, unnested pairs, and never bracket input or output operations. That is, for every b = (begin_atomic, t, u_1) there exists an e = (end_atomic, t, u_2) such that $b <_s e$ and $\forall m \in \mathit{OP}|_s \smallsetminus \{b, e\},\ m <_s b \vee e <_s m$; likewise for every e = (end_atomic, t, u_2) there exists a b = (begin_atomic, t, u_1) such that $b <_s e$ and $\forall m \in \mathit{OP}|_s \smallsetminus \{b, e\},\ m <_s b \vee e <_s m$. We use $\mathit{OP}|_a$ or $E|_a$ to represent the execution's *atomic actions*: the union of $E|_e$ and the set of minimal sequences of operations in each thread beginning with begin_atomic and ending with end_atomic.

Continuing with standard practice, we assume that the semantics of $\mathcal{L}$ defines, for any given execution, a *synchronizes-with order*, $<_{sw}$, that is a subset of $<_s$—that is, a partial order on $OP|_s$. In a lock-based language, for example, the *release* method for lock L might be modeled as (begin_atomic, t_1, u_1), (write, L, 0, t_1, u_2), (end_atomic, t_1, u_3);[1] an *acquire* might be (begin_atomic, t_2, u_4), (read, L, 0, t_2, u_5), (write, L, 1, t_2, u_6), (end_atomic, t_2, u_7). If operation u_3 precedes operation u_4 in $<_s$, we might require that it do so in $<_{sw}$ as well (since they operate on the same lock), but operations used to model methods of different locks might be unrelated by $<_{sw}$.

Given $<_{sw}$, we define *happens-before order*, $<_{hb}$, to be the irreflexive transitive closure of $<_p$ and $<_{sw}$. Finally, we assume that $\mathcal{L}$ defines, given $<_{hb}$, a *reads-see-writes* function W that specifies, for any given read operation r, the set of write operations $\{w_i\}$ whose values r is permitted to return. In most languages, r will be allowed to see w if w is the most recent previous write to the same variable on some happens-before path. In some languages (e.g., Java), r may be allowed to see w if the two operations are incomparable under $<_{hb}$. Languages may also differ as to whether atomic actions are *strongly atomic*—that is, whether nonatomic reads can see inside them, or nonatomic writes be seen within them [9] [15, TR version appendix].

In any consistent cut across $<_{hb}$, we define the *program state* to be (1) the prefixes of v_{in} and v_{out} that have been input and output prior to the cut, and (2) the most recent values written to the program's variables according to $<_{hb}$. If a variable has not yet been written, its value is undefined ($\perp$); in a program with a data race, the most recent write may not be unique, in which case the variable's value is indeterminate.

Two operations (read or write) *conflict* if they access the same variable and at least one of them writes it. An execution is *data-race free* if all conflicting operations are ordered by $<_{hb}$.

In this paper, we consider only *well-formed* executions. An execution E is well formed if and only if it satisfies the following three requirements.

Adherence to per-thread semantics: Given the code for thread t and the values returned by read and input operations, $\mathcal{L}$'s (independently specified) sequential semantics determine the set of legal histories for t; $E|_t$ must be among them. Moreover, begin_atomic and end_atomic operations in $E|_t$ must occur in disjoint matched pairs, with only read or write operations between them (as ordered by $<_p$).

Consistent ordering: For all t, operations of $E|_t$ are totally ordered by $<_p$. Operations with different tids are unordered by $<_p$. $E|_s$ is totally ordered by $<_s$, which is consistent with $<_p$. Reads and writes do not participate in $<_s$. Paired begin_atomic and end_atomic operations are contiguous in $<_s$.

Adherence to memory model: All values read are permitted by W, the reads-see-writes function induced by $<_p$, $<_s$, $<_{sw}$, and $<_{hb}$, according to $\mathcal{L}$'s semantics.

[1] The release sequence must be bracketed with begin_atomic…end_atomic, even though there is only one operation inside, in order to induce cross-thread ordering.

3 Example Definitions of Equivalence

In this section we suggest several possible definitions of equivalence for abstract executions. Two—*Singleton* and *ExternalEvents*—are intended to be extreme cases: the strictest and loosest definitions that strike us as plausible. Another—*FinalState*—is similar to *ExternalEvents*, restricted to programs that terminate. The other two—*Dataflow* and *SyncOrder*—are two of many possible in-between options.

Singleton. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if they differ only in the* uid*s of their operations; that is, there exists a one-one mapping (bijection) between OP_1 and OP_2 that preserves $<_p$, $<_s$, and the content other than* uid *in every operation.*

Singleton uses the strictest possible definition of determinism: there must be only one possible execution for a given program and input.

Dataflow. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if $ext(E_1) = ext(E_2)$ and there is a one-one mapping between OP_1 and OP_2 that preserves (1) the content other than* tid *and* uid *in every operation, and (2) the reads-see-writes function W induced, under $\mathcal{L}$'s semantics, by $<_p$, $<_s$, $<_{sw}$, and $<_{hb}$.*

Informally, *Dataflow* requires that reads see the same writes in both executions, and that the values in both reads and writes (including the input and output operations that "read" and "write" elements of v_{in} and v_{out}) be the same in both executions. Note that we do not require that the bijection preserve $<_p$ or $<_s$, nor do we require that the executions be data-race free.

SyncOrder. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if there is a one-one mapping between $E_1|_a$ and $E_2|_a$ that preserves (1) $<_s$, and (2) the content other than* uid *in every synchronization operation and in every* read *or* write *within an atomic action.*

SyncOrder requires that there be a fixed pattern of synchronization among threads in E_1 and E_2, with atomic actions reading and writing the same values in the same variables. Note that if executions are data-race free (something that *SyncOrder* does not require), then they are also sequentially consistent [2], so $E_1 \equiv_{\text{SyncOrder}} E_2 \wedge E_1, E_2 \in DRF \rightarrow E_1 \equiv_{\text{Dataflow}} E_2$.

ExternalEvents. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if $ext(E_1) = ext(E_2)$.*

ExternalEvents is the most widely accepted language-level definition of determinism. It guarantees that abstract executions on the same input look "the same" from the perspective of the outside world.

FinalState. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if they both terminate and their program states at termination (values of variables and of v_{in} and v_{out}) are the same.*

Like *ExternalEvents*, *FinalState* says nothing about how E_1 and E_2 compute. It requires only that final values be the same. Unlike *ExternalEvents*, *FinalState* requires agreement on variables *other* than output.

4 Discussion

Singleton is the strictest definition of equivalence, and thus of determinism. It is a common notion in the literature—corresponding, for example, to what Emrath and Padua called "internally determinate" [17] and Netzer and Miller "internally deterministic" [25]. It requires a single execution for any given source program and input. Interestingly, while we have not insisted that such executions be sequentially consistent, they seem likely to be so in practice: a language that admits non-sequentially consistent executions (e.g., via data races) seems likely (unless it is designed in some highly artificial way) to admit multiple executions for some ⟨program, input⟩ pairs.

By requiring abstract executions to be identical in every detail, *Singleton* rules out "benign" differences of any kind. It may therefore preclude a variety of language features and programming idioms that users might still like to think of as "deterministic."

Dataflow relaxes *Singleton* by loosening the requirements on control flow. Equivalent executions must still have the same operation sets (ignoring tid and uid), but the synchronization and program orders can be different, so long as values flow from the same writes to the same reads. In the literature, *Dataflow* is essentially equivalent to Karp and Miller's 1966 definition of "determinacy" [21], which was based on a dataflow model of computation. Intuitively, *Dataflow* can be thought of as an attempt to accommodate programming languages and idioms in which the work of the program is fixed from run to run, but may be partitioned and allocated differently among the program's threads.

SyncOrder also relaxes *Singleton*, but by admitting benign changes in data flow, rather than control flow. Specifically, *SyncOrder* requires equivalent executions to contain the exact same synchronization operations, executed by the same threads in the same order. It does *not* require that a read see the same write in both executions, but it does require that any disagreement have no effect on synchronization order (including output).

ExternalEvents is also a common notion in the literature. It corresponds to what Emrath and Padua called "externally determinate" [17] and, more recently, to the working definition of determinism adopted by Bocchino et al. [10]. The appeal of the definition lies in its generality. If output is all one cares about, *ExternalEvents* affords the language designer and implementor maximum flexibility. From a practical perspective, knowing that a parallel program will always generate the same output from the same input, regardless of scheduling idiosyncrasies, is a major step forward from the status quo. For users with a strong interest in predictable performance and resource usage, debugability, and maintainability, however, *ExternalEvents* may not be enough.

FinalState is essentially a variant of *ExternalEvents* restricted to programs that terminate (and whose internal variables end up with the same values in every execution). It corresponds to what Netzer and Miller called "externally deterministic" [25].

In the remainder of this section, we explore additional ramifications of our example definitions. In Section 4.1 we formalize containment properties: which definitions of equivalence, if they hold between a given pair of executions, imply which other definitions? Which definitions are incomparable? In Section 4.2 we identify programming languages and idioms that illustrate these containments. Finally, in Sections 4.3 and 4.4, we consider the practical issues of repetitive debugging and of deterministic implementation for nondeterministic languages.

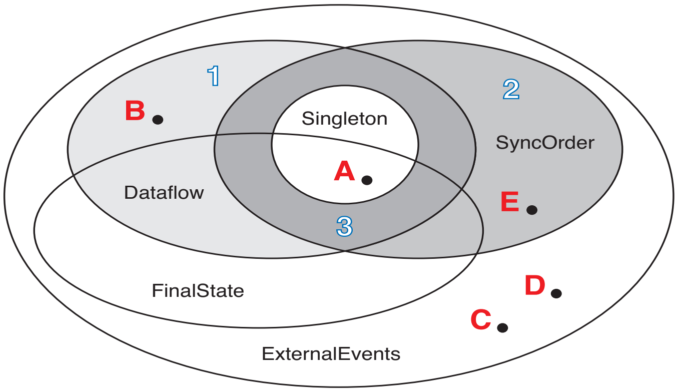

Fig. 1. Containment relationships among definitions of determinism, or, equivalently, abstract execution equivalence. Names of equivalence definitions correspond to ovals. Outlined numbers label the light, medium, and dark shaded regions. Bold letters show the locations of programming idioms discussed in Section 4.2.

4.1 Containment Properties

Figure 1 posits containment relationships among the definitions of determinism given in Section 3. The space as a whole is populated by sets $\{X_i\}$ of executions of some given program on a given input, with some given semantics. If region S is contained in region L, then all executions that are equivalent under definition S are equivalent under definition L as well; that is, S is a stricter and L a looser definition. (The regions can also be thought of as containing languages or executions: a language [execution] is in region R if for every program and input, all abstract executions generated by the language semantics [or corresponding to target executions generated by the implementation] are equivalent under definition R.) We justify the illustrated relationships as follows.

Theorem 1. Singleton *is contained in* Dataflow*,* SyncOrder*, and* ExternalEvents*.*

Proof: Suppose E_1 and E_2 are arbitrary equivalent executions under *Singleton*. By definition, E_1 and E_2 are identical in every respect other than the uids of their operations. None of the definitions of *Dataflow*, *SyncOrder*, or *ExternalEvents* speaks to uids. Each requires certain other portions of E_1 and E_2 (or entities derived from them) to be the same; *Singleton* trivially ensures this. □

Theorem 2. Singleton*,* Dataflow*,* SyncOrder*, and* FinalState *are all contained in* ExternalEvents*.*

Proof: For *Singleton*, this is proved in Theorem 1. For *Dataflow*, it follows from the definition: if E_1 and E_2 are *Dataflow* equivalent, then $\mathit{ext}(E_1) = \mathit{ext}(E_2)$.

For *SyncOrder*, suppose $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are arbitrary equivalent executions under *SyncOrder*. This means there is a bijection between OP_1 and OP_2 that preserves (among other things) both $<_s$ and the content other than uid in each input and output operation. Since input and output operations are totally ordered by $<_s$, and since v_{in} and v_{out} are defined to be the values in an execution's input and output operations, in order of $<_s$, we have $\mathit{ext}(E_1) = \mathit{ext}(E_2)$.

For *FinalState*, suppose E_1 and E_2 are equivalent under *FinalState*. Then E_1 and E_2 both terminate, and with the same state. Their input and output vectors, included in their terminating states, must therefore be the same: $\mathit{ext}(E_1) = \mathit{ext}(E_2)$. □

Theorem 3. *There are sets of executions that are equivalent under* Dataflow *but not under* SyncOrder.

Proof: This is the light gray region, labeled "1" in Figure 1. It corresponds to programs with benign synchronization races. Consider a program in which two threads each increment a variable under protection of a lock: acquire(L); x++; release(L). Under plausible semantics, one possible execution looks as follows (ignoring uids), where $<_p$ orders the operations of each thread as shown, and $<_s$ orders the atomic actions of thread 1 before those of thread 2:

```
(begin_atomic, $t_1$) (read, L, 0, $t_1$) (write, L, 1, $t_1$) (end_atomic, $t_1$)
    (read, x, 0, $t_1$) (write, x, 1, $t_1$) (begin_atomic, $t_1$) (write, L, 0, $t_1$) (end_atomic $t_1$)
(begin_atomic, $t_2$) (read, L, 0, $t_2$) (write, L, 1, $t_2$) (end_atomic, $t_2$)
    (read, x, 1, $t_2$) (write, x, 2, $t_2$) (begin_atomic, $t_2$) (write, L, 0, $t_2$) (end_atomic $t_2$)
```

Call this execution E_1. Another execution (call it E_2) looks the same, except that the tids in various operations are reversed: thread 2 changes x from 0 to 1; thread 1 changes it from 1 to 2. For *Dataflow*, the obvious bijection swaps the tids, and the executions are equivalent. For *SyncOrder*, there is clearly no bijection that preserves both synchronization order and the tids in each begin_atomic and end_atomic operation. □

Theorem 4. *There are sets of executions that are equivalent under* SyncOrder *but not under* Dataflow.

Proof: This is the medium gray region, labeled "2" in Figure 1. It corresponds to programs with benign data races. An example is shown at right. This is a racy program: neither the write nor the read of flag in t2 is ordered by $<_{hb}$ with the write in t1. Two abstract executions, E_1 and E_2 (not shown), may thus have different data flow: in E_1, the read of flag in t2 returns the value 1, while in E_2 it returns the value 2. However, these two executions have the same synchronization order: in both, only the two outputs are ordered by $<_s$, and they are ordered the same in both executions. Thus $E_1 \equiv_{\mathit{SyncOrder}} E2$ but $E_1 \not\equiv_{\mathit{Dataflow}} E_2$. □

```
        Initially flag == 0

t1:                t2:
    flag = 1           flag = 2
                       if (flag > 0)
                           print "flag > 0"
                           print "end"
```

Theorem 5. Singleton*,* Dataflow*, and* SyncOrder *all have nontrivial intersections with* FinalState*.*

Proof: *Singleton*, *Dataflow*, and *SyncOrder* clearly all contain sets of terminating executions that have the same final state. However, equivalent executions in *Singleton*, *Dataflow* or *SyncOrder* do not necessarily terminate. Suppose E_1 and E_2 are arbitrary equivalent executions under *Singleton*, *Dataflow* or *SyncOrder*, and also under *FinalState*. We can make both executions nonterminating by adding an additional thread to each that executes an infinite but harmless loop (it might, for example, read an otherwise unused variable over and over). The modified executions will no longer be in *FinalState* since they do not terminate, but they will still be in *Singleton*, *Dataflow*, or *SyncOrder*, since the loop will neither race nor synchronize with any other part of the program.

Conversely, there are executions that have different data flows or synchronization orders, but terminate with the same state. Examples in *FinalState* $\cap$ (*Dataflow* $\setminus$ *SyncOrder*) and *FinalState* $\cap$ (*SyncOrder* $\setminus$ *Dataflow*) appear in the proofs of Theorems 3 and 4, respectively, and an example for *FinalState* $\setminus$ (*SyncOrder* $\cup$ *Dataflow*) is easy to construct (imagine, for example, a program that has chaotic data flow and synchronization, but eventually writes a zero to every program variable before terminating). □

4.2 Programming Languages and Idioms

While equivalence relations and their relationships, seen from a theoretical perspective, may be interesting in their own right, they probably need to correspond to some intuitively appealing programming language or idiom in order to be of practical interest. As illustrations, we describe five programming idioms in this section, corresponding to the dots labeled A, B, C, D, and E in Figure 1.

Independent Split-Merge (Point A $\in$ *Singleton* in Figure 1.) Consider a language providing parallel iterators or cobegin, with the requirement (enforced through the type system or run-time checks) that concurrent tasks access disjoint sets of variables. If every task is modeled as a separate thread, then there will be no synchronization or data races, and the execution of a given program on a given input will be uniquely determined.

Bag of Independent Tasks (Point B $\in$ *Dataflow* $\setminus$ *SyncOrder* in Figure 1.) Consider a programming idiom in which "worker" threads dynamically self-schedule independent tasks from a shared bag. The resulting executions will have isomorphic data flow (all that will vary is the tids in the corresponding reads and writes), but their synchronization orders will vary with the order in which they access the bag of tasks.

Significantly, this idiom remains in *Dataflow* $\setminus$ *SyncOrder* even if we require that tasks be added to the bag in groups, and all of them completed before any new tasks can be added. One might consider such a restricted model to be an alternative characterization of the Independent Split-Merge idiom, but we prefer to consider it a separate language—one in which the maximum degree of concurrency in the abstract execution is limited to the number of worker threads.

One might also expect that a program with deterministic sequential semantics, no data races, and no synchronization races would have only a single abstract execution for a given input—that is, that *Dataflow* $\cap$ *SyncOrder* would equal *Singleton*. We speculate, however, that there may be cases—e.g., uses of rand()—that are easiest to model with more than one execution (i.e., with classically nondeterministic sequential semantics), but that we might still wish to think of as "deterministic parallel programming." We have left a region in Figure 1 (the dark gray area labeled "3") to suggest this possibility.

Parallel Iterator with Reduction (Point C $\in$ *ExternalEvents* $\setminus$ (*Dataflow* $\cup$ *SyncOrder*) in Figure 1.) Consider a language with explicit support for reduction by a commutative, associative function. The order in which such a function is applied to a set of operands need not be fixed, leading to executions with different synchronization orders and data flows, but only a single result. It seems plausible that we might wish to call programs in such a language "deterministic."

Parallel Atomic Commutative Methods (Point D $\in$ *ExternalEvents* $\setminus$ (*Dataflow* $\cup$ *SyncOrder*) in Figure 1.) In the split-merge and bag-of-tasks idioms above, we required that parallel tasks be independent. We may relax this requirement by allowing tasks to call methods of some shared object O, so long as the calls are atomic and (semantically) commutative. The memory allocator mentioned in Section 1 is an example of this idiom, as long as we ignore the possibility of running out of memory. Another example would be a memoization table that caches outputs of some expensive function.

If a program contains atomic, commutative method calls in otherwise independent tasks, the synchronization order for these calls may be different in different runs of the program on the same input. Data flow may also be different, because the internal state of the shared object may change with the synchronization order. Even the final state may be different, since commutativity is defined at a level of abstraction above that of individual variables. A given finite sequence of calls is guaranteed to lead to the same output, however, regardless of permutation, because the calls are atomic and commutative.

Chaotic Relaxation (Point E $\in$ *SyncOrder* $\setminus$ *Dataflow* in Figure 1.) Certain spatially partitioned, iterative computations can be proven to converge even if iterations are unsynchronized, allowing local computations to sometimes work with stale data [14]. Execution typically halts once all threads have verified that the error in their local values is less than some threshold ϵ.

Imagine a language that is specially designed for programs of this kind. Programmers can specify an ϵ for the convergence condition, then design an iterative algorithm for an array of data. Different executions may have different data flows, because the program is full of data races. For chaotic relaxation, however, these data races do not change the limit toward which the computation converges. If final results are rounded to a level of significance determined by ϵ before being output, the results will be deterministic despite the uncertainty of data flow. And as long as the output operations (which constitute the only synchronization in the program) are strictly ordered, the program will be deterministic according to *SyncOrder*.

4.3 Repetitive Debugging

One of the principal goals of deterministic parallel programming is to facilitate repetitive debugging. The definitions in Section 3 vary significantly in the extent to which they achieve this goal.

In a *Singleton* system, a debugger that works at the level of abstract executions will be guaranteed, at any breakpoint, to see a state that corresponds to some consistent cut across the happens-before order of the single execution. This guarantee facilitates repetitive debugging, though it may not make it trivial: a breakpoint in one thread may participate in an arbitrary number of cross-thread consistent cuts; global state is not uniquely determined by the state of a single thread. If we allow all other threads to continue running, however, until they wait for a stopped thread or hit a breakpoint of their own, then global state will be deterministic. Moreover (assuming a relatively fine-grain correspondence between target and abstract executions), monitored variables' values will change deterministically, since *Singleton* requires all runs of a program on a given input to correspond to the same abstract execution. This should simplify both debugging and program understanding.

Under *Dataflow*, monitored variables will still change values deterministically, but two executions may not reach the same global state when a breakpoint is triggered, even if threads are allowed to "coast to a stop." A program state encountered in one execution may never arise in an equivalent execution.

Consider the code fragment shown at right (written in a hypothetical language). Assume that f() is known to be a pure function, and that the code fragment is embedded in a program that creates two worker threads for the purpose of executing parallel iterators. In one plausible semantics, the elements of a parallel iteration space are placed in a synchronous queue, from which workers dequeue them atomically.

```
parfor i in [0, 1]
    A[i] = f(i)
print A[0]
print A[1]
```

Even in this trivial example, there are four possible executions, in which dequeue operations in threads 0 and 1, respectively, return $\{0, \bot\}$ and $\{1, \bot\}$, $\{1, \bot\}$ and $\{0, \bot\}$, $\{0, 1, \bot\}$ and $\{\bot\}$, or $\{\bot\}$ and $\{0, 1, \bot\}$. These executions will contain exactly the same operations, except for thread ids. They will have different program and synchronization orders. *Dataflow* will say they are equivalent; *Singleton* will say they are not. If we insist that our programming model be deterministic, *Dataflow* will clearly afford the programmer significantly greater expressive power. On the other hand, a breakpoint inserted at the call to f() in thread 0 may see very different global states in different executions; this could cause significant confusion.

Like *Dataflow*, *SyncOrder* fails to guarantee deterministic global state at breakpoints, but we hypothesize that the variability will be significantly milder in practice: benign data flow changes, which do not impact synchronization or program output, seem much less potentially disruptive than benign synchronization races, which can change the allocation of work among threads.

ExternalEvents and *FinalState*, for their part, offer significant flexibility to the language designer and implementor, but with potentially arbitrary differences in internal behavior across program runs. This would seem to make them problematic for repetitive debugging.

4.4 Deterministic Implementations

Generally speaking, given a deterministic parallel programming language, it should be straightforward to construct an implementation that achieves most of the concurrency that a programmer might expect on a given machine. This expectation is essentially an issue of liveness, and may be difficult to formalize, but the intuition is clear: if the language is capable of expressing only deterministic programs, then an implementation that captures the concurrency explicit in such programs will remain deterministic. In Independent Split-Merge programs, for example, an implementation is assured that synchronization ($<_s$) edges enter a task only at the beginning, and leave it only at the end, so scheduling decisions within a split-merge group can never violate happens-before.

The more interesting question is: in a language that admits nonequivalent abstract executions, how hard is it likely to be (how much run-time cost are we likely to incur) to construct an implementation that achieves a high degree of concurrency (scalability) while still guaranteeing that all target executions will correspond to equivalent abstract executions? Here the answer may depend on just how much nondeterminism the language itself allows. As noted in Section 1, Kendo [26] provides determinism (of roughly the *SyncOrder* variety) only for programs written in the data-race-free subset of C. Specifically, it resolves each synchronization race deterministically, given that deterministic resolution of prior synchronization races and the lack of data races uniquely determines program order in each thread, up to the next synchronization operation. While still too slow for production use (reported overheads are on the order of 1.6×), this is fast enough for convenient repetitive debugging.

For programs with data races, there is no known way to achieve any of our definitions of deterministic implementation without special-purpose hardware or very high worst-case overhead (one can, of course, serialize the execution—we count that as "very high overhead"). CoreDet [5], dOS [6], and Determinator [4] all achieve roughly *Singleton* semantics on conventional hardware, by executing threads in coarse-grain lockstep "epochs," with memory updates applied in deterministic order at epoch boundaries. Unfortunately, all impose common-case overheads of roughly 10×, making them unsuitable for production use and undesirable for debugging. Recent work on the DoublePlay system [31] suggests that it may be possible to execute arbitrary programs deterministically while limiting overhead to a relatively modest amount (comparable to that of Kendo) for executions whose behavior does not depend on data races. (For a brief survey of implementation techniques for deterministic execution, see the recent paper by Bergan et al. [7].)

5 Conclusions and Future Work

Deterministic parallel programming needs a formal definition (or set of definitions). Without this, we will really have no way to tell whether the implementation of a deterministic language is correct. History-based semantics seems like an excellent framework in which to create definitions, for all the reasons mentioned in Section 1. We see a wide range of topics for future research:

- Existing projects need to be placed within the framework. What are their definitions of execution equivalence?
- Additional definitions need to be considered, evaluated, and connected to the languages and programming idioms that might ensure them.
- We need to accommodate condition synchronization, and source-level spinning in particular. Even in *Singleton*, executions that differ only in the number of times a thread checks a condition before finding it to be true should almost certainly be considered to be equivalent.
- We need to decide how to handle operations (e.g., rand()) that compromise the determinism of sequential semantics. Should these in fact be violations? Should they be considered inputs? Should they perhaps be permitted only if they do not alter output?
- Languages embodying the more attractive definitions of determinism should be carefully implemented, and any losses in expressiveness or scalability relative to other definitions carefully assessed.

- We need to examine more thoroughly the issues involved in deterministic implementation of nondeterministic languages.

This final issue seems intriguing. While it may be easy to build an implementation in which all realizable target executions (of a given program and input) correspond to the same abstract execution, such an implementation may be unacceptably slow (e.g., sequential). It may be substantially more difficult to build an implementation that improves performance by exploiting the freedom to realize target executions corresponding to different but nonetheless equivalent abstract executions. This is in essence a question of liveness rather than safety, and it raises a host of new questions: Which executions can be realized by a given implementation? Are certain executions fundamentally more difficult to realize (without also realizing other executions that aren't safe)? What is the appropriate boundary between language- and implementation-level determinism? Progress on these questions, we believe, could significantly enhance the convenience, correctness, and performance of programming in the multicore era.

References

[1] V. Adve, L. Ceze, and B. Ford, organizers. 2nd Workshop on Determinism and Correctness in Parallel Programming. Newport Beach, CA, Mar. 2011.

[2] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. *17th Intl. Symp. on Computer Architecture*, Seattle, WA, May 1990.

[3] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization Sets: A Dynamic Dependence-Based Parallel Execution Model. *14th ACM Symp. on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. *9th Symp. on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. *15th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Pittsburgh, PA, Mar. 2010.

[6] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. *9th Symp. on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010.

[7] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The Deterministic Execution Hammer: How Well Does It Actually Pound Nails? *2nd Workshop on Determinism and Correctness in Parallel Programming*, Newport Beach, CA, Mar. 2011.

[8] E. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. *OOPSLA 2009 Conf. Proc.*, Orlando, FL, Oct. 2009.

[9] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. *4th Workshop on Duplicating, Deconstructing, and Debunking*, Madison, WI, June 2005.

[10] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. *OOPSLA 2009 Conf. Proc.*, Orlando, FL, Oct. 2009.

[11] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. *SIGPLAN 2008 Conf. on Programming Language Design and Implementation*, Tucson, AZ, June 2008.

[12] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent Collections. *Journal of Scientific Programming*, 18(3–4), Aug. 2010.

[13] L. Ceze and V. Adve, organizers. Workshop on Deterministic Multiprocessing and Parallel Programming. Seattle, WA, Nov.-Dec. 2009.

[14] C. Chazan and W. Miranker. Chaotic Relaxation. *Linear Algebra and Its Applications*, 2:199-222, 1969.

[15] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the Foundation of a Memory Consistency Model. *24th Intl. Symp. on Distributed Computing*, Cambridge, MA, Sept. 2010. Previously Computer Science TR 959, Univ. of Rochester, July 2010.

[16] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. *14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar. 2009.

[17] P. A. Emrath and D. A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, May 1988.

[18] R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501-538, Oct. 1985.

[19] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463-492, July 1990.

[20] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. *35th Intl. Symp. on Computer Architecture*, Beijing, China, June 2008.

[21] R. M. Karp and R. E. Miller. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390-1411, Nov. 1966.

[22] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. *32nd ACM Symp. on Principles of Programming Languages*, Long Beach, CA, Jan. 2005.

[23] S. Midkiff, V. Pai, and D. Bennett, organizers. Workshop on Integrating Parallelism Throughout the Undergraduate Computing Curriculum. San Antonio, TX, Feb. 2011.

[24] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. *35th Intl. Symp. on Computer Architecture*, Beijing, China, June 2008.

[25] R. H. B. Netzer and B. P. Miller. What Are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74-88, Mar. 1992.

[26] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. *14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar. 2009.

[27] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631-653, Oct. 1979.

[28] N. Shavit, organizer. Workshop on Directions in Multicore Programming Education. Washington, DC, Mar. 2009.

[29] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. *12th Intl. Conf. on Principles of Distributed Systems*, Luxor, Egypt, Dec. 2008.

[30] G. L. Steele, Jr. and V. A. Saraswat, organizers. Workshop on Curricula for Concurrency. Orlando, FL, Oct. 2009.

[31] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.