Taming the Parallel Effect Zoo
Extensible Deterministic Parallelism with LVish

Lindsey Kuper  Aaron Todd  Sam Tobin-Hochstadt  Ryan R. Newton
Indiana University
{lkuper, toddaar, samth, rrnewton}@cs.indiana.edu

Abstract
A fundamental challenge of parallel programming is to ensure that the observable outcome of a program remains deterministic in spite of parallel execution. Language-level enforcement of determinism is possible, but existing deterministic-by-construction parallel programming models tend to lack features that would make them applicable to a broad range of problems. Moreover, they lack extensibility: it is difficult to add or change language features without breaking the determinism guarantee.

The recently proposed LVar-based programming model, and the accompanying LVish Haskell library, took a step toward broadly-applicable guaranteed-deterministic parallel programming. The LVar model allows communication through shared monotonic data structures to which information can only be added, never removed, and for which the order in which information is added is not observable. LVish provides a \texttt{Par} monad for parallel computation that encapsulates determinism-preserving effects while allowing a more flexible form of communication between parallel tasks than previous guaranteed-deterministic models provided.

While applying LVar-based programming to real problems using LVish, we have identified and implemented three capabilities that extend its reach: inflationary updates other than least-upper-bound writes; transitive task cancellation; and parallel mutation of non-overlapping memory locations. The unifying abstraction we use to add these capabilities to LVish—which without suffering added complexity or cost in the core LVish implementation, or compromising determinism—is a form of \texttt{monad transformer}, extended to handle the \texttt{Par} monad. With our extensions, LVish provides the most broadly applicable guaranteed-deterministic parallel programming interface available to date. We demonstrate the viability of our approach both with traditional parallel benchmarks and with results from a real-world case study: a bioinformatics application that we parallelized using our extended version of LVish.

Categories and Subject Descriptors: D.3.3 [Language Constructs and Features]: Concurrent programming structures; D.1.3 [ Concurrent Programming]; Parallel programming; D.3.2 [Language Classifications]: Concurrent, distributed, and parallel languages

Keywords Deterministic parallelism

1. Expressive Deterministic Parallelism
For parallelism to become the norm it must become easier. One great stride in this direction would be to provide deterministic-by-default parallel languages that are broadly applicable, available, and practical—enabling more software to avoid heisenbugs by construction. Historically, language-level enforcement of determinism can be found in languages based on synchronous dataflow [8], data-parallel languages [5, 20], and languages with advanced permissions systems that prevent data races [5]. However, virtually all practical parallel programs are written in traditional languages (e.g., C++, Java, Fortran), rather than in these more restricted languages that could guarantee determinism.

It must be that the benefits of determinism do not yet in practice outweigh the limitations of guaranteed-deterministic programming models. One practical issue is that many applications do not fit into a single, restrictive paradigm—such as synchronous dataflow parallelism or functional task parallelism—which is what most guaranteed-deterministic parallel programming models offer. We find that would-be guaranteed-deterministic parallel programs either have multiple components spanning different paradigms, or they depend on deterministic algorithms not yet expressible in a guaranteed-deterministic fashion. For deterministic languages to make an impact, therefore, they require the breadth to span multiple paradigms and accommodate a wide variety of algorithms.

Determinism and its breadth dilemma Deterministic parallel languages necessarily restrict effects. If they allowed arbitrary access to shared memory locations, nondeterminism would directly follow. The particular restrictions on effects vary by language: in a stream-processing language (e.g., StreamIt [8]), the only effects possible within a stream filter are push() and pop() operations on a linear stream data structure. Deterministic data-parallel languages [5, 20], on the other hand, typically do not allow effects in parallel regions at all, encapsulating parallelism in aggregate operations such as map and fold that apply pure element-wise functions. Clearly, many deterministic parallel programs cannot be expressed with these abstractions: for example, an asynchronous quorum voting program, where a vote succeeds (and causes an effect) only when the number of “aye’s” exceeds a threshold.

Sharply restricted communication and synchronization capabilities have consequences not only for the immediate usability of guaranteed-deterministic languages, but for those languages’ extensibility as well. In an unrestricted parallel language, such as Java, new synchronization or communication constructs can always be implemented as needed, without changing the language—but no deterministic parallel language has offered anything comparable.

\footnotesize{\textsuperscript{1} We refer here to external determinism, also called determinacy. Of course, many parallel applications depend critically on observably nondeterministic behavior—for example, hardware designs and GUIs. These are not candidates for deterministic execution, but that still leaves many that are.}
The difficulty of extensibility means that having a fully fleshed-out set of built-in parallel primitives is more important in a deterministic parallel language than it is in a traditional, unrestricted language. If a guaranteed-deterministic parallel language must limit the user to a certain set of idioms, then those idioms should encompass as much functionality as possible.

Choosing such a set of broadly applicable built-in parallel idioms is not easy. They must preserve determinism even under arbitrary composition and even against an adversarial programmer. Determinism is a global property of the language that can be difficult to verify, as the proofs of determinism for such languages testify \cite{[3, 4, 9, 11]}, and composition of language features may not preserve determinism, even if those features behave deterministically in isolation. Finally, adding features to a parallel runtime system can become an increasingly delicate engineering challenge as the feature list grows \cite{7}.

**LVars: a step forward** In this paper we build on our previous work on the LVars programming model \cite{11, 12} and the accompanying LVish library for deterministic parallel programming in Haskell. LVars (which we review in more detail in Section 2) are shared monotonic data structures to which information can only be added, never removed, and for which the order in which information is added is not observable.

The key insight behind LVars is that the states a shared data structure can take on have an ordering, and updates preserve that ordering because writes take the least upper bound (lub) of the previous value and the new value. Because the lub operation is idempotent, multiple writes of the same value to a single location can be allowed deterministically. LVars are already far more expressive than the write-once IVars \cite{11} of prior work on deterministic parallelism, and are a step in the direction of broadly-applicable deterministic parallelism. Unfortunately, they fall short in a wide variety of domains where deterministic parallel algorithms should be expressible.

**Our contributions** In this paper we describe the design and implementation of extensions to LVish that enable the following capabilities, while leaving the basic model intact:

- Commutative (but non-idempotent) read-modify-write operations such as fetch-and-add (Section 5). We make use of this capability to parallelize PhyBin \cite{18}, a bioinformatics application for comparing genealogical histories (phylogenetic trees) that relies heavily on a parallel tree-edit distance computation.
- Full access to mutable state with enforced disjoint access for parallel threads (Section 5). The addition of this feature to LVish is, to our knowledge, the first integration of parallel updates to mutable memory (à la Deterministic Parallel Java \cite{3}) with blocking dataflow communication in a guaranteed-deterministic programming model.
- Deterministic speculation and cancellation, which have a particular synergy with a new data structure we add to LVish for memoization (Section 6).

The result of our work is a significantly extended LVish library, now well suited for parallelizing a far broader variety of pre-existing Haskell programs. LVish is implemented purely as a Haskell library, even though it provides features that usually require language extensions, e.g., enforced alias-free mutable data to support disjoint parallel update. Further, we show in Section 7 that our new library is effective, providing parallel speedup on benchmarks old and new, and a 3.35× parallel speedup on eight cores for the parallelized PhyBin application.

In order to implement our extensions to LVish, we introduce par-monad transformers (Section 4), an extension of traditional monad transformers. Using transformers to add new capabilities only where they are required has two benefits: first, the cost of added functionality is paid only when it is needed, and, second, it minimizes the impact of our changes on the core LVish scheduler. Moreover, the infrastructure we have created for par-monad transformers paves the way for future extensibility and provides a modular way to think about determinism guarantees.

## 2. Background: LVars and LVish

The LVars programming model \cite{11, 12} offers a principled approach to guaranteed-deterministic parallel programming with shared state. In this section we review previous work on LVars and LVish, a Haskell library that implements the LVars programming model; Sections 5 then describe our extensions to LVish.

An LVar is a mutable data structure that can be shared among multiple threads. Unlike an ordinary shared mutable data structure, though, LVars come with a determinism guarantee: a program in which all communication among threads takes place through LVars—and in which there are no other observable side effects—is guaranteed to evaluate to the same value on every run, regardless of thread scheduling. This determinism property holds because for every LVar, the set of states that the LVar can take on forms a lattice \cite{4} equivalent to that of LVar, and the semantics of reading from and writing to the LVar is defined in terms of this lattice of states. The two fundamental LVar operations are put, for writing, and get, for reading. At a high level:

- A put operation can only change an LVar's state in a way that is inflationary with respect to its lattice. Informally, the contents of an LVar must stay the same or "grow bigger" with each write.
- A get operation allows limited observations of the state of an LVar. The key idea is that reads from an LVar are threshold reads: they only return a value when the LVar's state meets a certain (monotonic) criterion, or "threshold", and the value returned is the same regardless of how high above the threshold the LVar's state goes. We give a concrete example later in this section.

Together, least-upper-bound puts and threshold gets guarantee that programs behave in an observably deterministic way, despite schedule nondeterminism and concurrent access to shared memory. Furthermore, since the LVars model is lattice-generic, it guarantees the safety of arbitrary compositions of programs mixing and matching concurrent data structures, so long as the state spaces of those data structures can be viewed as lattices and the operations their APIs expose are expressible in terms of puts and gets\cite{4}.

**The LVish library** LVish is an implementation of the LVars programming model as a library in Haskell. Like the monad-par library that preceded it \cite{16}, the LVish library provides a Par monad for encapsulating parallel computation. Par computations run in lightweight, library-level threads that are scheduled by a custom

---

\footnote{Formally, the lattice of states is given as a 4-tuple \((D, \subseteq, \bot, \top)\) where \(D\) is a set, \(\subseteq\) is a partial order on \(D\), \(\bot\) is \(D\)'s least element according to \(\subseteq\), and \(\top\) is \(D\)'s greatest element. We do not require that every pair of elements in \(D\) have a greatest lower bound, only a least upper bound; hence \((D, \subseteq, \bot, \top)\) is really a bounded join-semilattice with a designated greatest element \((\top)\). For brevity, we use the term "lattice" as a shorthand.}

\footnote{In practice, it is also important to be able to register latent event handlers that run when puts that change the state of an LVar occur, but these are equivalent to an implicit set of functions blocked on gets.}

---
work-stealing scheduler provided by LVish. LVish also provides a variety of LVar data structures (e.g., sets, maps, graphs) that support concurrent insertion, but not deletion, during Par computations.

In addition to the data structures the LVish library provides, users may implement their own LVar data structures (although note the proof obligations for data structure implementors, below). LVars can be quite sophisticated and correspond to many physical memory locations (e.g., implemented as a concurrent skip list or bag), but the simplest way to implement an LVar data structure (and the easiest way to satisfy said proof obligations) is to represent it as a single, pure value in a mutable box. LVish provides a PureLVar type constructor to facilitate the definition of such “pure” LVars.

An example: parallel “and” Consider an LVar that stores the result of a parallel logical “and” operation between two inputs. At any point in time, each input will have written true, false, or nothing yet. In Haskell, we encode this:

```haskell
data Inp = Bot | T | F
```
The state of a complete parallel-and LVar, then, would capture the state of each of its inputs, plus the possibility of an error (the top state in Figure 1). In Haskell we model this using the following type constructor for lattice states:

```haskell
type State = Maybe (Inp,Inp)
top :: State
	= Nothing
Maybe
```
is commonly used for computations that might fail. Indeed, here top (Nothing) represents a failure, whereas the Just (Bot, Bot) state is the least element of the lattice and represents the state of the LVar before any writes have taken place. Then, when one of the two inputs becomes available, the state moves to the second tier of states above (Bot, Bot), and then to the third if a second input arrives.

The join (lb) function for States combines their information, and is as simple as writing down the lattice from Figure 1 as a total function in Haskell:

```haskell
instance JoinSemiLattice State where
  -- Use the Maybe monad to keep this code simple
  join a b =
    do (x1,y1) <- a
       (x2,y2) <- b
       x3 <- joinInp x1 x2
       y3 <- joinInp y1 y2
       return (x3,y3)

joinInp :: Inp -> Inp -> Maybe Inp
joinInp x y | x == y = Just x
joinInp Bot x = Just x
joinInp T F = top
joinInp x y = joinInp y x
```

Next, we can use the PureLVar type constructor provided by LVish to define an LVar type called AndLV, whose states are of the State type we just defined:

```haskell
type AndLV = PureLVar State
```

Threshold reads from an AndLV Under what circumstances can we deterministically read from an AndLV? It is not safe, for example, to test whether one or both inputs have been written at a point in time. Rather, we must describe a monotonic threshold function for when the read may return. One handy way to do this is to create a threshold set of (pairwise incompatible) “trigger values”. When the state of the LVar moves above a trigger, the trigger is returned as the result of the get operation. For example:

```haskell
getAndLV :: AndLV -> Par Bool
getAndLV lv = do
  let bothtrue = [Just (T, T)]
  anyfalse = map Just [(F, Bot), (Bot, F), (F, F), (F, T), (T, F), (F, F)]
  res <- getPureLVar lv [bothtrue, anyfalse]
  return (res == bothtrue)
```

Here, bothtrue and anyfalse are the threshold triggers. These two sets are pairwise incompatible (that is, the lub of Just (T, T) and each element of anyfalse is top), and thus no more than one of them can be activated by monotonic change in lv’s state. When getPureLVar returns, res holds whichever of the triggers was activated. Note that getPureLVar may unblock after only one input is written, if that input is false; otherwise, it must wait for the second input.

Adding parallelism An AndLV variable can be shared between threads, but does not itself add parallelism. The next step is to build a combinator for launching two boolean computations in parallel, and returning the result of their logical and:

```haskell
asyncAnd :: Par Bool -> Par Bool -> Par Bool
asyncAnd m1 m2 = do
  res <- newPureLVar bottom
  fork (do b1 <- m1
          putPureLVar res (Just (toInp b1, Bot)))
  fork (do b2 <- m2
          putPureLVar res (Just (Bot, toInp b2)))
  getAndLV res
```

Finally, to run an asyncAnd computation, we can use the runPar operation provided by LVish, which converts Par a to a—initializing the library’s scheduler and running the parallel computation. Thus

![Figure 1. Lattice of states that a parallel-and LVar (that is, an AndLV) can take on. The five red states in the lattice correspond to a false result, and the one green to a true one.](image-url)
runPar provides the means by which parallel LVish code can be embedded inside ordinary, pure Haskell programs. For example, to fold asyncAnd over the results of 100 trivial boolean computations launched in parallel, we could write:

```haskell
main = do
  print (runPar
    foldr asyncAnd (return True)
    (concat (replicate 100 (return True, return False)))))
```

**Proof obligations for LVish data structures** When implementing a data structure with LVish, it is the data structure author’s obligation to ensure that the states of their data structure correspond to elements of a lattice, and that the operations in the API they expose would be expressible using the aforementioned put and get operations. To put it another way, operations on a data structure exposed as an LVar must have the semantic effect of a lub for writes or a threshold for reads, but none of this need be visible to clients. Any data structure API that provides such a semantics is guaranteed to provide deterministic communication.

Because AndLV has a finite lattice, its join function can be trivially and exhaustively verified to compute a lub. In fact, the following list comprehension generates every possible input:

```haskell
[ join x y | x ← [ bottom .. top ] , y ← [ bottom .. top ] ]
```

Likewise, it is trivial to verify this join’s associativity, commutativity, and idempotence. In general, however, definitions of LVar-based data structures, their join functions, or their get operations should occur only in trusted code. Fortunately, most LVish applications need not define any new LVar-based data structures and instead make use of those that the LVish library provides.

**A missing feature: task cancellation** The AndLV LVar just described makes it possible to do a “short-circuit” computation because getAndLV unblocks and returns a result as soon as any F is written. However, it is still the case that the other thread writing to the AndLV runs to completion. Although this cannot affect the deterministic outcome of the computation, it needlessly uses up cycles. This motivates the desire to be able to cancel an in-flight thread that cannot affect the deterministic outcome of a Par computation. We discuss our extension that enables cancellation in Section 5.

**Features not covered here** In this section we overviewed LVars and their use, but have not covered all features in detail. In particular, it is possible to register handlers that are invoked whenever an LVar changes, and also to freeze an LVar to read its contents exactly. For example, freezing enables iteration over the full contents of collection LVars, and can be done unsafely during parallel computation, or safely upon exiting the parallel computation (runParThenFreeze). Both these features are covered in detail in previous work [12].

### 3. Warm-up: Read-modify-write extension

In the original LVars model [11] [12], the only way for the state of an LVar to evolve over time is through a series of least-upper-bound (lub) updates resulting from calls to put operations. Unfortunately, this way of updating an LVar provides no efficient way to model an atomically incremented counter that occupies one memory location. Yet, atomic increments to such a counter are efficient, commutative, and ultimately fit well inside the LVish framework. Hence the most basic extension we make to LVish is to (optionally) relax its reliance on idempotence of operations. Thereafter we can safely add a restricted set of atomic read-modify-write operations that are inflationary with respect to a lattice, but do not compute a lub. We will see one example of an application that critically requires this functionality in Section 7.1.

We consider a new family of LVar update operations that are required to commute and be inflationary with respect to the lattice in question, but are not limited to the lub semantics of put. Specifically, for a lattice \( (D, \sqcup, \perp, \top) \), a data structure author may define a set of bump operations \( \text{bump}_i : D \to D \), which must meet the following two conditions:

- \( \forall a, i. \ a \sqsubseteq \text{bump}_i(a) \)
- \( \forall a, i, j. \ \text{bump}_i(a) = \text{bump}_j(a) \)

As a simple example, consider an LVar whose states are natural numbers, with 0 as the least element and with the usual \( \leq \) operation on natural numbers as the ordering on states. The ordering induces a lub operation equivalent to the \( \max \) operation on natural numbers. We can implement a family of bump operations that increment the LVar by various amounts: \( \{(+1), (+2), (+3), \ldots \} \). A critical point to note, however, is that it is not safe to update the same LVar with both put and bump. For example, a put of 4 and a \( \text{bump}_{+1} \) do not commute! If we start with an initial state of 0 and the put occurs first, then the state of the LVar changes to 4 since \( \max(0, 4) = 4 \), and the subsequent \( \text{bump}_{+1} \) updates it to 5. But if the \( \text{bump}_{+1} \) happens first, then the final state of the LVar will be \( \max(1, 4) = 4 \). Furthermore, multiple distinct families of bump functions only commute among themselves and cannot be combined. In practice, this distinction is enforced by the type system. For example, the LVish-provided set data structure Data.LVar.Set supports only put, whereas Data.LVar.Counter supports only bump.

Composing these data structures, however, is fine. For example, an LVar could represent a monotonically growing collection (which supports put) of counter LVars, where each counter is itself monotonically increasing and supports only bump. Indeed, the PhyBin application described in Section 7.1 uses just such a collection of counters.

**Determinism guarantee** The determinism of LVish [11] relies on the fact that the states of all LVars evolve monotonically with respect to their lattices, and that the lub operation is commutative and therefore the order in which puts occur does not matter. Together, these properties suffice to ensure that the threshold reads made by get operations are deterministic. The rest of an LVish program is purely functional, and its behavior is, in fact, a pure function of these get observations. Since bump operations are also commutative and inflationary with respect to the lattice of the LVar they operate on, we see that they preserve determinism, as long as programs do not use bump and put on the same LVar.

We can go further and generalize this argument, and in fact will need to for the other extensions described in this paper. Consider LVish with put, bump, and get effects. By commutativity, we can reduce the effects in a program execution to three unordered sets: \( (P, B, G) \). By slicing the system at this interface, we can decompose determinism proof obligations into two parts:

- The LVish (Haskell) code must guarantee that it implements a monotone function \( \mathcal{P}(G) \to \mathcal{P}(P) \times \mathcal{P}(B) \). That is, adding more get results on the left results only in more put/bump effects on the right (as more gets unblock and run, more put and bump operations can run); furthermore, those effects are a function of nothing other than get.
- The mutable (but monotonically growing) heap of LVars must likewise guarantee that the set of get results is a pure function of the set of puts and bumps. This is straightforward to show, given the lattice-based semantics of put and bump.

This communicating agents formulation of determinism for LVish puts us in a position to replace the monotonically growing heap with other deterministic agents that fulfill the same contract, as we
will see in Section 7. At that point, we will also discuss temporal contracts on the order of operations, going beyond the simple case of completely unordered sets at the interfaces.

**Deleveraging idempotency** Since lub is an idempotent operation, the previously existing LVish implementation assumed idempotence of all writes, which in turn enabled the scheduler to relax synchronization requirements at the cost of low-probability duplication of work [12]. Adding support for operations like bump makes this assumption untenable. Therefore, we re-engineered the LVish runtime system to (optionally) include additional synchronization.

**Fine-grained effect tracking** Naturally, it is best to pay the aforementioned synchronization overhead only when required. This requires static information about whether a given program uses bump. With that as one of our goals, we extend LVish to allow for static fine-grained effect tracking. The idea is to guarantee that only certain LVar effects can occur within a given Par computation. In Haskell, we can do so at the type level by indexing Par computations with a phantom type e that indicates their effect level. That is, the Par type becomes, instead, Par e s a, where e is a type-level encoding of booleans indicating whether or not writes, reads, non-idempotent (bump), or non-deterministic (10) operations are allowed to run inside it.

Moreover, in real LVish programs, the Par type constructor has a second type parameter, s, making Par e s a the complete type of a computation that returns a result of type a. The s parameter ensures that it is not possible to reuse an LVar from one runPar session to the next, just as the ST monad in Haskell prevents an STRef from escaping runST; likewise the types of individual LVars must be parameterized by s as well. For simplicity of presentation, we elided the e and s type parameters in Section 2 instead following the simpler Par a format of the earlier monad-par library [16], but we include them from this point onward.

To enable future additions of effect “switches” encoded in e, we follow the precedent of recent work by Kiselyov et al. on extensible effects in Haskell [10]; we abstract away the specific structure of e into type class constraints, which allow a Par computation to be annotated with the interface that its e type parameter is expected to satisfy. For example, a Par computation annotated with the effect level constraint HasPut can perform puts. Thus the signature for the put operation on LVars becomes:

```haskell
put :: HasPut e => IVar m a -> a -> Par e s ()
```

while the signature for an incrCounter operation uses the HasBump constraint:

```haskell
incrCounter :: HasBump e => Counter s -> Par s e ()
```

These constraints can also be negative. For example, the runPar function for executing Par computations in a purely functional context requires the absence of explicit freeze or IO operations:

```haskell
runPar :: (NoFreeze e, NoIO e) => (forall s. Par e s a) -> a
```

4. Par-monad transformers

The effect-tracking system of the previous section gives us a way to toggle on and off a fixed set of basic capabilities using the type

> Space constraints preclude full description here, but the key challenge is resolving a race between puts and attempts to register new handlers (callbacks) on an LVar. Our solution is a specialized variant of a reader-writer lock that requires zero writes to shared addresses if no handlers are currently being registered.

> To be precise, in the earlier 1.x releases of LVish, the e type parameter for effect level was instead used for “determinism level”, and was a simple type-level boolean switch distinguishing deterministic from quasi-deterministic Par computations [12]. The effect signatures in this paper generalize determinism levels and correspond to the newer LVish 2.x API.

4. Par-monad transformers

The effect-tracking system of the previous section gives us a way to toggle on and off a fixed set of basic capabilities using the type system—that is, with the switches embedded in the e parameterizing the Par type. These type-level distinctions are needed for defining restricted but safe idioms, but they do not address extensibility. For that, we turn to multiple monads rather than a single parameterized Par monad.

Working Haskell programmers use a variety of different monads: Reader for threading parameters, State for in-place update, Cont for continuations, and so on. All monads support the same core operations (bind and return from the Monad type class) and satisfy the three monad laws. However, each monad must also provide other operations that make it worth using. Most famously, the 10 monad provides various input-output operations.

A monad transformer, on the other hand, is a type constructor that adds “plug-in” capabilities to an underlying monad. For example, the StateT monad transformer adds an extra piece of implicit, modifiable state to an underlying monad. Adding a monad transformer to a type always returns another monad (preserving the Monad instance). In the same way, we can define a Par-monad transformer as a type constructor T where, for all Par monads a, T a is another Par monad with additional capabilities, and a value of type T m a, for instance. T (Par e s) a, is a computation in that monad. Indeed, Par-monad transformers are valid monad transformers (in the sense of providing a standard MonadTrans instance).

Just as Monad is a type class (interface) with associated laws, the semantics of a Par monad is captured by a series of type classes, all of which are closed under Par-monad transformer application. At minimum, a Par monad must have a fork operation, satisfying this type class:

```haskell
class (Monad m) => ParMonad m where
  fork :: m () -> m ()
```

Programs with fork create a binary tree of monadic actions with () (unit) return values.

Whereas the original LVish library provided a single, concrete Par type, here we allow any instantiation of the ParMonad type class. Additional type classes capture the interfaces to basic parallel data structures and control constructs such as futures (ParFuture), IVars (ParIVar), and more general LVars (ParLVar). For example, the class ParLVar provides new and put methods with the signatures below:

```haskell
class (ParMonad m) => ParFuture m where
  fork :: m () -> m ()
```

```haskell
class (ParMonad m) => ParLVar m where
  type lVar m :: a -> m a
  new :: m (lVar m a)
  put :: (lVar m a) -> m ()
  get :: (lVar m a) -> a
```

The ParFuture, ParIVar, and ParLVar type classes form a hierarchy: any implementation that can support LVars can support IVars, and any that can support IVars can support futures. Taken together, this framework for generic Par programming makes it possible for LVish programs to be reusable across a variety of schedulers. This can be quite useful; for example, we provide a ParFuture instance for the native GHC work-stealing scheduler [15].

**Example: threading state in parallel** Perhaps the simplest example of a Par-monad transformer is the standard StateT monad transformer (provided by Haskell’s Control.Monad.State package). However, even if a is a Par monad, for StateT s m to also be a Par monad, the state s must be splittable; that is, it must be specified what is to be done with the state at fork points in the control

5. By “the original”, we refer to 1.x releases of LVish, e.g., http://hackage.haskell.org/package/lvish-1.1.2

6. Although it may appear that generic treatment of Par monads as type variables a removes the additional metadata in a type such as Par e s a, note that it is possible to recover this information with type-level functions.
flow. For example, the state may be duplicated, split, or otherwise updated to note the fork. The below code promotes StateT to be a Par-monad transformer:

```haskell
class SplittableState a where
  splitState :: a → (a,a)

instance (SplittableState s, ParMonad m) ⇒
  ParMonad (StateT s m) where
  fork task =
    do s ← State.get
       let (s1,s2) = splitState s
       lift (fork (do runStateT task s1; return ()))

Note that here, put and get are not LVVar operations, but the standard procedures for setting and retrieving the state in a StateT. Here are two immediately useful applications of threaded, splittable state:

• PedigreeT keeps the index in the binary control-flow tree as implicit state, e.g., “LRLRL”. This is sometimes called the pedigree of the parallel computation [13]. In this case the split action is to add “L” or “R” for each branch of the fork, respectively. Pedigrees can then be augmented with counters that increase with certain sequential actions, thus providing a form of parallel “program counter”. Also, examining pedigrees at runtime can answer “happens before” or “happens in parallel” questions.

• RngT is an application of pedigrees to the problem of deterministic pseudo-random number generation. The idea is simple: either use the pedigree itself as a seed, or keep the random generator state itself with StateT. The interface to the user is a simple rand nullary function that can be called on any thread.

In fact, parallel deterministic random number generation was considered important enough for Intel to significantly modify the Cilk runtime system to support it directly [13]. In LVish, no such runtime system modification is necessary: instead, we add the StateT transformer to Par to track pedigree only for applications that need it. (Section 7.2 discusses the overhead of Par-monad transformers.) Further, given the above instances, we can declare all random number generators into splittable states, and thus define a very simple interface for random number generation, e.g.:

```haskell
instance RandomGen g ⇒ SplittableState g where
  splitState = System.Random.split
  randInt :: (ParMonad m, RandomGen g) ⇒ StateT g m Int
```

**Determinism guarantee** The StateT transformer preserves determinism because it is effectively syntactic, or syntactic. That is, StateT does not allow one to write any program that could not already be written using the underlying Par monad, simply by passing around an extra argument. This is because StateT only provides a functional state (an implicit argument and return value), not actual mutable heap locations. Genuine mutable locations in pure computations, on the other hand, require Haskell's ST monad, the safer sister monad to 10. We return to ST in Section 5.

**The case for pluggability** Why should parallel effects be plug-in, rather than baked-in? In summary, there are three reasons:

• Modularity: Runtime systems for parallel schedulers like Cilk and language runtimes like GHC’s grow into enormously complicated low-level concurrent codebases. Isolating parallel capabilities in transformers makes them modular and maintainable.

• Runtime cost: The transformers introduced in this paper introduce book-keeping and synchronization overheads (Section 7.2), which should be paid only by computations that use them. Expensive features should pay their own way.

• Composability: While a user only wants one copy of RngT—and thus it could be hard-coded into the scheduler if desired—other transformers make it useful to have more than one copy in the stack. For example, a program with two implicit states might stack two StateT transformers. This is not possible for capabilities baked into the core scheduler.

**Engineering note: independent extensibility** Because extensibility is an explicit goal, we must ask what functionality can be added by separate packages, deployed independently from LVish. The framework presented thus far enables new (trusted) packages to add transformers that preserve the ability to use core data structures; that is, they provide instances for the classes Par Monad, Par Future, and Par LVVar. In the other direction, separate packages can provide new data structures that work with the base Par monad. But how can new transformers provide instances for new data types they do not know about? This is simply the problem of “independent extensibility” in a new guise.

Fortunately, there is a good solution. The interactions between a concurrent data-structure implementation (such as Data.LVar.Map or Data.LVar.Counter) and the scheduler are limited and have a common structure. Thus, rather than splitting out fine-grained classes for each conceivable data structure (ParMap, ParCounter, and so on), we make ParLVVar into a general data-structure/scheduler interface [14]. For intuition, a small portion of the ParLVVar type class interface is shown below. As described in previous work [13], the implementation distinguishes between the type of complete LVars states, for which we use the type parameter a, and state changes, or “deltas”, for which we use a:

```haskell
class (Monad m, ...) ⇒ ParLVVar m where
  type LVVar m :: * → * → *
  newLV :: IO (a → m (LVVar m a d))
  liftLV :: ParLVVar m a d → (a → IO (Maybe d)) → m ()
```

With this approach, a generic monotonic Map package can work with any monad satisfying ParLVVar, including those produced by stacking transformers that were written with no knowledge of the data structure. Likewise, transformers that preserve ParLVVar instances work with past and future data structures.

5. Disjoint parallel update with ParST

LVish is based on the notion that it is fine for multiple threads to access and update shared memory, so long as updates commute and “build on” one another, adding information rather than destroying it. Yet it should be possible for threads to update memory destructively, so long as the memory updated by different threads is disjoint. This is the approach to deterministic parallelism taken by, for example, Deterministic Parallel Java (DPJ) [3], which uses a region-based type and effect system to ensure that each mutable region of the heap is passed linearly to a thread that then gains exclusive permission to update that region. In order to add this capability to LVish, though, we need destructive updates to interoperate with other LVish put/get/bump effects. Moreover, we wish to do so at the library level, without requiring language extensions.

Our solution is to provide a ParST transformer, a variant of the StateT transformer of Section 4. ParST allows arbitrarily complex mutable state, such as tuples of vectors (arrays). However, ParST enforces the restriction that every memory location in the state is reachable by only one pointer: alias freedom.

---

10 We retain interfaces like Par IVars as well because they provide a means to interoperate with legacy Par monads that provide only IVars or futures and have no notion of LVars, or with the built-in GHC work-stripping runtime itself, which provides only a Par Future instance.
Previous approaches to integrating mutable memory with pure functional code (i.e., the ST monad) work with LVish, but only allow thread-private memory. There is no way to operate on the same structure (for instance, on two halves of an array) from different threads. ParST exploits the fact that it is perfectly safe to do so as long as the different threads are accessing disjoint parts of the data structure. Below we demonstrate the idea using a simplified convenience module provided alongside the general (ParST) library, which handles the specific case of a single vector as the mutable state being shared.

```haskell
runParVecT 10 (
  do -- Fill all 10 slots with "a":
    set "a"

    -- Get a pointer to the state:
    ptr ← reify

    -- Call pre-existing ST code:
    new ← pickLetter ptr
    forkSTSplit (SplitAt 5) (write 0 new) (write 0 "c")

    -- ptr is again accessible here ...
)
```

This program demonstrates running a parallel, stateful session within a Par computation. The shared mutable vector is implicit and global within the monadic do block. We fork the control flow of the program with `forkSTSplit`, where `(write 0 new)` and `(write 0 "c")` are the two forked child computations. The `SplitAt` value describes how to partition the state into disjoint pieces: `(SplitAt 5)` indicates that the element at index 5 in the vector is the “split point”, and hence the first child computation passed to `forkSTSplit` may access only the first half of the vector, while the other may access only the second half. (We will see shortly how this generalizes.) Each child computation sees only a local view of the vector, so writing "c" to index 0 in the second child computation is really writing to index 5 of the global vector. This is exactly the splitting method in our parallel sort (Section 7.3).

Ensuring the safety of ParST hinges on two requirements:

- **Disjointness**: Any thread can get a direct pointer to its state. In the above example, `ptr` is an STVector that can be passed to any standard library procedures in the ST monad. However, it must not be possible to access `ptr` from `forkSTSplit`’s child computations. We accomplish this using Haskell’s support for higher-rank types ensuring that accessing `ptr` from a child computation causes a type error. Finally, `forkSTSplit` is a fork-join construct; after it completes the parent thread again has full access to `ptr`.

- **Alias freedom**: Imagine that we expanded the example above to have as its state a tuple of two vectors: \((v_1, v_2)\). (In fact, this is the state we need for the merge phase in Section 7.3) If we allowed the user to supply an arbitrary initial state to their ParST computation, then they might provide the state \((v_1, v_2)\), i.e., two copies of the same pointer. This breaks the abstraction, enabling them to reach the same mutable location from multiple threads (by splitting the supposedly-disjoint vectors at a different index).

Thus, in LVish, users do not populate the state directly, but only describe a recipe for its creation. Each type used as a ParST state has an associated type for descriptions of (1) how to create an initial structure, and (2) how to split it into disjoint pieces. We provide a trusted library of instances for commonly used types.

### State transformation

In comparison to the region-typing approach of DPJ, it can be painful to keep the state inside a single structure reachable from one variable. However, it is possible to define combinator libraries that make this much easier (in the spirit of the Lens library for Haskell). For example, we provide ways to either “zoom in”, that is, run a computation whose state is a sub-component of the current state, or “zoom out”, by placing the current state inside a newly constructed one. We use this ability inside our code for parallel merge sort (Section 7.3) to shift from a single vector state to having a second temporary buffer for the merge phase.

### Inter-thread communication

Disjoint state update does not solve the problem of communication between threads. Hence systems built around this idea often include other means for performing reductions, or require “commutativity annotations” for operations such as adding to a set. For instance, DPJ provides a `commutewith` form for asserting that operations commute with one another to enable concurrent mutation. In LVish, however, such annotations are unnecessary, because LVish already provides a language-level guarantee that all effects commute! Thus, a programmer using LVish with ParST can use any of the rich library of LV-based data structures to communicate results between threads performing disjoint updates, without requiring trusted code or annotations. Furthermore, to our knowledge, LVish now provides the first example of a deterministic parallel programming model allowing both DPJ-style, disjoint destructive parallel updates and blocking, dataflow-style communication between threads (through LVars).

### Determinism guarantee

The ParST transformer relies on the fact that the disjoint updates made by a `forkSTSplit` call are equivalent to a single sequential state update. This means that if ParST were a base monad instead of a transformer, its determinism would be a straightforward consequence of this disjointness property, which prevents data races. Indeed, ParST would be equivalent to a proper subset of DPJ, which is provably deterministic.

The complication is that a ParST computation may spawn arbitrary, asynchronous computations that use the underlying effects provided by the monads under it in the transformer stack, e.g., `put` and `get` on LVars. To convince ourselves that this is safe, we return to the “communicating agents” formulation of Section 3 to enable modular reasoning. The mutable heap of ST objects (STRef, STVector, and so on) becomes a third agent alongside the purely functional component of the LVish computation and the monotonically-growing heap. The purely functional agent exchanges `put/get` messages with the monotonically-growing heap and `read/write` messages with the mutable heap. In this case, however, there is a protocol that must be followed, and we cannot ignore ordering and control flow to reason only about the sets of messages exchanged.

In the basic LVish programming model there are two sources of ordering constraints: monadic bind, and data dependencies from `put` to `get`. Intuitively, we can think of the LVish agent emitting `Before(a, b)` messages, meaning that \((a, b)\) is in a `happens-before relation` for a pair of events \(a, b\) that it previously emitted. Normally, such a relation would be unnecessary; most Par monads are so order-insensitive that all their effects satisfy the following reordering-tolerance property:

\[
\text{(do m1; m2) == (do fork m1; m2)}
\]

There is a minor caveat here. DPJ requires that the type system statically determine disjointness of state updates, whereas LVish, can also allow complicated partitioning strategies that are only checkable at runtime. Nevertheless, DPJ could be extended with this functionality, and it does not affect the determinism argument.

\[\text{That is, the type of a child computation begins with } (\forall \ a . \ \text{ParST} \ldots).\]
But for a destructively mutable heap, ordering is important and ParST effects clearly cannot support the above property—write operations do not commute! In fact, ParST does not even expose a one-armed fork operation that allows ST effects in the child computation. Rather, it supports fork-join parallelism with forkSTsplit, which requires that both child computations complete before returning. Furthermore, the forkSTSplit control construct can be thought of as generating additional Before(a, b) messages to express these barriers.

On the mutable-heap side, the contract for determinism is the standard one: all read/write and write/write pairs must be ordered according to the Before relation. Of course, we do not track Before at runtime, so this must be guaranteed by construction. How can we guarantee this if there is a stack of monads composed with ParST? The key here is that get effects can only add more Before constraints, not take them away. Additional blocking operations can therefore never break the requirements for determinism in the mutable heap (race-freedom). Given that, the same argument as in Section 3 applies: all results returned to the LVish agent from the heap are deterministic, therefore its final value is.

ParST composition

The reader may legitimately be wondering: how can there be a ParST transformer, if there is no ST transformer in Haskell? The answer is that ParST is not a transformer supporting unfettered composition.

Instead, a given Par monad can either have the ST feature, or not. It is not safe to combine two copies of ParST, nor to apply ParST on top of certain other transformers that LVish might be eventually be extended with (e.g. ListT).

To implement this, each Par monad tracks one bit of information in its type: whether the ST switch has been turned on for this monad. Once this bit is turned on, new copies of ParST cannot be applied on top, but other transformers, such as RanT, can be added. Thus it is possible to compose reordering-tolerant transformers such as StateT and RanT freely on either side of the ParST transformation, without violating the invariants of the underlying state implementation.

6. Control-related Effects

In the previous section, we saw a Par transformer that restricts the control flow of an LVish program to retain determinism: ParST requires that child computations that modify the state are created in a fork-join, rather than asynchronous fork, fashion. In this section, we will instead look at transformers that add additional control-flow behaviors to a program: for example, the ability for one thread to cancel another.

Every Par monad provides continuation capture under the hook to be able to support work-stealing scheduling and blocking gets. This provides significant power for implementing new control constructs, but it does not change the fact that we must carefully identify limited idioms that retain determinism, and expose only those determinism-preserving constructs from the library of Par transformers.

6.1 Cancellation

It is common to speculatively create a parallel computation whose result may not be needed: for example, in search problems. In the parallel “and” example from Section 2 we saw that LVish programs written using the previously existing LVish library could create trees of parallel boolean operations and even allow them to make their results available before all branches completed execution. However, it was not possible to actually cancel the unneeded branches to avoid wasted CPU time.

Fundamentally, cancellation is a challenge for guaranteed-deterministic parallel programming because a cancelled thread might have side effects and the cancellation could race with those effects. With fine-grained effect tracking, though, we are able to provide a CancelT transformer providing operations such as forkCancel, which takes a computation as argument and runs it in parallel, returning a cancellable future; and cancel, which takes a CFuture and cancels the thread associated with it and all of that thread’s subthreads, transitively. It is an error to both cancel and read such a future, even if the read happens first. In our generic framework, the signatures for forkCancel and cancel include:

\[
\text{forkCancel} :: (ParVar m, ReadOnly m, \ldots) \Rightarrow \text{CancelT} m a \rightarrow \text{CancelT} m \text{ (CFuture m a)}
\]

\[
\text{cancel} :: (HasPut m2, \ldots) \Rightarrow \text{CFuture} m1 a \rightarrow \text{CancelT} m2 ()
\]

Note that forkCancel, which requires that the forked computation must be readOnly, uses the same monad, m, for the child and parent computations. This is only because lifting a read-only computation into one that includes writes (explicit subtype coercion) is done separately. In fact, because cancel may cause another thread to throw an exception, it counts as a put effect; thus a program with cancellation must have a non-read-only “trunk” that it connects read-only branches to.

Finally, if the user wants cancellation of child computations with arbitrary effects, a variant, forkCancelableIO, allows them but requires nondeterminism (that is, IO in its own effect signature.

Using forkCancelableIO we can write a version of the asyncAnd function from Section 2 which, when getAndLV returns a False, calls cancel to terminate any remaining (now useless) forked computations. Using forkCancelable directly is not possible because of the putParVar calls in the code. Because we have verified manually that this use of cancellable writes is safe, however, we could add a blessed version of asyncAnd to the library that works with ReadOnly computations.

Implementation

The CancelT transformer allocates one mutable location whenever a new CFuture is created by forkCancelable (regular forks continue to share the state of the parent). This location stores a tuple (live, children), which tracks whether the computation is still alive, and a list of the child C Futures, which must be cancelled if the current thread is cancelled. Thus the implementation is driven by polling a thread’s liveness every time a scheduler action (get, fork, put, and so on) is performed. Because scheduler actions are frequent, this is sufficient. Moreover, alternatives that support more direct preemption (e.g., using Haskell asynchronous exceptions to kill the underlying worker threads), require much more bookkeeping, as well as invasive modifications to the LVish scheduler itself.

6.2 Memoization

Even though cancellation allows us to write a more efficient version of asyncAnd—canceling wasted work—it remains the case that

\[\text{cancel} \rightarrow \text{put} \rightarrow \text{get} \rightarrow \text{put}\]

In fact, these subcomputations have an additional stipulation relating to exception semantics, which also applies to ReadOnly computations used in memoization. Briefly, normal LVish threads eagerly push exceptions up to the scheduler, which is necessary when threads perform side effects like put that may throw exceptions that appear deterministically. A cancellable future, on the other hand, must have no visible effect but its result, and thus we require that exceptions not be propagated to the parent until/unless the future is read.
ReadOnly cancelled computations are completely wasted: they cannot do externally useful work.

Canceling a computation that could do externally useful work would necessarily break determinism—or would it? In this section, we show how a cancellable, ReadOnly computation can help other threads along without interfering with determinism. The idea is that a cancellable ReadOnly computation can contribute work to a shared memo table LVar. Since the only observable effect of writing to the memo table is that calls to memoized functions run faster, determinism is preserved, regardless of whether the computation is cancelled.

A basic memo table has a direct encoding using only the public interface of Set and Map LVars. Specifically, we use one LVar for requests and a second for results:

```haskell
type Memo e s k v = (ISet s k, IMap k s v)
```

The set of requests is connected to a handler that launches a compute job for each unique request of type k. When a job completes, it stores the (k,v) pair into the IMap. Thus doing a lookup on the memo table consists of simply inserting into the set, and then performing a blocking get on the map. This provides an efficient way to memoize functions—even functions that have side effects inside the Par monad (i.e., makeMemo takes a function (a → Par e s b)). It is a great application of existing LVar data structures.

But a further synergy with CancelT is possible. The Memo type above has an e parameter that tracks the effect signature of the memoized function. However, making a memo table request means writing an element into the ISet—a put effect. Thus, reading from a memo table has a put effect! This in turn means that it cannot be cancelled. Fortunately, this is a place where we can identify a specific combination of parallel effects that compose well. It is safe for an alternate version of the memo-table get function to require a ReadOnly memoized function, and in return hide (bless as safe/unobservable) the put effect in the result signature:

```haskell
getMemoRO :: (ReadOnly e) ⇒ Memo e s k v → k → Par e s v
```

Then, with getMemoRO, we can safely use ReadOnly memo tables inside cancelled computations! Hence we retain a full determinism guarantee, while canceling unneeded work, and retaining partial solutions discovered in the cancelled threads. The result is that read-only computations that use memoized functions can allow one to learn something from a computation that never happened—deterministically!

**The rest of the Zoo** While we do not have space to cover all of them here, there are other interesting examples of transformers that deal with parallel control flow. One example is DeadlockT, which returns when all computations underneath a forked child have either returned or blocked indefinitely. This transformer is useful for detecting and responding to cycles in graphs of computations. Deadlock-detecting computations have the opposite effect requirement from cancellation: rather than requiring read-only computations for determinism, they require “blind” computations which may only write to the world outside the subcomputation. (If they could read, they could block on data outside of their control, which creates ambiguity between genuine deadlock and temporary blocking.)

Another example is BulkRetryT, which improves the ability of a Par monad to support the deterministic reservations 2 idiom efficiently, and is described in a workshop paper 17. In brief, Par1Var monads already support blocking reads, but to efficiently execute a parallel for loop with a large iteration space, it is often better to cheaply mark the iterations that fail and retry them in bulk. However, the approach of aborting and retrying rather than blocking requires that each iteration of computation have only idempotent effects. In this example and others, we see that fine-grained effect tracking is essential to how our zoo of additional capabilities interoperate.

### 7. Evaluation

In this section, we evaluate the performance of our extended LVish library. We begin with a case study describing our experience using LVish to parallelize PhyBin, a bioinformatics application, and compare the performance of our parallelized PhyBin with its competitors. Next, we benchmark to measure the runtime overhead incurred by our use of Par transformers. Finally, to measure the effectiveness of our ParST transformer for disjoint parallel update, we evaluate its performance on a parallel merge sort benchmark. All measurements come from a dual-socket (12-core) Intel Xeon X5660 system, running RHEL Linux 6.4.

#### 7.1 Case Study: PhyBin: all-to-all tree edit distance

A phylogenetic tree represents a possible ancestry for a set of N species. Leaf nodes in the tree are labeled with species’ names, and the structure of the tree represents a hypothesis about common ancestors. For a variety of reasons, biologists often end up with many alternative trees, whose relationships they need to then analyze. PhyBin is a medium-sized (3500-line) bioinformatics program for this purpose, initially released in 2010. The primary output of the software is a hierarchical clustering of the input tree set (a tree of trees), but most of its computational effort is spent computing an N×N distance matrix, which records the pairwise edit distance between trees. It is this distance computation that we parallelize in our case study.

The distance metric itself is called Robinson-Foulds (RF) distance, and the fastest algorithm for all-to-all RF distance computation is the HashRF algorithm 19, introduced by a software package of the same name20. HashRF is about 2-3× as fast as PhyBin. Both packages are dozens or hundreds of times faster than the more widely-used software that computes RF distance matrices (e.g., Phylip9, DendroPy21). These slower packages use N2−N full applications of the distance metric, which has poor locality in that it reads all trees in from memory N2−N times.

Before describing how the HashRF algorithm improves on this, we must observe that edit distance between trees (number of modifications to transform one to the other) can be reduced to symmetric set difference between sets of bipartitions. That is, each intermediate node of a tree can be seen as partitioning the set of leaves into those below and above the node, respectively. For example, with leaves A, B, C, D, and E, one bipartition would be ‘‘ABC|DE’’, while another would be ‘‘AB|CDE’’. Identical trees, of course, convert to the same set of bipartitions. Furthermore, after convert-
Table 1. PhyBin performance comparison with DendroPy, Phylip, and HashRF. All times in seconds.

<table>
<thead>
<tr>
<th>Trees</th>
<th>Species</th>
<th>PhyBin</th>
<th>DendroPy</th>
<th>Phylip</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>150</td>
<td>0.269</td>
<td>22.1</td>
<td>12.8</td>
</tr>
<tr>
<td></td>
<td>Phylin 1, 2, 4, 8 core HashRF</td>
<td>4.7</td>
<td>3</td>
<td>1.9</td>
</tr>
</tbody>
</table>

Figure 2. The overhead of adding one StateT transformer (left) or ParST transformer (right). The Y axis is the speedup/slowdown factor (higher better), and the X axis is the count of benchmarks. Each color represents one of the benchmarks drawn from Figure 4. For each benchmark, there is a different bubble per thread setting, with the area proportional to the number of threads. We do not see a trend with more or less overhead at larger numbers of threads. All times are the median of five runs.

7.2 Benchmark 1: overhead of transformers

Monad transformers have both direct and secondary costs. The direct cost is to pay for what they do; the secondary cost is that complicated monad-transformer stacks result in extremely complicated code that the GHC compiler must unravel to optimize effectively. Our Par transformer approach only requires paying these overheads when a specific capability is needed, but we must still account for what that cost is and whether it is prohibitive.

LVish’s primary focus is on non-traditional parallel applications such as k-CFA program analysis [12] or PhyBin. Nevertheless, here we also include a benchmark suite of traditional parallel kernels shown in Figure 4. We use these in Figure 2, as well, which summarizes the overhead added when running this benchmark suite with additional, unneeded transformers added. We measure overheads for adding a StateT or ParST transformer. (Note that a CancelT is just such a StateT.) These result in a 4% geomean slowdown, and 2% geomean speedup, respectively. Indeed, the interactions of these transformers with the GHC compiler’s optimizer are difficult to predict, but overall, overhead is not prohibitive.

7.3 Benchmark 2: non-copying parallel sorting

To measure the effectiveness of our ParST transformer, we ported a well-known parallel merge sort implementation originally written in Cilk and later reimplemented in DPJ [3]. We omit the details of the algorithm and comment only on its formulation with ParST. This is a destructive mergeSort function, which assumes a vector state, and leaves the sorted result occupying the same memory locations as the input:

mergeSort :: (ParMonad parM) => ParST (MVector s2 elt) s parM ()

This function works over any underlying monad parM, extended with the ParST effect. Internally, the algorithm must add a second buffer to have extra space for merging, shifting the state to (v1, v2). At the fork points, both of these buffers are split at the same locations. The code for the heart of the parallel sort is:

forkSTSplit (sz1, sz1) (do forkSTSplit (sz2, sz2) mergeSort mergeSort mergeL2R)

mergeL2R

As in both the DPJ and Cilk implementations, we need to unroll the recursive sorting process, splitting twice. This ensures that after each round the output ends up back in the original buffer. The type-
checking of a parameters ensures that the nested splits can access only exactly the data they have permission to.

In summary, the ParST-based Haskell implementation offers exactly the same determinism guarantee offered by DPJ. Our version has the disadvantage of being written with a more restrictive (single implicit state object) mechanism, but it has the advantage of being callable from a purely functional context (e.g., from within a function of type \( \text{Int} \rightarrow \text{Int} \)) with a guarantee that no visible side effects occur.

Performance The reason performance suffered in previous parallel sort implementations in Haskell \[7\] is that each recursive call to \text{mergeSort} had to perform an append (copy) to combine the halves together. The two independent recursions had to return fresh values, because no mechanism for (deterministic) mutation in parallel was available. The performance of such a copying merge sort is shown in Figure 4 (mergesortFP). This benchmark suite uses only a base \text{Par} monad, not the ParST transformer. It is also the only one of these benchmarks that completely stops scaling before twelve cores! Indeed, when sorting arrays larger than the last-level cache, mergingSort reads the entire input memory at least \( \log_2(N) \) times, greatly increasing memory traffic \[22\].

Eliminating the copying by using ParST causes scaling to continue to twelve cores. We look at two variations of this in Figure 5. Naturally, all these implementations of merge sort bottom out to sequential sorts below a granularity threshold. The two variants we examine bottom out to different sequential sorts: either (1) a pure Haskell sequential sort, or (2) a library call to a C sort (namely, the same sequential sort used by the Cilk implementation). The table in Figure 5 contains the times for the all-Haskell sort. It achieves a 10.7× parallel speedup on 12 cores. The line graph above it shows the other variant, alongside the DPJ and Cilk benchmarks. Our parallel version does add overhead relative to Cilk, with a best time of 0.42 instead of 0.29 seconds.

\[22\] Performing a multi-way merge sort could reduce the impact of this problem.

---

**Figure 4.** Benchmark suite of traditional parallel kernels in the LVish Par monad. These are runnable either with monad-par or LVish.

**Figure 5.** Non-copying merge sort. Parallel speedups shown relative to the Cilk single-thread execution time of 3.48 seconds.
accesses to shared state are disjoint [3]. In addition to the models already discussed, here we contrast our work on extending LVish with non-language-based approaches, in particular, those that attempt to run arbitrary threaded programs deterministically. The narrowest form of deterministic parallelism is repeatability: the property that, on a specific machine, whatever happens the first time a program is run will also happen on subsequent runs, given the same inputs. For example, the TERN system [6] uses a schedule memoization approach to improve debuggability by repeating the same thread interleavings as previous runs. Also of interest is consistent scheduling on a particular input. The recent work on DThreads [14] transparently converts multi-threaded programs into multi-process ones, enforcing a deterministic resolution of conflicting updates to memory. DThreads intercepts the pthreads API to hook into arbitrary programs.

While supporting legacy software makes this line of research very important, there are major differences between the approach taken by systems like DThreads, and that taken by LVish:

- LVish requires no reasoning about interleavings. Deterministic threading packages make thread interleavings a consistent behavior, but the programmer still needs to think about concurrency, given that they will not generally be able to predict the exact schedule chosen by the deterministic scheduling package. In LVish, all lattice-based actions commute, so interleavings are not relevant.
- Deterministic threading packages typically support lock-based, multi-threaded programs, but cannot handle other forms of synchronization based on user-space atomic memory operations—in particular, lock-free data structures such as those that underlie modern work-stealing runtime systems. By contrast, LVish is specifically focused on enabling the programmer to use fine-grained concurrent data structures.
- A language-based approach can ensure determinism by statically limiting what features can be combined (effects, transformers), rather than by runtime enforcement that carries a runtime overhead.

9. Conclusion

We present an extended version of the LVish library for deterministic parallelism, augmented with the ability to manage a wide variety of effects previously not seen in combination in any guaranteed-deterministic parallel programming system. Our extended library offers the well-known benefits of language-level enforcement of determinism, but without being limited to a single shared data structure or a single programming paradigm as previous deterministic-by-construction programming models have been. Furthermore, our case study and empirical results demonstrate that deterministic parallelism can be effective, while also retaining the ease of use that is the hallmark of deterministic parallel models.

Acknowledgments

Thanks to Aaron Turon for many illuminating conversations that helped develop the ideas in this paper, and to the anonymous PLDI reviewers for their insightful and helpful comments. This research was funded in part by NSF grant CCF-1218375.

References

A. Using LVish: two brief examples

The LVish library that we have described in this paper is available on [Hackage](https://hackage.haskell.org/package/lvish), the Haskell package repository. To install it, run:

```
$ cabal install 'lvish >= 2.0'
```

Once LVish is installed, you can compile and run the LVish programs from this paper, and many more.[4] The following simple example demonstrates the threshold read semantics of LVars. In this example, cart is an LVar representing a shopping cart to which Items, such as a Book or Shoes, can be added.

```haskell
import Control.LVish; import Data.LVar.PureMap

data Item = Book deriving (Ord, Eq)

p :: (HasPut e, HasGet e) => Par e s Int
p = do
  cart ← newEmptyMap
  fork (insert Book 2 cart)
  fork (insert Shoes 1 cart)
  getKey Book cart

main = print (runPar p)
```

Running this program deterministically prints 2. The two forked operations run asynchronously and in arbitrary order; the call `getKey` in `Book cart` is a blocking threshold read, and will block until the operation `insert` in `Book cart` has occurred.

This example also demonstrates a number of other features of LVish. First, `p` is a `Par` computation parameterized by an effect level with the constraints `HasPut` and `HasGet`, indicating that `p` may perform LVar writes and reads. Second, running a `Par` computation with `runPar` produces a pure result. LVish also provides a `runParIO` function for running `Par` computations that return results in the IO monad. Finally, this example demonstrates one of the many built-in data structures provided by LVish—a key-value `Map`. All of these data structures work with the rest of the LVish infrastructure without any additional effort on the programmer's part.

The following example demonstrates two more features of LVish: `handlers`, which are callbacks run every time the contents of an LVar change, and the `runParThenFreeze` operation, which freezes an LVar on the way out of a computation, allowing the exact contents of the LVar to be read in a deterministic fashion. Here, `traverse` is a function that performs a breadth-first traversal of a graph `g` starting from a given node `startNode` and finds all the nodes reachable from `startNode`.

```haskell
traverse :: HasPut e ⇒ G.Graph → Par e s (ISet s Int)
traverse g startNode = do
  seen ← newEmptySet
  h ← newHandler seen
  -- Callback to be run whenever a new node appears in the "seen" set.
  -- (Anode n m v) means n is in the seen set.
  (Anode n m v) -> m neighbors n h
  return ()
  insert v seen
  return seen

main = print (runParThenFreeze
  (traverse myGraph (0 :: G.Vertex)))
```

B. Repeating our results

In addition to the LVish library, we also provide the means for others to re-run our experiments. Infrastructure for the benchmarks in this paper, in addition to the source code of our library and further instructions, is available in the following GitHub repository:


With a checkout of that repository, and assuming GHC 7.6.3 and Cabal 1.18, the command

```
$ make everything
```

will compile the library and benchmarks and run them in a slightly-reduced configuration from the paper. Doing so will produce three primary outputs, for each of the three benchmarks from our paper.

- For PhyBin, presented in Section 7.1, the results are available in the phybin_results.txt file. These results can be regenerated with `make phybin_bench`. Note that this benchmarks PHYBin, but not the other systems we compare with.
- For the evaluation of transformer overhead, presented in Section 7.2, the results are found in the transformer_results.txt file. To regenerate just these results, run `make transformer_bench`.
- For parallel merge sort, presented in Section 7.3, the results are available in two text files, hs_mergesort_results.txt and c_mergesort_results.txt. The first of these shows the performance of merge sort with the leaf sequential sort implemented in Haskell; the latter with the leaf sequential sort implemented in C. These results can be regenerated with `make mergesort_bench`. Again, note that this does not benchmark the other systems we compare with.

For all of these benchmarks, the Makefile automatically runs up to four-core versions. The mergesort_bench_large target will run the full versions for merge sort; for the others, slight modifications to the Makefile will be needed. Finally, note that ongoing benchmarking of the LVish development repository uses a different mechanism: the HSBencher package and run_benchmarks.hs file, which upload data to a Google Fusion Table.[6]

Running our benchmarks in a pre-built environment

Our primary tool for making it easy to re-run our code and benchmarks is the Docker container tool, which provides lightweight virtualization on Linux systems. With Docker, you can automatically run our full suite of benchmarks with a pre-built version of GHC by running:

```
$ docker pull iuparfunc/pldi2014-artifact
$ docker run -e USER=pldi -i -t \
  iuparfunc/pldi2014-artifact:build /bin/bash
```

Then follow the instructions above—all of the files are in the pldi2014-artifact directory inside the Docker container. You can even automatically compile and run the benchmarks in the Docker environment with a single command.

```
$ docker build -t pldi2014-artifact \
  github.com/iu-parfunc/pldi2014-artifact
```

Then you can see the results by running the following command and looking at the generated files described above.

```
$ docker run -i -t pldi2014-artifact /bin/bash
```

---


[26] Available at http://docker.io