Taming the Parallel Effect Zoo

Extensible Deterministic Parallelism with LVish

Abstract

A fundamental challenge of parallel programming is to ensure that the observable outcome of a program will not change due to parallel execution, except to run faster—that is, that programs execute deterministically. 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 *LVars* programming model, and the accompanying *LVish* Haskell library, took a step toward solving the first of these problems. LVish allows parallel 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. It provides a `Par` monad for parallel computation that allows monotonicity-preserving effects, guaranteeing determinism while allowing for a more flexible form of communication between parallel tasks than previous guaranteed-deterministic models provided.

We show that LVish and the `Par` monad also address the extensibility problem. While applying LVar-based programming to real problems, we have identified and implemented three capabilities that extend its reach: monotonically increasing 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—without suffering added complexity or cost in the core LVish implementation, or compromising determinism—is a form of *monad transformer*, extended to handle the `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.

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[1] 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 [7], data parallelism [5, 16], and languages with advanced permissions systems that prevent data races [3]. 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 these models. Indeed, many applications do not fit into a single paradigm, which is what most deterministic parallel models, such as synchronous dataflow, 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 [7]), the only effects possible within a stream filter are `push()` and `pop()` operations on a linear stream data structure. Deterministic data-parallel languages [5, 16], on the other hand, typically do not enable 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.

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.

Yet extending the breadth of a deterministic language is not easy. The built-in parallel idioms 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 [3, 4, 8, 10], 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.

LVars: a step forward A significant step in the direction of broadly-applicable deterministic parallelism was the recent proposal of *LVars* by Kuper *et al*. [10, 11] 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 location 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. Unfortunately, while LVars are already far more

[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.

expressive than the write-once *IVars* [1] of prior work on deterministic parallelism, 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 3). We make use of this capability to parallelize PhyBin [14], 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). This provides the first integration of parallel updates to mutable memory (à la Deterministic Parallel Java [3]) with blocking data-flow 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. Moreover, LVish can be 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, as well as enabling PhyBin to perform $3.35\times$ better.

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 [10, 11] offers a principled approach to guaranteed-deterministic parallel programming with shared state. In this section we review LVars and *LVish*, a Haskell library that implements the LVars programming model; Sections 3-6 then describe our extensions to LVish.

An *LVar* is a memory location that can be shared among multiple threads. Unlike an ordinary shared mutable variable, 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 form a *lattice*[2] specific to that LVar, and the semantics of reads and writes is defined in terms of this lattice of states. The two fundamental LVar operations are `put`, for reading, and `get`, for writing. At a high level:

- A `put` operation can only change an LVar's state in a way that is *monotonically increasing* (or *inflationary*) with respect to its lattice. Informally, the contents of an LVar must stay the same or "grow bigger" with each write. This is guaranteed to be the case because `put` takes the *least upper bound* (lub) of the current state and the new state with respect to the lattice.
- A `get` operation allows only limited observations of the state of an LVar. The key idea is that reads from an LVar are *threshold reads*: the semantics of a `get` operation can be expressed by specifying a *threshold set* of minimum values that can be read from the LVar, where every two elements in the threshold set must have the lattice's greatest element $\top$ as their lub. A `get` call blocks until the LVar in question reaches a (unique) value in the threshold set, then unblocks and returns that value.

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 LVar 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 `put` and `get`.[3]

The LVish library *LVish* is an implementation of the LVars programming model as a monadic library in Haskell. The LVish library provides a `Par` monad for encapsulating parallel computation, and enables a notion of lightweight, library-level threads to be employed with a custom work-stealing scheduler.[4] LVish also provides a variety of LVar-based data types (*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-based data structures (although note the proof obligations for data structure implementors, below). LVar-based data structures 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-based 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.

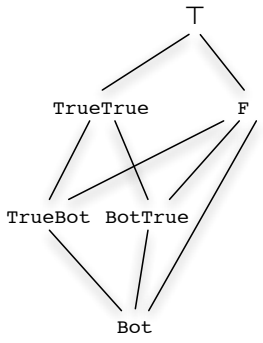


Figure 1. Lattice of states that an `AndVar` can take on.

An example: parallel "and" Consider an LVar that stores the result of a parallel logical "and" operation. Two threads contribute boolean values to the final result, which will be `True` if both threads

[2] Formally, the lattice of states is given as a 4-tuple $(D, \sqsubseteq, \bot, \top)$ where D is a set, $\sqsubseteq$ is a partial order on D, $\bot$ is D's least element according to $\sqsubseteq$, 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 $(D, \sqsubseteq, \bot, \top)$ is really a *bounded join-semilattice* with a designated greatest element ($\top$). For brevity, we use the term "lattice" as a shorthand.

[3] In practice, it is also important to be able to register latent *event handlers* that fire when `puts` occur, but these are equivalent to an implicit set of functions blocked on `gets`.

[4] LVish is available at `http://hackage.haskell.org/package/lvish`. It generalizes the original `Par` monad exposed by the *monad-par* library (`http://hackage.haskell.org/package/monad-par`), which allowed determinism-preserving communication between threads using *IVars*—single-assignment variables with blocking read semantics. IVars are a special case of LVars, corresponding to a lattice with one "empty" and multiple "full" states, where $\forall i.\ empty < full_i$.

contribute `True`, but `False` if either thread contributes `False`. The states that such an LVar can take on form the lattice shown in Figure 1. The `Bot` state is the least element of the lattice and represents the state of the LVar before any writes have taken place. Any write of `False` takes the state of the LVar to `F`, while a write of `True` will take the state of the LVar to one of the two intermediate states `BotTrue` or `TrueBot`, where it waits for the other write to occur. In LVish, we can define the type of this LVar as follows:

```
type AndVar = PureLVar AndState
data AndState = Bot | TrueBot | BotTrue | TrueTrue | F
```

To express the ordering in Figure 1, we also need to implement a *join* (lub) operation for `AndState`:

```
instance JoinSemiLattice AndState where
  join x y | x == y = x
  join TrueBot BotTrue = TrueTrue
  join TrueTrue TrueBot = TrueTrue
  join TrueTrue BotTrue = TrueTrue
  join Bot x = x
  join F  _  = F
  join x y = join y x
```

The greatest element $\top$ in Figure 1 corresponds to the situation in which conflicting writes have occurred (in which case LVish will raise an exception), but it need not appear in the definition of `AndVar` and should be thought of as a semantic modeling tool only.

Threshold reads from `AndVar`s Under what circumstances can we deterministically read from an `AndVar`? According to the semantics of threshold reads discussed above, the states `TrueTrue` and `F` together comprise a valid threshold set for an `AndVar`. Intuitively, this corresponds to the idea that when either `TrueTrue` or `F` is reached, there is no more information to be learned about the `AndVar`'s contents: if two writes of `True` have arrived, then we are done, and if at least one write of `False` has arrived, no further writes will matter. The `getAnd` operation, written in terms of the `getPureLVar` function provided by LVish, takes an LVar and a list of states that comprise the states of the threshold set. The call to `getPureLVar` blocks until one of `TrueTrue` or `F` is reached, and then returns that state.

```
getAnd :: AndVar → Par AndState
getAnd avr = getPureLVar avr [TrueTrue, F]
```

Putting it together We are now ready to implement the `asyncAnd` function, which takes two `Par` computations, each of which produces a boolean value, and returns a `Par` computation that produces a boolean. The `asyncAnd` function creates a new `AndVar avr` and forks two threads, each of which uses the `putPureLVar` function provided by LVish to write to `avr`. A `getAnd` call on `avr` blocks until one of the two threshold states has been reached, and `asyncAnd` returns `True` or `False` depending on whether the reached threshold state was `TrueTrue` or `F`.

```
asyncAnd :: Par Bool → Par Bool → Par Bool
asyncAnd m1 m2 = do
  avr ← newPureLVar Bot
  fork (do b1 ← m1
           putPureLVar avr (if b1 then TrueBot else F))
  fork (do b2 ← m2
           putPureLVar avr (if b2 then BotTrue else F))
  x ← getAnd avr
  return (x == TrueTrue)
```

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.

In the case of `AndVar`, the `join` function can be trivially and exhaustively verified to compute a lub. For example, one can verify the totality of `join` by evaluating the following list comprehension in GHCi:

```
[ join x y | x ← [Bot,TrueBot .. F]
           , y ← [Bot,TrueBot .. F] ]
```

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 `AndVar` LVar just described makes it possible to do a "short-circuit" computation because `getAnd` unblocks and returns a result as soon as any `False` is written. However, it is still the case that the other thread writing to the `AndVar` 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 6.

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

In the previously existing LVish model, the only way for the state of an LVar to evolve over time is through a series of least-upper-bound 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 LVar 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. We will see one example of an application that critically requires this functionality in Section 7.1.

In this section, 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 $(D, \sqsubseteq, \bot, \top)$, a data structure author may define a set of *bump* operations $\texttt{bump}_i : D \to D$, which must meet the following two conditions:

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

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 no single LVar may use *both* `put` and `bump`. For example, a `put` of 4 and a $\texttt{bump}_{(+1)}$ 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 $\texttt{bump}_{(+1)}$ updates it to 5. But if the $\texttt{bump}_{(+1)}$ 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, a `(Set Counter)` contains a monotonically growing set of mutable counters that monotonically increase. Indeed, the PhyBin application (Section 7.1) uses a distance matrix of counters.

Determinism guarantee The proof of determinism for LVish [10] relies on the fact that the states of all LVars evolve monotonically with respect to their lattices. Monotonic growth is sufficient 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 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 $\mathcal{P}(G) \rightarrow \mathcal{P}(P) \times \mathcal{P}(B)$. That is, adding more `get` results on the left results only in *more* `put`/`bump` effects on the right (that is, 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 5. 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 [11]. Adding support for operations like `bump` makes this assumption untenable. Therefore, we re-engineered the LVish runtime system to include additional synchronization.[5]

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`. To that end, 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, a `Par Int` computation becomes, instead, `Par e Int`, where `e` is a type-level encoding indicating whether or not writes, reads, or non-idempotent (`bump`) or non-deterministic (`IO`) operations are allowed to run inside it.

To enable future extensibility of this effect "switchboard", we follow the precedent of recent work by Kiselyov *et al*. on extensible effects in Haskell [9]: we abstract away the specific structure of `e` into constraints. For example, a `Par` computation annotated with the effect level constraint `HasPut` can perform `puts`. More concretely, the signature for a `putIVar` operation, which takes an IVar and a value to be written to it and returns a `Par` computation, might be written as follows:

```
putIVar :: HasPut e ⇒ IVar a → a → Par e ()
```

while the signature for an `incrCounter` operation might use the `HasBump` constraint:

```
incrCounter :: HasBump e ⇒ Counter → Par e ()
```

These constraints can also be *negative*; for instance, a Par computation annotated with the `ReadOnly` constraint can perform `gets`, but not `puts` or `bumps`. In Section 6.1, we will see that computations supporting task cancellation must have the `ReadOnly` constraint.

4. `Par` monad transformers

The effect-tracking system of the previous section give us a way to toggle on and off a fixed set of basic capabilities using the type system. 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`, `return`) and satisfy the three monad laws. However, each monad must also provide other operations that make it worth using. Most famously, the `IO` 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.

We have already discussed the `Par` monad in which LVar effects run; now we define it explicitly. A *`Par` monad* is any monad that has, at minimum, a `fork` operation. Programs with `fork` create a binary tree of monadic actions with `()` (unit) return values.

```
class (Monad m) ⇒ ParMonad m where
  fork :: m () → m ()
```

The original LVish library provides a single, concrete `Par` type. We generalize this by introducing the above class, together with classes capturing the interfaces to basic parallel data structures and control constructs such as futures, `IVars`, and `Maps`. For example, the class `ParIVar` provides `new` and `put` methods with the signatures below.[6]

```
-- Slightly simplified IVar class:
class (ParMonad m) ⇒ ParIVar m  where
  new  :: m (IVar m a)
  put  :: IVar m a → a → m ()
```

Taken together, this framework for generic `Par` programming makes it possible for LVish to define *`Par` monad transformers* (or `Par` transformers for short) that add plug-in capabilities for any type implementing the core `Par` interface (just as the set of types satisfying `Monad` is closed under monad transformer application).

Example: threading state in parallel Perhaps the simplest example of a `Par` transformer is the standard `StateT` transformer (provided by Haskell's `Control.Monad.State` package). However, for `StateT s m` to also be a `Par` monad, the state must be *splittable*; that is, it must be specified what is to be done with the state at `fork` points in the control flow. For example, the state may be duplicated, split, or otherwise updated to note the fork. Here are two immediately useful applications of threaded state:

[5] 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.

[6] Although it may appear that generic treatment of `Par` monads as type variables `m` removes the additional metadata in a type such as `Par e a`, note that it is possible to recover this information with type-level functions.

- `PedigreeT` keeps the index in the binary control-flow tree as implicit state, *e.g.*, "LRRLL". This is sometimes called the *pedigree* of the parallel computation [12]. 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 [12]. In LVish, no such runtime system modification is necessary: instead, we add `StateT` as a plug-in capability on top of `Par`, and only track pedigree for applications that need it. (Section 7.2 discusses the overhead of `Par` transformers.) We define the `StateT` transformer as follows:

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

instance (SplittableState s, ParMonad m) ⇒
         ParMonad (StateT s m) where
  fork task =
     do s ← get   -- monadic function of no arguments
        let (s1,s2) = splitState s
        put s2
        lift (fork (do runStateT task s1; return ()))
```

Note that here, `put` and `get` are not LVar operations, but the standard procedures for setting and retrieving the state in a `StateT`.

Determinism guarantee The `StateT` transformer preserves determinism because it is effectively *syntactic sugar*. 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 `IO`. We return to `ST` in Section 5.

The case for pluggability Why should `Par` 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 to be enormously complicated low-level concurrent code. 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 if capabilities are 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 and our extensions to it. The framework presented thus far enables new (trusted) packages to add transformers that preserve the ability to use core date structures; that is, they provide instances for the classes `ParMonad`, `ParFuture`, and `ParIVar`. And, as was already possible in LVish, 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`) and the scheduler are limited. Thus the connection between the scheduler and the data structure can be severed and replaced with a generic interface that we call `LVarSched`. This interface captures low-level actions and callbacks. For example, to perform a `get` on an arbitrary LVar, callback actions are registered that check if new updates to the data structure allow the `get` to unblock.

With this approach, a generic monotonic `Map` package can work with any scheduler implementation, including those produced by stacking transformers that were written with *no knowledge of the data structure*. Likewise, transformers that implement `LVarSched`[7] 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 are monotonic. 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.

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 different parts of the same structure (for instance, two halves of an array) from different threads. `ParST` exploits the fact that it is perfectly safe to do so, as long as the memory accessed by different threads is disjoint. 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.

```
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)
```

[7] We retain interfaces like `ParIVar` 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. For example, some of our applications can run either on LVish or on the native GHC work-stealing runtime (for which we expose only the `ParFuture` interface).

```
    (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. Such an idiom might be useful in, for instance, an in-place parallel sort (see Section 7.3).

We ensure the safety of `ParST` with two key provisions:

- **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,[8] 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 the 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 parallel merge sort 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_1), *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 our extended LVish, users do not populate the state directly, but only describe a *recipe* for its creation. Each type used as a 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 in-place 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 `commuteswith` 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 LVar-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 that can combine 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 [3].[9]

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 and 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:

```
(do m1; m2)  ==  (do fork m1; m2)
```

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.[10] 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

[8] That is, the type of a child computation begins with `(∀ s . ParST ...)`.

[9] There is a minor caveat here. DPJ requires that the type system statically determine disjointness of state updates, whereas in LVish, we 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.

[10] To be precise, `ParST` does provide a `ParMonad` instance, but any attempt to reference the state in a forked computation results in a runtime error.

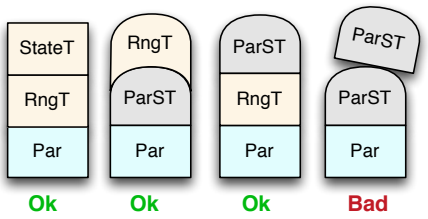


Figure 2. Monad transformer stacks. `Par` monads can either be reordering-tolerant (flat top) or not (round top). Transformers based on `StateT` preserve what is below them. In our current library, `ParST` is the only transformer, that *removes* the reordering-tolerance of the stack below it.

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. Rather, `ParST` is a one-time option to turn on the "`ST` switch" for base `Par` monads. 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`). For this reason, `ParST` is restricted to only apply directly to a `Par` monad with zero or more permitted transformers on top. For example, `StateT` and `RngT` transformers are permitted, as shown in Figure 2.

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 hood[11] to be able to support work-stealing scheduling and blocking `get`s. 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 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 computation 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 enable to provide a `CancelT` transformer providing operations such as `forkCancelable`, which takes a computation as argument and runs it in parallel, returning a `ThreadId`, and `cancel`, which takes a `ThreadId` and cancels the thread associated with it and all of that thread's subthreads, transitively. In our generic framework, the signatures for `forkCancelable` and `cancel` are:

```
forkCancelable :: (ParMonad m, LVarSched m) ⇒
                  CancelT m () → CancelT m ThreadId
cancel :: (ParMonad m) ⇒ ThreadId → CancelT m ()
```

This notion of cancellation allows us to write a version of the `asyncAnd` function from Section 2 which, when it reaches the `F` state, calls `cancel` to terminate any remaining (now useless) computations.

Note that `forkCancelable`, and only `forkCancelable`, enforces a restriction that the forked computation must be `ReadOnly`.[12] Normal (non-cancellable) forks, even in the `CancelT` monad, may have arbitrary effects. In fact, a variant of `forkCancelable` allows cancellation of threads with arbitrary effects *but* requires nondeterminism (that is, `IO`) in its effect level constraint.

Implementation The `CancelT` transformer allocates one mutable location whenever a new `ThreadId` 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 `ThreadId`s, 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 `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 LVar *memo table*. 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:

```
type Memo e k v = (ISet k, IMap k 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 doing a blocking `get` on the map. This provides an efficient way to memoize functions—even functions that have side effects within the `Par` monad (*i.e.*, `makeMemo` takes a function `(a → Par e s b)`). It is a great application of the previously existing LVish library. But a further synergy with `CancelT` is possible. The `Memo` type above has an `e` parameter that tracks the effect constraint of the memoized function. However, making a memo table request means writing an ele-

[11] That is, a `ParIVar` is always also a `Cont` monad.

[12] 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 their parent, which is necessary when threads perform side effects like `put` that may throw exceptions that appear deterministically. A `ReadOnly` computation, on the other hand, is like a future with no visible effect but its result, and we require that exceptions not be propagated to the parent until/unless the future is read.

ment 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 the `put` effect in the result signature:

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

Internally, within our trusted library, we use an unsafe type coercion to remove the `put` effect. Then, with the above `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 several 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 "sealed" 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. `ParIVar` 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 an in-place parallel merge sort benchmark. All measurements come from a dual-socket 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[13] 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 $N \times N$ 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 [15], introduced by a software package of the same name.[14] HashRF is about 2-3$\times$ 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.*, Phylip[15], DendroPy[16]). These slower packages use $\frac{N^2-N}{2}$ full applications of the distance metric, which has poor locality in that it reads all trees in from memory $\frac{N^2-N}{2}$ 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 `''AB|CDE''`, while another would be `''ABC|DE''`. Identical trees, of course, convert to the same set of bipartitions. Furthermore, after converting trees to sets of bipartitions, set difference may be computed using standard set data structures.

The HashRF algorithm makes use of this fact and adds a clever trick that greatly improves locality. Before computing the actual distances between trees, it populates a table mapping each observed bipartition to the set of trees that contain it. In the original PhyBin source:

```
type BipTable = Map DenseLabelSet (Set TreeID)
```

Above, a `DenseLabelSet` encodes an individual bipartition as a bit vector. PhyBin uses purely functional data structures for the `Map` and `Set` types, whereas HashRF uses a mutable hash table. Yet in both cases, these structures *grow monotonically* during execution. The full algorithm for computing the distance matrix is:

```
global: biptable, distmat
(1) for t ∈ alltrees:
      for bip ∈ t:
        insert(biptable, (t, bip))
(2) for (_, trset) ∈ biptable:
      for t1 ∈ alltrees:
        for t2 ∈ alltrees:
          if t1 ∈ trset `xor` t2 ∈ trset
          then increment(distmat[t1,t2])
```

The second phase of the algorithm is still $O(N^2)$, but it only needs to read from the much smaller `trset` during this phase. All loops above are potentially parallel.

Parallelization The LVish methodology applies directly to this application:

- The `biptable` in the first phase is a map of sets, which are directly replaced by their LVar counterparts.
- The `distmat` in the second phase is a vector of monotonic `bump` counters.

In fact, the parallel port of PhyBin using LVish was so straightforward that, after reading the code, parallelizing the first phase took only 29 minutes.[17] Once the second phase was ported, the distance computation sped up by a factor of 3.35$\times$ on 8 cores (Table 1).

[13] `http://hackage.haskell.org/package/phybin`

[14] `https://code.google.com/p/hashrf/`

[15] `http://evolution.genetics.washington.edu/phylip.html`

[16] `http://pythonhosted.org/DendroPy/`

[17] Git commit range will be linked here in non-blinded version.

Trees	Species	PhyBin			DendroPy	Phylip
100	150	0.269			22.1	12.8
		PhyBin 1, 2, 4, 8 core				HashRF
1000	150	4.7	3	1.9	1.4	1.7

Table 1. PhyBin performance comparison with DendroPy, Phylip, and HashRF. All times in seconds.

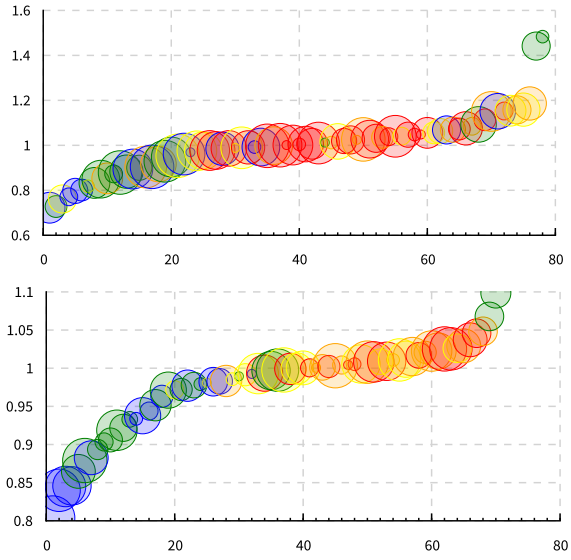


Figure 3. The overhead of adding one `StateT` transformer (top) or `ParST` transformer (bottom). Each color represents one of the benchmarks drawn from Figure 4. For each benchmark, there is a different bubble per thread setting, with the area proportional to the number of threads. We do *not* see a trend with more or less overhead at larger numbers of threads. All times are the median of five runs.

This is exactly where we would like to use LVish—to achieve modest speedups for modest effort, in programs with complicated data structures (and high allocation rates), and *without* changing the determinism guarantee of the original functional code.

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 PhyBin, or, in Kuper *et al.*'s work, k-CFA [11]. Nevertheless, here we also include a benchmark suite of traditional parallel kernels shown in Figure 4. We use these in Figure 3 as well, which summarizes the overhead added when rerunning 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: in-place 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`. Being an in-place sort, the `mergeSort` function assumes a vector state, and takes no other arguments:

```
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 (the merge phase is out-of-place), 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)
  (do forkSTSplit (sz2,sz2) mergeSort mergeSort
      mergeL2R)
mergeR2L
```

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 `s` 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 `Int` → `Int`) with a guarantee that no visible side effects occur.

Performance The first performance problem we encounter with merge sort is seen in Figure 4. This benchmark suite is using only a base `Par` monad, not the `ParST` transformer; thus the merge sort listed there is an out-of-place sort. It is also the only one of these benchmarks that completely stops scaling before 12 cores! Indeed, when sorting arrays larger than the last-level cache, this merge sort algorithm is not ideal (multi-way merge sort is better); but we expect that problem to be greatly *exacerbated* by copying these large arrays.

Eliminating the copying by using `ParST` causes scaling to continue to 12 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 $10.7\times$ 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.

8. Related Work

Work on deterministic parallel programming models is long-standing. As we discussed in Section 1, deterministic parallel languages must restrict effects so that schedule nondeterminism cannot be observed—whether that means avoiding shared mutable state entirely, as in data-parallel languages [5, 16], allowing sharing only by a limited form of message passing, as in dataflow-based or stream processing languages [4, 7, 8], or ensuring that concurrent 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.

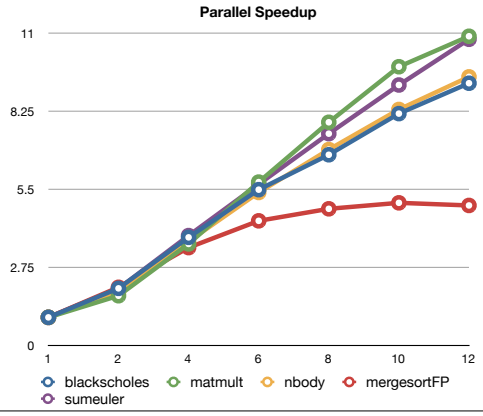
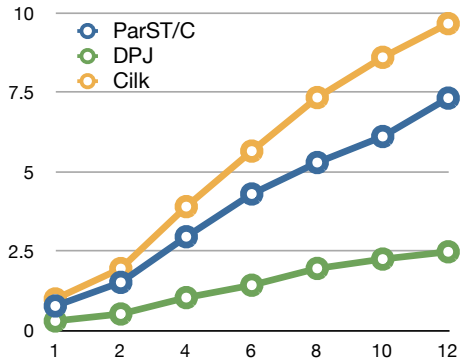


Figure 4. Benchmark suite of traditional parallel kernels in `Par`.



Threads	1	2	4	6	8	10	12
ParST/HSonly	36.5	18.0	9.2	6.3	4.8	4.6	3.4

Figure 5. In-place, merge sort. Parallel speedups shown relative to the Cilk single-thread execution time of 3.48 seconds.

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 [13] 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 now performs competitively with systems with fewer guarantees, while maintaining the ease of use that is the hallmark of deterministic parallel models.

References

[1] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11, October 1989.

[2] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, New York, NY, USA, 2012. ACM.

[3] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009.

[4] 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. *Sci. Program.*, 18, August 2010.

[5] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP*, 2011.

[6] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, 2010.

[7] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS*, 2002.

[8] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*. North Holland, Amsterdam, Aug 1974.

[9] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Haskell*, 2013.

[10] L. Kuper and R. R. Newton. LVars: lattice-based data structures for deterministic parallelism. In *FHPC*, 2013.

[11] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *POPL*, 2014.

[12] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *PPoPP*, 2012.

[13] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *SOSP*, 2011.

[14] R. R. Newton and I. L. Newton. PhyBin: binning trees by topology. *PeerJ*, 1:e187, 10 2013.

[15] S.-J. Sul and T. L. Williams. A randomized algorithm for comparing sets of phylogenetic trees. In *APBC*, 2007.

[16] V. Weinberg. Data-parallel programming with Intel Array Building Blocks (ArBB). *CoRR*, abs/1211.1581, 2012.