Seminar: Programming models and Code Generation
Winter Term 2014/2015

Quasi deterministic parallel programming using LVars

Nishanth Nagendra
Technische Universität München

15.01.2015

Abstract

A *deterministic program* always produces the same observable result on multiple runs and a *deterministic by construction parallel programming model* allows only such programs to be written offering programmers freedom from hard to reproduce non deterministic bugs. This report talks about *Lvish*, a *quasi deterministic parallel programming model* that extends *LVars*, its predecessor, which is a deterministic model. LVars generalize existing single assignment models by allowing communication through shared monotonic data structures with monotonic write operations and threshold read operations that block until a lower bound is reached. The semantics are defined in terms of an application specified lattice. LVish extends LVar by allowing to freeze them in order to read their exact contents and to attach event handlers to an LVar triggering a callback when their state changes. It offers quasi determinism in the worst case which means on every run it either produces the same result or raises an error. The LVish model yields promising parallel speedup on benchmarking applications using a prototype implementation of a library in haskell.

1 Introduction

The purpose of a deterministic-by-construction programming model is to rule out programs like the following from being written, or force them to behave deterministically. [1]Written in a hypothetical parallel language.

```
let _ = put lv 3 in   /write 3 in lv/
let par v = get lv   /read from lv/
    _ = put lv 4   /write 4 in lv/
        in v   /final value is v/
```

The *let par* expression, evaluates the two subexpressions in parallel. The value of v is the value of the entire program. Depending on the scheduler *get lv* or *put lv*4 can run first, v might be either 3 or 4 during multiple runs giving rise to non determinism.

A popular approach for enforcing determinism is to require that variables can only be assigned to once, giving us a single-assignment language. Our program would be invalid because it tries to write to *lv* twice. Single-assignment variables with blocking read semantics, often known as IVars, turn up in all kinds of deterministic parallel systems. But the single-assignment rule disallows a number of programs that we would like to permit such as writing the same value to an IVar twice. As another example, consider the problem of filling in various parts of a data structure in parallel. [1]Say that *lv* contains a pair, and we are writing to *lv* twice, with each write filling in one component of the pair.

```
let par   _ = put l (4, )
          _ = put l (, 3) /error/
          in get l
```

Read from *lv* is possible once both writes have completed, so it will always be (4, 3). But single-assignment rules out this program(although we could fake it by using a distinct IVar for each component of the pair, but with more sophisticated data structures it becomes difficult to fake a solution with IVars.)

There is a need to generalize the single-assignment model to admits programs like these but retains determinism. Deterministic algorithms must be expressible in a style that guarantees determinism, and non-deterministic behaviors, where desired, must be requested explicitly. Enforcing deterministic semantics simplifies composing, porting, reasoning about, debugging, and testing parallel software. Existing models tend to lack features that would make them applicable to a broader range of problems. Moreover, they lack extensibility.

Existing models[2] A number of models currently exist as a part of the long running effort towards deterministic by construction parallel programming.

- *No-shared-state parallelism*: Pure functional programming in Haskell with par and pseq combinators is an example. It uses function level task parallelism or futures. This kind of parallelism does not allow for any shared mutable state between tasks, forcing tasks to produce values independently. Models based on pure data parallelism like *Data parallel Haskell, River Trail API for Java script* also fall in this category,

- *Data-flow parallelism*: Kahn process networks, snychronous data flow systems are examples. In these models, tasks can only communicate via channels which are FIFO queues where the read operation on these queues is blocking until data is available. Stream-processing languages such as *StreamIt* are based on KPN.,

- *Single-assignment parallelism*: IVars are an example which are memory locations that can be either full or empty and can only be assigned once. Reads are blocking for IVars. They have been used in the *monad-par* Haskell library, *Intel CnC(Concurrent collections, Concurrent ML(as SnycVars)*.,

- *Imperative-disjoint parallelism*: The state accessed by concurrent threads are disjoint. *Deterministic Parallel Java* falls under this category.

A common theme that emerges in the study of deterministic-by-construction systems is the notion of monotonicity. Many deterministic parallel programs cannot be expressed with these models. Sharply restricted communication and synchronization capabilities have consequences not only for the immediate usability of guaranteed-deterministic languages, but for the extensibility of those languages as well.

Motivating Example Consider applications in which independent sub-computations contribute information to shared data structures that are unordered, irregular, or application-specific. Following is an example that highlights the issues of ordering which create a friction between parallelism and determinism.

In a directed graph where v represents a particular node, we wish to find all (or the first k degrees of) node connected to v, then analyze each node in that set. Existing parallel solutions[2] often use a non-deterministic traversal of the connected component even though the final connected component is deterministic. For example, IVars cannot accumulate sets of visited nodes, nor can they be used as mark bits on visited nodes, since they can only be written once and not tested for emptiness. Streams, on the other hand, impose an excessively strict ordering for computing the unordered set of vertex labels in a connected component. A purely functional attempt is given in the original paper[4] where connected-component discovery is parallel, but members of the output set do not become available to other computations until component discovery is finished, limiting parallelism. An LVars[4] based programming model can possibly allow us to write a breadth first search traversal where

threads communicate using shared monotonic data structures. Consumers of the data structure may execute as soon as data is available, but may only observe irrevocable, monotonic properties of it.

2 LVars and LVish

2.1 LVars

An LVar is a memory location that can be shared between multiple threads whose contents can only grow bigger over time, for some definition of bigger that the user of the LVar gets to specify.

LVars, short for lattice variables, are a generalization of IVars, which are variables that can only be written to once. (The I is for immutable.) LVars, on the other hand, allow multiple writes, as long as those writes are monotonically increasing with respect to a user-specified lattice. Meanwhile, LVar reads have to specify a threshold set of possible states. A read will block until one of the states in the threshold set has been reached, and then return that state. Information can never be removed and the order in which it is added is not observable. IVars turn out to be a special case of LVars. The property of monotonically increasing write operations is enforced by the language. The semantics of *put* says that a write to a location is the least upper bound of the new value and the current value.

```
let par _ = put num 3
        _ = put num 2
  in get num /always 3/
```

Monotonic writes alone aren't enough to ensure deterministic programs. The above program is nondeterministic where num's notion of bigger is ¿=. Despite writes always being monotonically increasing. Get might read either 2 or 3, depending on the order of *putorget* operations determined by the scheduler.

- To maintain determinism, *get* operation is restricted: instead of reading the LVars exact value, we read one of a set of lower bounds on its value that are given as an extra argument to *get*. The *get* operation will block until the LVar reaches a value that is at or above one of those lower bounds. For the above example, the threshold set will just be {3}. *Get* operation is called a threshold read, because the value it returns is the threshold we have crossed.

- Once num's value it is at least 3, it will remain at or above 3 forever. As long as we only access them through the *putorget* interface, LVars are thread-safe: we can safely share them between threads without introducing unintentional non-determinism. That is, *puts* and *gets* can happen in any order, without changing the value that a program evaluates to.

The resulting model is general enough to subsume the existing single-assignment model. The user-specified set just needs to be a *join-semilattice*, which is a partially ordered set in which every two elements have a least upper bound.

To describe Lvar formally, a parallel call-by-value λ-calculus named λ_{LVar} is introduced in the paper[4] which is extended with a *store* and communication primitives *put* and *get* that operate on data in the *store*. The definition of λ_{LVar} is parameterized by the choice of lattice and therefore λ_{LVar} is actually a family of languages, rather than a single language. An LVar may have an arbitrary number of states forming a set D, which is partially ordered by a relation $\sqsubseteq$. Formally, D is a *bounded-join-semilattice* augmented with a greatest element $\top$.

- D has a least element $\bot$, representing the initial empty state of an Lvar,

- D has a greatest element $\top$, representing the error state that results from conflicting updates to an Lvar,

- D comes equipped with a partial order $\sqsubseteq$, where $\bot \sqsubseteq d \sqsubseteq \top$ for all $d \in D$.

- Every pair of elements in D has a least upper bound (lub) written $\sqcup$ which means it is

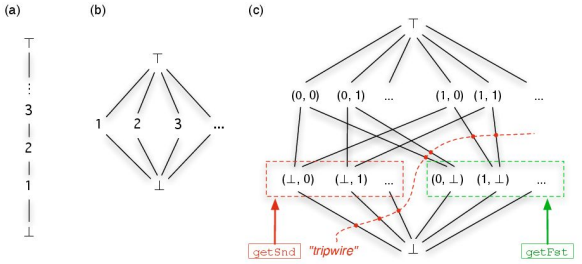

Figure 1: Example lattices[4]: (a) Positive integers ordered by $\leq$ (b) IVar containing a natural number (c) Pair of natural number-valued IVars

possible for two sub-computations to independently update an LVar, and then deterministically merge the results by taking the lub of the resulting two states.

Data structures like arrays, tree, maps , streams etc. can be represented as lattice. *Figure* 2 gives three examples of lattices for common data structures.

During the evaluation of a λ_{LVar} program, a *store* S keeps track of the states of LVars. Each LVar is represented by a *binding* from a location l, drawn from a countable set *Loc*, to its *state*, which is some element $d \in D$. Although each LVar in a program has its own state, the states of all the LVars are drawn from the same lattice D. We can do this with no loss of generality because lattices corresponding to different types of LVars could always be union ed into a single lattice (with shared $\bot$ and $\top$ Elements).

Definition 1: A *store* is either a finite partial mapping $S : Loc \overset{\text{fin}}{\to} (D - \{\top\})$, or the distinguished element $\top_S$.

The state space of *stores* forms a *bounded-join-semilattice* augmented with a greatest element, just as D does, with the empty store $\bot_S$ as its least element and $\top_S$ as its greatest element. The $\sqsubseteq$ and $\sqcup$ operations defined on elements of D can be lifted to the level of stores.

Definition 2: A store S is *less than or equal to* a store S' (written $S \sqsubseteq_S S'$) iff:

- $S' = \top_S$, or
- $dom(S) \subseteq dom(S')$ and for all $l \in dom(S)$, $S(l) \sqsubseteq S'(l)$.

Definition 3: The lub of stores S_1 and S_2 (written $S_1 \sqcup_S S_2$) is defined as follows:

- $S_1 \sqcup_S S_2 = \top_S$ iff there exists some $l \in dom(S_1) \cap dom(S_2)$ such that $S_1(l) \sqcup S_2(l) = \top$.
- Otherwise, $S_1 \sqcup_S S_2$ is the store S such that:
  - $dom(S) = dom(S_1) \cup dom(S_2)$, and
  - For all $l \in dom(S)$:

$$S(l) = \begin{cases} S_1(l) \sqcup S_2(l) & \text{if } l \in dom(S_1) \cap dom(S_2) \\ S_1(l) & \text{if } l \notin dom(S_2) \\ S_2(l) & \text{if } l \notin dom(S_1) \end{cases}$$

By Definition 3:, if $d_1 \sqcup d_2 = \top$, then $[l \mapsto d_1] \sqcup_S [l \mapsto d_2] = \top_S$. Notice that a store containing a binding $l \mapsto \top$ can never arise during the execution of a λ_{LVar} program, because an attempted write that would take the state of l to $\top$ would raise an error before the write can occur.

Communication Primitives The new, put, and get operations create, write to, and read from LVars, respectively.

- *new* extends the store with a binding for a new LVar whose initial state is $\bot$, and returns the location l of that LVar (i.e. a pointer to the LVar).
- *put* takes a pointer to an LVar and a singleton set containing a new state and updates the LVars state to the least upper bound of the current state and the new state, potentially pushing the state of the LVar upward in the lattice. Any update that would take the state of an LVar to $\top$ results in an error.

$$\frac{\{p\}\ C\ \{q\}}{\{p * r\}\ C\ \{q * r\}}$$

Figure 2: Standard frame rule.

- *get* performs a blocking threshold read. It takes a pointer to an Lvar and a *threshold set* Q, which is a non-empty subset of D that is *pairwise incompatible*, meaning that the lub of any two distinct elements in Q is $\top$. If the LVar's state d_1 in the lattice is *at or above* some $d_2 \in Q$, the *get* operation unblocks and returns d_2. Note that d_2 is a unique element of Q, for if there is another $d_2' \neq d_2$ in the threshold set such that $d_2' \sqsubseteq d_1$, it would follow that $d_2 \sqcup d_2' \sqsubseteq d_1$, which contradicts the requirement that Q be pairwise incompatible.

For Lambda Lvar syntax and semantics, remaining definitions and its details please refer[4].

Proof of determinism: The key to the determinism is the frame rule given in [**?**]. A frame property captures the idea of local reasoning about programs that alter state[1]. Here, C is a program, and $\{p\}C\{q\}$ says that if the assertion p is true before C runs, then the assertion q will be true afterwards. For example, p and q might describe the state of the heap before and after it is updated by C. It tells that running C starting from a state satisfying the assertion $p \quad r$ will result in a state satisfying the assertion $q \quad r$. The assertion $p * r$ is satisfied by a heap if the heap can be split into two non-overlapping parts satisfying p and r, respectively.

For LVars, the frame property says that independent effects commute with each other. Consider an expression e that runs starting in *store* S and steps to the expression e, updating the *store* to S. Frame property guarantees that if e starts from a larger *store*$S \sqcup R$, where R is some other store *framing on* to S, then e will update the *store* to $S \sqcup R$, and e will step to e. The *frame store* R is a *store* resulting from some other independently-running computation. S and R (or S and R) need not be disjoint we are combining them with $\sqcup$, not with . If they did have to be disjoint, it would mean that concurrent operations cannot touch the same parts of the store. For LVars, this kind of total disjointness is unnecessary, since updates commute! Instead, we can behave as though S and R are disjoint, even though they might not be.

Diamond Lemma: This does the heavy lifting of the determinism proof: it establishes the diamond property, which says that if a configuration steps to two different configurations, there exists a single third configuration to which those configurations both step. For supporting lemmas, its details and complete proofs please refer to[4]. Lemmas there also show how cases of deadlocks and livelocks are also handled.

The LVars model guarantees determinism by monotonic write operations[which can commute] and threshold read operations. But it is not as general-purpose as one might hope. Consider an unordered graph traversal. A typical implementation involves a monotonically growing set of *seen nodes*; neighbors of seen nodes are fed back into the set until it reaches a fixed point. These are not expressible using the threshold read and least-upper-bound write operations described above. The problem is that these computations rely on negative information about a monotonic data structure, i.e., on the absence of certain writes to the data structure. In a graph traversal, neighboring nodes should only be explored if the current node is not yet in the set; a fixpoint is reached only if no new neighbors are found; and at the end of the computation it must be possible to learn exactly which nodes were reachable. But in the LVars model, asking whether a node is in a set means waiting until the node is in the set, and it is not clear how to lift this restriction while retaining determinism.

2.2 LVish

In this regard, Lvish[3] extends the Lvars model with 2 additional capabilities. First, it adds event

handlers, a mechanism for attaching a callback function to an LVar that runs, asynchronously, whenever events arrive (in the form of monotonic updates to the Lvar). Second, a primitive for freezing an LVar, which means: once an LVar is frozen, any further writes will throw an exception; on the other hand, it becomes possible to discover the exact value of the LVar, learning both positive and negative information about it, without blocking.

Freezing does not commute with writes. If a freeze is interleaved before such a write, the write will raise an exception; if it is interleaved afterwards, the program will proceed normally. Although it appears like determinism is lost, these programs satisfy the property of Quasi determinism that is they will always output the same result or an error.

$$\texttt{addHandler}\ lv\ \{1,3,5,\dots\}\ (\lambda x.\,\texttt{put}\ lv\ x+1) \quad (1)$$

This registers a handler for lv that executes the callback function $\lambda x.put\ lv\ x+1$ for each odd number that lv is at or above. Event sets are a mathematical modeling tool only. Event handlers in LVish invoke their callback for all events in their event set Q that have taken place even if those events occurred prior to the handler being registered. If a callback only executed for events that arrived after its handler was registered, or only for the largest event in its handler set that had occurred, then it could result in non-determinism. For example if puts are interleaved with handler registration as a part of a *par* expression then a *get* operation outside would block, or not, depending on how the handler registration was interleaved with the *puts*. By instead executing a handlers callback once for each and every element in its event set below or at the LVars value, quasi-determinism is guaranteed.

Quiescence through handler pools: Event handlers are asynchronous, so a separate mechanism is needed to determine when they have reached a quiescent state, i.e., when all callbacks for the events that have occurred have finished running. Detecting quiescence is crucial for implementing fixpoint computations. To build flexible data-flow networks, it is also helpful to be able to detect quiescence of multiple handlers simultaneously. This design includes handler pools, which are groups of event handlers whose collective quiescence can be tested.

```
let h = newPool
in addInPool h lv Q f;
quiesce h
```

The quiesce operation blocks until all of the callbacks running in a particular handler pool, in this case h, are done running. where lv is an LVar, Q is an event set, and f is a callback. Handler pools are created with the *newPool* function, and handlers are registered with *addInPool*. Quiescence of a handler is a non-monotonic property: it can move in and out of quiescence as more puts to an LVar occur. There is no risk to quasi-determinism, however, because quiesce does not yield any information about which events have been handled, any such questions must be asked through Lvar *get*.

The freeze operation is a non-blocking read that lets us find out the exact contents of an LVar. After freezing, any further writes that would change LVar's state will raise an exception! A safe practice is to quiece before freezing For a program that performs freezes, Only two possible outcomes are guaranteed: the deterministic result of all the effects, or a write-after-freeze exception that can assist in debugging the synchronization bug. This property is called quasi-determinism, and the Lvish model is quasi-deterministic.

Freezing introduces quasi-determinism to the programming model, because we might freeze before we have quiesced, or because our thread that's freezing and quiescing might be racing with some other thread to write into a shared LVar. But, if it is ensured that freezes only ever happen after all writes have completed, then the computations will be deterministic, because there wouldn't be any write-after-freeze races. This is enforced at the implementation level which takes care of quiescing and freezing at the end of a computation. In the LVish library, this is provided by the runParThenFreeze function.

[3]contains a quasi-deterministic, parallel, call-by-value λ_{LVish}-calculus extended with a store containing LVars. It extends the original LVar formalism to support event handlers and freezing. Rather than modeling the full ensemble of event handlers, handler pools, quiescence, and freezing as separate primitives, it instead formalizes the *freeze-after* patternwhich is a simpler primitive combining all others.

An LVars state is now a pair (d, frz), where d is an element of the application-specific set D and frz(Short for freeze) is a status bit of either $true$ or $false$. We define an ordering $\sqsubseteq_p$ on LVar states (d, frz) in terms of the application-specific ordering $\sqsubseteq$ on elements of D. Every element of D is "freezable" except $\top$. Informally:

- Two unfrozen states are ordered according to the application-specific $\sqsubseteq$; that is, $(d, false) \sqsubseteq_p (d', false)$ exactly when $d \sqsubseteq d'$.
- Two frozen states do not have an order, unless they are equal: $(d, true) \sqsubseteq_p (d', true)$ exactly when $d = d'$.
- An unfrozen state $(d, false)$ is less than or equal to a frozen state $(d', true)$ exactly when $d \sqsubseteq d'$.
- A frozen state is less than an unfrozen state if the unfrozen state is $\top$; that is, $(d, true) \sqsubseteq_p (d', false)$ exactly when $d' = \top$.

The addition of status bits to the application-specific lattice results in a new lattice $(D_p, \sqsubseteq_p, \bot_p, \top_p)$. $\sqcup_p$ for the lub operation that $\sqsubseteq_p$ induces. During the evaluation of LVish programs, a *store* S keeps track of the states of LVars and represents them by a binding from a location l, drawn from a set *loc*, to its state, which is some pair (d, frz) from the set D_p. For detailed syntax and semantics of Lvish please refer to the paper[3].

Communication primitives: They are same as Lvar except that new will now create a binding for a new Lvar whose initial state is ($\bot$, false). *Put* takes a pointer to the Lvar and a new lattice element which is a pair. *Get* similarly takes a threshold set containing pairs. $freeze\ e_{lv}\ after\ e_{events}\ with\ e_{cb}$ has the following semantics: It attaches the callback e_{cb} to the LVar e_{lv}. The expression e_{events} must evaluate to a event set Q; the callback will be executed, once, for each lattice element in Q that the LVars state reaches or surpasses. The callback takes a lattice element as its argument. Its return value is ignored, so it runs solely for effect. For instance, a callback might itself do a put to the LVar to which it is attached, triggering yet more callbacks. If the handler reaches a quiescent state, the LVar e_{lv} is frozen, and its exact state is returned (rather than an under-approximation of the state, as with *get*).

To keep track of the running callbacks, LVish includes an auxiliary form of the above pattern taking additional arguments: the set of running callbacks B and a set of values H in the event set for which callbacks have already been launched. It detects quiescence by checking: Every event of interest that has occurred must be handled (be in H). Second, all existing callback threads must have completed and terminated with a value. If yes, then LVars state is freezed. To ensure quiescence is a permanent state, rather than a transient one, freezing is usually the very last step of an algorithm, permitting its result to be extracted. The $runParThenFreeze$ function inside LVish library does this, and thereby guarantees full determinism.

Proof of Quasi determinism for Lvish: According to the paper[3] the main theorem says that if two executions starting from a configuration σ terminate in configurations σ' and σ'', then either σ' and σ'' are the same configuration (up to a permutation on locations), or one of them is *error*.

Independence property: Like LVars, the LVish model uses the frame property again to capture the idea that independent effects commute with each other but this time with some restrictions. If we know that a *store* S evaluating expression e steps to a new *store* S' with e' then it is valid to say that $S \sqcup_S S''$ evaluating e steps to a new store $S' \sqcup_S S''$

with e' if the following conditions are satisfied:

- The status bits of $S' \sqcup_S S''$ and S agree with each other for all locations shared between them.
- Locations in S'' cannot share the same name with locations newly allocated during the transition from one configuration to another.

Please refer to the original paper and the technical report for elaborate details on lemmas and their proofs.

3 Implementation and Performance

A prototype implementation of Lvish is available as a monadic library in Haskell at:

`http://hackage.haskell.org/package/lvish`

It adopts the basic approach of the par monad. The existing Par monad[6] in Haskell offers an API for parallel programming. It provides by default a work-stealing scheduler and supports forking tasks. The programmer specifies how information flows across computations but not the order in which they will be evaluated at runtime. Information flow is described using IVars. With LVish we get a lattice-generic infrastructure: the Par monad itself, a thread scheduler, support for blocking and signaling threads, handler pools, and event handlers. This infrastructure does not guarantee quasi-determinism, only data structure authors should import it who will implement a specific monotonic data structure using this library providing efficient parallel access, exporting a limited interface for its users ensuring quasi-determinism. LVish library provides its own custom scheduler and programs run only inside the par monad allowing only Lvish-sanctioned side effects to provide compile time guarantees of determinism and quasi determinism.

In order to specify the type of the par computation it provides a phantom type that indicates the determinism level.

```
data Determinism = Det | QuasiDet
```

Par type constructor:

```
Par : : Determinism -> * -> *
```

following suite of run functions

```
runPar : : Par Det a -> a
runParIO : : Par l v l a -> IO a
runParThenFreeze : : Deep frz a =>
        Par Det a -> FrzType a
```

If the Lvish code uses freeze then it is marked as QuasiDet or Det if it is fully deterministic. *runparIO* allows us to execute Lvish code in the IO monad if the determinism level is arbitrary. *runparthenFreeze*s automatically freezes the Lvar when it returns back its exact value guaranteeing determinism.

Key Ideas: *Atoms and Idempotence:* Atoms are elements not equal to $\perp$ but whose only smaller element is $\perp$. Lattices for which every element is the lub of some set of atoms are called atomistic, and most application-specific lattices used by LVish programs have this property and the corresponding data structure usually exposes operations that work at the atom level, semantically limiting puts to atoms, gets to threshold sets of atoms, and event sets to sets of atoms. A compact way to represent a change to the lattice is deltas allowing to easily and efficiently communicate such changes between *puts* and *gets*/handlers. For a set, a delta is an element; for a map, a key/value pair.

Idempotence, meaning that $d \sqcup d = d$ for any element d. Repeated puts or freezes have no effect, if the scheduler is allowed to occasionally duplicate work, it is possible to save on synchronization costs. Since LVish computations are guaranteed to be idempotent, we could use such a scheduler (standard Chase-Lev deque[**?**]). Idempotence helps to deal with races between *put* and $get/addHandler$.

Internally par monad represents computations

in continuation passing style, in terms of their interpretation in the IO monad. The status bit of an LVar is tied together with a bag of waiting listeners, which include blocked gets and handlers.

```
data Status d = Frozen |
        Active (B.Bag(Listener d))
data LVar a d = LVar{state :: a,
        status :: IORef(Status d)}
```

The bag module supports atomic insertion and removal, and concurrent traversal.

```
put :: Bag a -> a -> IO(Token a)
remove :: Token a -> IO()
foreach :: Bag a ->
   (a -> Token a -> IO()) -> IO()
```

A listener for an LVar is a pair of callbacks, one called when the LVars lattice value changes, and the other when the LVar is frozen.

```
data Listener d = Listener{
   onUpd :: d -> Token(Listener d)
   -> SchedQ -> IO(),
onFrz :: Token( Listener d)
   -> SchedQ -> IO() g
```

Internally, the Par monad represents computations in continuation-passing style, in terms of their interpretation in the IO monad.

```
type ClosedPar = SchedQ -> IO()
type ParCont a = a -> ClosedPar
mkPar :: (ParCont a -> ClosedPar)
   -> Par l v l a
```

The *ClosedPar* type represents ready-to-run *Par* computations, which are given direct access to the CPU-local scheduler queue. Rather than returning a final result, a completed *ClosedPar* computation must call the scheduler, sched, on the queue. A *Par* computation, on the other hand, completes by passing its intended result to its continuation yielding a *ClosedPar* computation.

```
data HandlerPool = HandlerPool{
numCallbacks :: Counter,
blocked :: B.Bag ClosedPar}
```

The counter tracks the number of currently executing callbacks, which is used to implement quiesce. A bag of threads that are blocked waiting for the pool to reach a quiescent state. Please refer to[7] for more details on the library functions including *quiesce*, *addhandler* etc.

Communication primitives:

- *getLV* provides *get* semantics. It takes a global and a delta threshold function as arguments.
- *putLV* provides *put* semantics. Takes as argument an update function that performs *put* on the underlying data structure. In situations where it races with a $freeze$, the *put* happens first and then the LVar freeze status is checked. To ensure correctness in such race situations,in addition to checking status bit, a mark bit is used in each CPU scheduler state.
- $freezeLV$ provides freeze semantics. Takes an LVar as argument.

Evaluation: k-CFA study In [3] a case study on k-CFA (parallelizing control flow analysis) is done to evaluate the performance of the Lvish implementation. K-CFA provide a hierarchy of increasingly precise methods to compute the control flow of a program or precisely the flow of values to expressions. The figure on the following page shows performance data[7] for LVish on 2 benchmarks. One is the blur benchmark based on k=2. The implementation using LVish for the 2 benchmarks uses lock-free data structures. Another synthetic benchmark is notChain which is simply a chain of 300 not functions that negates the benifit of the sharing approach in the LVish model.

Related work: In [5], feasibility of LVar threshold reads is explored in a distributed setting. Distributed systems involve replication of data objects across several locations making them robust to data loss and give better data locality. Eventual consistency is a policy in which the updates eventually reach the replicas and the replicas need

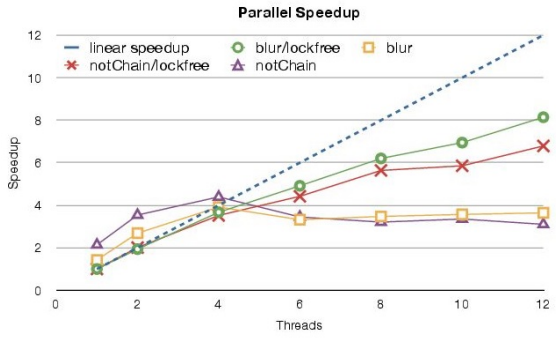

Figure 3: Performance of LVish on parallelizing k-CFA[3]

not agree at all times. Convergent replicated data types provide a simple mathematical framework for reasoning about and enforcing the eventual consistency of such replicated objects, based on viewing them as elements of a lattice and replica conflict resolution as the lattice's join operation. CvRDTs are provably eventually consistent, but queries of CvRDTs allow inconsistent intermediate states of replicas to be observed. LVar-style threshold queries to CvRDTs extend them with support for deterministic queries and show that they are strongly consistent queries. A threshold query that returns an answer when executed on a replica will return the same answer every subsequent time, and on a different replica, it will eventually return the same answer, otherwise will block until it does so. It is therefore impossible to observe different results from the same threshold query, whether at different times on the same replica, or whether on different replicas.

4 Summary

In an attempt towards creating better deterministic parallel programming models, Lvish has extended the existing usage of monotonicity as a theme to provide better applicability, usability, performance, providing a single generic model that subsumes many of the single assignment models, thereby representing a larger class of problems which can be effciently parallelized. The performance with regards to certain benchmark applications are an evidence of the same. [7]discusses the viability of Lvish for certain graph algorithms, its drawbacks and proposed extension. Some of the drawbacks to highlight are the fact that Lvish side effects are only monotonic updates and if a data structure needs to be updated multiple times it will have to be done out of place thereby frequently allocating LVars becoming a source of additional memory allocation and ineffcient relative to in-place versions. Blocking *get* semantics could slow down until the dependency is met reducing the performance, hence instead of blocking, the tasks could be retried at a later time/rescheduled when there dependency has been satisfied. The overhead in going through the scheduler many times has also been highlighted in[4] for applications like Maximum independent set. The paper proposes a solution similar to an abort/retry feature.

References

[1] Lindsey Kuper. Blog. composition.al.

[2] Lindsey Kuper. Lattice-based data structures for deterministic parallel and distributed programming. *Dissertation*, 2014.

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

[4] Kuper L. and Newton R. Lattice-based data structures for deterministic parallelism. *FHPC*, September 2013.

[5] Kuper L. and Newton R. Deterministic threshold queries of distributed data structures. *Draft*, July 2014.

[6] Simon Marlow. *Parallel and Concurrent programming in Haskell*. O'Reily, 2013.

[7] Praveen Narayanan and Ryan R. Newton. Graph algorithms in a guaranteed deterministic language. *Workshop on Deterministic and Correctness in Parallel programming (WoDet 2014)*, March 2014.