Graph Algorithms in a Guaranteed-Deterministic Language

Praveen Narayanan  
Indiana University  
pravnar@indiana.edu

Ryan R. Newton  
Indiana University  
rrnewton@gmail.com

Abstract
Deterministic implementations of graph algorithms have recently been shown to be reasonably performant. In this paper we explore a follow-on question: can deterministic graph algorithms be expressed in guaranteed-deterministic parallel languages, which are necessarily restrictive in what concurrency idioms they employ? To find out, we implement several graph algorithms using the LVish library for Haskell (a deterministic language), which reveals its strengths as well as limitations. We surmount these limitations by (1) implementing a functional version of the deterministic reservations mechanism, and (2) adding a new mechanism to LVish called BulkRetry. We present results from an early-stage prototype.

1. Introduction
Graph algorithms, which analyze collections of nodes connected by edges, are important to many areas of computing. Indeed, their importance is increasing, in part driven by the availability of large graph datasets found in social networks and elsewhere. Parallelizing graph algorithms has been a priority but is complicated by the fact that graph algorithms—such as breadth-first search (BFS), maximal independent set (MIS), or minimum spanning forest (MSF)—expose irregular parallelism mirroring the irregular structure of graphs themselves. Nevertheless, Blelloch et al., in a 2012 paper, demonstrate reasonably good parallel performance on a selection of graph algorithms in spite of using internally deterministic algorithms. This determinism property has a number of benefits, including that multiple machines can execute the same algorithm with identical outcomes, unlocking possibilities for redundancy, failure recovery, or optimizations that trade re-execution against storage or network communication. Also, as had been widely recognized, deterministic systems are easier to debug during development.

Yet while the value of deterministic algorithms is widely recognized, how does one arrive at an implementation of a deterministic algorithm? The answer is usually “by writing code in a general purpose, nondeterministic language”. That is, very carefully. Still, languages and libraries can make it easier or more difficult for developers to reach their determinism goal. For example, in previous work, the authors provide a toolbox of parallel collection operations such as scan, reduce, and filter. Using these operations it becomes easier to write deterministic code—at least, easier than when using lower-level primitives like threads and locks directly.

In this paper we explore a more extreme proposition: implementing graph algorithms in a guaranteed deterministic language. Such a language goes beyond encouraging deterministic code, making it impossible to write nondeterministic code. Code written in such a language may be functionally incorrect, but the compiler guarantees that it cannot violate the determinism of the system. By contrast, while using the primitives in, the user must still be trusted to follow a number of rules to ensure determinism. For example, certain operations[1] are allowed in parallel with one another while others are not—a mistake or oversight in following the protocol can result in nondeterminism of the entire application. We say that code written against an unsafe API of this sort is trusted code.

One potential criticism of a deterministic-language approach is that when considering a complete software system consisting of compiler, runtime, libraries, and application code, there will always be a mix of trusted and untrusted code. This is true not only with respect to determinism guarantees, but any language-level guarantee such as type and memory safety: there must be trusted code somewhere inside the implementation. Still, there is a qualitative difference between unsafe programming environments, where every new project involves new trusted code, and language-based approaches in which the trusted code base should be small and constant.

With the goal of minimizing the code that must be trusted to ensure determinism, the approach we take in our work is to begin with a deterministic host language, Haskell[2] and combine it with a parallel programming library called LVish[9]. LVish sanctions certain side-effects on shared, mutable data structures as safe to use with existing functional applications. In this paper, we study how well-suited the LVish abstractions are to supporting graph algorithms, and describe extensions (Section 6) to the library that make it a better tool for efficiently implementing graph algorithms. We make the following contributions:

- We examine the parallel programming idioms used to unsafely implement three graph algorithms (BFS, MIS, MSF), comparing them to the simpler (blocking, dataflow) operations provided by LVish.
- We measure the performance of existing LVish primitives applied to graph algorithms, finding them wanting.
- We adapt the deterministic reservations mechanism to a functional, and guaranteed-deterministic setting.
- We propose an extension to LVish, the BulkRetry primitive, to execute graph algorithms more efficiently while retaining the semantics of the simplest versions of the algorithms.

---

[1] An example would be the find and link operations on the disjoint sets datatype in [1].

[2] Haskell caveats: programs that throw exceptions may not uniquely constrain which exception they throw; instead it can be compiler dependent. Also, while determinism holds for pure computations, it is possible to access a full imperative language by running computations within Haskell. Fortunately, the type system enforces a separation, which makes it possible, for example, to compile and run untrusted code received over the network, enforcing that it does not perform IO or induce nondeterminism [13].
2. Background: LVars and LVish

Broadly speaking, deterministic languages use one or more of the following strategies to ensure determinism:

1. Restrict communication between parallel code regions (e.g. functional and dataflow paradigms).
2. Enforce permissions on who has access to shared data at each point in the program (e.g. Deterministic Parallel Java [2]).
3. Restrict writes to shared data to operations that commute.
4. Ensure threads proceed in lock-step (PRAM).

The LVish library supports a combination of options 1-3. It includes basic mechanism for dynamic creation of parallel tasks: i.e., \texttt{fork} (\texttt{t x}) will enqueue the computation \texttt{t x} for execution on some processor. All communication between tasks happens through \texttt{LVars}, concurrent data structures whose state space can be described by a join-semilattice, and where the state of each LVar may only evolve monotonically within that lattice. The simplest LVars are dataflow variables called IVars, which may be either full or empty, and support a blocking get operation. For example, with IVars, it is straightforward to replace \texttt{fork}—which does not return any values—with a notion of futures:

```haskell
import Data.LVar.IVar as IV

spawn comp =
  do vr ← IV.new
  -- Do the work in the child thread:
  fork (do res ← comp)
    IV.put vr res
  -- Return an IVar that serves as the future:
  return vr
```

More complicated LVars include concurrent data structures for sets, bags, finite maps, vectors, and so on, typically implemented using lock-free atomic memory operations. The existing LVish library includes a collection of such structures.

Proofs of LVish’s determinism rest on the fact that operations on LVars commute. Modifications \texttt{add} information, never subtract it, and observations of the state of LVars, when they return, yield an answer that is invariant under subsequent growth of the LVars’s state. We elide detailed explanation of the programming model here, referring the reader instead to previous work [8, 9]. However, one more programming primitive is necessary for our purposes.

**Handlers**

Handlers in LVish are callbacks spawned every time an LVар changes. Though spawned on demand, they are equivalent to an infinite number of \texttt{fork}’ed computations blocked and waiting on distinct gets. For example, it becomes simple to discover a connected component in a graph by seeding a \texttt{set LVar} with a start vertex, and using a handler to react to all new vertices by adding their neighbors:

```haskell
connected :: Graph → NodeID → Par (ISet NodeID)
connected g startV = do
  seen ← newEmptySet
  addHandler seen (\nd → parMapN (putInSet seen) (nbrs g nd))
  putInSet seen startV
  return seen
```

The handler function (the \(\lambda\) expression) is called only when the LVар changes. Repeated insertions of the same \texttt{NodeID} will not cause redundant handler invocations. The \texttt{Par} return type of the function above is a \texttt{monadic} type that indicates the computation performs parallelism side effects. Yet unlike the \texttt{IO} monad, the \texttt{Par} monad \texttt{Par a → a} can be used within purely functional code by using \texttt{runPar}, which has type \texttt{Par a → a}. This is possible because \texttt{Par} computations are deterministic.

3. Deterministic Graph Algorithms

We focus on three fundamental graph problems in this paper:

- **Breadth-first search**: Given a graph \(G = (V, E)\) and a source vertex \(s \in V\), return a BFS tree with root \(s\) and nodes from \(V\).
- **Maximal independent set**: Given a graph \(G = (V, E)\), return a set \(S \subseteq V\) such that no two vertices in \(S\) are neighbors, and all vertices not in \(S\) have at least one neighbor in \(S\).
- **Spanning forest**: Given an undirected graph \(G = (V, E)\), return a collection \(F \subseteq E\) such that \(\forall i, \exists T_i \in F\) where \(T_i\) is a spanning tree for \(C_i\), a connected component of \(G\).

**Tie-breaking**

All three of these algorithms may have multiple valid answers for a given input graph, and thus nondeterministic answers are often acceptable. To achieve a deterministic result, tie-breaking between valid answers is required—if there are two valid BFS tree topologies, we must select one consistently. This is often accomplished by allowing vertex IDs to serve as \texttt{priorities}, for example, when selecting which of two valid parents to choose in a BFS tree, we can guarantee we select the higher priority parent, at the expense of some extra work.

Without specifying a particular implementation strategy, we may still observe that certain primitives are required for parallel implementations of these algorithms:

- **Reduction variables or prioritized writes**: To record the highest-priority parent of a particular vertex, or to maintain an intermediate state of a growing spanning forest, we need reduction variables that support commutative operations.
- **Parallel schedules respecting to data-dependent priorities**: The parallelism in these algorithms mirrors the structure of the graphs; we need a way to process vertices in parallel and yet ensure that higher-priority vertices or edges take their turn first. We will see that this can be accomplished either with \texttt{blocking} dataflow-style communication, or with other mechanisms that achieve the same effect (Section 4.3).

In the next section, we present simple and safe formulations of graph algorithms using LVars to provide the above capabilities. In Section 5 we review how these algorithms have been implemented efficiently and deterministically in previous work [1], unsafe APIs and primitives. Finally, in Section 6 we will attempt to reconcile the two, improving the efficiency of the LVar-based implementations.

4. Graph Algorithms in LVish

In this section we show how to apply existing LVish primitives to implement the algorithms BFS and MIS. Then, we demonstrate how the \texttt{deterministic reservation} concept can be implemented using existing LVish primitives, gaining guaranteed determinism but losing space efficiency.

4.1 Breadth-first search trees

We already showed a simple example of a \texttt{connected component} computation, which performs an asynchronous, parallel depth-first traversal of a graph. Here we show a modification of the algorithm to return a full BFS tree topology, as in [1], rather than merely the set of node IDs in the connected component.

As mentioned in Section 3 we need to use a reduction variable to keep track of only the minimum or maximum \texttt{NodeID} that could
be a parent for a given node: that is, a MinVar instead of an IVar
(both being instances of IVars).

import Data.LVar.MinVar

type NodeID = Int

bfsTree :: Graph → NodeID → Par (Vector MinVar)
bfsTree gr start = do
    -- Decompose into vertex and edge vectors:
    let (Graph verts edges) = gr
    -- A new array with a MinVar for each vertex:
    parents ← Vector.generateM (length verts)
    (ix → newMinVar maxInt)
    seen ← newEmptySet
    let handler nd = mapM (eachNbr nd) (nbrs g nd)
        eachNbr nd nbr = do putInSet seen nbr
            -- Write ‘nd’ to the MinVar:
            putMin parents nbr nd
        addHandler seen handler -- Register callback.
    seen ← putInSet seen start -- Kick things off.
    return parents

Each MinVar in the parents vector begins initialized to maxInt and
will be overwritten with a smaller number if it is part of the con-
nected component. As before, the handler attached to seen drives
the computation. Note, however, that LVish programs are gener-
ally asynchronous, and non-blocking where possible. For example,
the bfsTree function above returns immediately, not waiting for the
cascade of handler invocations to complete. Rather, a synchroniza-
tion operation is required to wait on spawned tasks to complete
and freeze the final value of the parents array above (see [9]).

4.2 Maximal independent set

In LVish, our implementation of the maximal independent set algo-
rithm closely matches the greedy algorithm that is usually de-
scribed as follows: iterate over the vertices of the graph and grow
the independent set by adding a vertex to it only if none of its
(higher-priority) neighbors have been already included. Because
processing a vertex depends only on higher-priority neighbors, cy-
cles in the graph do not cause cycles in loop iteration dependencies.

Here is the LVish code, which uses an IStructure datatype
equivalent to an array of IVars:

data Flag = Chosen | NbrChosen

mis :: Graph → Par (IStructure Flag)
mis gr = do
    let (Graph verts edges) = gr
    flagsArr ← newIStructure (length verts)
    parFor (0, length verts) (λ ix → do
        let checkNbr acc nbr =
            do nbfFlag ← get flagsArr nbr
                case nbfFlag of
                Chosen → do put flagsArr ix NbrChosen
                        return True
                _ → return acc
            -- Only if none of the nbrs are chosen can we be:
            nbf_chosen ← foldM checkNbr False (nbrs g ix)
            unless nbf_chosen
                (put flagsArr ix Chosen))
        return flagsArr

Each vertex is assigned one of three flags representing the sta-
tus of the vertex with respect to the computed independent set:
Chosen, NbrChosen, or undecided, with undecided represented,
not explicitly, but by an empty slot in the IStructure. The output of
the LVish program is an IStructure Flag, i.e., an array of IVars,
each containing a flag and corresponding to a single vertex. IVars
are the very simplest form of LVars that are either empty or full,
and contain at most a single value. In LVish, IVars allow for multi-
ple puts of the same value, and also provide a get operation that
blocks when the IVar is empty.

The algorithm uses the flags IStructure as the shared data
structure among the iterates, and all iterates execute in parallel
using parFor. The loop inside each iterate is an idiomatic fold1 over
the neighbors of the corresponding vertex. The result of this is a,
boolean called nbf_chosen indicating whether any of the neighbors
is already in the independent set. If this variable returns false, the
iterate uses a put to asynchronously update the vertex status to
Chosen. The folded function (checkNbr) behaves as follows: get the
neighbor’s status, and if it is Chosen then asynchronously update
the vertex status to NbrChosen and return True, else pass along the
value of the accumulator (called chosen in the code).

Note that the blocking get causes iterates to wait as higher
priority neighbors update the flag of their associated vertex. Also
note that while checkNbr may issue multiple put calls for the same
vertex, this is allowed by the LVish implementation of the IVars
since they will all be setting the status to the same value, i.e.,
NbrChosen.

4.3 Minimum spanning forest

Minimum spanning forest provides an interesting example where
LVish is not a good fit out of the box. This is in contrast with
BFS and MIS, where blocking gets provide exactly the required
synchronization, and monotonc updates are the only concurrent
writes. To implement MSF, we needed to add a new library func-
tion based on a technique introduced in [11]. We describe that mech-
anism here before proceeding.

Deterministic Reservations are a method of parallelizing iterative
greedy algorithms introduced by Blelloch et al.. A C++/Cilk
implementation is available as part of the Problem-Based Bench-
mark Suite (PBBS). The approach focuses on the deterministic
parallel processing of loops with loop-carried dependencies. The
idea is to allow the user to break the iteration space into rounds,
where each round selects a deterministic subsets of available ite-
rates which have no conflicting reads or writes. No attempt is
made to automatically detect conflicts, rather they are declared and
checked by the user.

The main algorithm takes in a sequence of iterates and consists of
a series of rounds. In each round, a predetermined prefix of the
iterates is processed in two phases, each of which are parallel loops
over the prefix. The first phase runs the reserve routine of each
iterate in the prefix, which resolves data conflicts by reserving the
access to shared data through (deterministic) priorities. The second
phase computes the commit component of each iterate, processing
those iterates for which the reservations succeeded. The failed
iterates, along with new iterates, are part of the next prefix for the
subsequent round, and the algorithm thus proceeds until all iterates
have been exhausted.

The formulation of deterministic reservations in LVish is some-
what different, because arbitrary side effects are not permitted in
parallel. Instead, it takes the form of a supercombinator that takes
five arguments:

forSpeculative (0,n) state0 reserve commit update

The full type of the function is:

forSpeculative :: (Ord b) ⇒ (Int,Int) → Par b state0 reserve function commit function update

-- Start and end
→ st
→ (Int → st → Par b) reserve function
→ (Int → st → b → Par Bool) commit function
→ (st → Par st) state update function
→ Par ()

1 Both the MIS and BFS implementations in LVish use fully asynchronous,
data-driven scheduling. This is in contrast to the PBBS implementations,
where MIS and BFS use deterministic reservations and therefore are sched-
uled in synchronous rounds.
The main difference from the C++ version, is that this version routes a state argument, \( st \), through each phase of each of the rounds, given the user the chance to update the state once per round. The three phases (reserve, commit, update) are composed sequentially, with barrier synchronizations between; update is not a loop, but a single callback after the commit phase but before the next round. In the reserve phase, the users function optionally returns a value of type \( b \). If it does, that signals that the iteration should attempt to commit this round. The \( b \) returned by the reserve call is then passed to the commit call for the same iteration. The iteration may still fail in the commit phase, however, which is why commit returns a \( \text{Bool} \), with true indicating a fully successful commit.

Regular LVish side effects (monotonic updates) are permissible in any of the reserve, commit, and update functions. Indeed, recall that anything permitted by the type system must at least retain determinism. There are no informal contracts that must be fulfilled. However, since only monotonic updates are allowed as side effects, if we wish to update a data structure multiple times we will have to do so out-of-place (in a functional style). Frequently, we will allocate new LVars in the update phase, and then mutate them in the reserve phase and read them in the commit phase (or vice versa). Unfortunately, these use-once LVars are a source of additional memory allocation and inefficiency relative to in-place versions.

**Parallelizing MSF** Minimum spanning forest can be implemented by first sorting edges by weight, and then sequentially processing edges, greedily adding them to the spanning forest if they are not already redundant. To check redundancy, a disjoint sets data structure\(^?\) is maintained, which tracks to which set each vertex belongs at each point in the algorithm. The PBBS deterministic parallel implementation is simply a speculative version of the sequential loop. The writes performed by each iterate are to link two disjoint sets together by including a new edge in the spanning forest. These writes must not conflict, and so priority based “locking” must be used during the reserve phase to ensure that non-conflicting iterations are executed in parallel.

To code the algorithm in LVish, we first define a new datatype to hold the state between phases (at):\[\text{data MsfState} = \text{MsfState} \{ \text{lastSets} :: \text{Vector NodeID}, \text{thisSets} :: \text{IStructure NodeID}, \text{reserves} :: \text{Vector MinVar} \}\]

Here, we use an array of \( \text{NodeID} \) to represent the disjoint sets. That is, we use \( \text{NodeID} \) as the set identifiers, and the \( \text{Vector NodeID} \) says, for each index \( i \), which set identifier the \( i \)th vertex falls under. \( \text{thisSets} \) and \( \text{lastSets} \) are two copies of the same thing. Because our version of this algorithm is out-of-place, we separately keep the disjoint sets data structure from the last round (\( \text{lastSets} \), read only), and the one that we populate this round (\( \text{thisSets} \), write only).

Note that the type of these two arrays different, \( \text{Vector vs. IStructure} \), because the former is a regular Haskell array, and the latter is an LVVar. The IStructure is converted into a vector at the end of the round by freezing it \([9]\). This prevents further modifications, but snapshots the LVVar so that regular Haskell code can have full access to consume the contents.

We will not go over the whole code for MSF, but will look at the important pieces. The \text{reserve} function passed to forSpeculative is given by:

```haskell
-- Reserve an iteration which corresponds to an edgeid:
\text{reserve} \text{eid MsfState(lastSets,reserves)} = \text{do}
\text{let} (a,b) = \text{edges!eid}
\text{(u,v)} = \text{if} a \preceq b \text{then} (b,a) \text{else} (a,b)
\text{uset} = \text{lastSets!u}
\text{if} u == v
\text{then return Nothing}
\text{else do -- Attempt reservation:}
\text{putMin (reserves!uset) eid}
\text{return (Just ())}
```

Here we see the LVVar \( \text{MinVar} \) again. It is used to implement the priority write. In this case, the minimum \text{eid} written to a set’s \( \text{MinVar} \) during the reserve phase is the winner for that round. The final \( \text{MinVar} \) value is read out in the commit phase to check who won:

```
\text{commit finalOutput eid MsfState(lastSets,thisSets,reserves)} = \text{do}
\text{let} (u,v) = \text{edges!eid}
\text{(uset,vsset) = (lastSets!u, lastSets!v)}
\text{winner \leftarrow freezeMinVar (reserves!uset)}
\text{if} (winner \preceq \text{eid}) \text{then (do}
\text{-- Mark this edge as included in the final output:}
\text{put finalOutput eid (}
\text{let} \text{newSetId = min uset vsset}
\text{-- Link disjoint sets together in the next round:}
\text{put thisSets uset newSetId}
\text{put thisSets vsset newSetId}
\text{return True)
\text{else return False}
```

**Determinism Guarantee** forSpeculative, like all LVVar library functions, guarantees determinism irrespective of the reserve and commit functions the user writes. In part, this is enforced by runtime checking of the consistency of writes to LVars. Problems that would otherwise cause nondeterminism are explicitly caught (deterministically) and raised as exceptions: for example the “multiple-put on IStructure” exception.

Finally, there are some interesting possibilities enabled by the above formulation of deterministic reservations. Because arbitrary Par effects can occur in any iteration of the reserve or commit functions, this means that a reserve can do blocking reads on far-away data, and that, like most LVish functions, the speculative for loop can consume partially populated data structures incrementally.

### 5. Unsafe Deterministic Implementations

Having seen deterministic-safe formulations of the algorithms BFS, MIS, and MSF in LVish, we now review where the liabilities lie when implementing the very same algorithms in a general purpose (non-deterministic) parallel language, as in the PBBS C++ reference implementations.

The deterministic algorithms in PBBS rely on shared state that is accessed concurrently on multiple processors: this is the only form of communication between tasks. And yet it must be used carefully: operations on shared state must be commutative and linearizable, further, for deterministic reservations, all potential loop dependencies must be enumerated explicitly by the programmer without missing any. Finally, APIs (such as disjoint sets’), which provide methods that are threadsafe individually but not mutually, require the user to think carefully about phases and barrier synchronization to maintain determinism.

The simplest example is in the memory cell data type supporting a priority read and write, which can be used for selecting among parallel choices. It has the property that while two priority writes commute, a priority read and write do not commute. The programmer thus has to ensure that operations containing priority reads are not called in parallel with those containing priority writes.

### 6. One solution: BulkRetryT

So far, we have seen asynchronous LVish formulations of three graph algorithms, and we have reviewed bulk-synchronous C++ implementations from previous work \([1]\). The latter are fast, but
written as unsafe, trusted code. The former use determinism-safe primitives, but, unfortunately are slow. In fact, their performance problem is unlikely to be solved by further tuning of the code and ironing out of Haskell/C++ performance differences. The problem is that each of the LVish algorithms (BFS, MIS, MSF) schedules\(O(|V|)\) or \(O(|E|)\) parallel tasks; further, these tasks may block if its dependencies are not met, which in turn means storing a continuation and rescheduling when its dependency is fulfilled. Each of these steps incurs overhead interacting with the parallel scheduler that is paid on a per-vertex/per-edge basis and cannot be amortized.

PBBS, on the other hand, is able to reduce overhead with the following techniques:

- Using parallel loops that never require per-iteration interaction with the parallel scheduler. Even though Cilk’s \(\alpha\) is competitive, per-vertex/edge is too fine-grained.
- Never storing state for loop iterates that fail to run, beyond the indices of the iterates themselves. That is, pbbs, uses a retry strategy rather than a blocking one.

Can we achieve the same benefits while keeping a guaranteed-deterministic programming interface in LVish? Fortunately, yes, but slightly different techniques are required to be robust against arbitrary inter-iteration dependencies.

**Not Assuming sequential-loops** The deterministic reservation mechanism used in PBBS executes a speculative parallel evaluation of semantically-sequential loops. That is, iteration \(i + k\) may depend on \(i\) (forward dependencies) but not the other way around (reverse dependencies). Unfortunately, while this assumption is reasonable for many algorithms, it cannot be enforced by the compiler when loop iterations run arbitrary C++ or Haskell code.

PBBS’s \texttt{speculative_for} can run awry on reverse dependencies because it processes a constant-sized prefix of loop iterations in each round. Thus if all the iterations in a given round fail, then the algorithm tries exactly the same iterations again the next round and does not make progress. This problem could be fixed by expanding the prefix each round so that it would eventually encompass all iterations. In the worst case, the very last iteration is the only one that is enabled to run. Unfortunately, this approach can explode the number of times the reserve phase of the iterates are run to \(O(N^2)\) for \(N\) loop iterations.

The problem is even worse for LVish loops with potentially blocking iterates: running two iterates sequentially for performance reasons results in deadlock if they have a reverse dependency. This would seem to require that each iterate be a parallel task, which we know is inefficient. A better solution to amortizing scheduler overhead while avoiding deadlock is to switch to a retry strategy.

**Abort/retry rather than blocking** For any parallel language that is both deterministic and in which all tasks are idempotent (like LVish), it is possible to retry tasks rather than blocking them. That is, if a task \(T\) encounters an unmet dependency, rather than storing its continuation at that point, it can simply be rescheduled for execution later (ideally after the dependency has been updated).

This abort/retry scheduling approach is already used by some systems, including Intel’s Concurrent Collections (CnC)\(^3\). Abort/retry becomes especially beneficial when considering parallel loops rather than individual tasks. Registering a task with its dependencies to be reexecuted when the dependencies change still requires significant per-task bookkeeping, which we would not want to pay for each iteration of a parallel loop. Rather, when executing a parallel loop we can try many iterations in bulk, and then retry them in bulk. Indeed, this is exactly the strategy used by PBBS’s \texttt{speculative_for}.

**BulkRetry in LVish** To include a retry-based parallel for-loop in LVish, we need to be able to optionally intercept blocking get operations and instead register the failed iteration numbers in a data structure. Fortunately, the semantics of LVish get operations assure that we will not need to worry about idempotence for the reexecuted portion of the task. We use a tweaked version of the Par monad (\text{i.e.} we layer on a “monad transformer”). This tweaked Par monad keeps in thread-local state an arbiter object representing the current parallel loop; when blocking operations are encountered it checks whether the computation is currently in a loop:

- If in loop: register the failed iteration with the loop arbiter. Escape from the current loop iteration.
- Otherwise: block normally.

One advantage of the \texttt{BulkRetry} mechanism is that only minor modifications are required to reuse the BFS and MIS implementations from Section \(\[3\]\). First, the type must be changed from a vanilla Par to a \texttt{BulkRetryT} Par. Second, the parallel loop, for example in MIS, must be replaced with its bulk-retry counterpart.

The disadvantage of \texttt{BulkRetry} is the same as \texttt{for_speculative} in PBBS: the schedule is constrained by synchronous rounds. That is, rather than retrying an aborted task as soon as its dependencies are updated, it must wait until the next round before retrying. However, this reduction in parallelism should not be a problem, as the prefix sizes are chosen to ensure abundant parallelism within each round.

### 7. Evaluation, Future Work, and Conclusion

Our prototype is complete and functional, but requires much tuning work to improve the constant factors in its performance, and to improve scaling of the runtime system. For example, sequential performance of MIS is 19.1 slower than the C++ code delivered with PBBS. However, our main hypothesis, that \texttt{BulkRetry} will improve overheads and scaling, is borne out in Table \(\[3\]\). General Haskell/C++ differences account for much of our current performance problems, although careful tuning of the Haskell code should be able to narrow this gap to under 3x. Other overheads will be more difficult to address: namely, Haskell code tends to be allocation intensive, and its garbage collection (when using the GHC compiler) is parallel but not concurrent. Our graph benchmarks spend over 50% of their time in GC with four or more threads. Nevertheless, GHC is the only mature compiler that provides the determinism guarantee we require, and thus we will continue to work on these issues.

**Pipelining latency benefits** In spite of their poor throughput on their own, the Haskell MIS and BFS implementations are fully asynchronous, which allows them to work well when they are one kernel among many in a larger application. For example, if a downstream phase of processing processes a connected component discovered by a call to BFS, then it can begin executing as soon as a single vertex is discovered, not waiting for the BFS to complete. In a worst-case scenario of a graph containing a linear chain (of, say, 10M vertices), this makes the difference between downstream computations starting after 9 seconds (C++/PBBS) vs. under a millisecond (LVish).

**More data structures** The graph benchmarks in this paper are data-structure intensive workloads. Probably the most significant region for improvement is not in the GHC compiler itself, but

\(^3\) Although language-level enforcement of idempotence is not possible in the case of CnC, whose flagship implementations are in C++ and Java.

\(^4\) On the random local graph topology, with 10M vertices.
in the data structure libraries, both included with LVish and otherwise. For example, we use the existing ‘Data.LVar.SLSet’ data structure (based on a concurrent skip-list) for storing failed iterates. It supports balanced parallel iteration after freezing, but is still a high-bookkeeping, GC-unfriendly data structure. A hierarchical bit-vector may serve better for the purpose of storing failed iterates. Exploring this and other data-structures, in addition to tuning the LVish scheduler, will be primary focuses of our future work.

References


<table>
<thead>
<tr>
<th>MIS</th>
<th>random local</th>
<th>rMat</th>
<th>3D grid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 10^7 7</td>
<td>m = 5 × 10^7</td>
<td>m = 5 × 10^7</td>
</tr>
<tr>
<td>n = 10^7</td>
<td>(1)</td>
<td>(2)</td>
<td>(4)</td>
</tr>
<tr>
<td>misBR</td>
<td>7.419</td>
<td>2.314</td>
<td>5.019</td>
</tr>
</tbody>
</table>

Table 1. Running times for three graph topologies for the vanilla LVish and BulkRetry MIS implementations. Run on a 4-core Intel i7-3770 CPU at 3.40 Ghz.