

# Contentious Data Structures

Leveraging Immutability for Parallelism

---

Sean Laguna

Oct 20, 2016

# Intro

---

# Motivation

- many programs not trivially parallelizable, but also not terribly complicated
- most tools either not accommodating too bare metal (synchronization primitives)
- solutions (Clojure Erlang, Chapel) too committal or invasive

# Observations on Easing Concurrency

- understanding the relevant/slanted operations helps
- data layout shadows algorithm via access pattern
- records help mediate between function and state, but incur cost, which quickly becomes prohibitive

# Requirements for a Method

- less boilerplate than synchronization primitives
  - no learning a new series of pitfalls
- more flexible than “big data”-style solutions
  - handles data contention (in particular, races)
- efficient to the point of benefit
  - speedup over serial implementation on a 2-core machine?

# Pinpointing the Task

In:

- description of stepwise or iterative task(s), algebraic properties of them
- data on which to operate, in a provided container
- number of desired threads

Out:

- concurrent execution of task(s) on that many threads

# Needs of Scientific Programs

- handle large amounts of data with computational interdependence; scaling
- runtime options to match users needs; expressiveness
- small methodological changes should not require big design changes or manifest verification difficulties
- hardware multiplicity: distributed, manycore, gpu

# Supporting Material

---

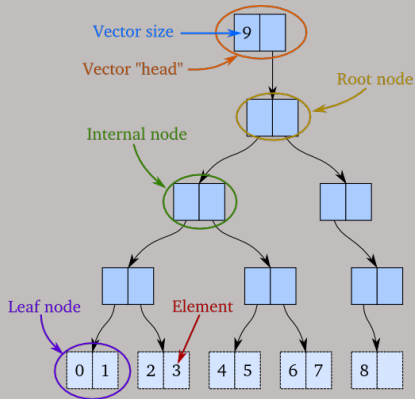


# (Functionally) Persistent Data Structures

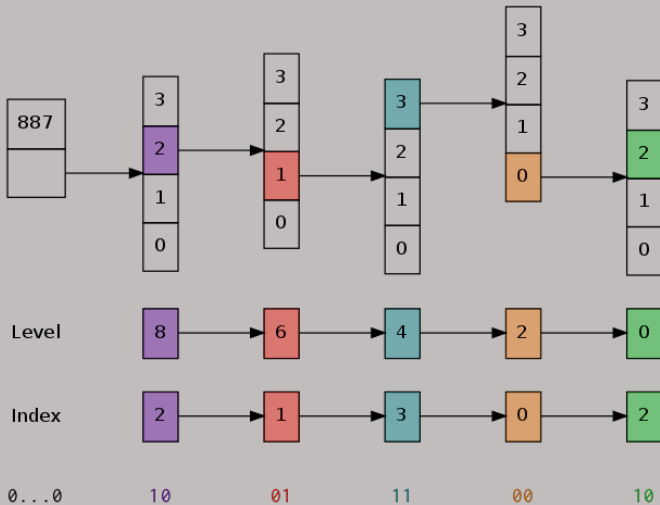
- not storage persistence, but uncanny similarities
- Okazaki, Hickey
- data structures don't have to be immutable to look immutable

# Bit-partitioned Tries

- underlying structure that provides effect
- yields a vector or “hash map”



**Figure 1:** Definition of parts of bit-partitioned trie. From <http://hypirion.com/musings/understanding-persistent-vector-pt-1>



**Figure 2:** Using the bit-partitioned trie as a vector. From <http://hypirion.com/musings/understanding-persistent-vector-pt-2>

# Dependency Graphs

Identify...

- opportunities for parallelism
- points that necessitate concurrency control

# Cases

1. All edges  $e \in E(P)$  incident to  $n$  are **incoming** to  $n$ , and all edges  $e \in E(Q)$  incident to  $n$  are **outgoing** from  $n$ :
  - $n$  connects  $P$  to  $Q$ , and  $Q$  can only be computed once  $P$  has been.
2. All edges  $e \in E(P)$  incident to  $n$  are **incoming** to  $n$ , and all edges  $e \in E(Q)$  incident to  $n$  are **incoming** to  $n$ :
  - $P$  and  $Q$  can be computed in parallel, but the value of  $n$  can only be determined once both  $P$  and  $Q$  have been computed.
3. All edges  $e \in E(P)$  incident to  $n$  are **outgoing** from  $n$ , and all edges  $e \in E(Q)$  incident to  $n$  are **outgoing** from  $n$ :
  - $P$  and  $Q$  can be computed in parallel, but only once the value of  $n$  has been determined.

# Atomic Operations

- lock-free MPMC work queue
  - for managing tasks among threads
- IDs for versioning container
- managing the interrelationships (much more later)

# Three “Views” of The Method

1. theoretic (dependency graph and algebra)
2. memory-based (data layout and transformation)
3. operative (implementation and execution)

## Related Efforts

- Clojure: reducers/transducers, etc; Erlang
- CITRUS; cuckoo hashing
- PRCU, RLU (read-copy-update successors); thread-local storage
- transactional memory
  - notably the identification and accommodation of pathological execution ordering



# Inner Workings

---

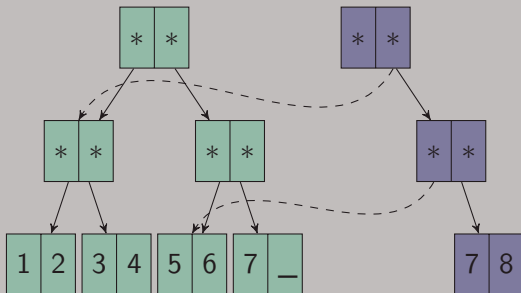
# Overview

- each thread gets an ID and its own snapshot of the data structure at each step
- each thread will perform its work with that snapshot as if the values were correct
- conflict detection and resolution will propagate any late updates, using knowledge of the partition and operations at hand to finalize all values

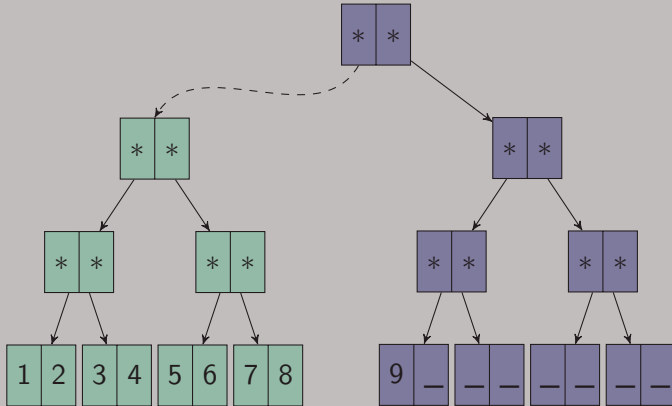
Remember... Persistent updates create new copies of the vector. So, they will look immutable. Transient updates will change the snapshot

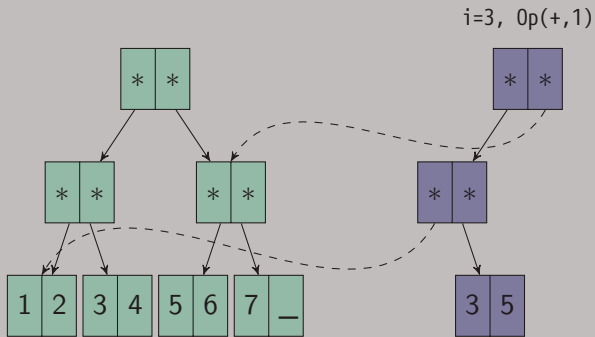
- transient vector with extra stuff for parallel operation
  - snapshots (freeze, detach + reattach)
  - conflict detection and resolution

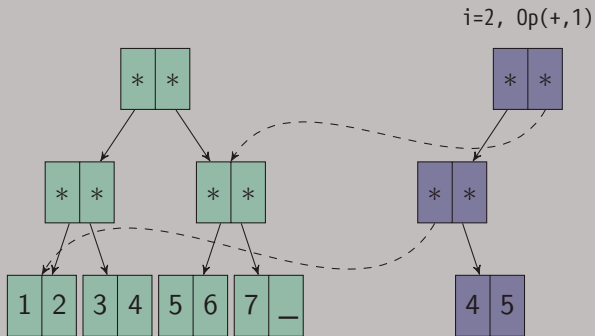
i=na, Op(push\_back,8)

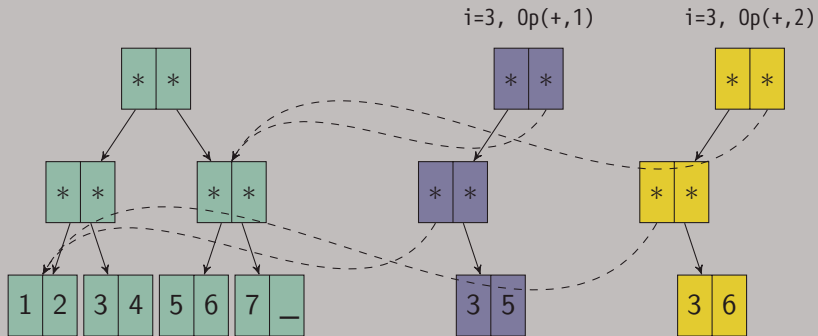


i=na, Op(push\_back,9)











- take a persistent “global snapshot” of the vector to check against later
- each proc’s snapshot will “hang off” this one
  - so will auxiliary CT vectors that contribute to the computed values

```
// if we want our output to depend on input
tracker.emplace(dkey, imap, op);
// make a latch for reattaching splinters
dep.latches.emplace(dkey, new boost::latch(ctts::HWCONC));
// make sure we have a valid _orig if not monotonic
if (!ctts::is_monotonic(imap)) {
    // locked this->_data
    std::lock_guard<std::mutex> lock(dlck);
    this->_orig = this->_data;
    this->_data = this->_data.new_id();
}
```

- create a splinter for a processor to compute with
- separate interface:
  - can only perform registered operation
  - can only access/modify specified domain and range for that operation

```

auto &dep_tracker = dep_tracker_it->second;
// must use _orig if not monotonic
if (!ctts::is_monotonic(dep_tracker.imaps[0])) {
    dep_tracker._used[p] = _orig;
} else {
    // locked this->_data
    std::lock_guard<std::mutex> lock(dlck);
    dep_tracker._used[p] = this->_data;
    this->_data = this->_data.new_id();
}
    spl_t_vector<T> spl_t(dep_tracker._used[p], dep_tracker.ops);
    dep.splinters.emplace(spl_t._data.get_id(), false);
return spl_t;

```

- for all disjunctly-modified leaves, perform most-common-branch assignment
- for non-disjunctly modified leaves, copy disjunctly-modified values
  - pray the other values will be fine

# Conflict Detection and Resolution

- for every potentially contended value, the conflict detection function is run
- if a conflict is detected, the resolution function is run
  - normally, this involves recovering the difference using the inverse of the “outer operator”, and then using the original operator again upon the fresh value and this difference
  - other possibilities exist too, under special circumstances

```

for (size_t ci : dep_tracker.contended) {
    for (size_t cmap : dep_tracker.imap(ci)) {
        T cmap& = dep._data[cimap];
        // use iterators to get const references; faster
        auto curr = this->_data.cbegin() + ci;
        auto trck = dep_tracker._used[p].cbegin() + ci;
        T diff = dep_op.inv(*curr, *trck);
        if (diff != dep_op.identity) {
            cmap = dep_op.f(cmap, diff);
            // since this changed,
            // we may need to resolve it in the future, too
            dep.maybe_contended.insert(cimap);
        }
    }
}

```

# Implementation

---



# User Interface

// creation

```
cont_vector<double> cont_inp;  
for (size_t i = 0; i < n; ++i) {  
    cont_inp.unprotected_push_back(rand());  
}
```

// reduce

```
auto cont_ret = cont_inp.reduce(contentious::plus<double>);  
contentious::tp.finish();
```

// foreach

```
auto ret1 = cont_inp.foreach(contentious::mult<double>, 2);  
auto ret2 = ret1->foreach(contentious::mult<double>, cont_other);  
contentious::tp.finish();
```

```

// stencil
for (int t = 1; t < r; ++t) {
    int icurr = t % t_store;
    int iprev = (t-1) % t_store;
    grid[icurr] = grid[iprev]->stencil<-1, 0, 1>(
                                                {1.0*s, -2.0*s, 1.0*s});
    if (t % (t_store-1) == (t_store-2)) {
        contentious::tp.finish();
    }
}
contentious::tp.finish();

```

```
template <typename T>
cont_vector<T> cont_vector<T>::reduce(const ctts::op<T> op)
{
    // our reduce dep is just one value
    auto dep = cont_vector<T>();
    dep.unprotected_push_back(op.identity);

    freeze(dep, ctts::alltoone<0>, op);
    // no template parameters because no auxiliary variables
    exec_par<>(ctts::reduce_splt<T>, dep);

    return dep;
}
```

```

template <typename T>
std::shared_ptr<cont_vector<T>> cont_vector<T>::foreach(
    const ctts::op<T> op,
    const T &val)
{
    auto dep = std::make_shared<cont_vector<T>>(*this);

    // template parameter is the arg to the foreach op
    freeze(*dep, ctts::identity, op);
    exec_par<T>(ctts::foreach_splt<T>, *dep, val);

    return dep;
}

```

It would be really nice to get type inference on the template parameters.

```

template <typename T>
void foreach_splt(cont_vector<T> &cont, cont_vector<T> &dep,
                  const uint16_t p, const T &val)
{
    size_t a, b;
    std::tie(a, b) = partition(p, cont.size());
    spl_t_vector<T> spl_t = cont.detach(dep, p);
    const binary_fp<T> fp = spl_t.ops[0].f;
    auto end = spl_t._data.begin() + b;
    for (auto it = spl_t._data.begin() + a; it != end; ++it) {
        T &target = *it;
        target = fp(target, val);
    }
    cont.reattach(spl_t, dep, p, a, b);
}

```

```

template <typename T>
template <int... Offs>
std::shared_ptr<cont_vector<T>> cont_vector<T>::stencil(const std::vector<T> &coeffs)
{
    constexpr size_t NS = sizeof...(Offs);
    std::array<ctts::imap_fp, NS> offs{ {ctts::offset<Offs>...} };

    auto dep = std::make_shared<cont_vector<T>>(*this);
    freeze(*dep, ctts::identity, ctts::plus_fp<T>);
    for (size_t i = 0; i < NS; ++i) {
        ctts::op<T> fullop = {
            0,
            boost::bind<T>(ctts::multplus_fp<T>, _1, _2, coeffs[i]),
            ctts::minus_fp<T> // the "inverse" is that of the "outer operator"
        };
        freeze(*this, *dep, offs[i], fullop);
    }
    for (uint16_t p = 0; p < ctts::HWCONC; ++p) {
        auto task = std::bind(ctts::stencil_splt<T, NS>, std::ref(*this), std::ref(*dep), p);
        ctts::tp.submit(task, p);
    }
    return dep;
}

```

```

template <typename T, size_t NS>
void stencil_splt(cont_vector<T> &cont, cont_vector<T> &dep, const uint16_t p)
{
    // get bounds (a, b) and "safe" bounds (ap, bp) for stencil (tedious)
    for (size_t i = 0; i < NS; ++i) {
        os[i] = a - imaps[i+1](a);
        ioffs[i] = os[i] - os[0];
        fs[i] = tracker.ops[i+1].f;
    }
    spl_t_vector<T> spl_t = cont.detach(dep, p);
    auto trck = tracker._used[p].cbegin() + (ap+os[0]);
    auto end = spl_t._data.begin() + bp;
    for (auto it = spl_t._data.begin() + ap; it != end; ++it, ++trck) {
        T &target = *it;
        for (size_t i = 0; i < NS; ++i) {
            target = fs[i](target, *(trck + ioffs[i]));
        }
    }
    cont.reattach(spl_t, dep, p, a, b);
}

```

Reduce: reduction of values using multiplication as the operator

- vs seq, vec, async, omp

Foreach: data-parallel aggregate operation using addition

- vs serial implementation

Heat: 1D spatial finite difference solver for heat equation

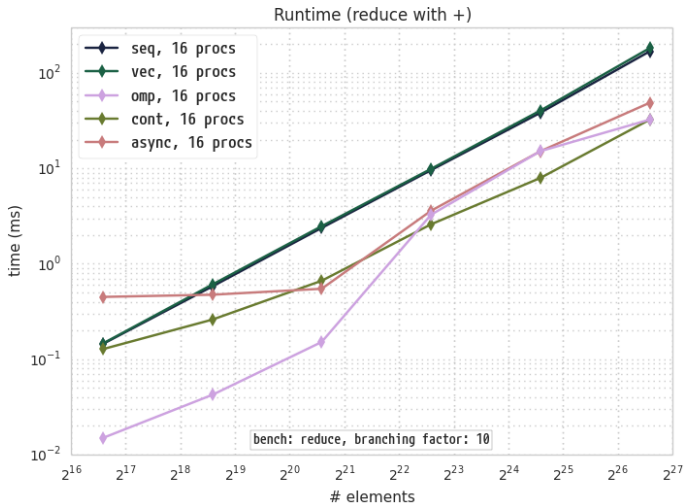
- vs serial implementation

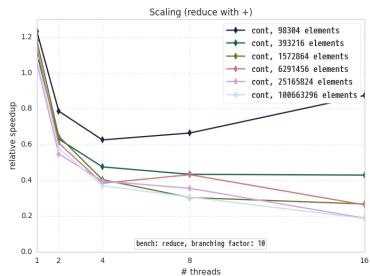
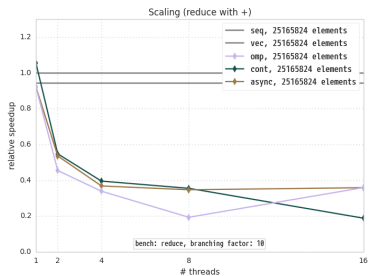
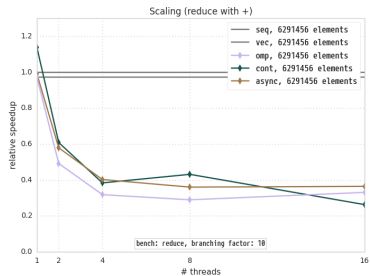
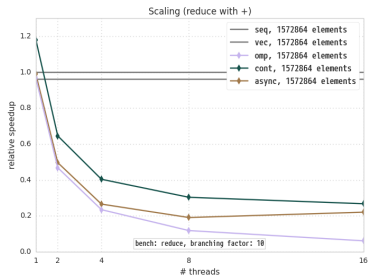


# Results

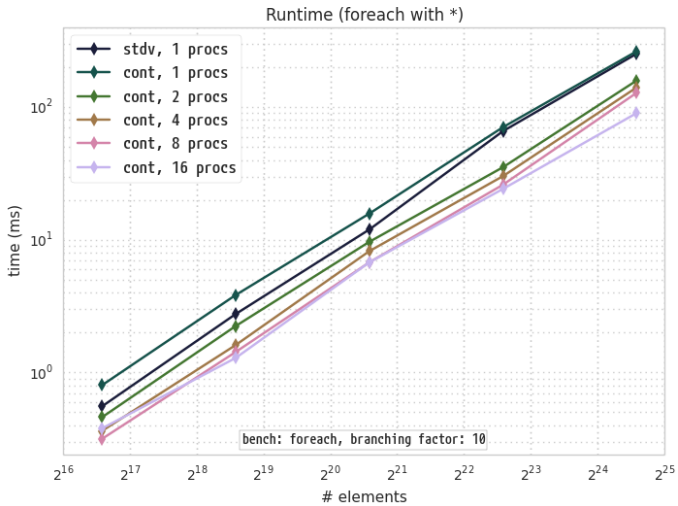
---

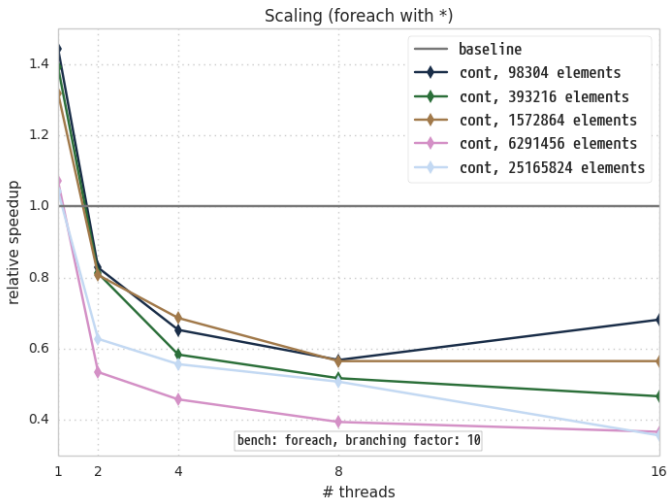
# Reduce



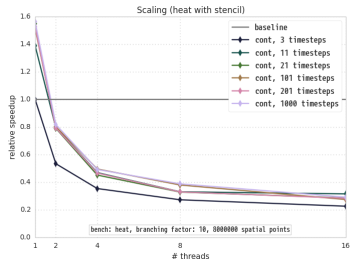
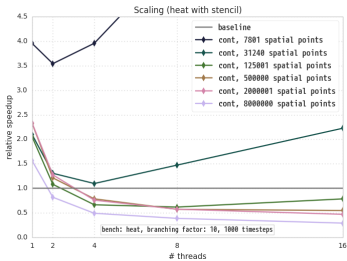
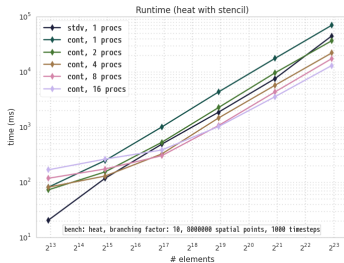


# Foreach





# Heat



# Observations

---

# General Outcome

- small problem sizes don't fare well
  - perhaps tolerable
  - not easy to parallelize along time domain
- large number of timesteps doesn't make things worse
  - you can always cap the unresolved depth, anyway



# Poor Performance?

- dissatisfying performance across the board
  - shared memory is dissatisfying, caching is dissatisfying
  - OpenMP is a bit less dissatisfying
- even SSE/AVX instructions perform similarly
- `std::vector` struggles as its size increases
  - contiguous memory means large allocations

# Where to Go From Here

---

# Improvements

- API could use some polish
  - copying problem
  -
- incorporate vector size changes into tracking system, or as operations
- performance leads
  - memory arenas: reuse leaves to avoid redundant work
  - branching factor: tuned, heterogeneous
  - different trie indexing scheme: gather contended values
  - computation ordering: minimize potential conflicts

- fundamental problem has been “reduced” to partitioning and managing the input and output index sets, and the mapping between them

**Thank you!**

---