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
- solutions (Clojure, Erlang, Chapel) too committal or invasive

Observations on Easing Concurrency

- understanding the relevant/slated 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” depending on indexing scheme
- branching factor (elements/node) of 2… in practice, 2^5, 2^6, 2^{10}

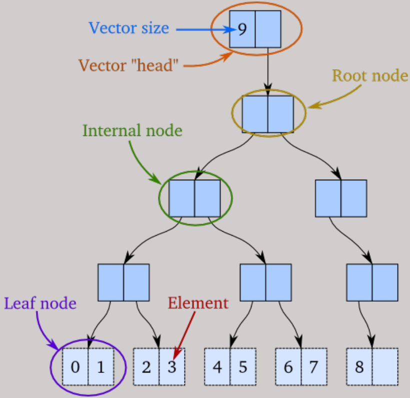


Figure 1: Definition of parts of bit-partitioned trie. From
`http://hypirion.com/musings/understanding-persistent-vector-pt-1`

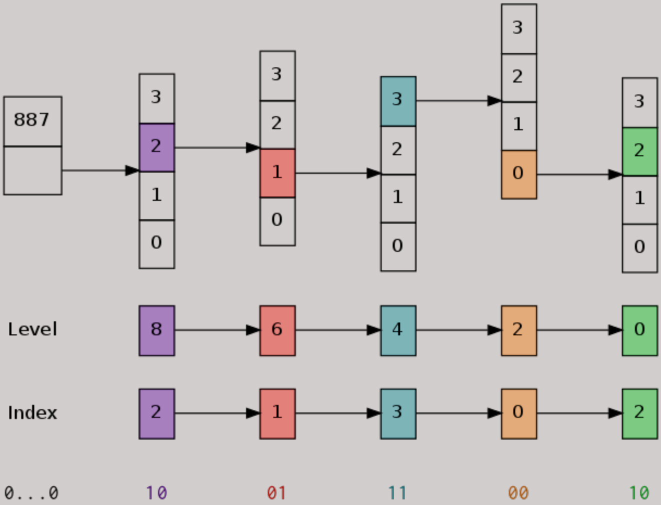
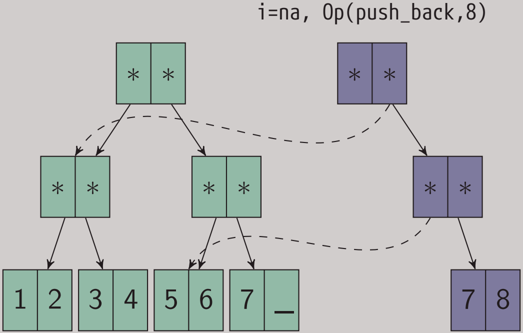


Figure 2: Using the bit-partitioned trie as a vector. From
http://hypirion.com/musings/understanding-persistent-vector-pt-2

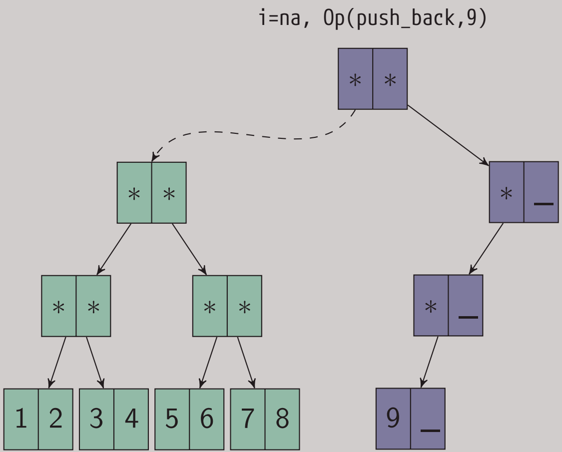
Persistent and Transient Vectors

- persistence: always perform path-copying
- transience: perform path-copying whenever a node's ID differs from the root's ID

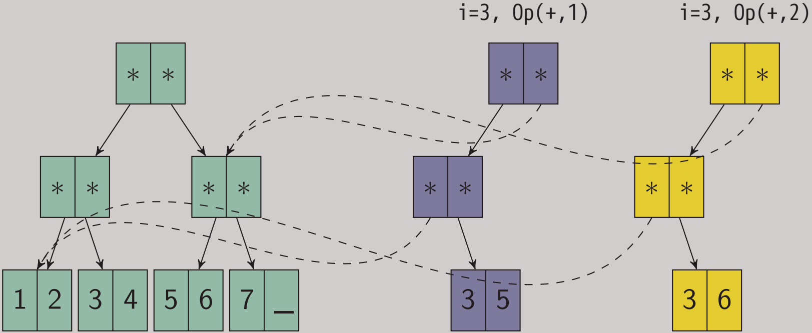
Persistent Vector



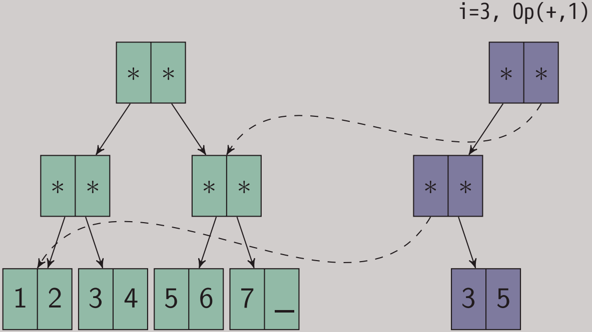
...cont'd



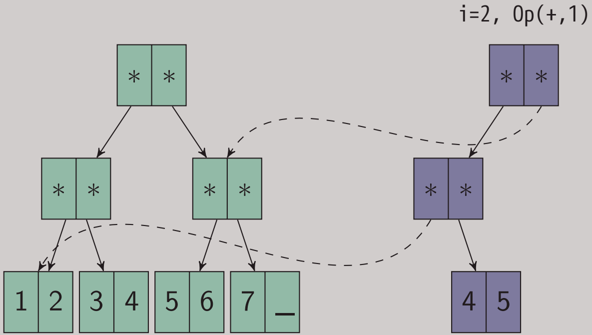
Persistent Set



Transient Set



...cont'd



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 $e \in E(Q)$ indicent 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 $e \in E(Q)$ indicent to n are **incoming** to n:
   - P and Q can be computed in parallel, but the value of n can only be determineded 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 $e \in E(Q)$ indicent 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)

Recap: Three "Views" of The Method

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

Related Efforts

- Clojure, leveraging persistence
  - reducers/transducers, etc
- concurrent data structures
  - CITRUS/CASTLE trees
  - cuckoo hashing
  - PRCU, RLU (read-copy-update successors)
- transactional memory
  - notably the identification and accommodation of pathological execution ordering

Methods and Implementation

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, and return the head of the modified vector. Transient updates will change the copy directly if the path to that leaf has already been modified via that copy

Contentious Vector

transient vector with extra stuff for parallel operation

- a `tracker` to keep track of
  - global snapshot
  - partition
  - operator (actual operator, “outer inverse”, identity)
  - index mapping (and list of “contended” indices)
  - any frozen auxiliary `ctvectors`
  - any [corresponding] splinters
  - latches to count down reattachment
- global and thread-local snapshot management
  - `freeze`, `detach` & `reattach` methods
  - fast `assign` of range of values from one `ctvector` to another
- conflict detection and resolution methods

Contentious Namespace

some extra stuff

- a threadpool for ordering concurrent but stepwise "tasks"
  - no synchronization between steps, but can force resolution via `finish`
  - producer-consumer queues for tasks and resolutions
- basic instances of:
  - partition functions
  - index mappings
  - operators

Credit to Facebook's Folly Library

Facebook's Folly library provides:

- `ProducerConsumerQueue`: threadpool
- `LifoSem`: threadpool
- `AtomicHashMap`: trackers, latches, splinters
- marginal performance benefits in the form of consistency

Freeze

- take a persistent “global snapshot” of the vector
- take a persistent “local snapshot” of the vector for each thread
- take a transient copy (“splinter”) of each local snapshot for computing
  - they “hang off” the local snapshot
  - so do auxiliary contentious vectors that contribute to the computed values

```cpp
// 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();
```

Detach

- 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

```cpp
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();
}
splinter<T> splt(dep_tracker._used[p], dep_tracker.ops);
dep.splinters.emplace(splt._data.get_id(), false);
return splt;
```

Reattach

- 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 cimap : 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);
        }
    }
}
```

Practical Usage

User Interface

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

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

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

Detail - reduce

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

Detail - foreach

```
template <typename T>
std::shared_ptr<ctvector<T>> ctvector<T>::foreach(
                                        const ctts::op<T> op,
                                        const T &val)
{
    auto dep = std::make_shared<ctvector<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

Thread-level View of foreach

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

Detail - stencil

```
template <typename T>
template <int... Offs>
std::shared_ptr<ctvector<T>> ctvector<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<ctvector<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;
```

Thread-level View of stencil

```
template <typename T, size_t NS>
void stencil_splt(ctvector<T> &cont, ctvector<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;
    }
    splinter<T> splt = cont.detach(dep, p);
    auto trck = tracker._used[p].cbegin() + (ap+os[0]);
    auto end = splt._data.begin() + bp;
    for (auto it = splt._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(splt, dep, p, a, b);
```

Closer Look: Stencils

1D Heat Equation

Heat Equation

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0, \text{ or}$$
$$\frac{\partial u}{\partial t} - \alpha \frac{\partial^2 u}{\partial x^2} = 0$$

4-point finite difference stencil

$$u_x^{t+1} = a(u_{x+1}^t + u_{x-1}^t) + b(u_x^t),$$
$$a = r \text{ and } b = (1 - 2r) \text{ for } r = \kappa / h^2$$

Data Dependencies

Processor p has a chunk of of the domain with values

$$x_p = \{\alpha * p, \alpha * (p+1) - 1\}, \text{ where}$$
$$\alpha = X/P$$

p can compute:

$$u_{x_{t1}}^{t+1} \text{ for } x_{t1} \in \{\alpha * p + 1, \ldots \alpha * (p+1) - 2\},$$
$$u_{x_{t2}}^{t+2} \text{ for } x_{t2} \in \{\alpha * p + 2, \ldots \alpha * (p+1) - 3\},$$
$$\ldots$$
$$u_{x_{t\alpha}}^{t+p} \text{ for } x_{t\alpha} \in \emptyset$$

with no communiation.

...cont'd

After α steps through time, processor p must wait for processor $p-1$ to finish computing its data. Likewise, waiting indefinitely for both processor $p-1$ and $p+1$ to complete only allows for $\alpha/2$ steps through time to be completed. Thus, we can say that

$$u^{t+i}_{\alpha*p+i-1} \text{ depends on } p-1, \text{ and}$$
$$u^{t+i}_{\alpha*(p+1)-1-(i-1)} \text{ depends on } p+1,$$
$$\text{for } i \in \{1, \alpha\}.$$

Results

Tests

Reduce: reduction of values using multiplication as the operator

- vs `seq`, `vec`, `async`, `omp`

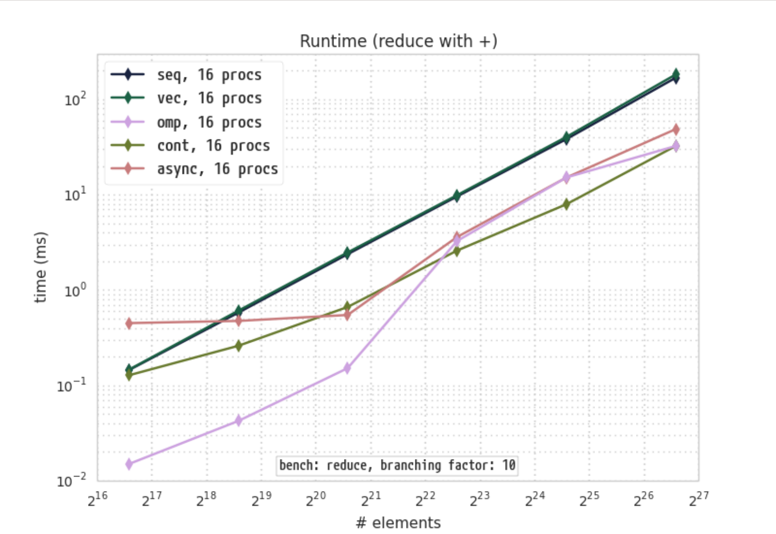
Foreach: data-parallel aggregate operation using addition

- vs serial implementation

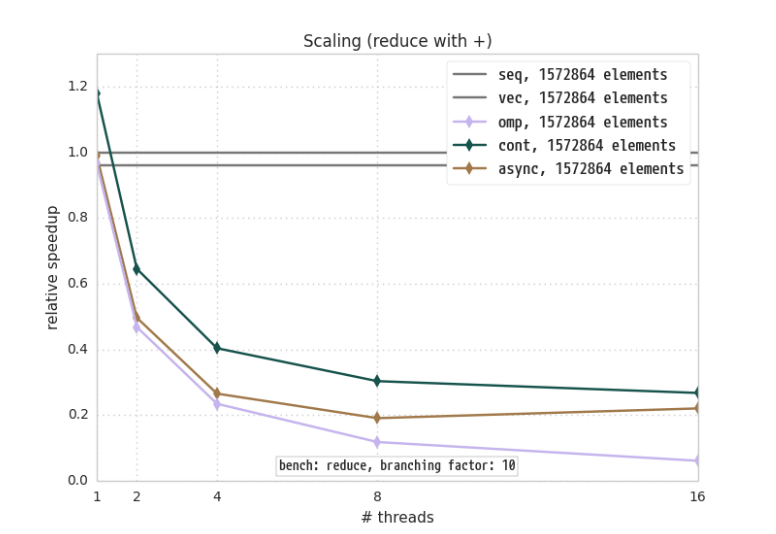
Heat: 1D spatial finite difference solver for heat equation

- vs serial implementation

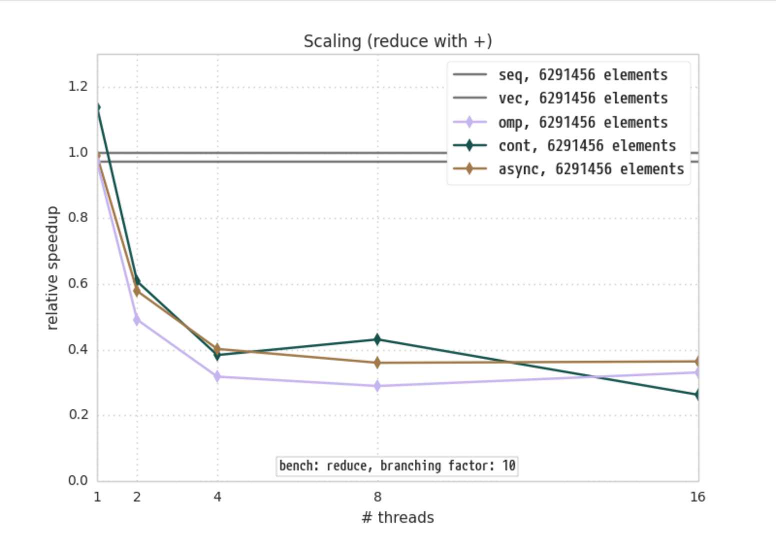
reduce - runtime at various problem sizes



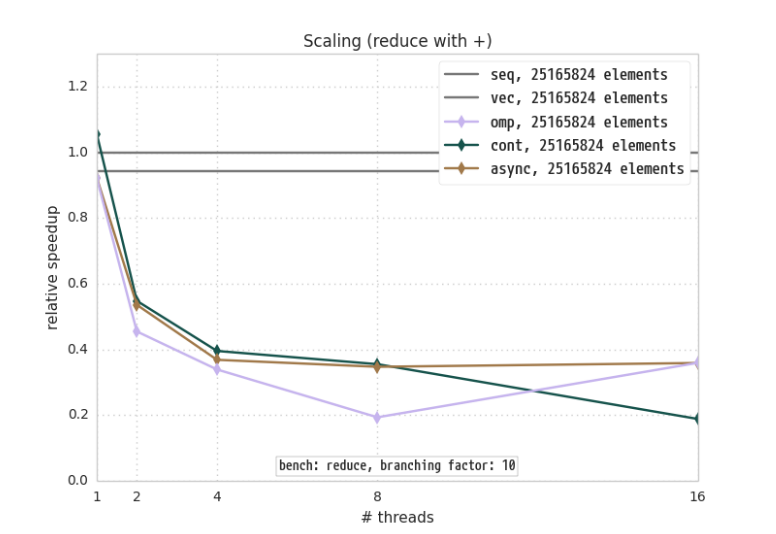
`reduce` - speedup for "small" problem size



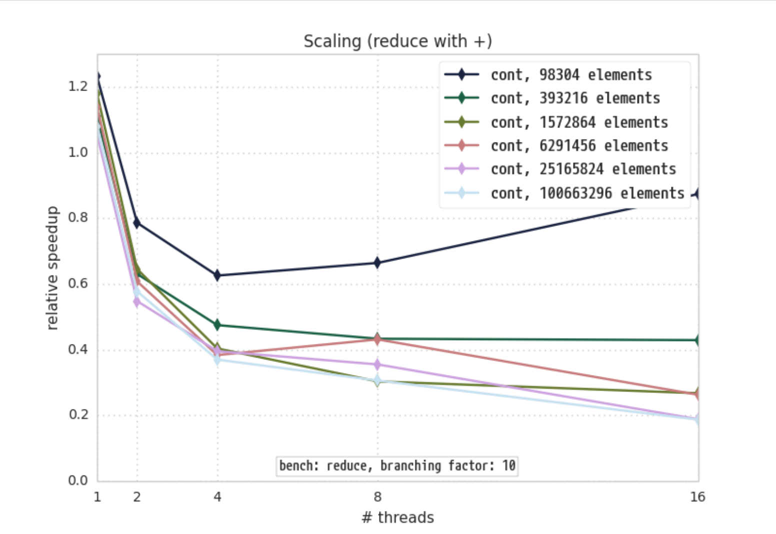
`reduce` - speedup for "medium" problem size



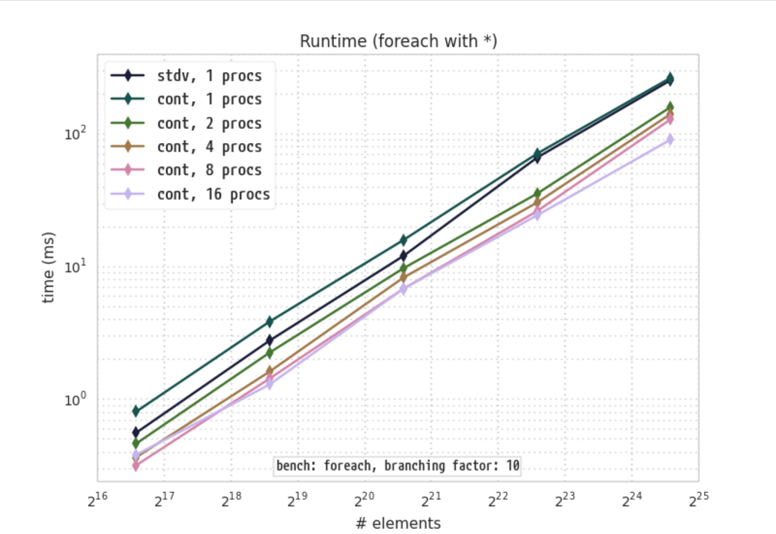
reduce - speedup for "large" problem size



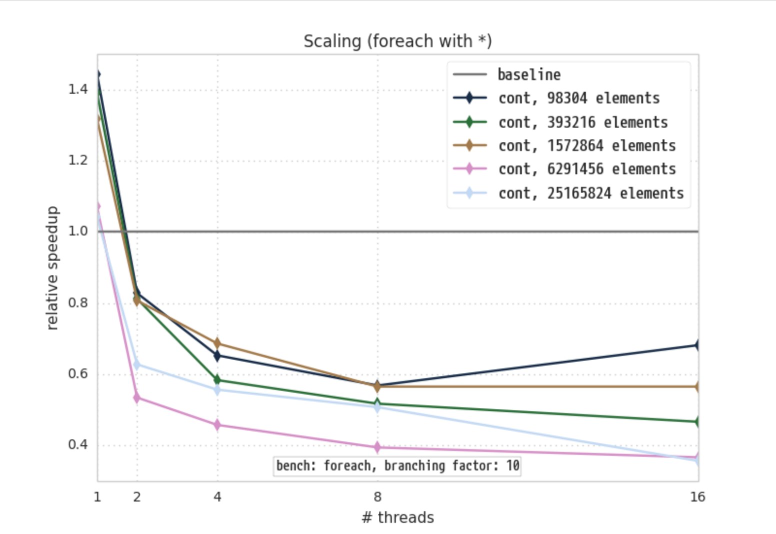
reduce - speedup summary



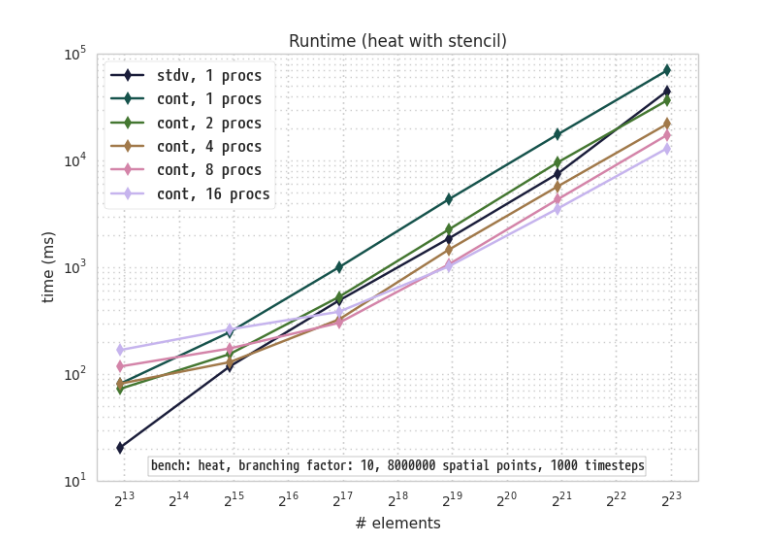
foreach - runtime at various problem sizes



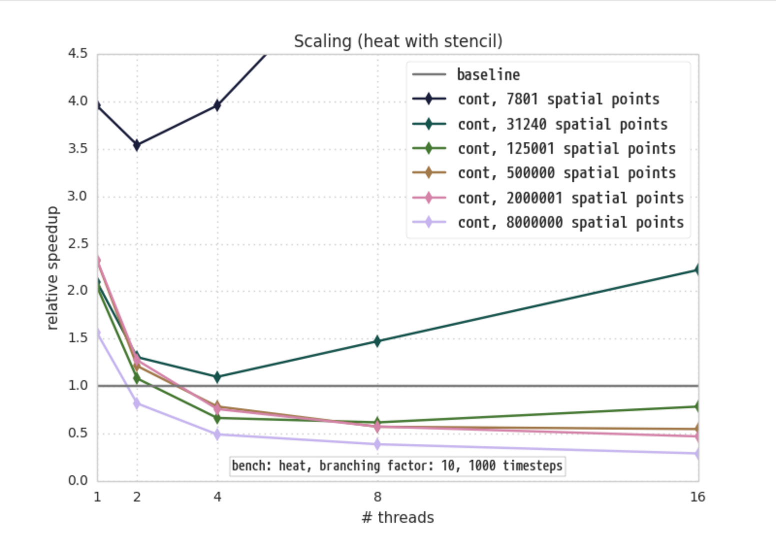
foreach - speedup summary



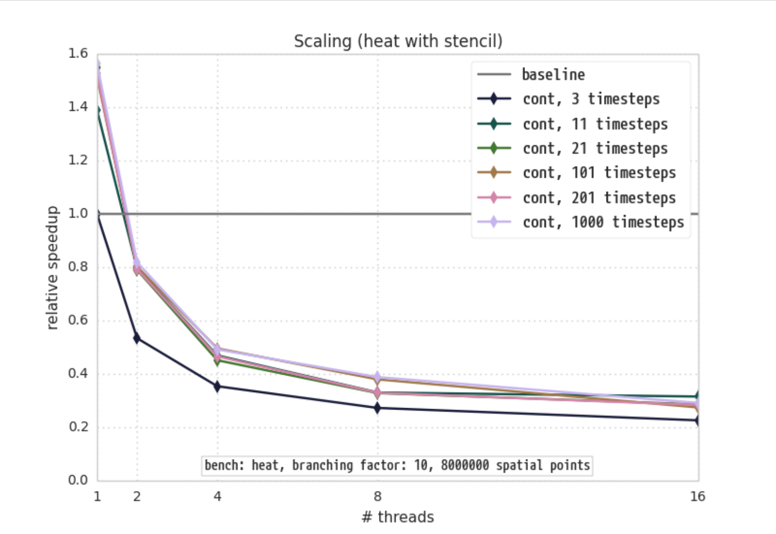
heat - runtime at various spatial sizes



heat - speedup summary for spatial domain size variation



heat - speedup summary for time domain size variation



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
  - be stricter about correctness for splinters
  - 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

Partitioning

- fundamental problem has been “reduced” to partitioning and managing the index sets for each processor, and the mapping between them
  - data dependencies affect the optimal partition
  - provides a framework for tackling this problem

Thank you!