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/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”

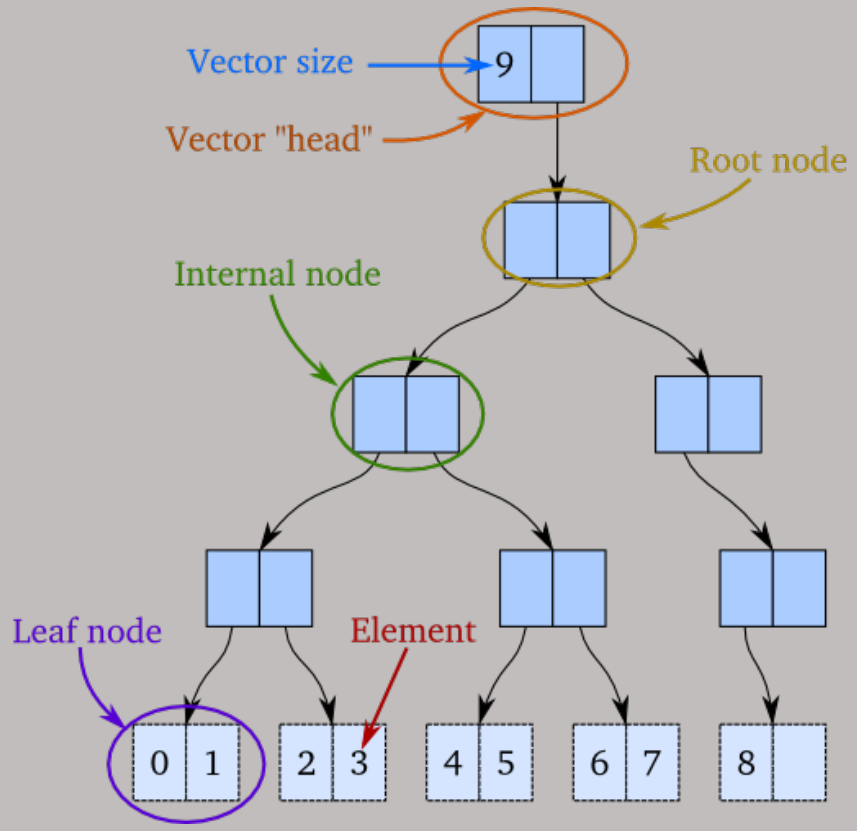


Figure 1: Definition of parts of bit-partitioned trie. From `http://hypirion.com/musings/understanding-persistent-vector-pt-1`

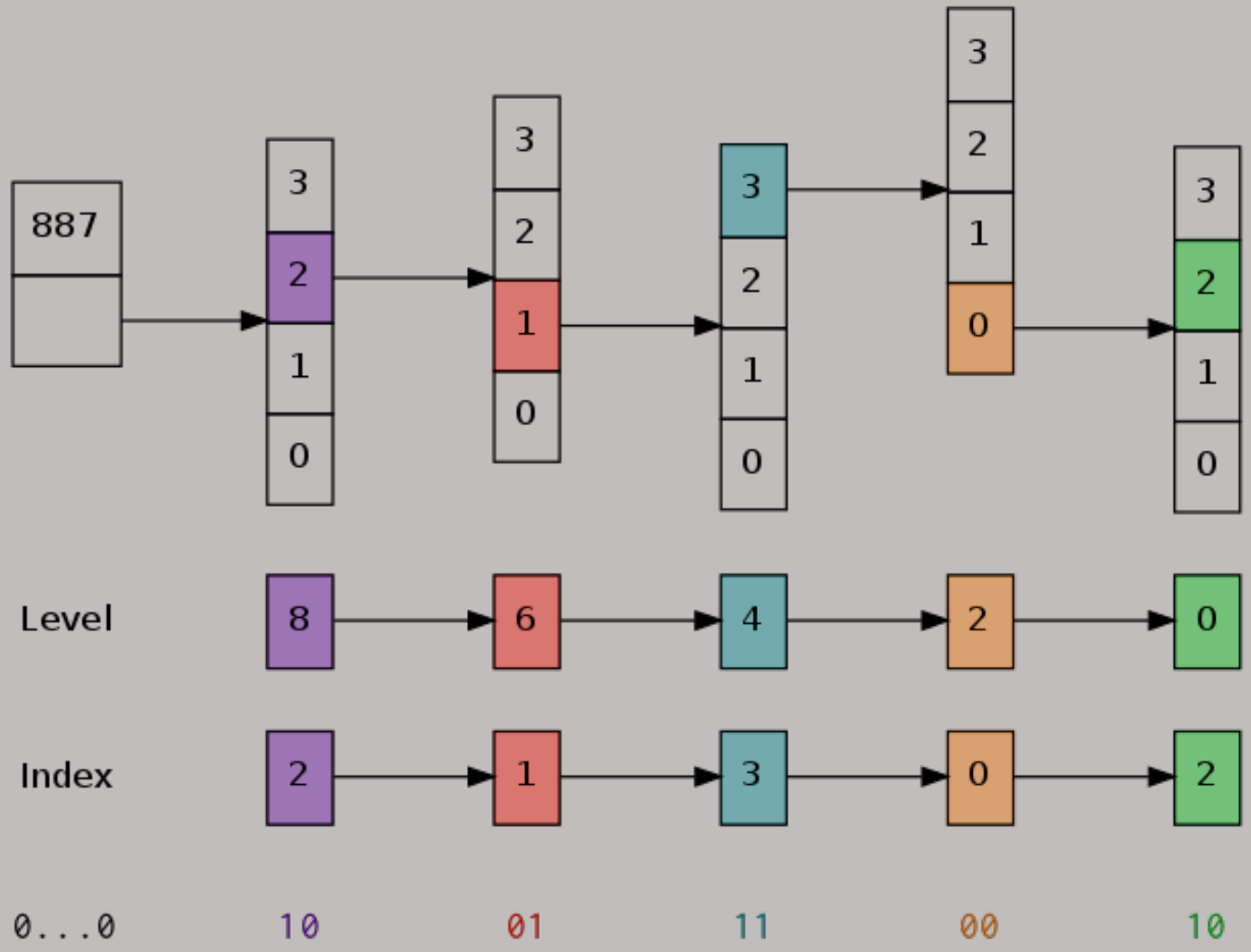


Figure 2: Using the bit-partitioned trie as a vector. From `http://hypirion.com/musings/understanding-persistent-vector-pt-2`

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 $e \in E(Q)$ 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
- IDs for versioning container
- managing the interrelationships (much more later)

3 “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: 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

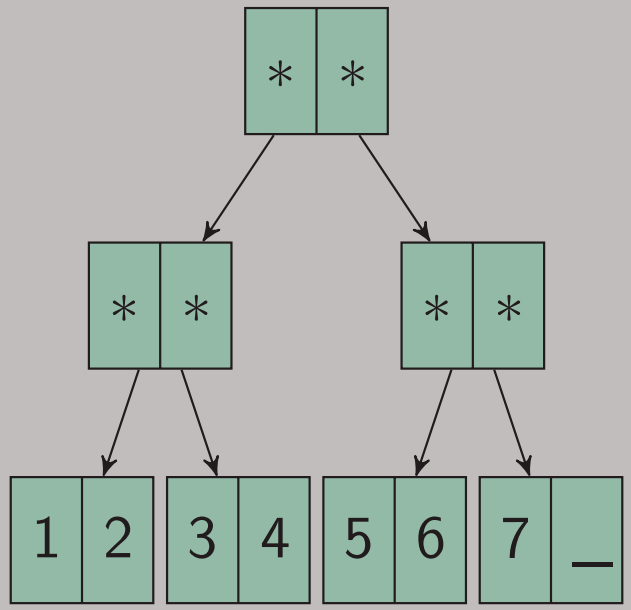
Overview

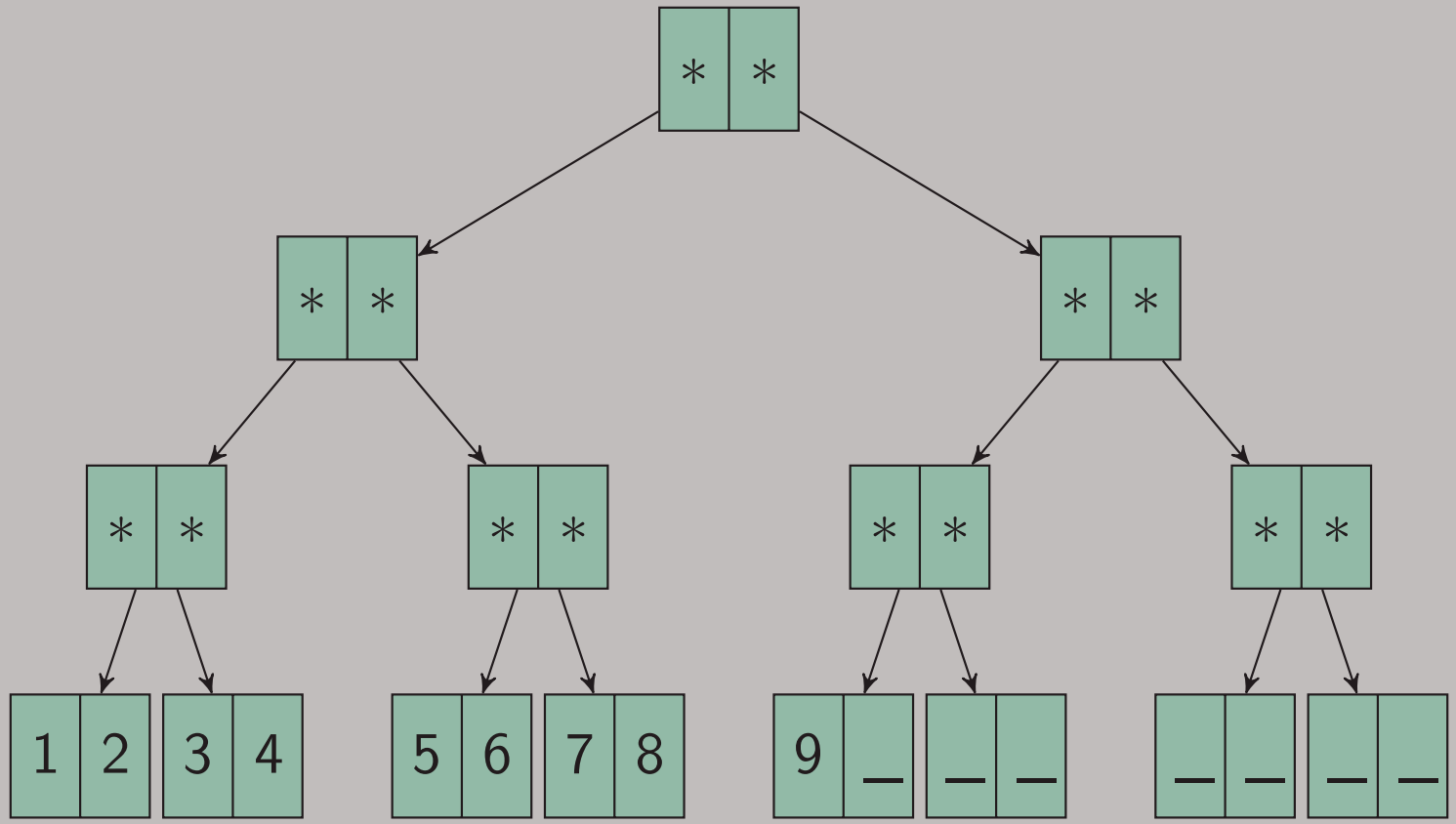
- each thread gets an ID and its own snapshot of the data structure at each step
- each thread will perform its work with that snapshot as if the values were correct
- conflict detection and resolution will propagate any late updates, using knowledge of the partition and operations at hand to finalize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

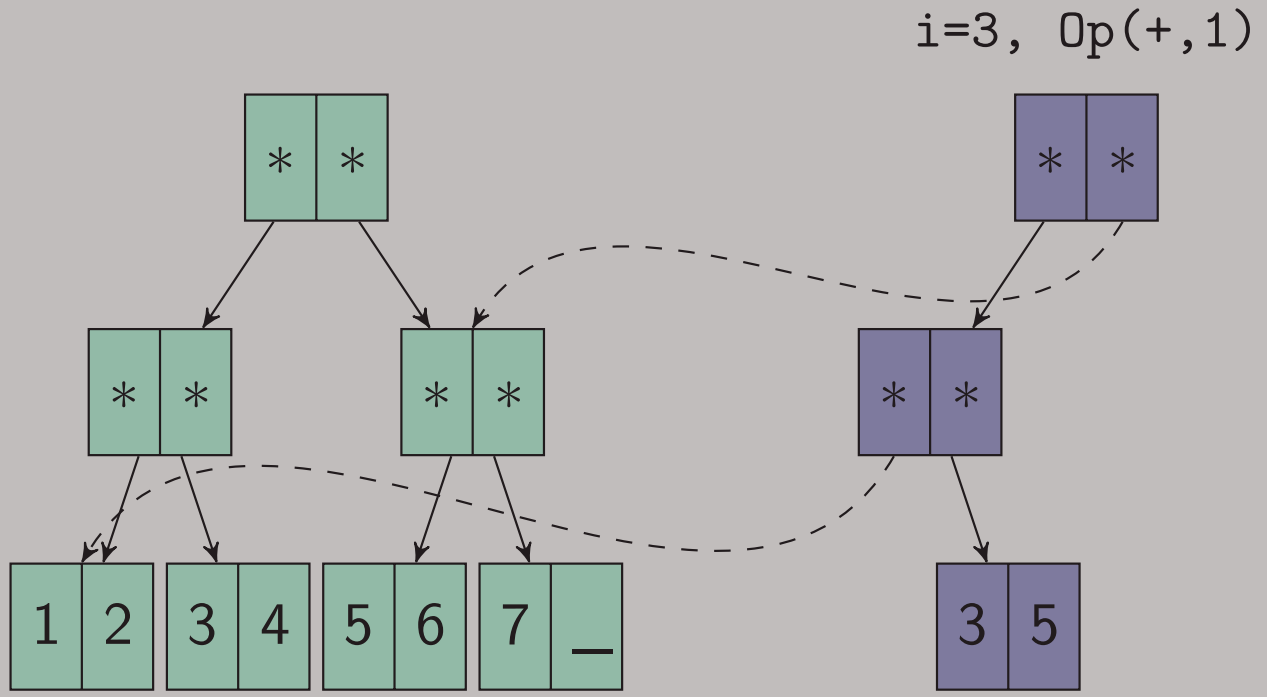
Remember... Persistent updates create new copies of the vector. So, they will look immutable. Transient updates will change the snapshot

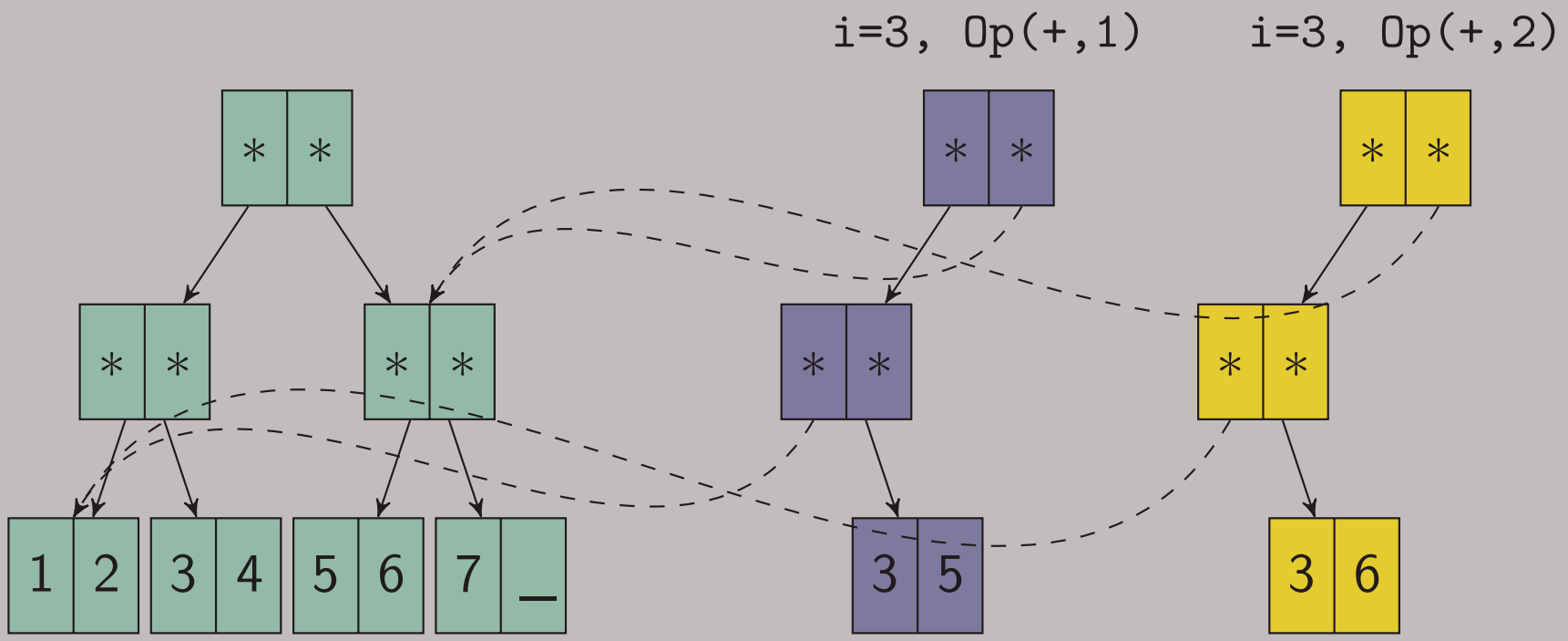
Contentious Vector

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


*
*
*
*
*
*
1
2
3
4
5
6
7
_


* *
* *
* *
* * * *
* * * *
1 2 3 4
5 6 7 8
9 _ _ _
_ _ _ _


i=3, Op(+,1)
*
*
*
*
*
*
*
*
1
2
3
4
5
6
7
_
3
5


i=3, Op(+,1)
i=3, Op(+,2)
1
2
3
4
5
6
7
_
3
5
3
6

Freeze

- 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

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

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

Implementation

User Interface

Detail

Tests

Reduce: reduction of values using multiplication as the operator

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

Foreach: data-parallel aggregate operation using addition

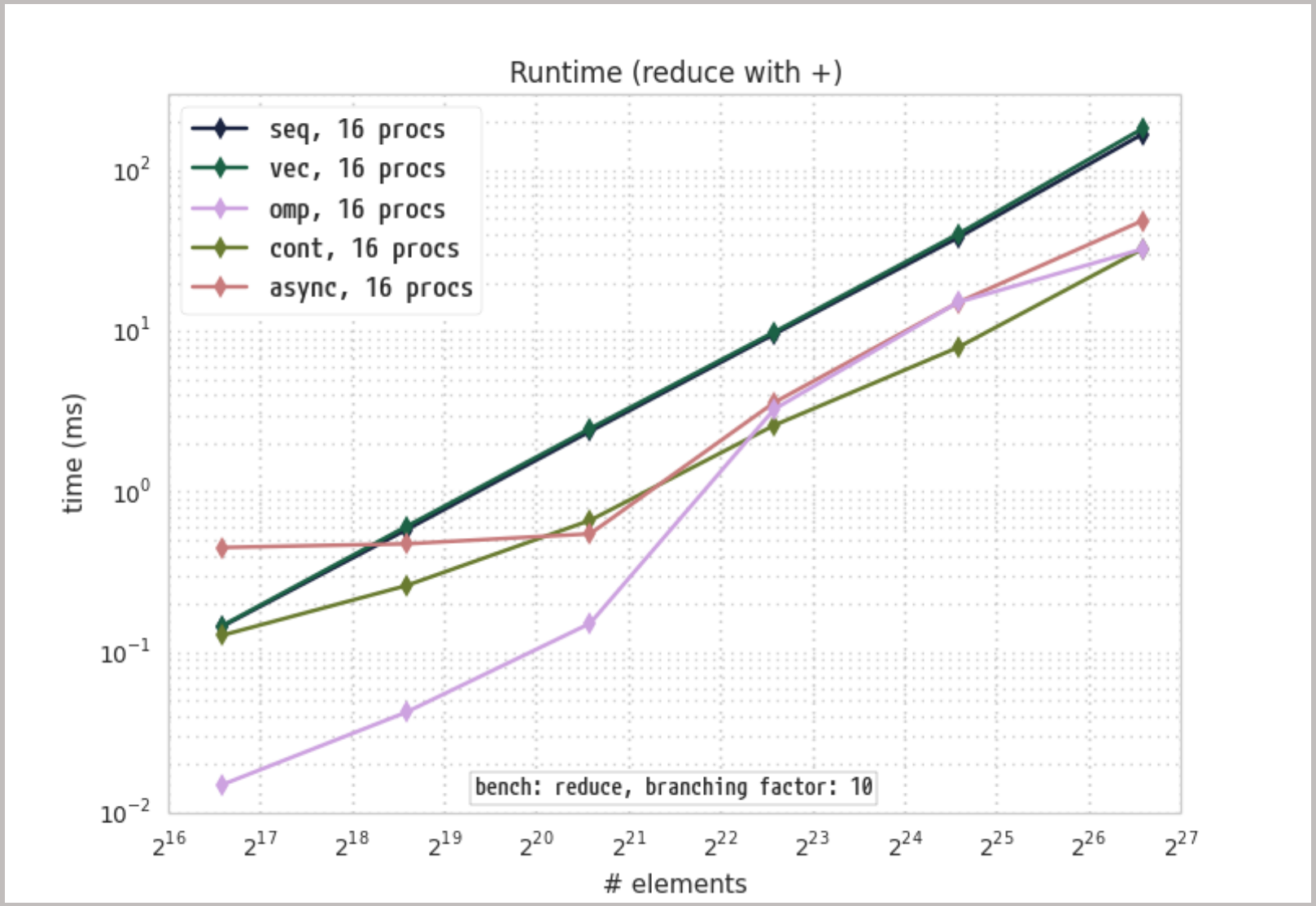
- vs serial implementation

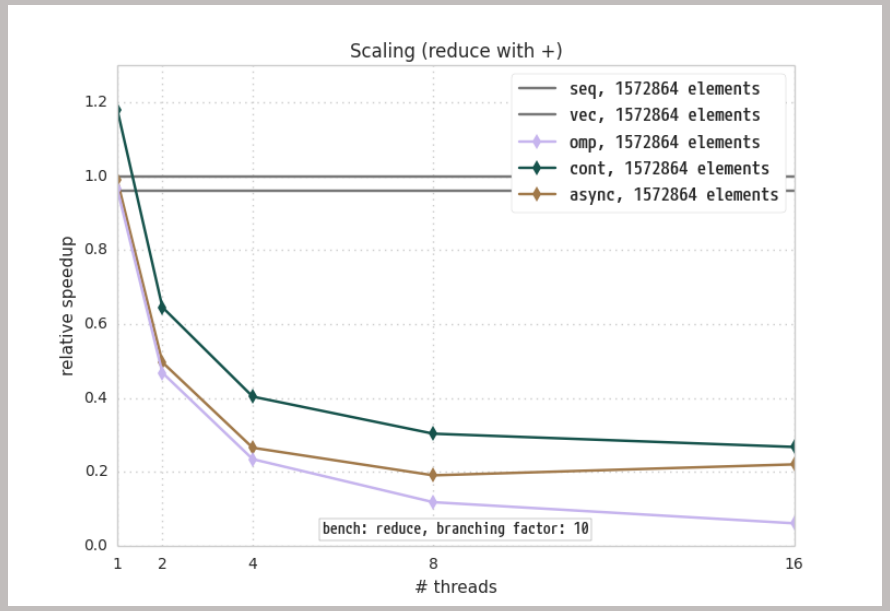
Heat: 1D spatial finite difference solver for heat equation

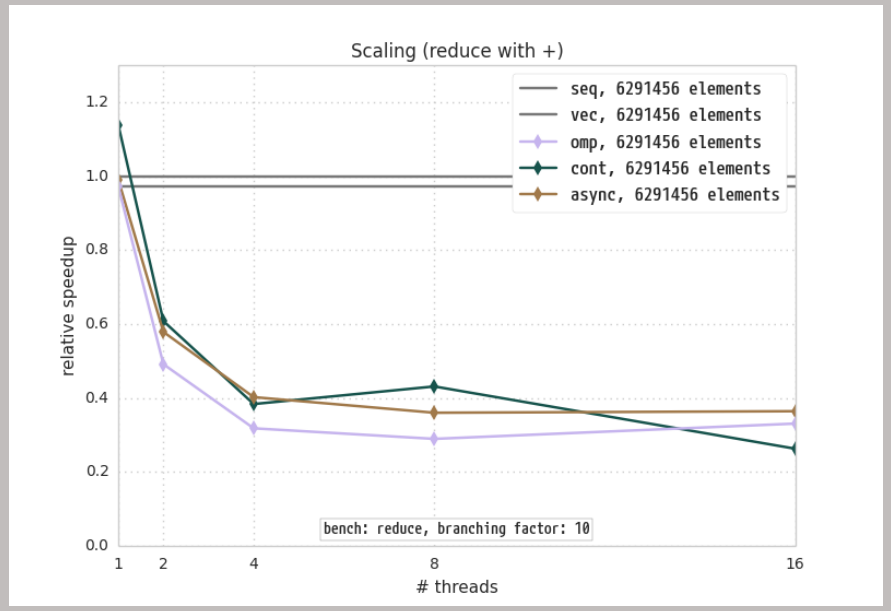
- vs serial implementation

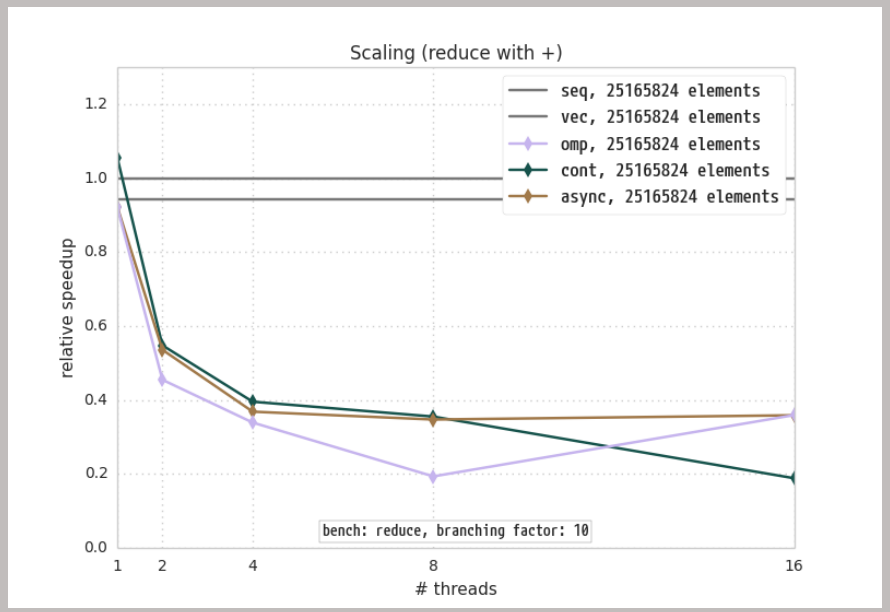
Results

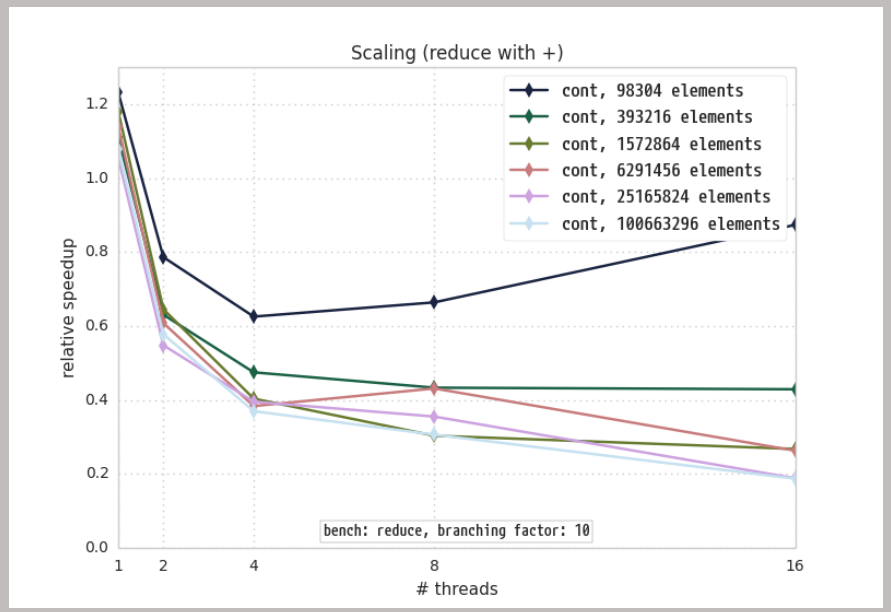
Reduce


Runtime (reduce with +)
seq, 16 procs
vec, 16 procs
omp, 16 procs
cont, 16 procs
async, 16 procs
time (ms)
10^2
10^1
10^0
10^{-1}
10^{-2}
bench: reduce, branching factor: 10
2^{16} 2^{17} 2^{18} 2^{19} 2^{20} 2^{21} 2^{22} 2^{23} 2^{24} 2^{25} 2^{26} 2^{27}
elements

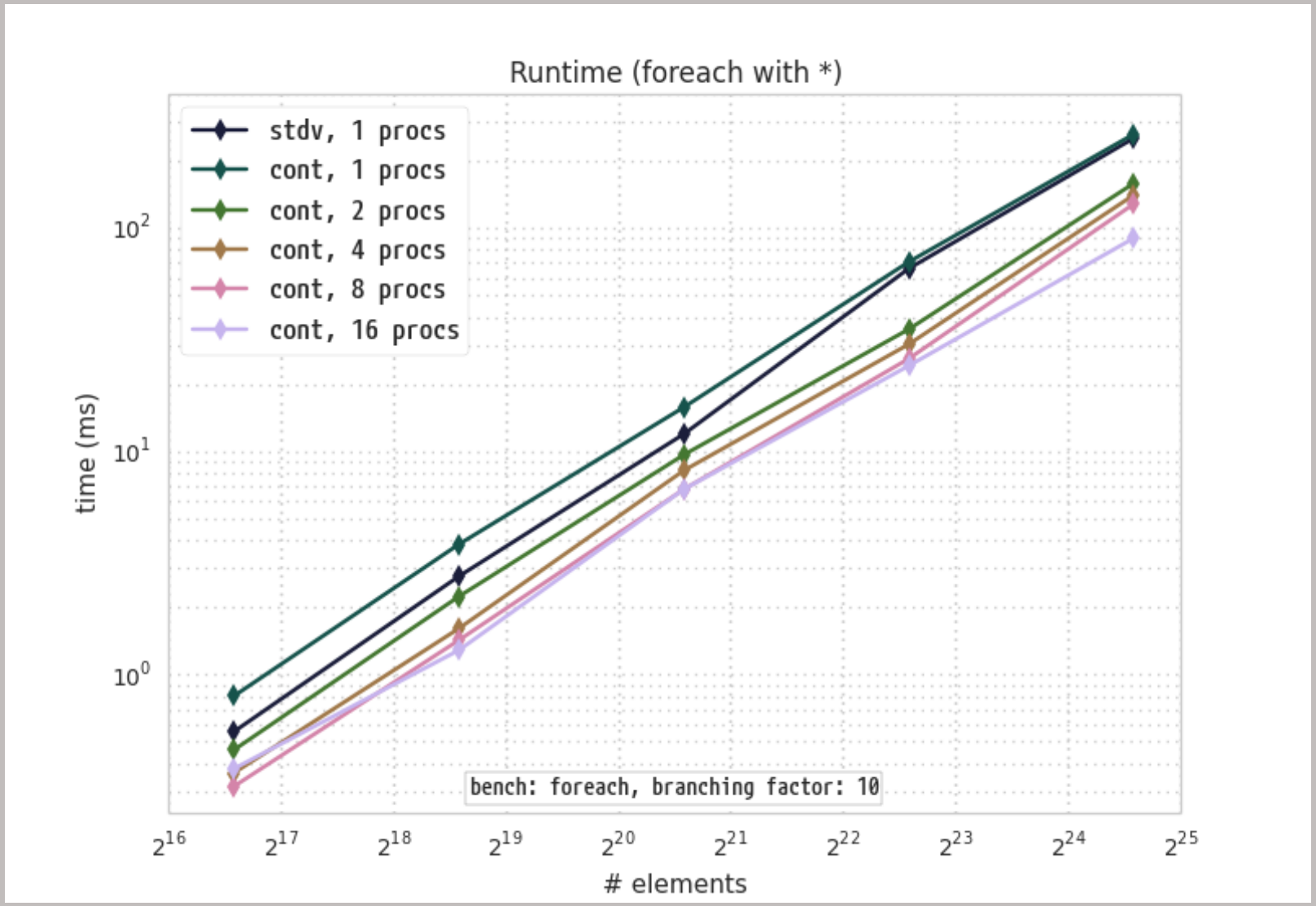

Scaling (reduce with +)
seq, 1572864 elements
vec, 1572864 elements
omp, 1572864 elements
cont, 1572864 elements
async, 1572864 elements
relative speedup
threads
bench: reduce, branching factor: 10

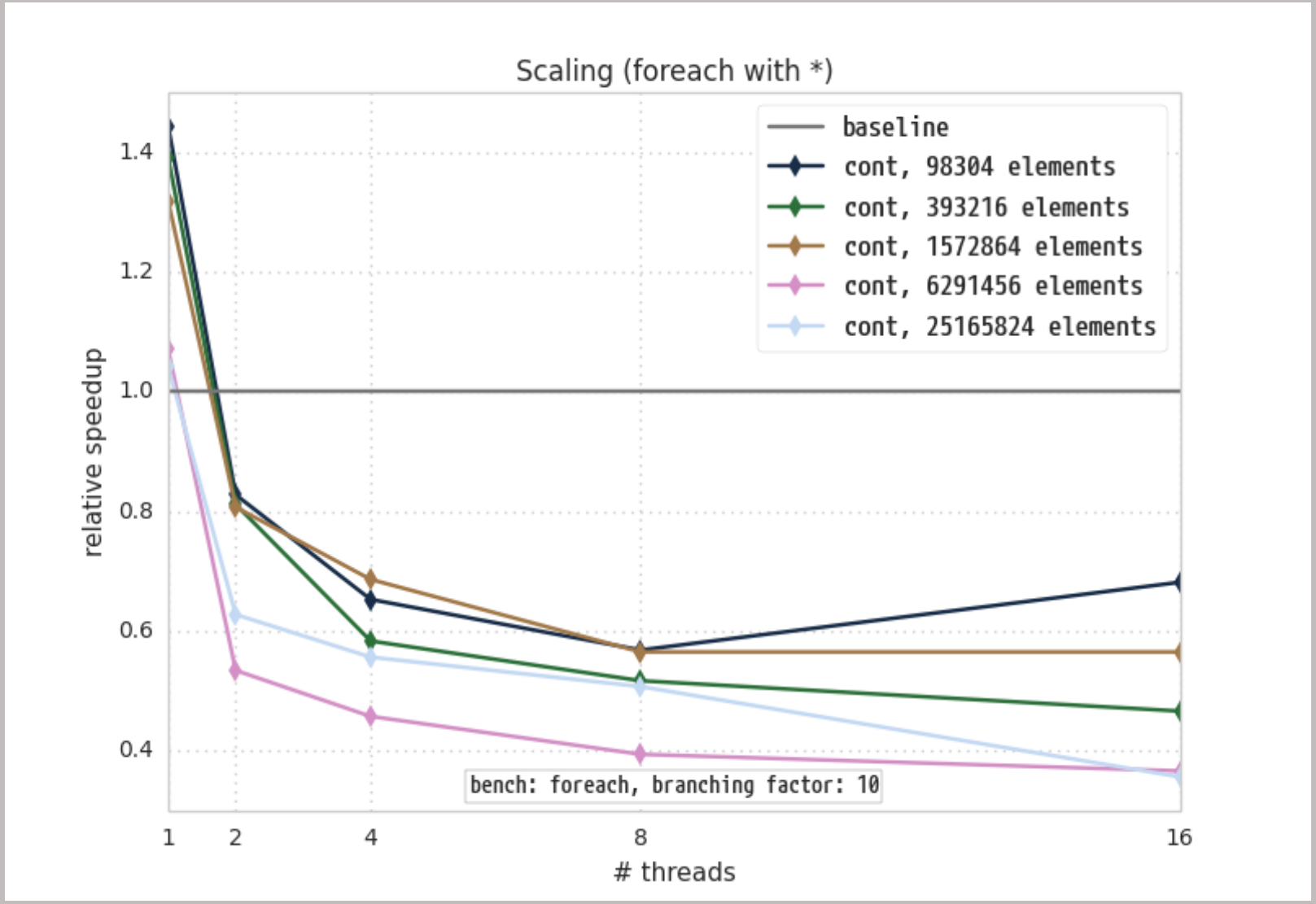

Scaling (reduce with +)
seq, 6291456 elements
vec, 6291456 elements
omp, 6291456 elements
cont, 6291456 elements
async, 6291456 elements
relative speedup
threads
bench: reduce, branching factor: 10


Scaling (reduce with +)
seq, 25165824 elements
vec, 25165824 elements
omp, 25165824 elements
cont, 25165824 elements
async, 25165824 elements
relative speedup
threads
bench: reduce, branching factor: 10

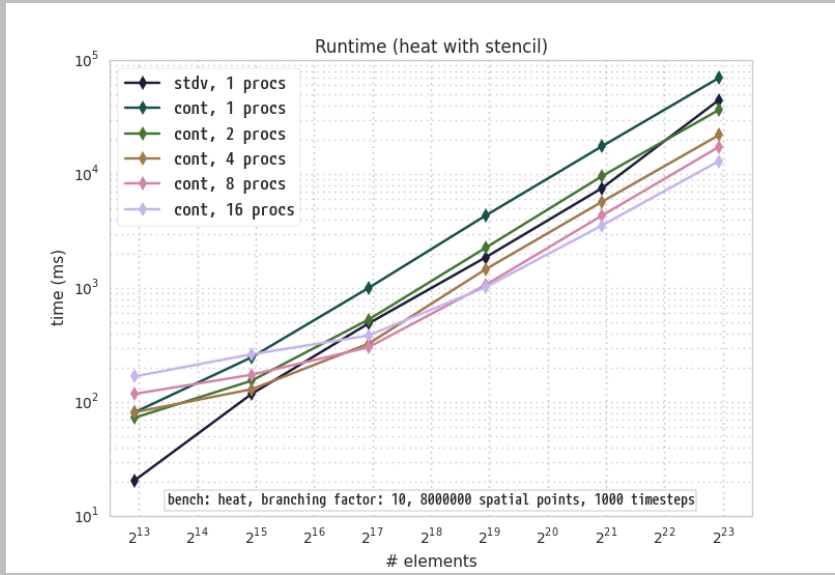

Scaling (reduce with +)
cont, 98304 elements
cont, 393216 elements
cont, 1572864 elements
cont, 6291456 elements
cont, 25165824 elements
cont, 100663296 elements
relative speedup
threads
bench: reduce, branching factor: 10

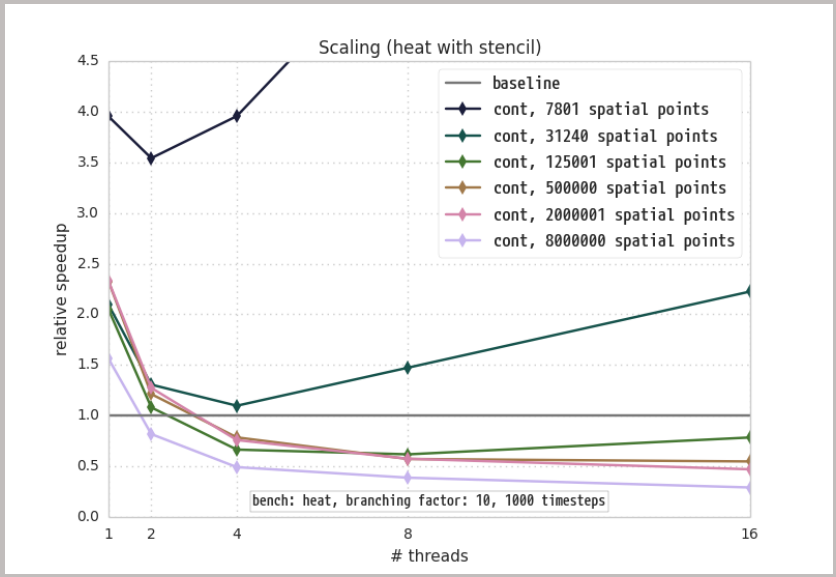
Foreach

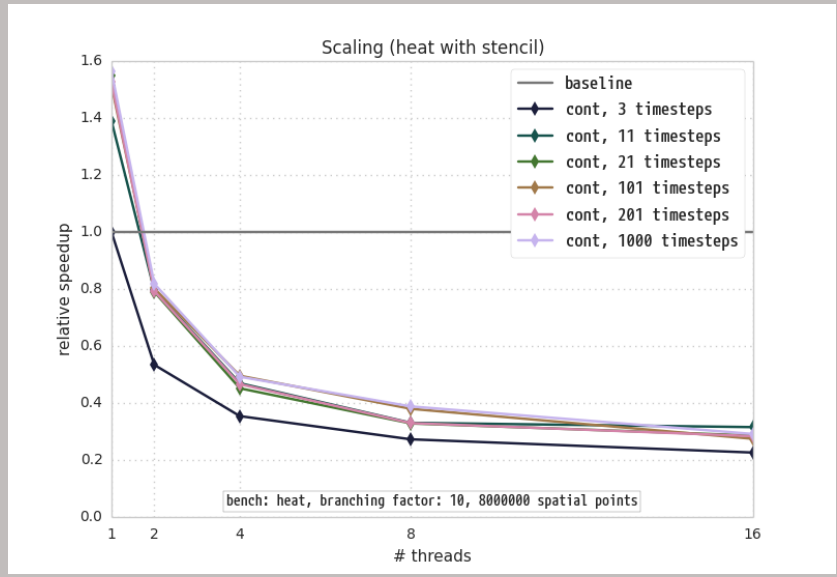

Runtime (foreach with *)
stdv, 1 procs
cont, 1 procs
cont, 2 procs
cont, 4 procs
cont, 8 procs
cont, 16 procs
time (ms)
10^2
10^1
10^0
bench: foreach, branching factor: 10
2^{16} 2^{17} 2^{18} 2^{19} 2^{20} 2^{21} 2^{22} 2^{23} 2^{24} 2^{25}
elements


Scaling (foreach with *)
baseline
cont, 98304 elements
cont, 393216 elements
cont, 1572864 elements
cont, 6291456 elements
cont, 25165824 elements
relative speedup
1.4
1.2
1.0
0.8
0.6
0.4
bench: foreach, branching factor: 10
1
2
4
8
16
threads

Heat


Runtime (heat with stencil)
time (ms)
elements
stdv, 1 procs
cont, 1 procs
cont, 2 procs
cont, 4 procs
cont, 8 procs
cont, 16 procs
bench: heat, branching factor: 10, 8000000 spatial points, 1000 timesteps


Scaling (heat with stencil)
relative speedup
threads
baseline
cont, 7801 spatial points
cont, 31240 spatial points
cont, 125001 spatial points
cont, 500000 spatial points
cont, 2000001 spatial points
cont, 8000000 spatial points
bench: heat, branching factor: 10, 1000 timesteps


Scaling (heat with stencil)
relative speedup
threads
baseline
cont, 3 timesteps
cont, 11 timesteps
cont, 21 timesteps
cont, 101 timesteps
cont, 201 timesteps
cont, 1000 timesteps
bench: heat, branching factor: 10, 8000000 spatial points

Observations

Where to Go From Here

Future Directions

Test Pre

Test

```
#! /usr/bin/Rscript --vanilla
library(knitr)
library(rmarkdown)
file <- list.files(pattern='.Rmd')
rmarkdown::render(file)
```