Channels, Concurrency, and Cores

A story of Concurrent ML

Andy Wingo ~ wingo@igalia.com

wingolog.org ~ @andywingo

agenda

An accidental journey

Concurrency quest

Making a new CML

A return

start from home

Me: Co-maintainer of Guile Scheme

Concurrency in Guile: POSIX threads

A gnawing feeling of wrongness

pthread gnarlies

Not compositional

Too low-level

Not I/O-scalable

Recommending pthreads is malpractice

fibers: a new hope

- Lightweight threads
- Built on coroutines (delimited continuations, prompts)
- Suspend on blocking I/O
- Epoll to track fd activity
- Multiple worker cores

the sages of rome

Last year...

Me: Lightweight fibers for I/O, is it the right thing?

Matthias Felleisen, Matthew Flatt: Yep but see Concurrent ML

Me: orly. kthx

MF & MF: np

time to learn

Concurrent ML: What is this thing?

How does it relate to what people know from Go, Erlang?

Is it worth it?

But first, a bit of context...

from pl to os

Event-based concurrency

```
(define (run sched)
  (match sched
    (($ $sched inbox i/o)
     (define (dequeue-tasks)
       (append (dequeue-all! inbox)
               (poll-for-tasks i/o)))
     (let lp ((runq (dequeue-tasks)))
       (match runq
         ((t . runq)
          (begin (t) (lp runq)))
         (()
          (lp (dequeue-tasks))))))))
```

from pl to os

```
(match sched
  (($ $sched inbox i/o)
   ...))
```

Enqueue tasks by posting to `inbox`

Register pending I/O events on `i/o` (`epoll` fd and callbacks)

Check for I/O after running current queue

Next: layer threads on top

```scheme
(define tag (make-prompt-tag))

(define (call/susp fn args)
  (define (body) (apply fn args))
  (define (handler k on-suspend) (on-suspend k))
  (call-with-prompt tag body handler))

(define (suspend on-suspend)
  (abort-to-prompt tag on-suspend))

(define (schedule k . args)
  (match (current-scheduler)
    (($ $sched inbox i/o)
     (enqueue! inbox (lambda () (call/susp k args))))))
```

suspend to yield

```
(define (spawn-fiber thunk)
  (schedule thunk))

(define (yield)
  (suspend schedule))

(define (wait-for-readable fd)
  (suspend
   (lambda (k)
     (match (current-scheduler)
       (($ $sched inbox i/o)
        (add-read-fd! i/o fd k))))))
```

back in rome

Channels and fibers?

Felleisen & Flatt: CML.

Me: Can we not tho

Mike Sperber: CML; you will have to reimplement otherwise

Me: ...

channels

Tony Hoare in 1978: Communicating Sequential Processes (CSP)

“Processes” rendezvous to exchange values

Unbuffered! Not async queues; Go, not Erlang

channel recv

```
(define (recv ch)
  (match ch
    (($ $channel recvq sendq)
     (match (try-dequeue! sendq)
       (#(value resume-sender)
        (resume-sender)
        value)
       (#f
        (suspend
         (lambda (k)
           (enqueue! recvq k))))))))
```

(Spot the race?)

select begets ops

Wait on 1 of N channels: `select`

Not just `recv`

(select (*recv A*) (*send B*))

Abstract channel operation as data

```
(select (recv-op A) (send-op B))
```

Abstract `select` operation

```
(define (select . ops)
  (perform (apply choice-op ops)))
```

which op happened?

Missing bit: how to know which operation actually occured

`(wrap-op op k)`: if *op* occurs, pass its result values to *k*

```
(perform
 (wrap-op
  (recv-op A)
  (lambda (v)
    (string-append "hello, " v))))
```

If performing this op makes a rendezvous with fiber sending "world", result is "hello, world"

this is cml

John Reppy PLDI 1988:
"Synchronous operations as first-class values"

`exp` : `(lambda () exp)`

`(recv ch)` : `(recv-op ch)`

PLDI 1991: "CML: A higher-order concurrent language"

Note use of "perform/op" instead of "sync/event"

what’s an op?

Recall structure of channel recv:

- Optimistic: value ready; we take it and resume the sender
- Pessimistic: suspend, add ourselves to recvq

(Spot the race?)

what’s an op?

General pattern

Optimistic phase: Keep truckin’

- *commit* transaction
- *resume* any other parties to txn

Pessimistic phase: Park the truck

- *suspend* thread
- *publish* fact that we are waiting
- *recheck* if txn became completable

what’s an op?

```
(define (perform op)
  (match optimistic
    (#f pessimistic)
    (thunk (thunk))))
```

Op: data structure with `try`, `block`, and `wrap` fields

Optimistic case runs op’s `try` fn

Pessimitic case runs op’s `block` fn

channel recv-op try

```
(define (try-recv ch)
  (match ch
    (($ $channel recvq sendq)
     (match (atomic-ref sendq)
       (() #f)
       ((and q (head . tail))
        (match head
          (#(val resume-sender state)
           (match (CAS! state 'W 'S)
             ('W
              (resume-sender)
              (CAS! sendq q tail) ; ?
              (lambda () val))
             (_ #f)))))))))
```

when there is no try

`try` function succeeds? Caller does not suspend

Otherwise pessimistic case; three parts:

```
(define (pessimistic block)
  ;; 1. Suspend the thread
  (suspend
   (lambda (k)
     ;; 2. Make a fresh opstate
     (let ((state (fresh-opstate)))
       ;; 3. Call op's block fn
       (block k state)))))
```

opstates

Operation state (“opstate”): atomic state variable

- `W`: “Waiting”; initial state
- `C`: “Claimed”; temporary state
- `S`: “Synched”; final state

Local transitions `W->C`, `C->W`, `C->S`

Local and remote transitions: `W->S`

Each instantiation of an operation gets its own state: operations reusable

channel recv-op block

Block fn called after thread suspend

Two jobs: publish resume fn and opstate to channel's `recvq`, then try again to receive

Three possible results of retry:

- Success? Resume self and other
- Already in `S` state? Someone else resumed me already (race)
- Can't even? Someone else will resume me in the future

```
(define (block-recv ch resume-recv recv-state)
  (match ch
    (($ $channel recvq sendq)
     ;; Publish -- now others can resume us!
     (enqueue! recvq (vector resume-recv recv-state))
     ;; Try again to receive.
     (let retry ()
       (match (atomic-ref sendq)
         (() #f)
         ((and q (head . tail))
          (match head
            (#(val resume-send send-state)
             ;; Next slide :)
            (_ #f)))))))))
```

```
(match (CAS! recv-state 'W 'C)      ; Claim our state
  ('W
   (match (CAS! send-state 'W 'S)
     ('W                              ; We did it!
      (atomic-set! recv-state 'S)
      (CAS! sendq q tail)             ; Maybe GC.
      (resume-send) (resume-recv val))
     ('C                              ; Conflict; retry.
      (atomic-set! recv-state 'W)
      (retry))
     ('S                              ; GC and retry.
      (atomic-set! recv-state 'W)
      (CAS! sendq q tail)
      (retry))))
  ('S #f))
```

ok that's it for code

Congratulations for getting this far

Also thank you

Left out only a couple details: `try` can loop if sender in `C` state, `block` needs to avoid sending to self

but what about select

`select` doesn't have to be a primitive!

`choose-op` `try` function runs all `try` functions of sub-operations (possibly in random order) returning early if one succeeds

`choose-op` `block` function does the same

Optimizations possible

cml is inevitable

Channel `block` implementation necessary for concurrent multicore send/receive

CML `try` mechanism is purely an optimization, but an inevitable one

CML is strictly more expressive than channels – for free

suspend thread

In a coroutine? Suspend by yielding

In a pthread? Make a mutex/cond and suspend by `pthread_cond_wait`

Same operation abstraction works for both: pthread<->pthread, pthread<->fiber, fiber<->fiber

lineage

1978: CSP, Tony Hoare

1983: occam, David May

1989, 1991: CML, John Reppy

2000s: CML in Racket, MLton, SML-NJ

2009: Parallel CML, Reppy et al

CML now: `manticore.cs.uchicago.edu`

This work: `github.com/wingo/fibers`

novelties

Reppy's CML uses three phases: `poll`, `do`, `block`

Fibers uses just two: there is no `do`, only `try`

Fibers channel implementation lockless: atomic sendq/recvq instead

Integration between fibers and pthreads

Given that `block` must re-check, `try` phase just an optimization

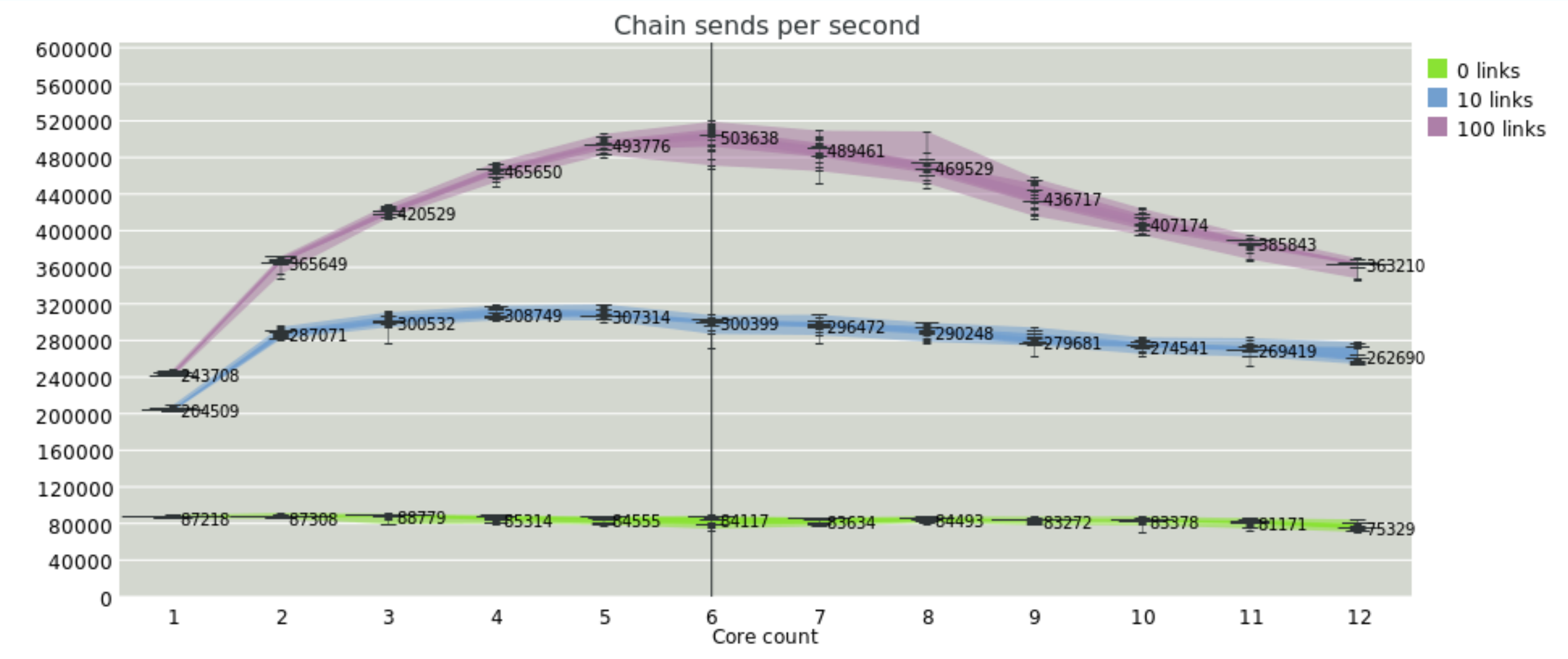
what about perf

Implementation: `github.com/wingo/fibers`, as a Guile library; goals:

- ❧ Dozens of cores, 100k fibers/core
- ❧ One `epoll` sched per core, sleep when idle
- ❧ Optionally pre-emptive
- ❧ Cross-thread wakeups via inbox

System: 2 x E5-2620v3 (6 2.6GHz cores/socket), hyperthreads off, performance cpu governor

Results mixed



Good: Speedups; Low variance

Bad: Diminishing returns; NUMA cliff; I/O poll costly

caveats

Sublinear speedup expected

- Overhead, not workload

Guile is bytecode VM; 0.4e9 insts retired/s on this machine

- Compare to 10.4e9 native at 4 IPC

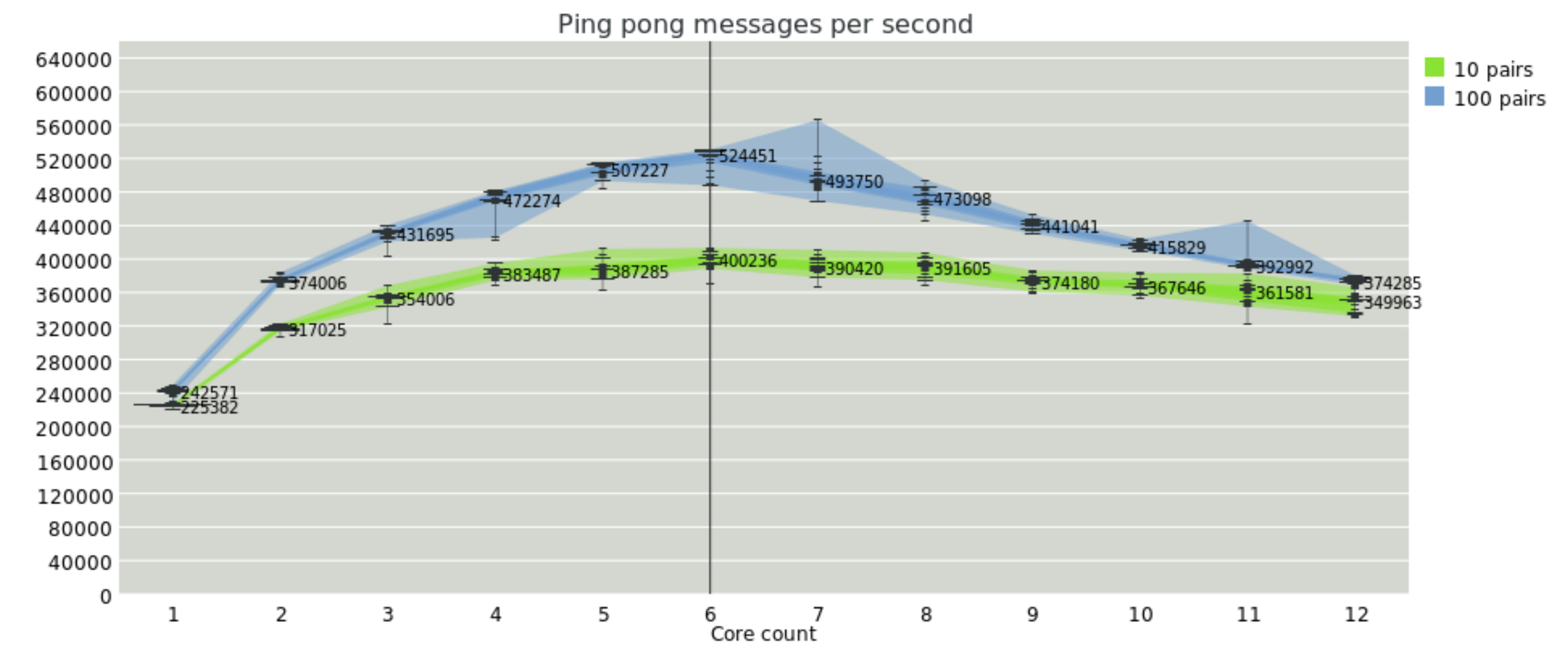
Can’t isolate test from Fibers

- `epoll` overhead, wakeup by fd

Can’t isolate test from GC

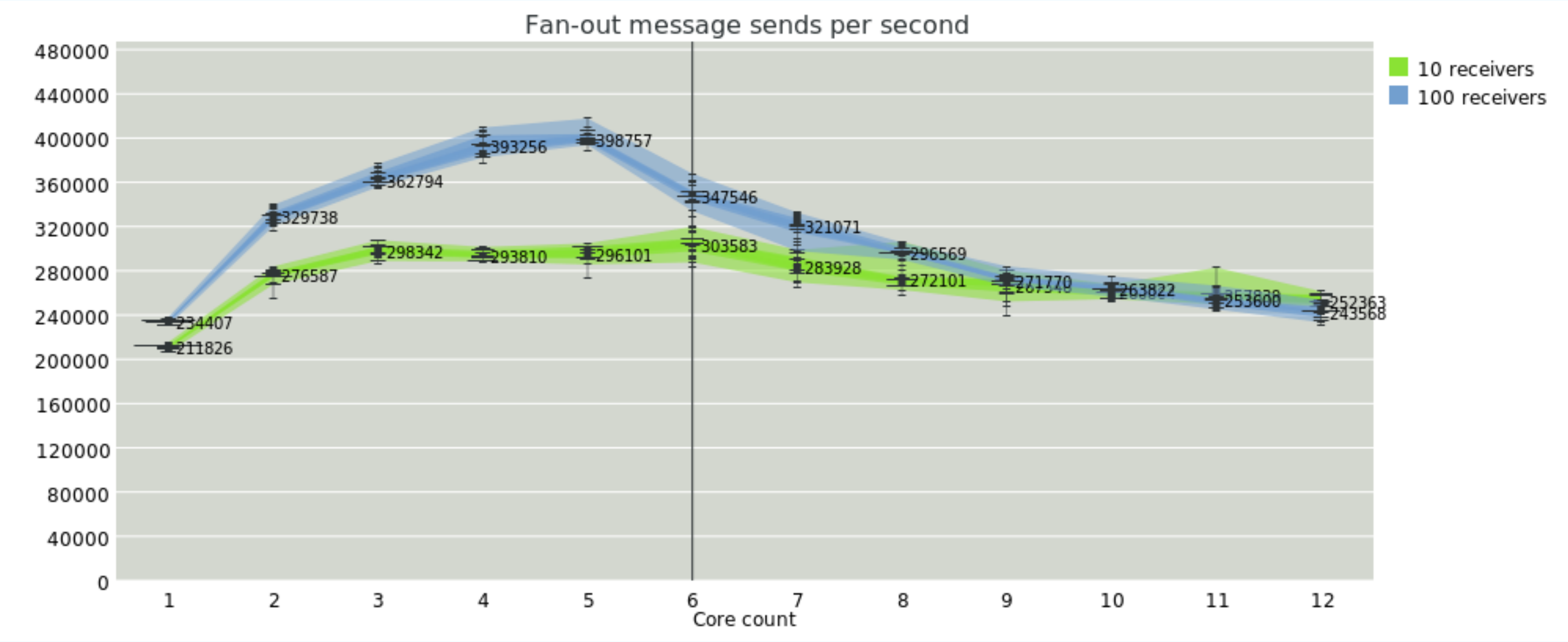
- STW parallel mark lazy sweep, STW via signals, NUMA-blind

Pairs of fibers passing messages; random core allocation



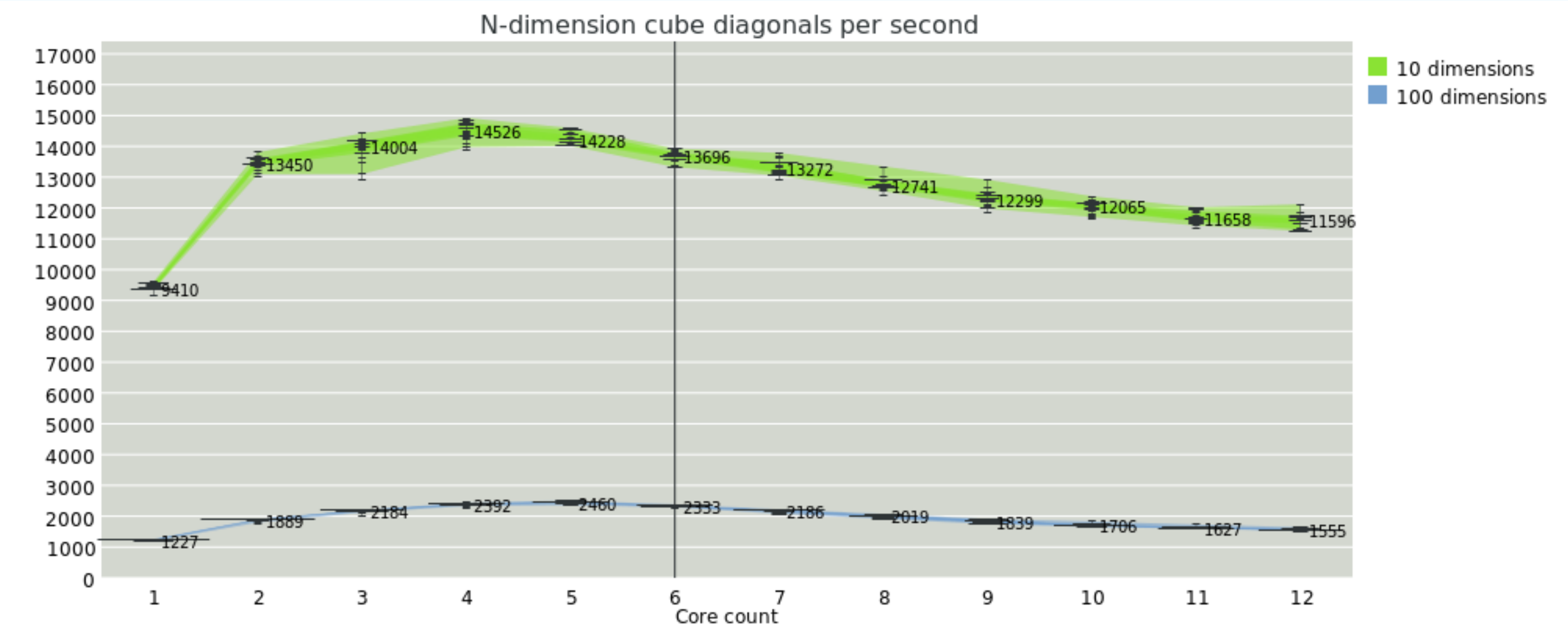
More runnable fibers per turn = less I/O overhead

One-to-n fan-out



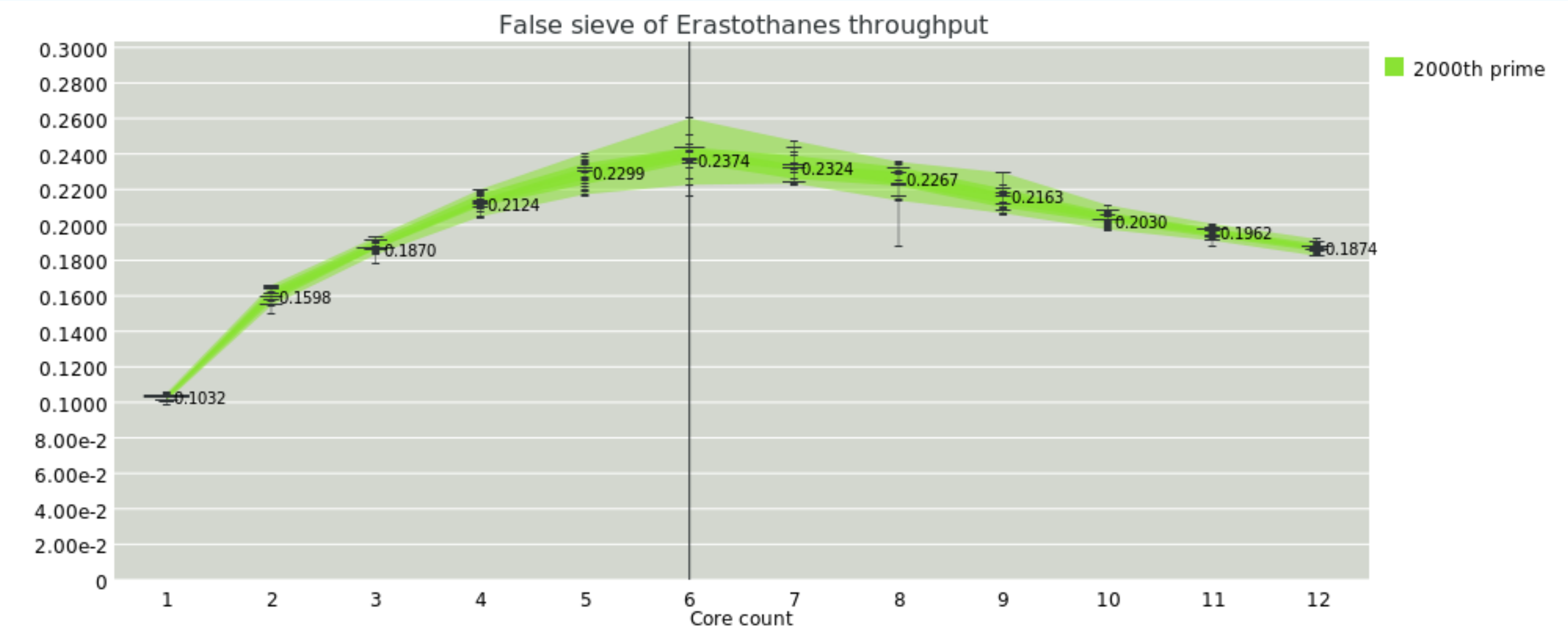
More "worker" fibers = less worker sleep/wake cost

n-dimensional cube diagonals



Very little workload; serial parts soon a bottleneck

False sieve of Erastothenes



Nice speedup, but NUMA cliff

but wait, there's more

CML "guard" functions

Other event types: cvars, timeouts, thread joins...

Patterns for building apps on CML: "Concurrent Programming in ML", John Reppy, 2007

CSP book: `usingcsp.com`

OCaml "Reagents" from Aaron Turon

and in the meantime

Possible to implement CML on top of channels+select: Vesa Karvonen's impl in F# and core.async

Limitations regarding self-sends

Right way is to layer channels on top of CML

summary

Language and framework developers: the sages were right, build CML!

You can integrate CML with existing code (thread pools etc)

`github.com/wingo/fibers`

`github.com/wingo/fibers/wiki/Manual`

Design systems with CSP, build them in CML

Happy hacking! ~ @andywingo