DYALOG
Eastbourne 2026

Improving APL Threads

Peter Mikkelsen



Intro – APL threads?

- Interpreter is single threaded (mostly)
- Thread 0
- `&` - spawn new thread
- Stack and context shared
- Any thread can spawn a new one

Intro – APL threads?

- Lightweight threads – not OS threads
- Interpreter handles scheduling
- Only one thread runs at any given time
- Switches at specific points only

Why use APL threads?

- Why bother with threads?
- ...not to speed up pure CPU bound applications
- Blocking operations
  - Allow the interpreter to run other APL code while it waits
- Independent things as different threads

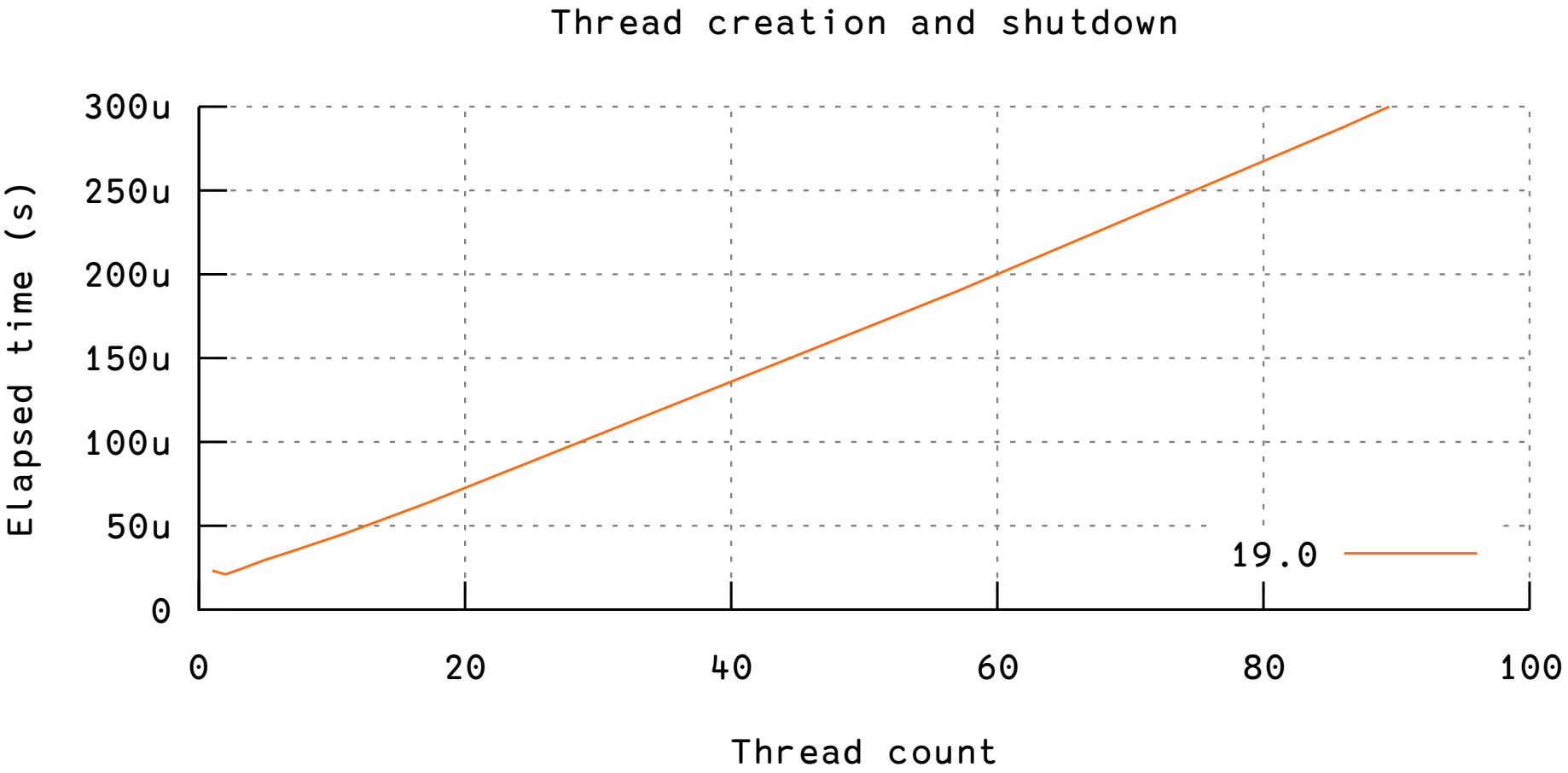
How many are enough?

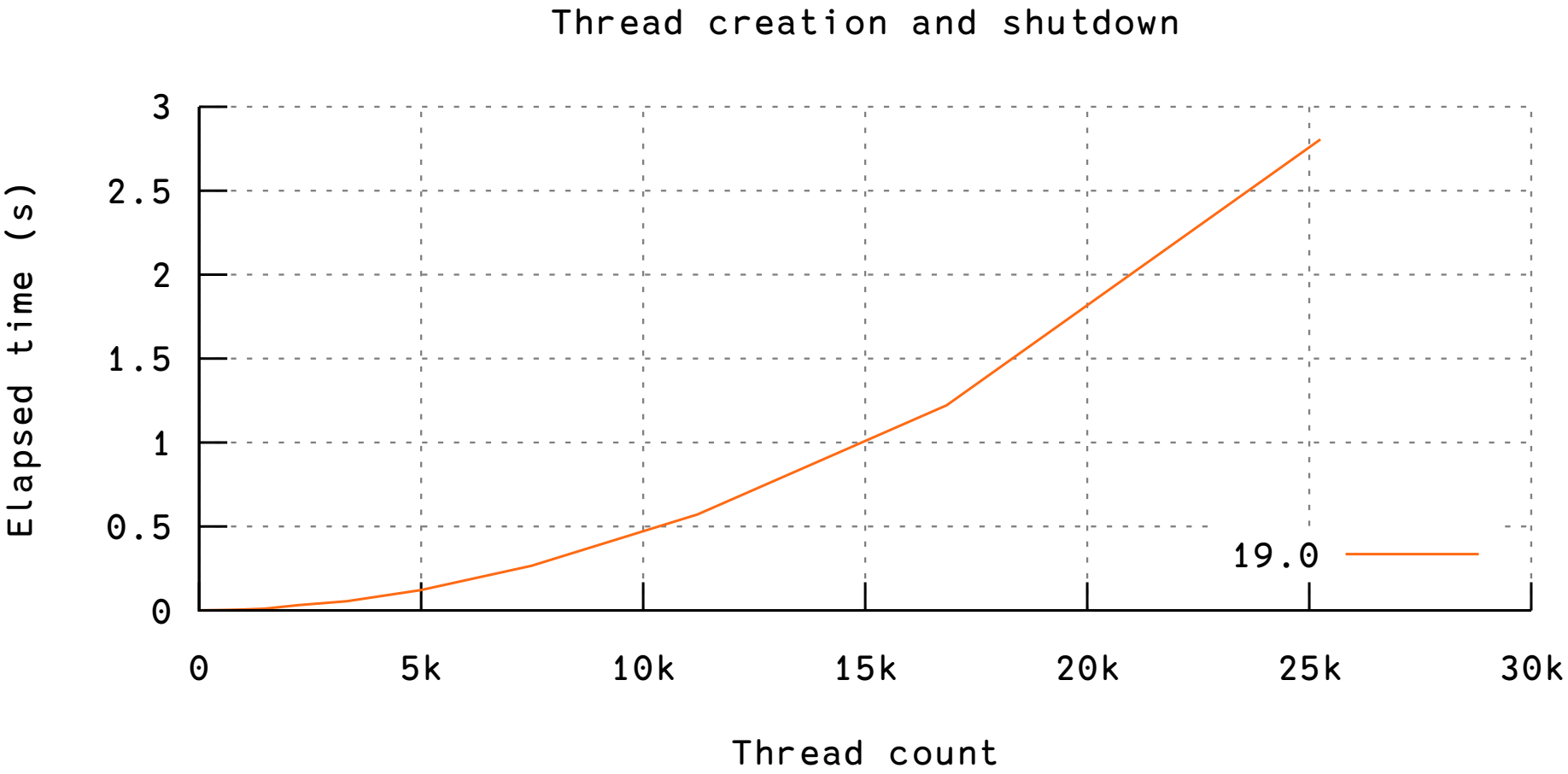
- 10?
- 100?
- 100000?
- Influences data structures and algorithms
- My opinion: as many as possible
  - Lets the user design their code as they see fit
  - Example: servers with 1 thread per connection

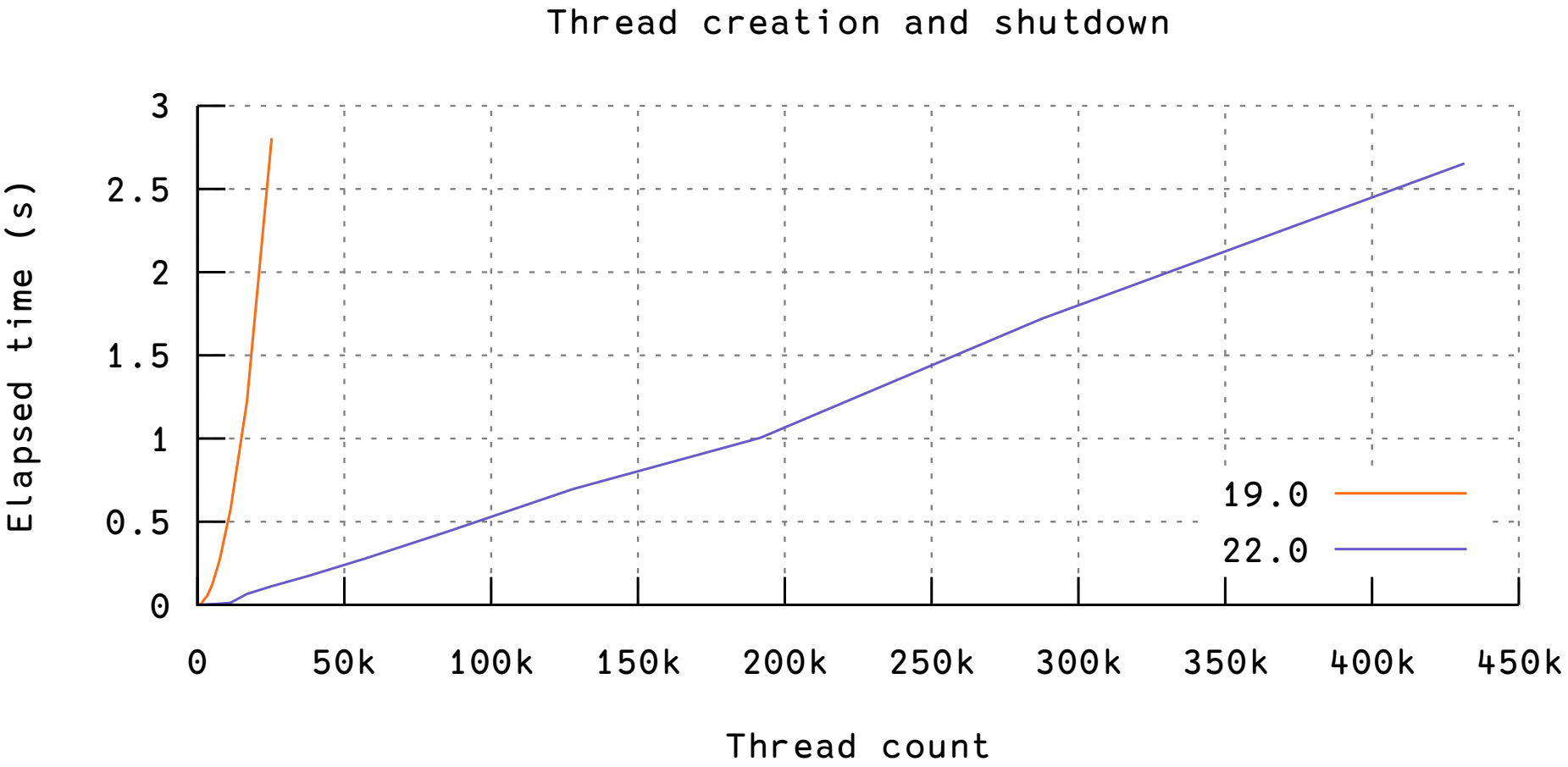
Overview of the presentation

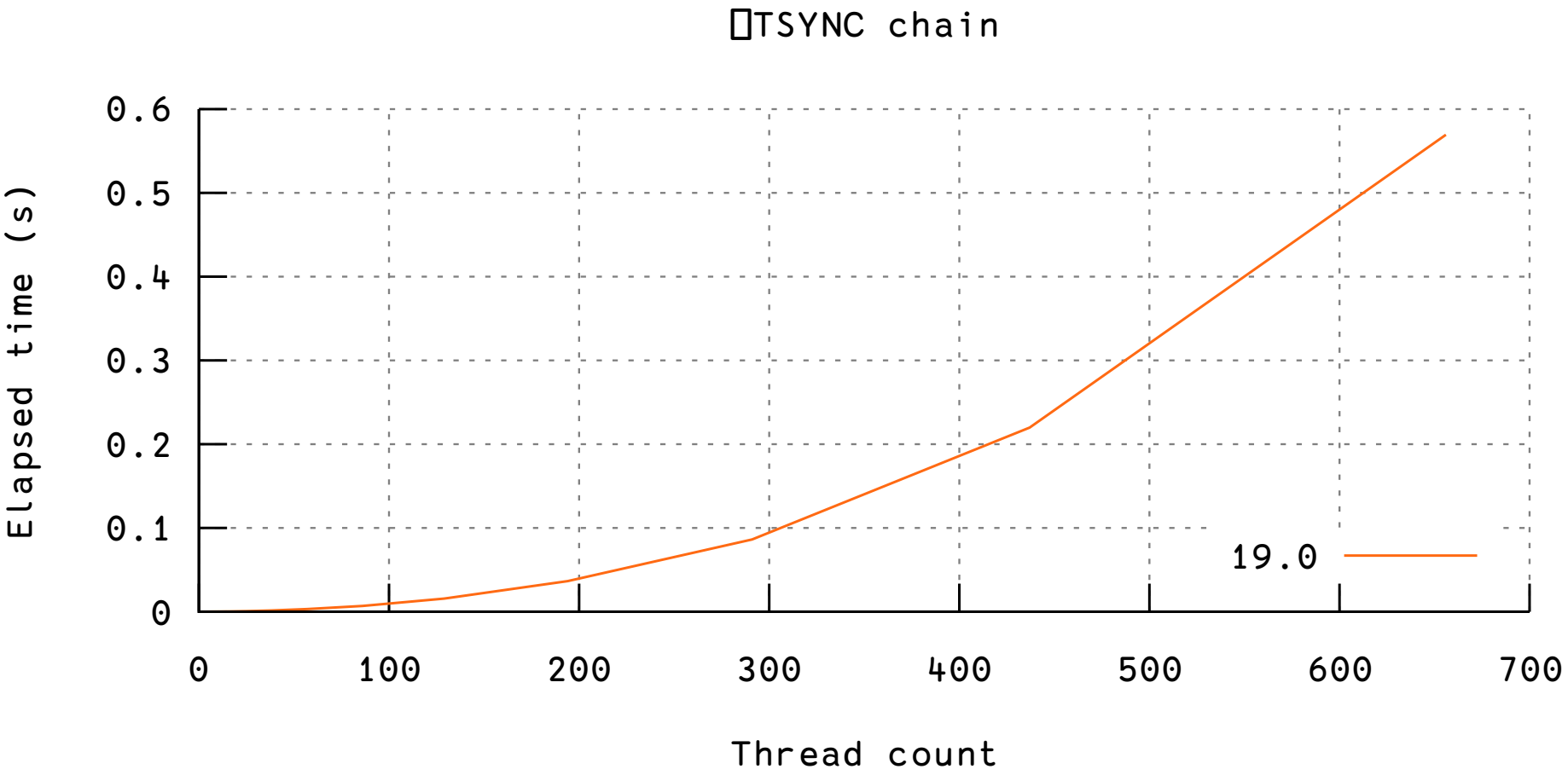
- A brief overview of what I noticed when digging into the code
  - And what we plan to improve soon
- Performance (scalability, predictability)
- Scheduling (overhead, properties)
- Deadlock detection (performance, completeness)

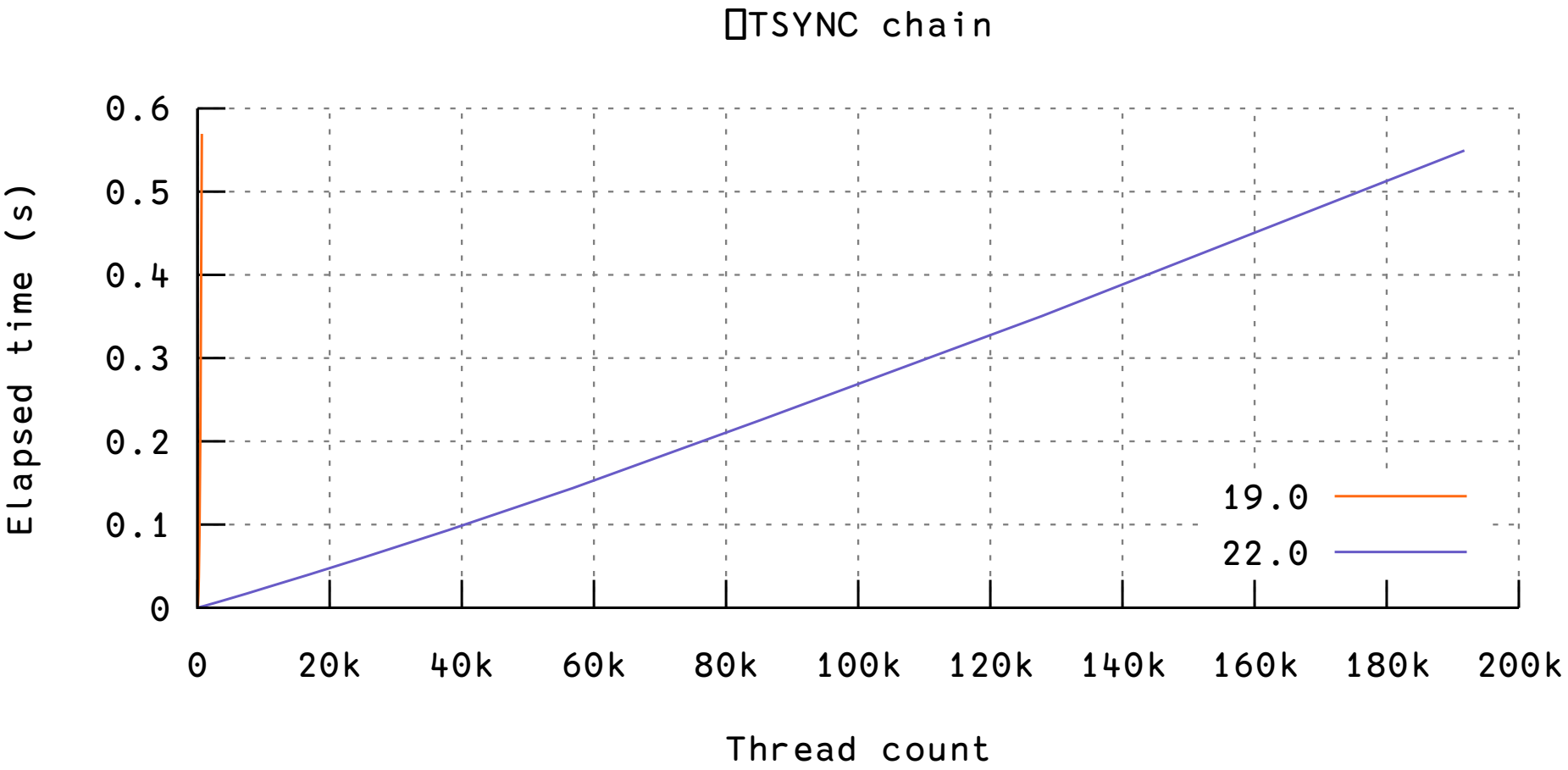
Part 1: Performance


Thread creation and shutdown
Elapsed time (s)
300u
250u
200u
150u
100u
50u
0
0
20
40
60
80
100
Thread count
19.0


Thread creation and shutdown
Elapsed time (s)
3
2.5
2
1.5
1
0.5
0
0
5k
10k
15k
20k
25k
30k
Thread count
19.0


Thread creation and shutdown
Elapsed time (s)
3
2.5
2
1.5
1
0.5
0
0
50k
100k
150k
200k
250k
300k
350k
400k
450k
Thread count
19.0
22.0


TSYNC chain
Elapsed time (s)
0.6
0.5
0.4
0.3
0.2
0.1
0
0
100
200
300
400
500
600
700
Thread count
19.0


TSYNC chain
Elapsed time (s)
0.6
0.5
0.4
0.3
0.2
0.1
0
0
20k
40k
60k
80k
100k
120k
140k
160k
180k
200k
Thread count
19.0
22.0

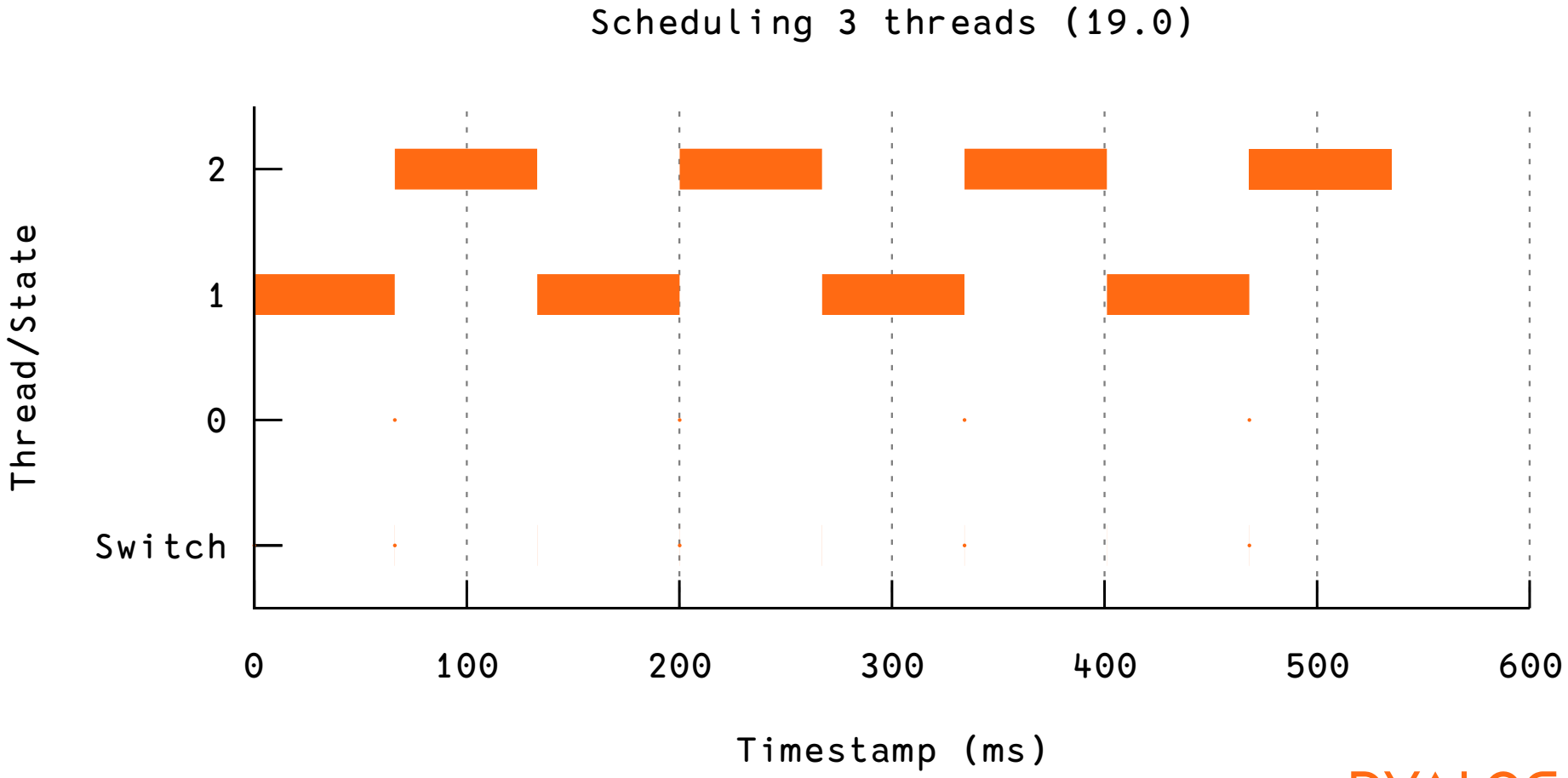
Performance summary

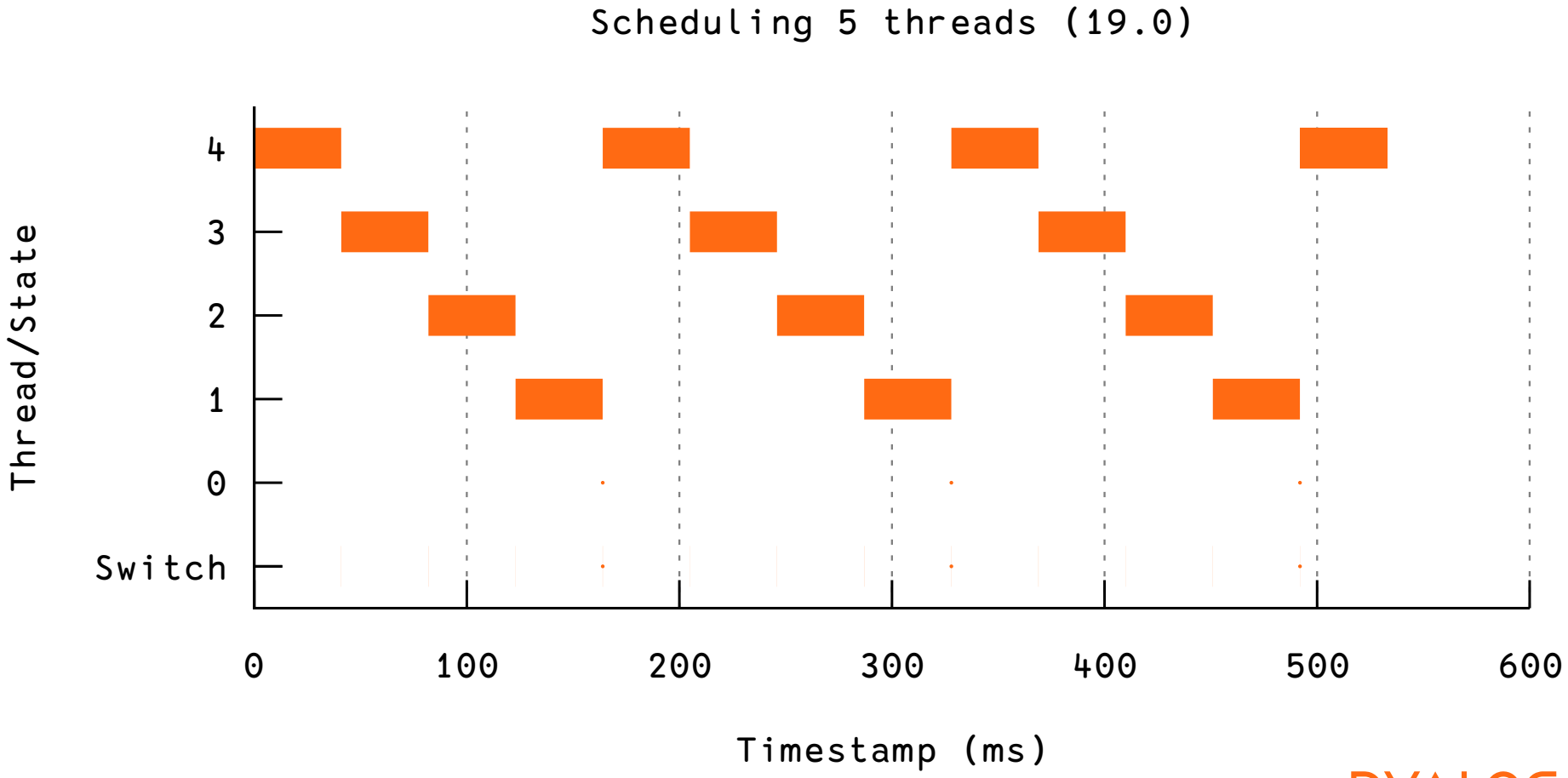
- Review of all the thread-related system functions
  - `⎕TGET`, `⎕TPUT`, `⎕TSYNC`, `⎕TKILL`, …
- Avoid looping over all threads
- Use better data structures/algorithms
- With many threads, a relatively big speedup

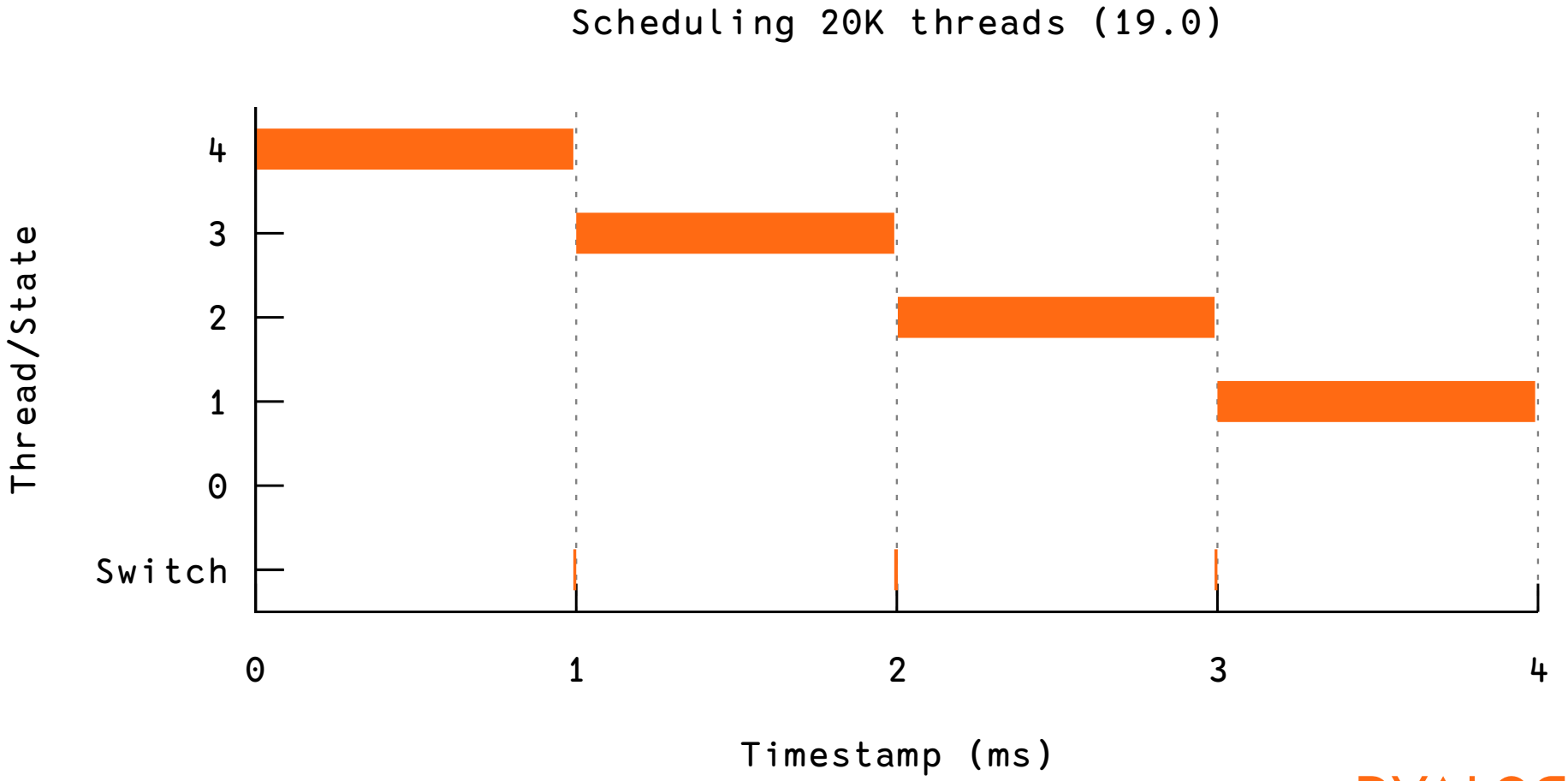
Part 2: Scheduler

Scheduler properties

- Fairness
  - Equally likely to run ↔ equal amount of time
- Responsiveness
  - How long until a thread gets scheduled
- Throughput
  - How much actual work gets done
  - Low overhead, and focus on tasks which need attention quickly

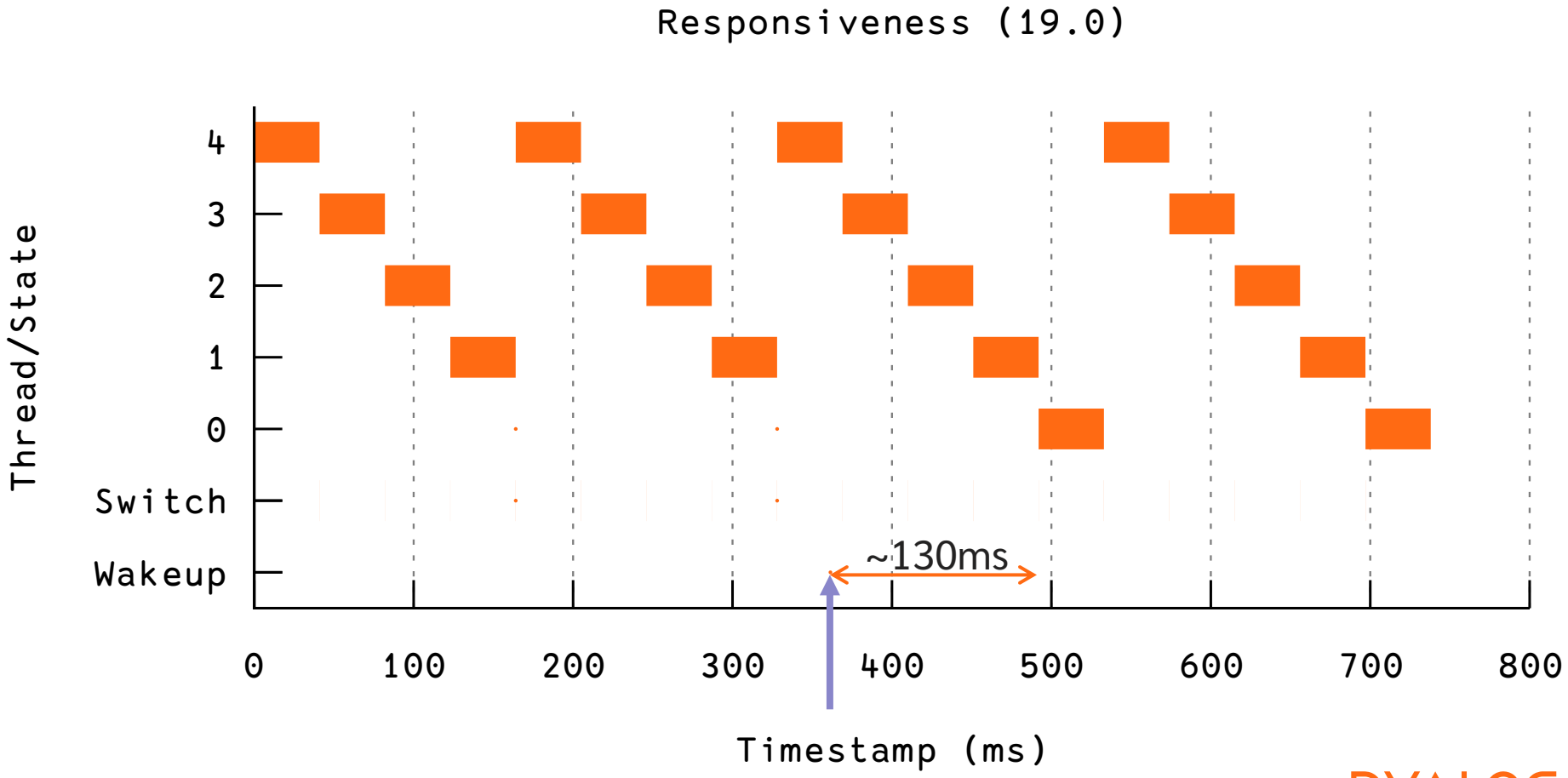

Scheduling 3 threads (19.0)
Thread/State
2
1
0
Switch
0
100
200
300
400
500
600
Timestamp (ms)


Scheduling 5 threads (19.0)
Thread/State
4
3
2
1
0
Switch
0
100
200
300
400
500
600
Timestamp (ms)


Scheduling 20K threads (19.0)
Thread/State
4
3
2
1
0
Switch
0
1
2
3
4
Timestamp (ms)

The old scheduling algorithm

- Give each thread `200ms÷NumberOfThreads`
  - Helps a bit with responsiveness, but increases overhead
- Check to see if the next one (in the vector) is ready
- Things can increase the overhead
  - Lots of threads which aren't ready
  - Thread states with more complicated "ready now?" checks
  - Stack size

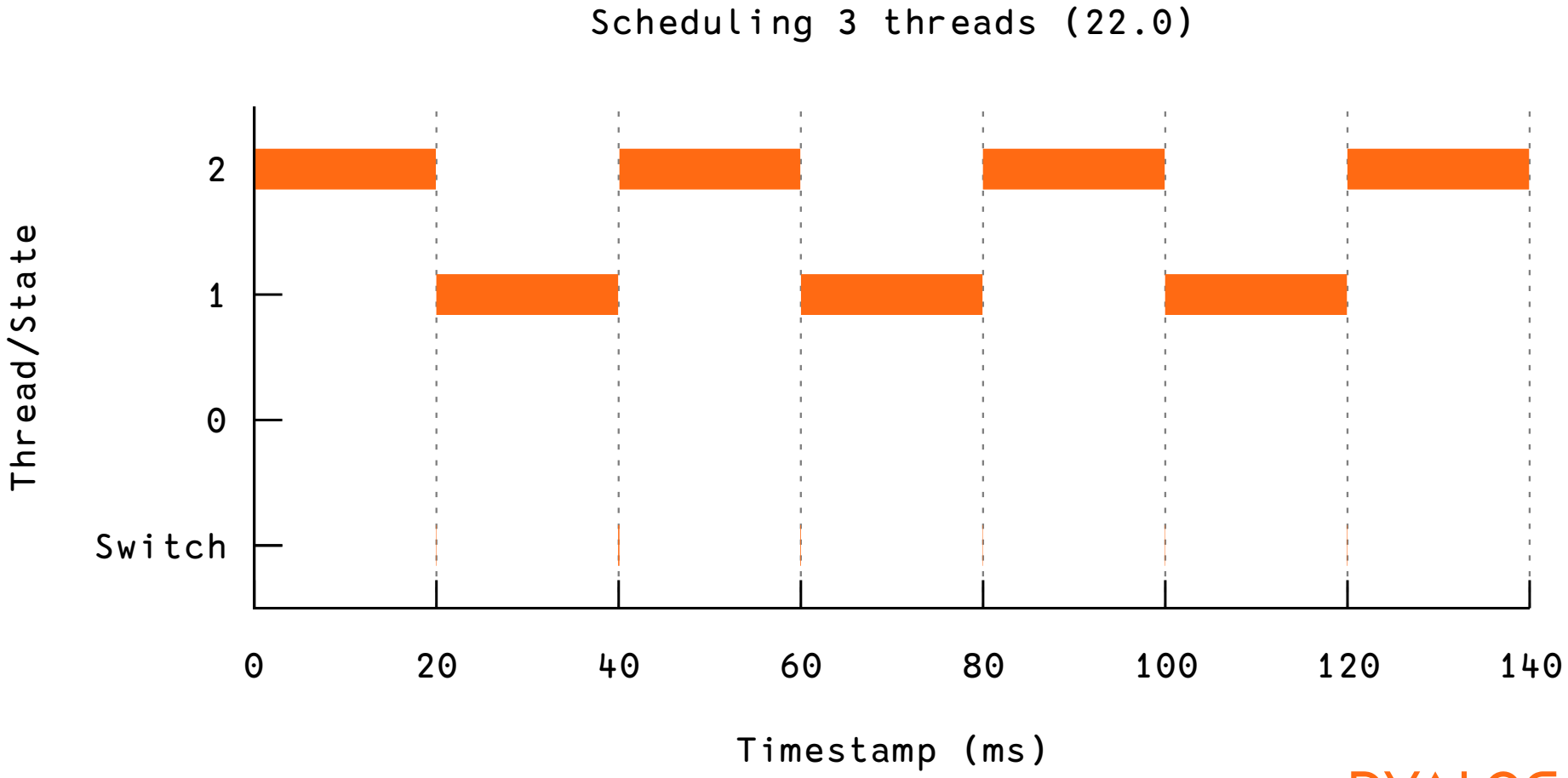

Responsiveness (19.0)
Thread/State
4
3
2
1
0
Switch
Wakeup
~130ms
0
100
200
300
400
500
600
700
800
Timestamp (ms)

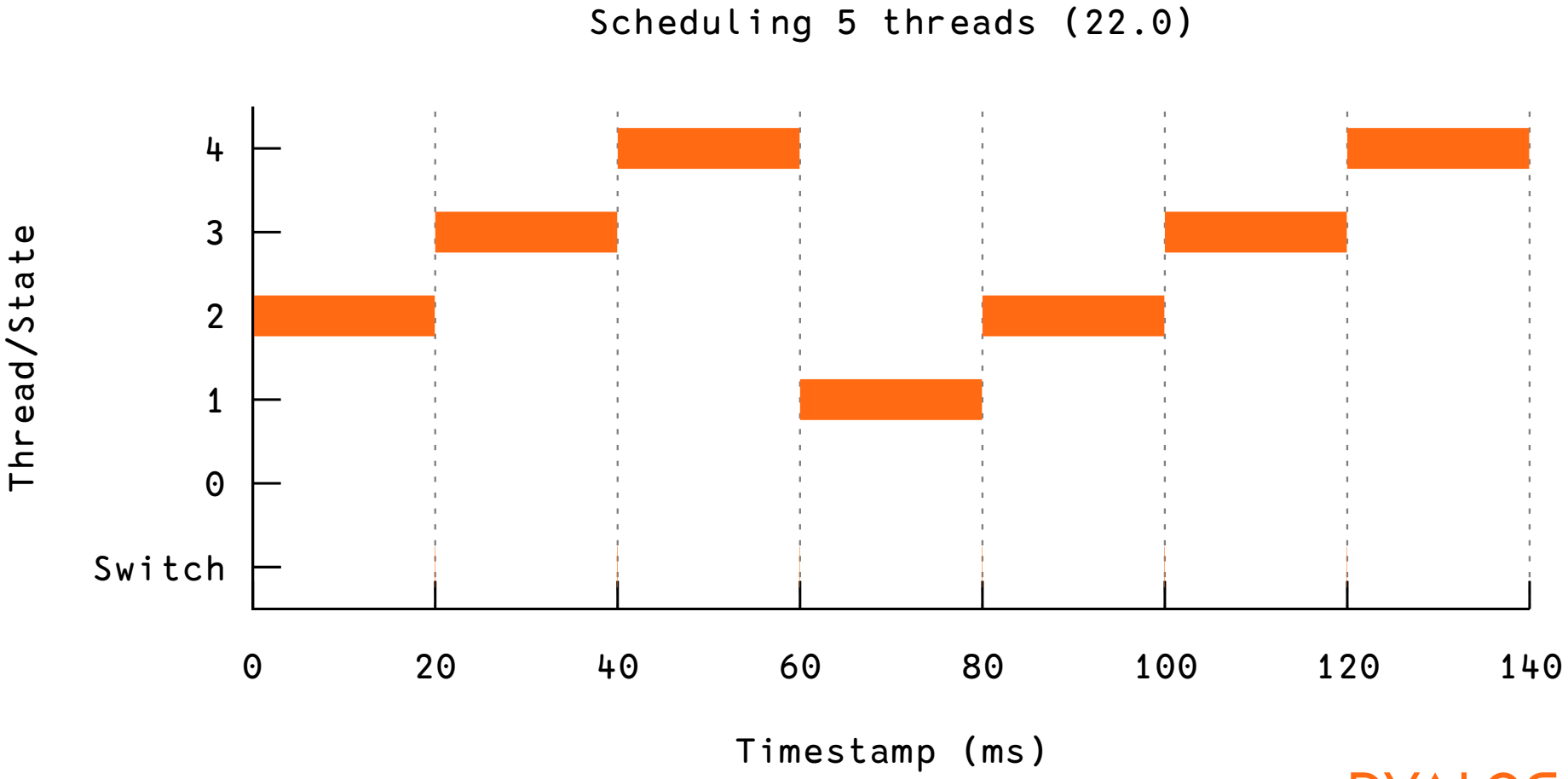
So what's the problem?

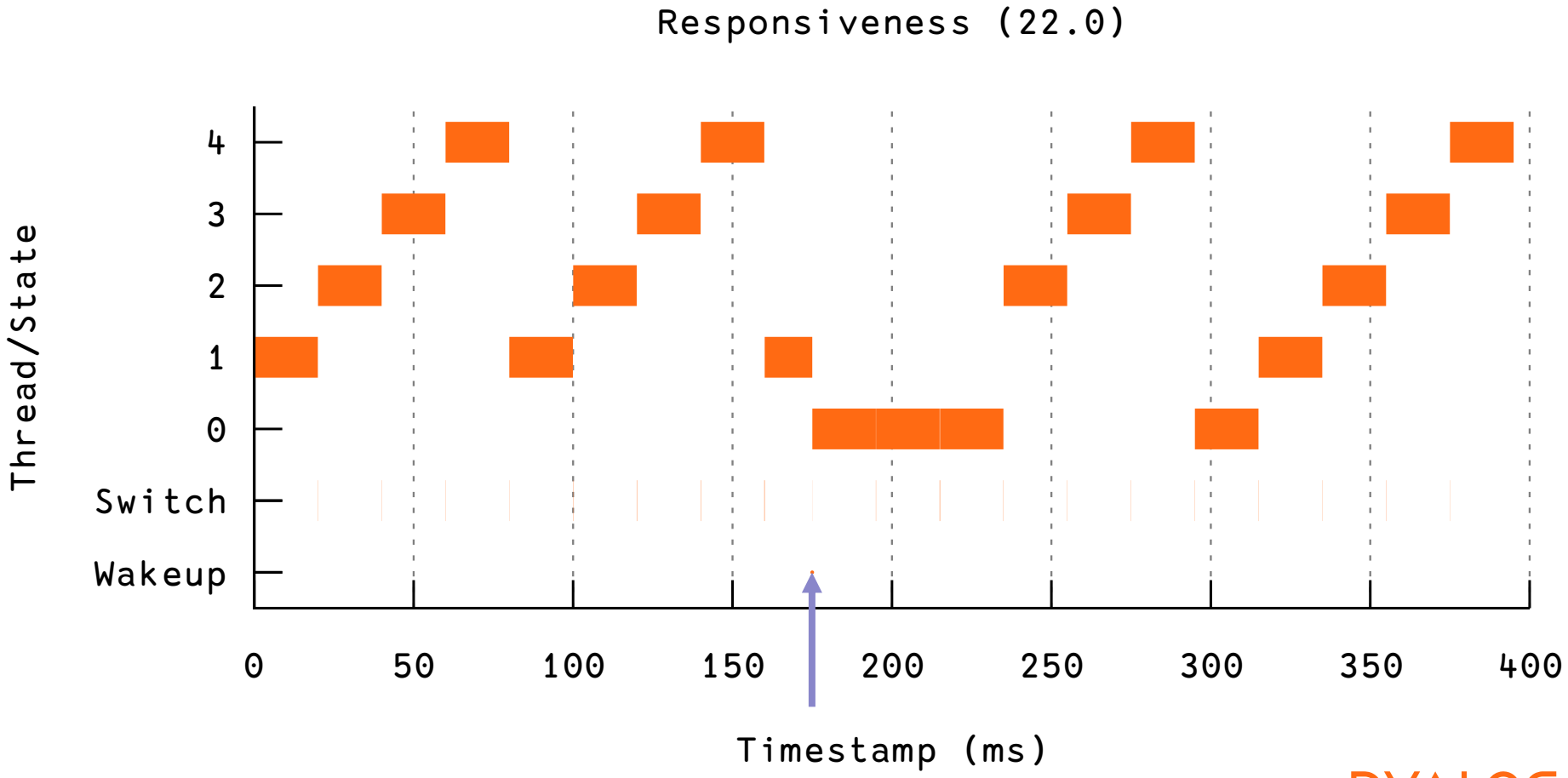
- The scheduler has too little information
  - When did thread A start to wait?
  - What is it waiting for?
  - What threads are waiting for thing X?
  - No queue of threads which are ready

So what's the problem?

- We can do better, and start recording this kind of information
  - Run the thread which has run the least so far
  - When a "wakeup event" happens, add threads to a queue
  - Never loop around checking for threads to see if they are ready


Scheduling 3 threads (22.0)
Thread/State
2
1
0
Switch
0
20
40
60
80
100
120
140
Timestamp (ms)

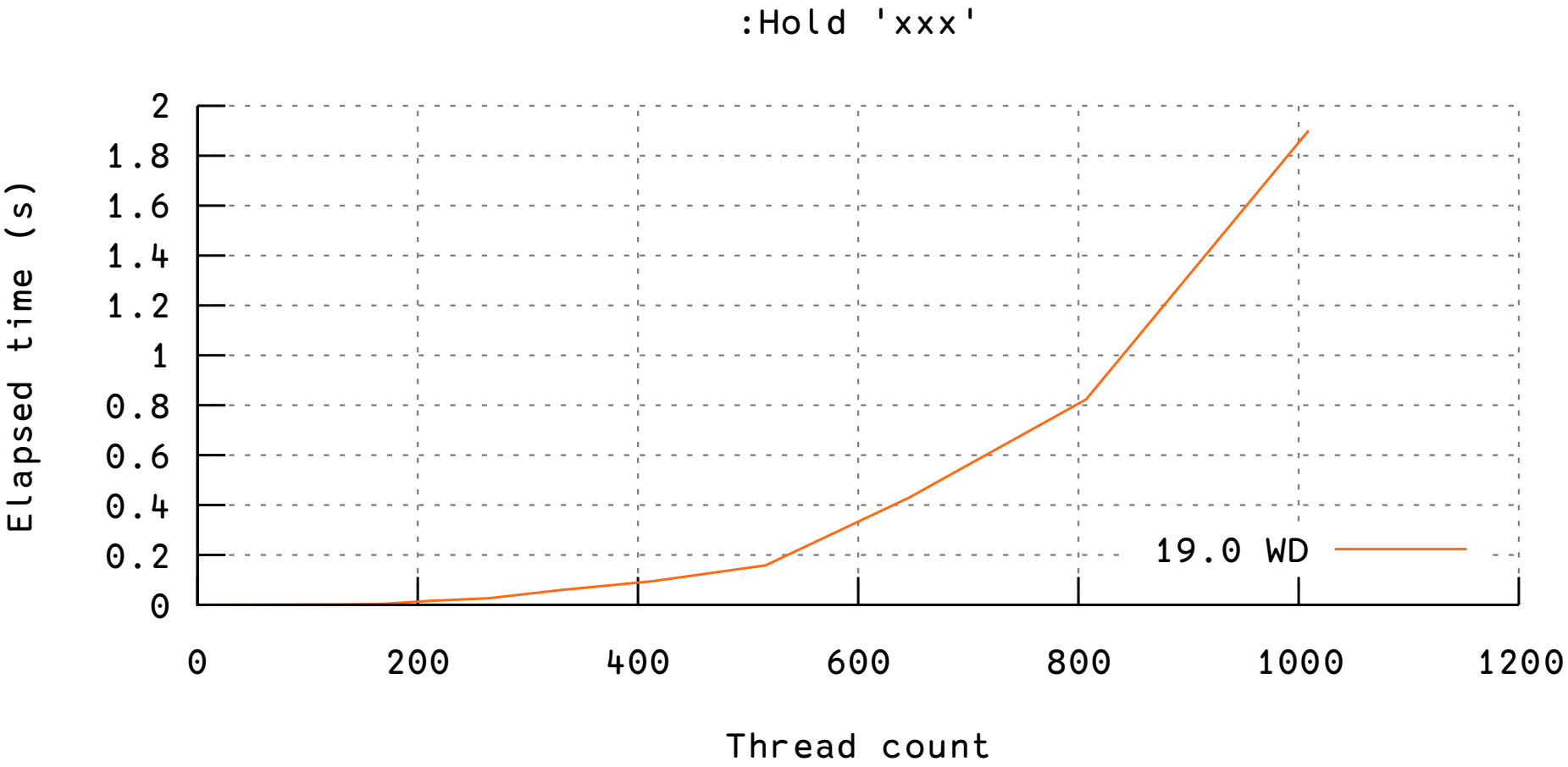

Scheduling 5 threads (22.0)
Thread/State
4
3
2
1
0
Switch
0
20
40
60
80
100
120
140
Timestamp (ms)

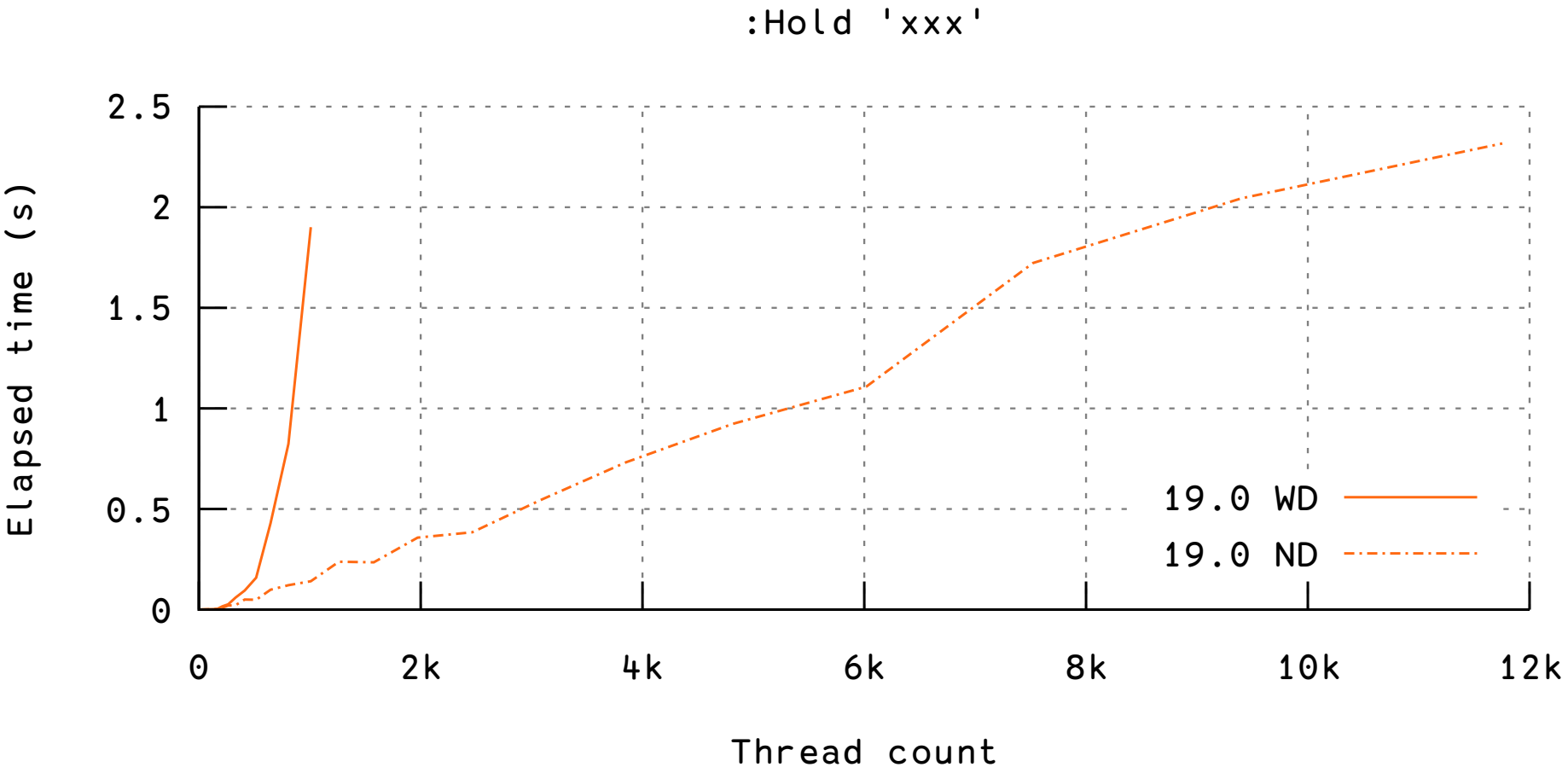

Responsiveness (22.0)
Thread/State
4
3
2
1
0
Switch
Wakeup
0
50
100
150
200
250
300
350
400
Timestamp (ms)

Scheduler summary

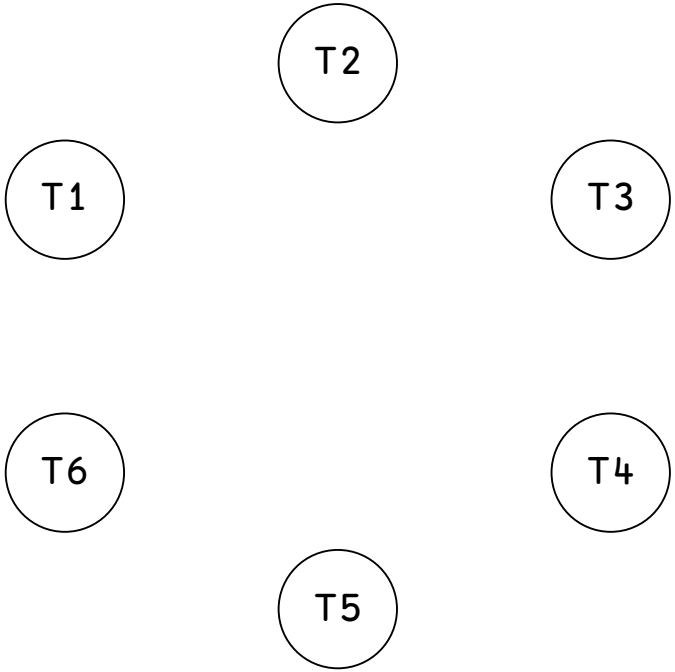
- The more the scheduler knows, the better decisions
- Fixed time slices ↔ constant overhead from the scheduler
  - Except when a thread becomes ready and needs attention
- Fair, as it prioritises threads which have run the least, not just "the next in the vector"
  - 1 ⎕TPUT thread, 10 ⎕TGET threads

Part 3: Deadlocks


:Hold 'xxx'
Elapsed time (s)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
200
400
600
800
1000
1200
Thread count
19.0 WD

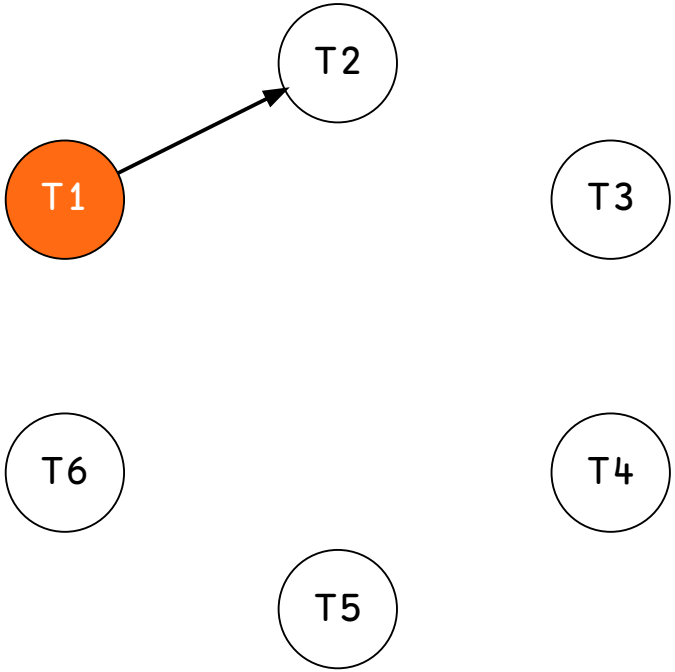

:Hold 'xxx'
Elapsed time (s)
2.5
2
1.5
1
0.5
0
0
2k
4k
6k
8k
10k
12k
Thread count
19.0 WD
19.0 ND

Using a graph

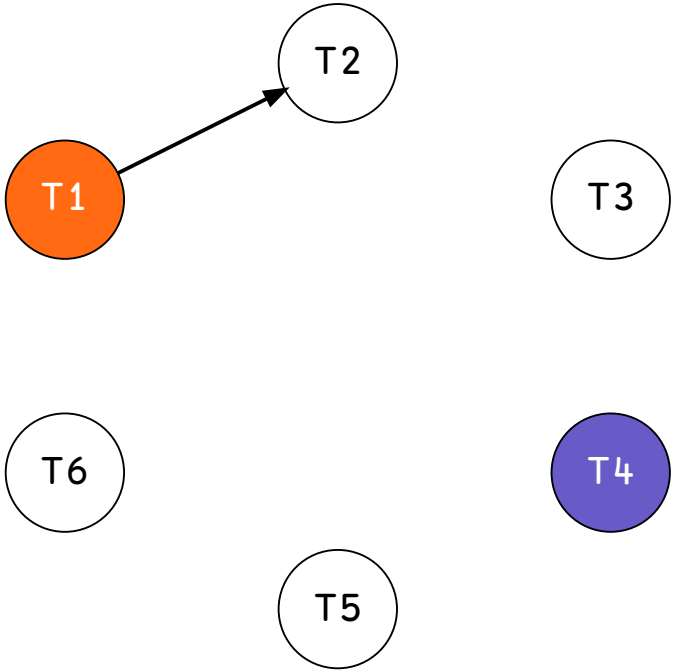


- The graph is maintained on all status changes
- Nodes are allocated at thread creation time

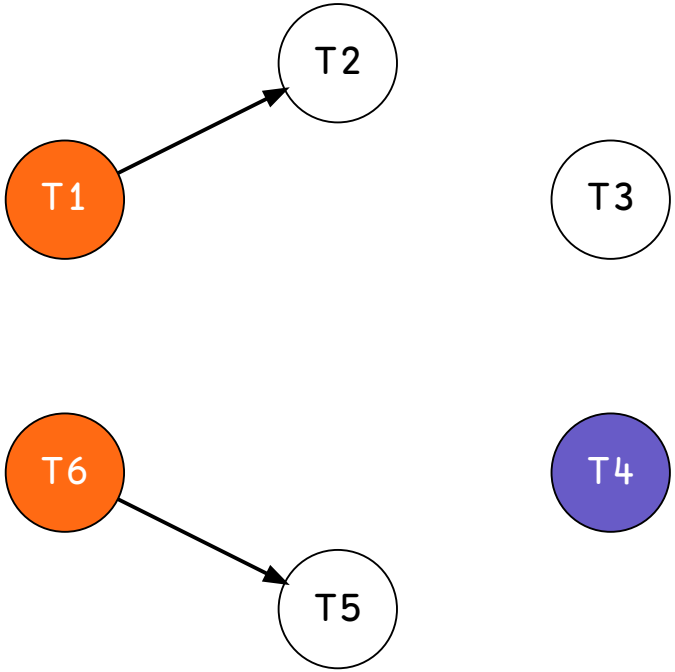
Using a graph



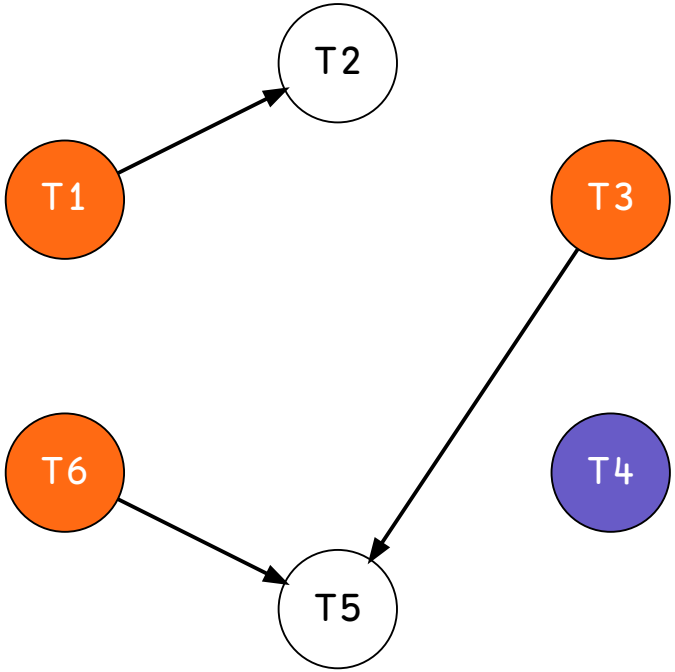
Using a graph



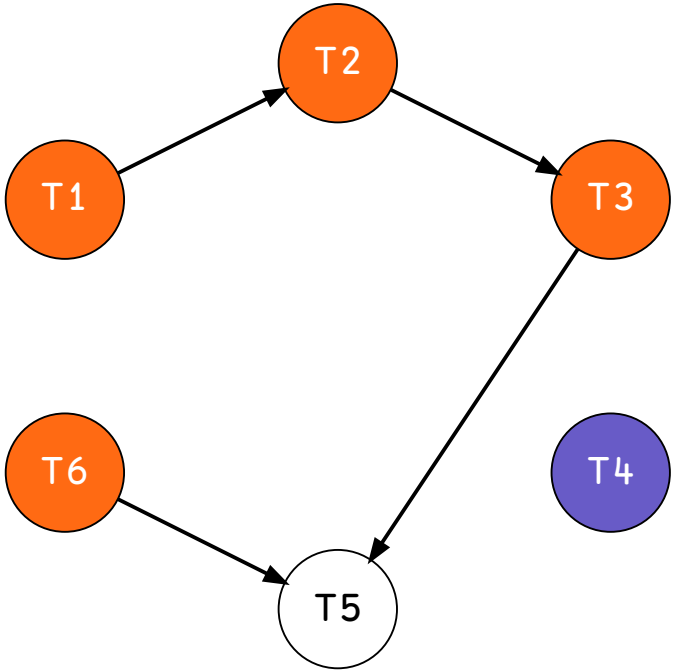
Using a graph



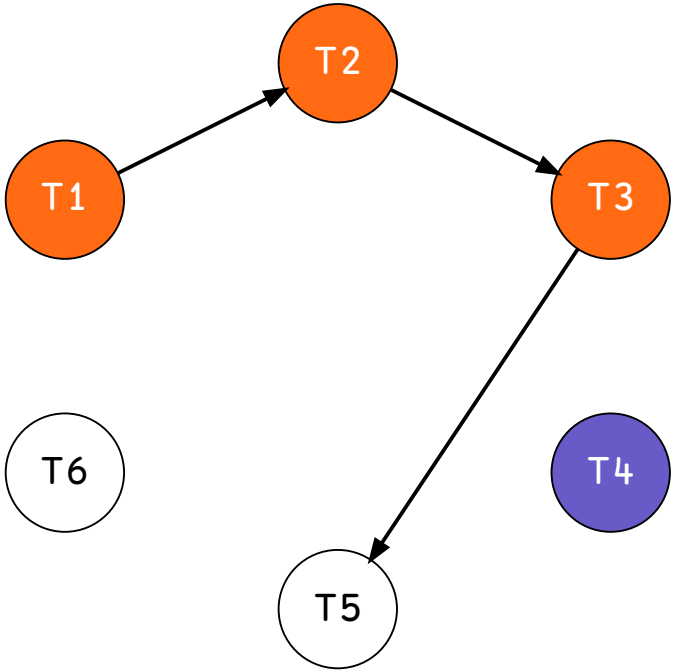
Using a graph



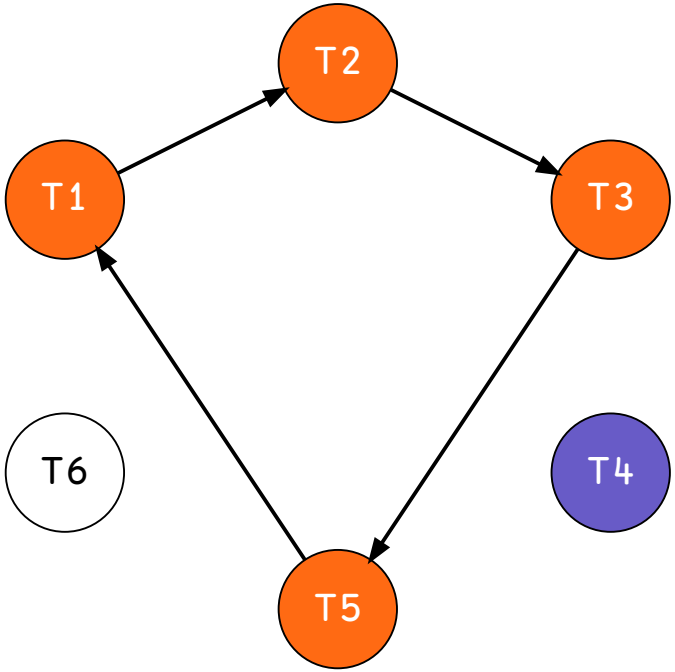
Using a graph



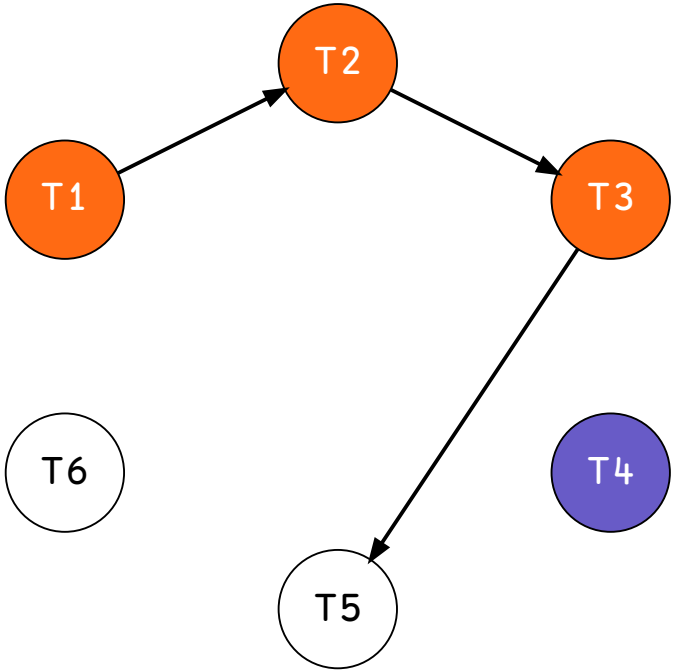
Using a graph



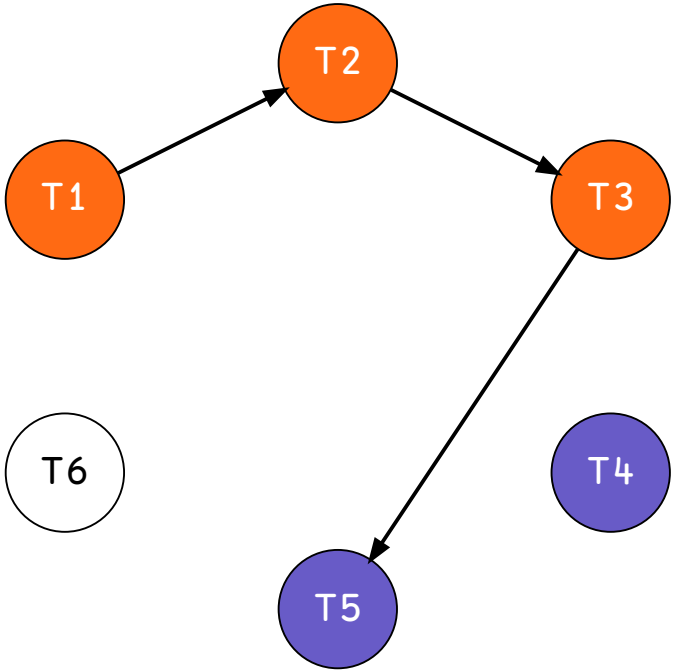
Using a graph



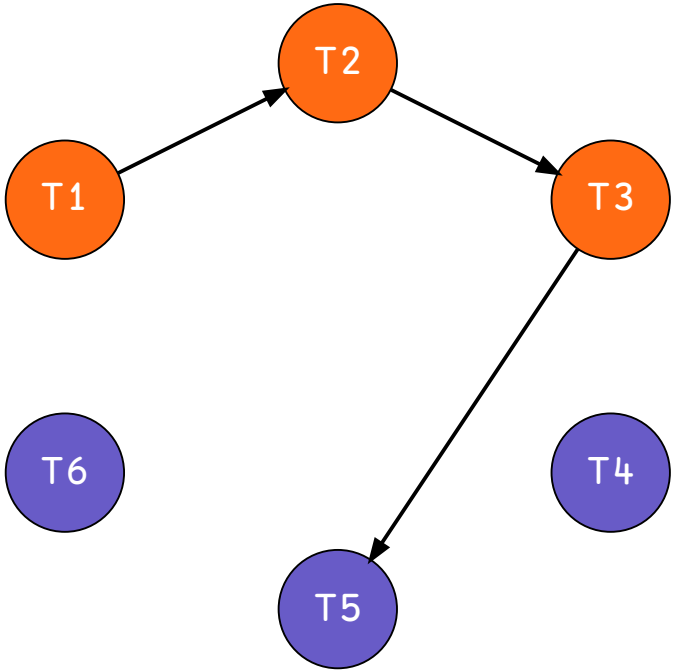
Using a graph



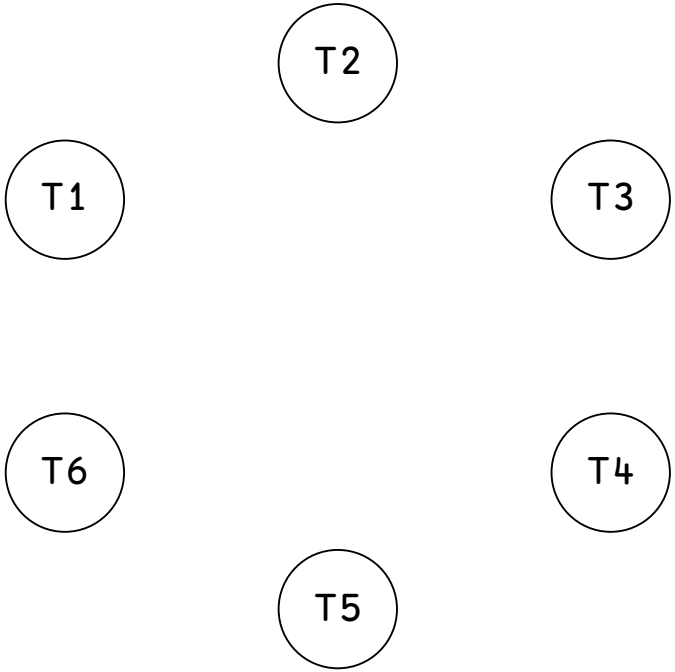
Using a graph



Using a graph

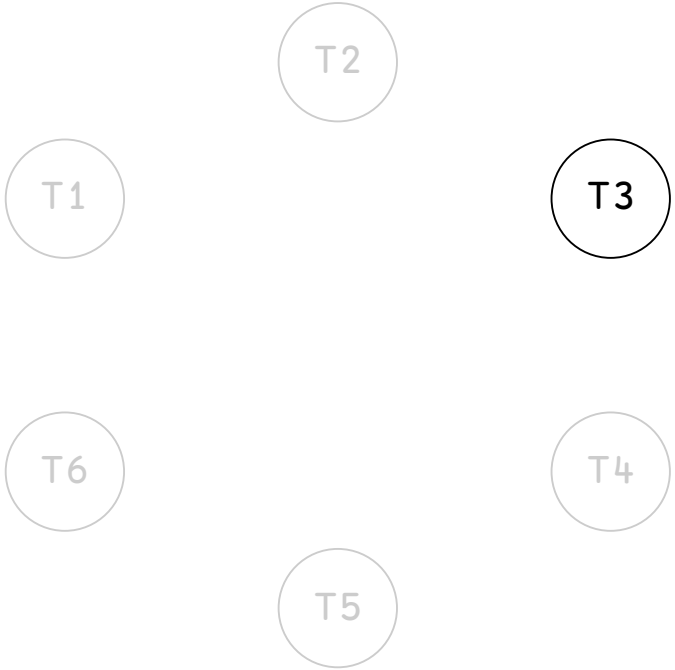


Using a graph



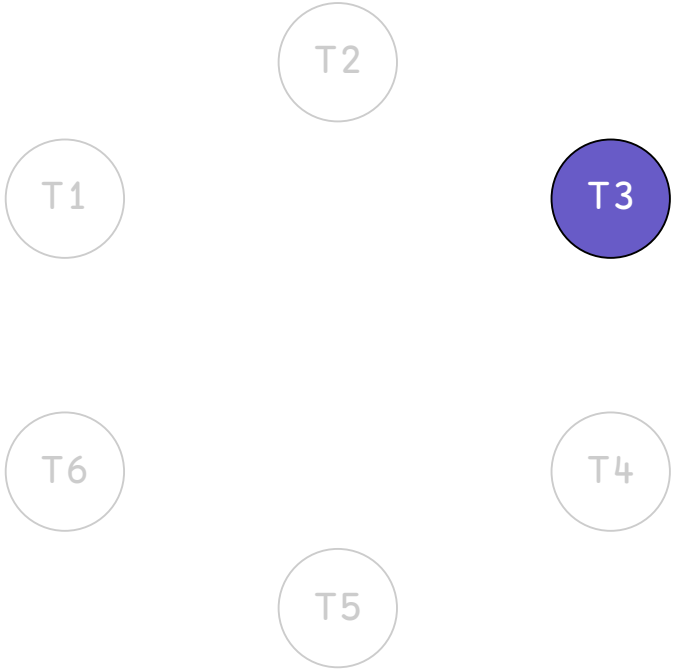
- Sometimes switching isn't allowed at all
  - High priority callbacks

Using a graph

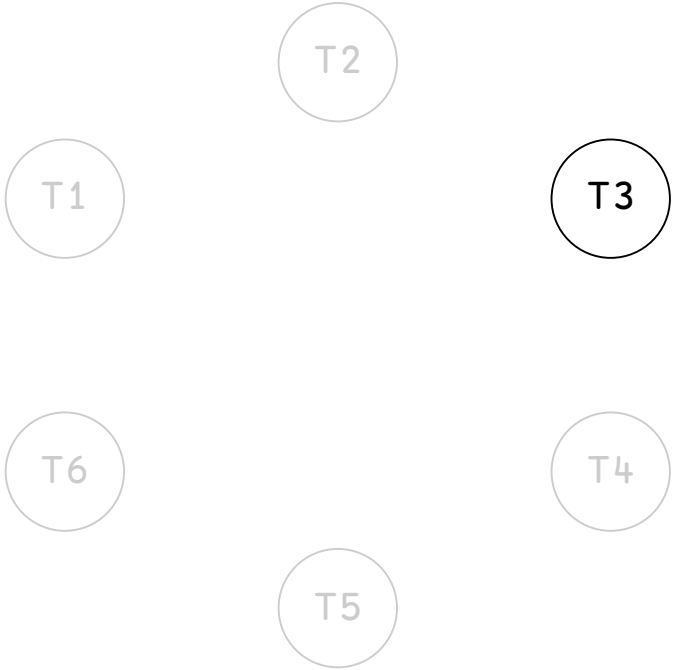


- Sometimes switching isn't allowed at all
  - High priority callbacks
- Same algorithm, but ignore all other threads

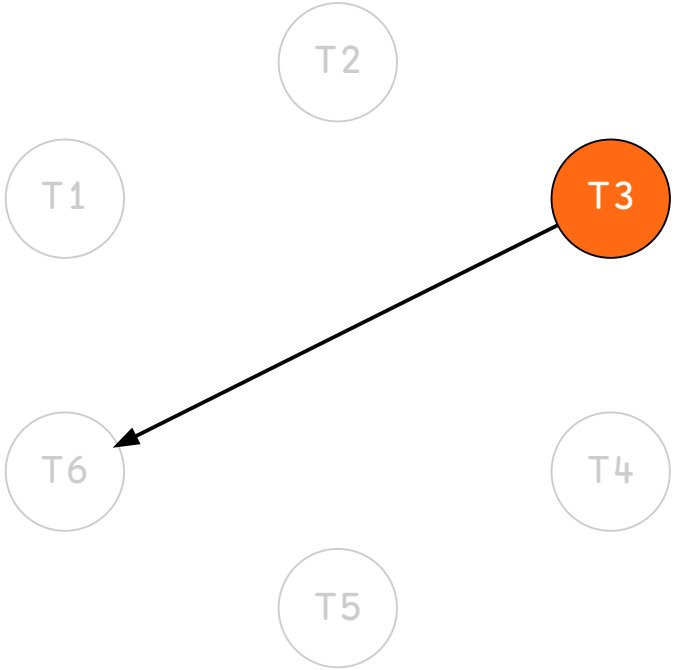
Using a graph

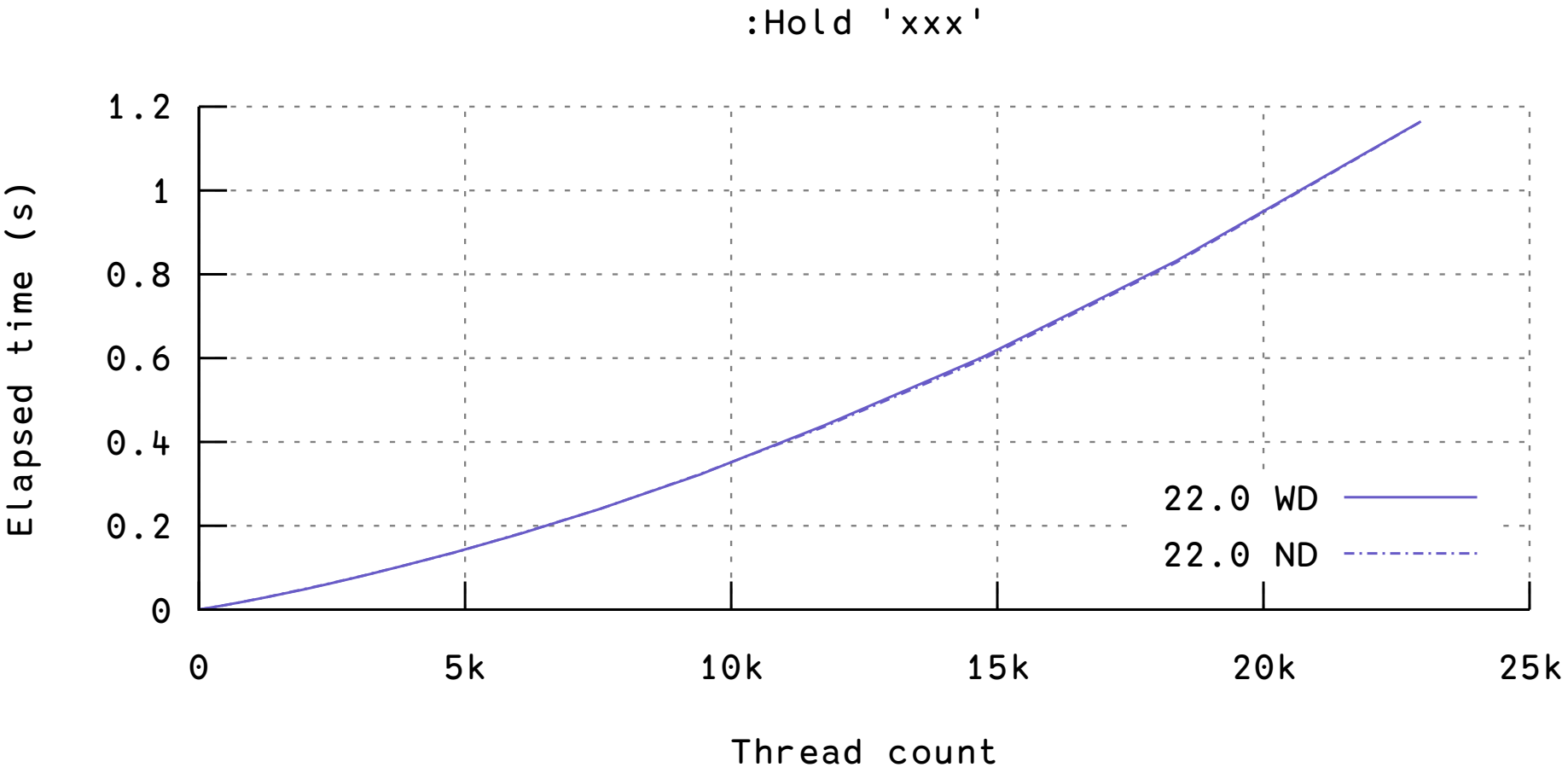


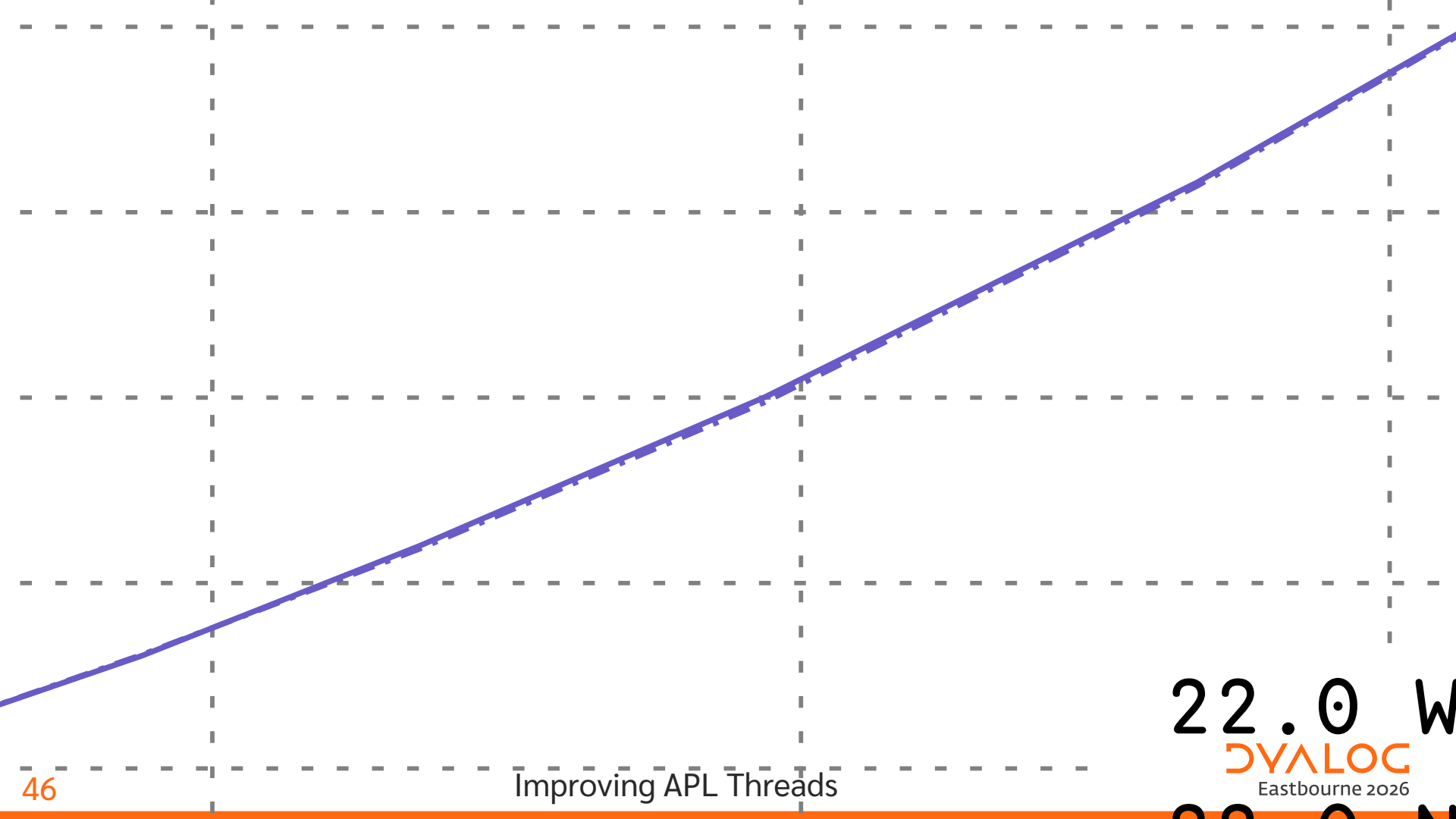
Using a graph

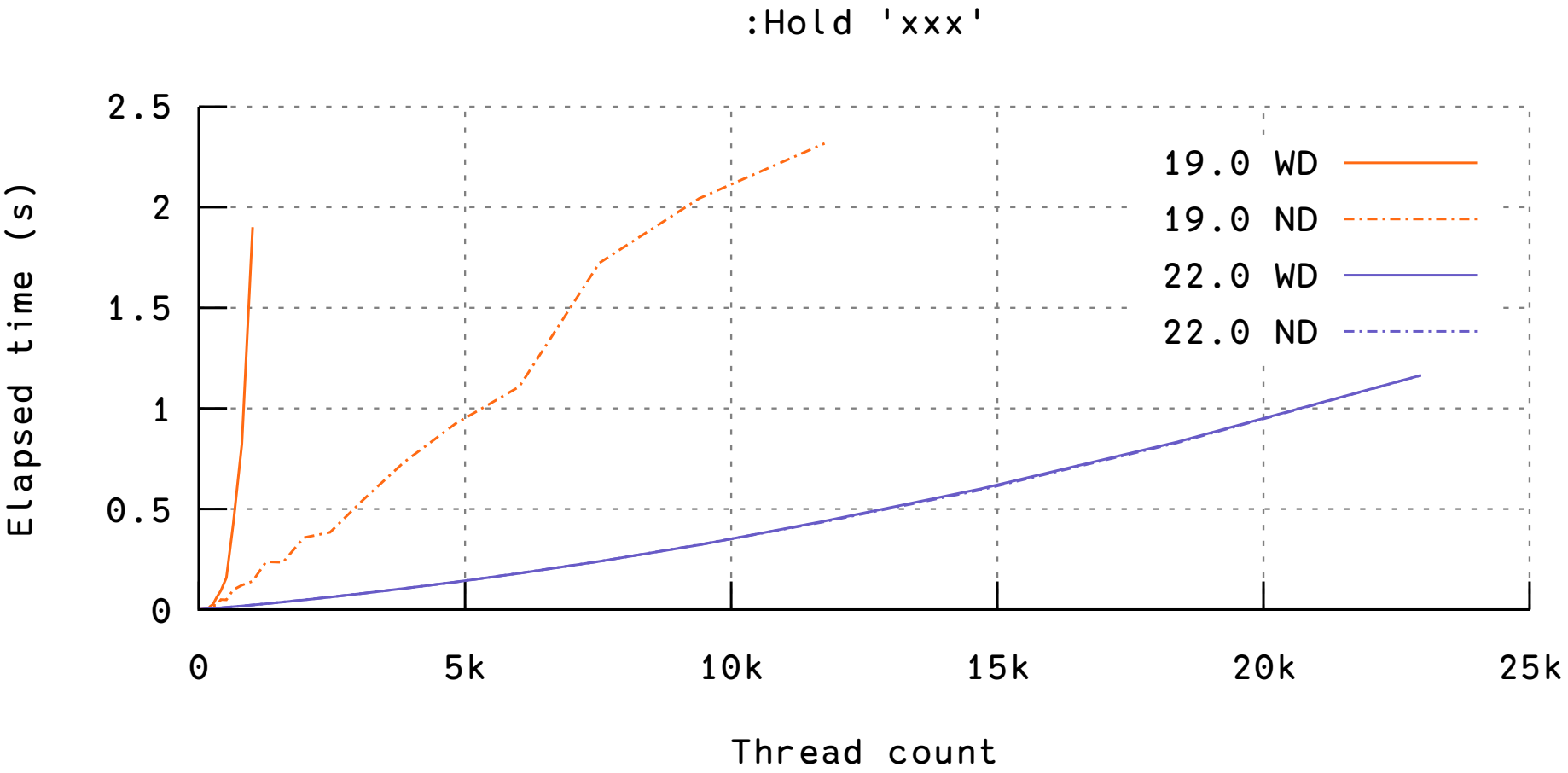


Using a graph




:Hold 'xxx'
Elapsed time (s)
1.2
1
0.8
0.6
0.4
0.2
0
0
5k
10k
15k
20k
25k
Thread count
22.0 WD
22.0 ND




:Hold 'xxx'
Elapsed time (s)
2.5
2
1.5
1
0.5
0
0
5k
10k
15k
20k
25k
Thread count
19.0 WD
19.0 ND
22.0 WD
22.0 ND

Deadlocks summary

- A general mechanism
- Detects more cases
- Is quicker/more predictable
- Only uses memory which is allocated at thread creation time

Summary

- Only a brief overview of what's in development
  - Things might still change (performance, etc)
- Mostly unnoticeable, except for improved performance and:
  - More deadlocks detected
  - Different scheduling order
  - ↑ Probably good in the long run

Summary

- Who knows what we'll do in the future
  - A parallel operator?
  - More switch points?
    - ⎕NGET, ⎕NREAD, ⎕NAPPEND for example?
- Good to get the foundations in a good shape first
- Hope you'll consider giving threads a go if you haven't