

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS

KEYNOTE FOR BPOE-9 @ASPLOS2018

THE NINTH WORKSHOP ON BIG DATA BENCHMARKS,
PERFORMANCE, OPTIMIZATION AND EMERGING HARDWARE

By Jean-Pierre Lozi
Oracle Labs

CLUSTER COMPUTING

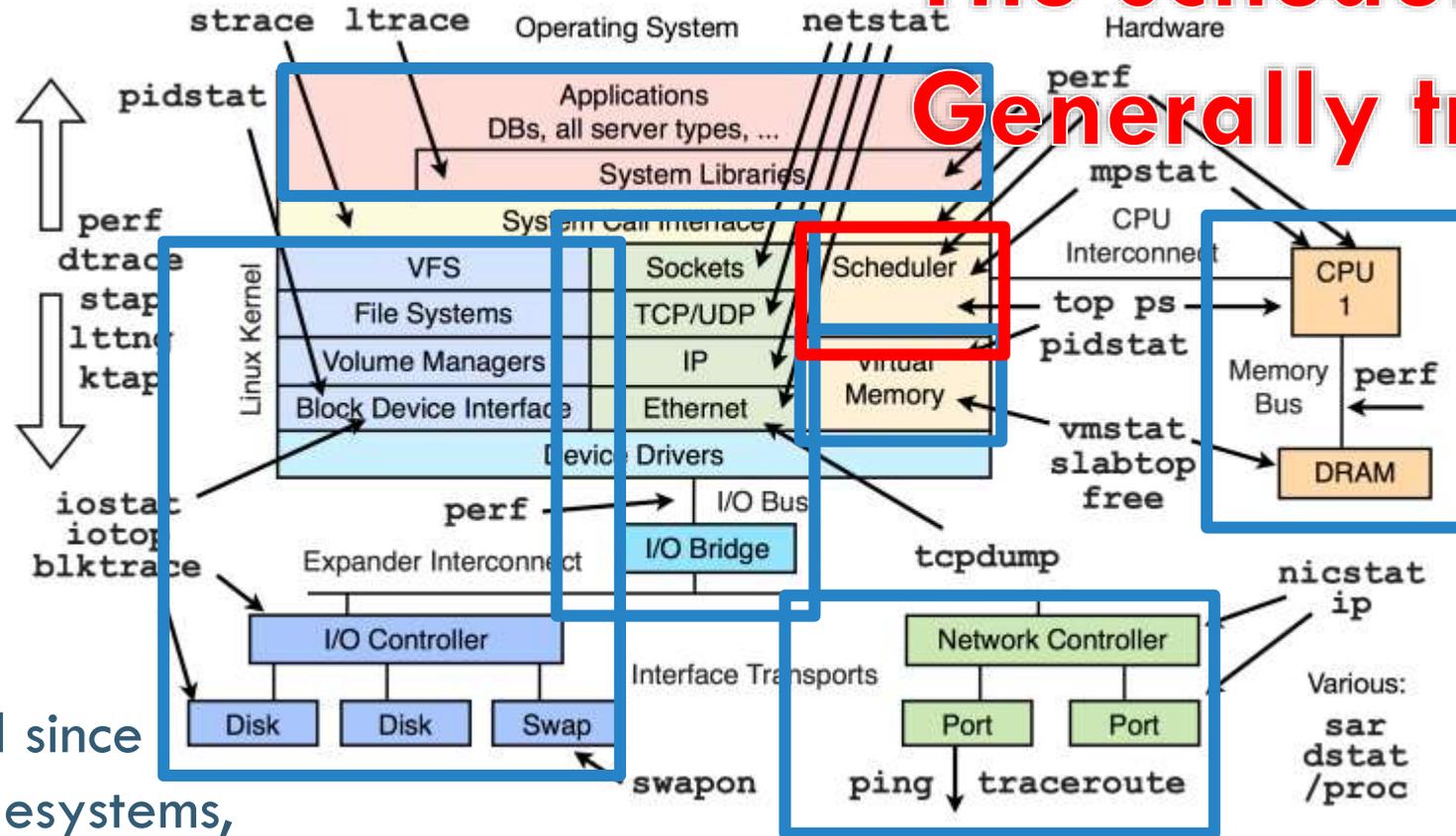
- **Multicore servers with dozens of cores**
 - Common for e.g., a hadoop cluster, a distributed graph analytics engine, multiple apps...
 - High cost of infrastructure, high energy consumption
- **Linux-based software stack**
 - Low (license) cost, yet high reliability
- **Challenge: don't waste cycles!**
 - Reduces infrastructure and energy costs
 - Improves bandwidth and latency



WHERE TO HUNT FOR CYCLES?

The scheduler???
Generally trusted!

Applications, libraries:
often main focus



NUMA, bus
usage:
Placement,
replication,
interleaving,
many recent
papers

Storage: optimized since
decades! E.g., many filesystems,
RDBMSes bypassing the OS

Network stack, NICs,

reducing network usage (e.g. HDFS): common optimizations 3

IS THE SCHEDULER WORKING IN YOUR CLUSTER?

- It must be! 15 years ago, Linus Torvalds was already saying:

*“And you have to realize that there are not very many things that have aged as well as the scheduler. **Which is just another proof that scheduling is easy.**”*

- Since then, people have been running applications on their multicore machines all the time, and they run, CPU usage is high, everything seems fine.
- But would you notice if some cores remained idle intermittently, when they shouldn't?
 - **Do you keep monitoring tools (htop) running all the time?**
 - **Even if you do, would you be able to identify faulty behavior from normal noise?**
 - **Would you ever suspect the scheduler?**

THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.



THIS TALK

- **This is how we found our first performance bug.** Which made us investigate more...
- **In the end: four Linux scheduler performance bugs that we found and analyzed**
- **Always the same symptom: idle cores while others are overloaded**
 - The bug-hunting was tough, and led us to develop our own tools
- **Performance overhead of some of the bugs :**
 - 12-23% performance improvement on a popular database with TPC-H
 - **1.37× performance improvement on HPC workloads**
- **Not always possible to provide a simple, working fix...**
 - Intrinsic problems with the design of the scheduler?

THIS TALK

Jean-Pierre Lozi
Université Nice Sophia-Antipolis
jplozi@unice.fr

Baptiste Lepers
EPFL
baptiste.lepers@epfl.ch

Justin Funston
University of British Columbia
jfunston@ece.ubc.ca

Fabien Gaud
Coho Data
me@fabiangaud.net

Vivien Quéma
Grenoble INP / ENSIMAG
vivien.quema@imag.fr

Alexandra Fedorova
University of British Columbia
sasha@ece.ubc.ca

Main takeaway of our analysis: *more research must be directed towards implementing an efficient scheduler for multicore architectures, because contrary to what a lot of us think, this is **not a solved problem!***

Need convincing? Let's go through it together...

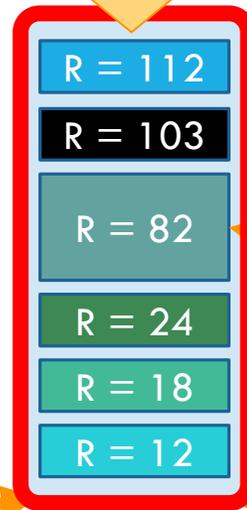
...starting with a bit of background...

THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by *runtime*

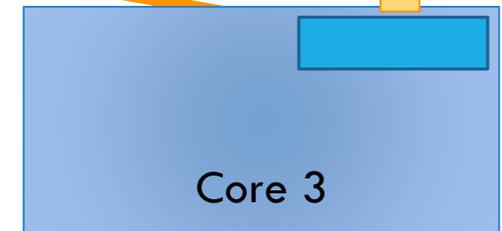
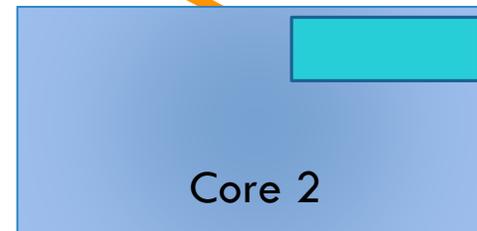
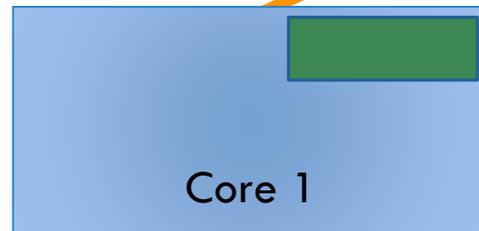
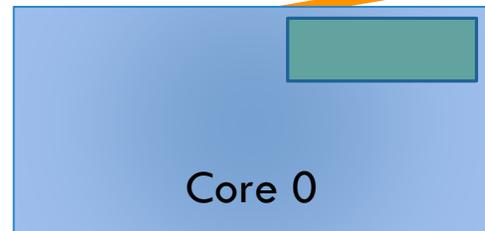
Cores get their next task from the global runqueue

Of course, cannot work with a single runqueue because of contention



When a thread is done running for its *timeslice* : enqueued again

Some tasks have a lower *niceness* and thus have a longer *timeslice* (allowed to run longer)



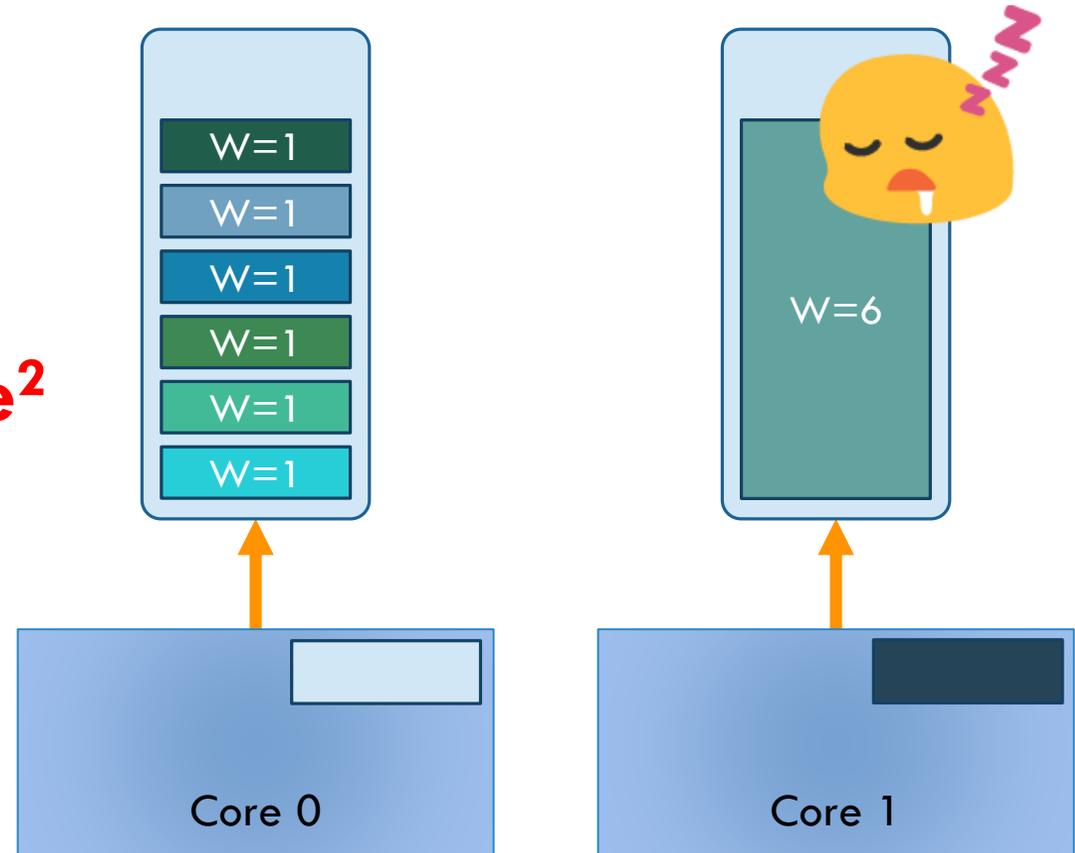
CFS: IN PRACTICE

- One runqueue per core to avoid contention
- CFS **periodically** balances “loads”:

$$\text{load}(\text{task}) = \text{weight}^1 \times \% \text{ cpu use}^2$$

¹The lower the niceness, the higher the weight

²We don't want a high-priority thread that sleeps a lot to take a whole CPU for itself and then mostly sleep!

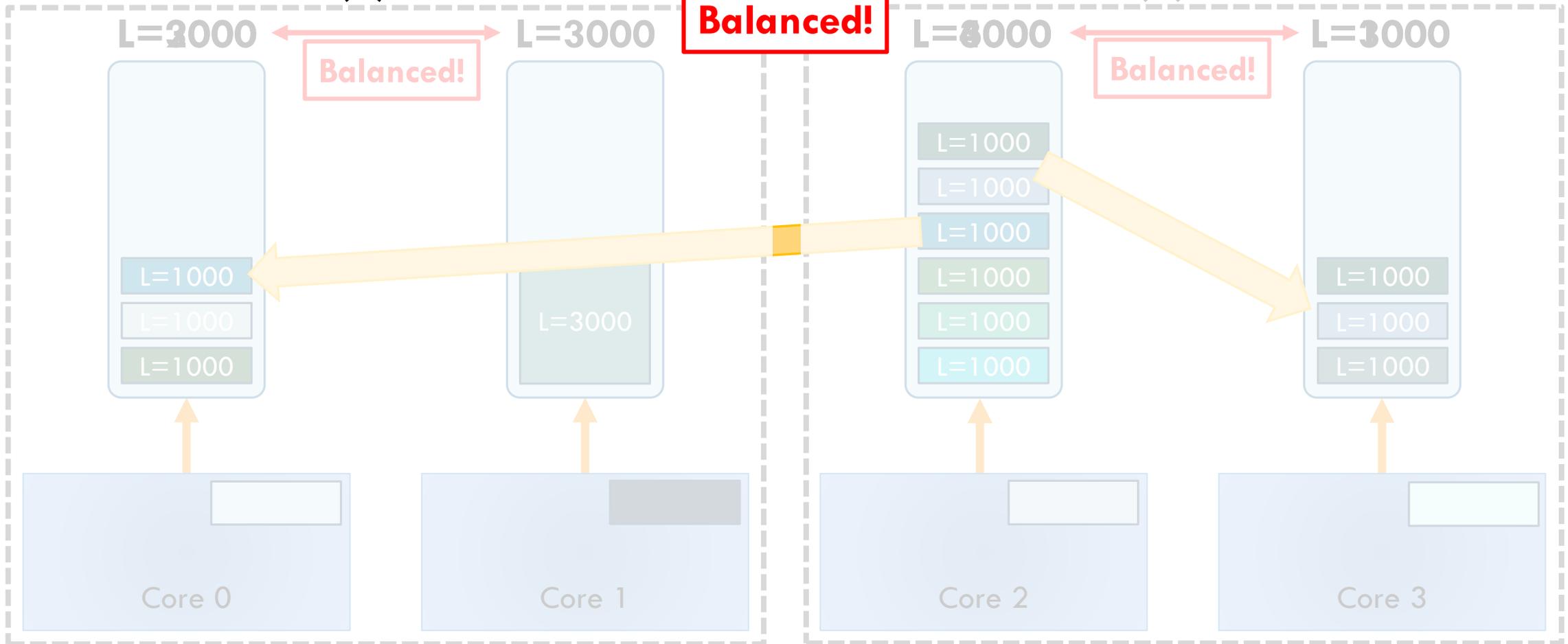


- Since there can be many cores: **hierarchical approach!**

CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

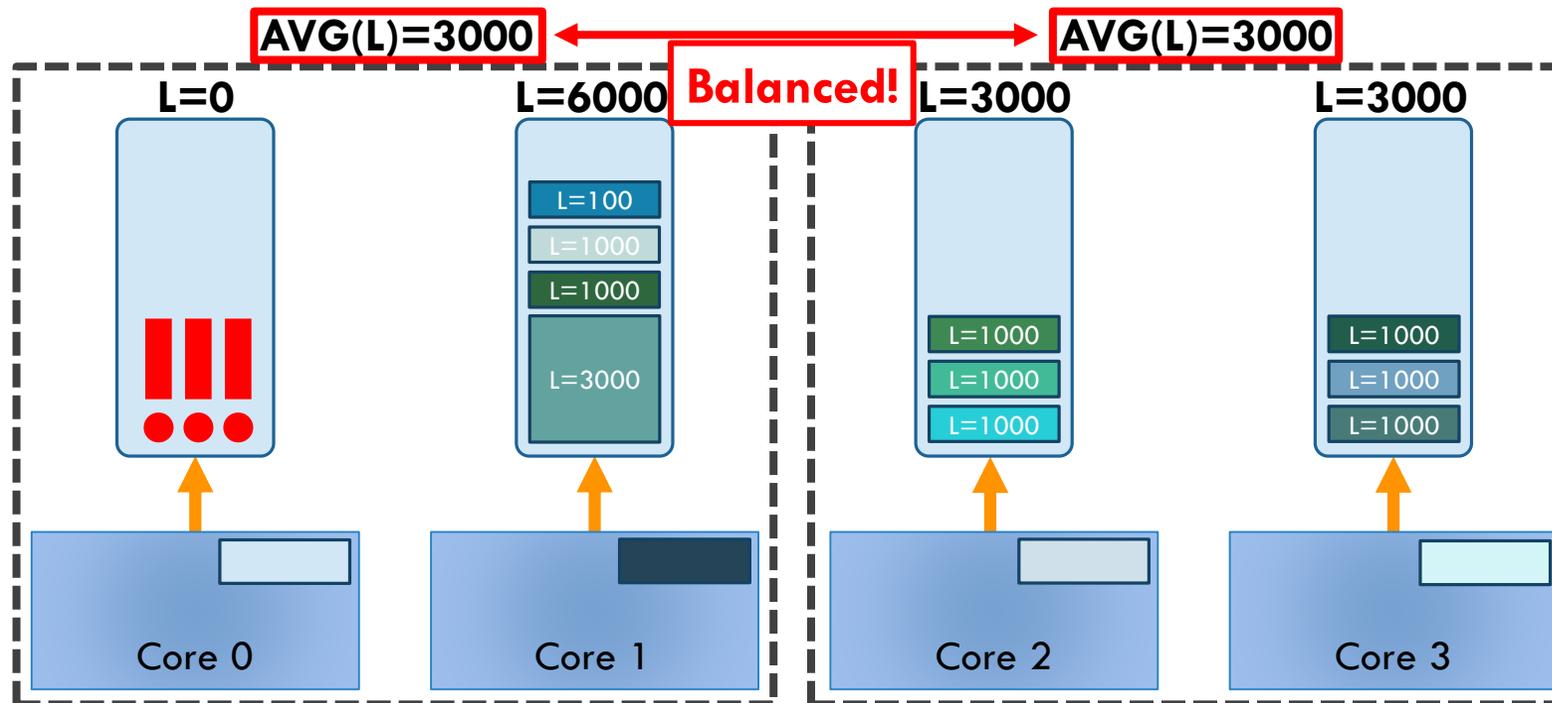
AVG(L)=3000

AVG(L)=3000



CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

- Note that only the *average load of groups* is considered
- If for some reason the lower-level load-balancing fails, nothing happens at a higher level:



CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- **One of them aims to increase fairness between “sessions”.**
- **Objective:** making sure that launching lots of threads from one terminal doesn't prevent other processes on the machine (potentially from other users) from running.
 - Otherwise, easy to use more resources than other users by spawning many threads...

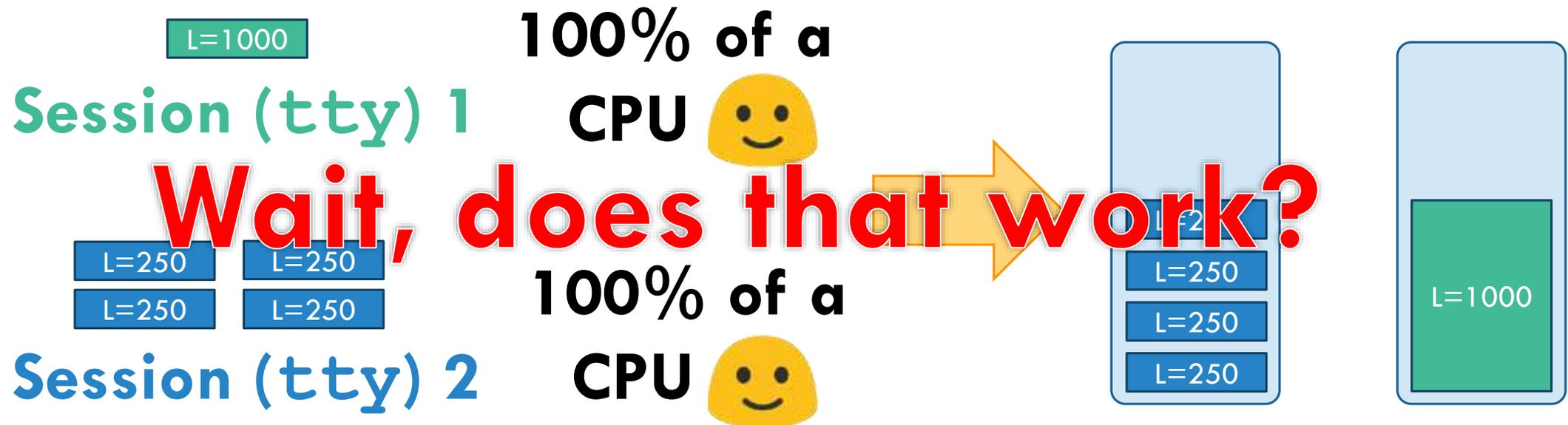
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- **One of them aims to increase fairness between “sessions”.**



CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- **Solution:** divide the load of a task by the number of threads in its `tty`...

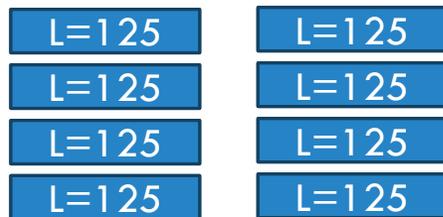


BUG 1/4: GROUP IMBALANCE



$$\begin{aligned}\text{Load}(\text{thread}) &= \%cpu \times \text{weight} / \#\text{threads} \\ &= 100 \times 10 / 1 \\ &= 1000\end{aligned}$$

Session (tty) 1



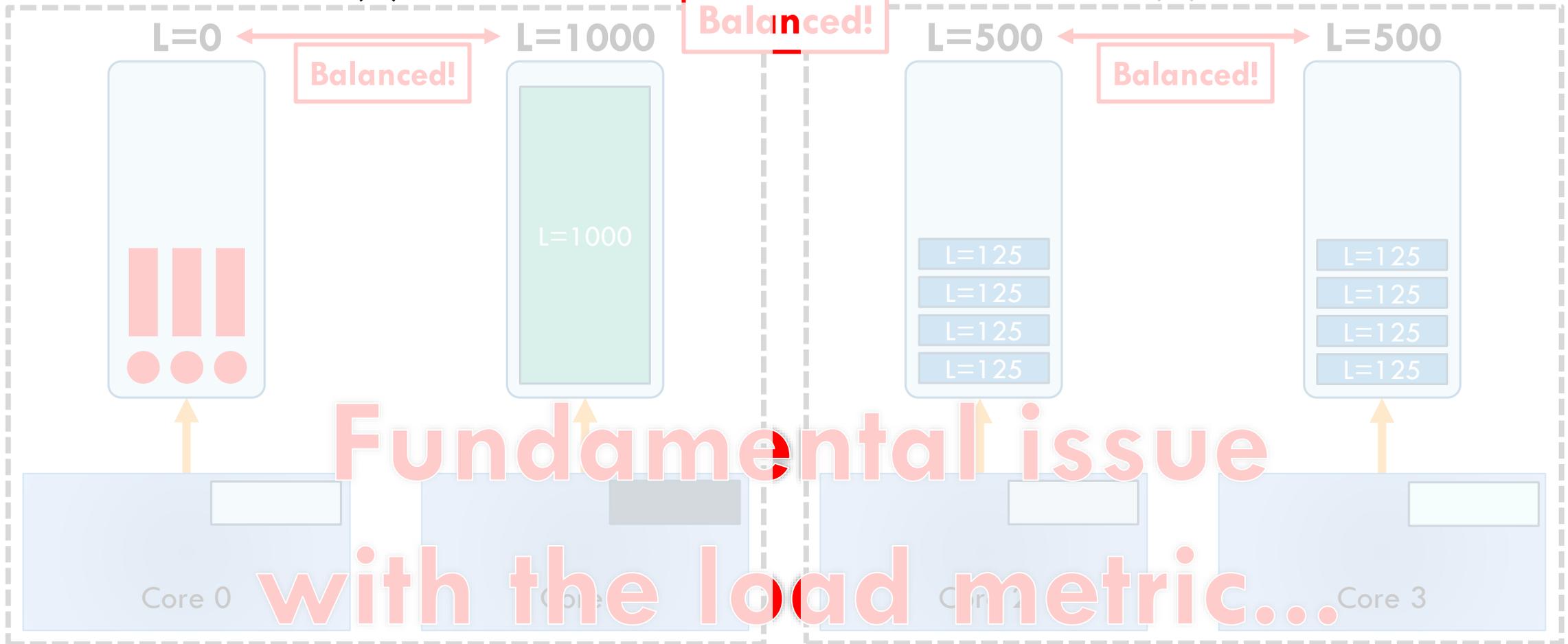
$$\begin{aligned}\text{Load}(\text{thread}) &= \%cpu \times \text{weight} / \#\text{threads} \\ &= 100 \times 10 / 8 \\ &= 125\end{aligned}$$

Session (tty) 2

BUG 1/4: GROUP IMBALANCE

AVG(L)=500

AVG(L)=500

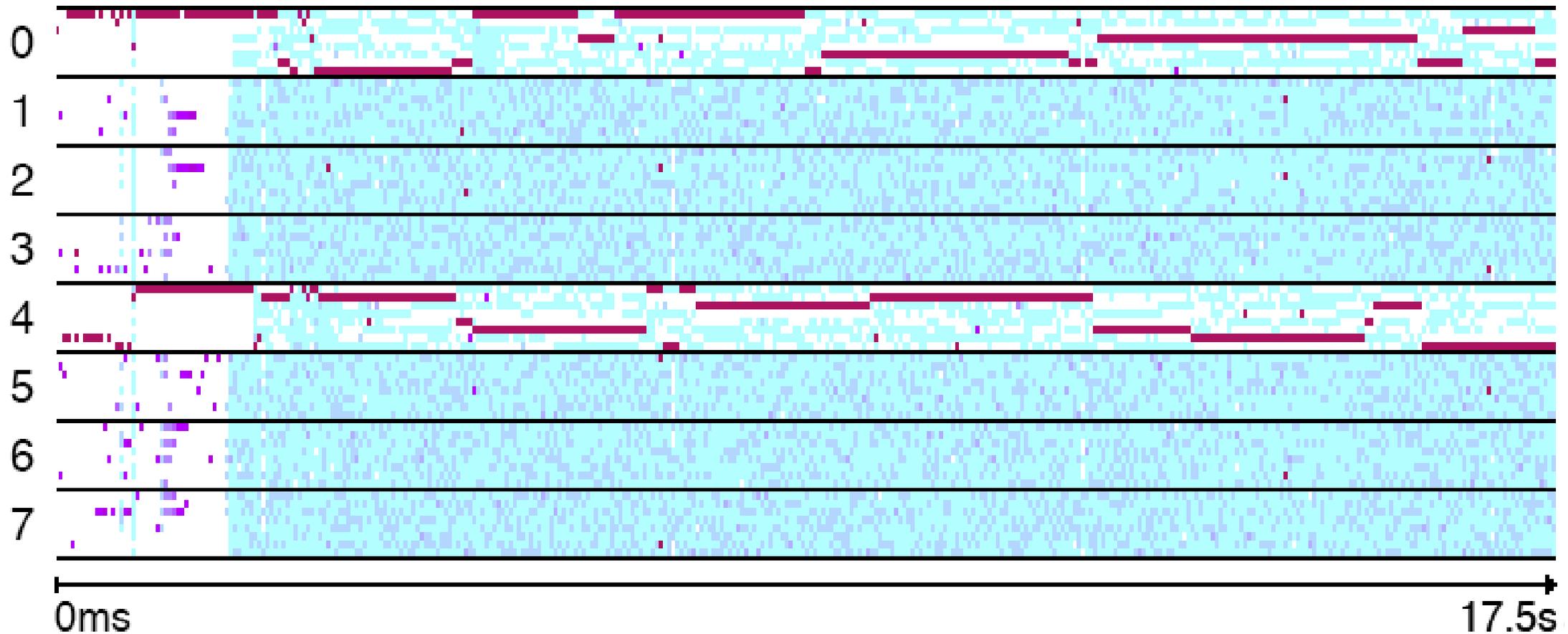


Load:

0

1

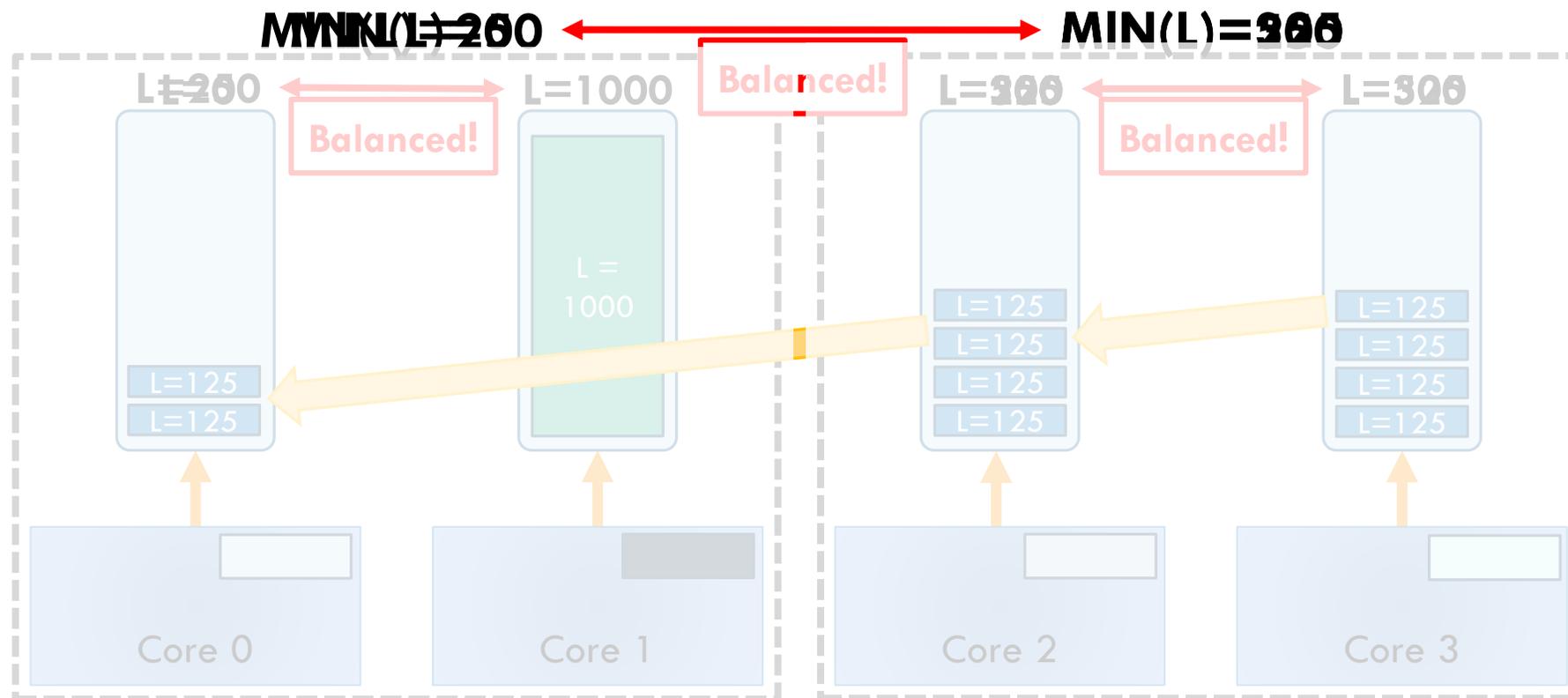
1024



- **The bug happens at two levels :**
 - Other core on pair of core idle
 - Other cores on NUMA node less busy...

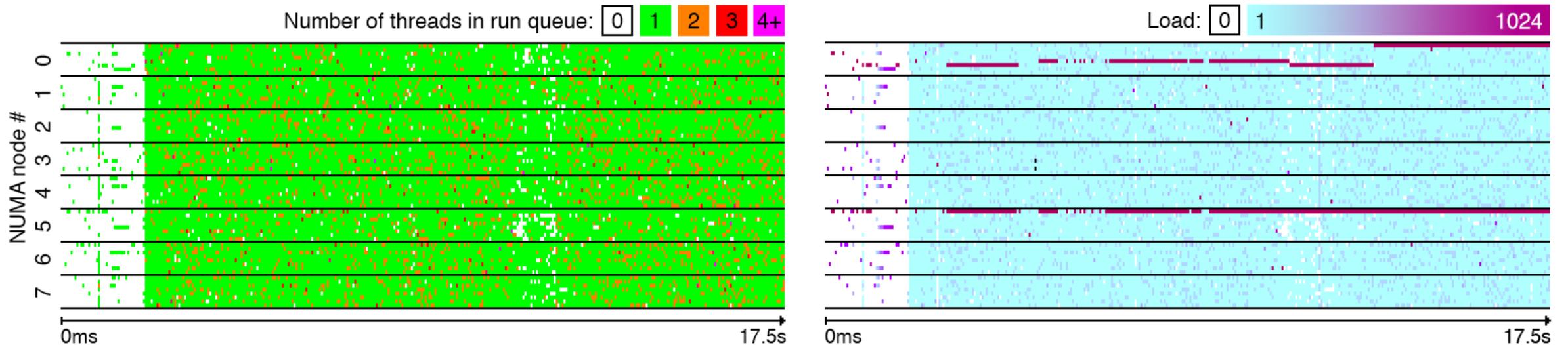
BUG 1/4: GROUP IMBALANCE

- A simple solution: balance the *minimum load* of groups instead of the *average*



BUG 1/4: GROUP IMBALANCE

- A simple solution: balance the *minimum* load of groups instead of the *average*

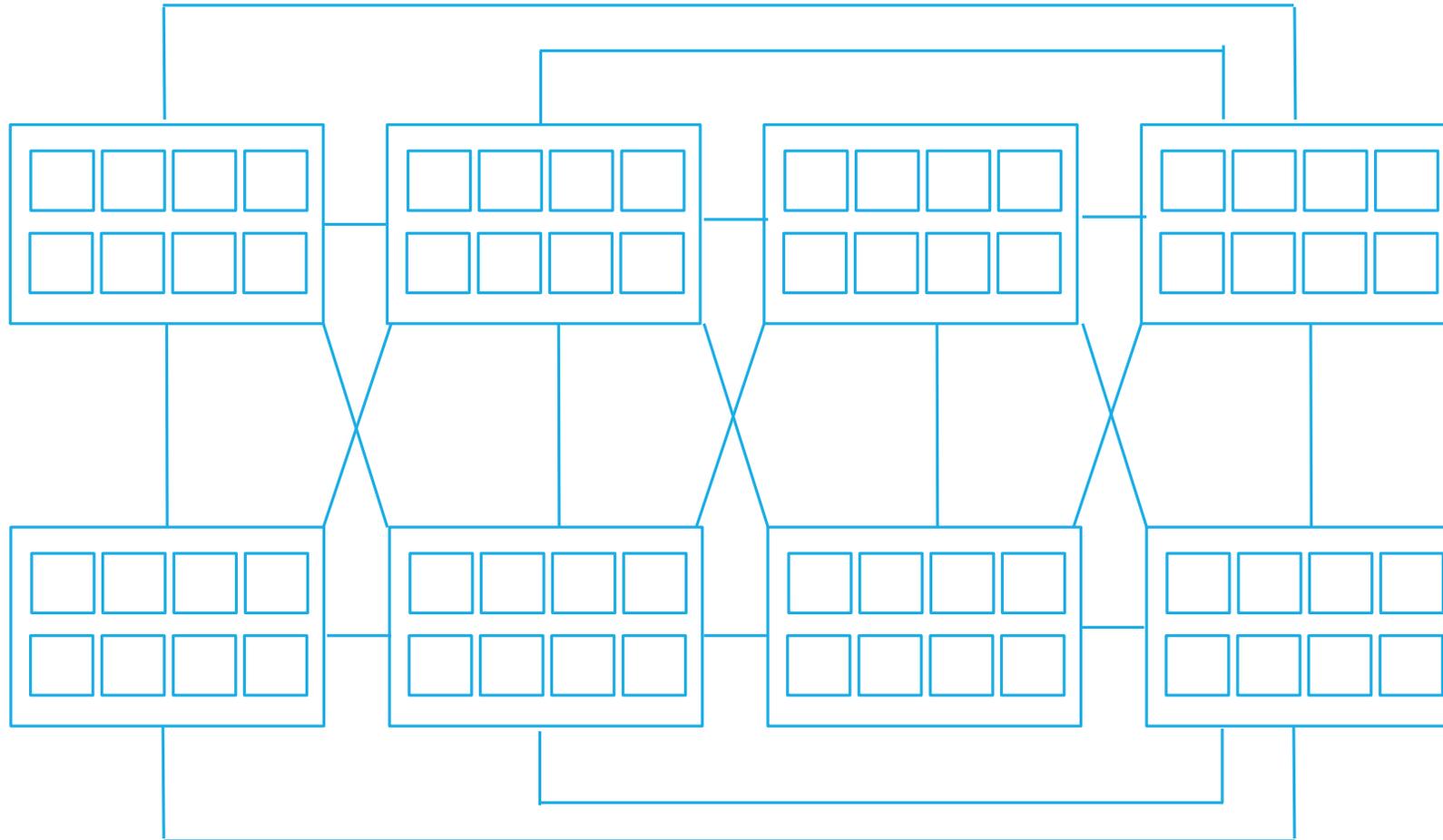


- **After the fix, make runs 13% faster, and R is not impacted**
- **A simple solution, but is it ideal? Minimum load more volatile than average...**
 - *May cause lots of unnecessary rebalancing. Revamping load calculations needed?*

BUG 2/4: SCHEDULING GROUP CONSTRUCTION

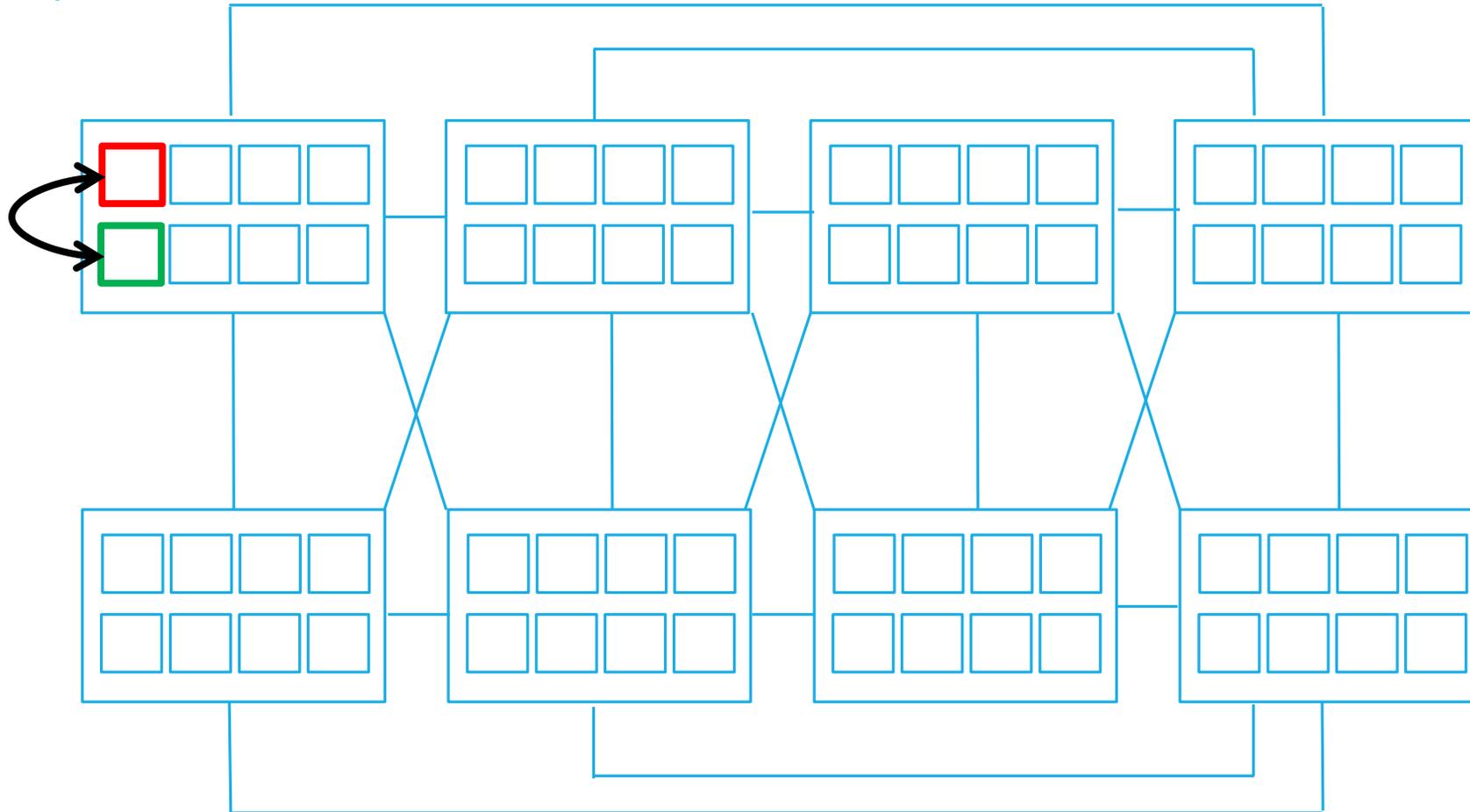
- **Hierarchical load balancing** is based on groups of cores named *scheduling domains*
 - Based on affinity, i.e., pairs of cores, dies, CPUs, NUMA nodes, etc.
- Each scheduling domain contains groups that are the lower-level scheduling domains
- **For instance, on our 64-core AMD Bulldozer machine:**
 - At level 1, each **pair of core** (scheduling domains) contain **cores** (scheduling groups)
 - At level 2, each **CPU** (s.d.) contain **pairs of cores** (s.g.)
 - At level 3, each **group of directly connected CPUs** (s.d.) contain **CPUs** (s.g.)
 - At level 4, the **whole machine** (s.d.) contains **group of directly connected CPUs** (s.g.)

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



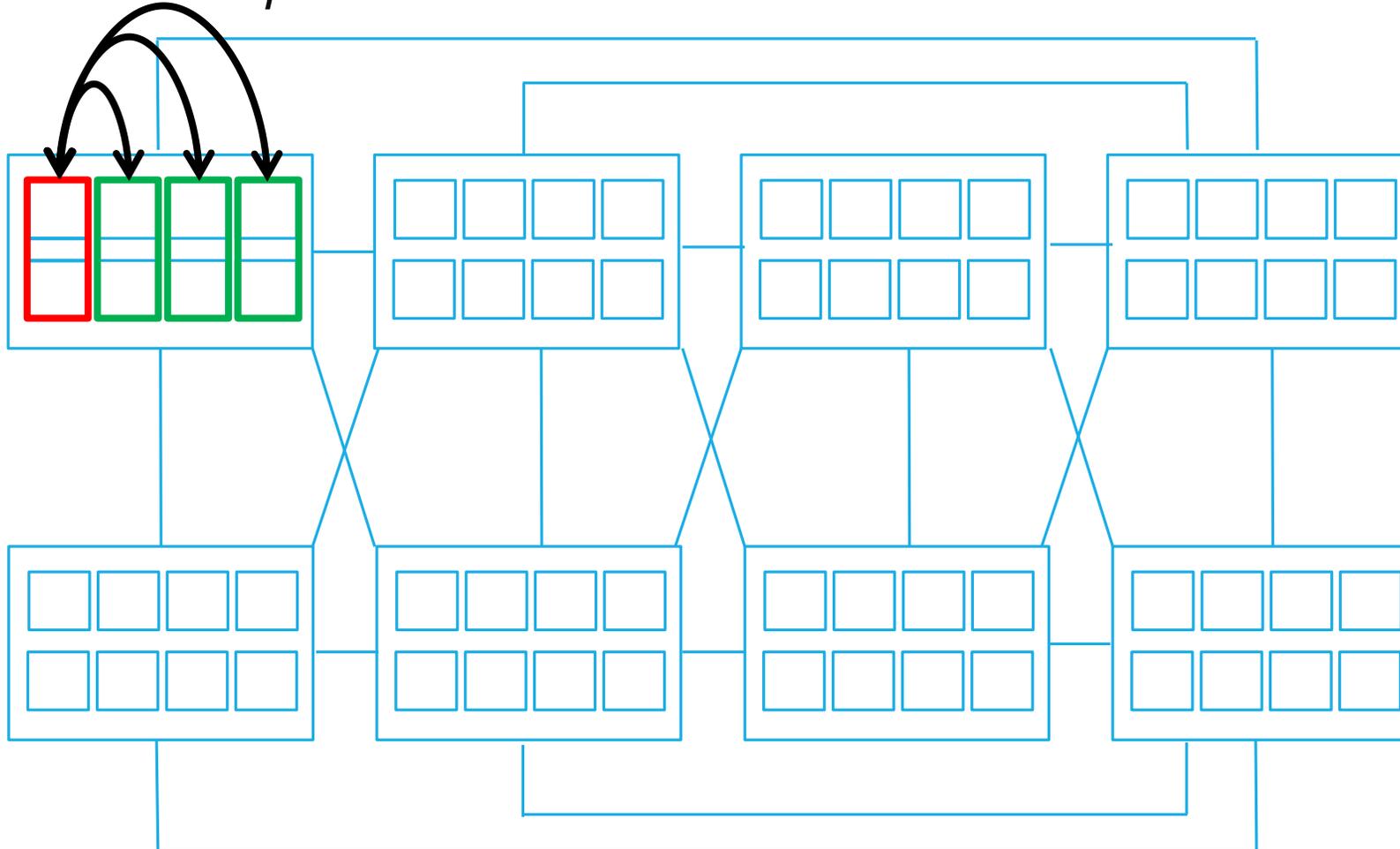
Bulldozer 64-core:
Eight CPUs, with
8 cores each,
non-complete
interconnect graph!

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



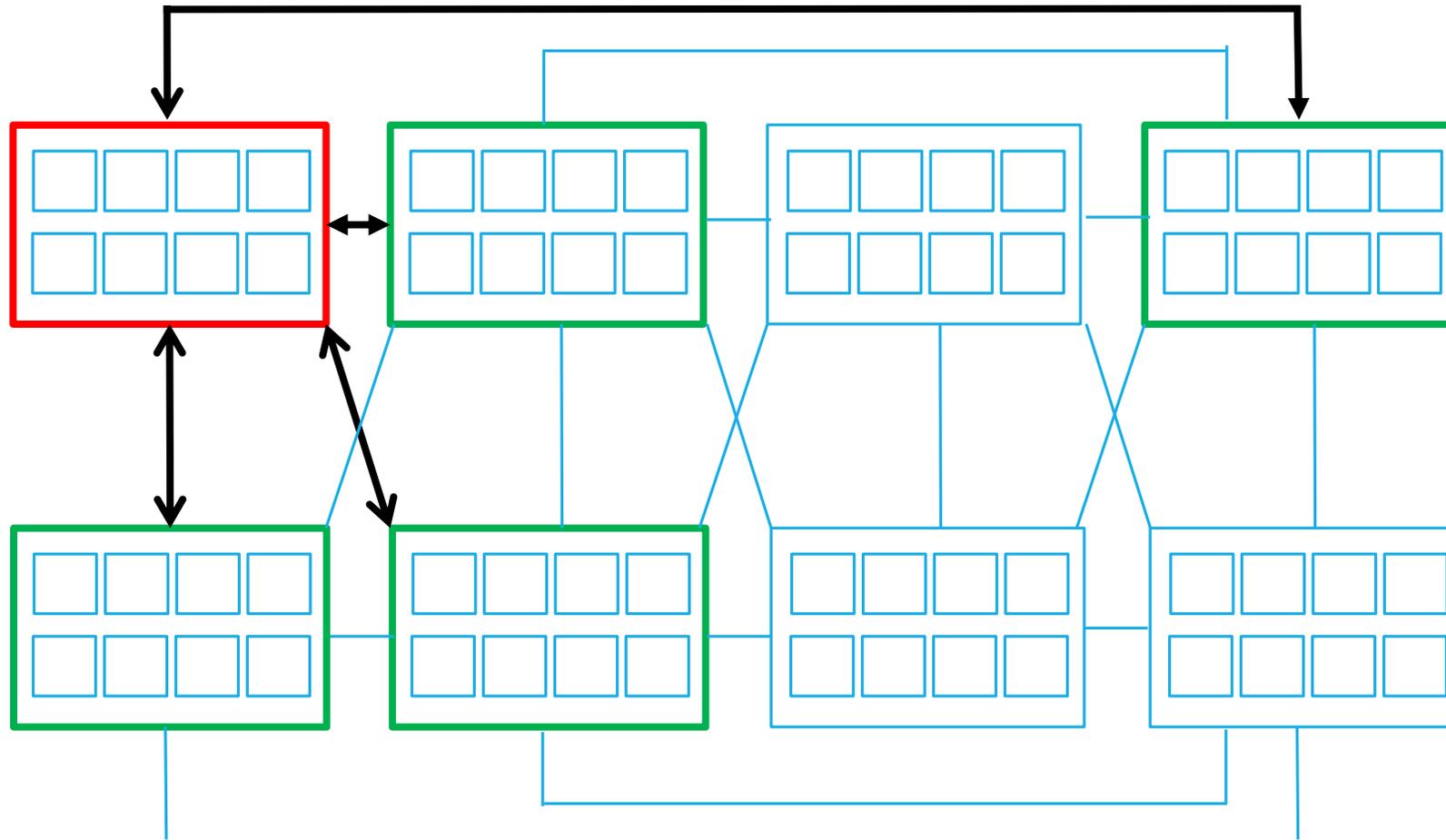
At the **first level**, the **first core** balances load with the other core **on the same pair** (because they share resources, high affinity)

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



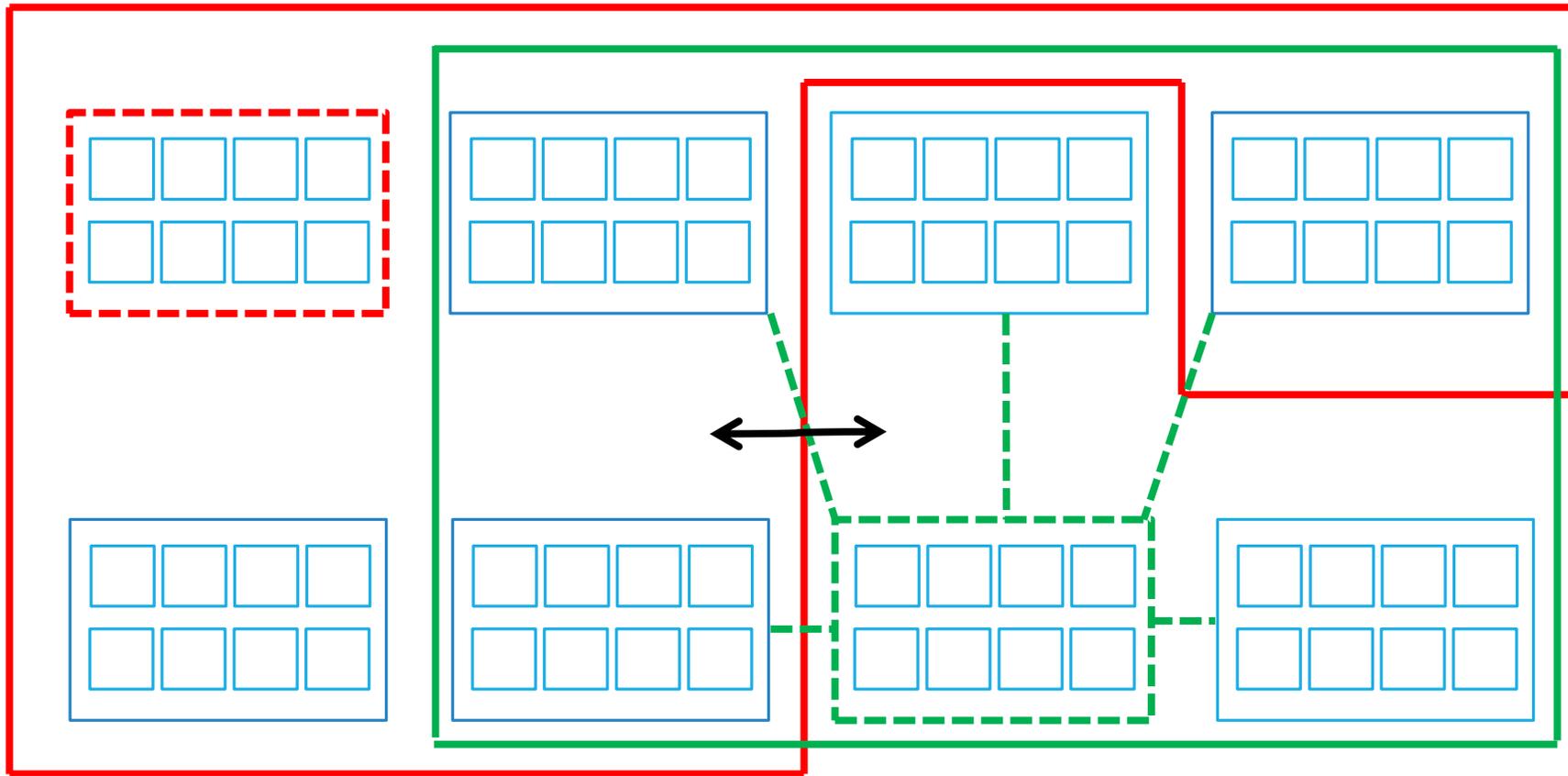
At the 2nd level,
the **first pair**
balances load
with other pairs
on the same CPU

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



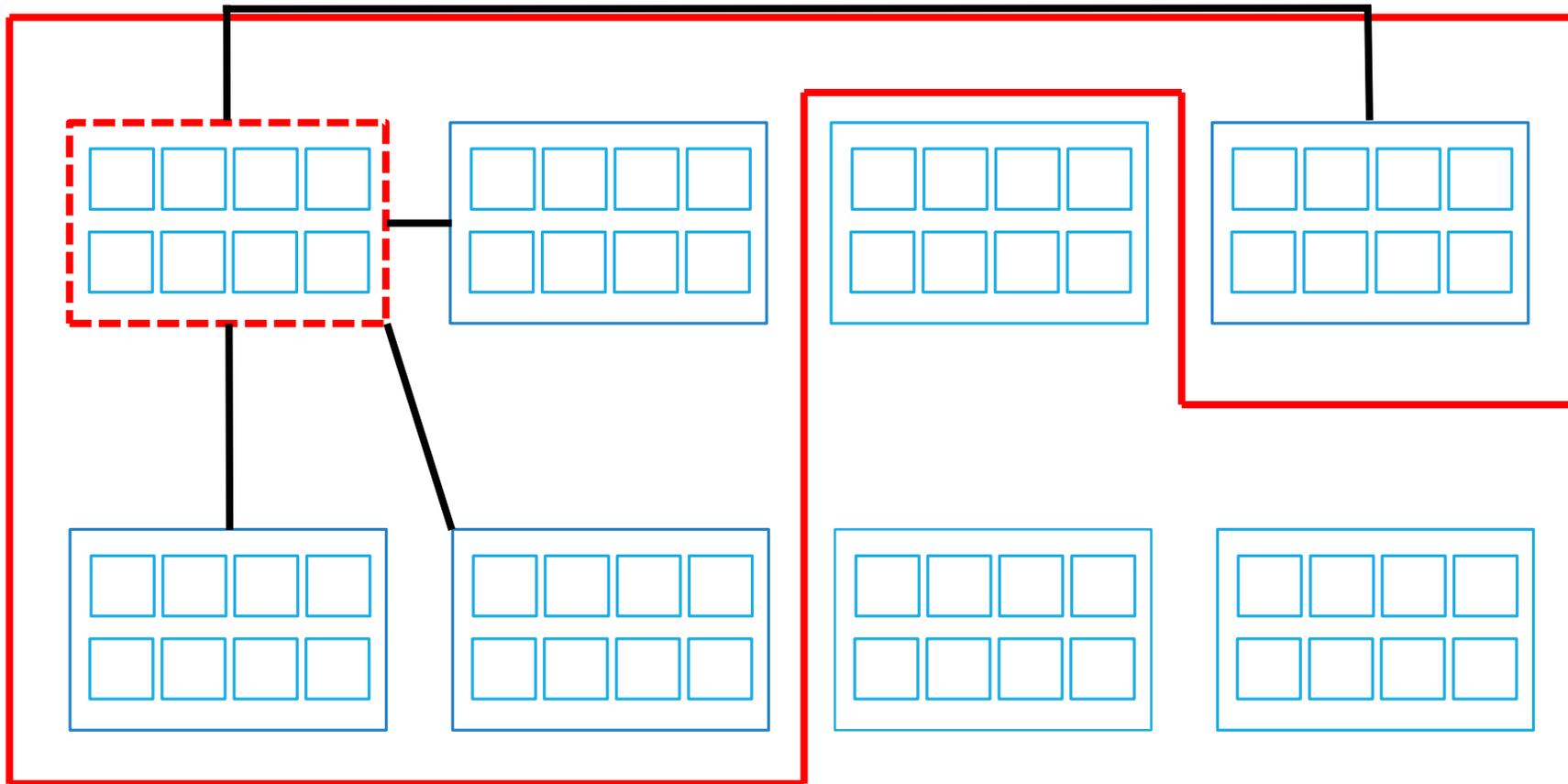
At the 3rd level,
the **first CPU**
balances load
with **directly**
connected CPUS

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



At the 4th level, the **first group of directly connected CPUs** balances load with **the other groups of directly connected CPUs**

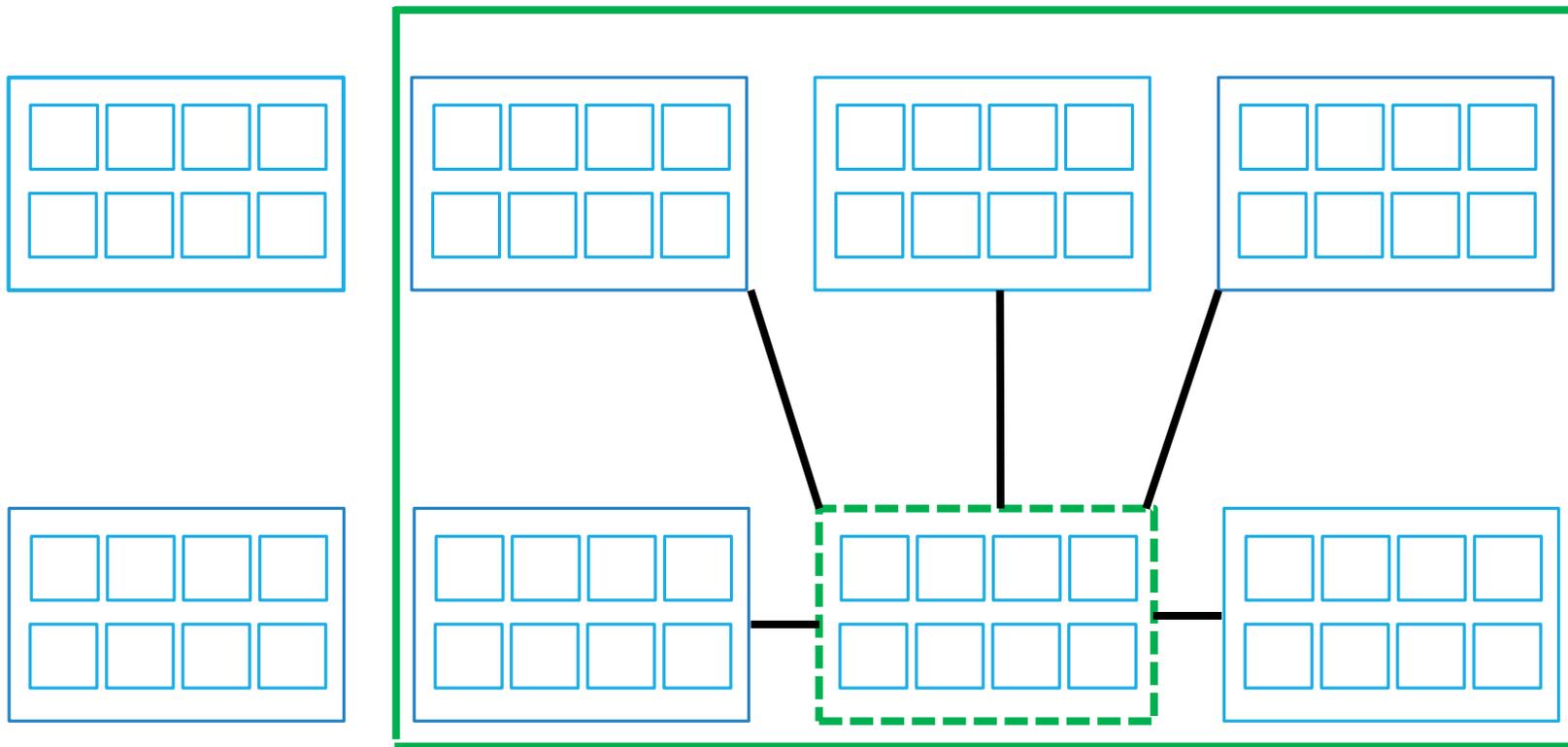
BUG 2/4: SCHEDULING GROUP CONSTRUCTION



**Groups of CPUs
built by:**

**(1) picking first
CPU and looking
for all directly
connected CPUs**

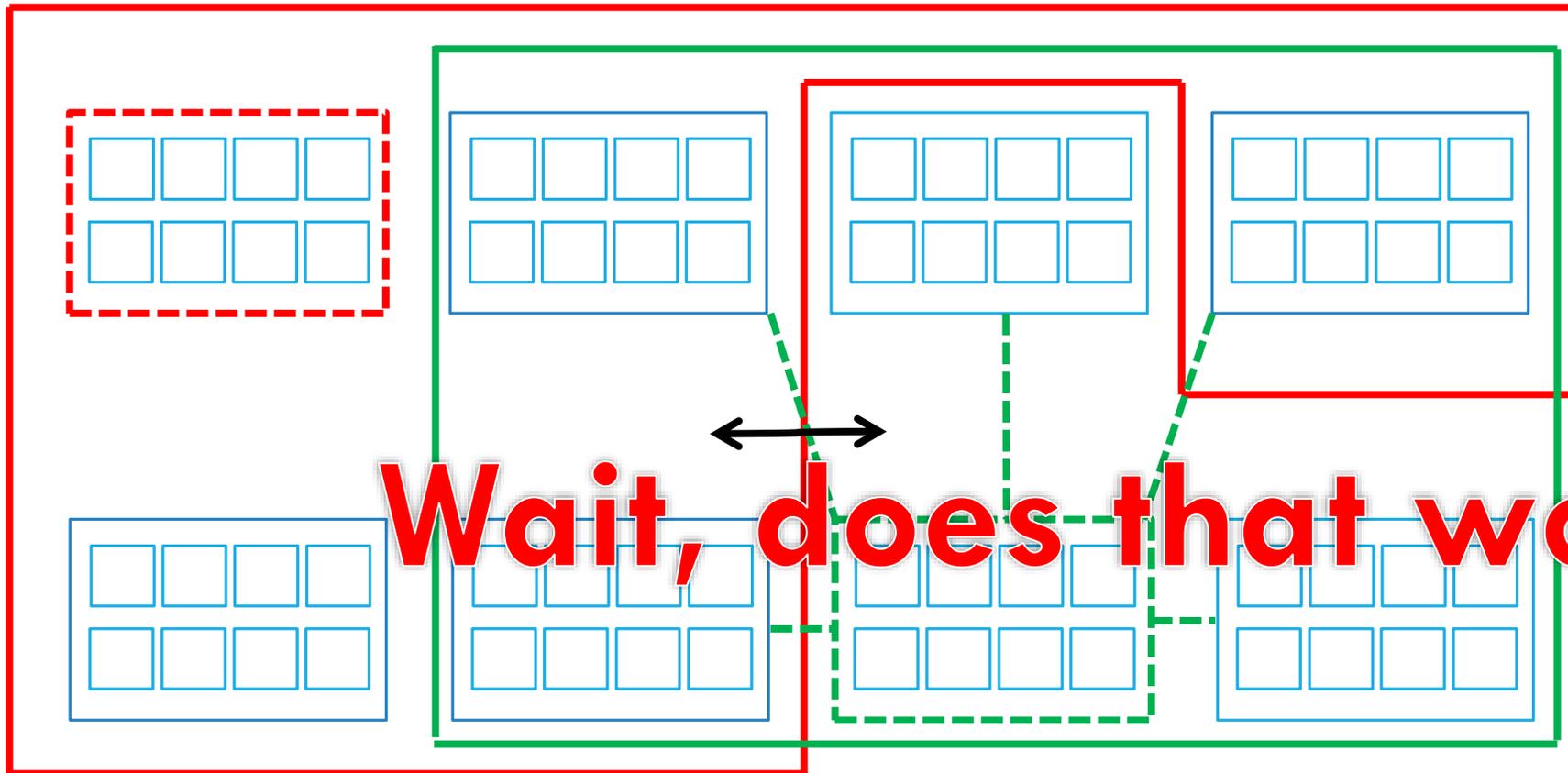
BUG 2/4: SCHEDULING GROUP CONSTRUCTION



**Groups of CPUs
built by:**

**(2) picking first
CPU not in a
group and
looking for all
directly
connected CPUs**

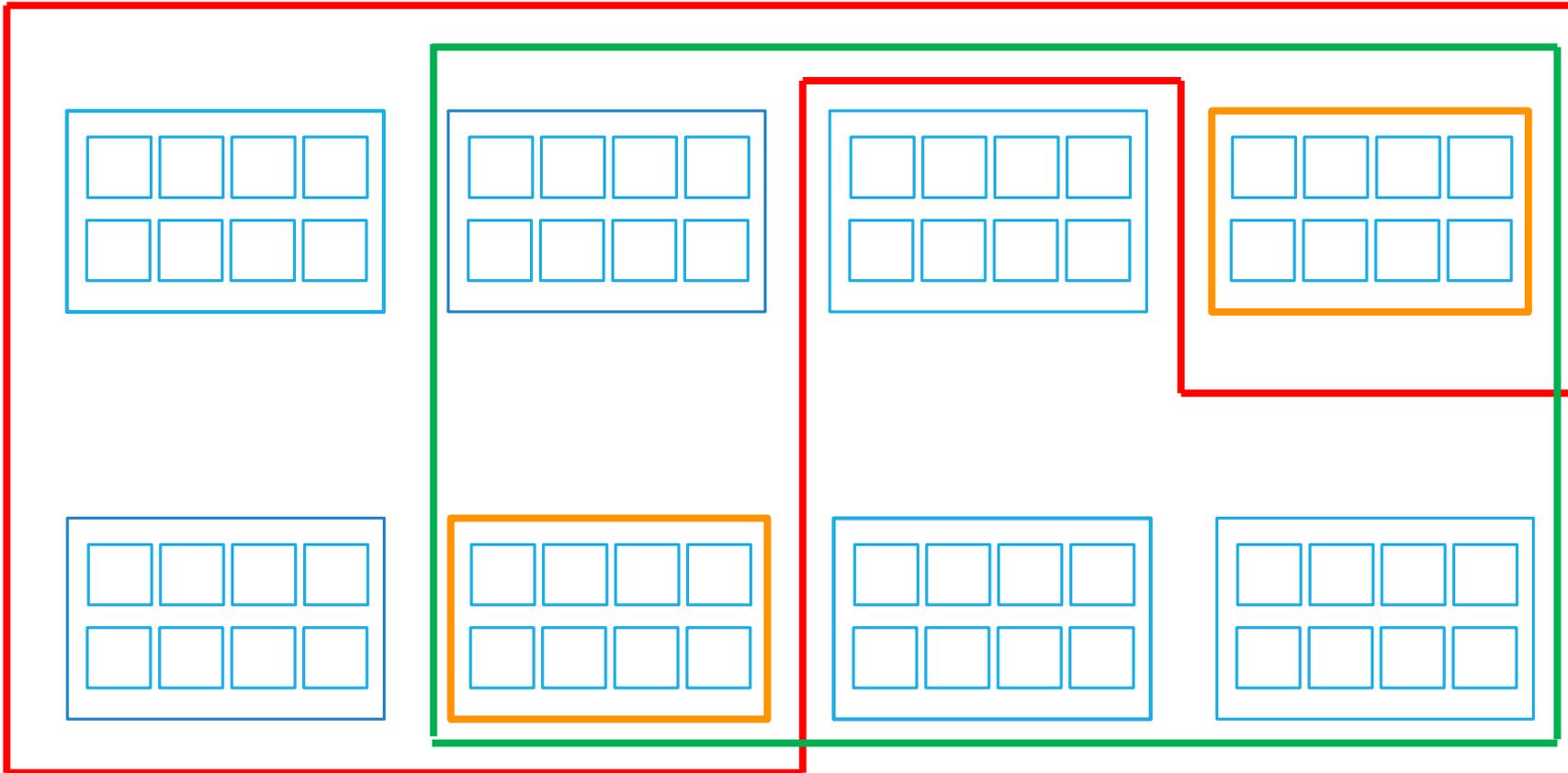
BUG 2/4: SCHEDULING GROUP CONSTRUCTION



And then stop,
because all CPUs
are in a group

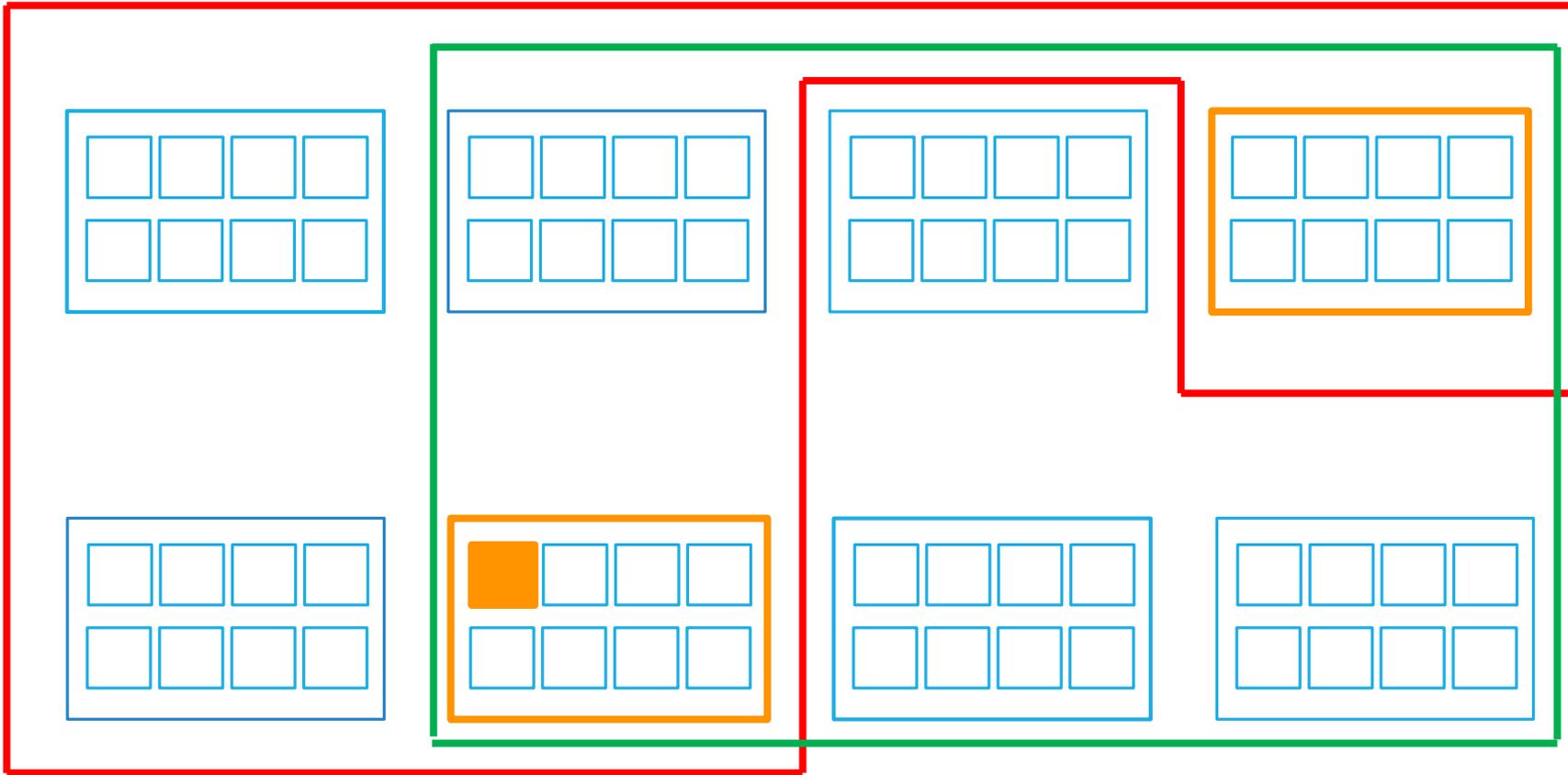
Wait, does that work?

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



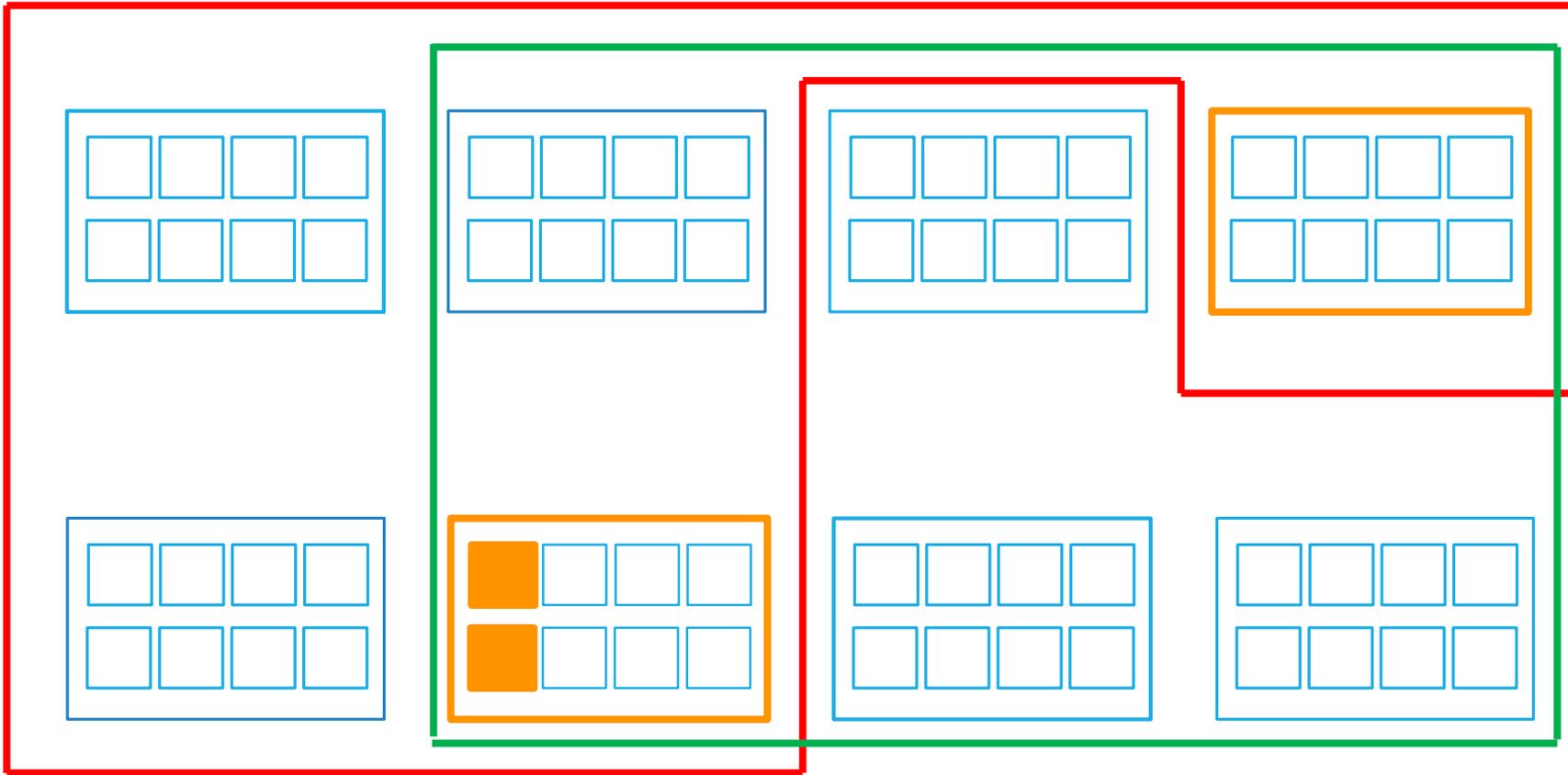
Suppose we taskset an application on **these two CPUs**, two hops apart (16 threads)

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



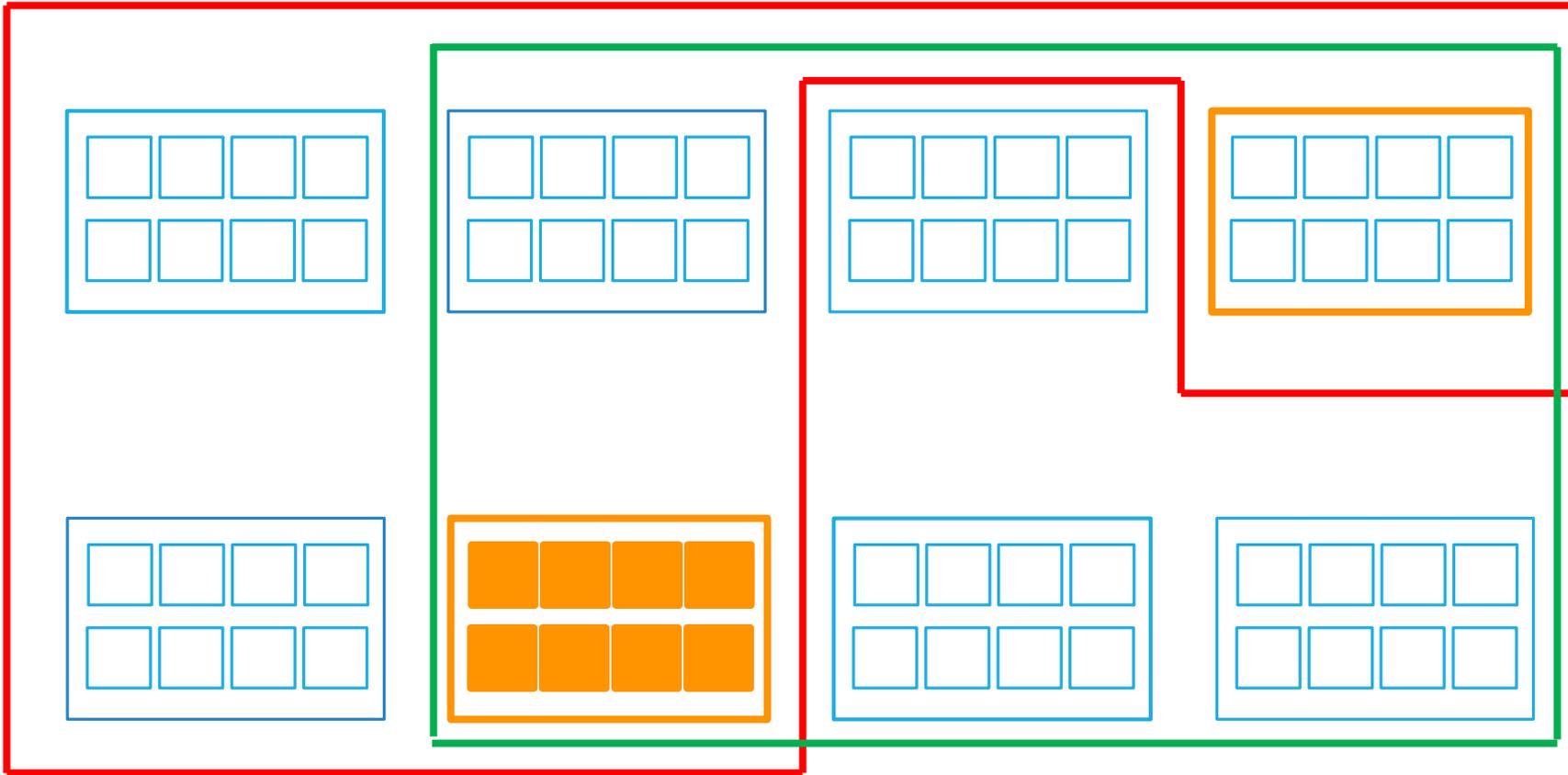
And threads
are created
on this core

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



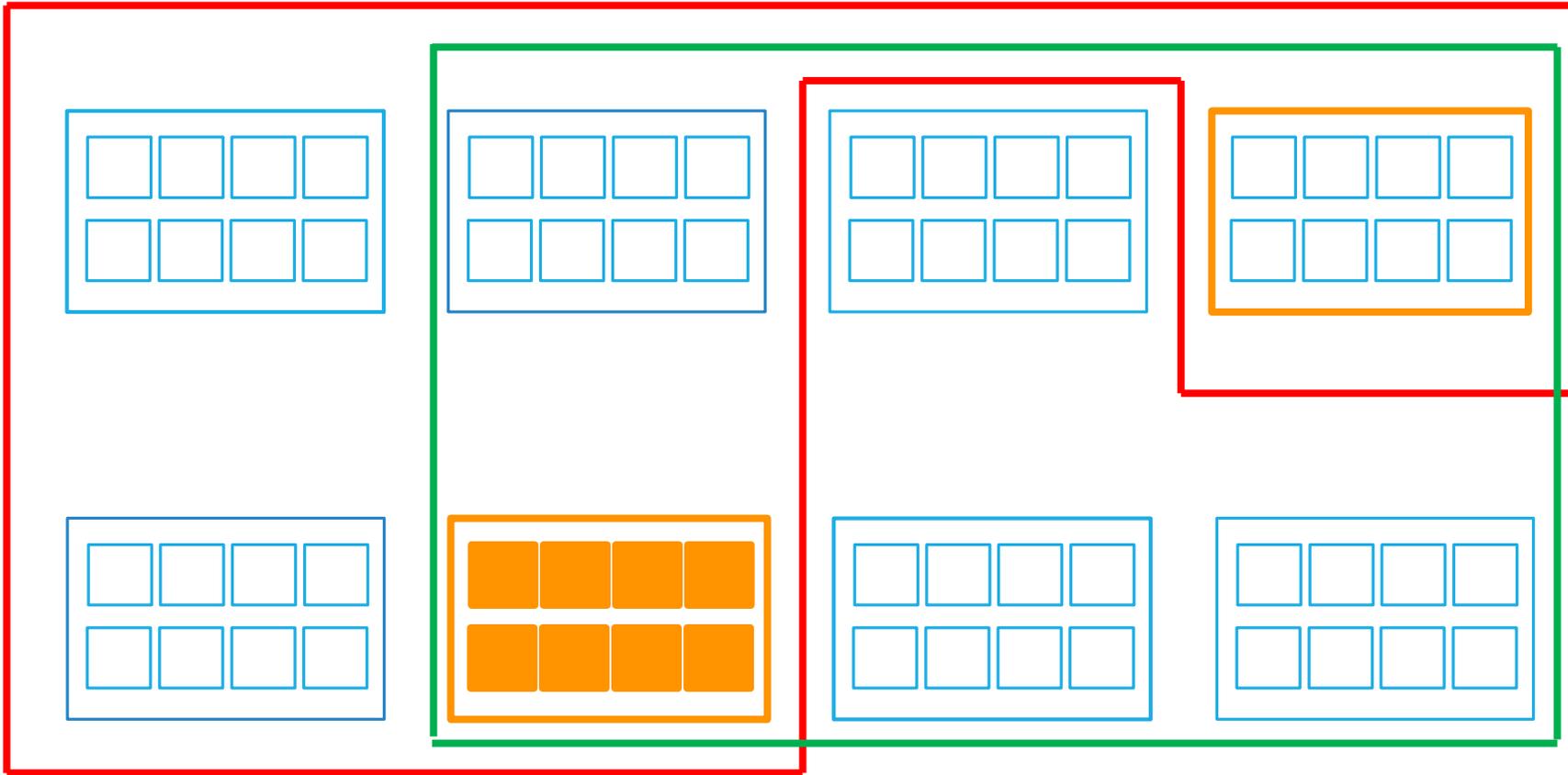
Load gets
correctly balanced
**on the pair of
cores**

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



Load gets
correctly balanced
on the CPU

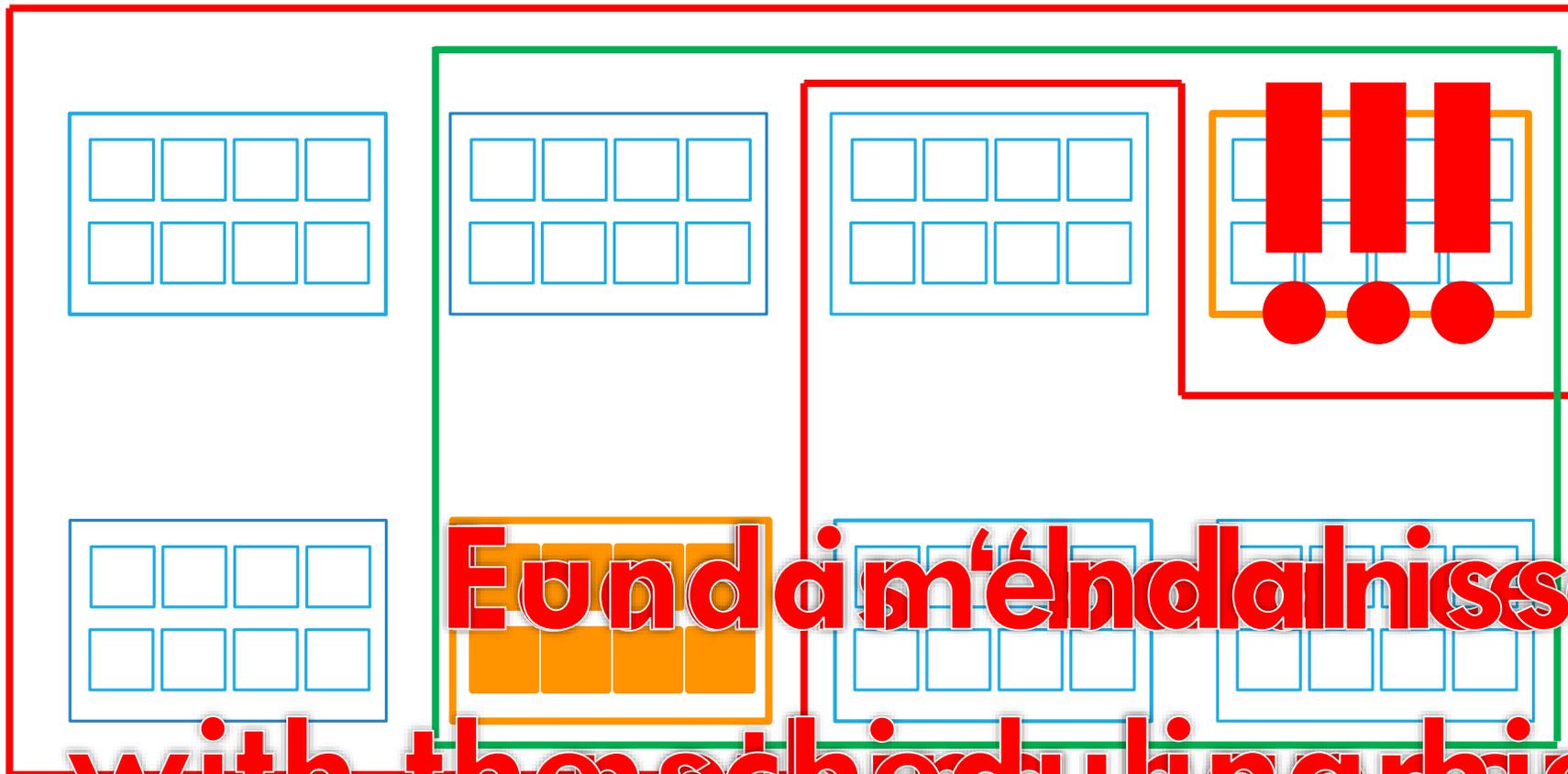
BUG 2/4: SCHEDULING GROUP CONSTRUCTION



At level 4,
stealing between
the **red** and **green**
groups...

**Overloaded node
in both groups!**

BUG 2/4: SCHEDULING GROUP CONSTRUCTION



$$\text{load}(\text{red}) = 16 * \text{load}(\text{thread})$$

$$\text{load}(\text{green}) = 16 * \text{load}(\text{thread})$$

Fundamental principle:
with the scheduling hierarchy!

BUG 2/4: SCHEDULING GROUP CONSTRUCTION

- **Fix: build the domains by creating one “directly connected” group for every CPU**
 - Instead of the first CPU and the first one not “covered” by a group
- Performance improvement of NAS applications on two nodes :

Application	With bug	After fix	Improvement
BT	99	56	1.75x
CG	42	15	2.73x
EP	73	36	2x
LU	1040	38	27x

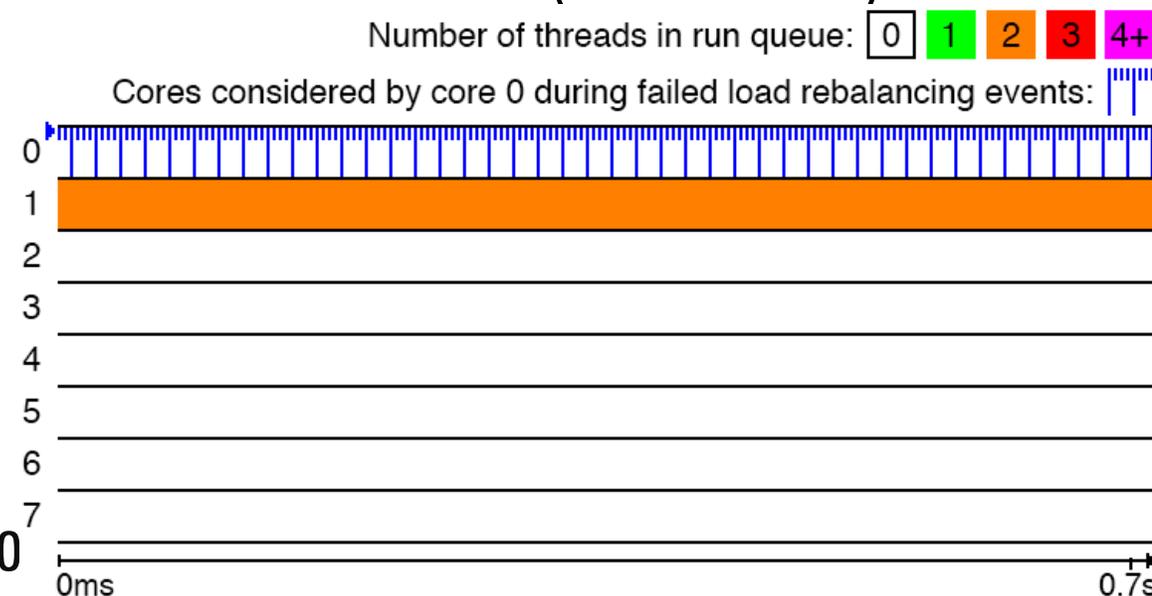
- **Very good improvement for LU because more threads than cores if can't use 16 cores**
 - Solves spinlock issues (incl. potential convoys)

BUG 3/4: MISSING SCHEDULING DOMAINS

- In addition to this, when domains re-built, **levels 3 and 4 not re-built...**
 - I.e., no balancing between directly connected or 1-hop CPUs (i.e. any CPU)
 - **Happens for instance when disabling and re-enabling a core**
- **Launch an application, first thread created on CPU 1**
 - First thread will stay on CPU 1, next threads will be created on CPU 1 (default Linux)
 - All the threads will be on CPU 1 forever!

Application	With bug	After fix	Improvement
BT	122	23	5.2x
CG	134	5.4	25x
EP	72	18	4x
LU	2196	16	137x

THE OS SCHEDULER: A PERFORMANCE-CRITICAL CO



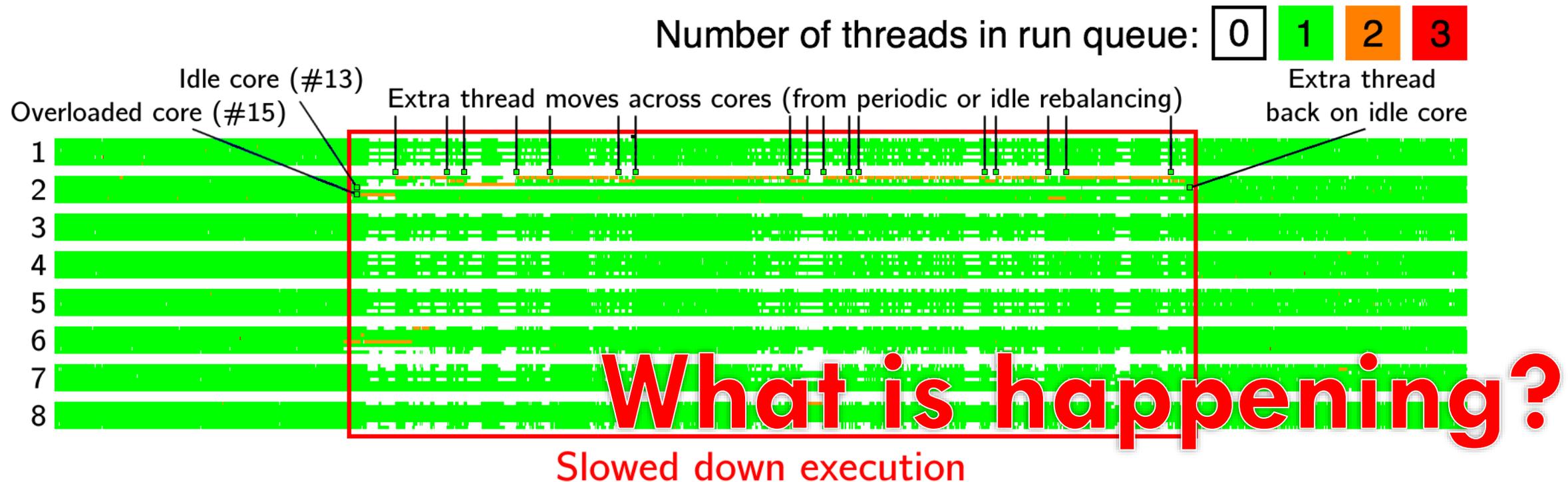
BUG 4/4: OVERLOAD-ON-WAKEUP

- Until now, we analyzed the behavior of the the periodic, **(buggy)** hierarchical load balancing that uses **(buggy)** scheduling domains
- **But there is another way load is balanced:** threads get to pick on which core they get woken up when they are done blocking (after a lock acquisition, an I/O)...
- **Here is how it works:** when a thread wakes up, it looks for non-busy cores on the same CPU in order to decide on which core it should wake up.
- **Only cores that are on the same CPU, in order to improve data locality...**

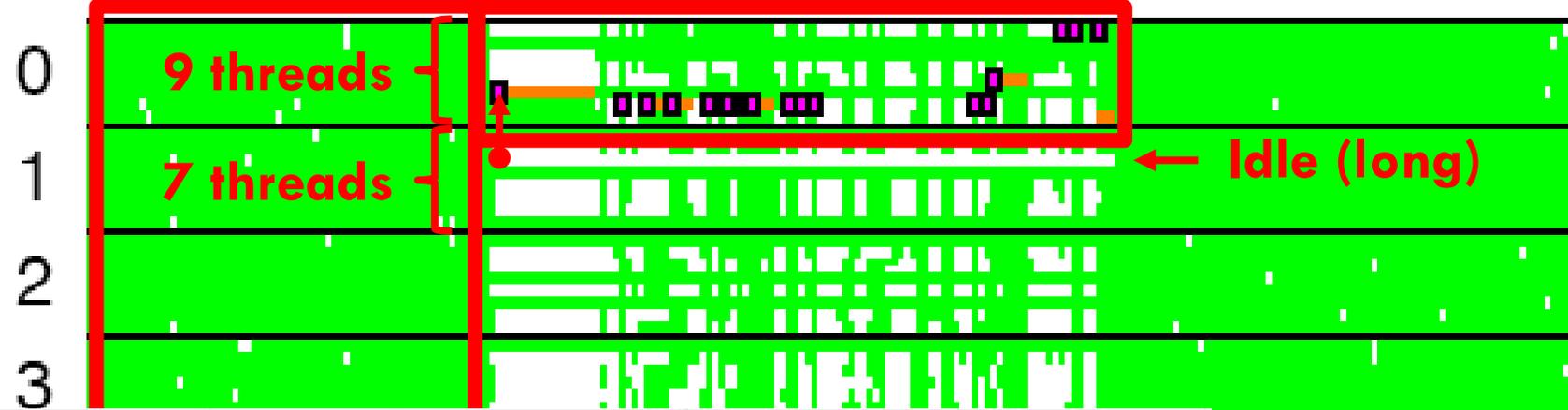
Wait, does that work?

BUG 4/4: OVERLOAD-ON-WAKEUP

- Commercial DB with TPC-H, 64 threads on 64 cores, nothing else on the machine.
- With threads pinned to cores, works fine. **With Linux scheduling, execution much slower, phases with overloaded cores while there are long-term idle cores!**



BUG 4/4



- Begin
- Occa during
- **Now,** all exe
- **Barrie**

Bug fixes	TPC-H request #18	Full TPC-H benchmark
None	55.9s	542.9s
<i>Group Imbalance</i>	48.6s (−13.1%)	513.8s (−5.4%)
<i>Overload-on-Wakeup</i>	43.5s (−22.2%)	471.1s (−13.2%)
Both	43.3s (−22.6%)	465.6s (−14.2%)

ared
ows down
ywhere...
up on

idle core, because waking up algorithm only considers local CPU!

- **Periodic rebalancing can't rebalance load most of the time because many idle cores**
⇒ Hard to see an imbalance between 9-thread and 7-thread CPU...
- **“Solution”**: wake up on core idle for the longest time (not great for energy)

WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem
- **To recap, on Linux, load balancing works that way:**
 - Hierarchical rebalancing uses a metric named *load*,
↑ **Found fundamental issue here**
 - to periodically balance threads between *scheduling domains*.
↑ **Found fundamental issue here**
 - In addition to this, threads balance load by *selecting core where to wake up*.
↑ **Found fundamental issue here**

Wait, was anything working at all? 😊

WHERE DO WE GO FROM HERE?

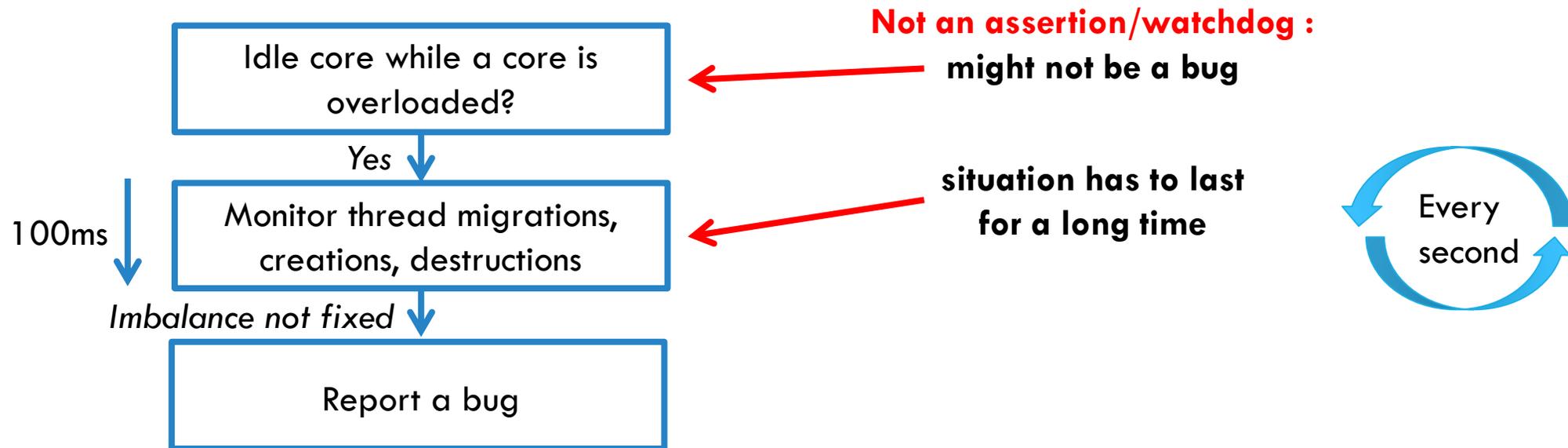
Many major issues went unnoticed for years in the scheduler...

How can we prevent this from happening again?

- **Code testing**
 - No clear fault (no crash, no deadlock, etc.)
 - Existing tools don't target these bugs
- **Performance regression**
 - Usually done with 1 app on a machine to avoid interactions
 - Insufficient coverage
- **Model checking, formal proofs**
 - Complex, parallel code: so far, nobody knows how to do it...

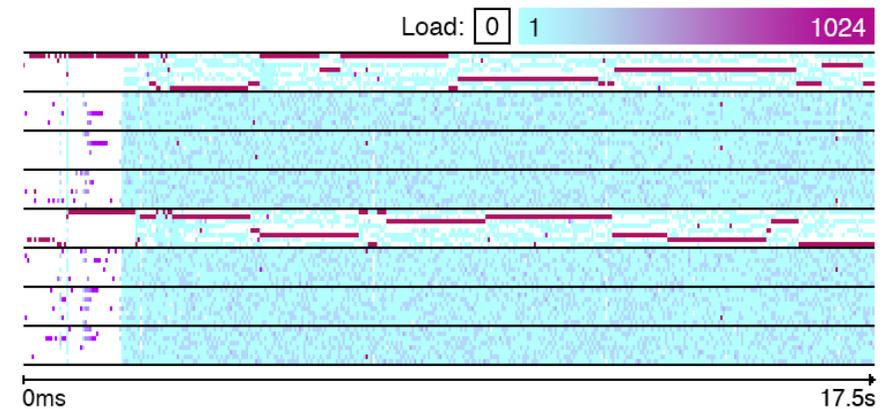
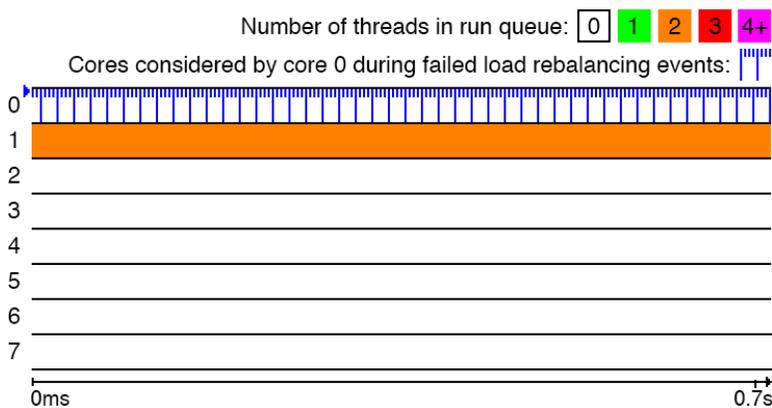
WHERE DO WE GO FROM HERE?

- **Idea 1:** short-term hack — implemented a sanity checker



WHERE DO WE GO FROM HERE?

- **Idea 2:** fine-grained tracers!
 - Built a simple one, turned out to be the only way to really understand what happens
 - Aggregate metrics (CPI, cache misses, etc.) not precise enough



- **Could really be improved!**

WHERE DO WE GO FROM HERE?

- **Idea 3:** produce a dedicated profiler!
 - Lack of tools!
 - Possible to detect if slowdown comes from scheduler or application?
 - Would avoid a lot of wasted time!
 - Follow threads, and see if often on overloaded cores when shouldn't have?
 - Detect if threads unnecessarily moved to core/node that leads to many cache misses?

WHERE DO WE GO FROM HERE?

- **Idea 4:** produce good scheduler benchmarks!
 - **Really needed, and virtually inexistent!**
 - **Not an easy problem:** insane coverage needed!
 - **Using combination of many real applications:** configuration nightmare!
- Simulated workloads?
 - Have to do **elaborate work:** spinning and sleeping not efficient
 - Have to be **representative of reality**, have to **cover corner cases**
 - *Use machine learning? Genetic algorithms?*

WHERE DO WE GO FROM HERE?

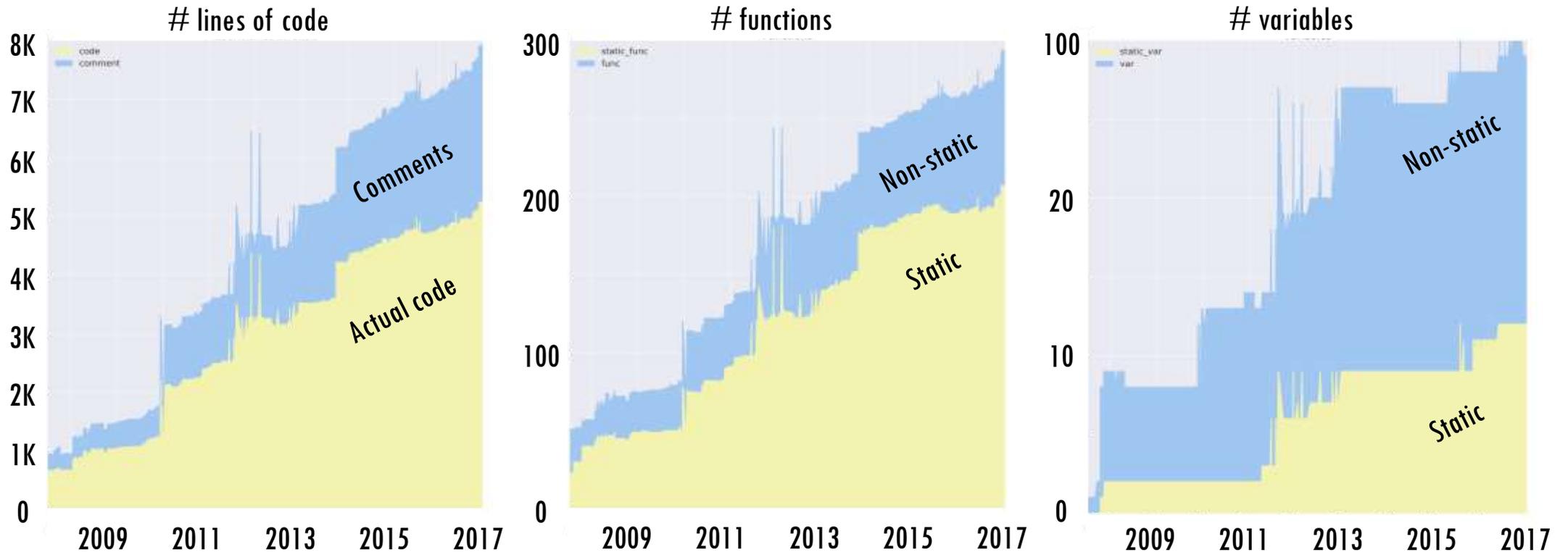
- **Idea 5:** switch to simpler schedulers, easier to reason about!

Let's take a step back: *why* did we end up in this situation?

- Linux used for many classes of applications (big data, n-tier, cloud, interactive, DB, HPC...)
- Multicore architectures increasingly diverse and complex!
- **Result:** very complex monolithic scheduler supposed to work in all situations!
 - Many heuristics interact in complex, unpredictable ways
 - Some features greatly complexify, e.g., load balancing (tasksets, cgroups/autogroups...)
- Keeps getting worse!
 - **E.g., task_struct:** 163 fields in Linux 3.0 (07/2011), 215 fields in 4.6 (05/2016)
 - **20,000 lines of C!**

WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!



WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!
- **Proving the scheduler implementation correct: not doable!**
 - Way too much code for current technology
 - We'd need to detect high-level abstractions from low-level C: a challenge!
 - Even if we managed that, how do we keep up with updates?
 - *Code keeps evolving with new architectures and application needs...*
- *We need another approach...*

WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!
- **Write simple, schedulers with proven properties !**
 - A scheduler is tailored to a (class of) parallel application(s)
 - Specific thread election criterion, load balancing criterion, state machine with events...
 - **Machine partitioned into sets of cores that run \neq schedulers**
 - Scheduler deployed together with (an) application(s) on a partition
- **Through a DSL, for two reasons:**
 - Much easier, safer and less bug-prone than writing low-level C kernel code !
 - Clear abstractions, possible to reason about them and prove properties
 - *Work conservation, load balancing live and in finite # or rounds, valid hierarchy...*

WHERE DO WE GO FROM HERE?

- **Idea 6: ???**
- Any other ideas?

CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.
- **Analysis:** fundamental issues in the load metric, scheduling domains, scheduling choices...
- Very bug-prone implementation following years of adapting to hardware
- **Can't ensure simple "invariant": no idle cores while overloaded cores**
- **Proposed fixes:** not always satisfactory
- **What can we do?** Many things to explore!

▪ **Our takeaway:** *more research must be directed towards implementing efficient and reliable schedulers for multicore architectures!*

Your turn!