

# THE LINUX SCHEDULER: A DECADE OF WASTED CORES

Jean-Pierre Lozi  
[jplozi@unice.fr](mailto:jplozi@unice.fr)



Baptiste Lepers  
[baptiste.lepers@epfl.ch](mailto:baptiste.lepers@epfl.ch)



Fabien Gaud  
[me@fabiengaud.net](mailto:me@fabiengaud.net)

The logo for COHO, with the word 'COHO' in large, bold, orange letters, and the letters 'D A T A' in smaller, grey letters below it.

Alexandra Fedorova  
[sasha@ece.ubc.ca](mailto:sasha@ece.ubc.ca)



Justin Funston  
[jfunston@ece.ubc.ca](mailto:jfunston@ece.ubc.ca)

Vivien Quéma  
[vivien.quema@imag.fr](mailto:vivien.quema@imag.fr)



# IS THE SCHEDULER OF YOUR MACHINE WORKING?

# IS THE SCHEDULER OF YOUR MACHINE WORKING?

- 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.**”*

# IS THE SCHEDULER OF YOUR MACHINE WORKING?

- 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.

# IS THE SCHEDULER OF YOUR MACHINE WORKING?

- 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 ?

# IS THE SCHEDULER OF YOUR MACHINE WORKING?

- 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?**

# IS THE SCHEDULER OF YOUR MACHINE WORKING?

- 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?**

# IS THE SCHEDULER OF YOUR MACHINE WORKING?

- 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

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.
- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.

# THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.
- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.
  - **Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.

# THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.
- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.
  - **Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.
  - **Contention over some resource (spinlock, etc.)?** We investigated this for a long time, but couldn't find anything that looked off.

# THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.
- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.
  - **Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.
  - **Contention over some resource (spinlock, etc.)?** We investigated this for a long time, but couldn't find anything that looked off.
  - **Overhead of context switches?** Threads moved a lot but we proved that the overhead was negligible.

# THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.
- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.
  - **Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.
  - **Contention over some resource (spinlock, etc.)?** We investigated this for a long time, but couldn't find anything that looked off.
  - **Overhead of context switches?** Threads moved a lot but we proved that the overhead was negligible.
- **We ended up suspecting the core behavior of the scheduler.**

# THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.
- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.
  - **Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.
  - **Contention over some resource (spinlock, etc.)?** We investigated this for a long time, but couldn't find anything that looked off.
  - **Overhead of context switches?** Threads moved a lot but we proved that the overhead was negligible.
- **We ended up suspecting the core behavior of the scheduler.**
  - *We implemented high-resolution tracing tools and saw that some cores were idle while others overloaded...*

# 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...

# 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, analyzed and fixed**
  - **Always the same symptom: idle cores while others are overloaded**
  - The bug-hunting was tough, and led us to develop our own tools

# 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, analyzed and fixed**
  - **Always the same symptom: idle cores while others are overloaded**
  - The bug-hunting was tough, and led us to develop our own tools
- **After fixing some of the bugs :**
  - 12-23% performance improvement on a popular database with TPC-H
  - **137× performance improvement on HPC workloads**

# 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, analyzed and fixed**
  - **Always the same symptom: idle cores while others are overloaded**
  - The bug-hunting was tough, and led us to develop our own tools
- **After fixing some of the bugs :**
  - 12-23% performance improvement on a popular database with TPC-H
  - **137× performance improvement on HPC workloads**
- **Not always possible to provide a simple, working fix...**
  - Intrinsic problems with the design of the scheduler?

# THIS TALK

**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!*

# THIS TALK

**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...***

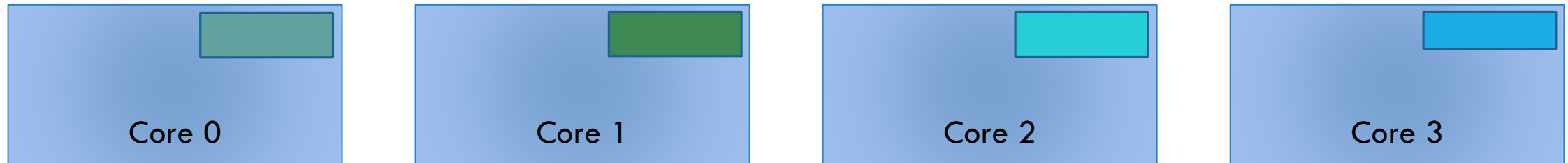
# THIS TALK

**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...*

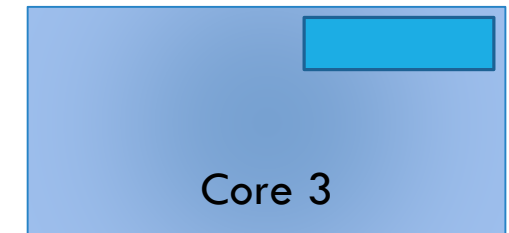
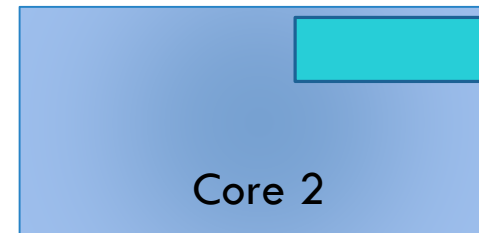
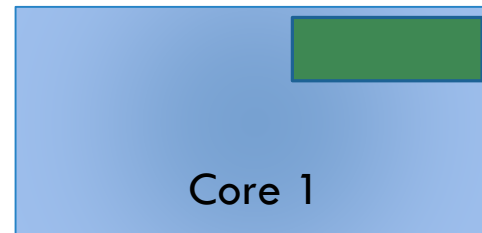
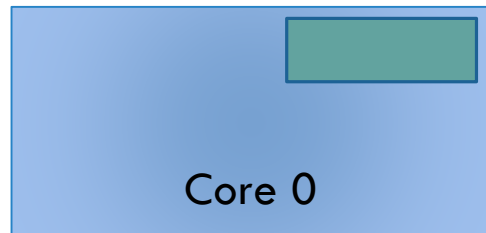
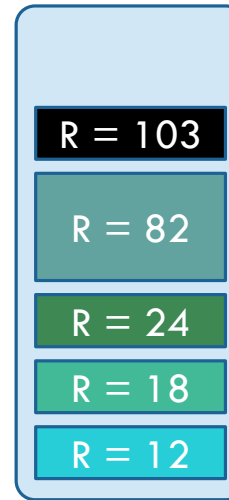
# THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT





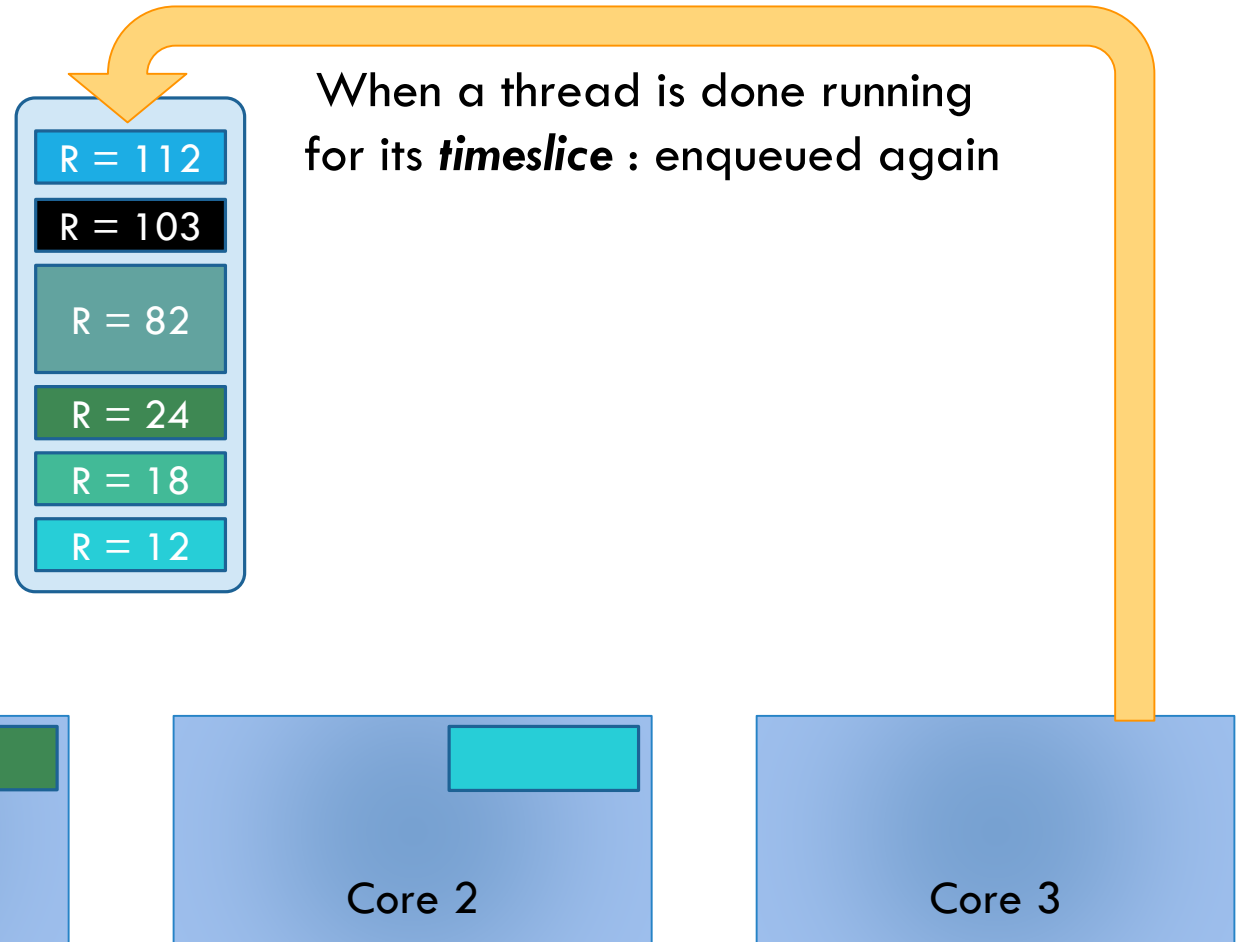
# THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by *runtime*



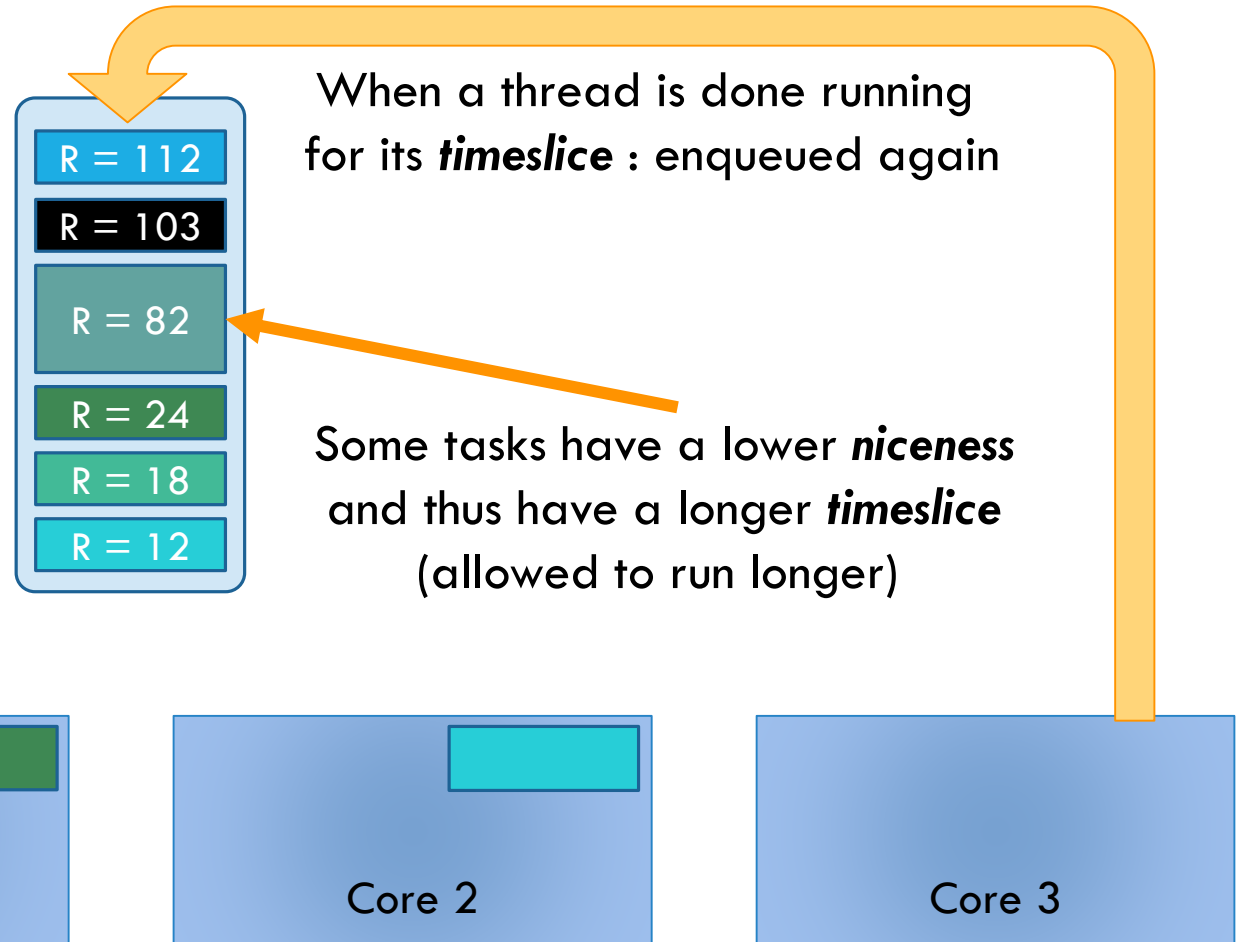
# THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by *runtime*



# THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by **runtime**



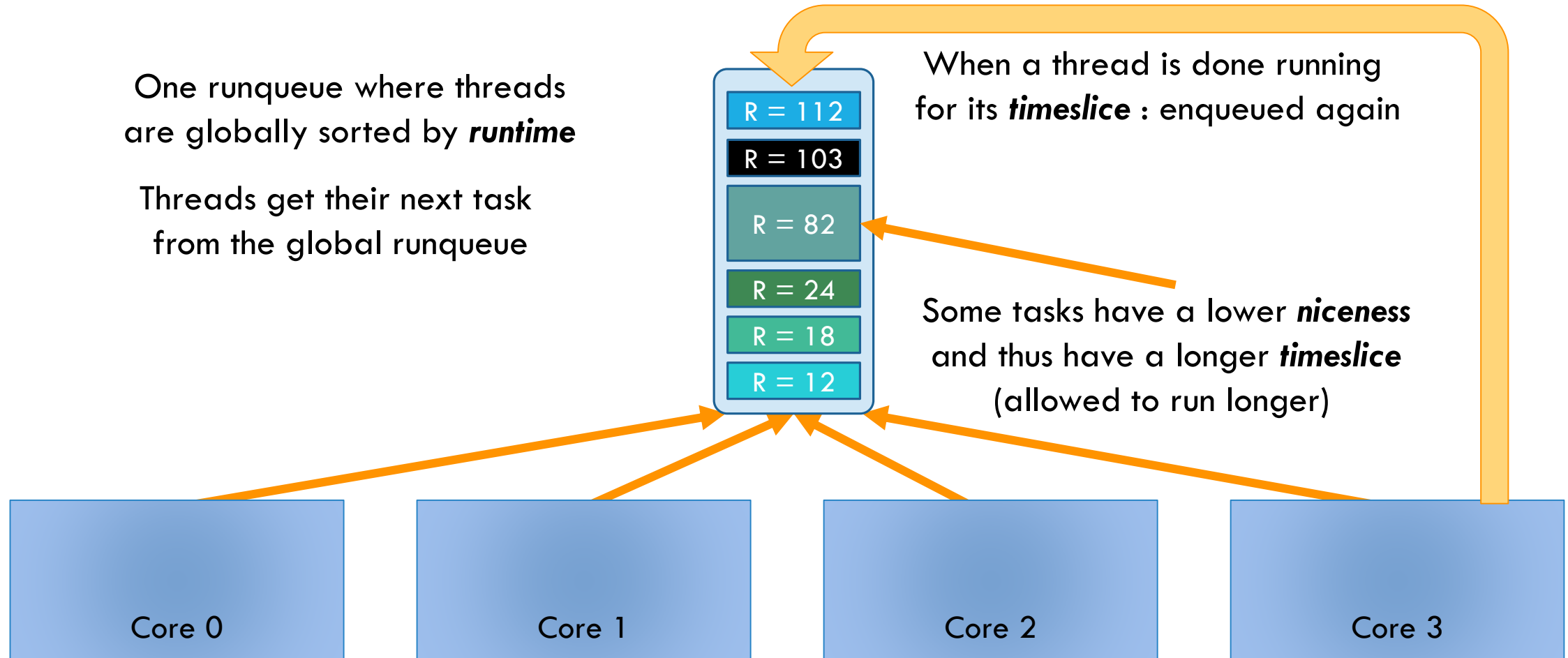
# THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by **runtime**

Threads get their next task from the global runqueue

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)

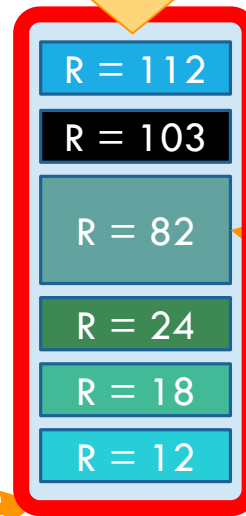


# THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by **runtime**

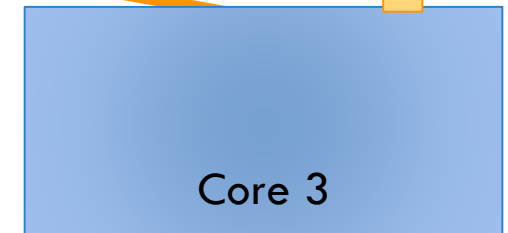
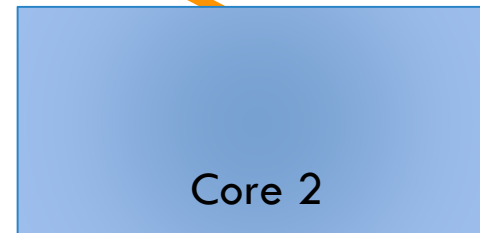
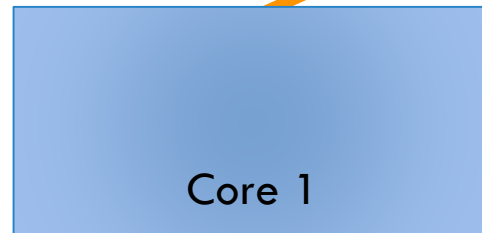
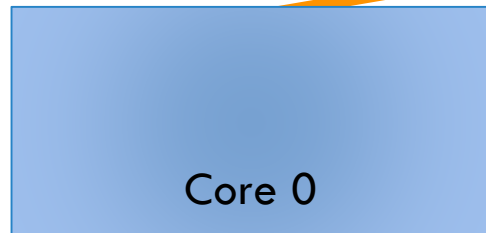
Threads get their next task from the global runqueue

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



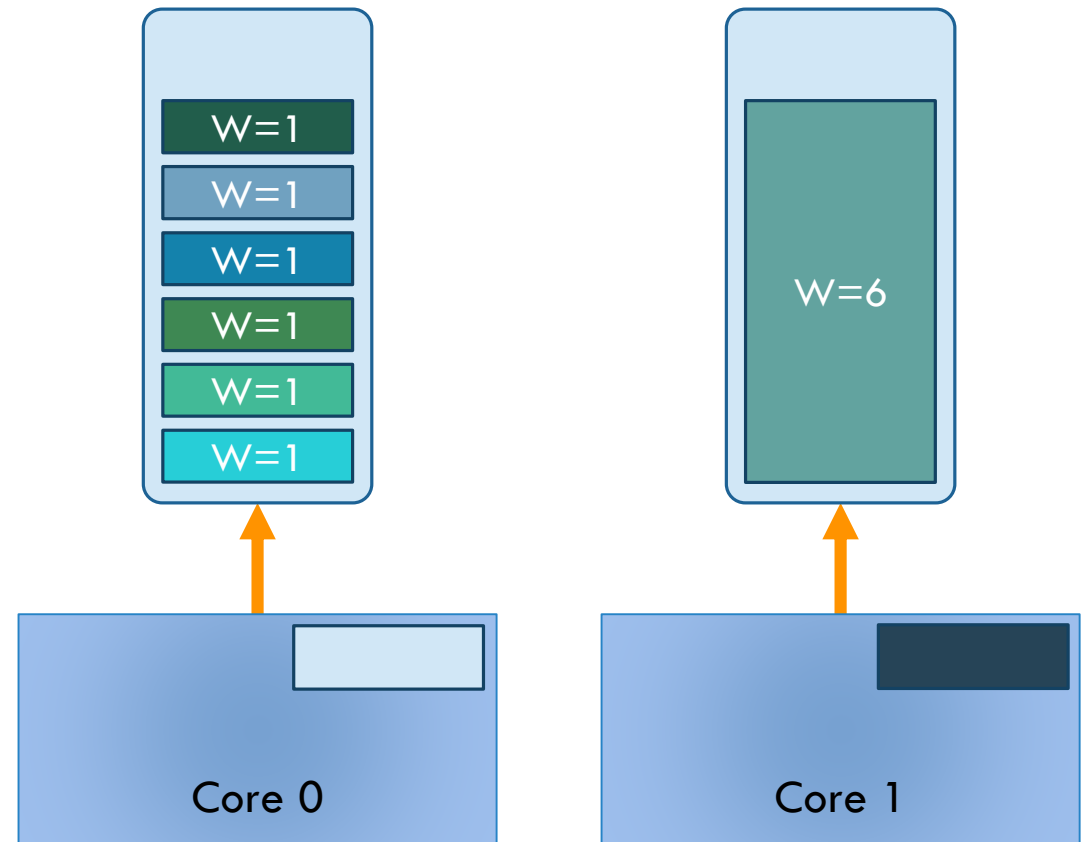
When a thread is done running for its **timeslice** : enqueued again

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

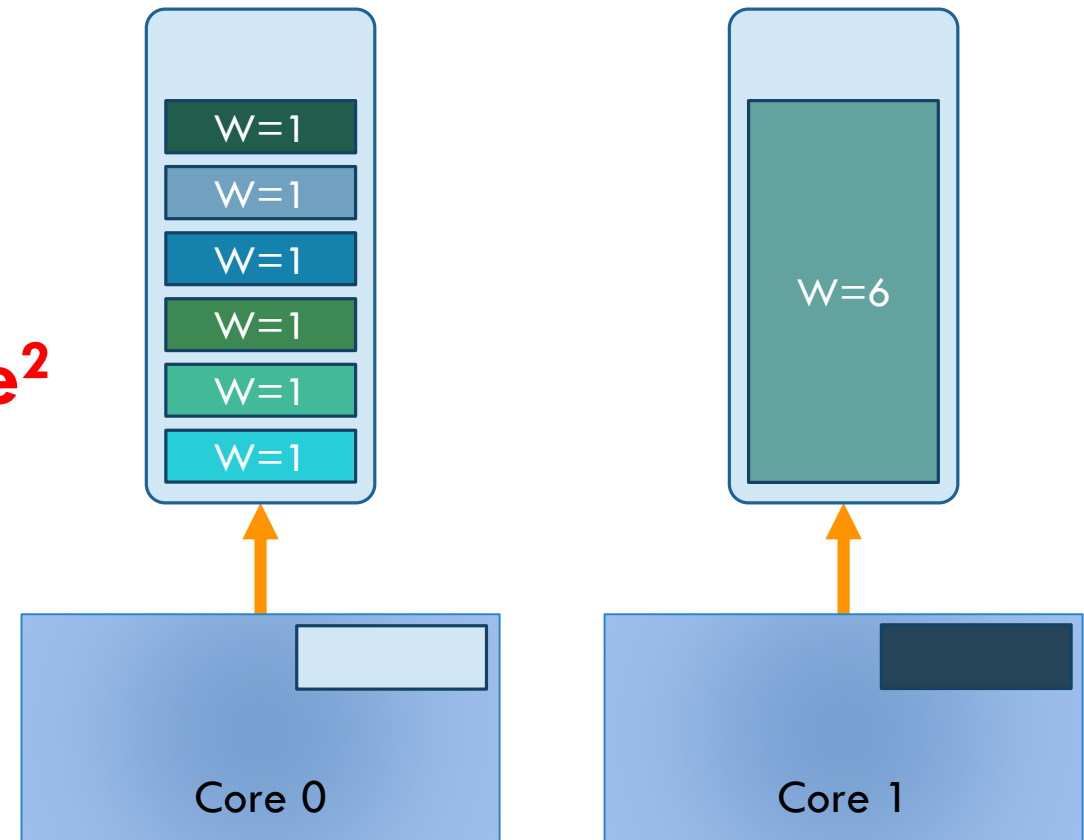


# CFS: IN PRACTICE

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

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

<sup>1</sup>The lower the niceness, the higher the weight



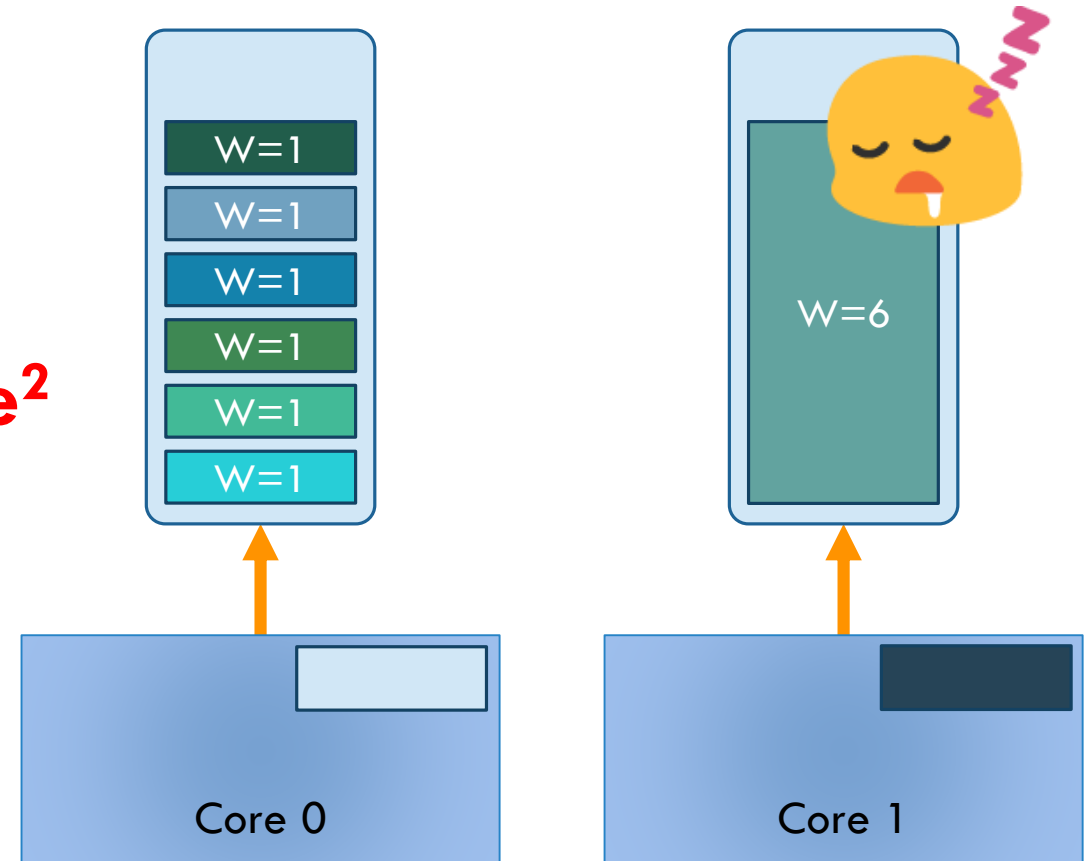
# CFS: IN PRACTICE

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

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

<sup>1</sup>The lower the niceness, the higher the weight

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





# CFS: IN PRACTICE

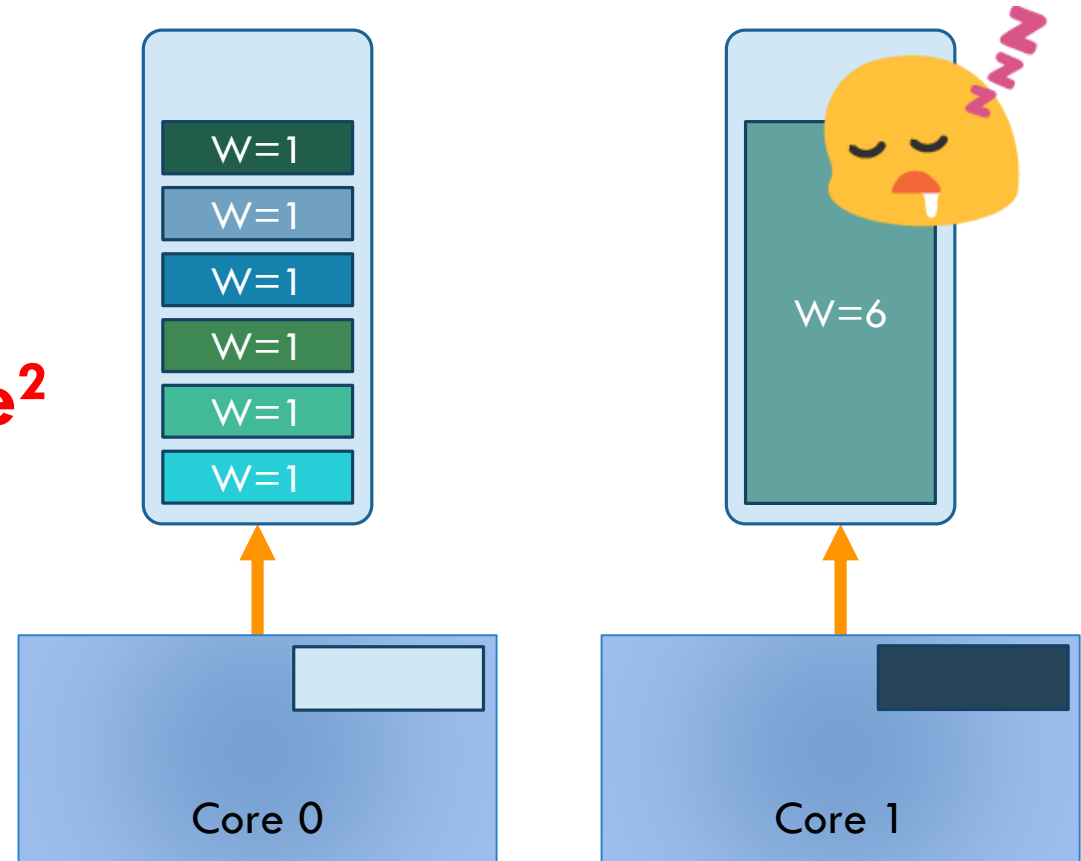
- One runqueue per core to avoid contention
- CFS **periodically** balances “loads”:

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

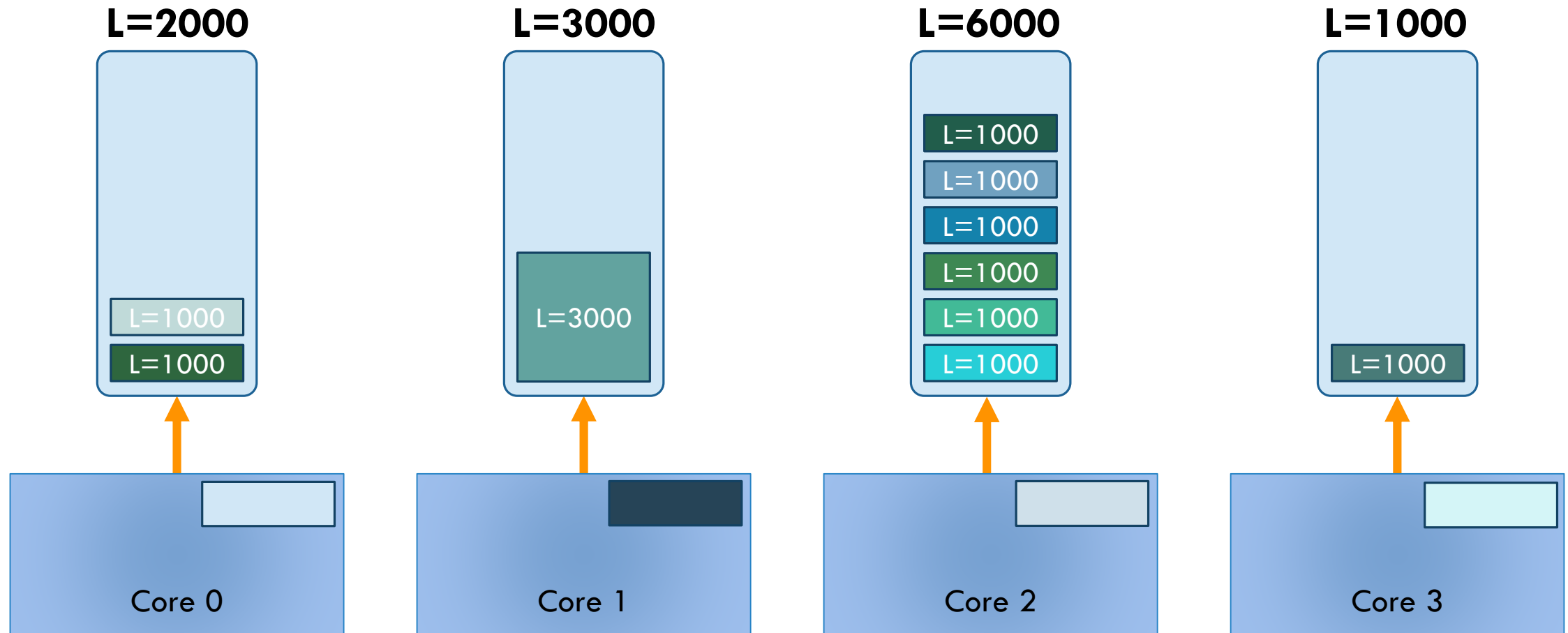
<sup>1</sup>The lower the niceness, the higher the weight

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

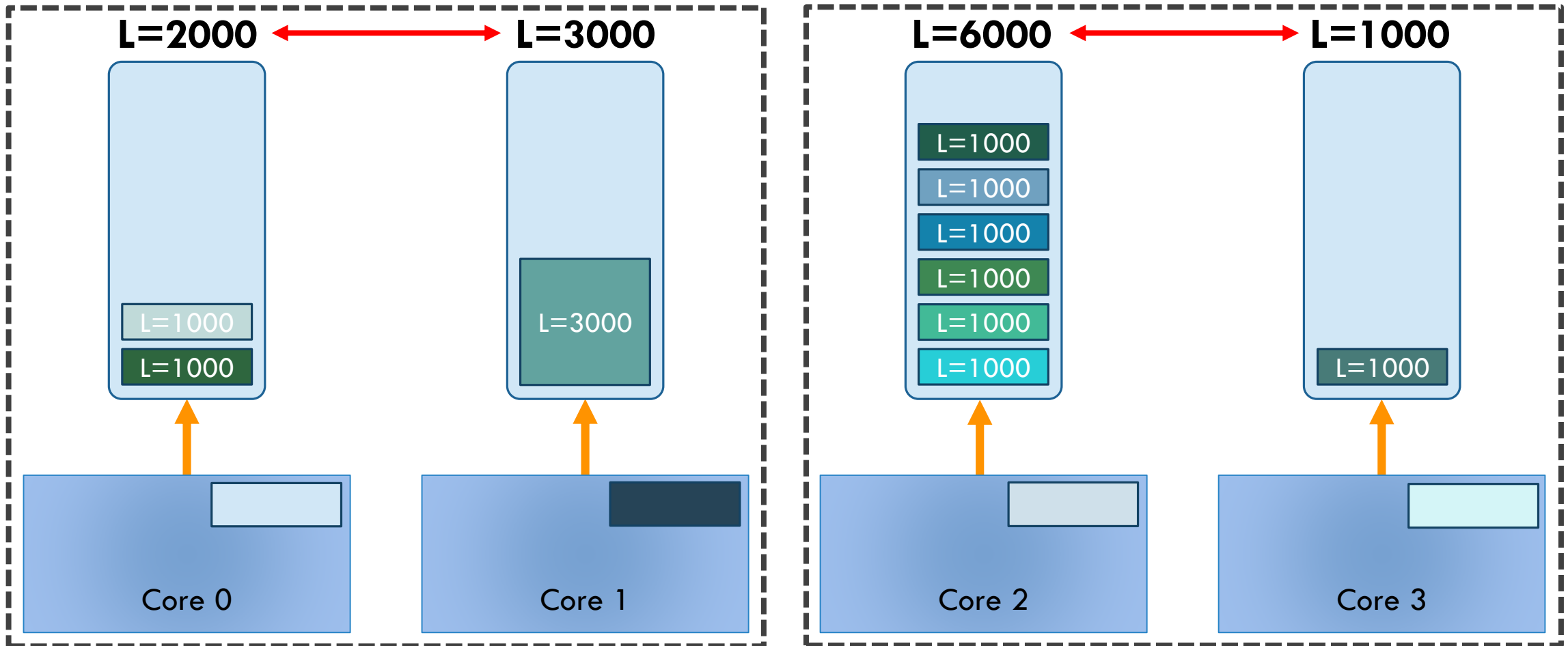
- Since there can be many cores: **hierarchical approach!**



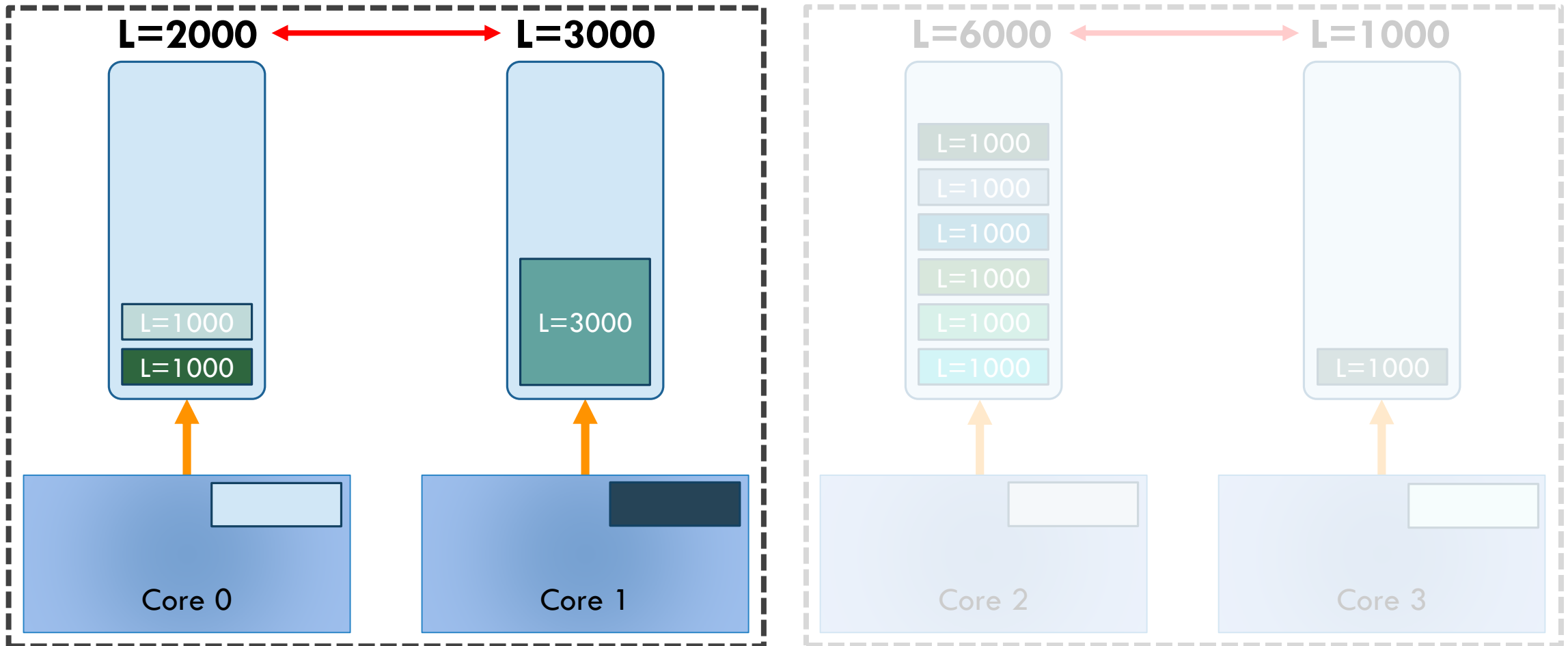
# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING



# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

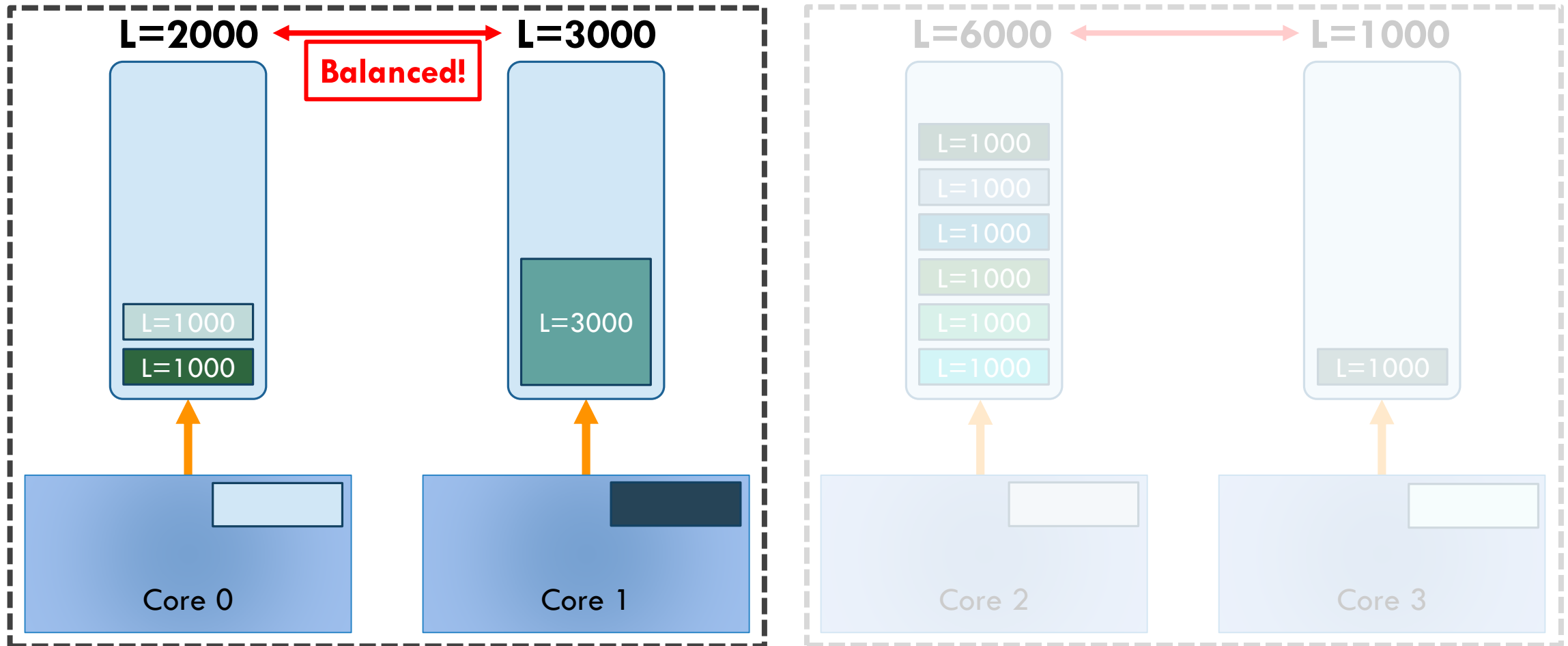


# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

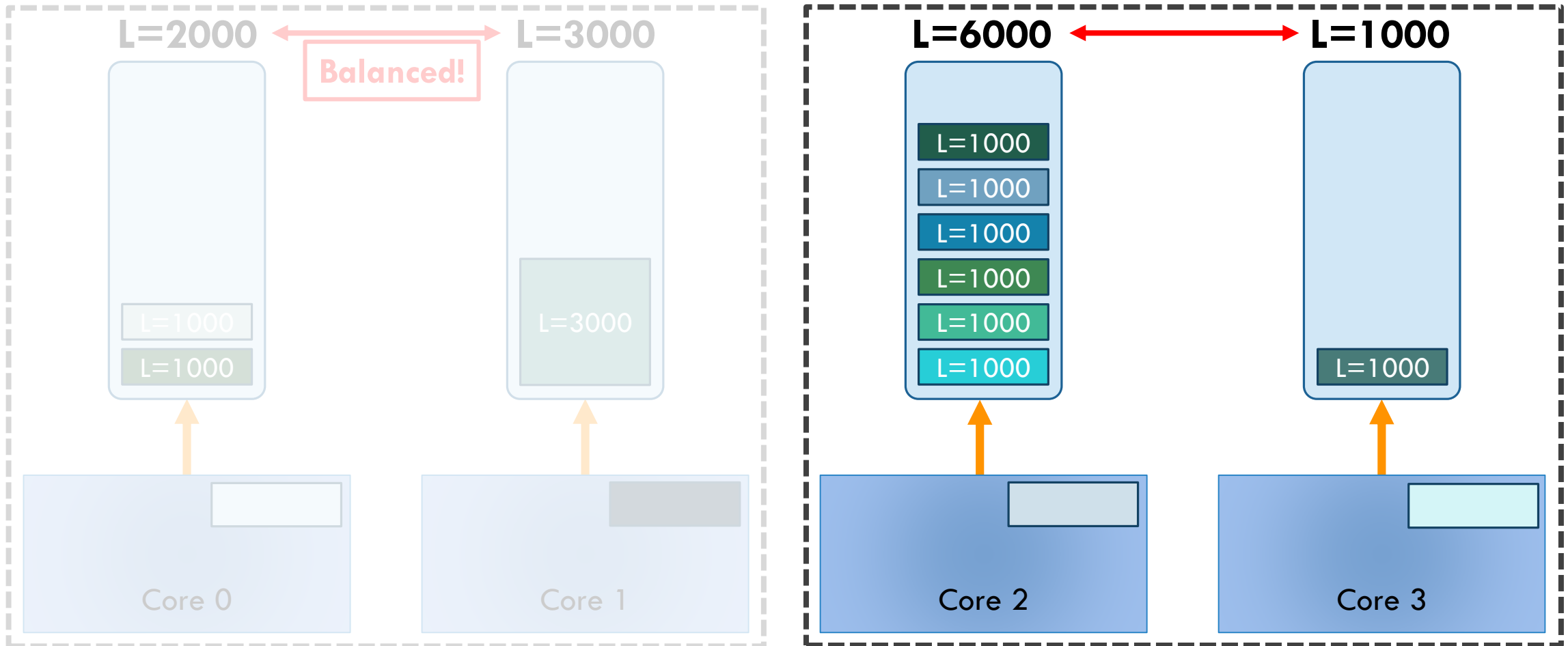


THE LINUX SCHEDULER: A DECADE OF WASTED CORES 8

# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

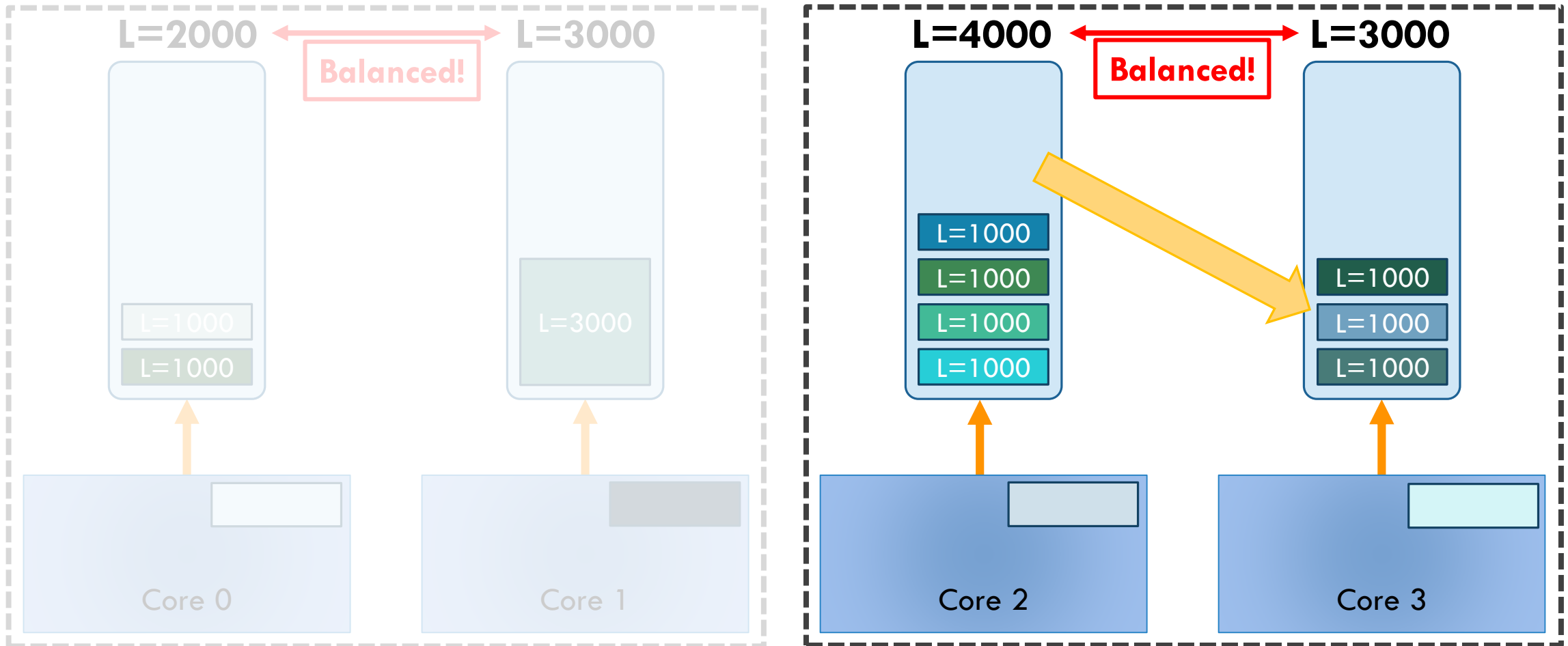


# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING



THE LINUX SCHEDULER: A DECADE OF WASTED CORES 8

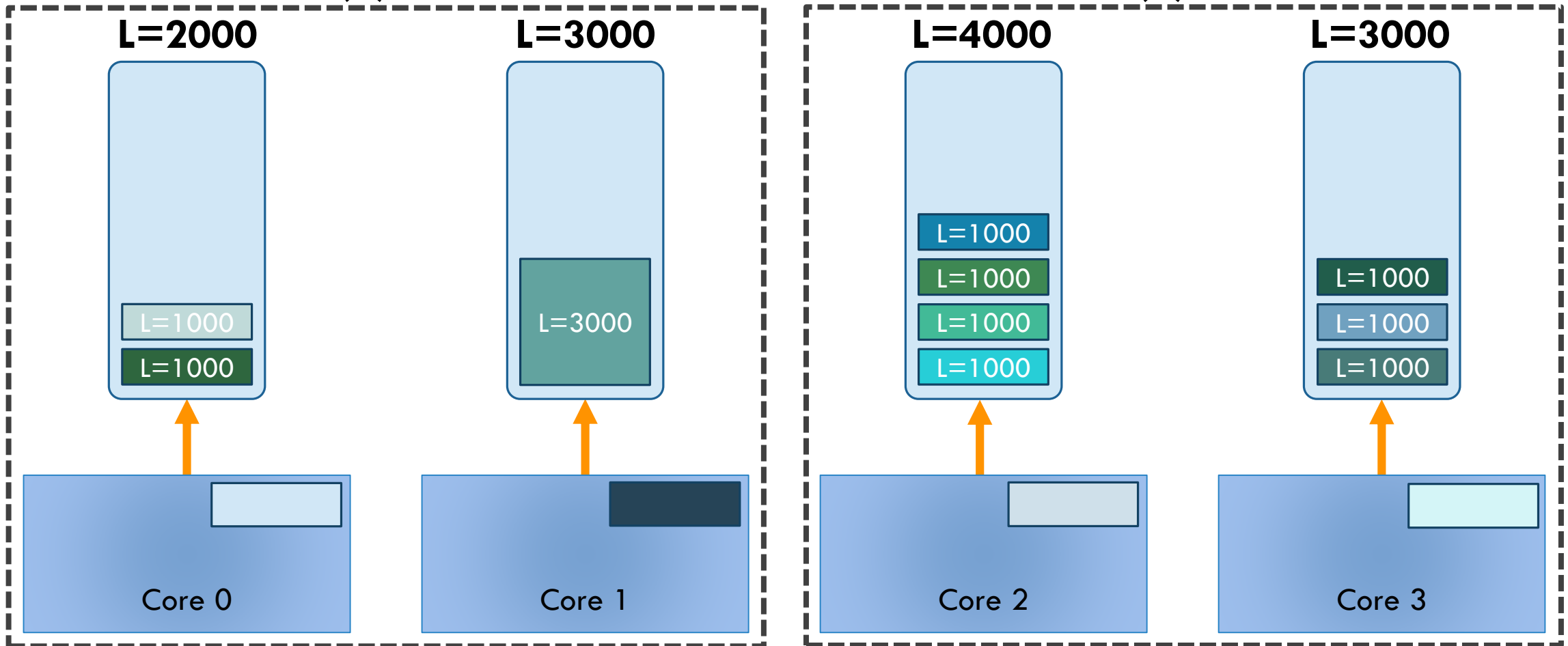
# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING



# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

AVG(L)=2500

AVG(L)=3500

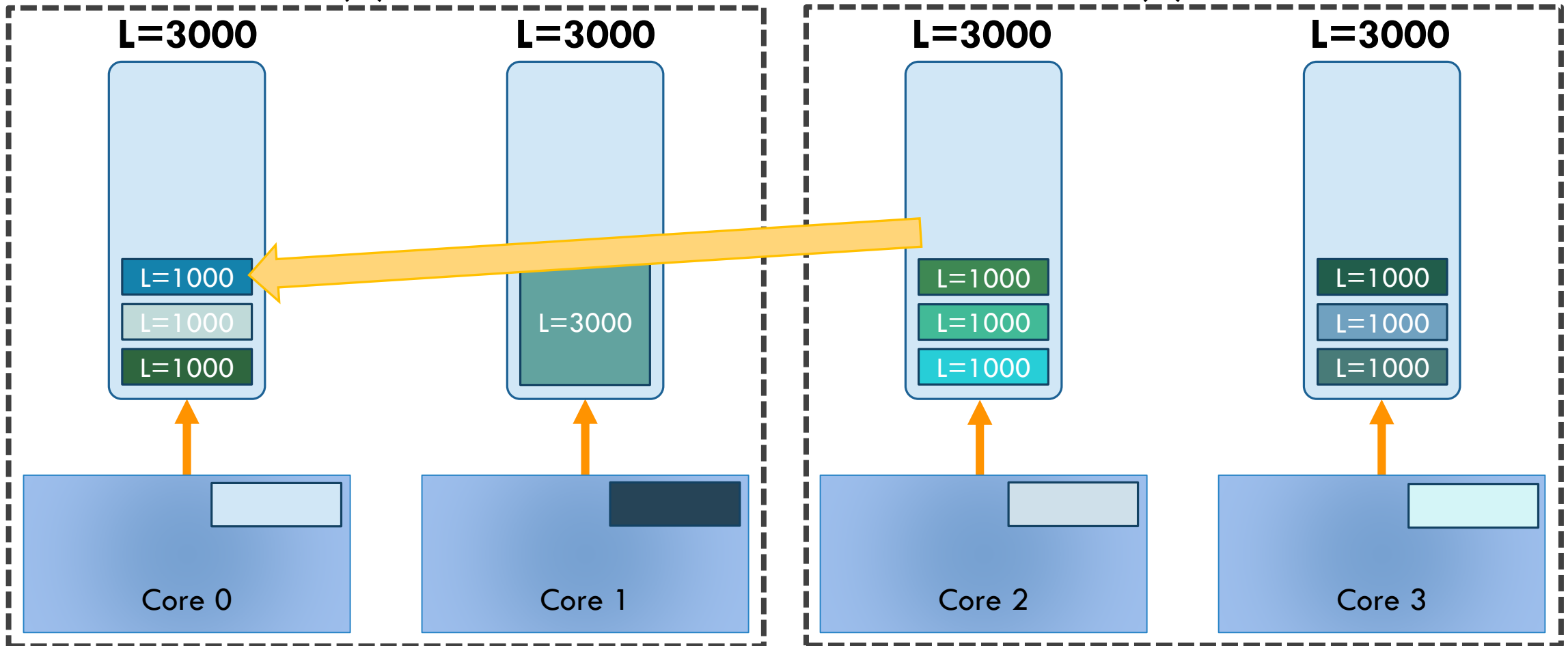




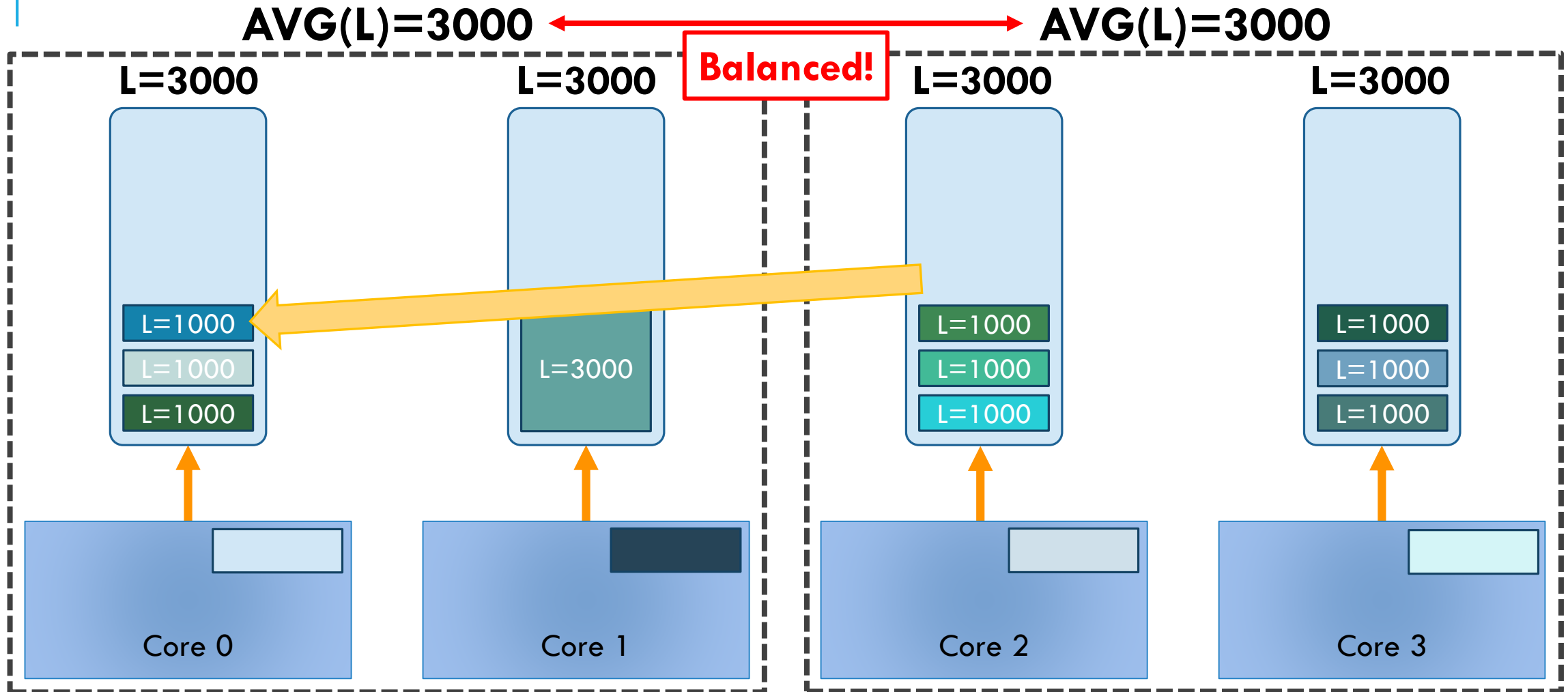
# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

AVG(L)=3000

AVG(L)=3000



# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING



# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

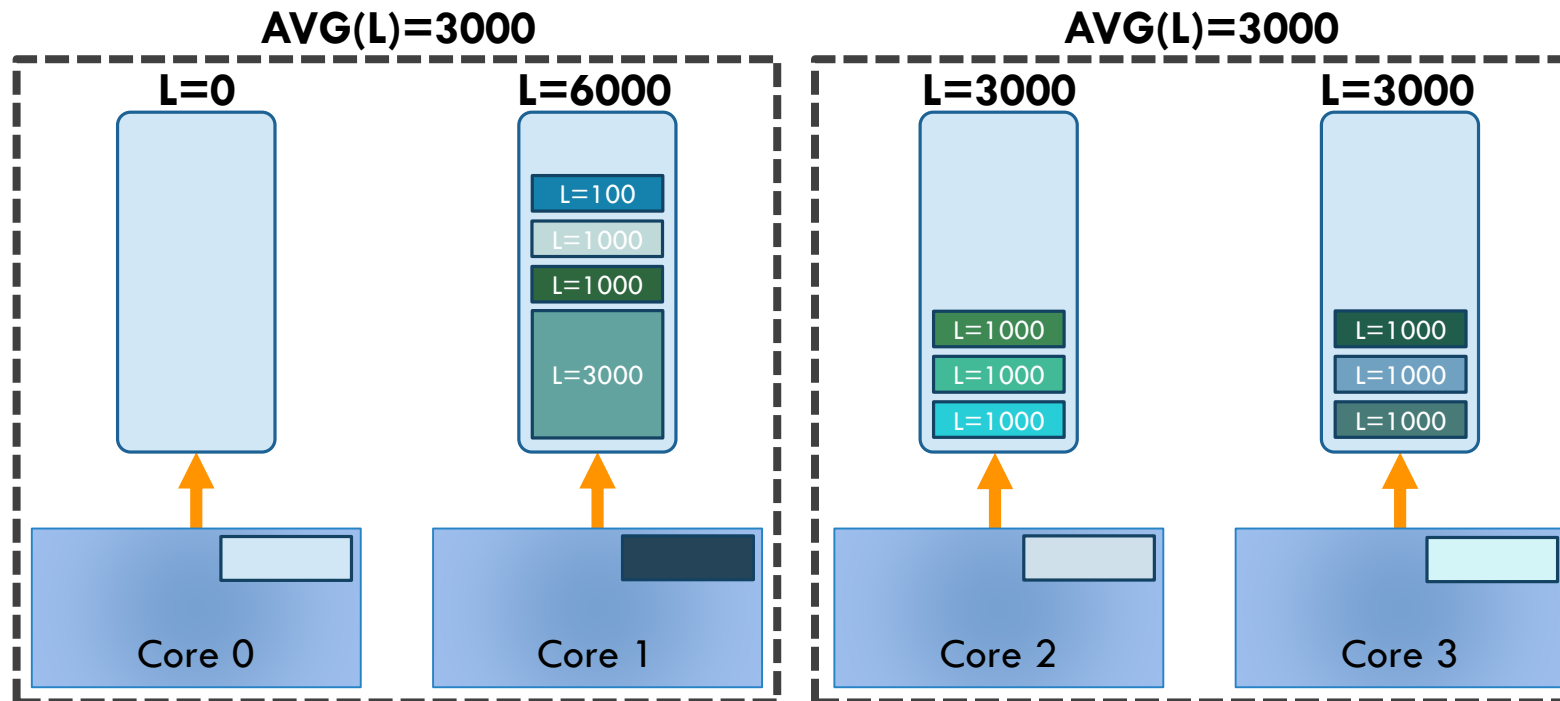
- Note that only the *average* load of groups is considered

# 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:

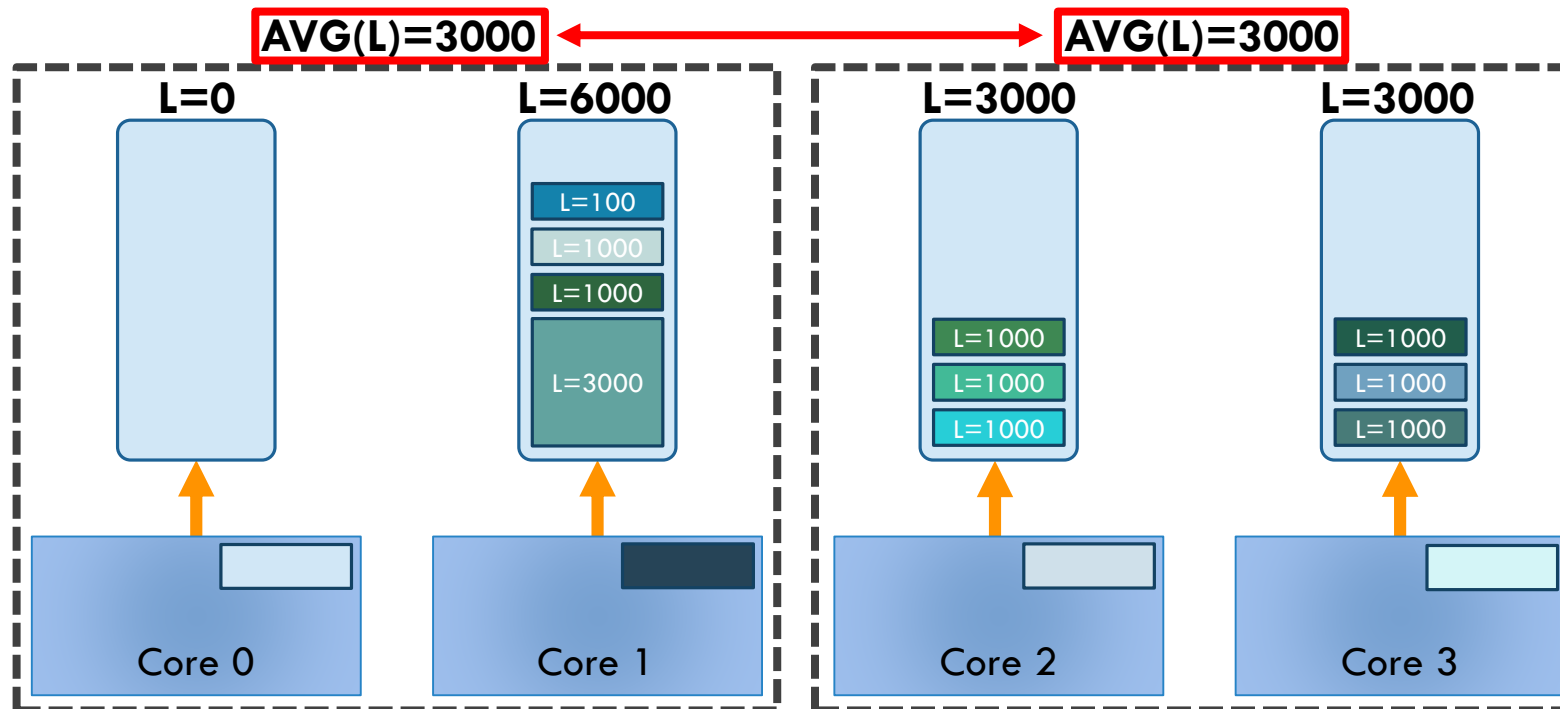
# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

- 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:



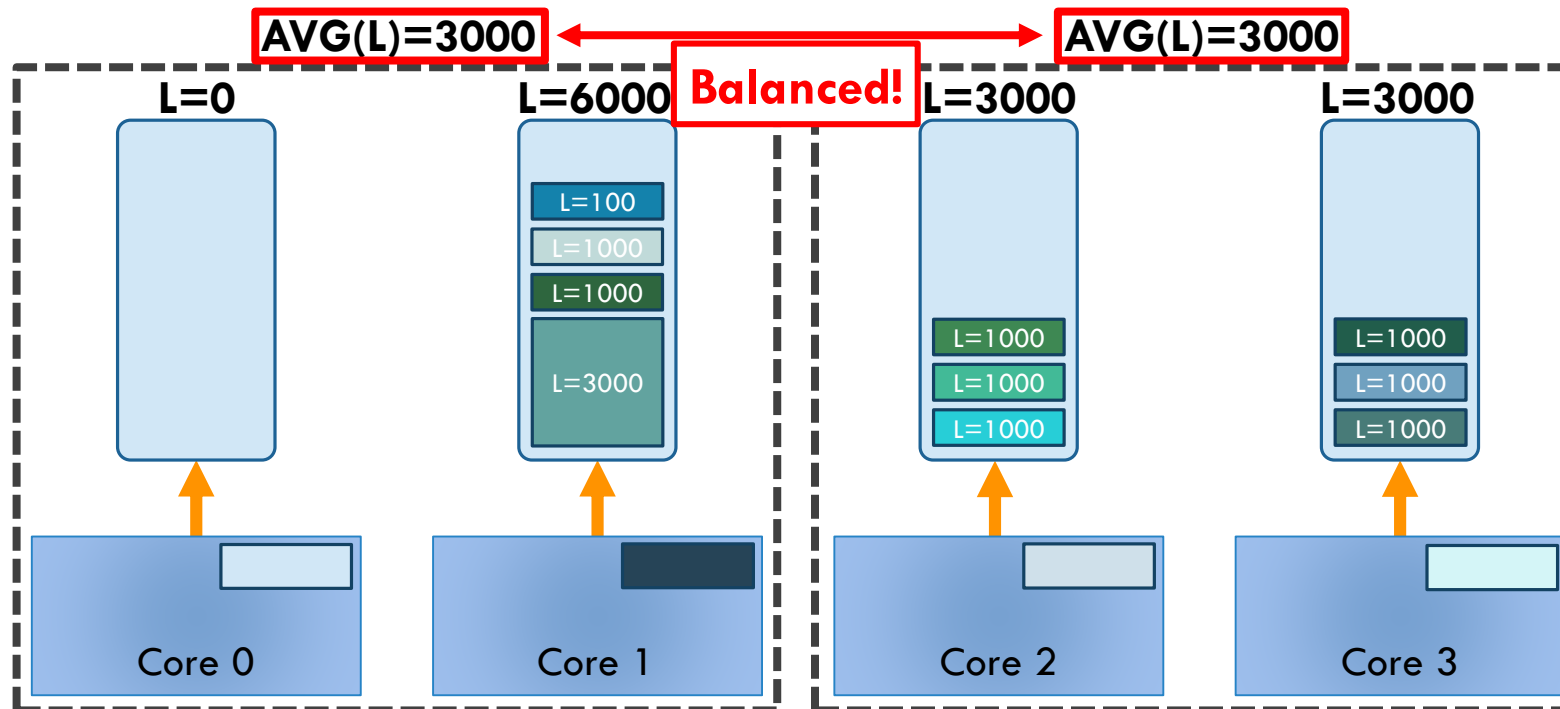
# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

- 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:



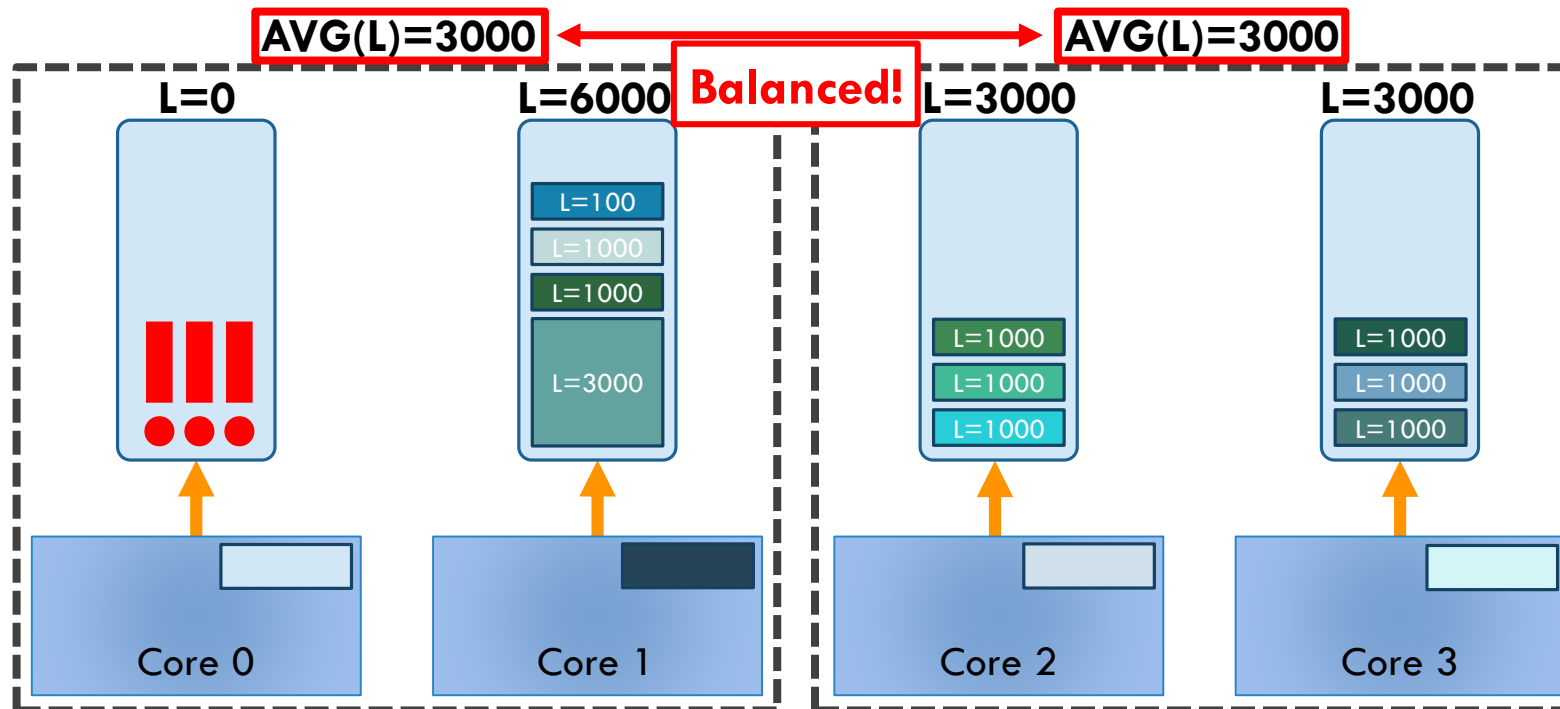
# CFS IN PRACTICE : HIERARCHICAL LOAD BALANCING

- 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 : 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.**

# 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.
- **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.

# 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”.**

L=1000

Session (tty) 1

L=1000

L=1000

L=1000

L=1000

Session (tty) 2

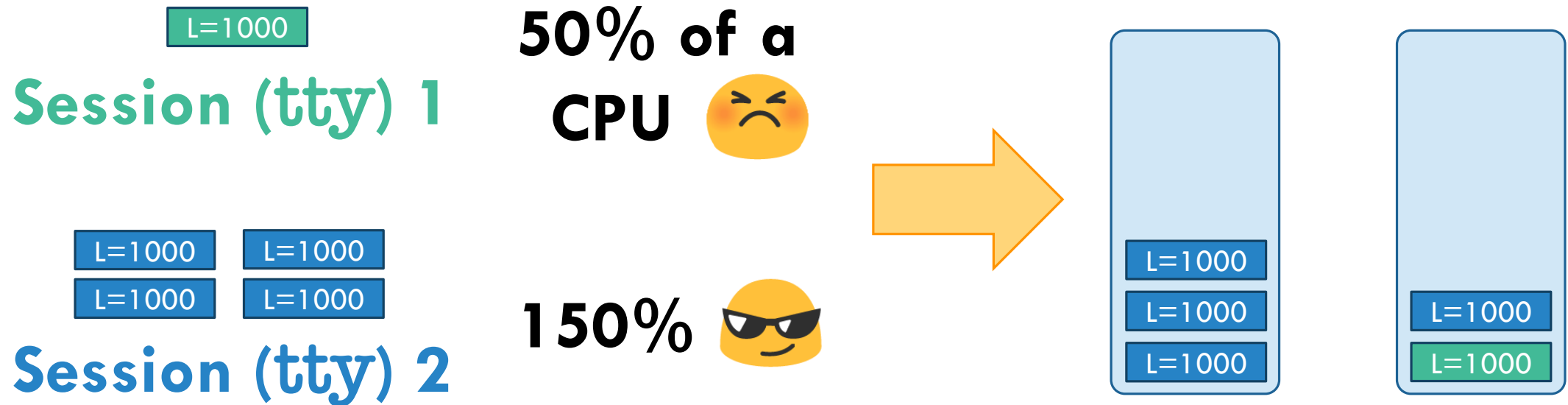
# CFS IN PRACTICE: MORE HEURISTICS

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



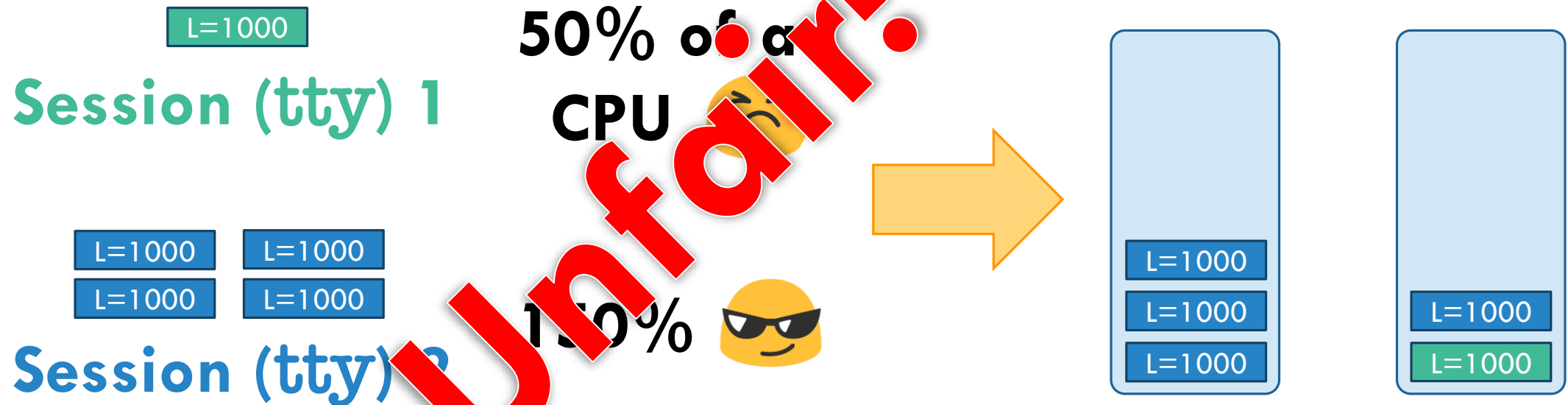
# CFS IN PRACTICE: MORE HEURISTICS

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



# 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...

# 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...

L=1000

Session (tty) 1

L=250

L=250

L=250

L=250

Session (tty) 2

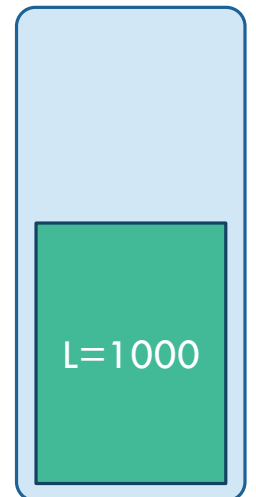
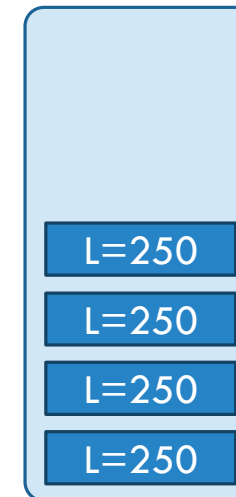
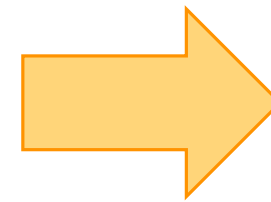
# CFS IN PRACTICE: MORE HEURISTICS

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

L=1000  
Session (tty) 1

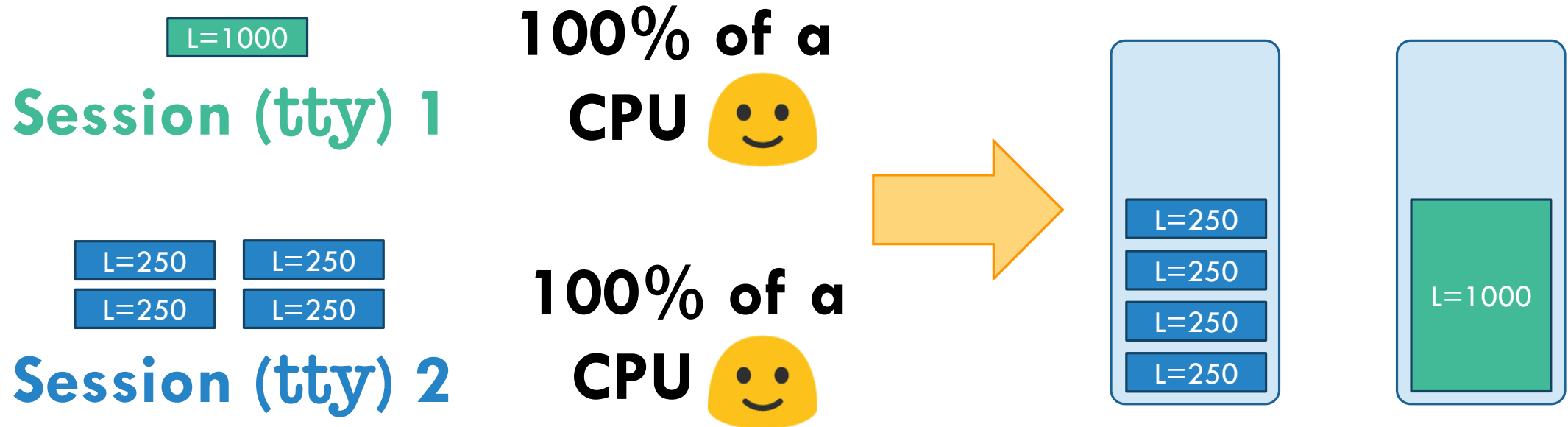
L=250 L=250  
L=250 L=250

Session (tty) 2



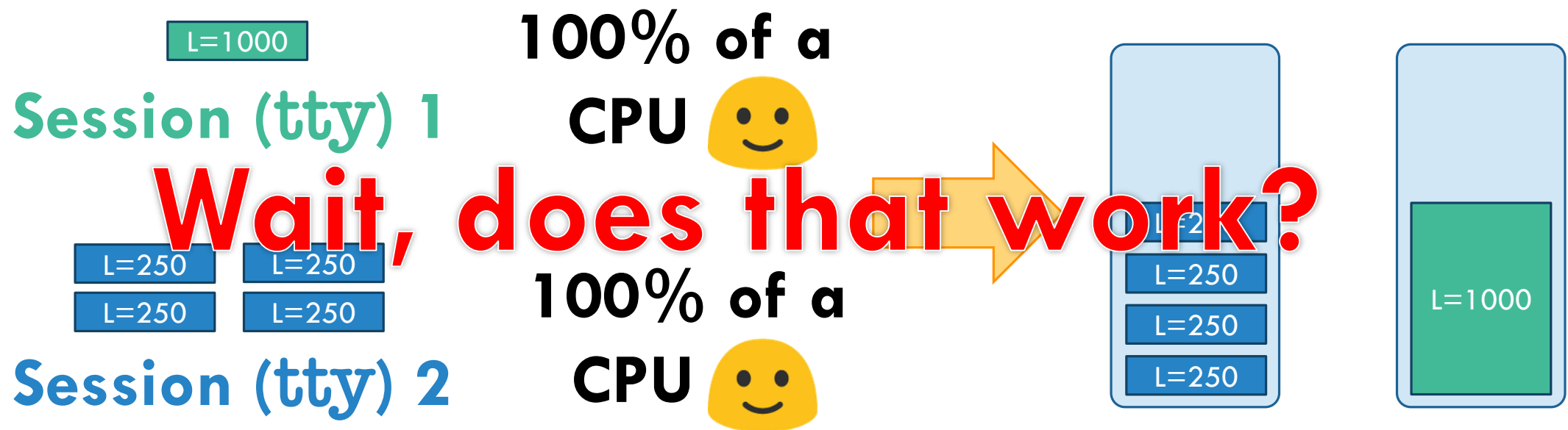
# CFS IN PRACTICE: MORE HEURISTICS

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



# CFS IN PRACTICE: MORE HEURISTICS

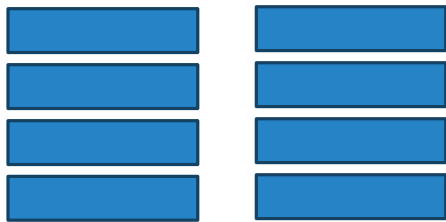
- Load calculations are actually more complicated, use more heuristics.
- **Solution:** divide the load of a task by the number of threads in its tty...



# BUG 1/4: GROUP IMBALANCE



Session (tty) 1



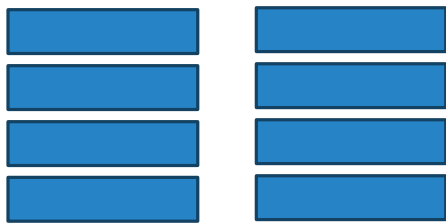
Session (tty) 2

# BUG 1/4: GROUP IMBALANCE

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



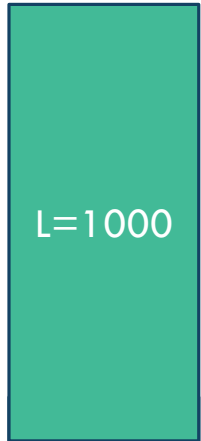
Session (tty) 1



Session (tty) 2

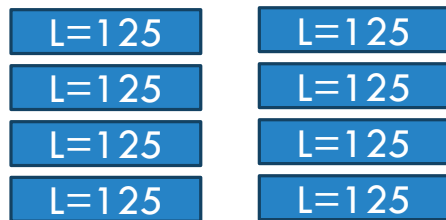
$$\begin{aligned}\text{Load(thread)} &= \%cpu \times \text{weight} / \#threads \\ &= 100 \times 10 / 8 \\ &= 125\end{aligned}$$

# BUG 1/4: GROUP IMBALANCE



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

## Session (tty) 1

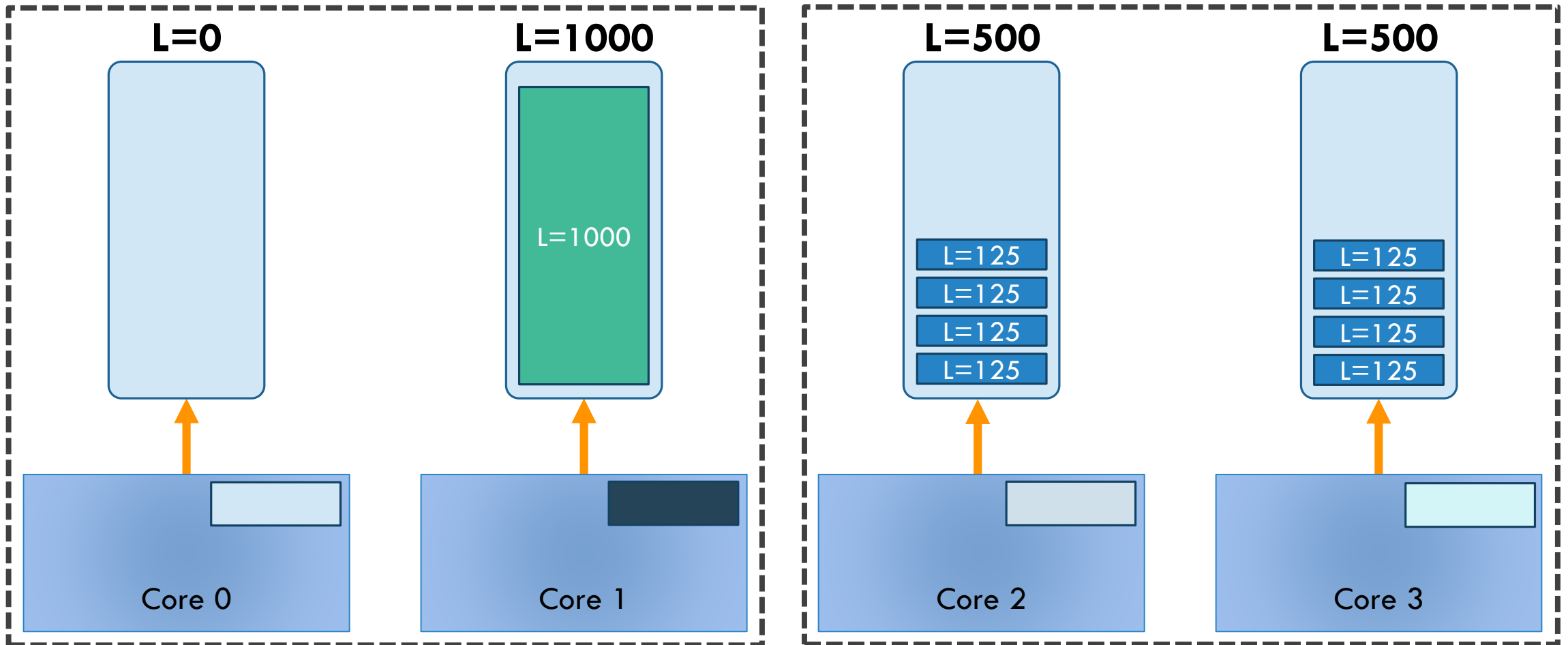


$$\begin{aligned}\text{Load(thread)} &= \%cpu \times \text{weight} / \#threads \\ &= 100 \times 10 / 8 \\ &= 125\end{aligned}$$

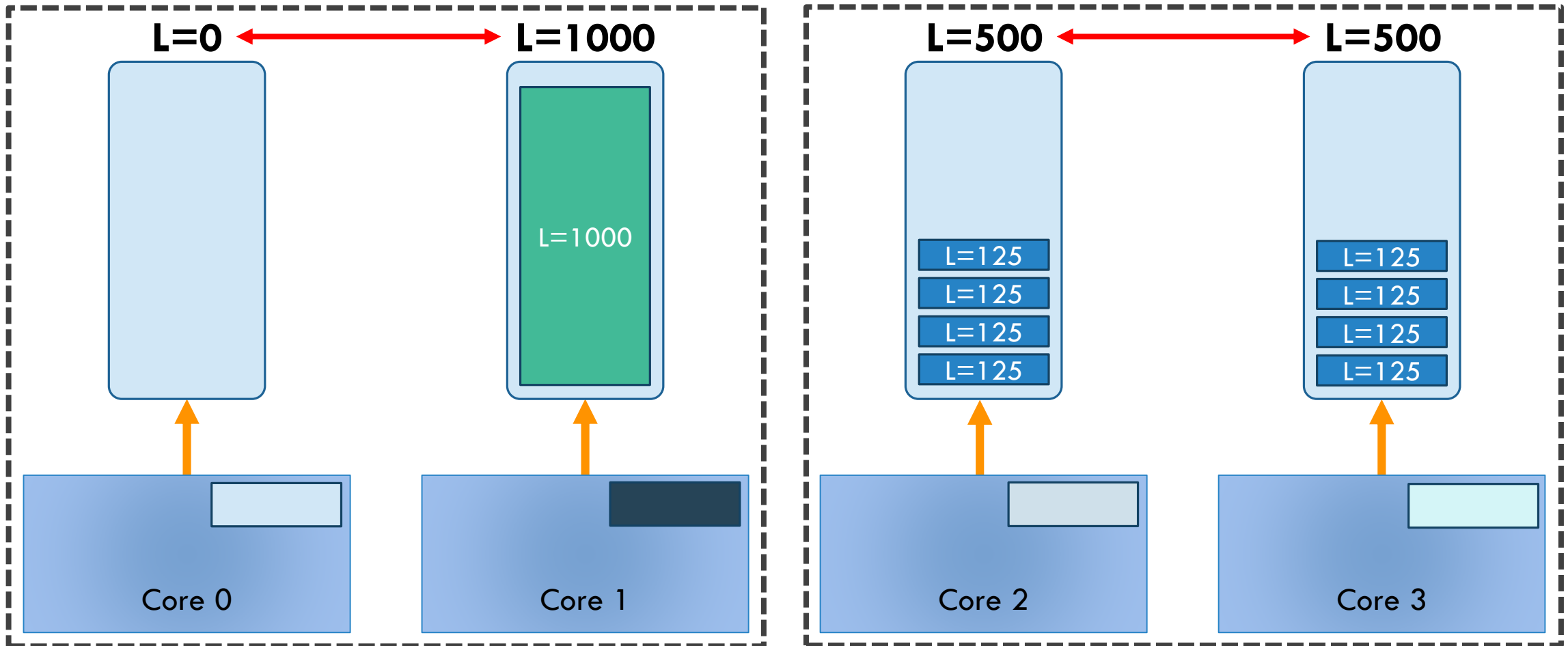
## Session (tty) 2



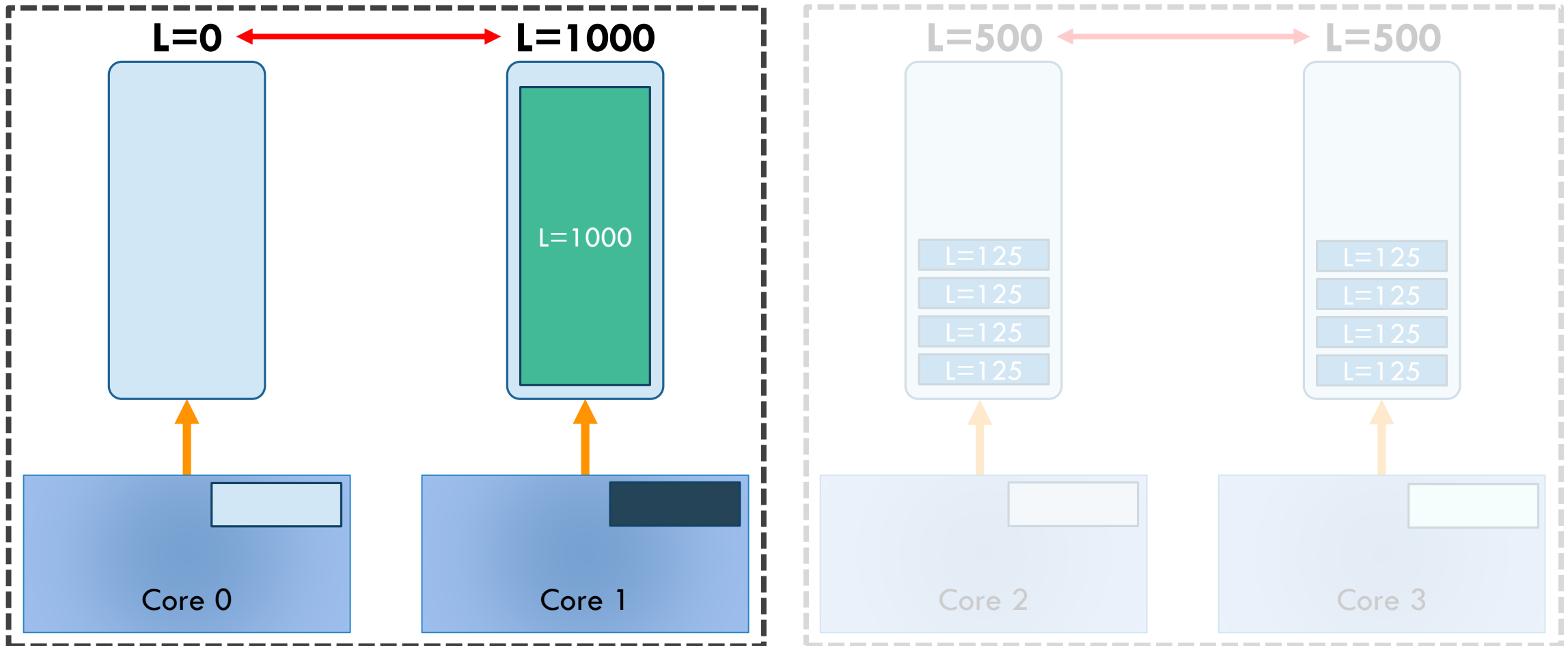
# BUG 1/4: GROUP IMBALANCE



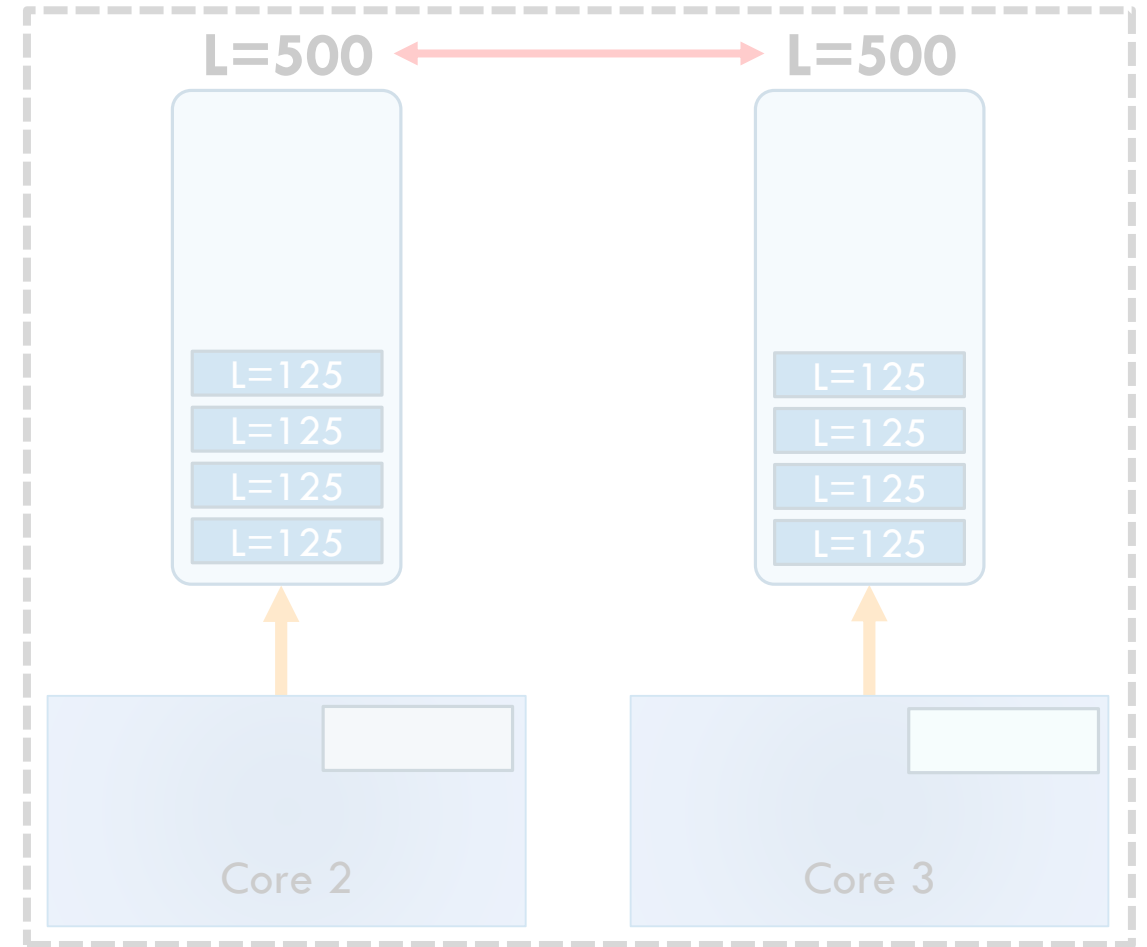
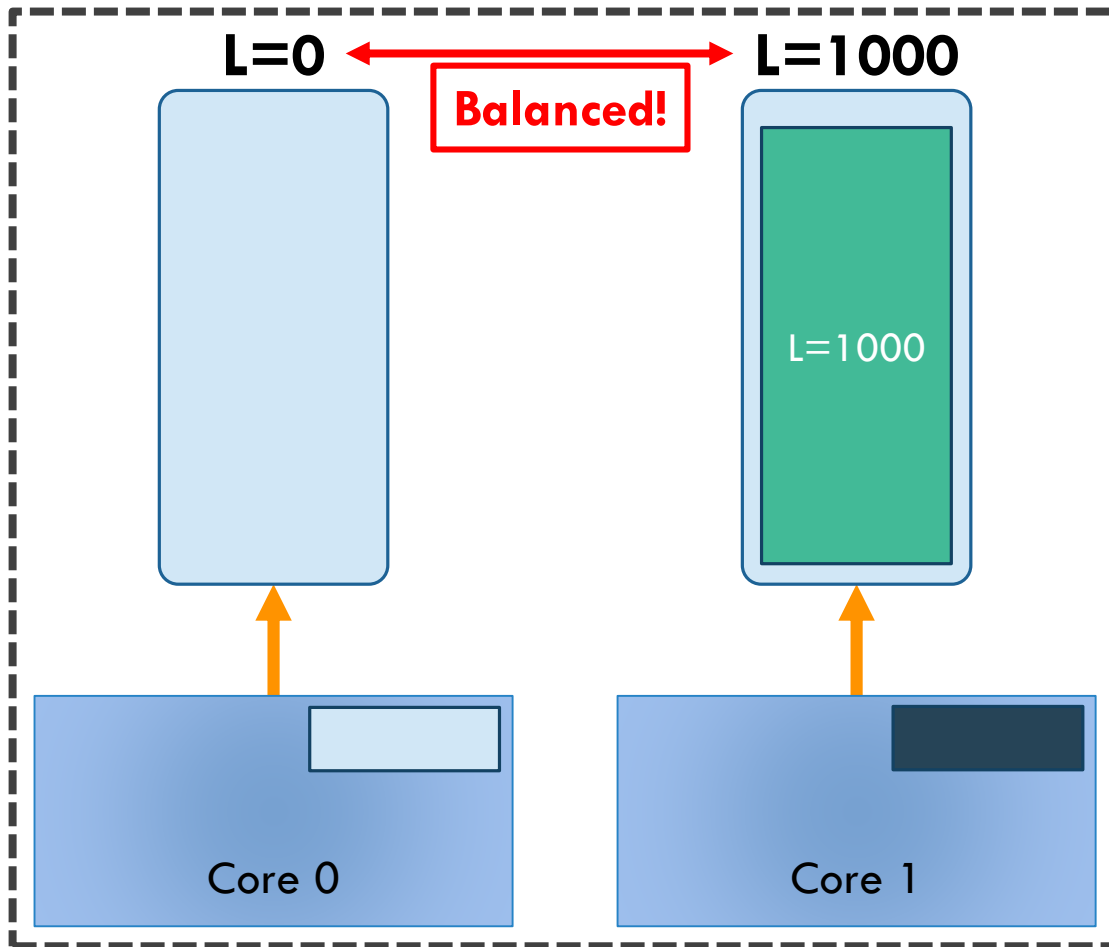
# BUG 1/4: GROUP IMBALANCE



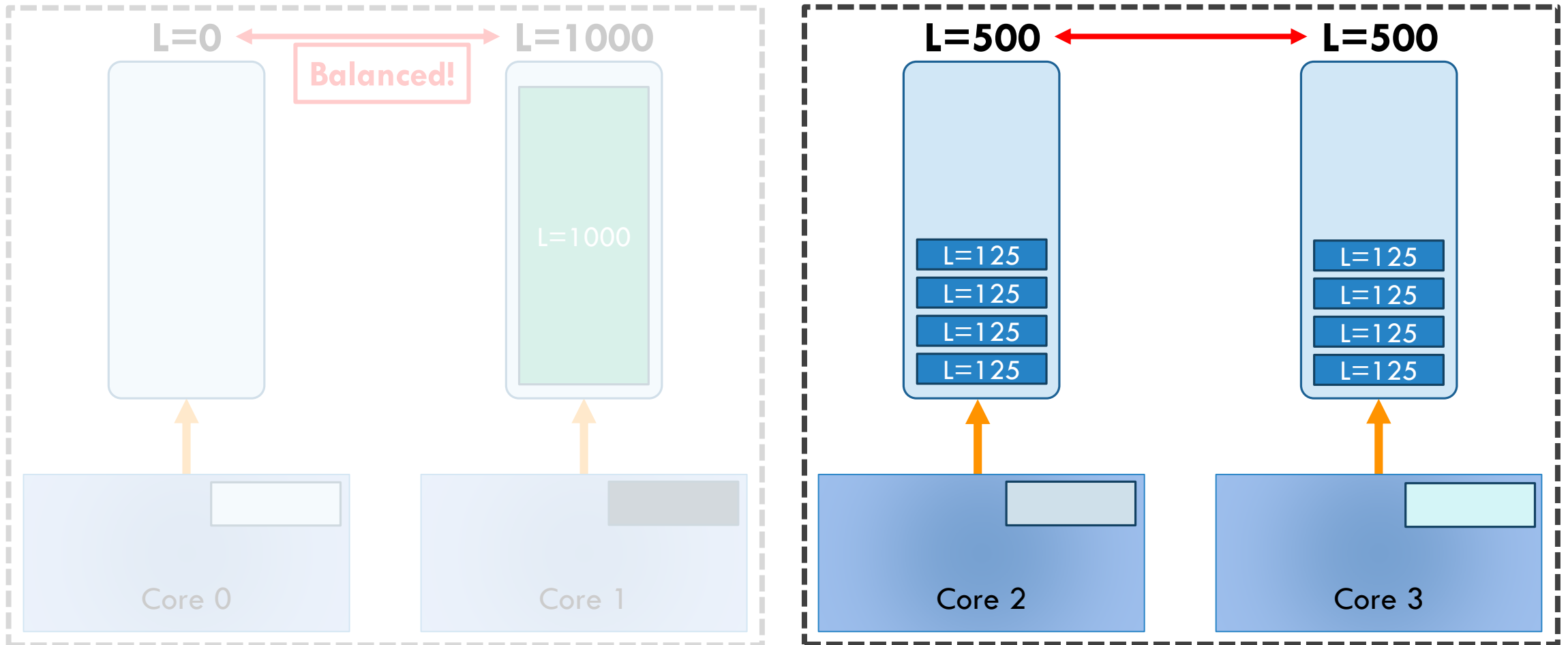
# BUG 1/4: GROUP IMBALANCE



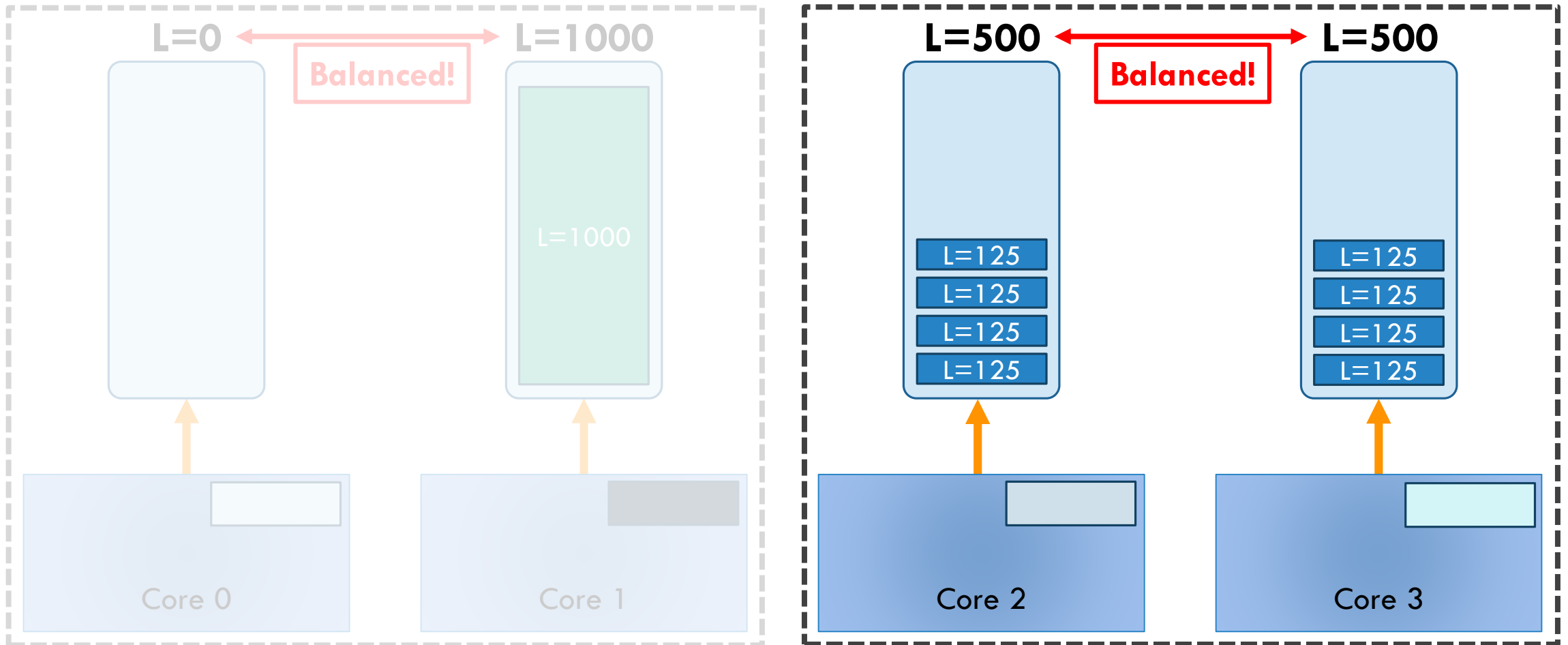
# BUG 1/4: GROUP IMBALANCE



# BUG 1/4: GROUP IMBALANCE



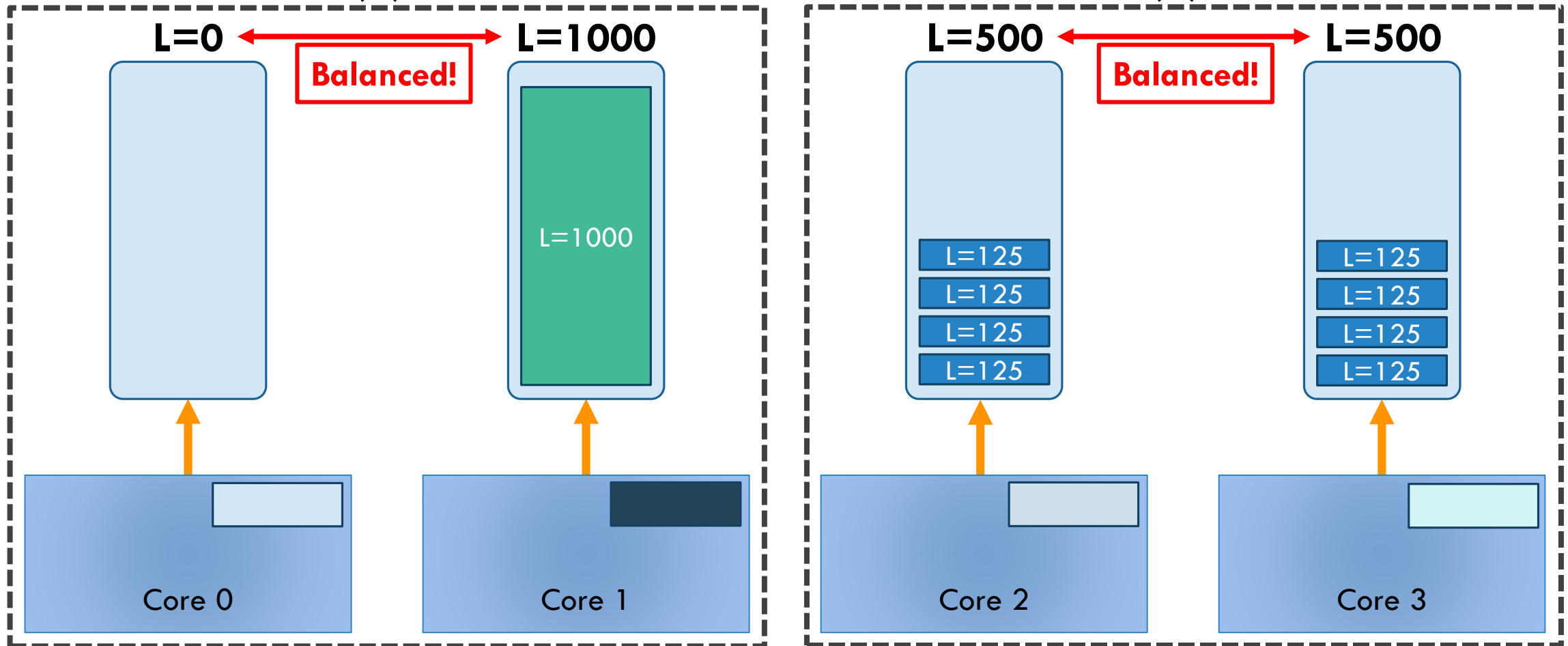
# BUG 1/4: GROUP IMBALANCE



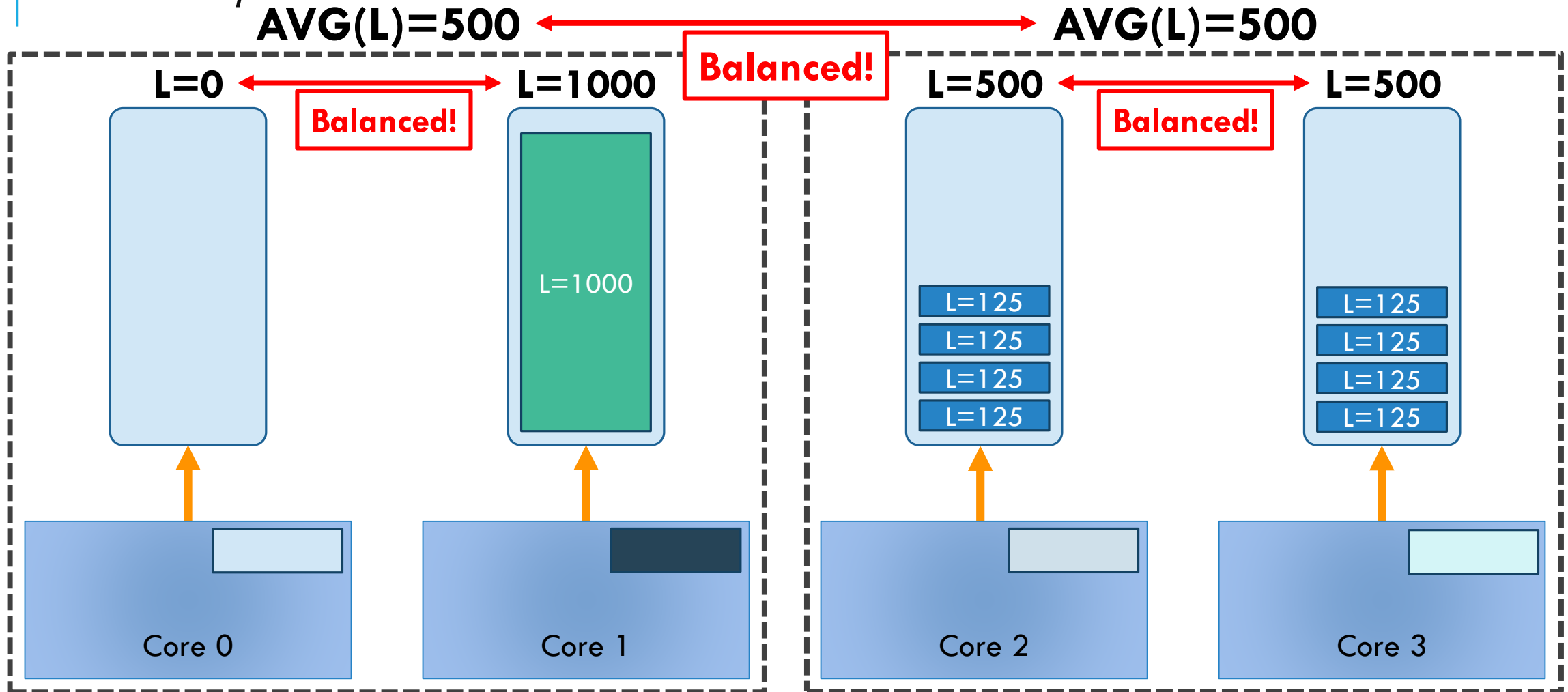
# BUG 1/4: GROUP IMBALANCE

$AVG(L)=500$

$AVG(L)=500$

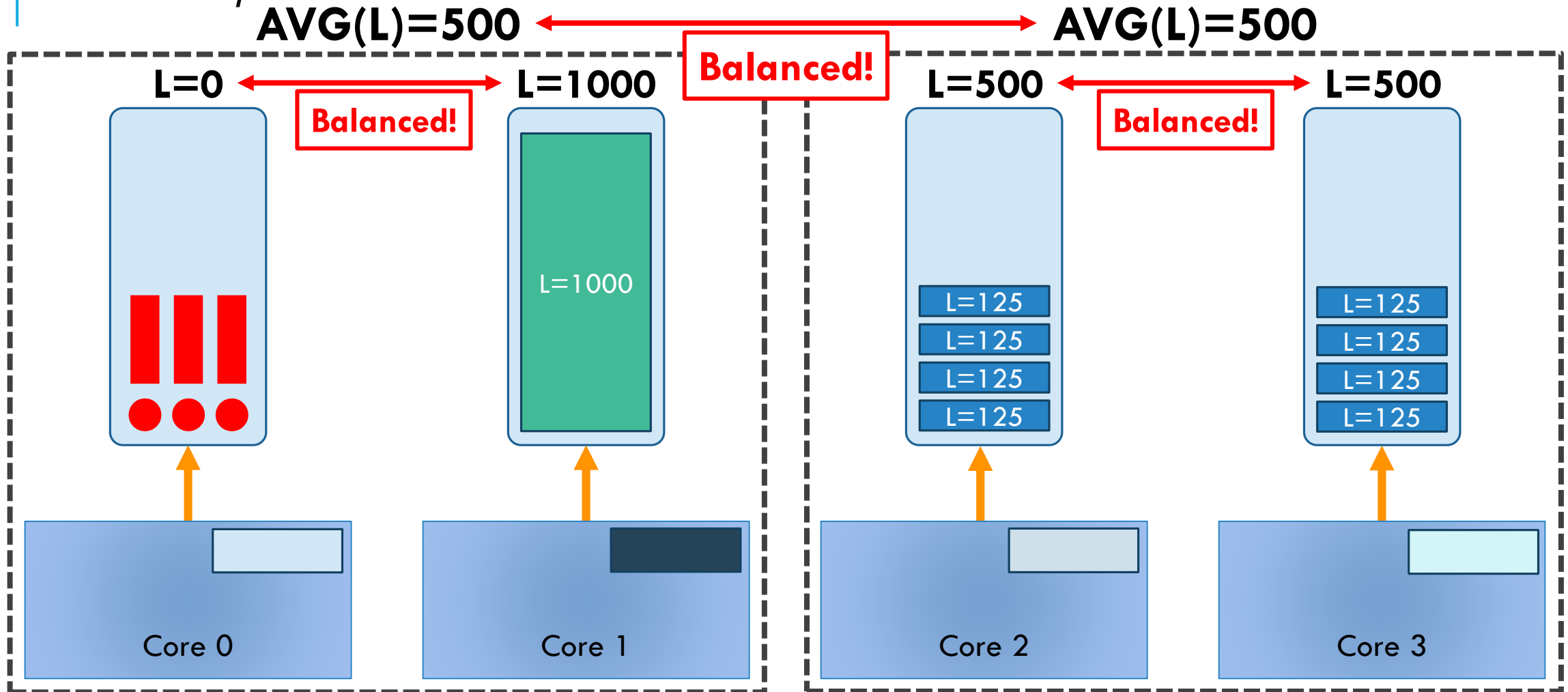


# BUG 1/4: GROUP IMBALANCE

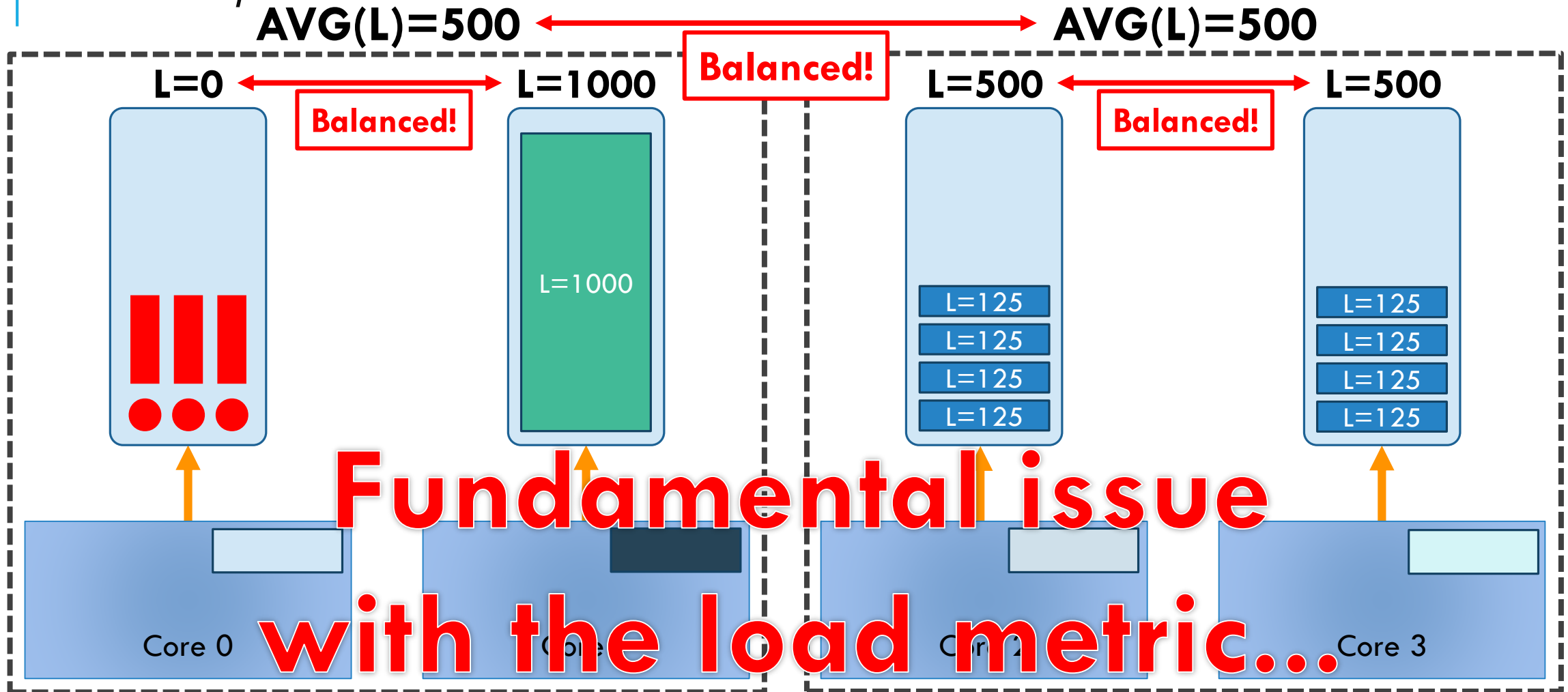




# BUG 1/4: GROUP IMBALANCE



# BUG 1/4: GROUP IMBALANCE



# BUG 1/4: GROUP IMBALANCE

- **Another example, on a 64-core machine, with load balancing:**
  - First between pairs of cores (Bulldozer architecture, a bit like hyperthreading)
  - Then between NUMA nodes

# BUG 1/4: GROUP IMBALANCE

- **Another example, on a 64-core machine, with load balancing:**
  - First between pairs of cores (Bulldozer architecture, a bit like hyperthreading)
  - Then between NUMA nodes
- **User 1 launches :**  
ssh <machine> R &  
ssh <machine> R &

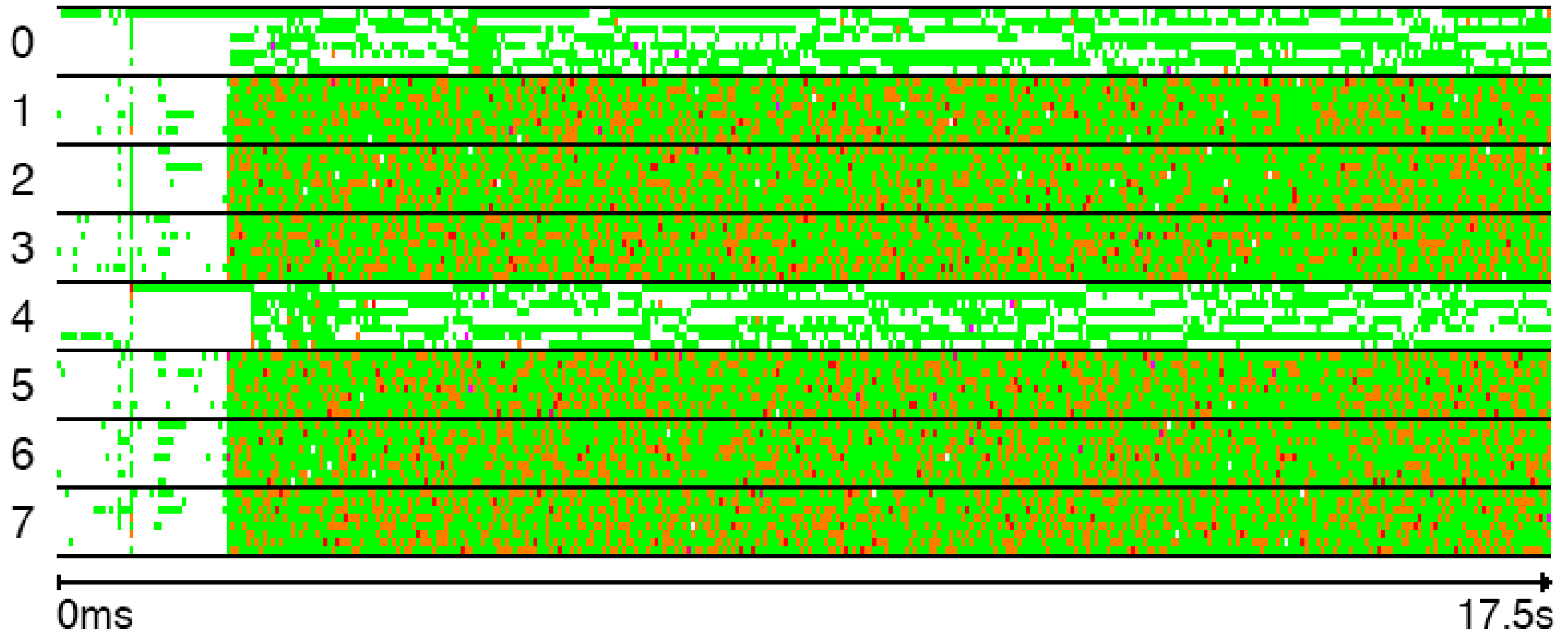
# BUG 1/4: GROUP IMBALANCE

- **Another example, on a 64-core machine, with load balancing:**
  - First between pairs of cores (Bulldozer architecture, a bit like hyperthreading)
  - Then between NUMA nodes
- **User 1 launches :**  
ssh <machine> R &  
ssh <machine> R &
- **User 2 launches :**  
ssh <machine> make -j 64 kernel

# BUG 1/4: GROUP IMBALANCE

- **Another example, on a 64-core machine, with load balancing:**
  - First between pairs of cores (Bulldozer architecture, a bit like hyperthreading)
  - Then between NUMA nodes
- **User 1 launches :**  
ssh <machine> R &  
ssh <machine> R &
- **User 2 launches :**  
ssh <machine> make -j 64 kernel
- **The bug happens at two levels :**
  - Other core on pair of core idle
  - Other cores on NUMA node less busy...

Number of threads in run queue: 0 1 2 3 4+



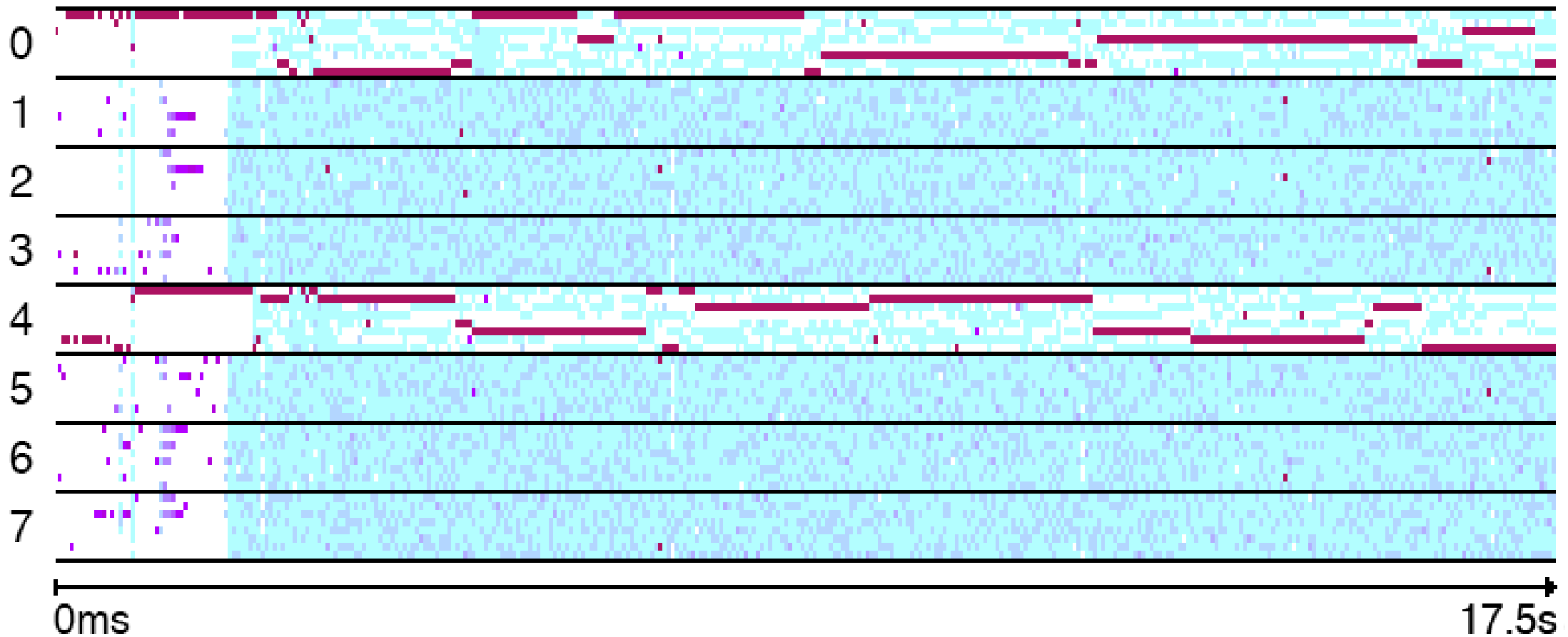
- **The bug happens at two levels :**
  - Other core on pair of core idle
  - Other cores on NUMA node less busy...

Load:

0

1

1024

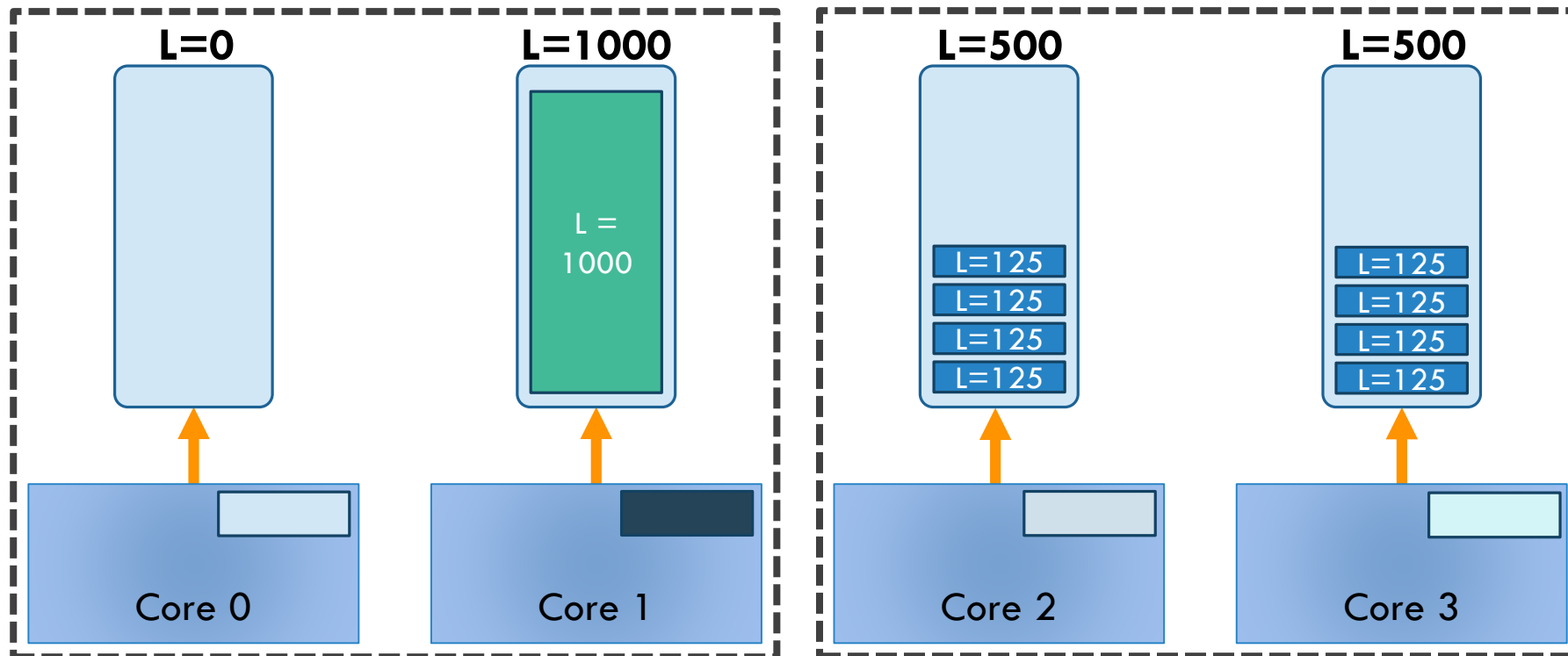


- **The bug happens at two levels :**
  - Other core on pair of core idle
  - Other cores on NUMA node less busy...



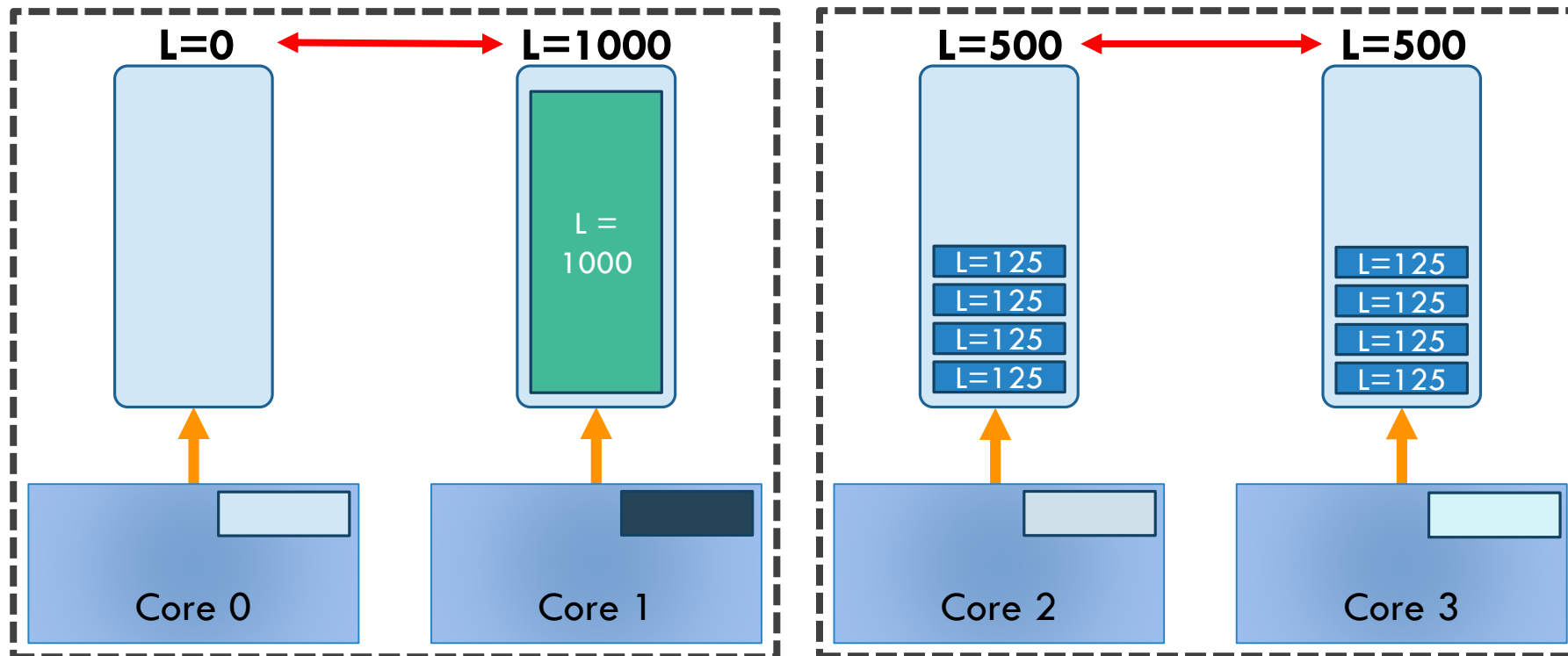
# BUG 1/4: GROUP IMBALANCE

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



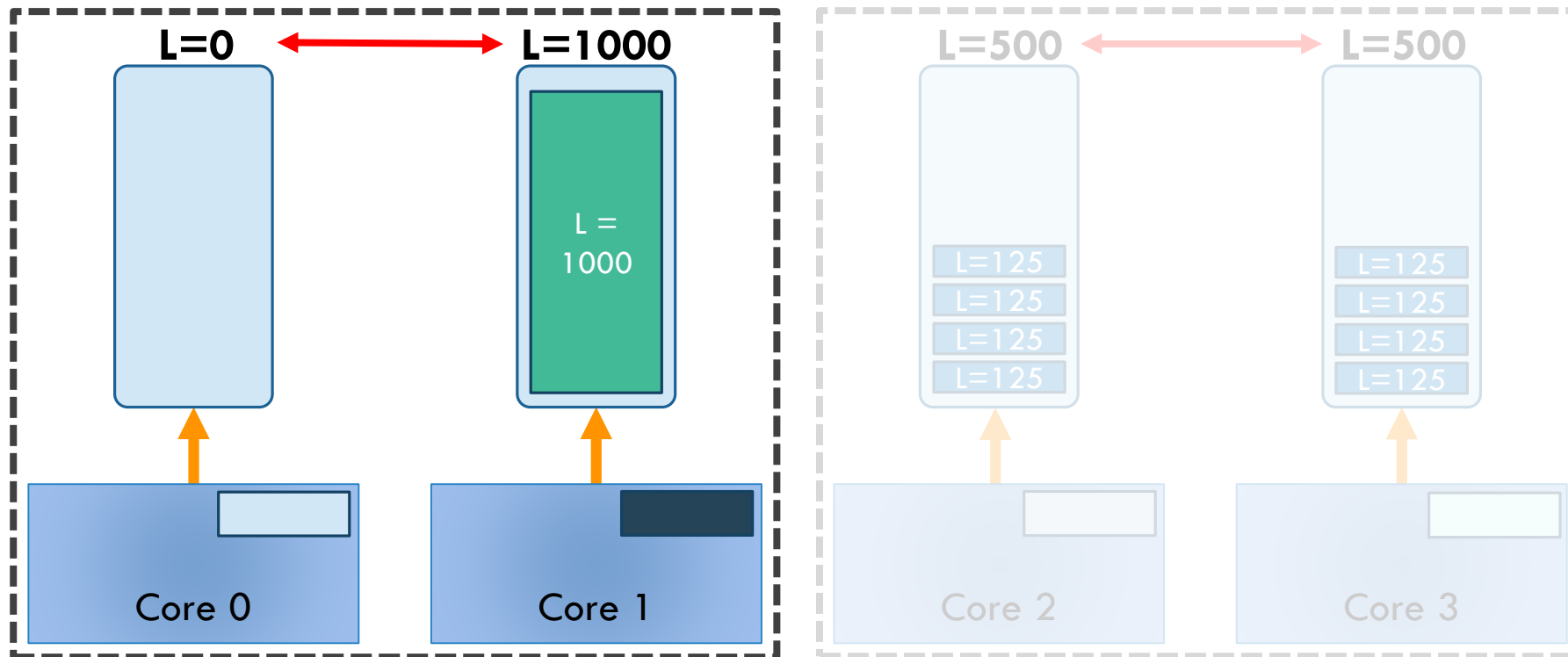
# BUG 1/4: GROUP IMBALANCE

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



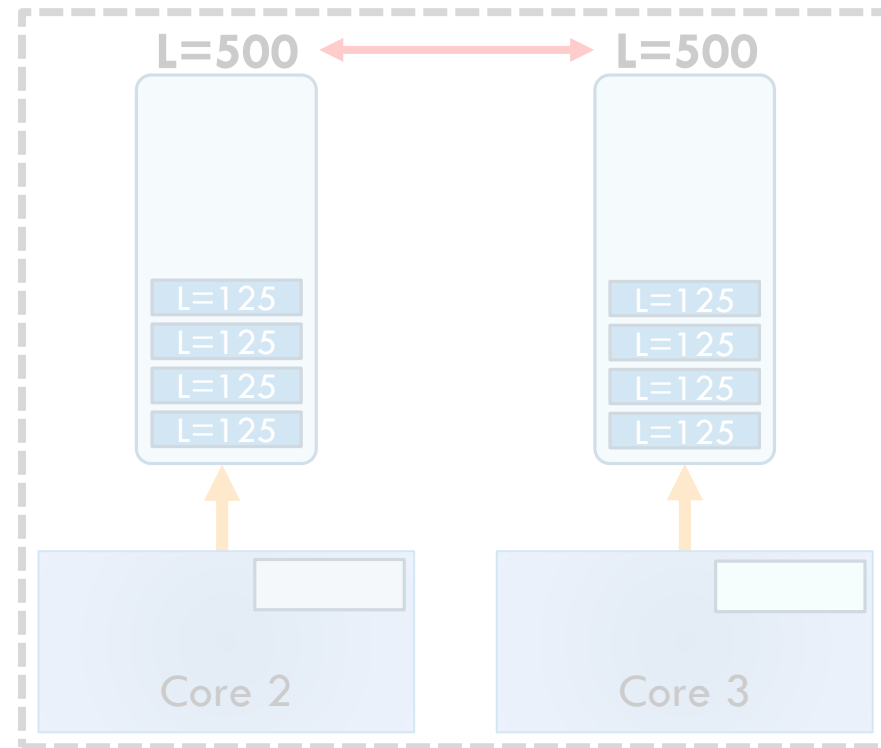
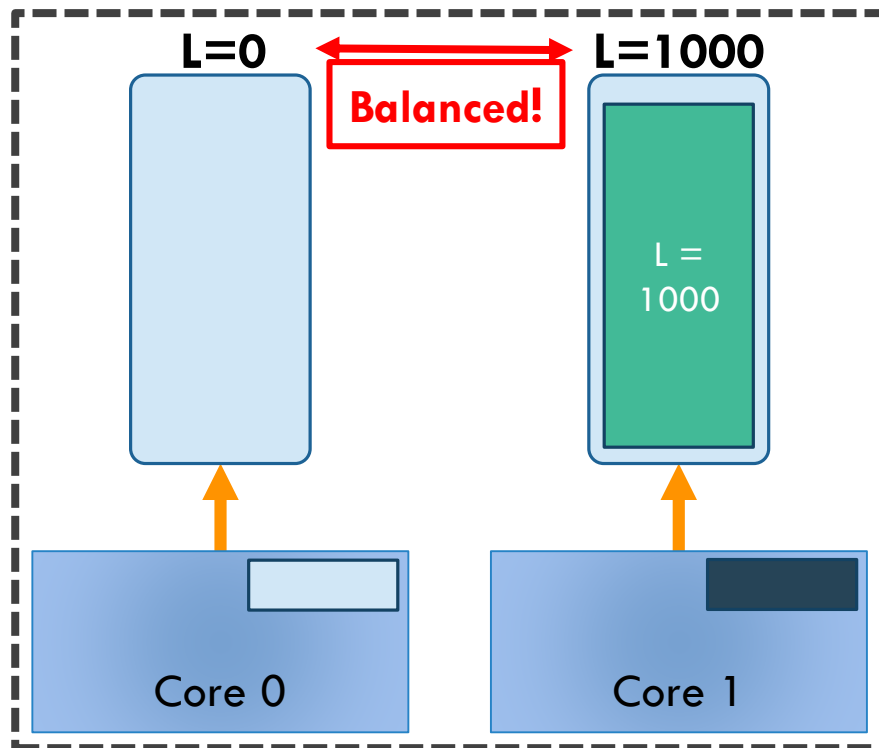
# BUG 1/4: GROUP IMBALANCE

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



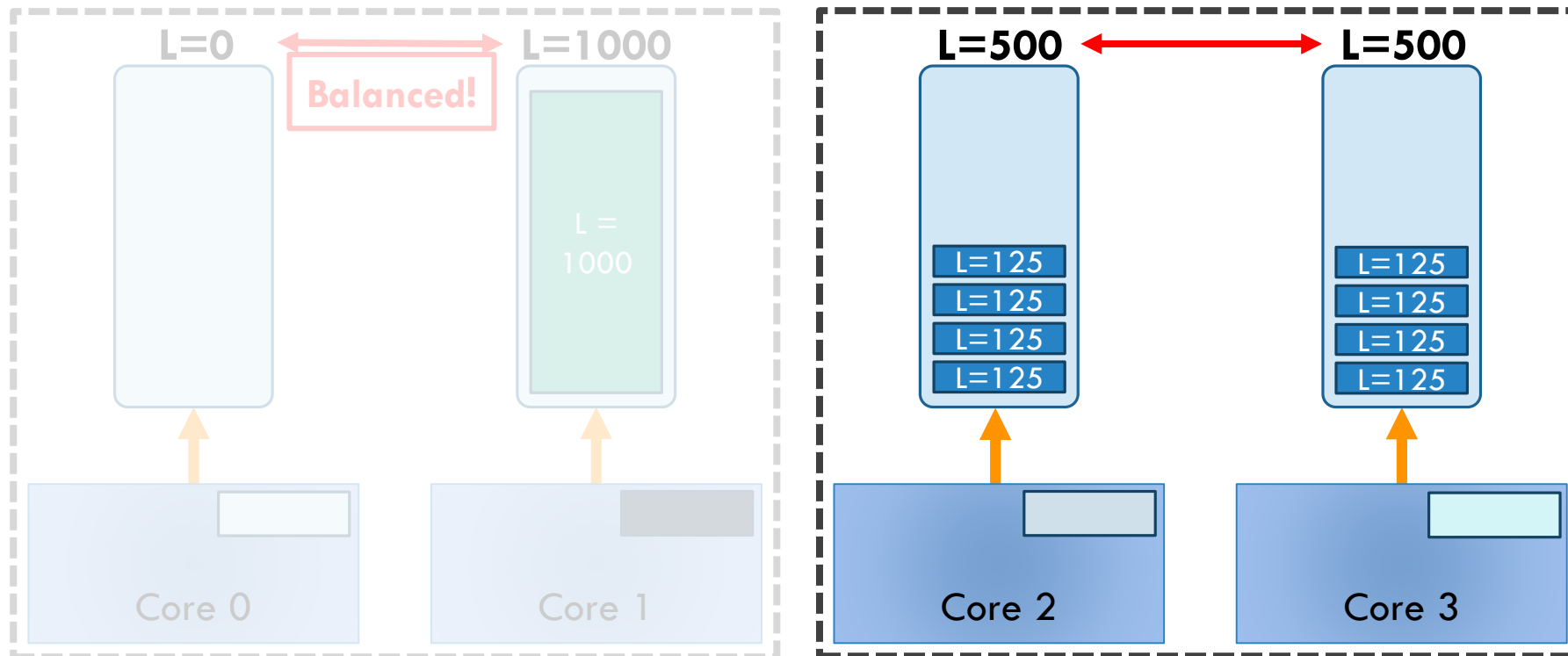
# BUG 1/4: GROUP IMBALANCE

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



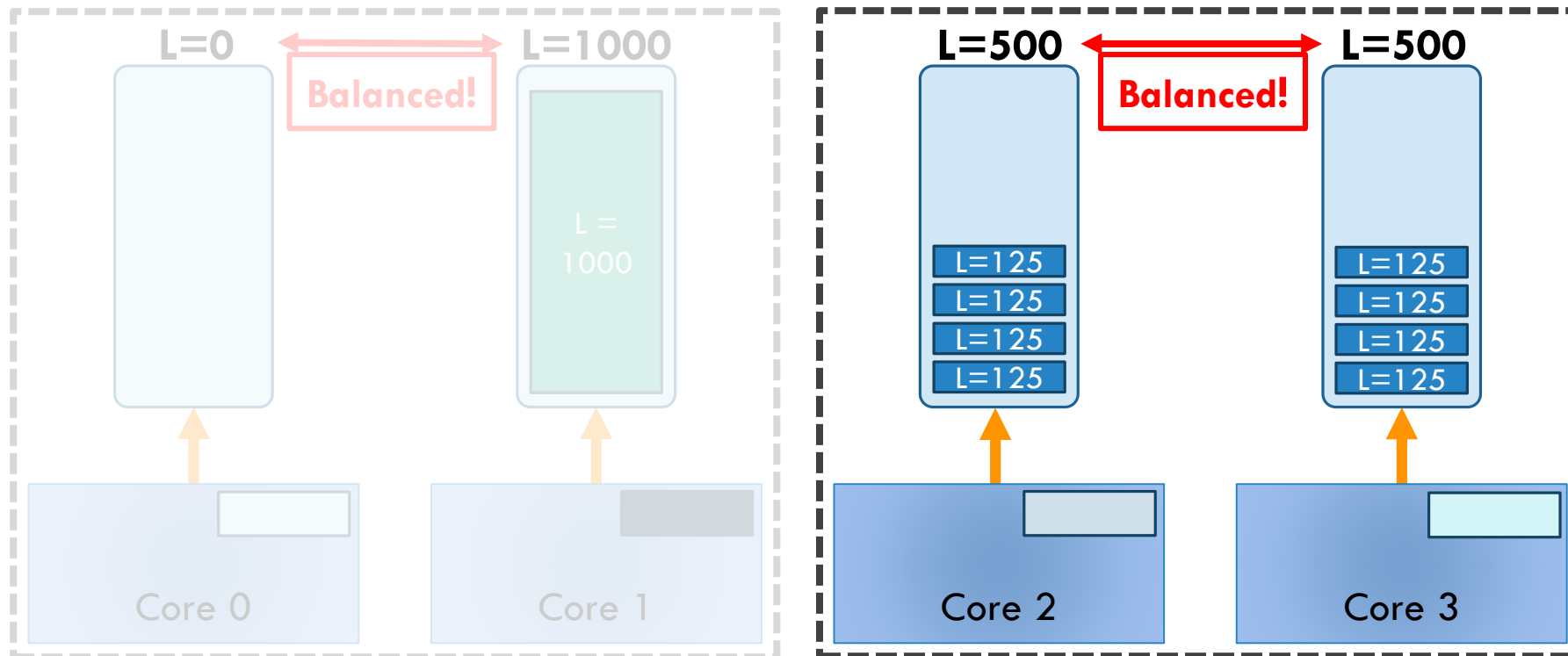
# BUG 1/4: GROUP IMBALANCE

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



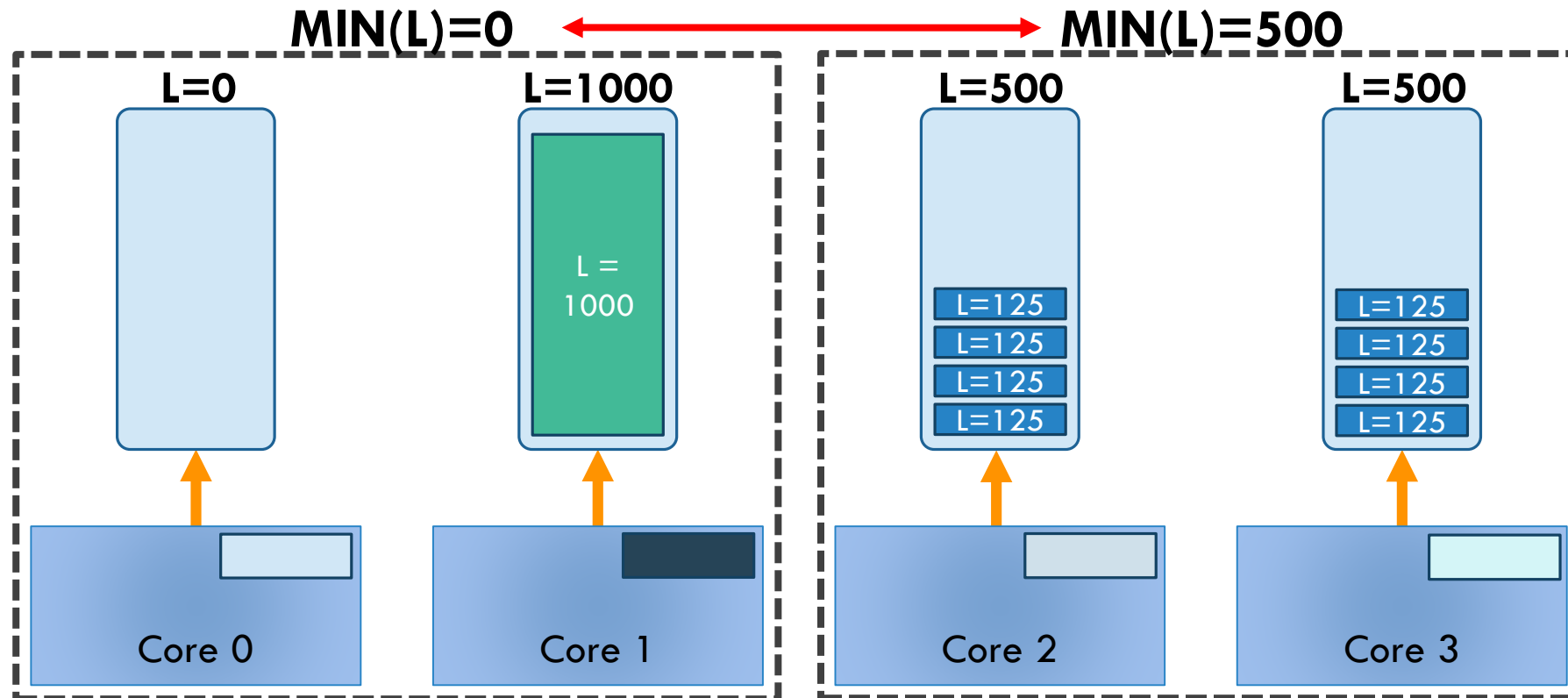
# BUG 1/4: GROUP IMBALANCE

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



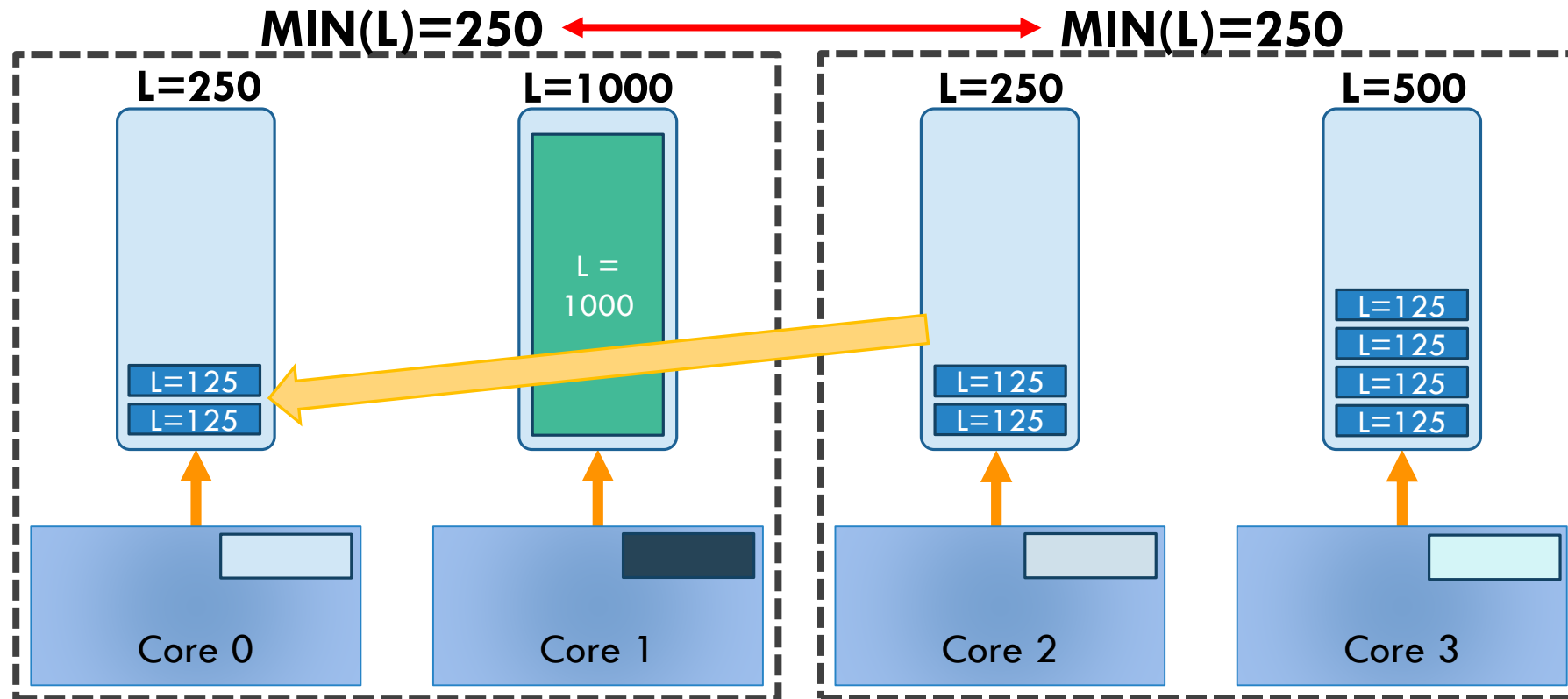
# BUG 1/4: GROUP IMBALANCE

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



# BUG 1/4: GROUP IMBALANCE

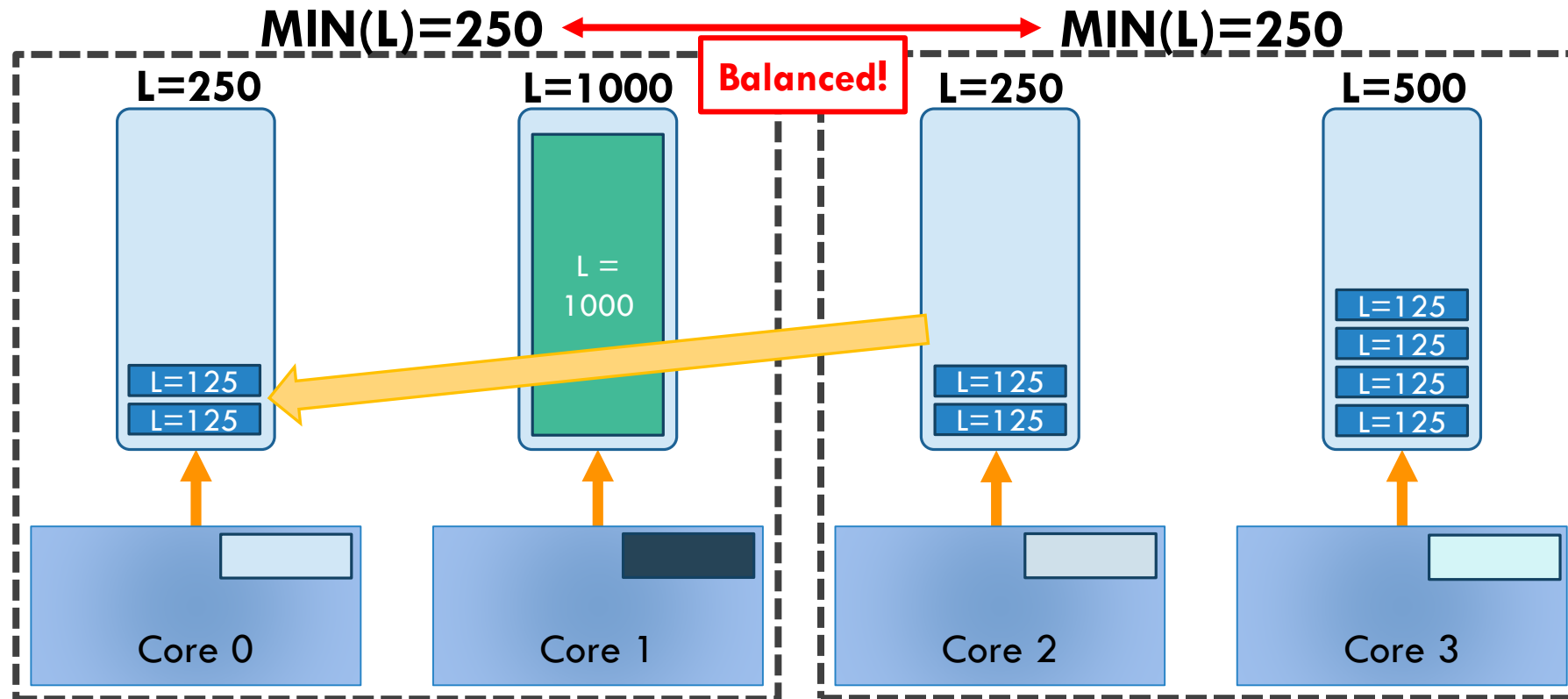
- A simple solution: balance the *minimum load* of groups instead of the *average*





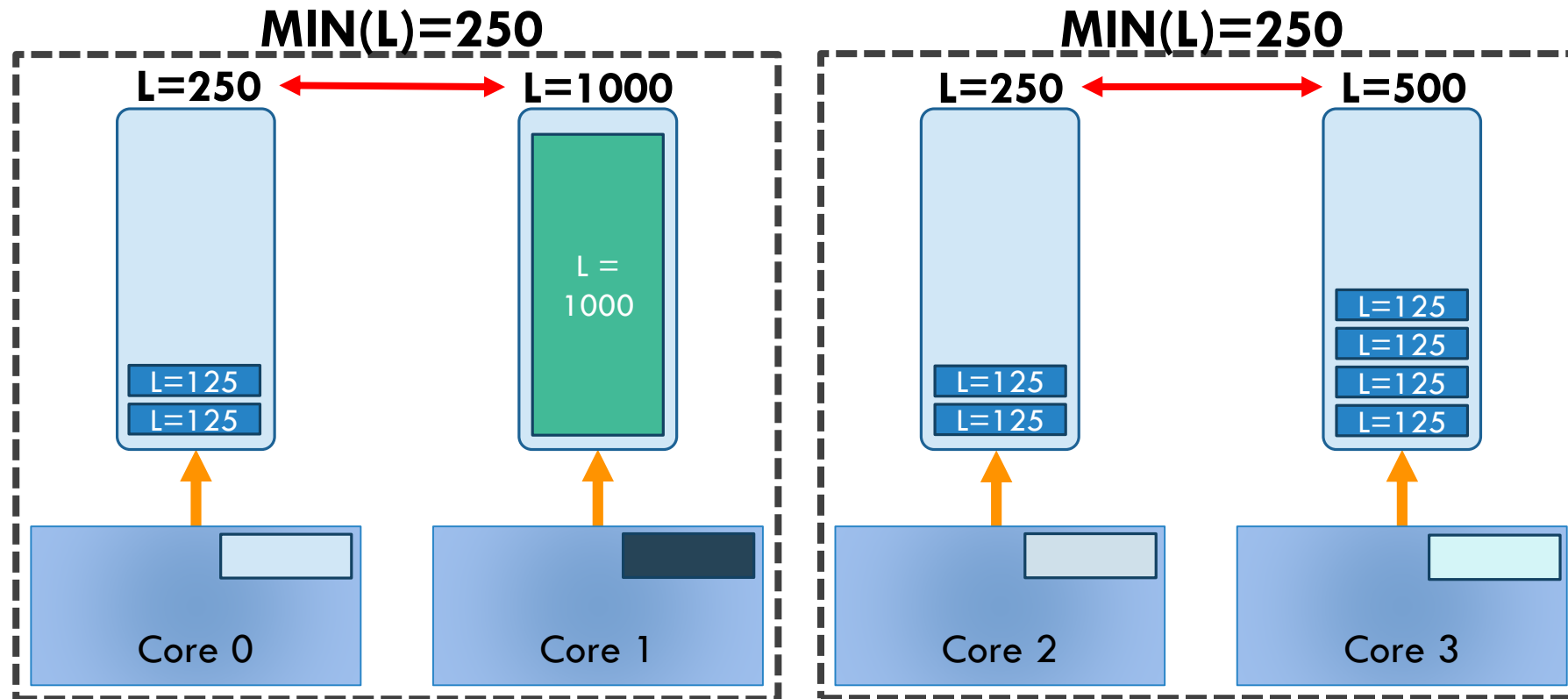
# BUG 1/4: GROUP IMBALANCE

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



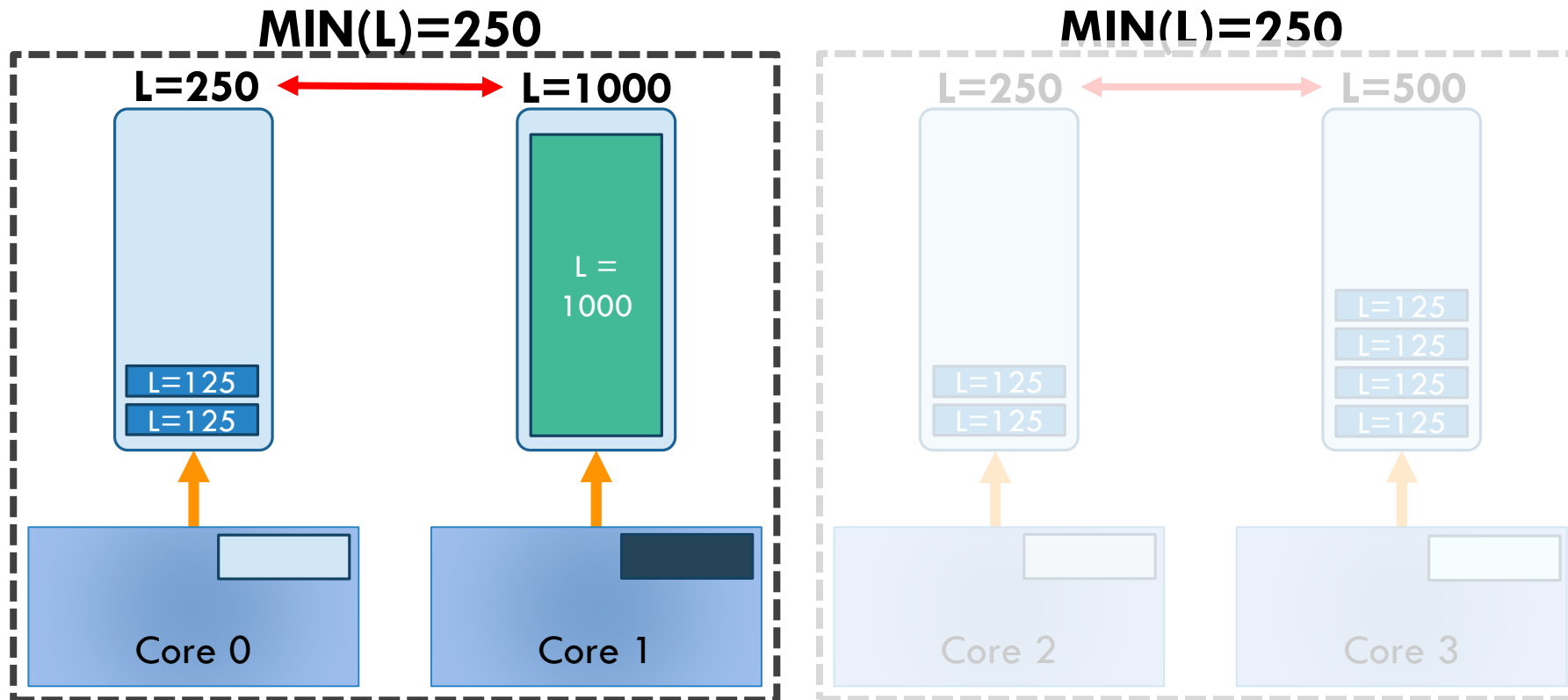
# BUG 1/4: GROUP IMBALANCE

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



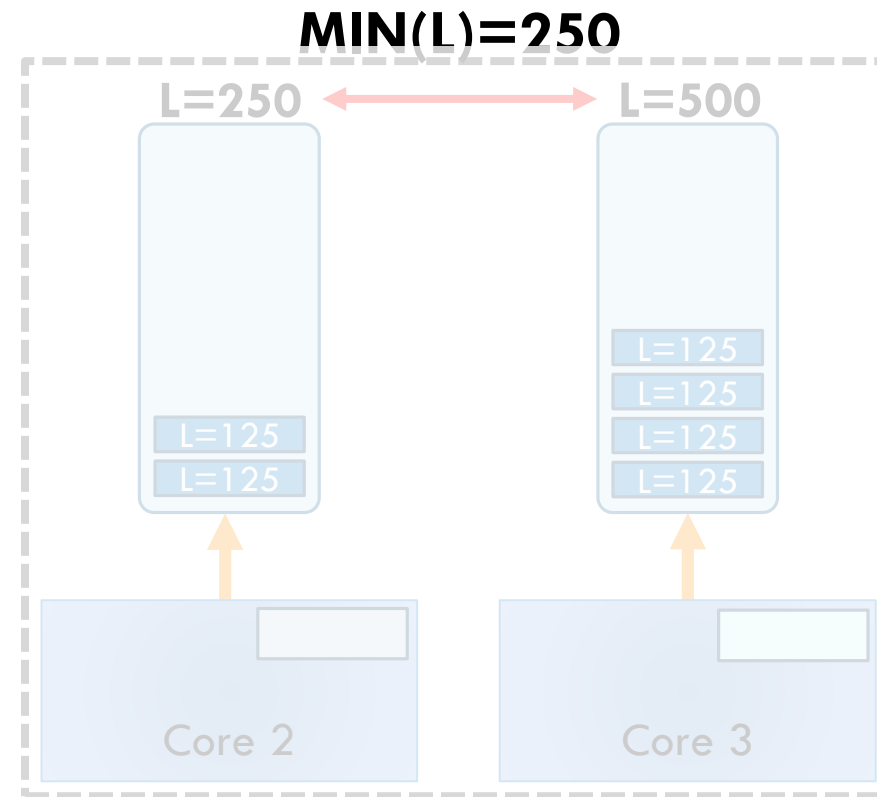
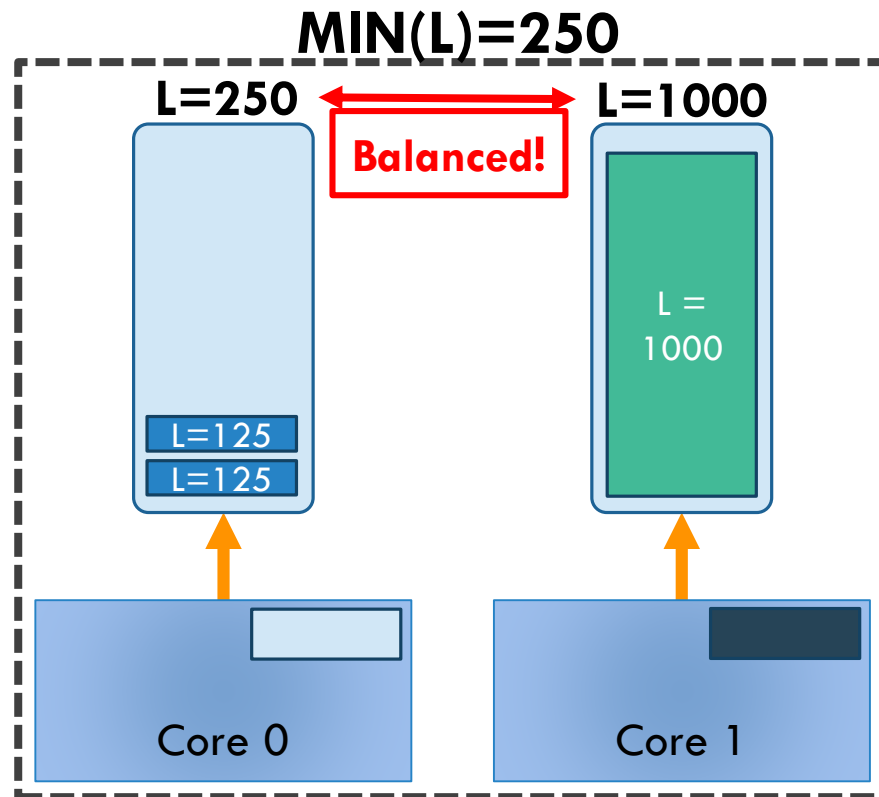
# BUG 1/4: GROUP IMBALANCE

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



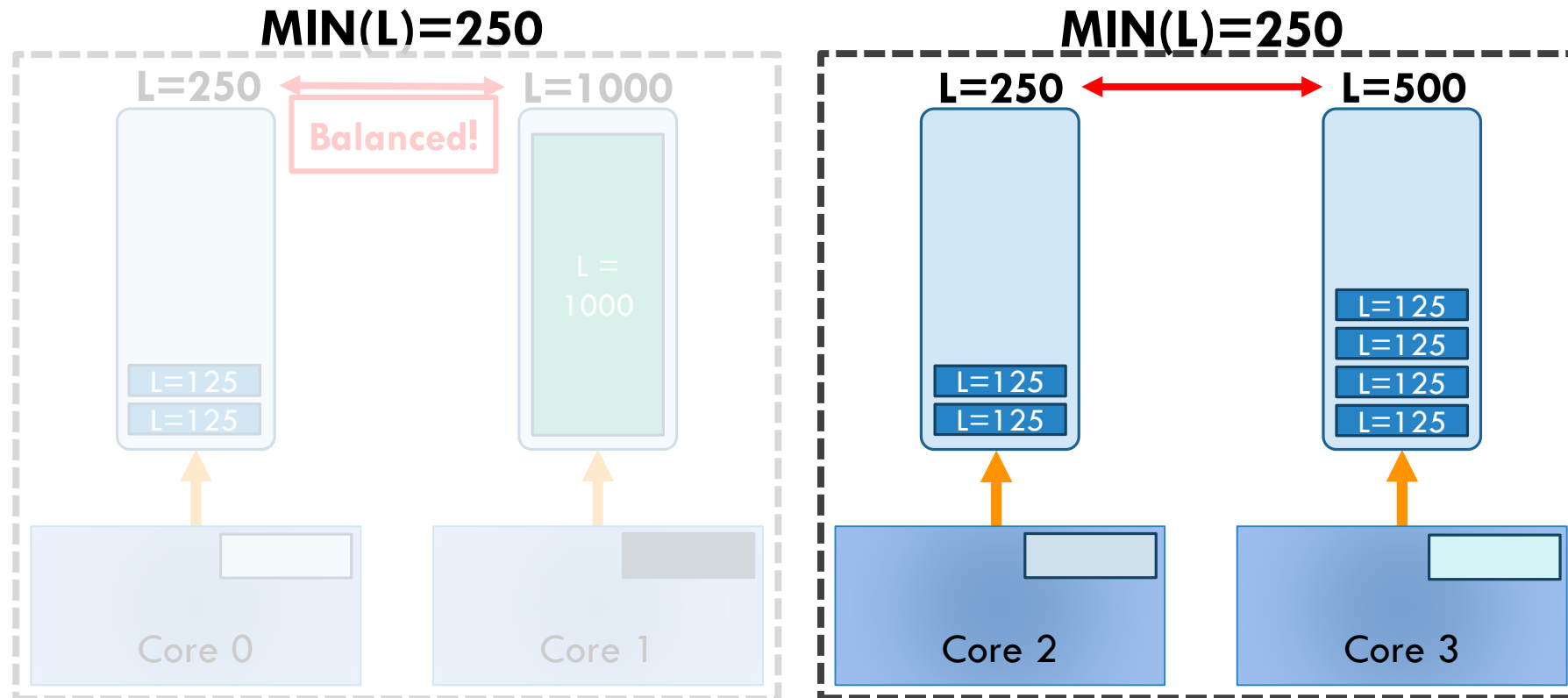
# BUG 1/4: GROUP IMBALANCE

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



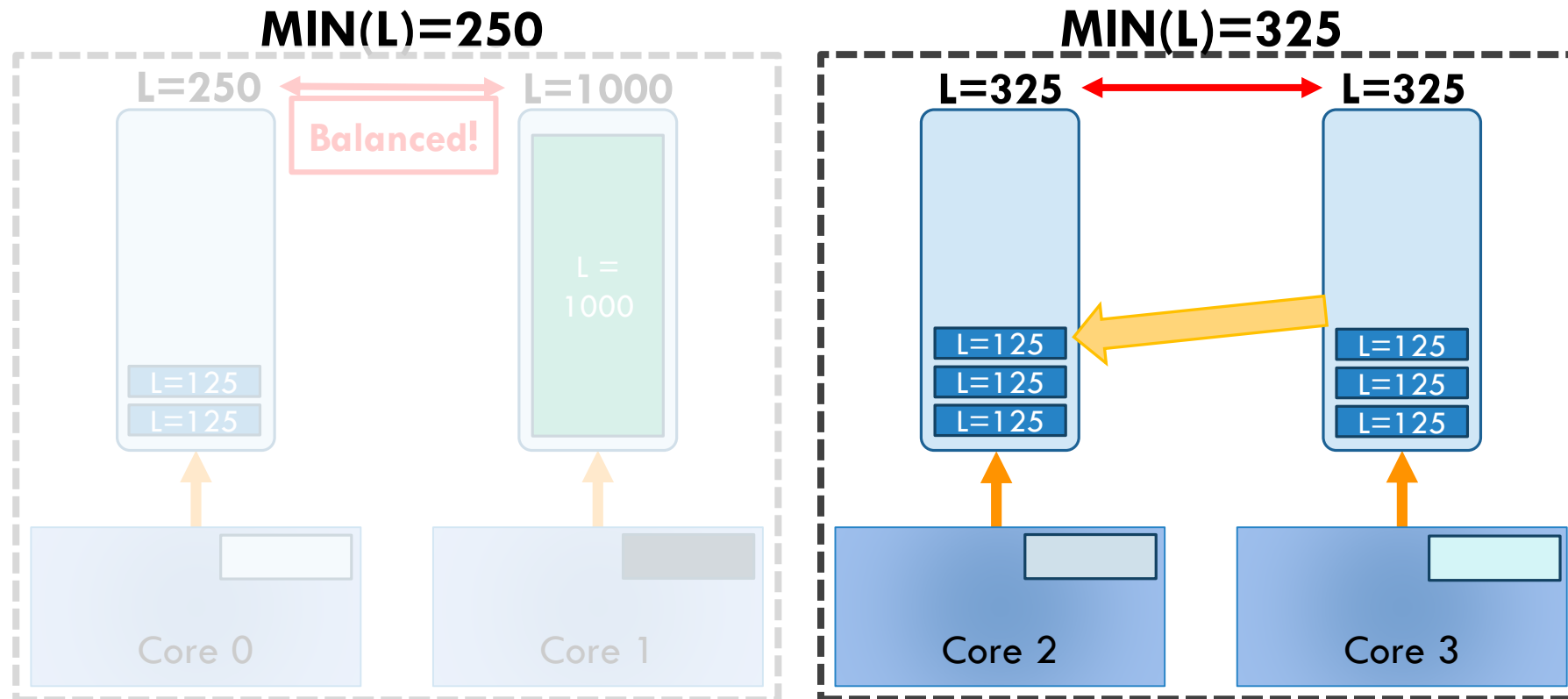
# BUG 1/4: GROUP IMBALANCE

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



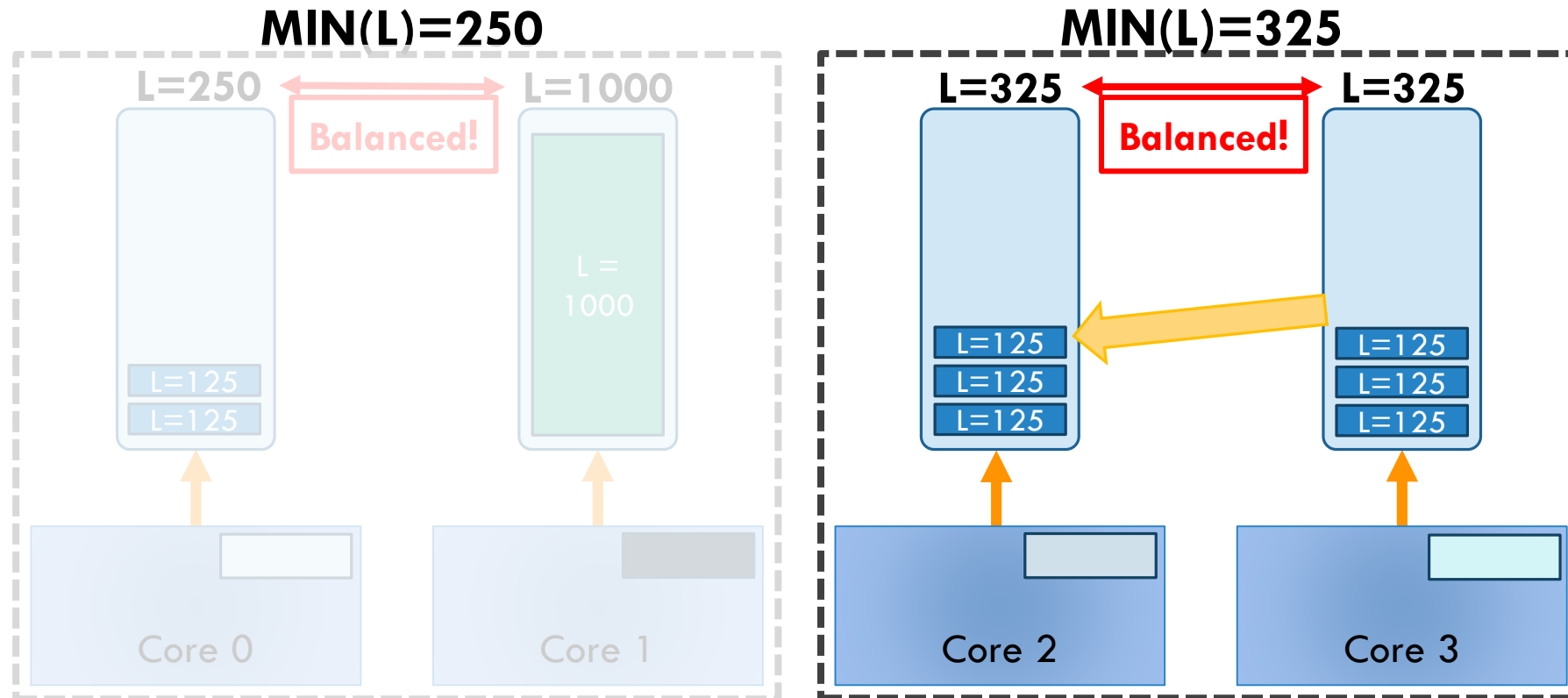
# BUG 1/4: GROUP IMBALANCE

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



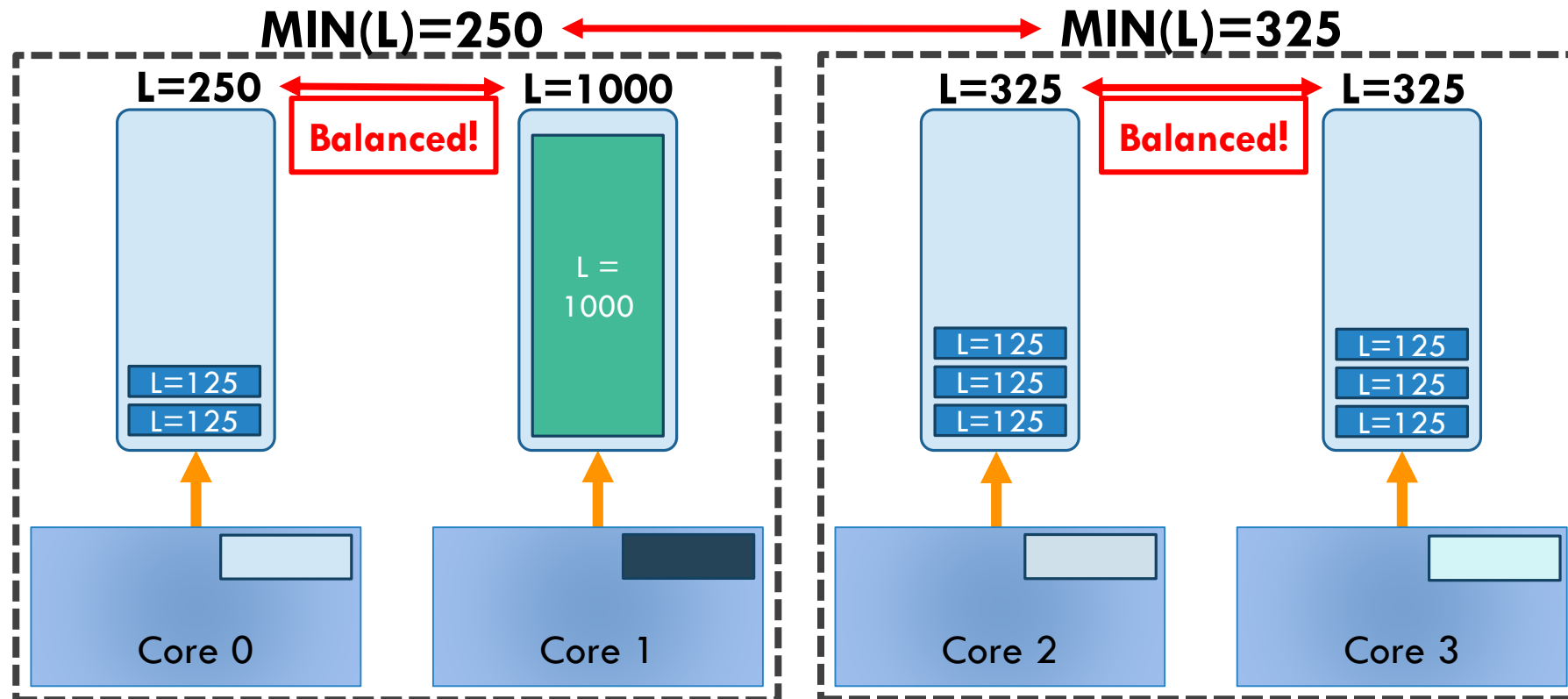
# BUG 1/4: GROUP IMBALANCE

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



# BUG 1/4: GROUP IMBALANCE

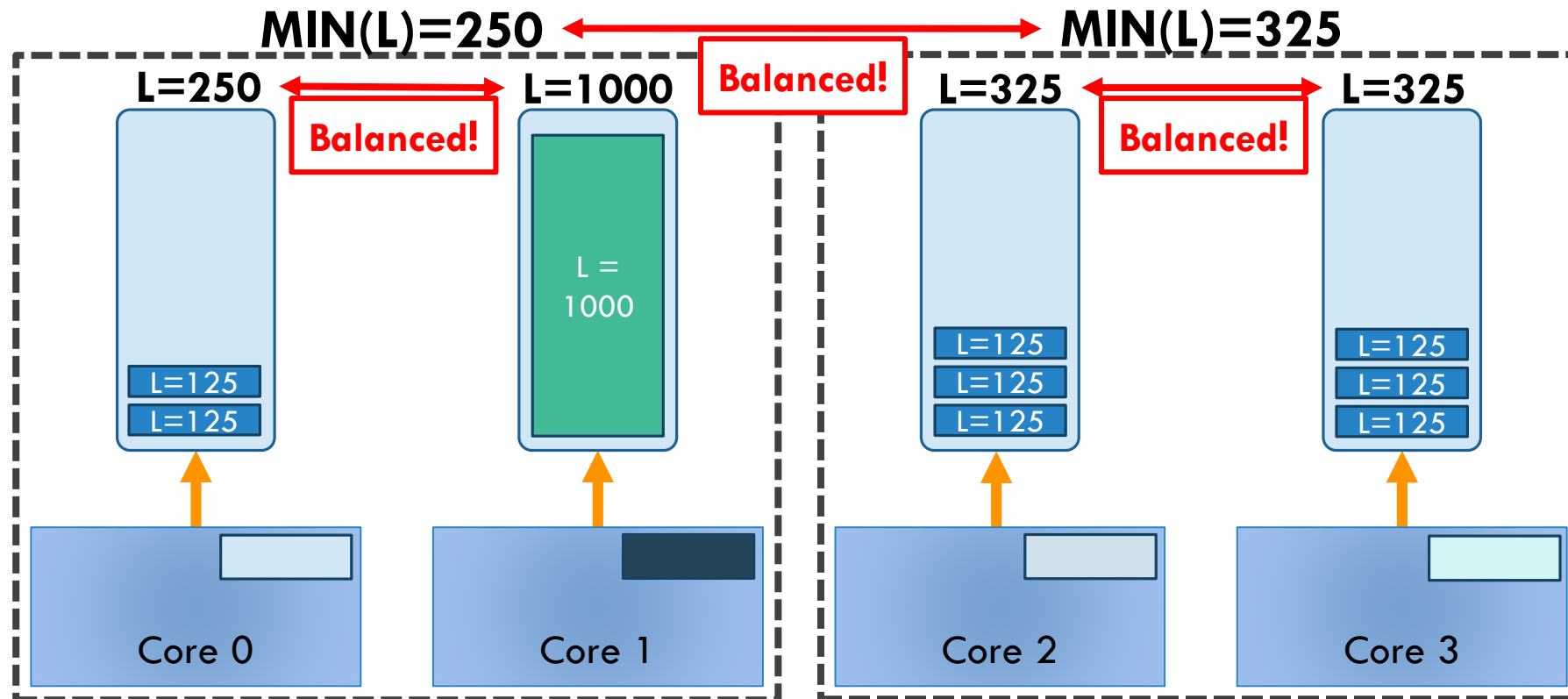
- A simple solution: balance the *minimum* load of groups instead of the *average*





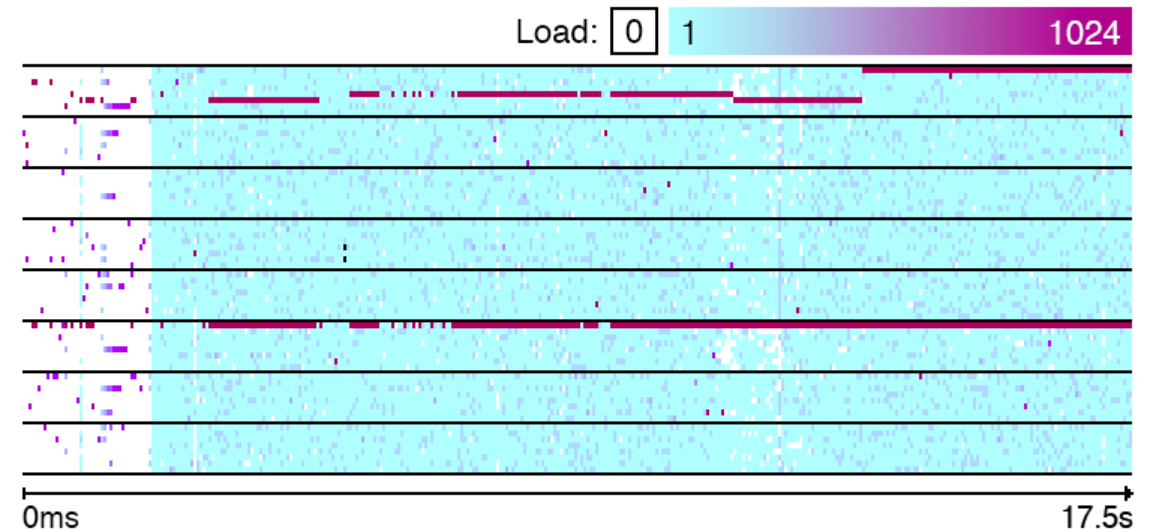
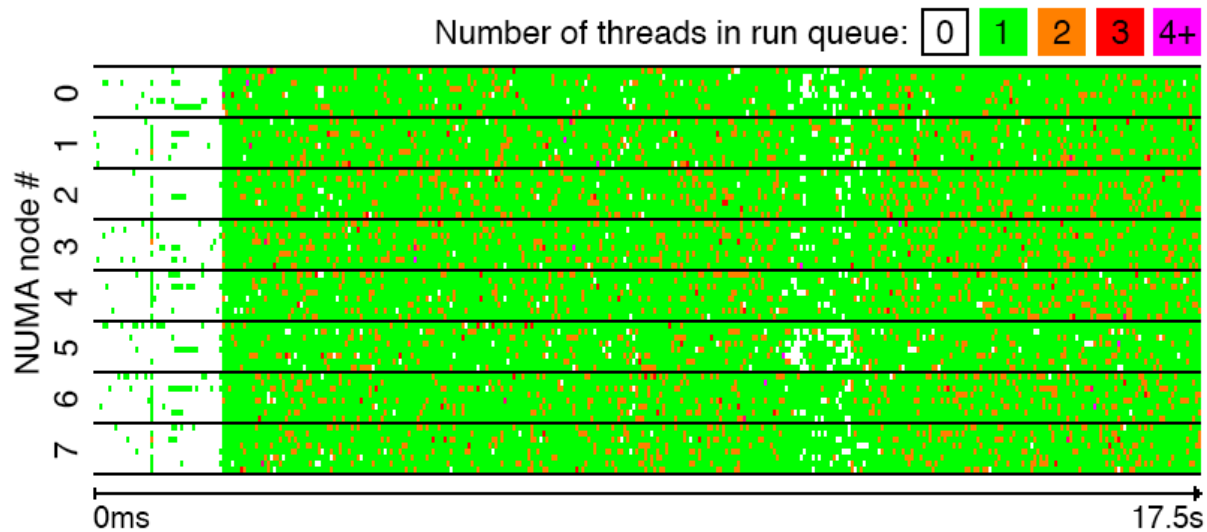
# BUG 1/4: GROUP IMBALANCE

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



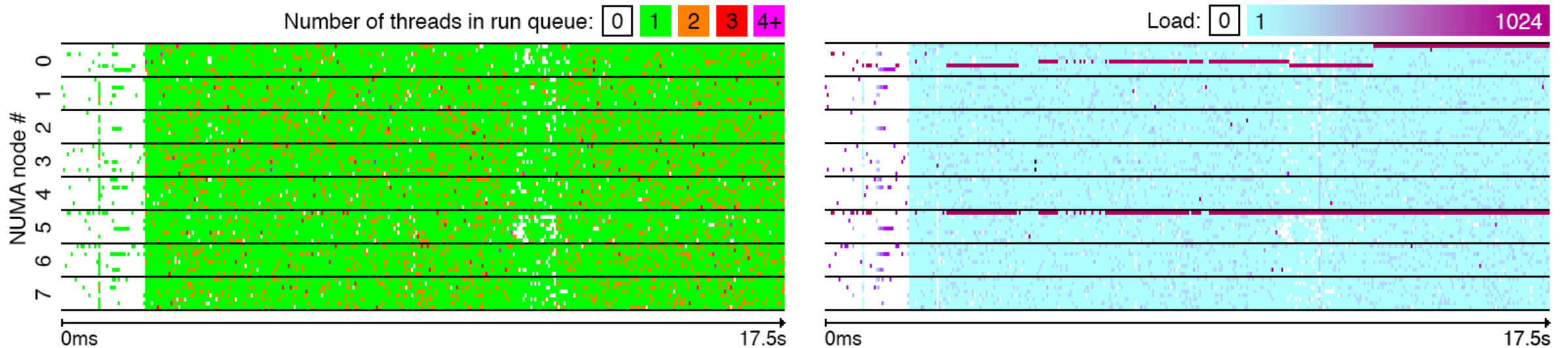
# BUG 1/4: GROUP IMBALANCE

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



# BUG 1/4: GROUP IMBALANCE

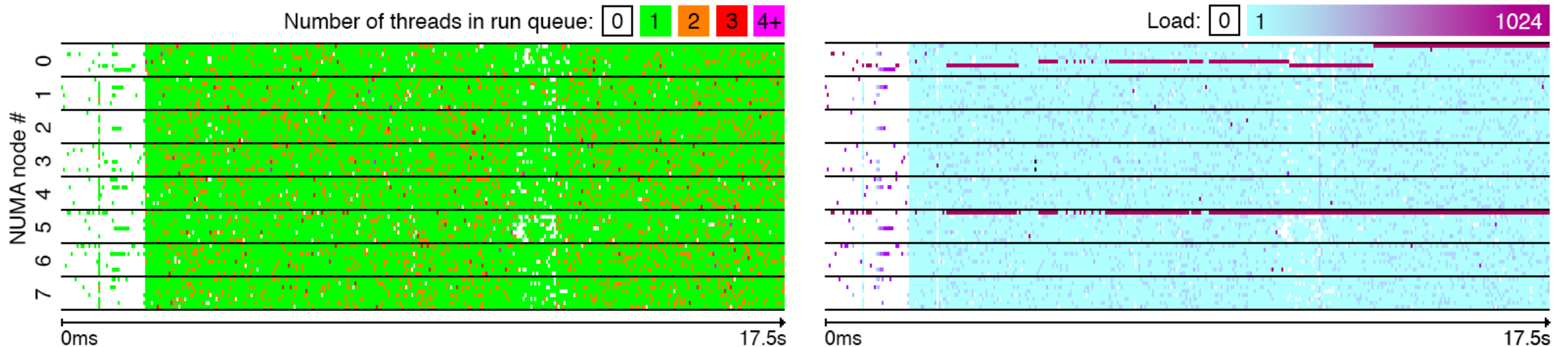
- A simple solution: balance the *minimum* load of groups instead of the *average*



- After the fix, make runs 13% faster, and R is not impacted

# BUG 1/4: GROUP IMBALANCE

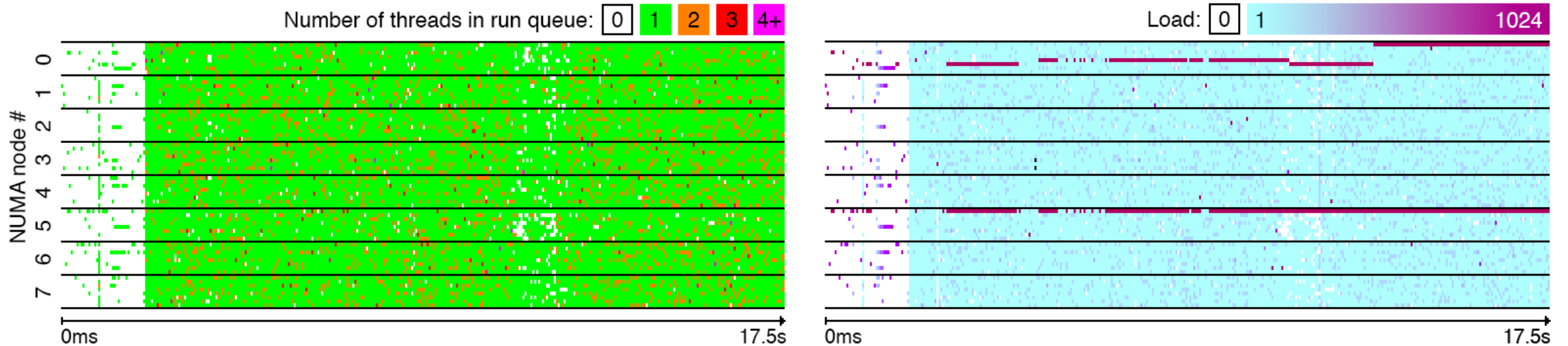
- A simple solution: balance the *minimum* load of groups instead of the *average*



- 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...

# 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*

# 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.

# 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



# 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:**

# 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)

# 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.)

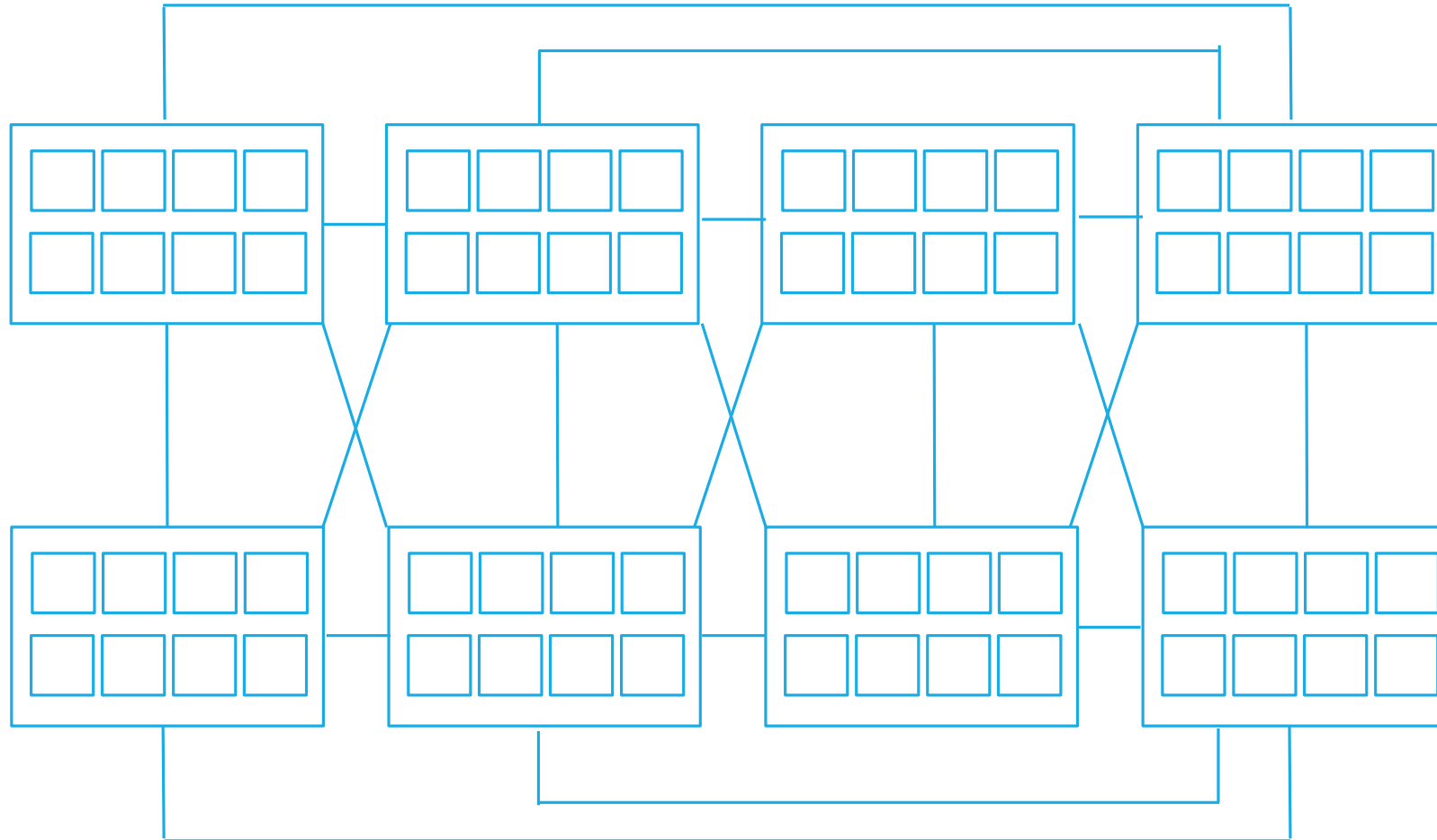
# BUG 2/4: SCHEDULING GROUP CONSTRUCTION

- **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.)

# BUG 2/4: SCHEDULING GROUP CONSTRUCTION

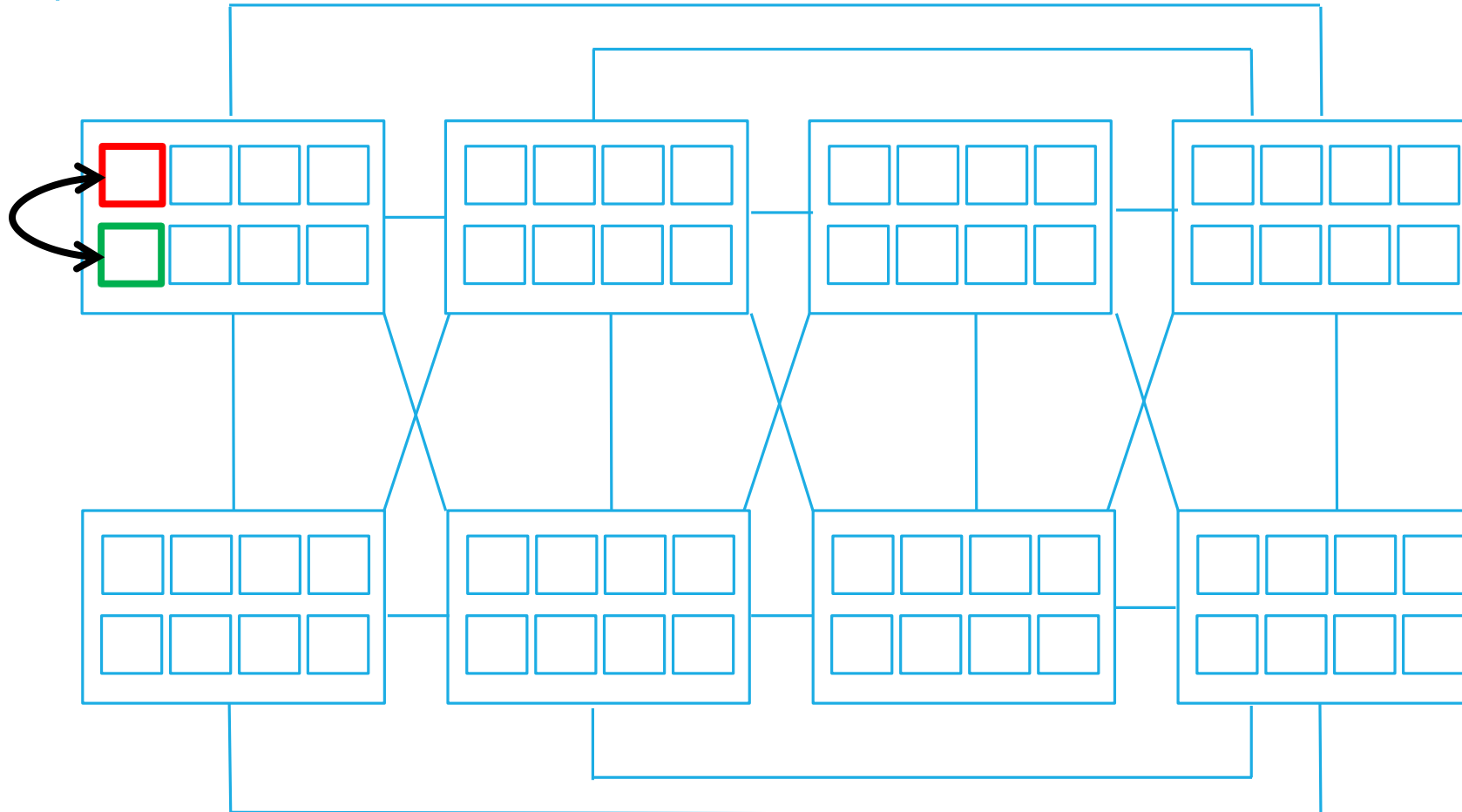
- **Hierarchical load balancing** is based on groups of cores named *scheduling domains*
  - Based on affinity, i.e., pairs of cores, dies, CPUs, NUMA nodes, etc.
- Each scheduling domain contains groups that are the lower-level scheduling domains
- **For instance, on our 64-core AMD Bulldozer machine:**
  - 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



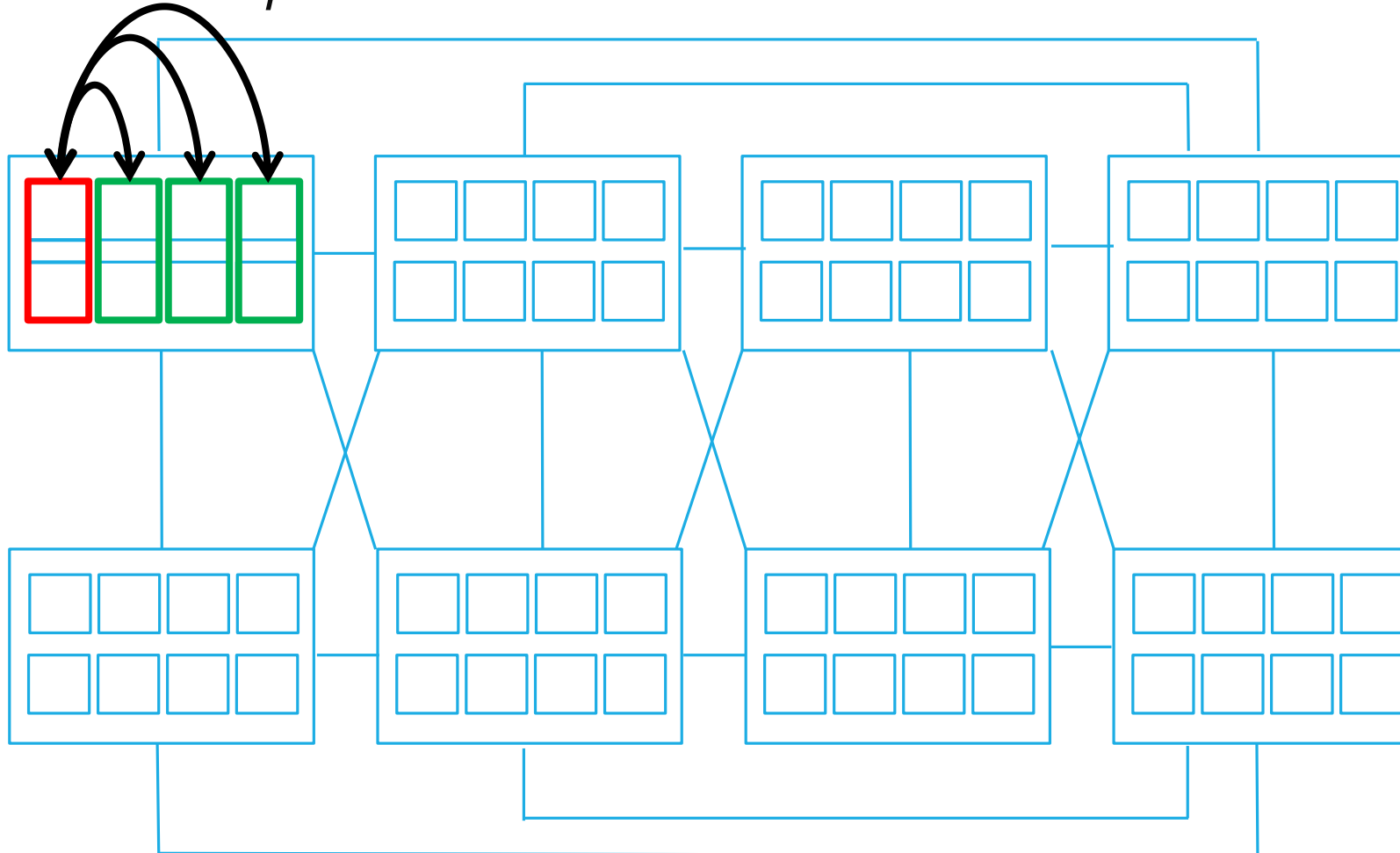
**Bulldozer 64-core:**  
Eight CPUs, with  
8 cores each,  
**non-complete**  
**interconnect graph!**

# 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)

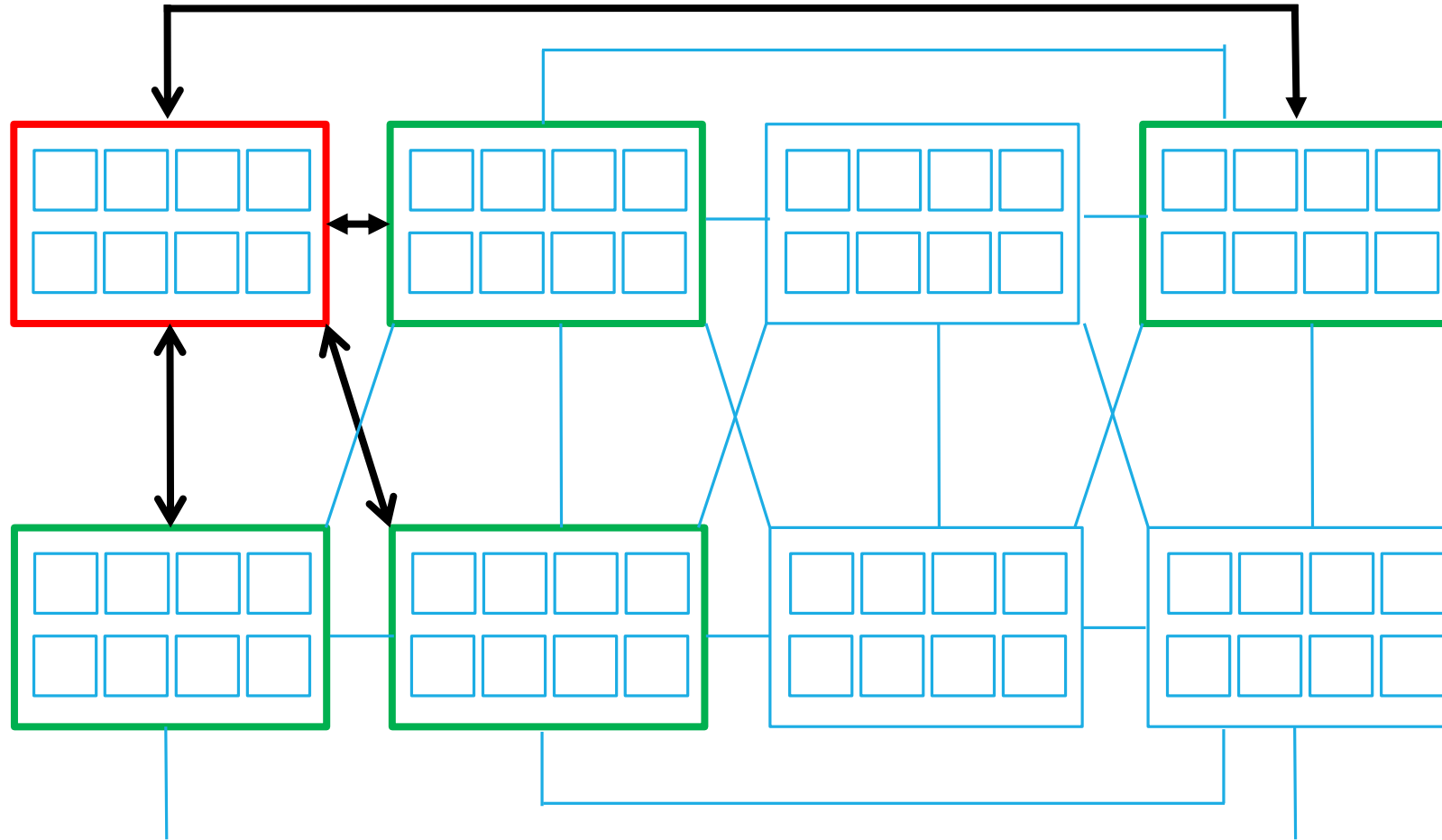
# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



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

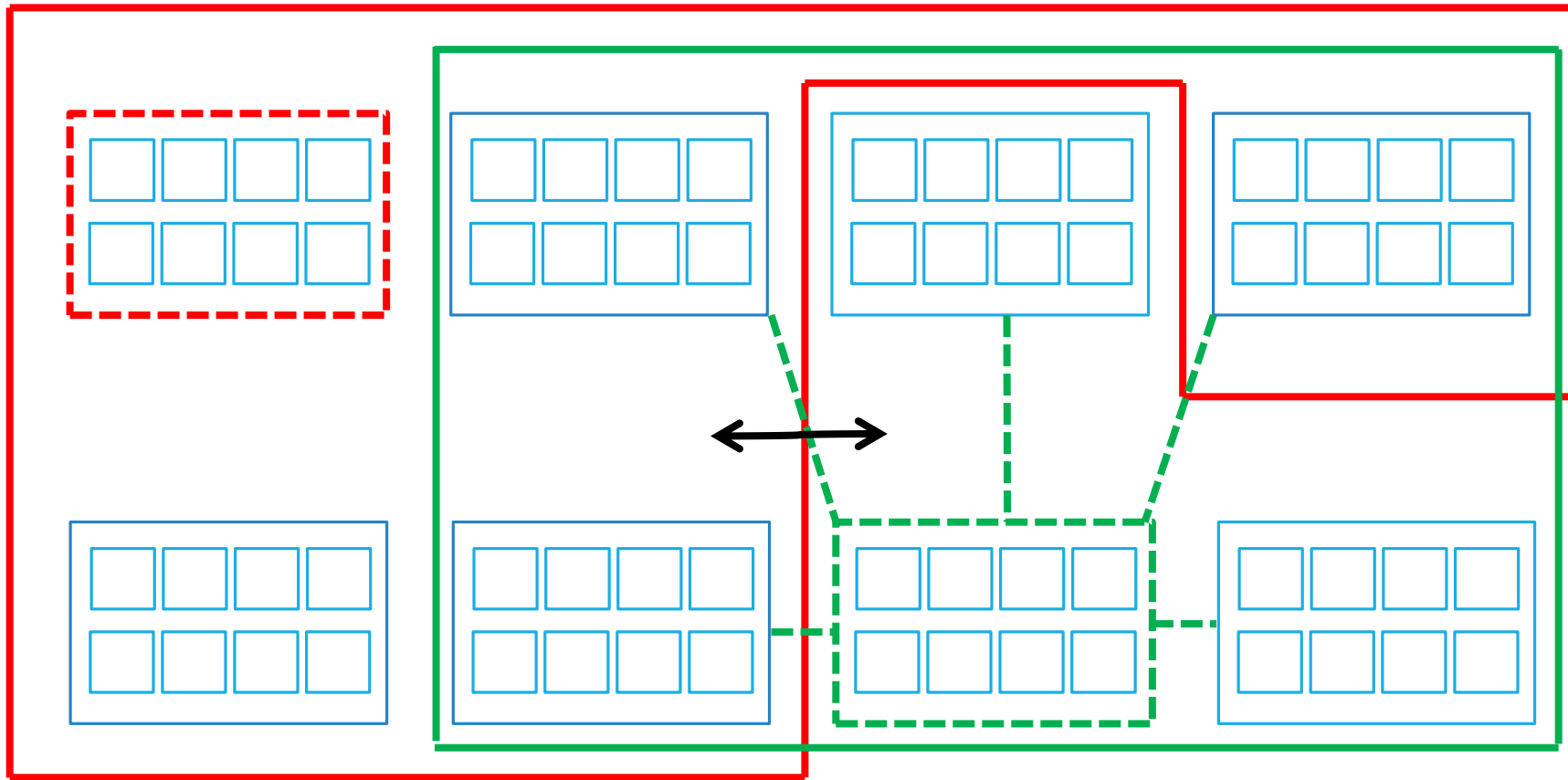


# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



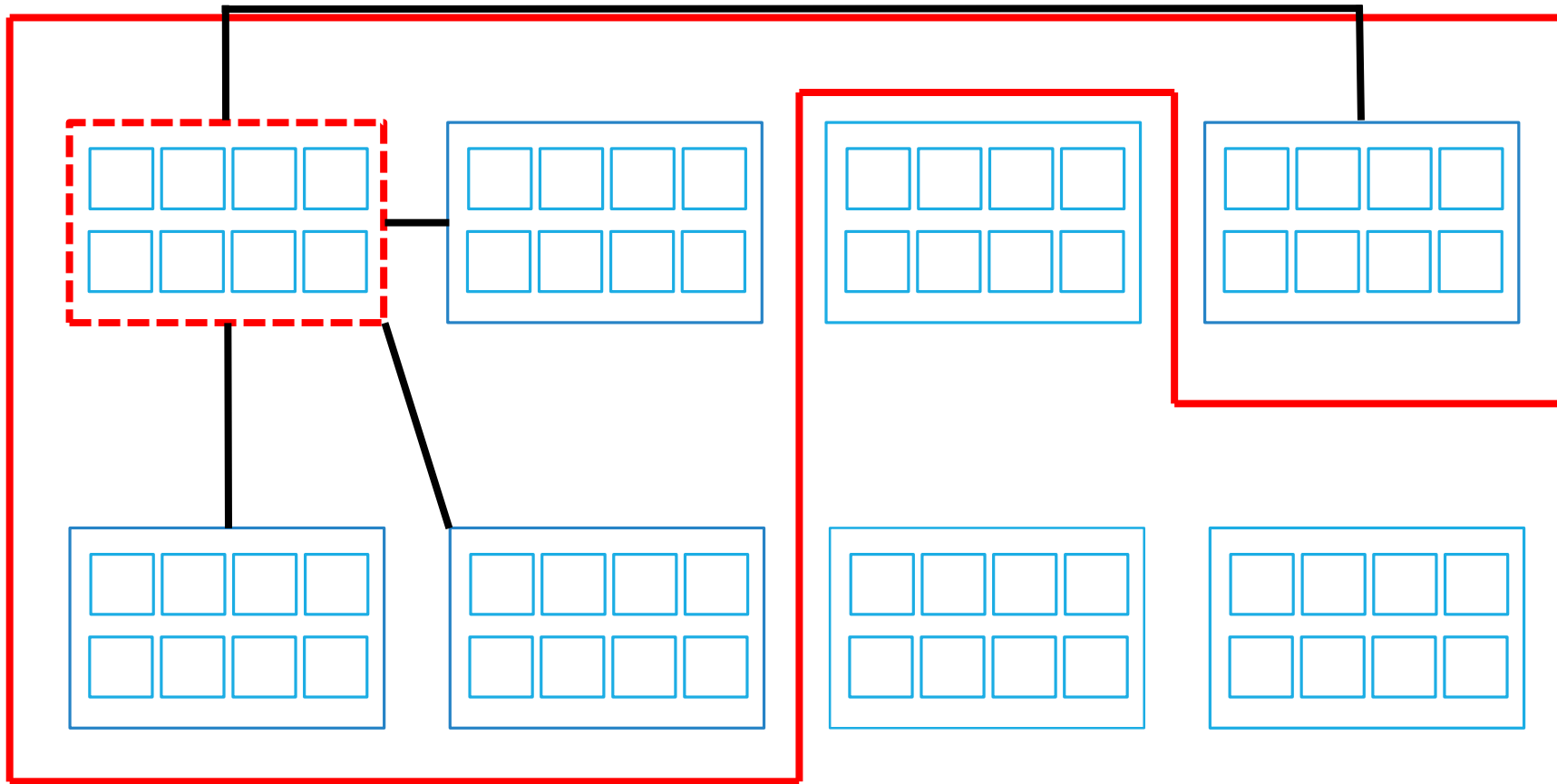
At the 3<sup>rd</sup> level,  
the **first CPU**  
balances load  
with **directly**  
**connected CPUS**

# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



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

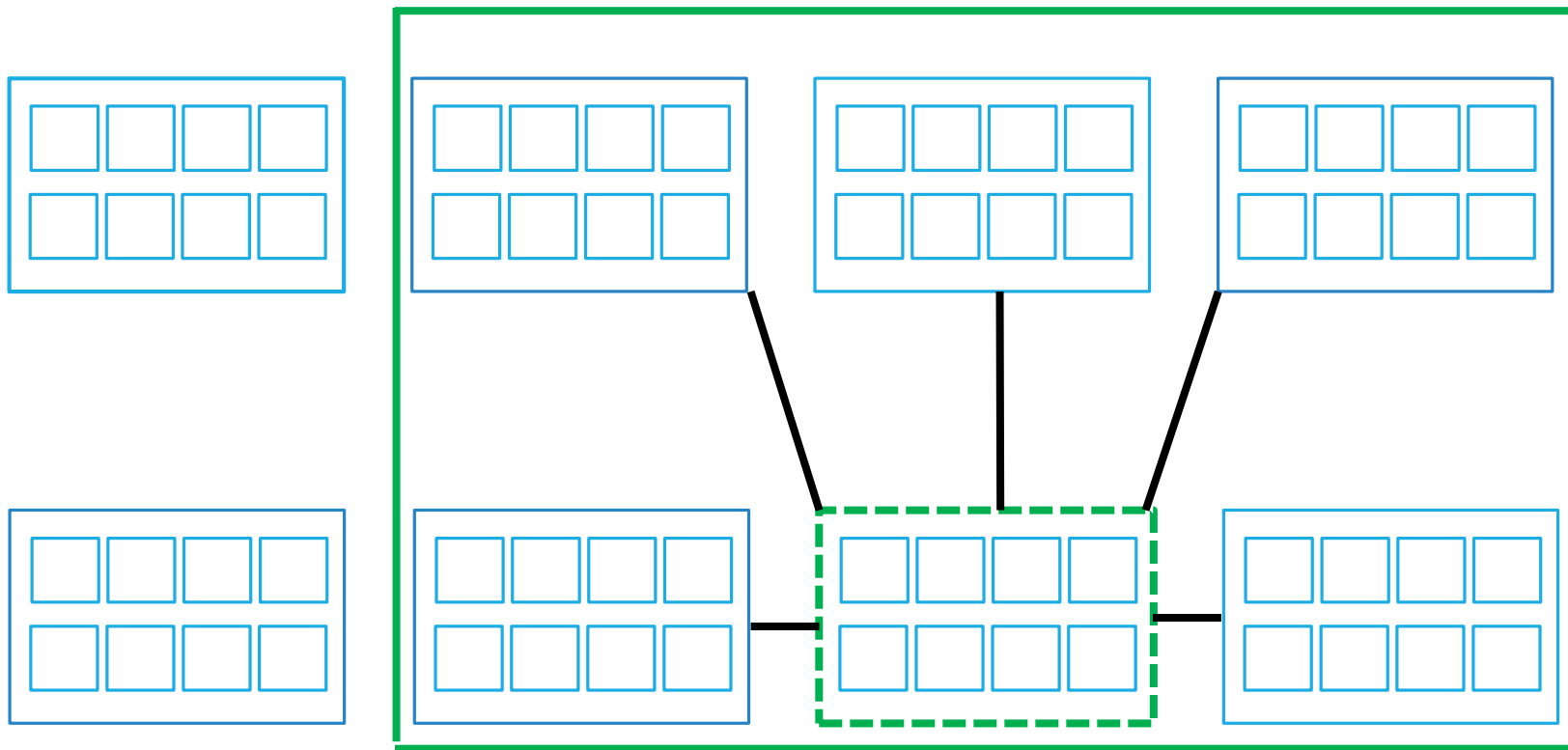
# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



**Groups of CPUs  
built by:**

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

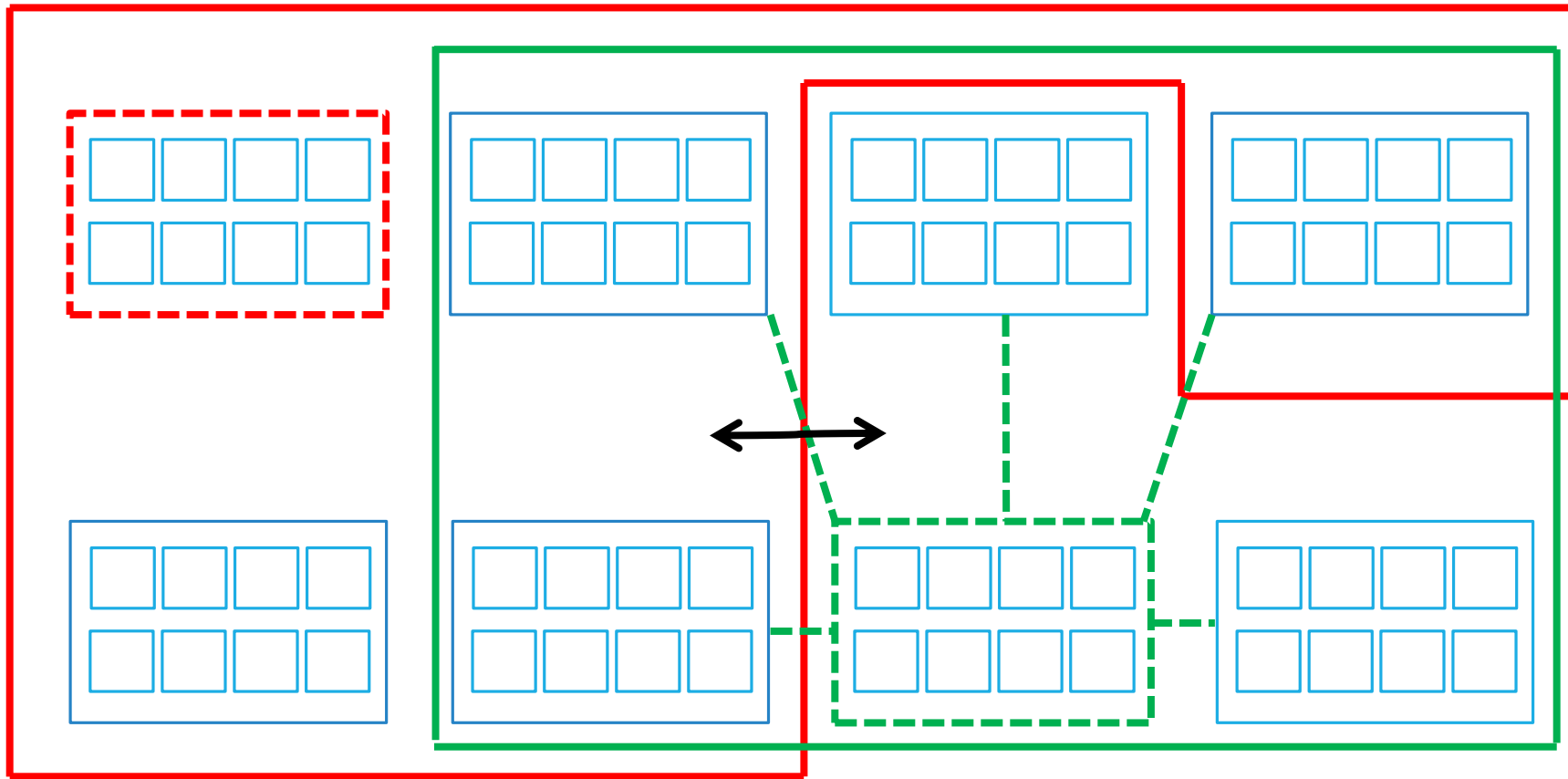
# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



**Groups of CPUs  
built by:**

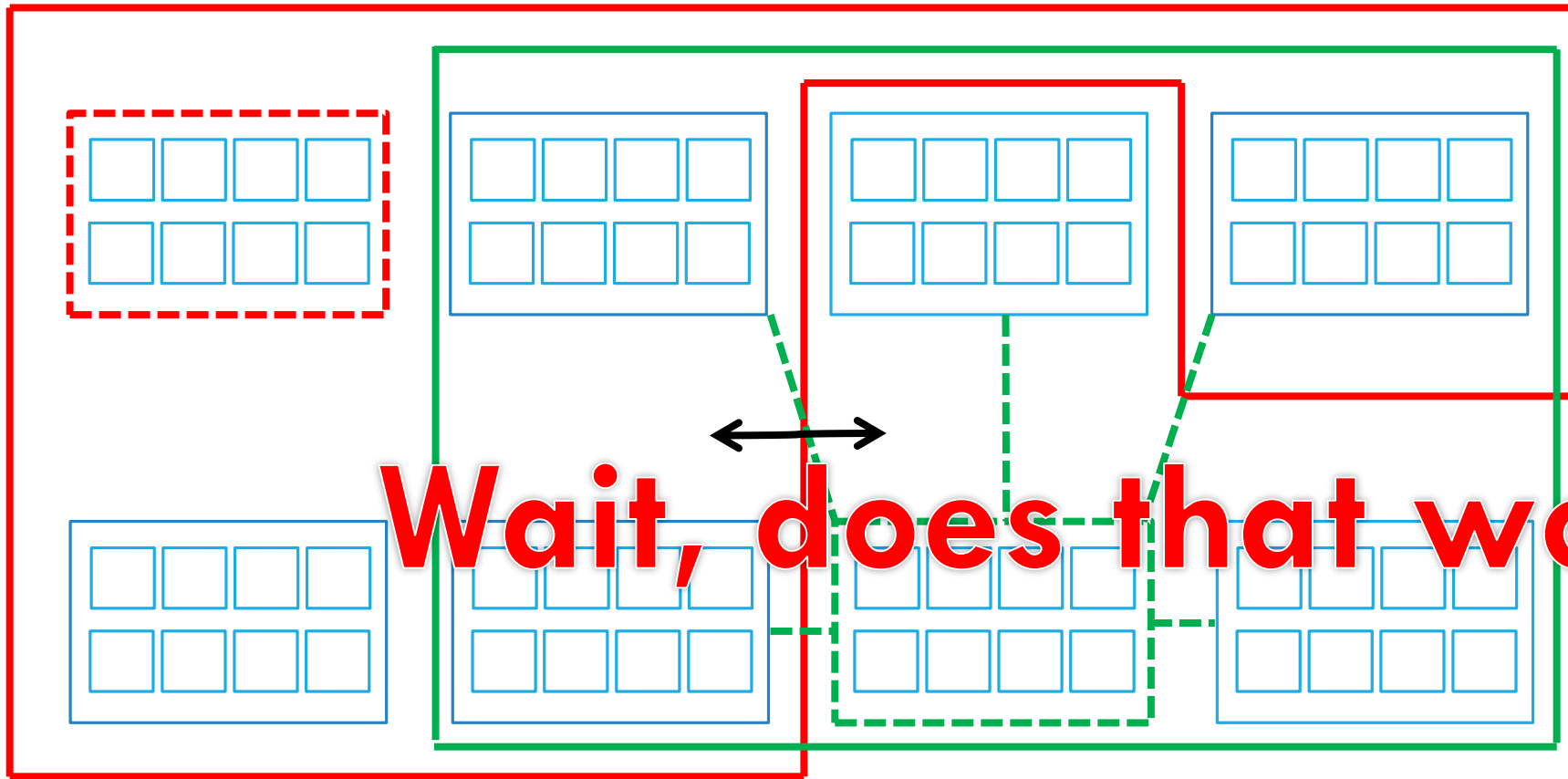
**(2) picking first  
CPU not in a  
group and  
looking for all  
directly  
connected CPUs**

# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



And then stop,  
**because all CPUs  
are in a group**

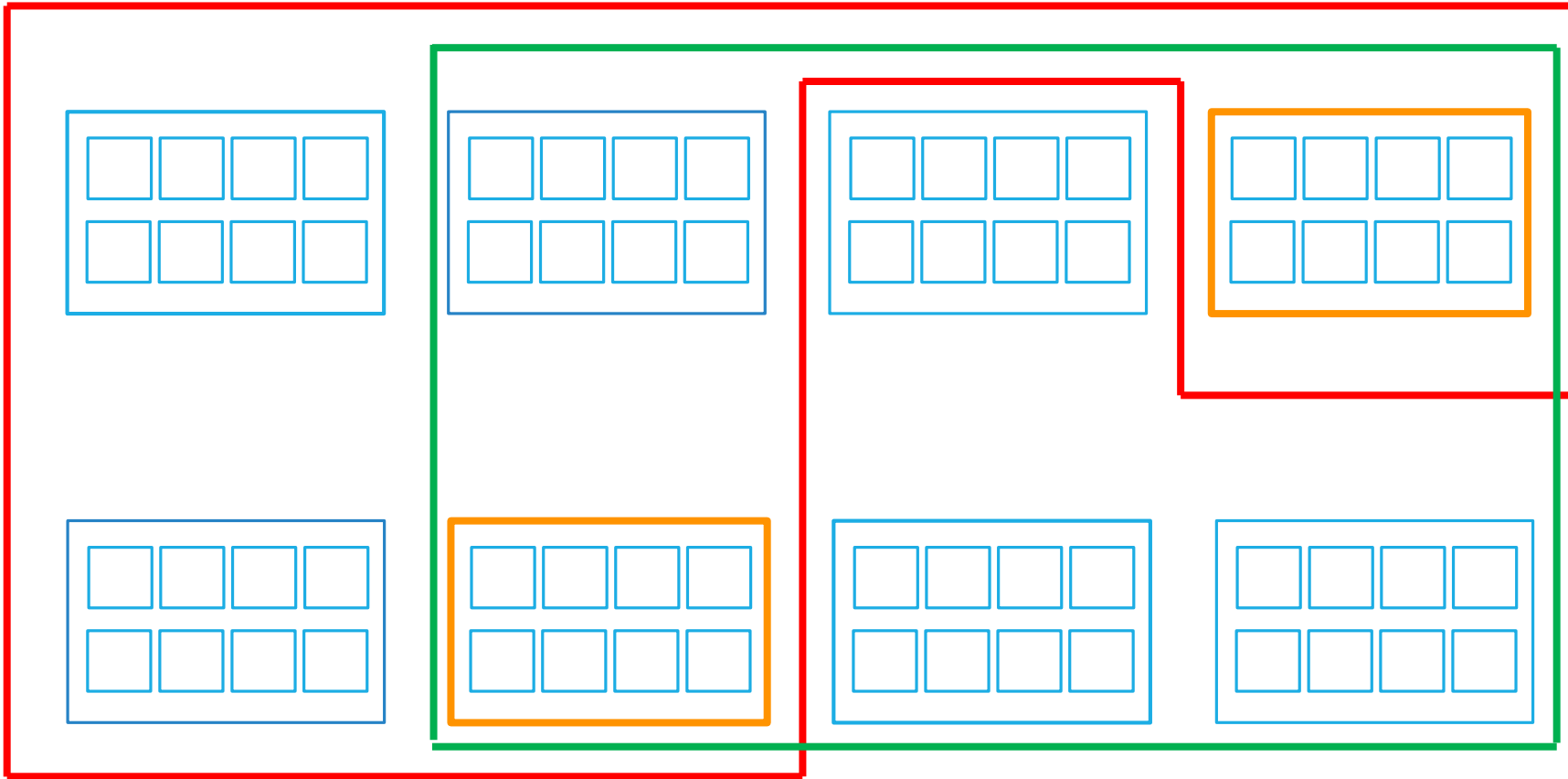
## BUG 2/4: SCHEDULING GROUP CONSTRUCTION



And then stop,  
**because all CPUs**  
**are in a group**

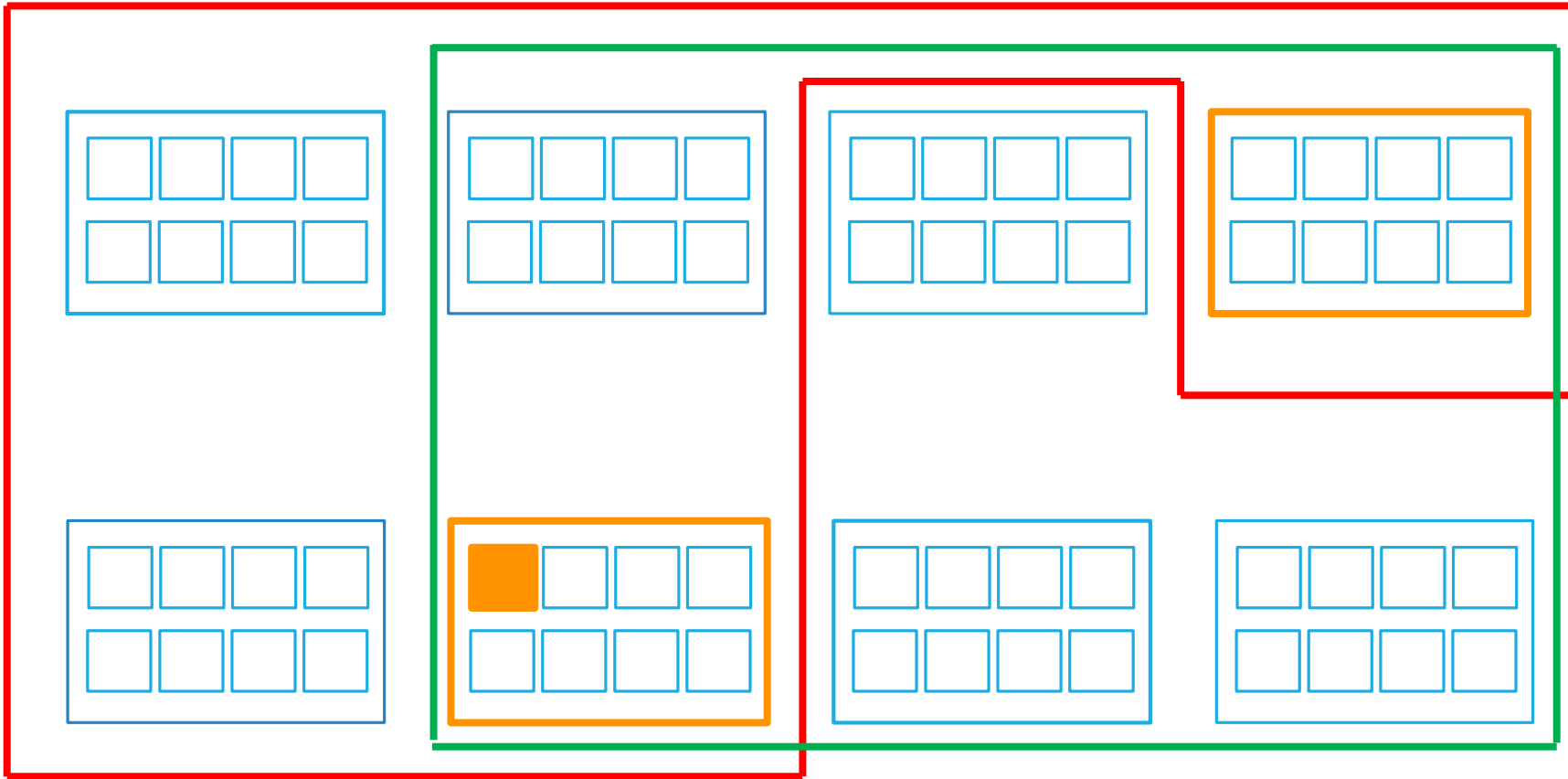
**Wait, does that work?**

# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



Suppose we  
taskset an  
application on  
**these two nodes**,  
two hops apart

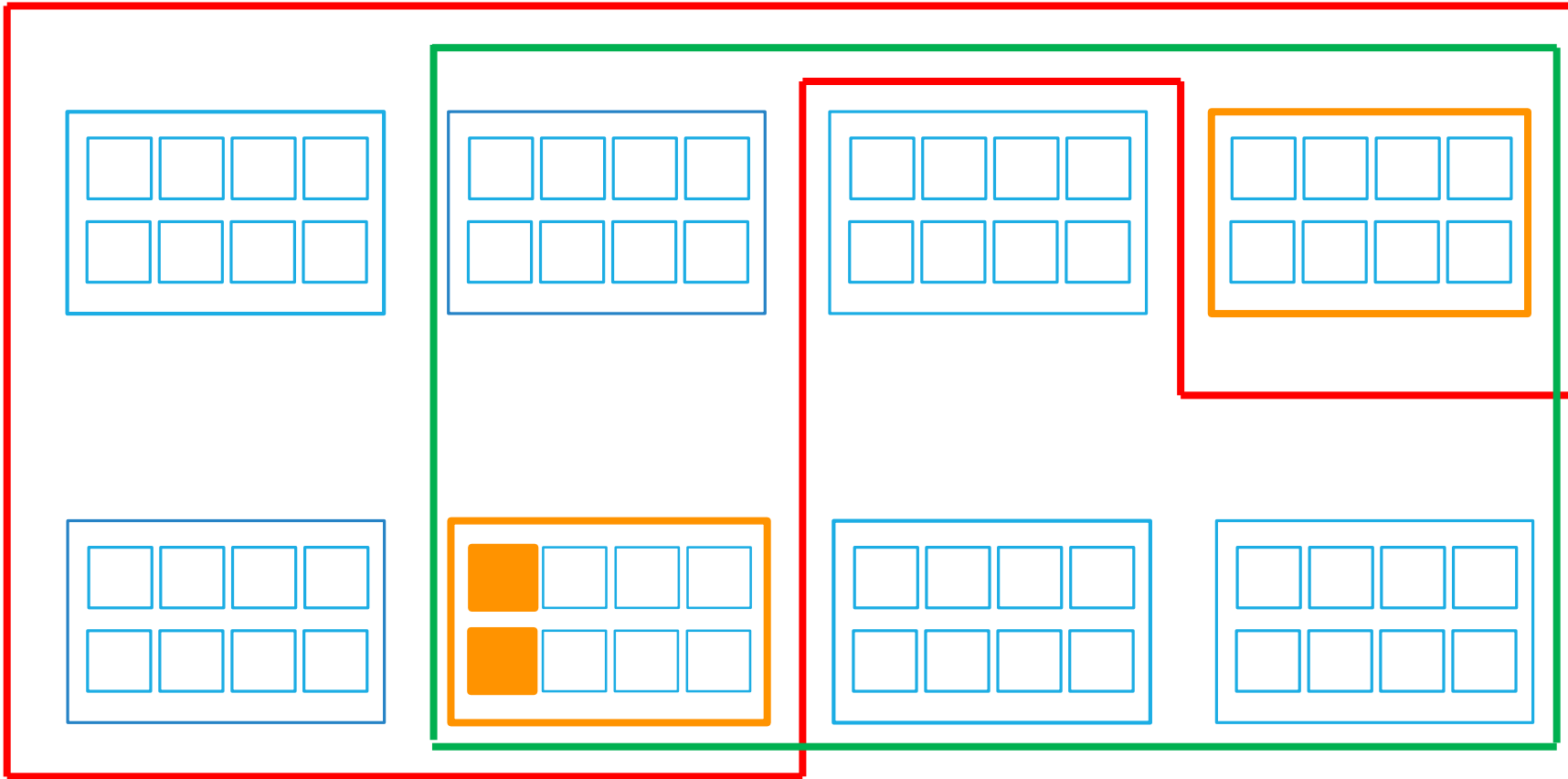
# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



And threads  
are created  
**on this core**

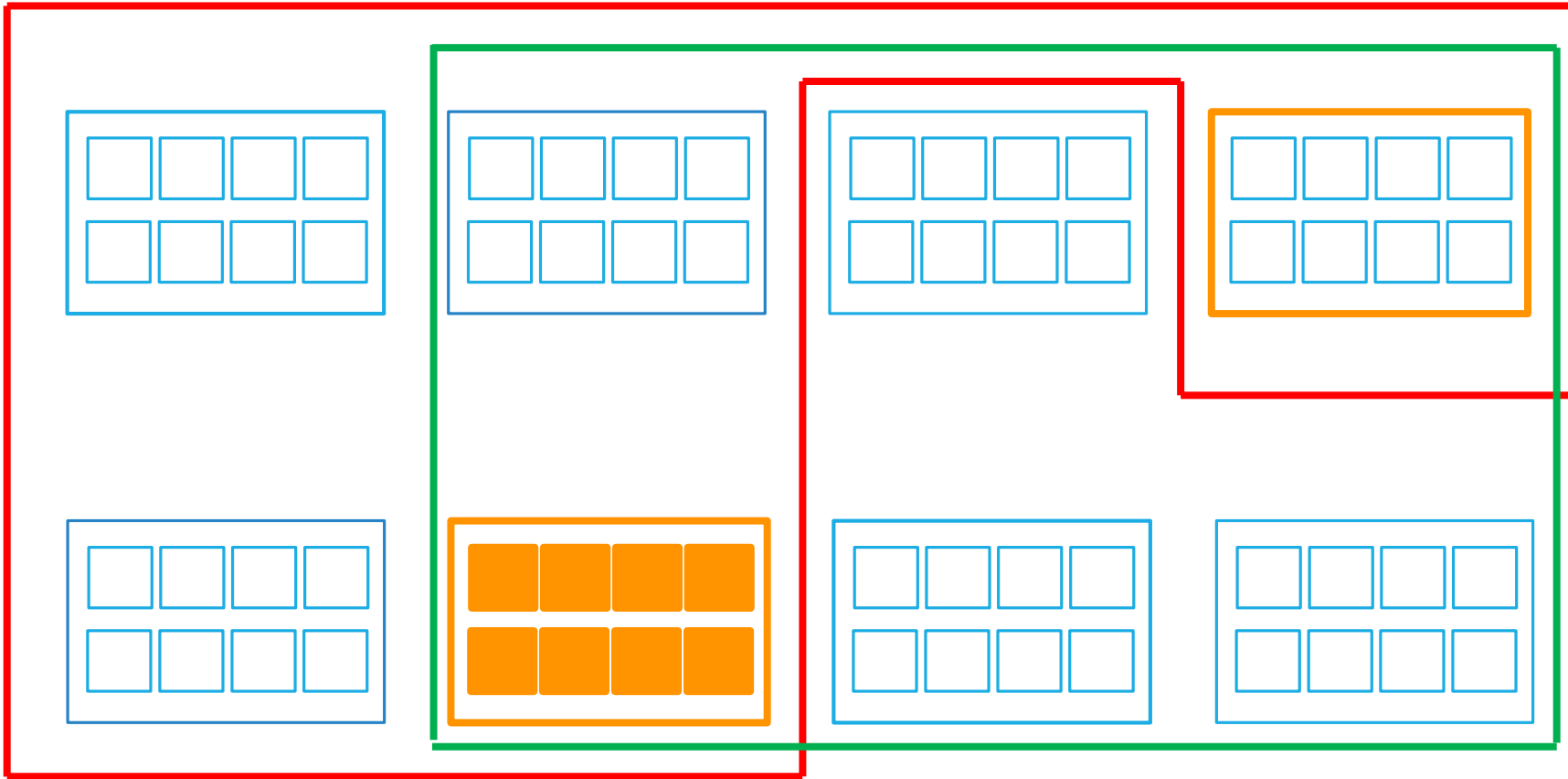


# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



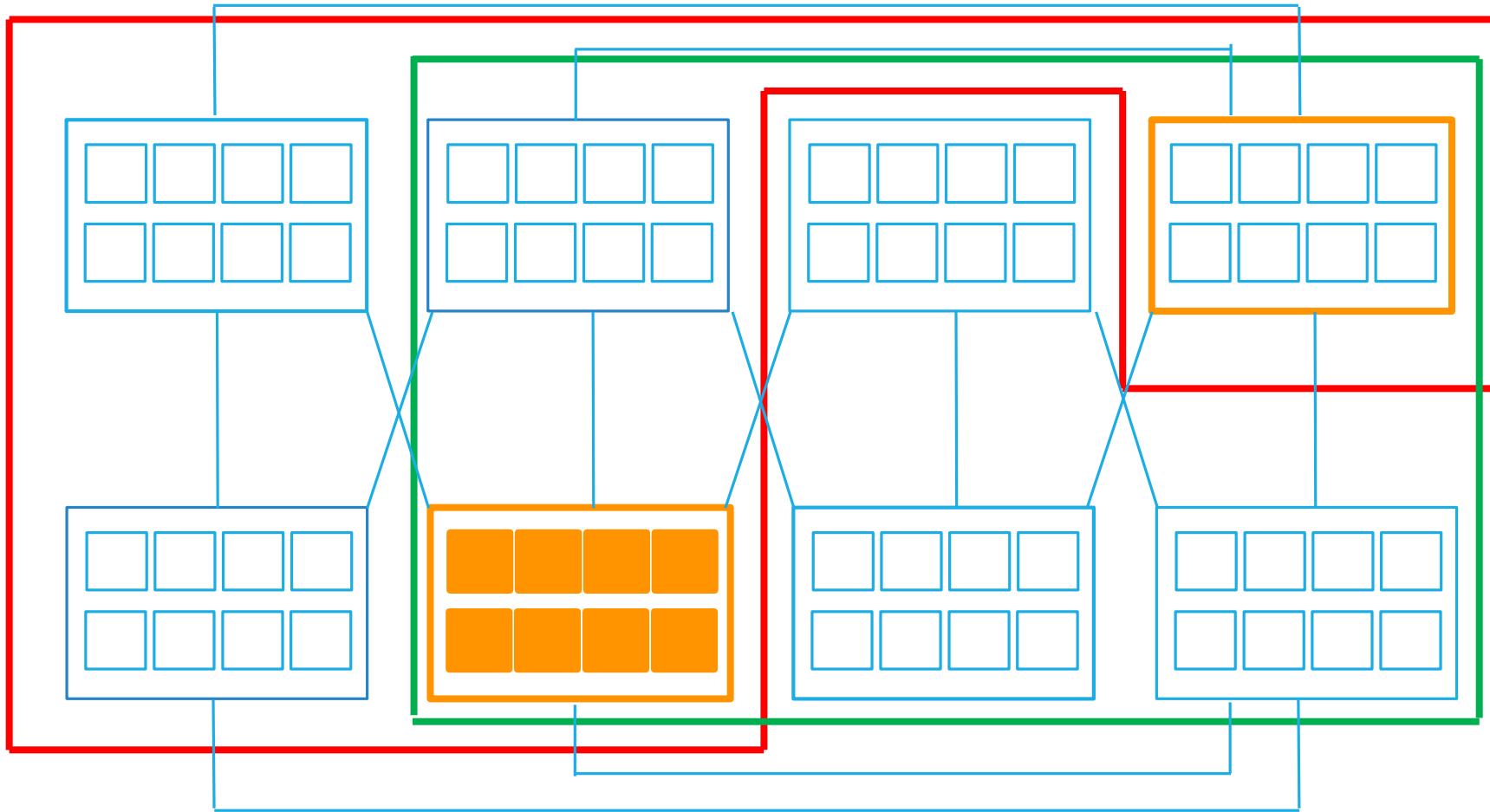
Load gets  
correctly balanced  
**on the pair of  
cores**

# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



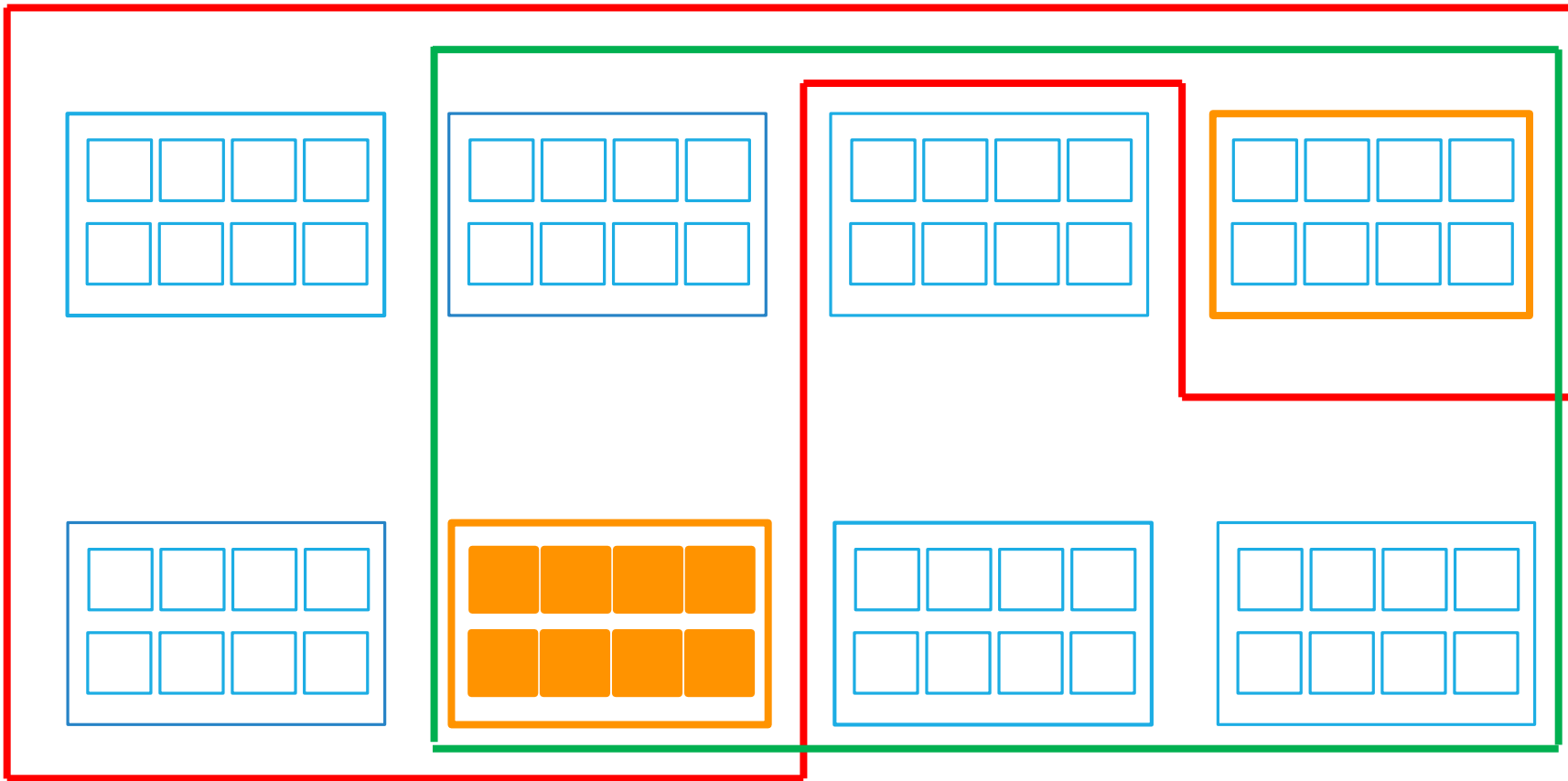
Load gets  
correctly balanced  
**on the CPU**  
(8 threads)

# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



No stealing  
at level 3,  
because nodes  
not directly  
connected (1 hop  
apart)

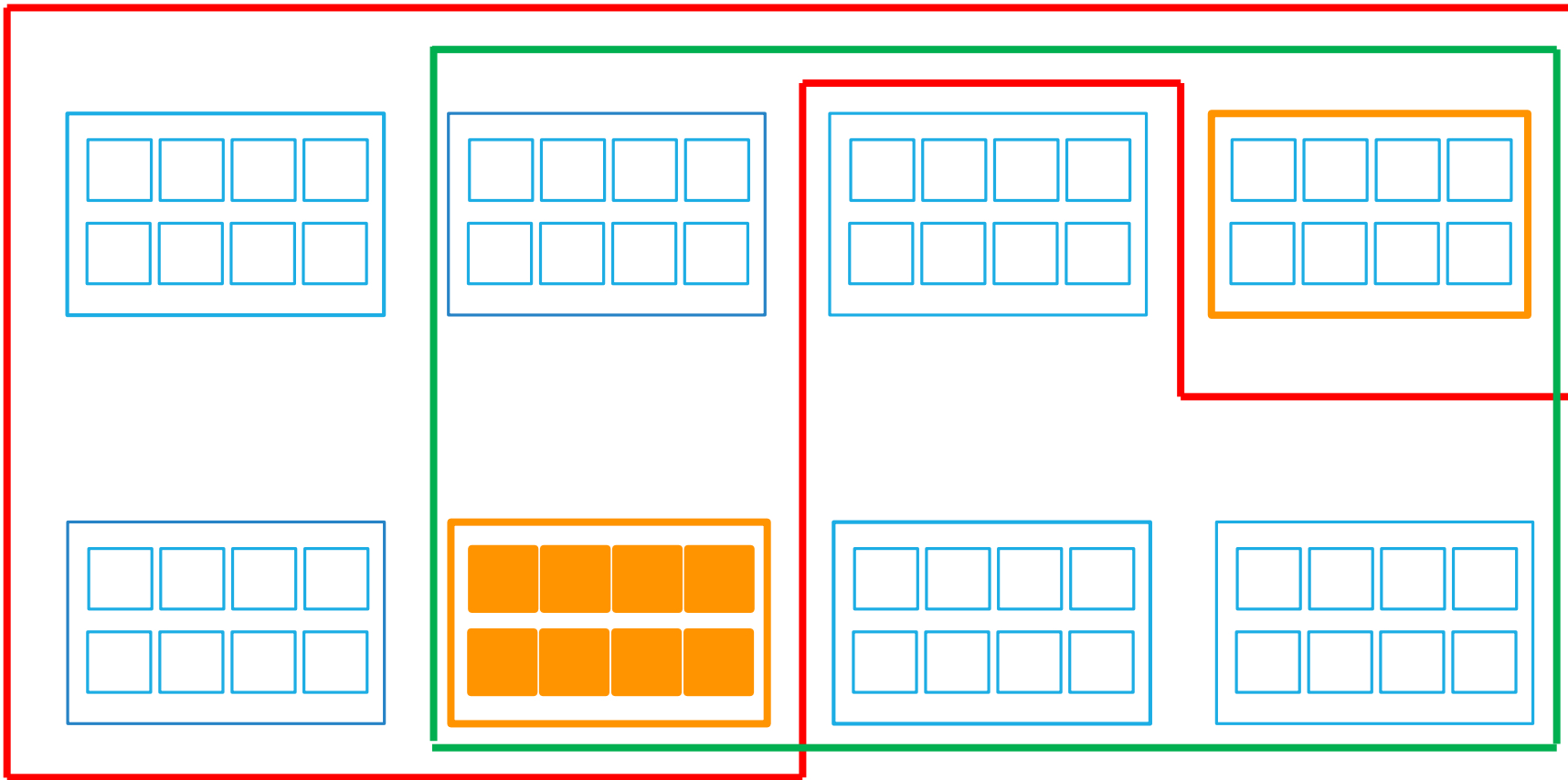
# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



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

**Overloaded node  
in both groups!**

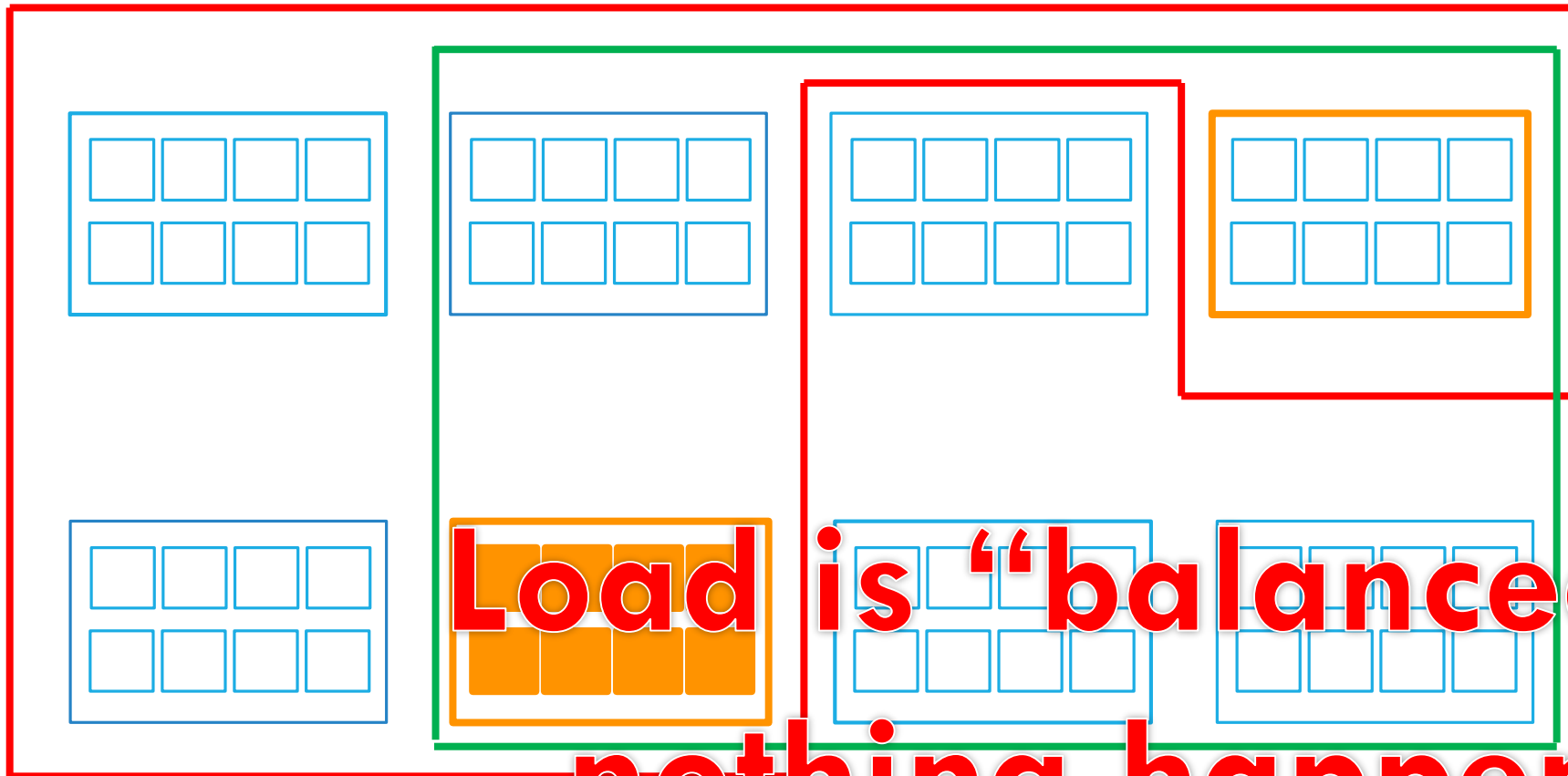
# BUG 2/4: SCHEDULING GROUP CONSTRUCTION



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

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

# BUG 2/4: SCHEDULING GROUP CONSTRUCTION

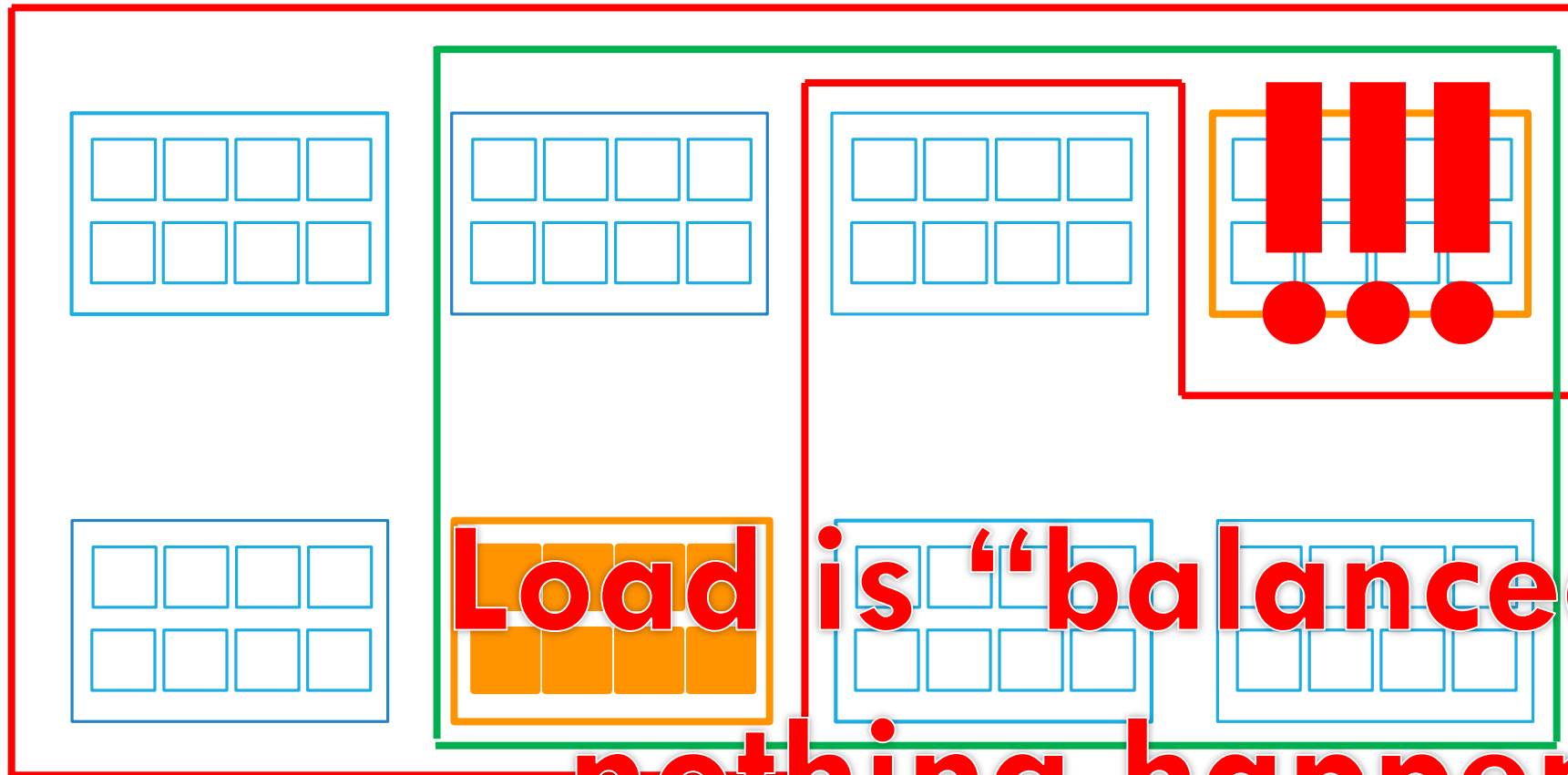


**load(red) =  
16 \* load(thread)**

**load(green) =  
16 \* load(thread)**

**Load is “balanced”:  
nothing happens**

# BUG 2/4: SCHEDULING GROUP CONSTRUCTION

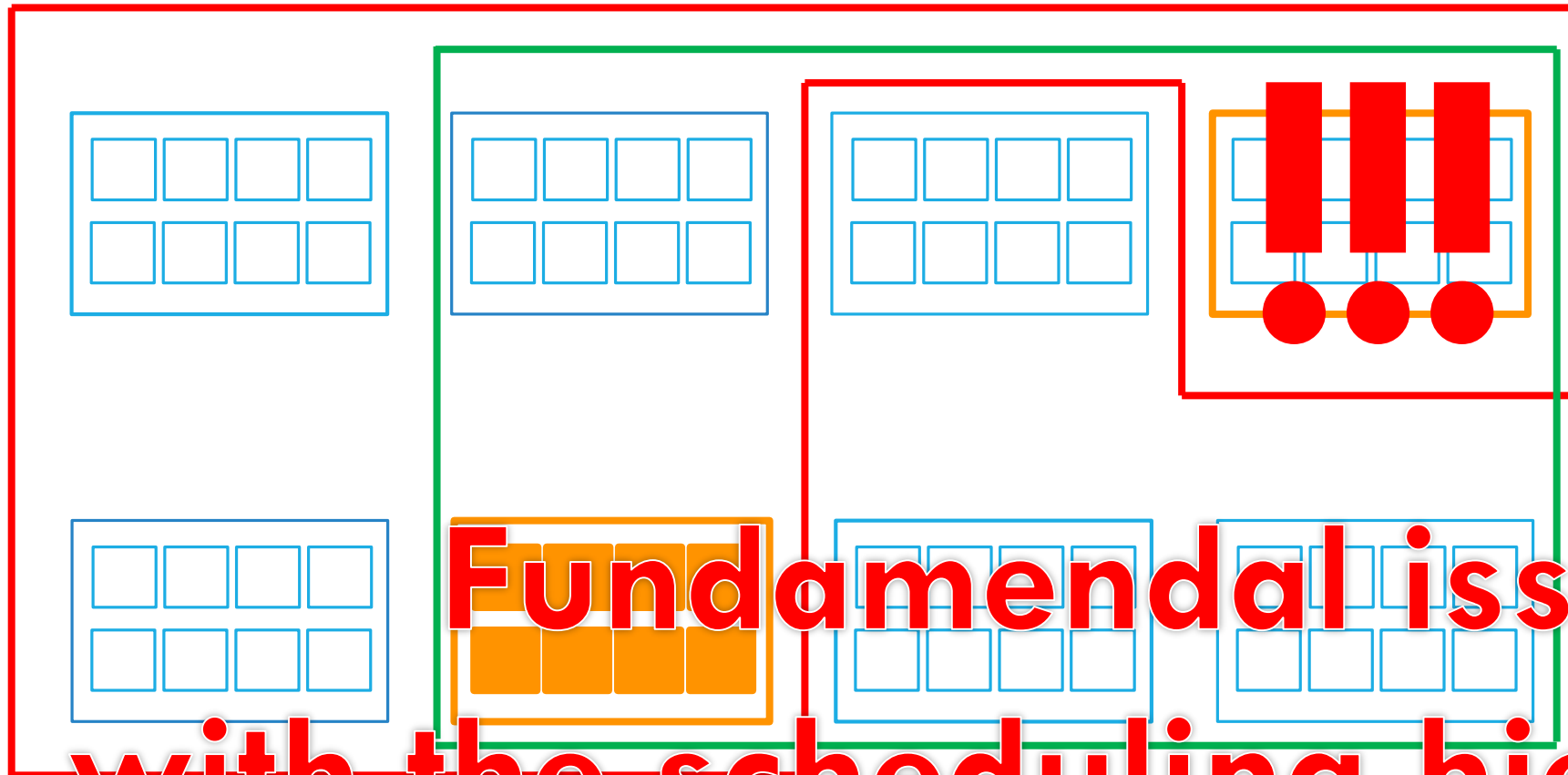


$\text{load}(\text{red}) = 16 * \text{load}(\text{thread})$

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

Load is "balanced"  
nothing happens

# 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 issue**  
**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

# 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

# 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 (convoys)

# BUG 3/4: MISSING SCHEDULING DOMAINS

- In addition to this, when domains re-built, **levels 3 and 4 not re-built...**

# 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)

# 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**

# 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**

# 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)

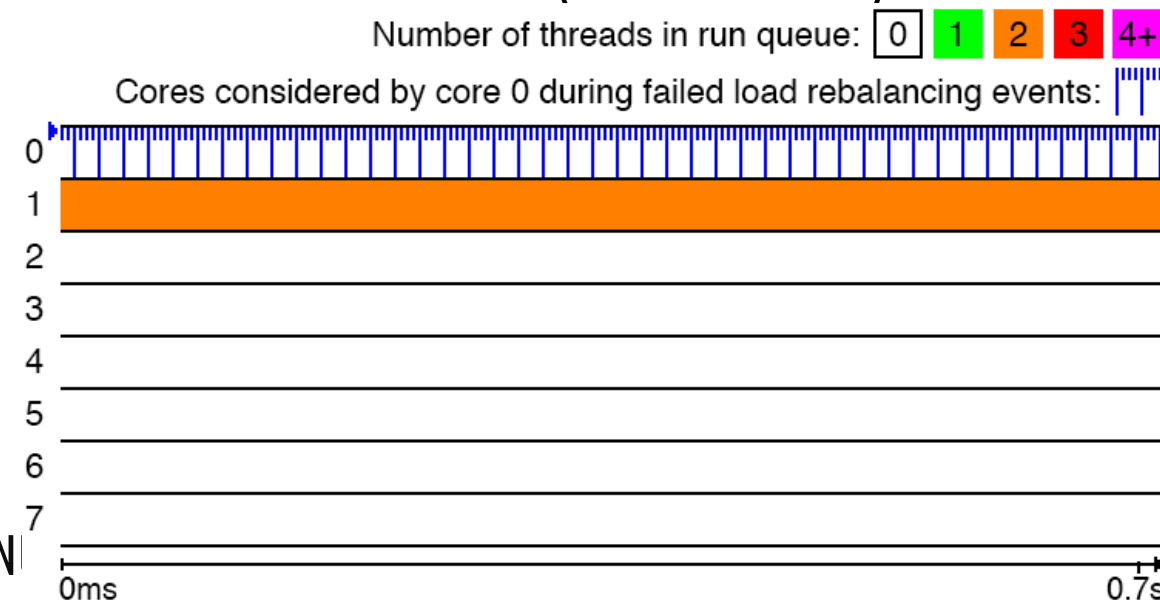


# BUG 3/4: MISSING SCHEDULING DOMAINS

- 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 application will be on CPU 1 forever!

# BUG 3/4: MISSING SCHEDULING DOMAINS

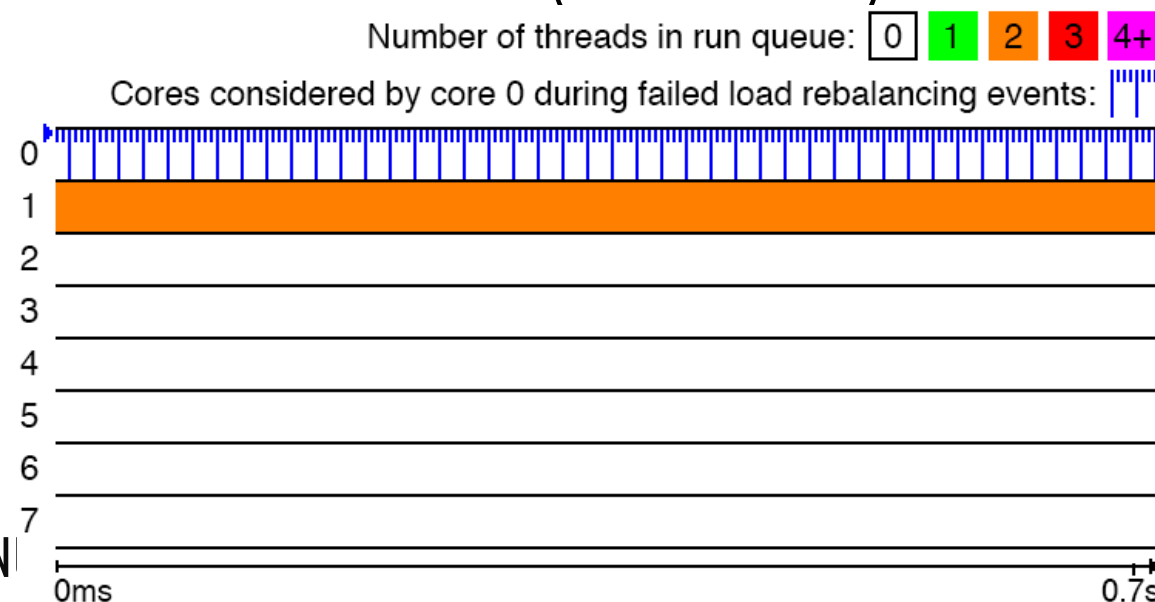
- In addition to this, when domains re-built, **levels 3 and 4 not re-built...**
  - I.e., no balancing between directly connected or 1-hop CPUs (i.e. any CPU)
  - **Happens for instance when disabling and re-enabling a core**
- **Launch an application, first thread created on CPU 1**
  - First thread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



# 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 application 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



# 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

# 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)...

# 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.

# 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...**

# 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.**

# 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!**

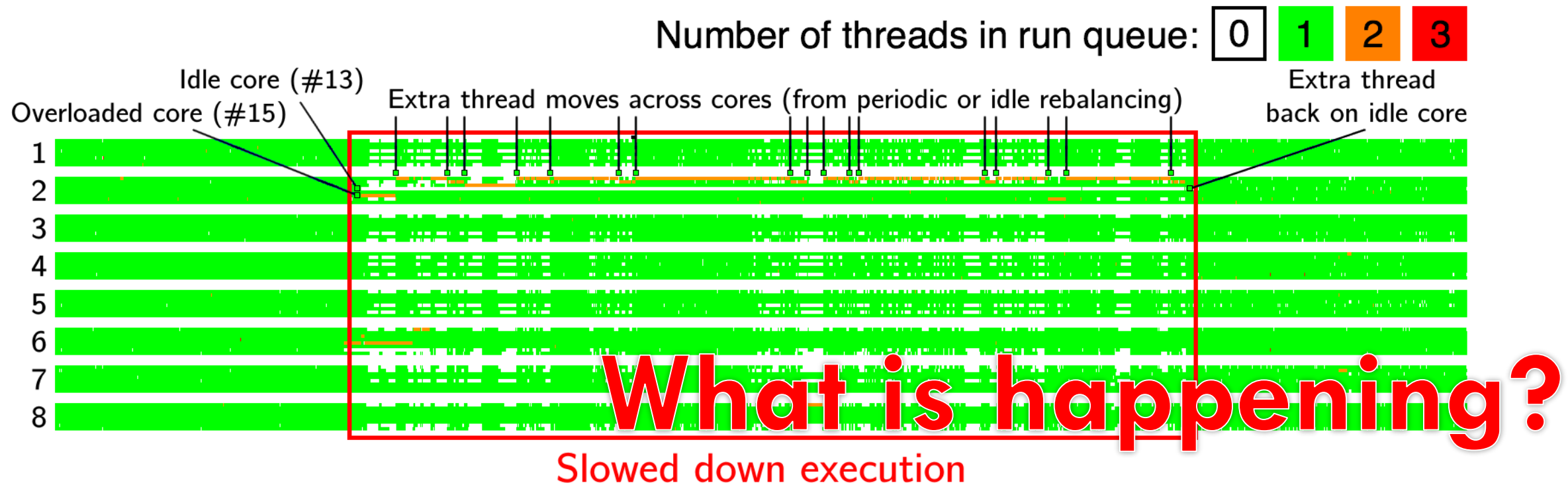
# BUG 4/4: OVERLOAD-ON-WAKEUP

- 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!**

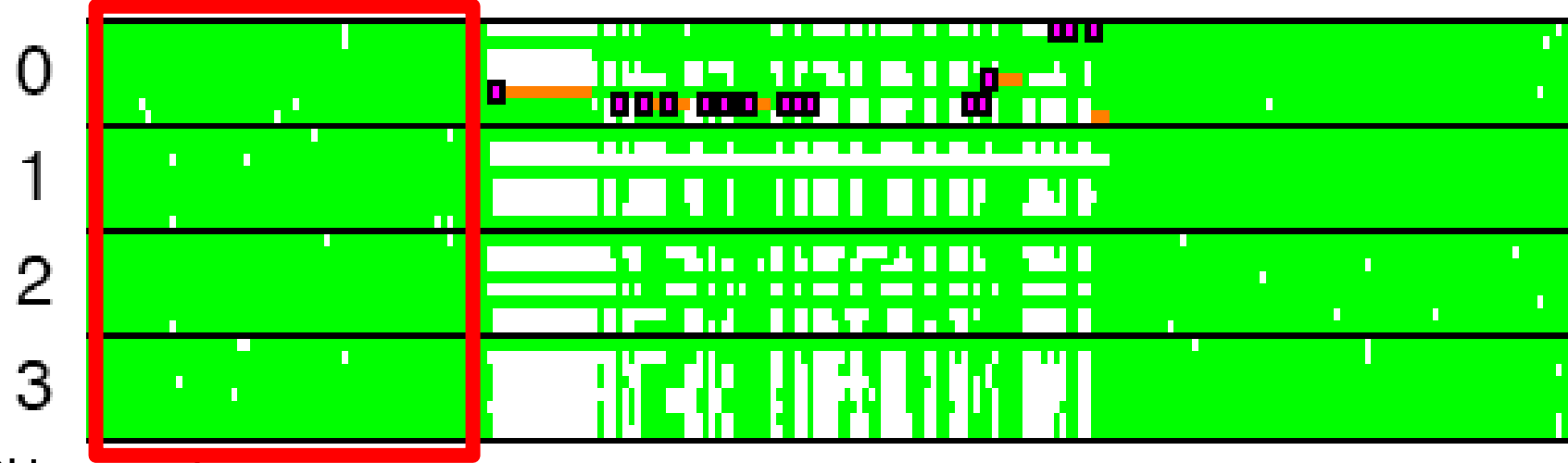


# BUG 4/4: OVERLOAD-ON-WAKEUP

- 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!**

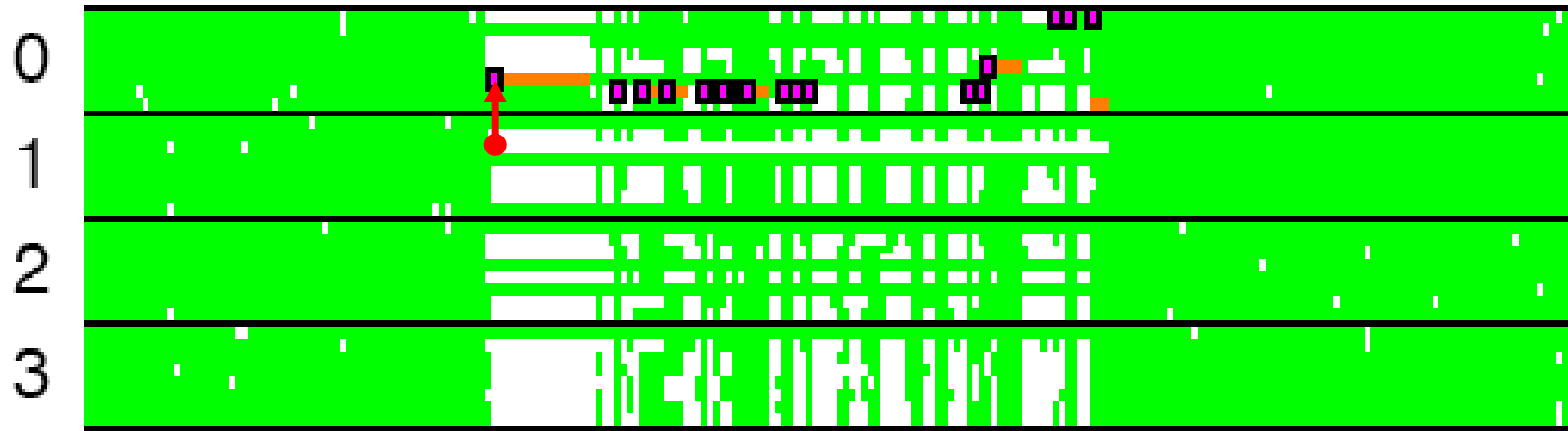


# BUG 4/4



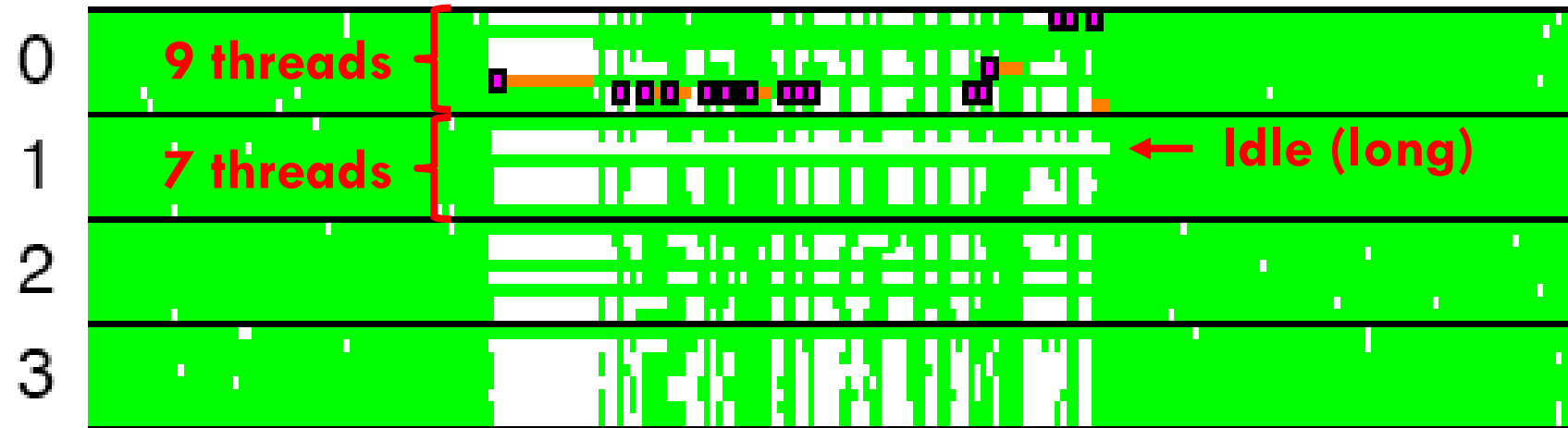
- Beginning: 8 threads / CPU, cores busy

# BUG 4/4



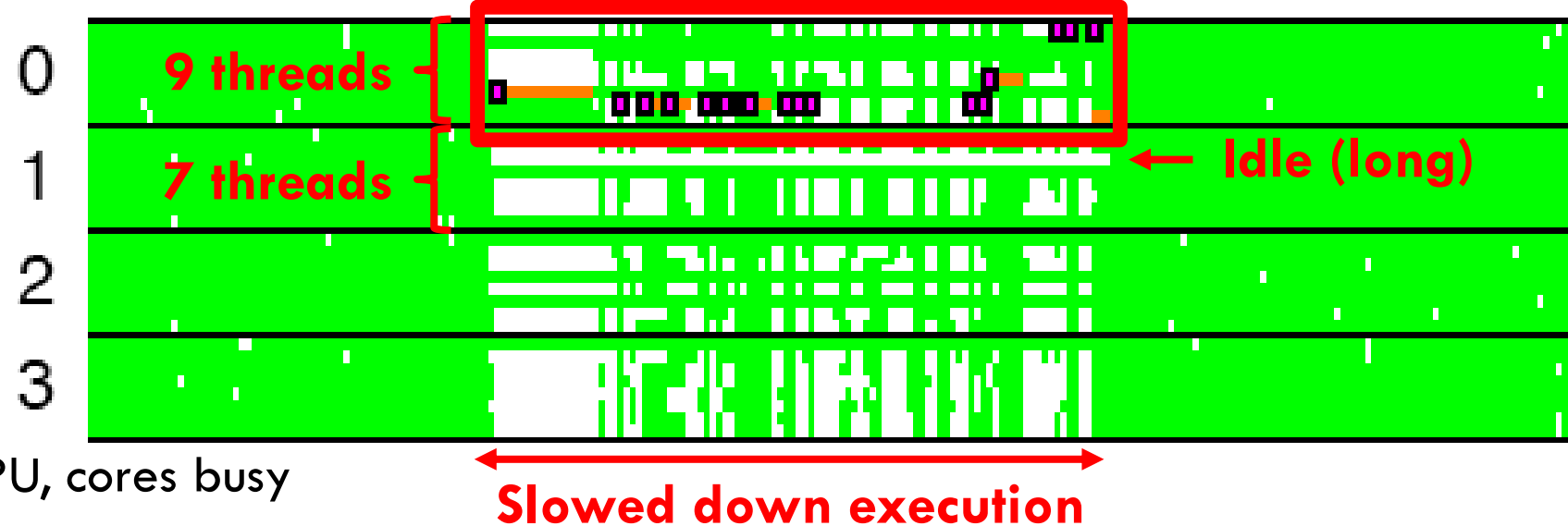
- Beginning: 8 threads / CPU, cores busy
- Occasionally, **1 DB thread migrated to other CPU** because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)

# BUG 4/4



- Beginning: 8 threads / CPU, cores busy
- Occasionally, **1 DB thread migrated to other CPU** because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)
- **Now, 9 threads on one CPU, and 7 on another one.** CPU with 9 threads slow, slows down all execution because all threads wait for each other (barriers), i.e. idle cores everywhere...

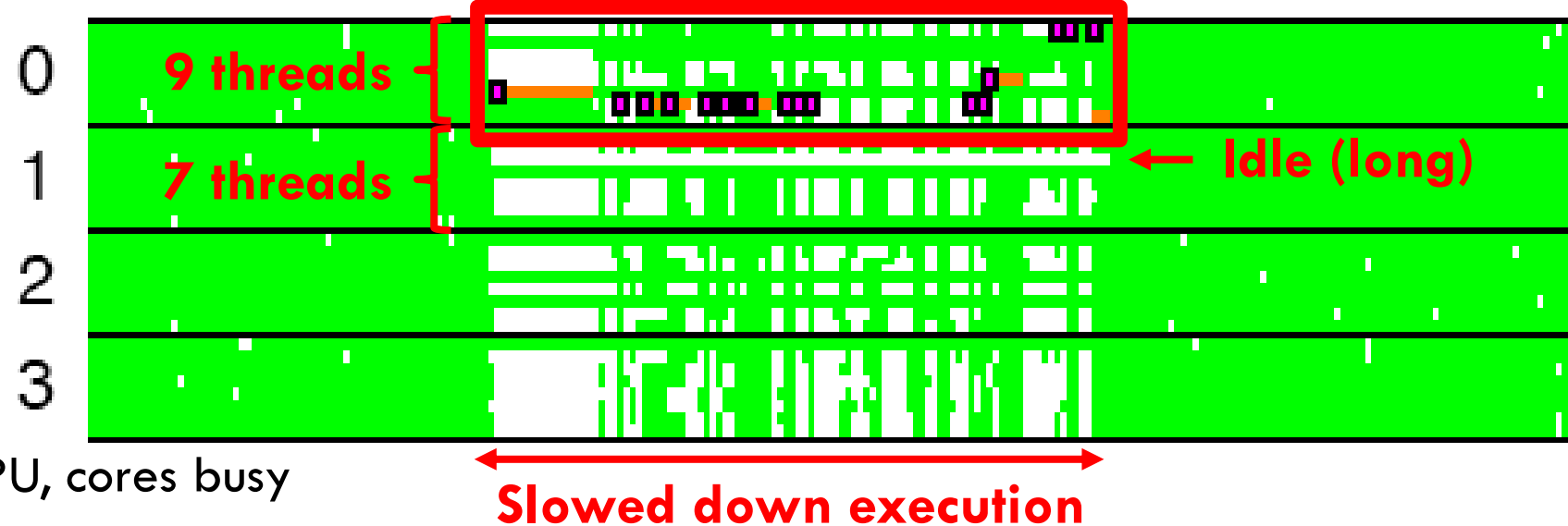
# BUG 4/4



- Beginning: 8 threads / CPU, cores busy
- Occasionally, **1 DB thread migrated to other CPU** because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)
- **Now, 9 threads on one CPU, and 7 on another one.** CPU with 9 threads slow, slows down all execution because all threads wait for each other (barriers), i.e. idle cores everywhere...
- **Barriers: threads keep sleeping and waking up, but extra thread never wakes up on idle core, because waking up algorithm only considers local CPU!**



# BUG 4/4



- Beginning: 8 threads / CPU, cores busy
- Occasionally, **1 DB thread migrated to other CPU** because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)
- **Now, 9 threads on one CPU, and 7 on another one.** CPU with 9 threads slow, slows down all execution because all threads wait for each other (barriers), i.e. idle cores everywhere...
- **Barriers: threads keep sleeping and waking up, but extra thread never wakes 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)

# BUG 4/4



- Begin
- Occa during
- Now, all exe
- Barrier

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

# 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*,

# 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*,  
↑ **Fundamental issue here**

# 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*,
  - ↑ **Fundamental issue here**
  - to periodically balance threads between *scheduling domains*.

# 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*,  
↑ **Fundamental issue here**
  - to periodically balance threads between *scheduling domains*.  
↑ **Fundamental issue here**

# 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*,  
↑ **Fundamental issue here**
  - to periodically balance threads between *scheduling domains*.  
↑ **Fundamental issue here**
  - In addition to this, threads balance load by *selecting core where to wake up*.



# 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*,  
↑ **Fundamental issue here**
  - to periodically balance threads between *scheduling domains*.  
↑ **Fundamental issue here**
  - In addition to this, threads balance load by *selecting core where to wake up*.  
↑ **Fundamental issue here**

# 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*,  
↑ **Fundamental issue here**
  - to periodically balance threads between *scheduling domains*.  
↑ **Fundamental issue here**
  - In addition to this, threads balance load by *selecting core where to wake up*.  
↑ **Fundamental issue here**

Wait, does anything work 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?**

# 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

# 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

# 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?

- A pragmatic “solution”: can’t prevent bugs, let’s detect them with a *sanity checker*

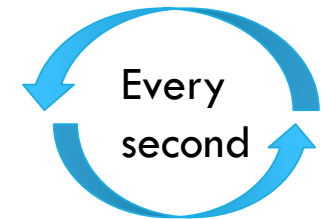
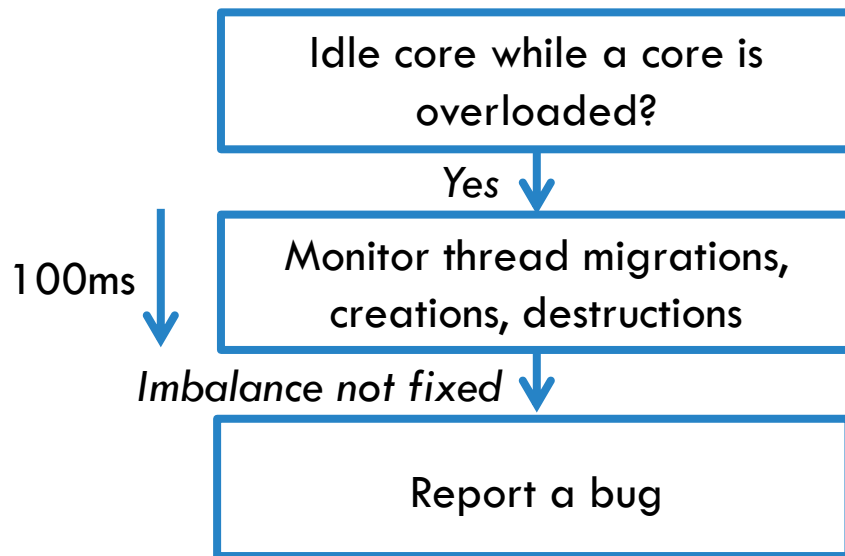
# WHERE DO WE GO FROM HERE?

- A pragmatic “solution”: can’t prevent bugs, let’s detect them with a *sanity checker*
- Always same symptom: some idle cores while others overloaded



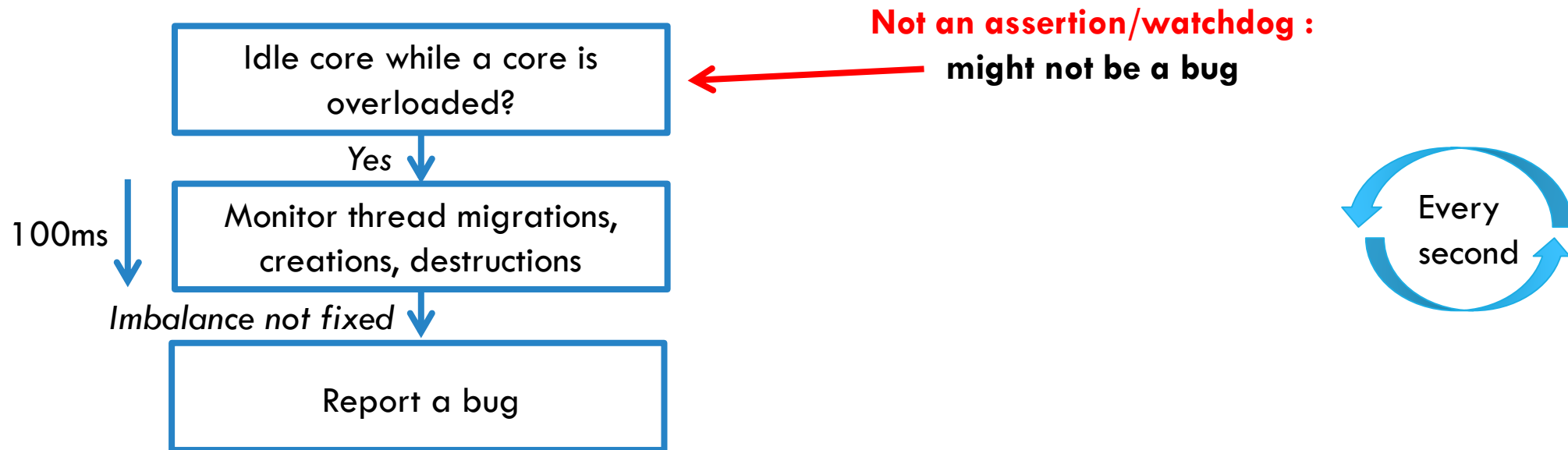
# WHERE DO WE GO FROM HERE?

- A pragmatic “solution”: can’t prevent bugs, let’s detect them with a *sanity checker*
- **Always same symptom: some idle cores while others overloaded**



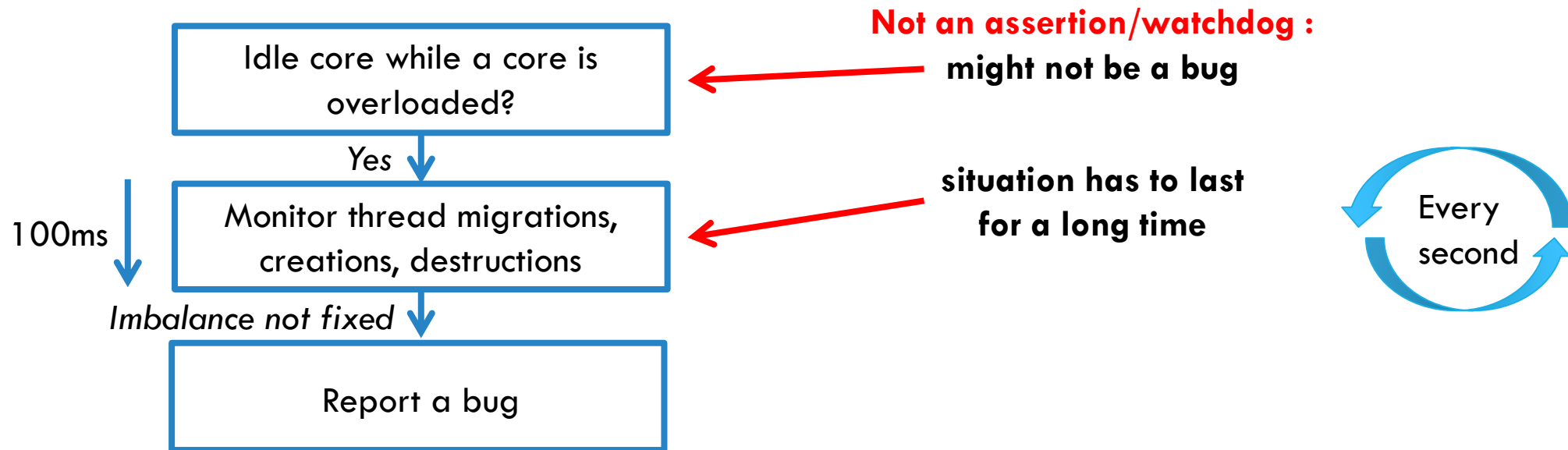
# WHERE DO WE GO FROM HERE?

- A pragmatic “solution”: can’t prevent bugs, let’s detect them with a *sanity checker*
- **Always same symptom: some idle cores while others overloaded**



# WHERE DO WE GO FROM HERE?

- A pragmatic “solution”: can’t prevent bugs, let’s detect them with a *sanity checker*
- **Always same symptom: some idle cores while others overloaded**



# WHERE DO WE GO FROM HERE?

- **We might miss some bugs.** Not an issue, bugs that impact performance happen often

# WHERE DO WE GO FROM HERE?

- **We might miss some bugs.** Not an issue, bugs that impact performance happen often
  - **We'll eventually catch them**

# WHERE DO WE GO FROM HERE?

- **We might miss some bugs.** Not an issue, bugs that impact performance happen often
  - **We'll eventually catch them**
- Low overhead, possible to reduce period (will just take longer to detect bugs)

# WHERE DO WE GO FROM HERE?

- **We might miss some bugs.** Not an issue, bugs that impact performance happen often
  - **We'll eventually catch them**
- Low overhead, possible to reduce period (will just take longer to detect bugs)
- All bugs presented here detected with sanity checker

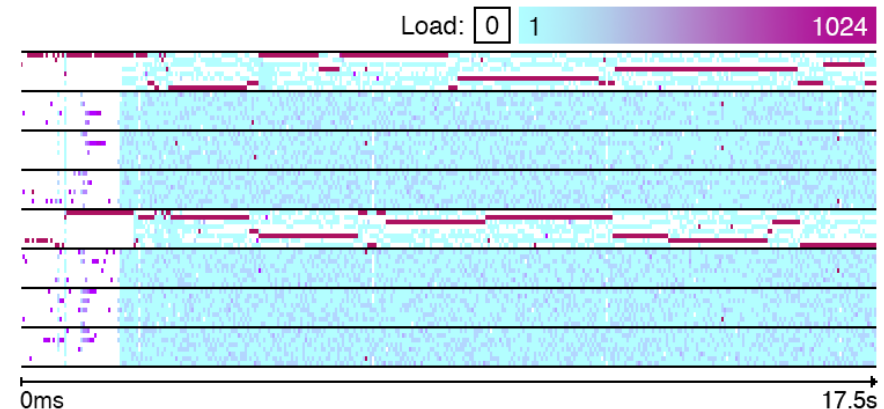
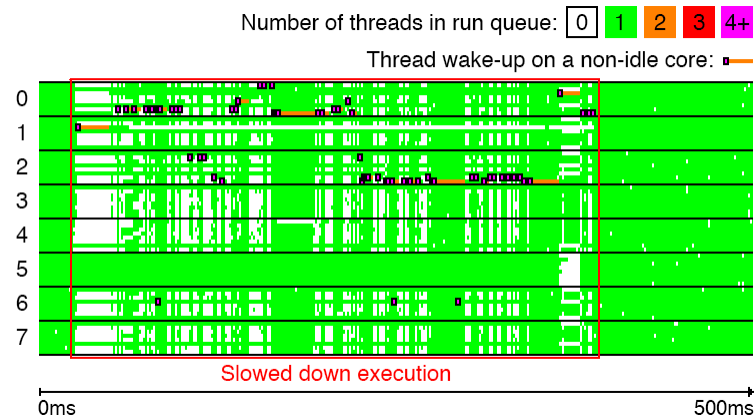
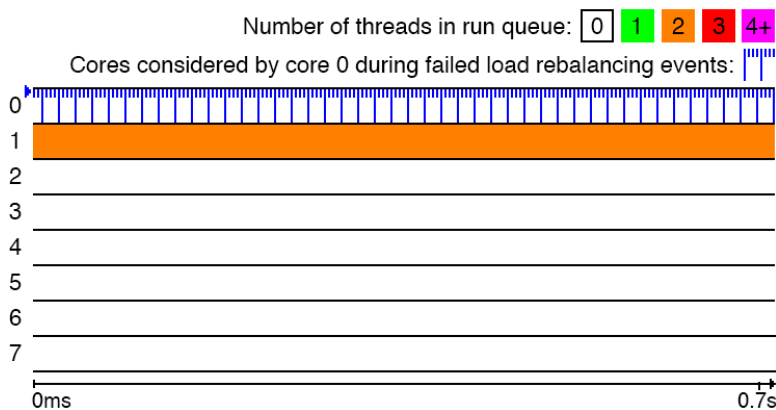
# WHERE DO WE GO FROM HERE?

- **We might miss some bugs.** Not an issue, bugs that impact performance happen often
  - **We'll eventually catch them**
- Low overhead, possible to reduce period (will just take longer to detect bugs)
- All bugs presented here detected with sanity checker
- **Possible to replay bugs, and produce graphical traces to understand them better**



# WHERE DO WE GO FROM HERE?

- **We might miss some bugs.** Not an issue, bugs that impact performance happen often
  - **We'll eventually catch them**
- Low overhead, possible to reduce period (will just take longer to detect bugs)
- All bugs presented here detected with sanity checker
- **Possible to replay bugs, and produce graphical traces to understand them better**



# CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.

# 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...

# 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

# 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**

# 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

# 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
- **Proposed pragmatic detection approach (“sanity checker”):** helpful

# 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
- **Proposed pragmatic detection approach (“sanity checker”):** helpful
- **Code testing, performance regression, model checking / proofs:** can't work for now.



# 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
- **Proposed pragmatic detection approach ("sanity checker"):** helpful
- **Code testing, performance regression, model checking / proofs:** can't work for now.

■ **Our takeaway:** *more research must be directed towards implementing an efficient and reliable scheduler for multicore architectures!*

# 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
- **Proposed pragmatic detection approach ("sanity checker"):** helpful
- **Code testing, performance regression, model checking / proofs:** can't work for now.

■ **Our takeaway:** *more research must be directed towards implementing an efficient and reliable scheduler for multicore architectures!*

**Your turn!**