Shared-nothing Architecture

Rust Prague Meetup

March 2024

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

Victor Ciura
Principal Engineer
M365 Substrate

About me

Advanced Installer

Clang Power Tools

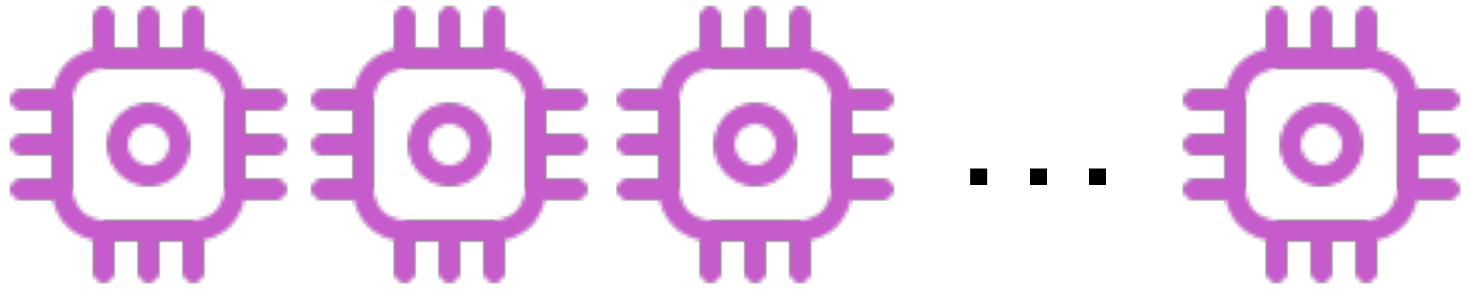
Visual C++

M365 Substrate

🐦 @ciura_victor
🐘 @ciura_victor@hachyderm.io
🦋 @ciuravictor.bsky.social

Context

- Servers have 100+ cores now, with several NUMA regions



- We want to minimize latency & total cores used

- Typical services are I/O-bound

Execution Model

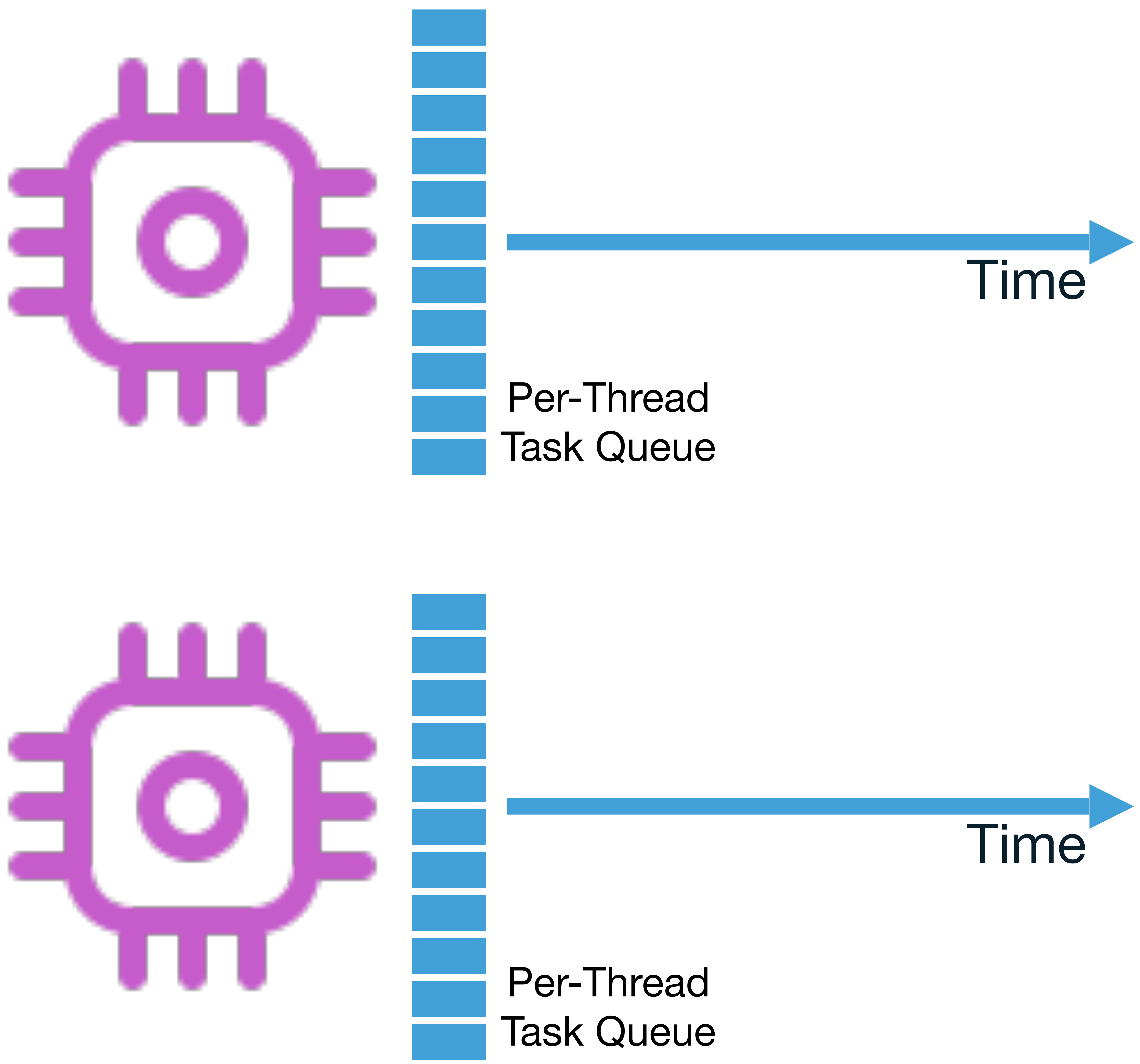
Goals:

- Define how application code runs in a Rust process
- High performance
- Simple to understand
- Correct by construction
- Easy to compose (ergonomic)

Execution Model

Model:

- Cooperative multitasking
- One OS thread pinned to each core
- No preemptive context switching (no locks!)
- Async operations produce tasks which are added to a per-thread queue



Execution Model

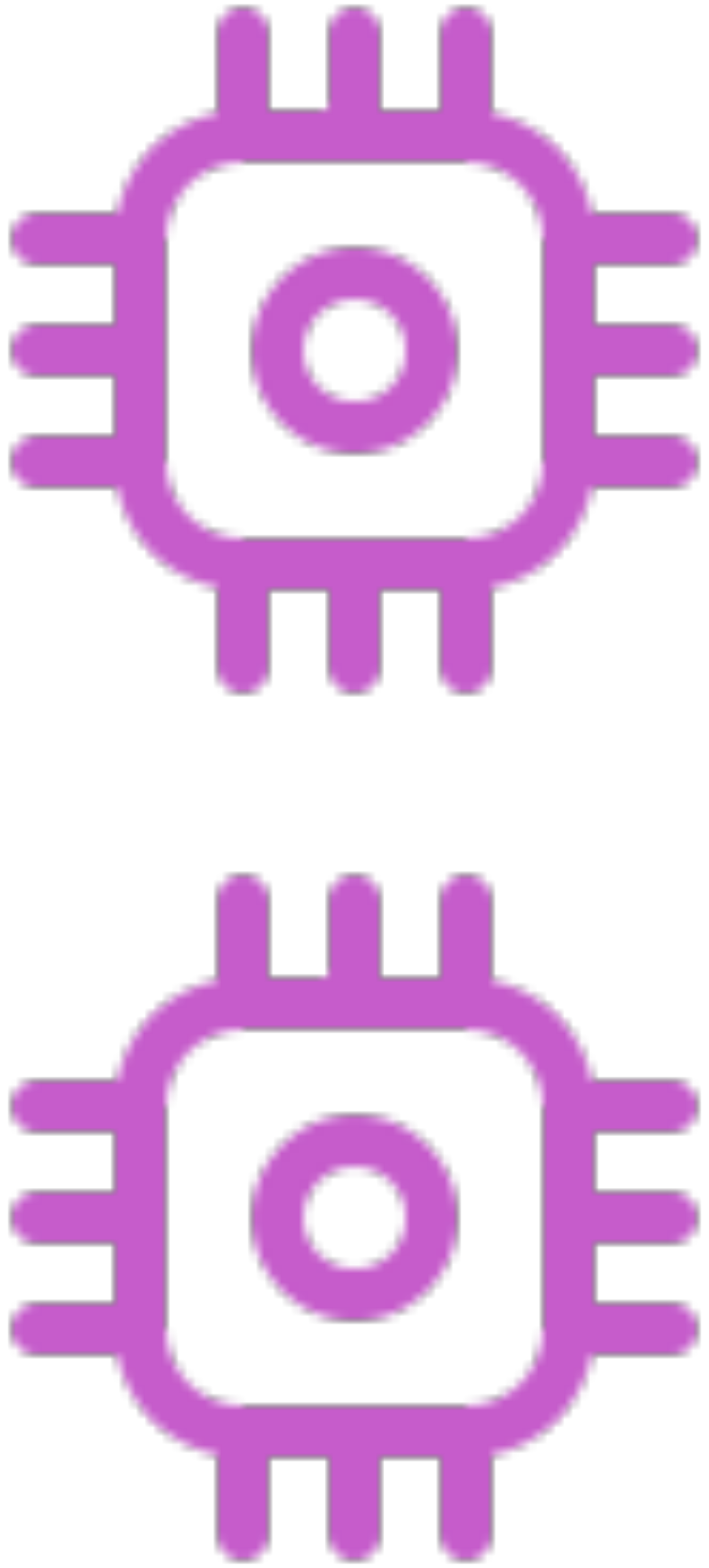
🙂 Benefits:

- Faster serial performance
- Near-linear scaling
- Reduced tail latency for I/O-heavy operations

😕 Problems:

- Compute-intensive tasks delay processing of queued tasks
- Can leave some cores idle while other cores are overcommitted

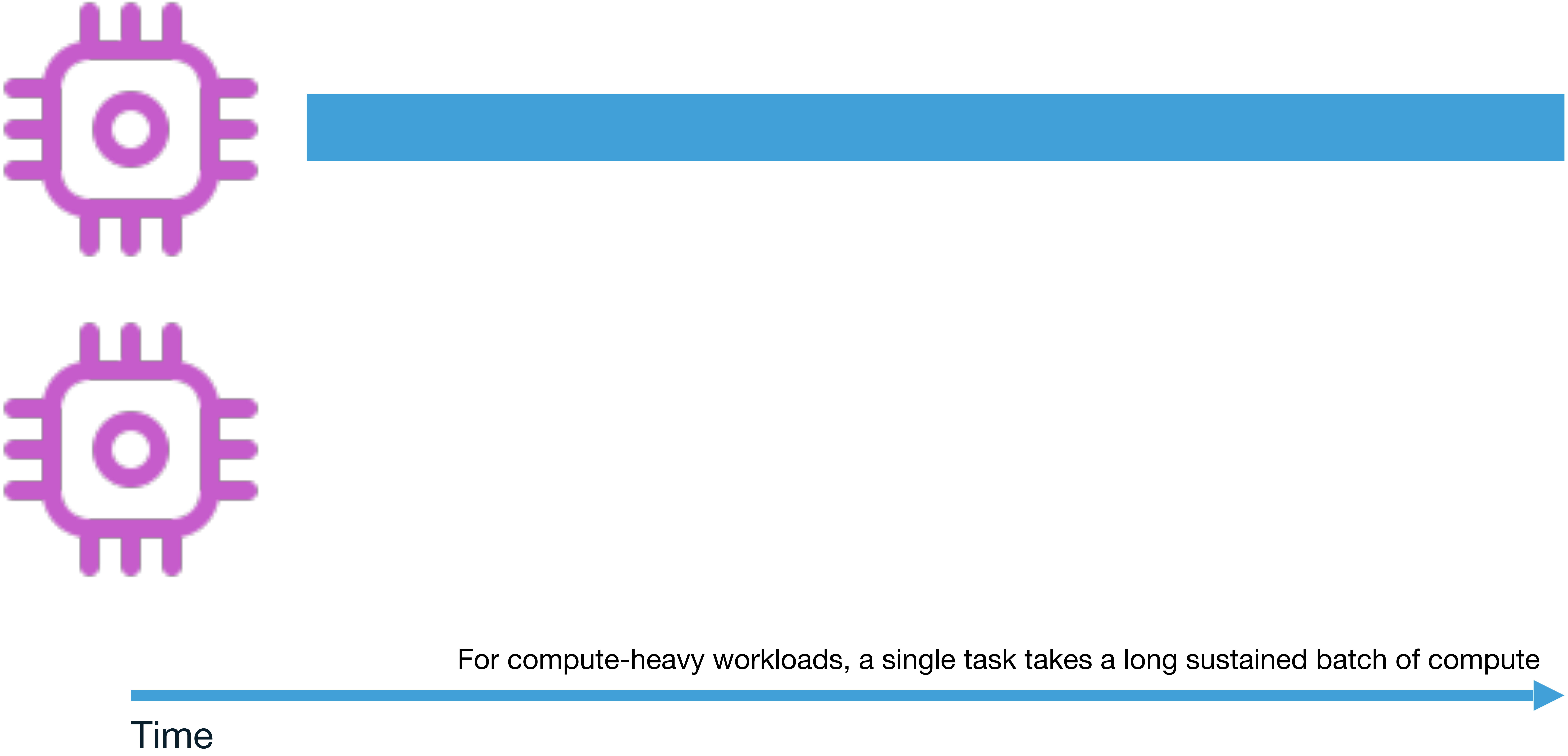
Compute-heavy workloads



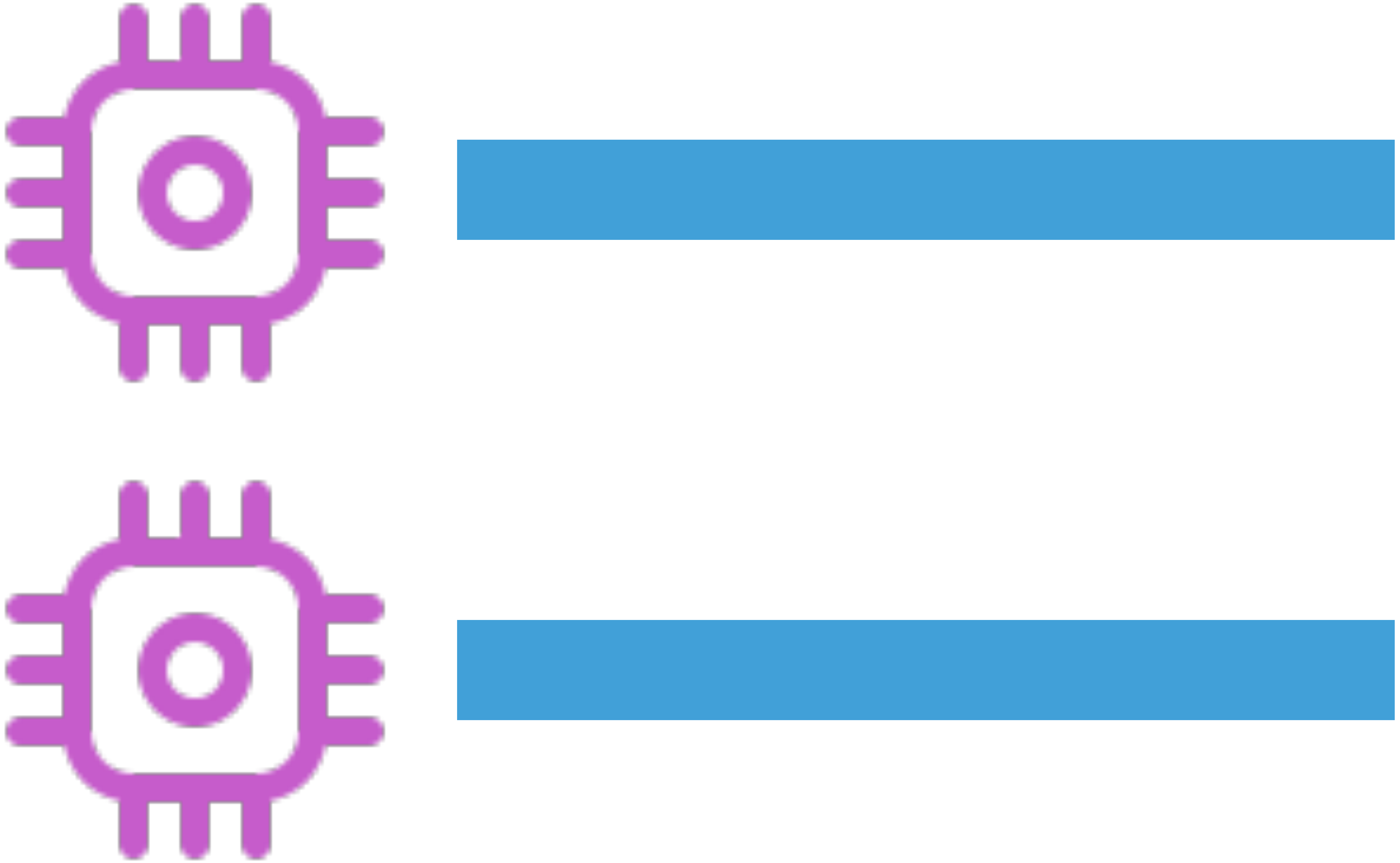
For compute-heavy workloads, a single task takes a long sustained batch of compute

Time

Compute-heavy workloads



Compute-heavy workloads

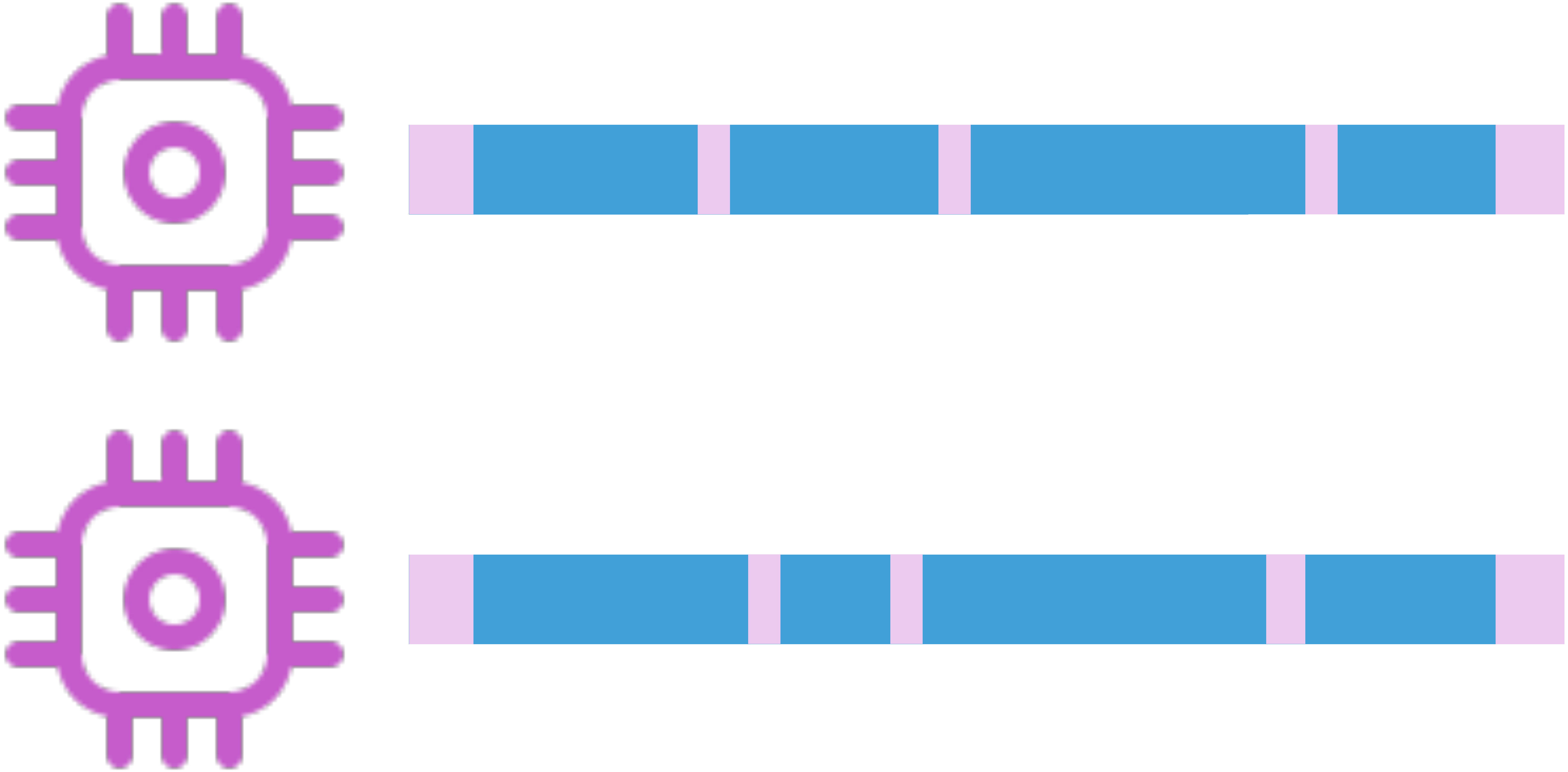


+ extra CPUs, you can cut that task length dramatically by splitting it across CPUs

Time

(but you get some synchronization overhead)

Compute-heavy workloads

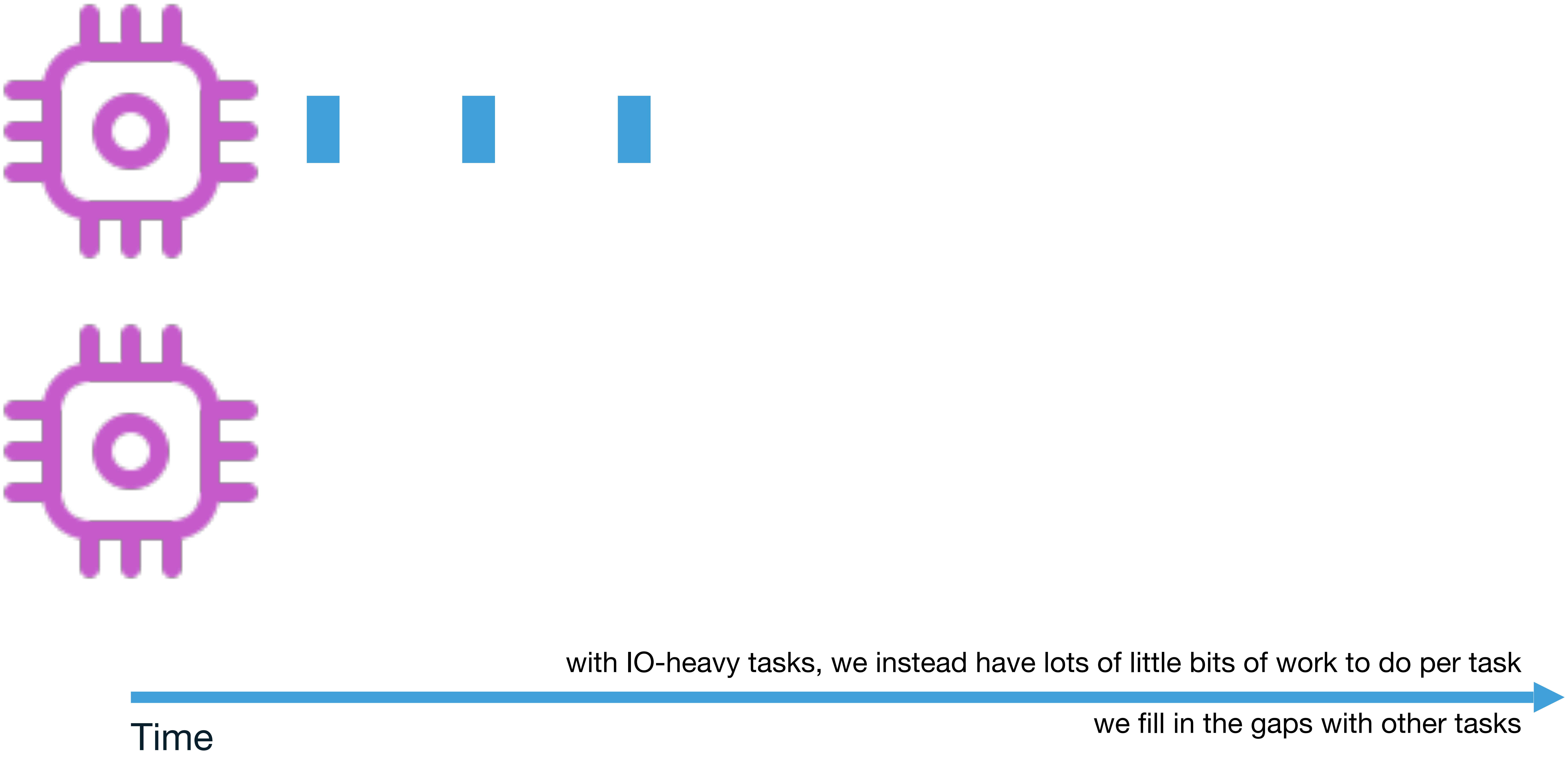


+ extra CPUs, you can cut that task length dramatically by splitting it across CPUs

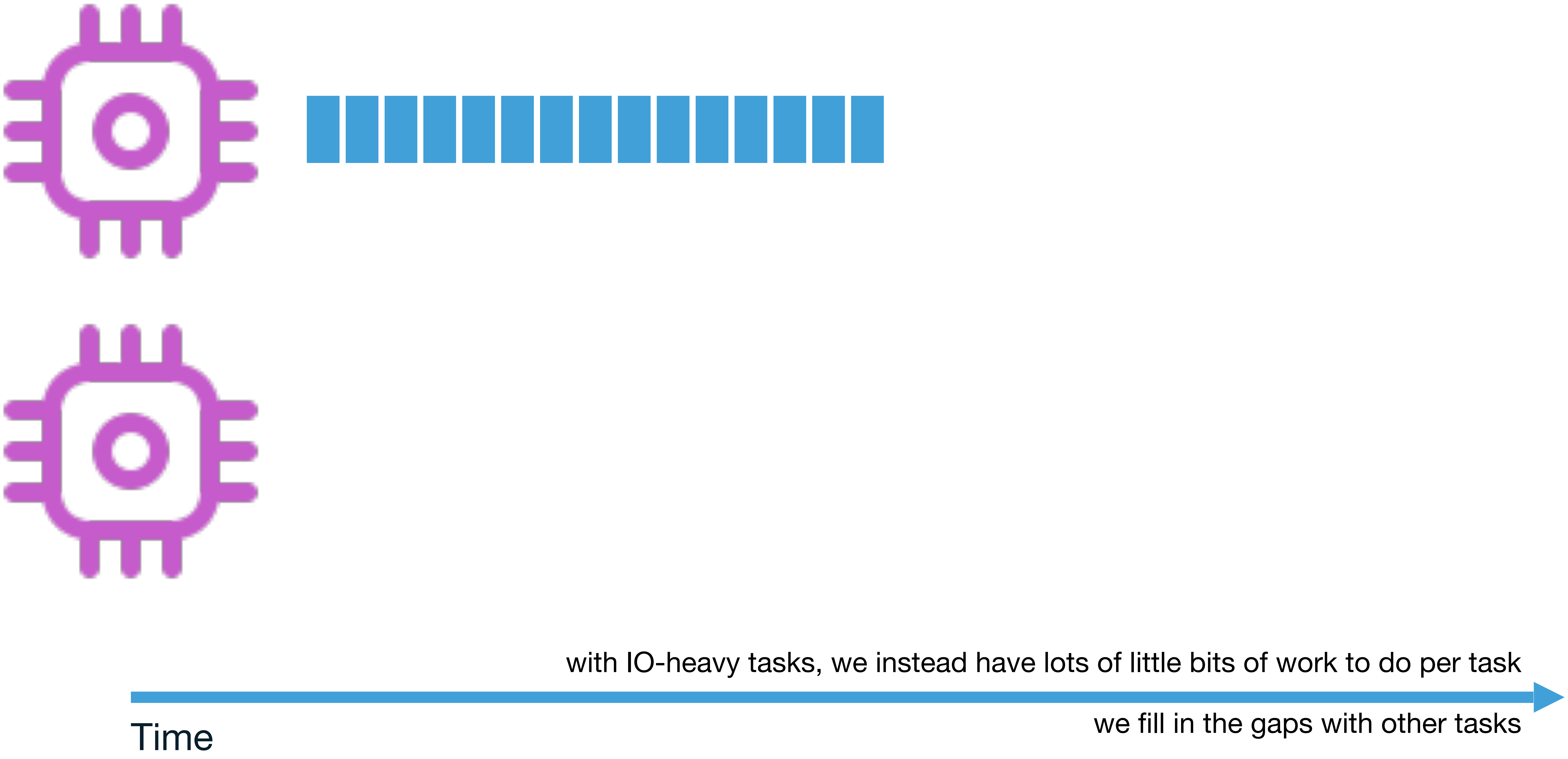
Time

(but you get some synchronization overhead)

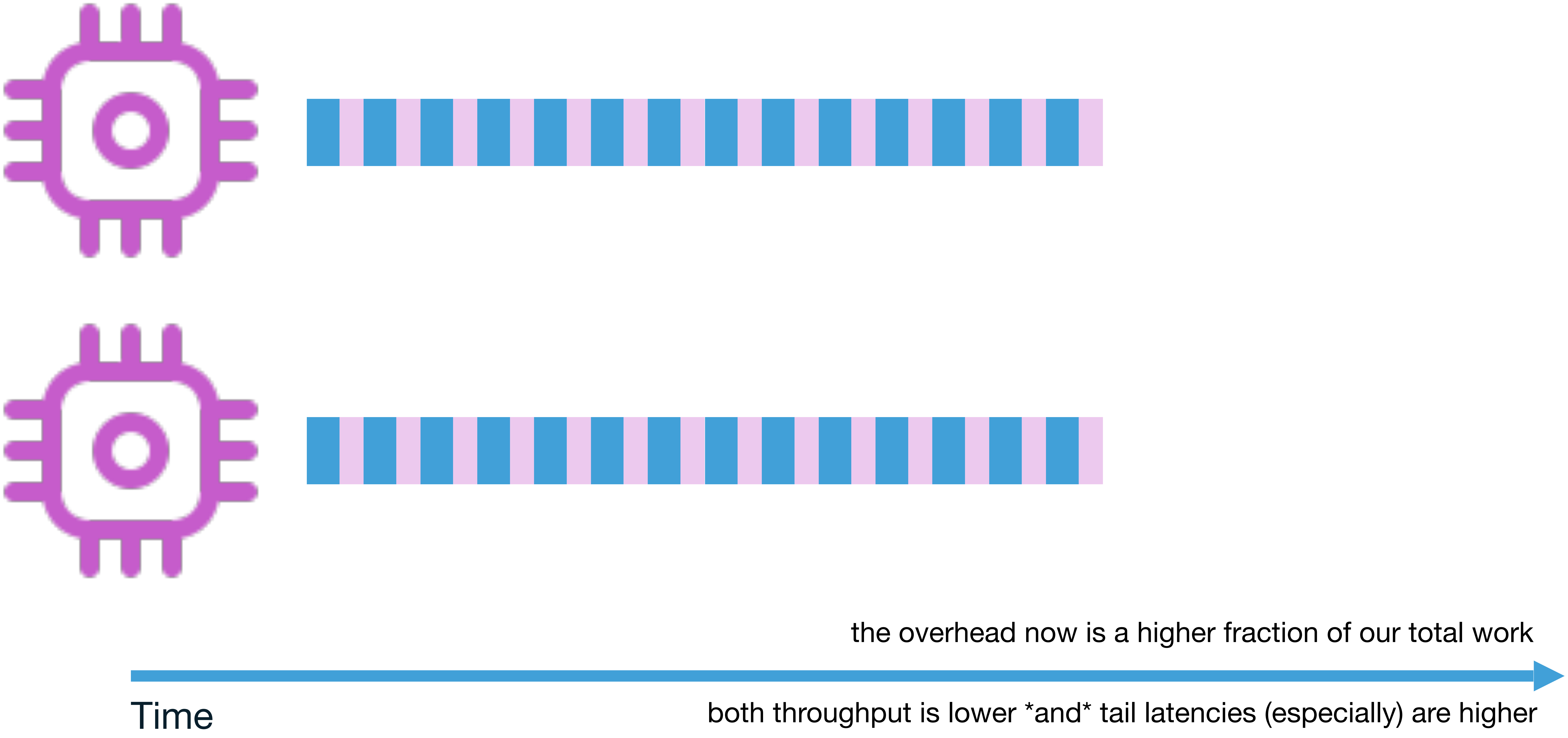
I/O-heavy workloads



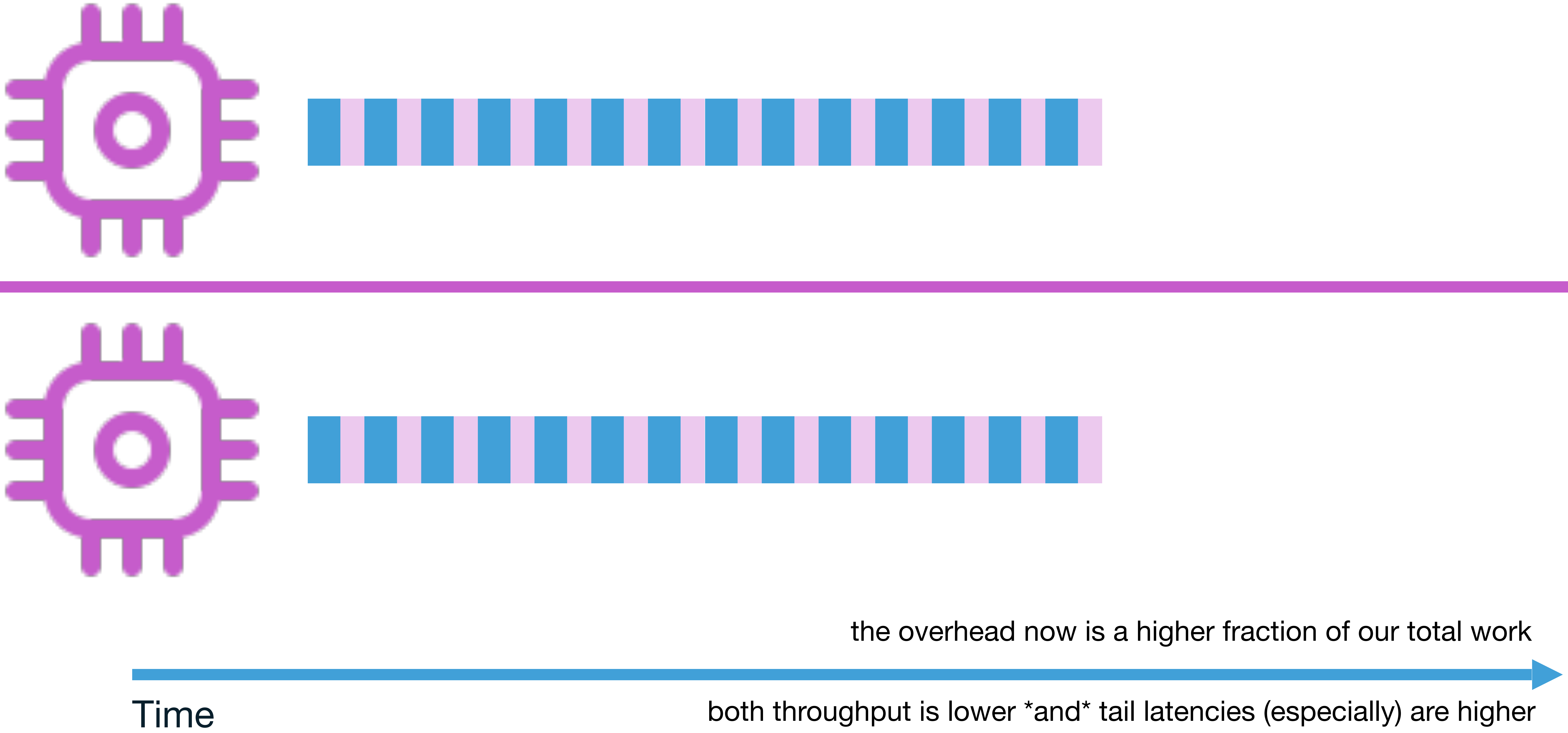
I/O-heavy workloads



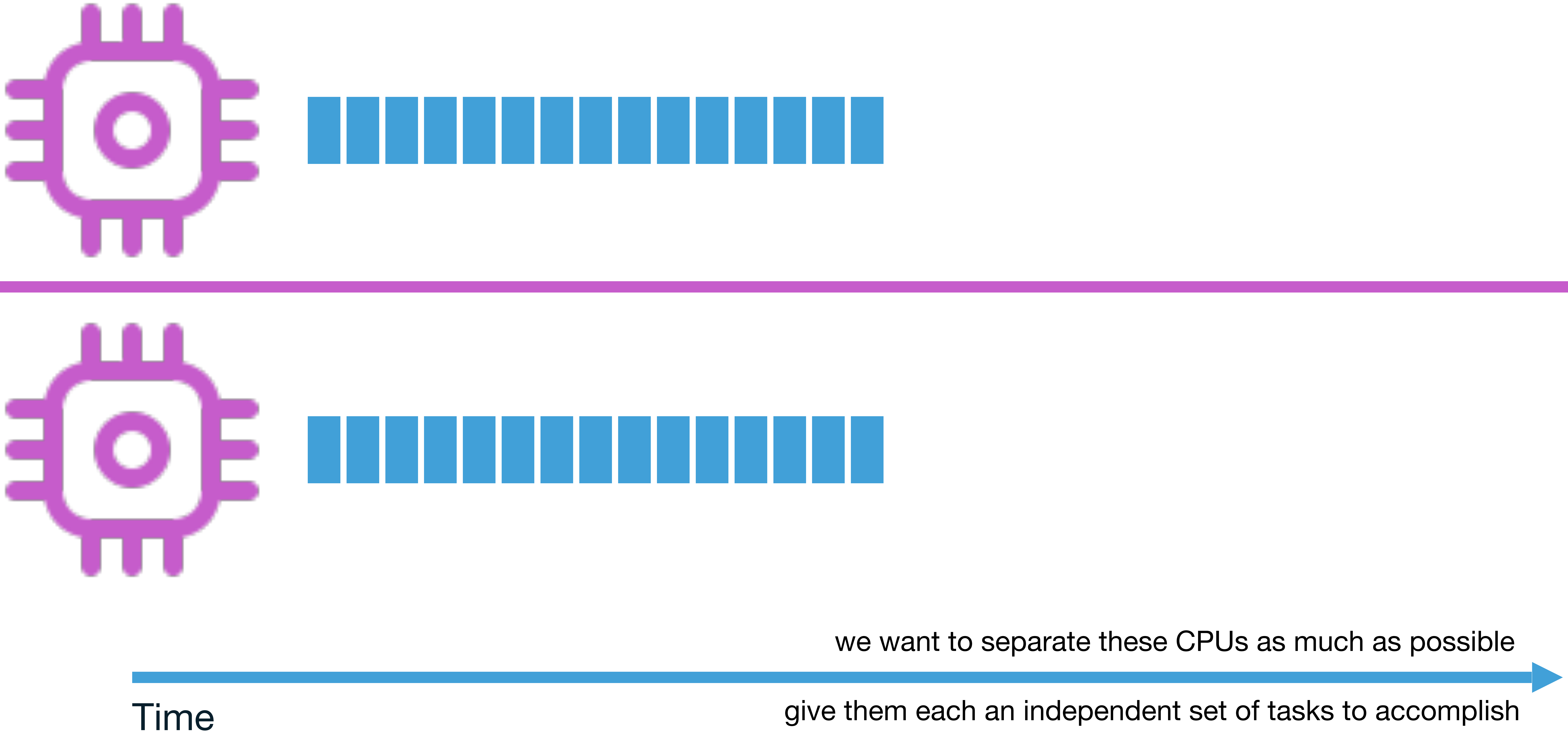
I/O-heavy workloads



I/O-heavy workloads



“Shared nothing” ideal



Shared nothing

"Shared nothing" is well supported in Rust:

- stack allocations
- move by default, rather than aliasing
- deep immutability
- whole-part semantics for structs
- explicit references
- explicit copy/clone
- deterministic destruction

Sometimes you need to share, though...

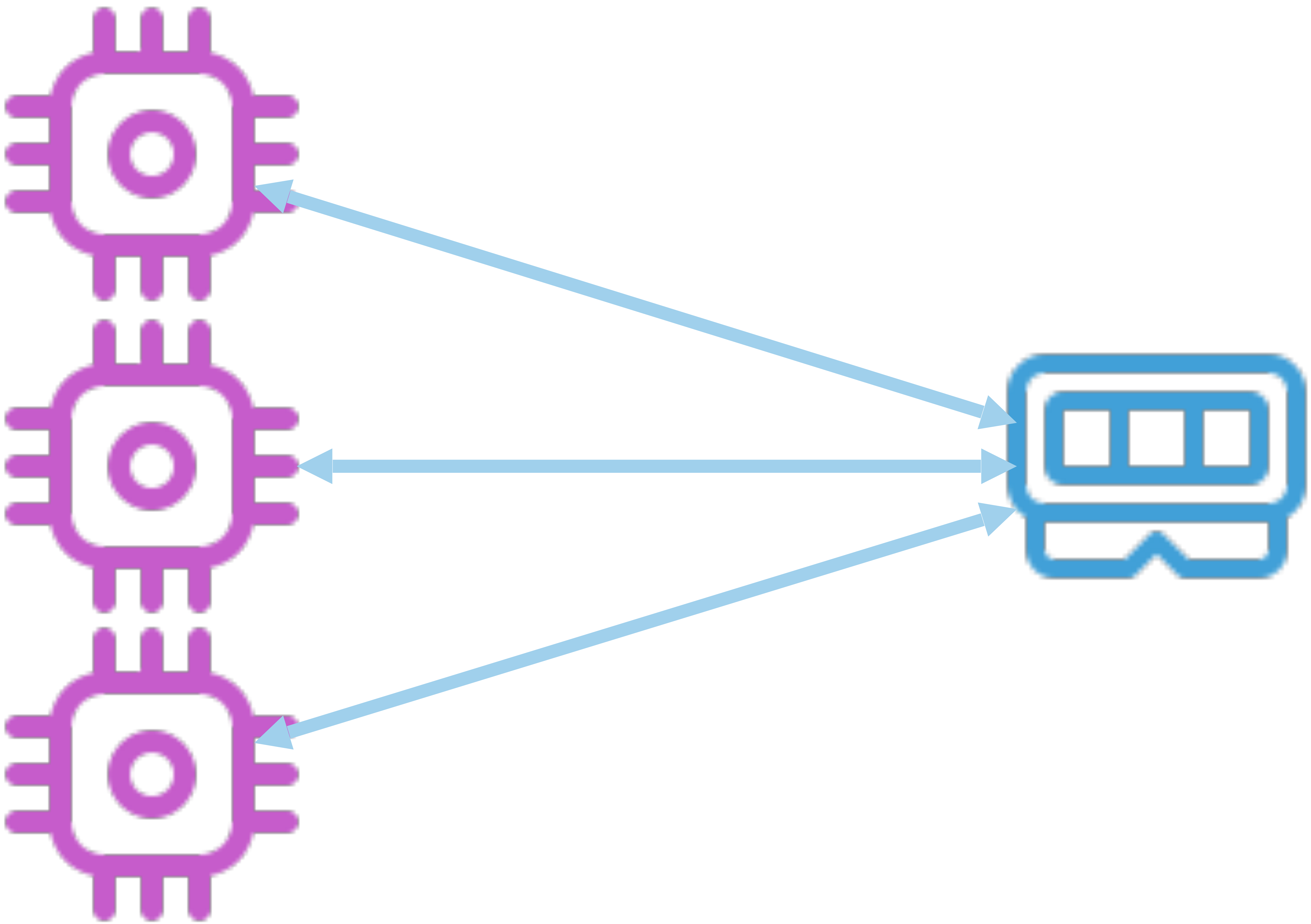
- this needs to be deliberate, rather than accidental
- explicit language constructs
- specialized data structures/policies

We want to support sharing between cores with as low overhead as possible,

- requires exploring how memory is structured on these big NUMA machines

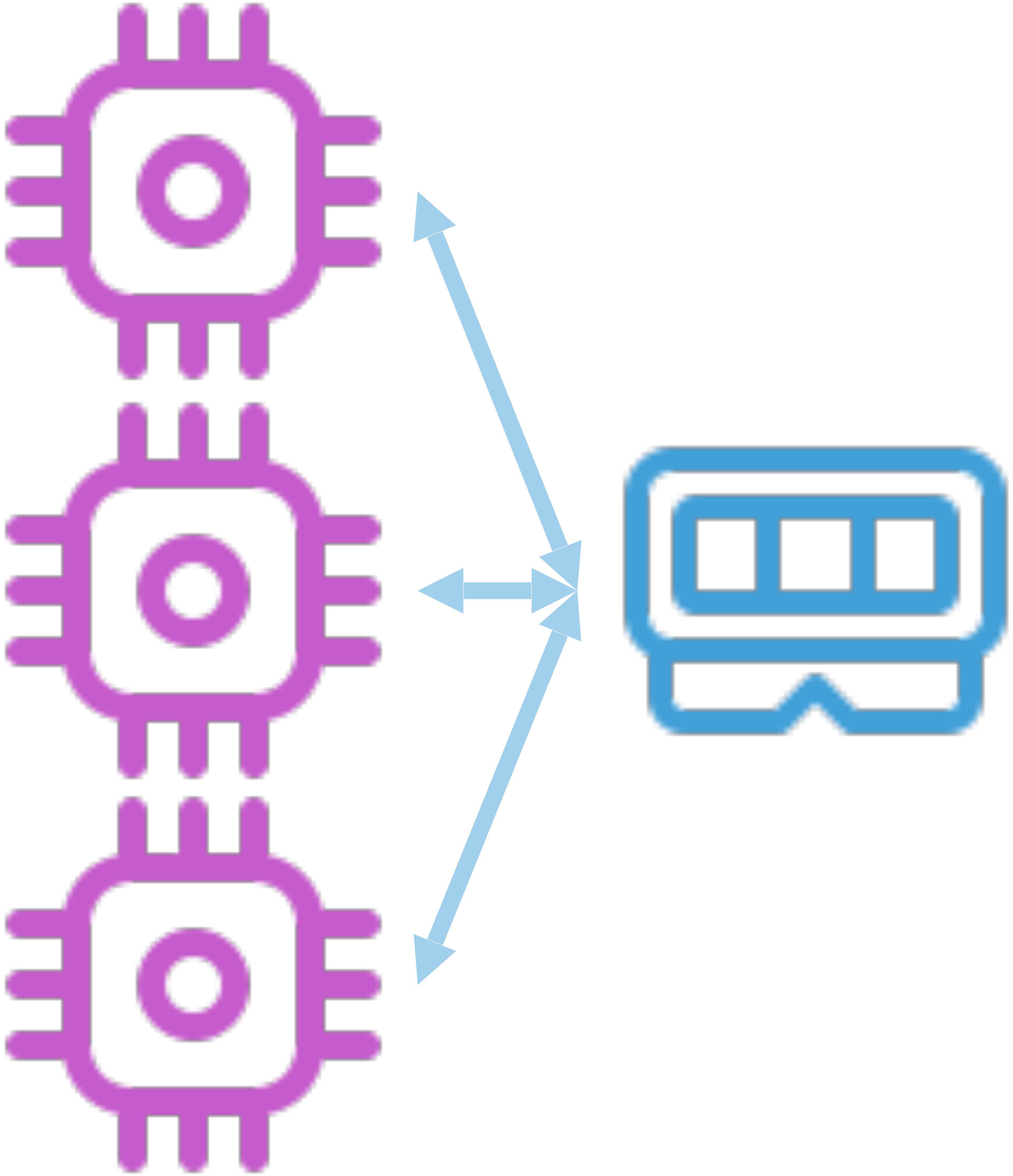
Uniform memory access

Uniform memory access

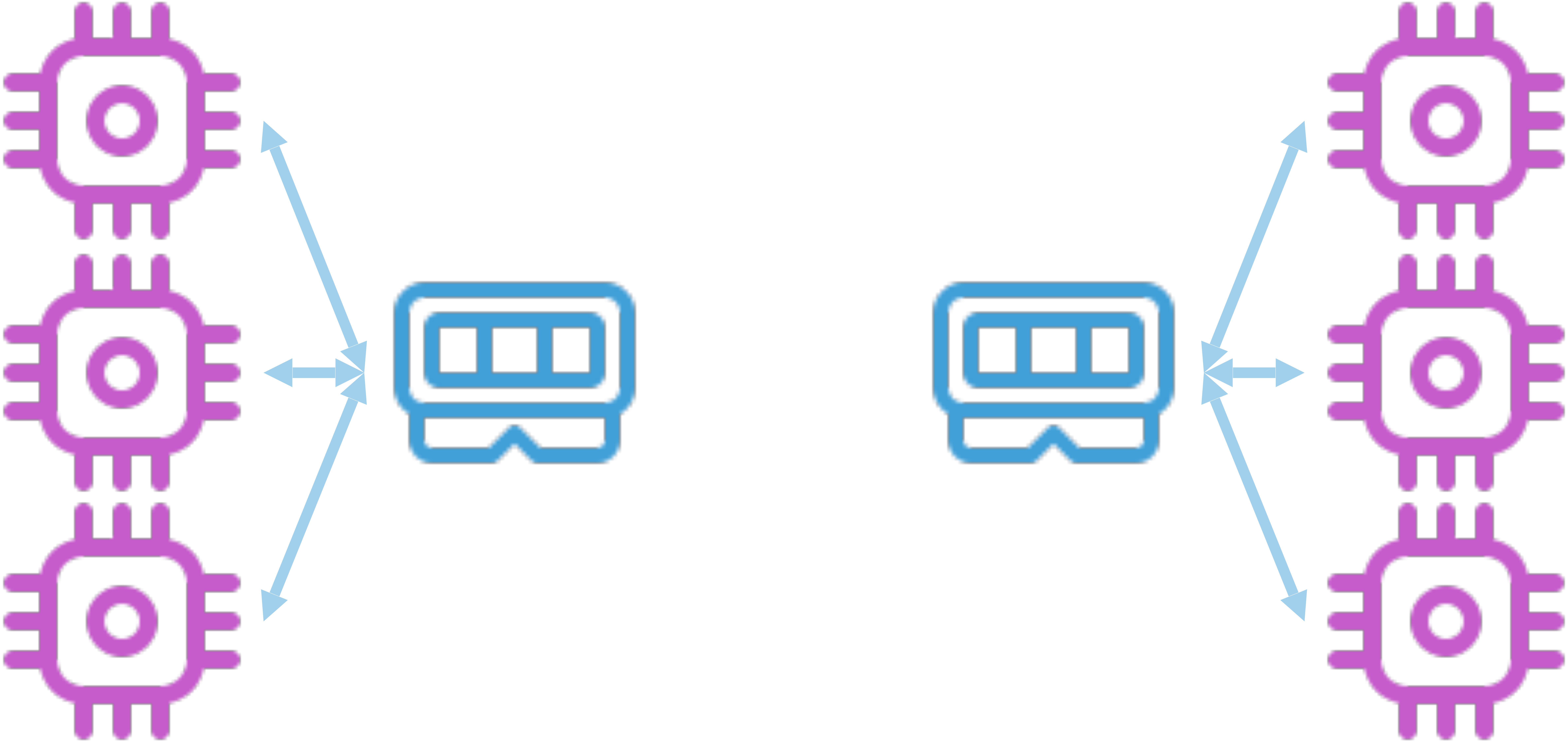


Uniform memory access

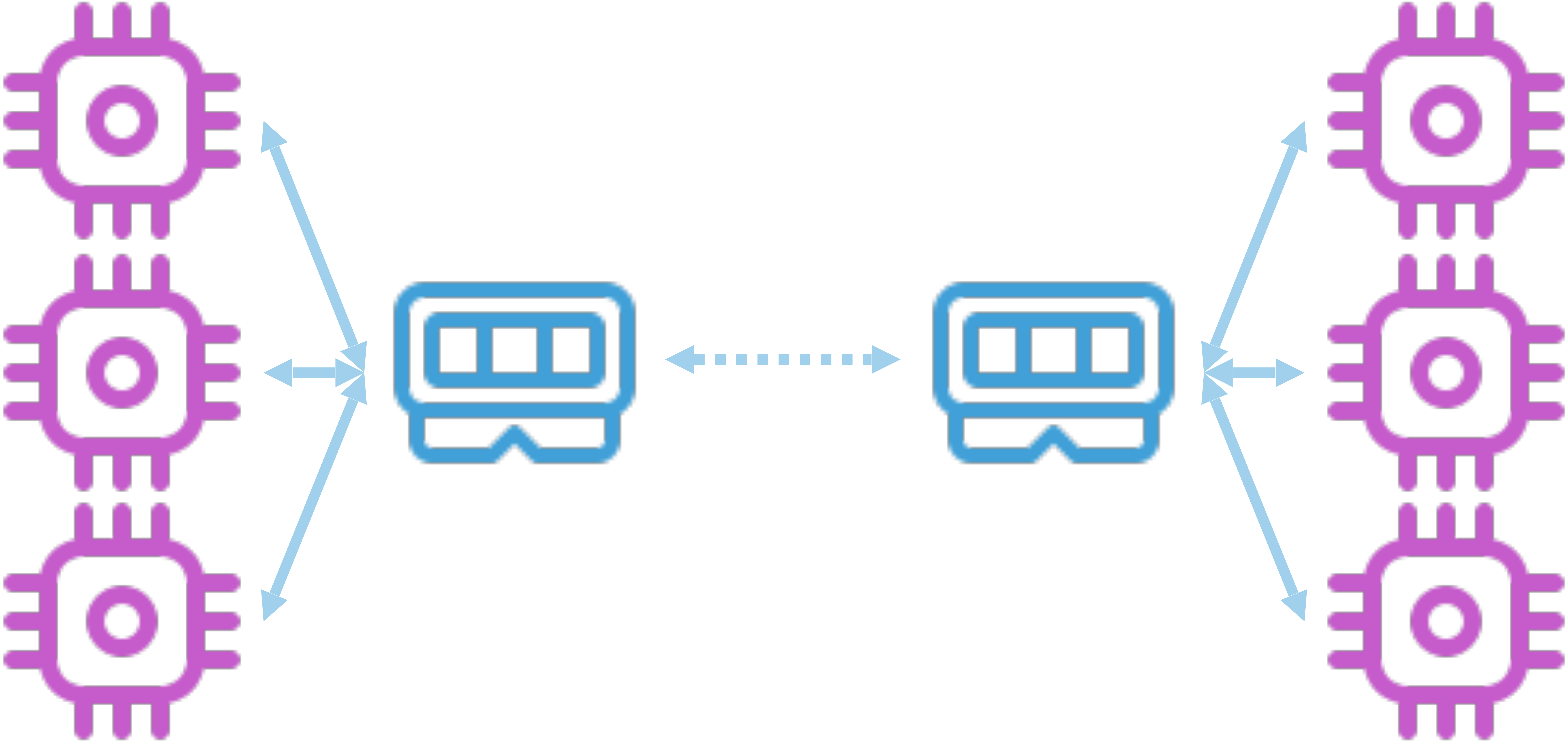
Non-uniform memory access



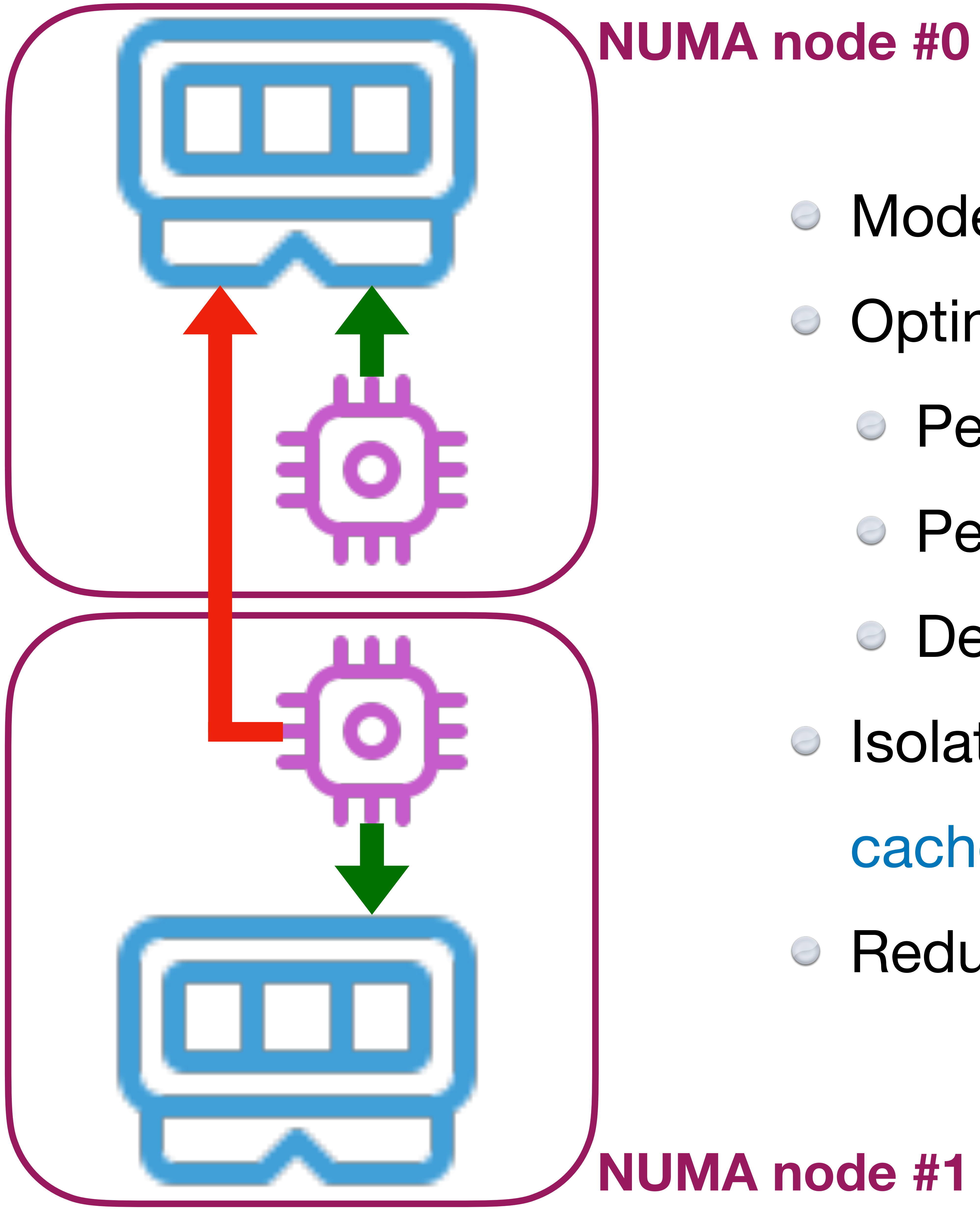
Non-uniform memory access



Non-uniform memory access



NUMA Memory Model



- Modern hardware has NUMA effects
- Optimize for NUMA:
  - Per-core mutable state
  - Per-node immutable state
  - Dedicated NUMA-aware sharing abstractions
- Isolation helps even without NUMA by reducing cache trashing and memory diffusion
- Reduce memory distance

NUMA Effects

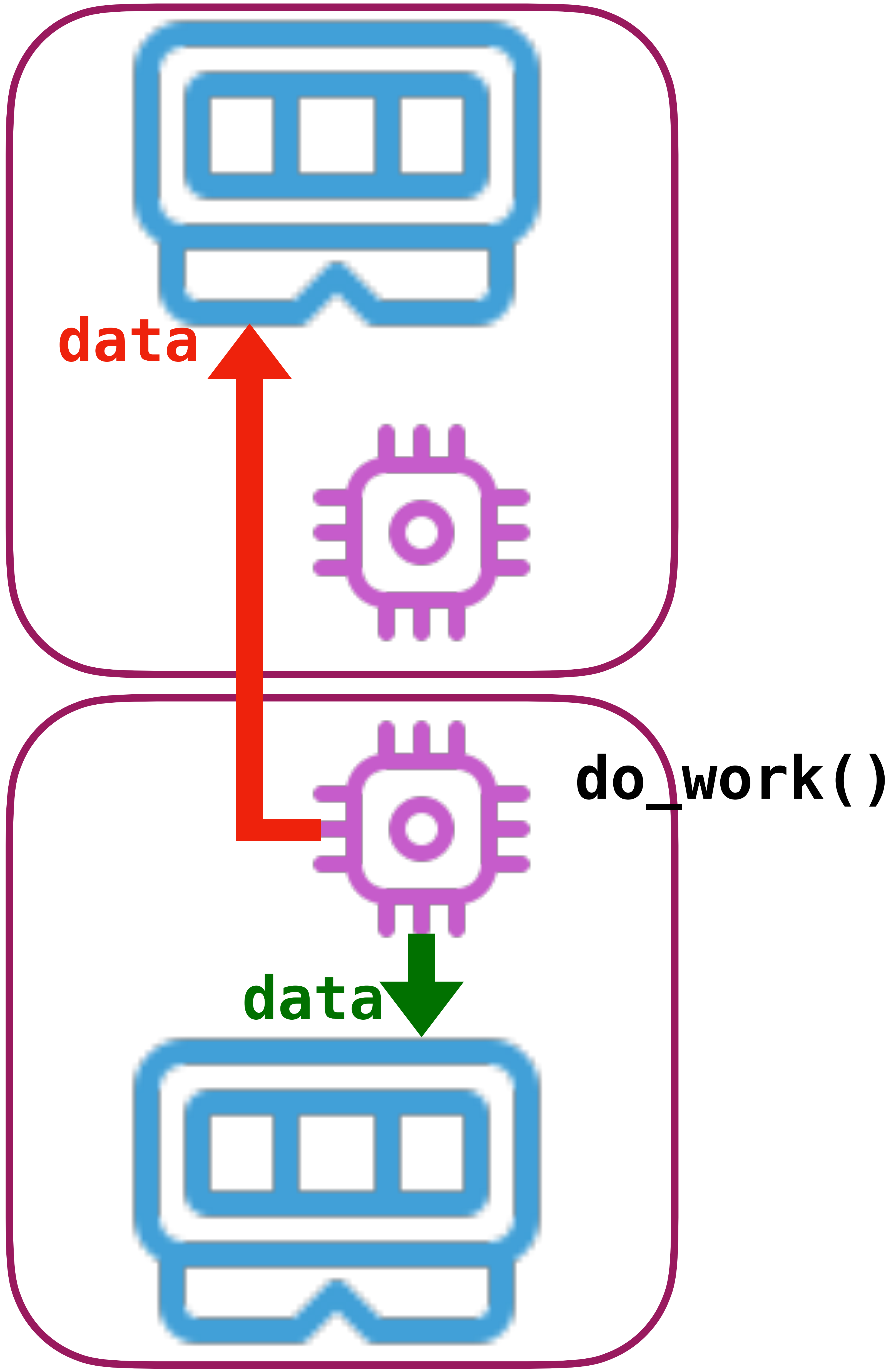
```
pub fn do_work(data: Vec<LargeData>) {
    // ...
}
```

- We have some function running on a thread which takes a bunch of large data to operate on
- If this data lives on a different NUMA node, and we access it repeatedly, this can be slower

NUMA Effects

```
pub fn do_work(data: Vec<LargeData>) {
    let data = make_closer(data);
    // ...
    work(data);
}

pub fn make_closer<T: Clone>(t: T) -> T {
    << redacted magic 🪄 >>
    t.clone()
}
```



Ongoing work

How do we expose this to service code?

What diagnostics/tools do we need?

What prior art we can build on?

Ongoing work

How do we expose this to service code?

- !Send types
- Core pinning
- “Pack up” API

What diagnostics/tools do we need?

What prior art we can build on?

Ongoing work

How do we expose this to service code?

- !Send types
- Core pinning
- “Pack up” API

What diagnostics/tools do we need?

- Type errors
- Linters
- Runtime detection

What prior art we can build on?

Ongoing work

How do we expose this to service code?

- !Send types
- Core pinning
- “Pack up” API

What diagnostics/tools do we need?

- Type errors
- Linters
- Runtime detection

What prior art we can build on?

- NUMA-aware allocators
- NUMA metadata crates
- NUMA-aware data structures

Open Questions

<slide intentionally left blank>

Shared-nothing Architecture

Rust Prague Meetup

March 2024

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

Victor Ciura
Principal Engineer
M365 Substrate