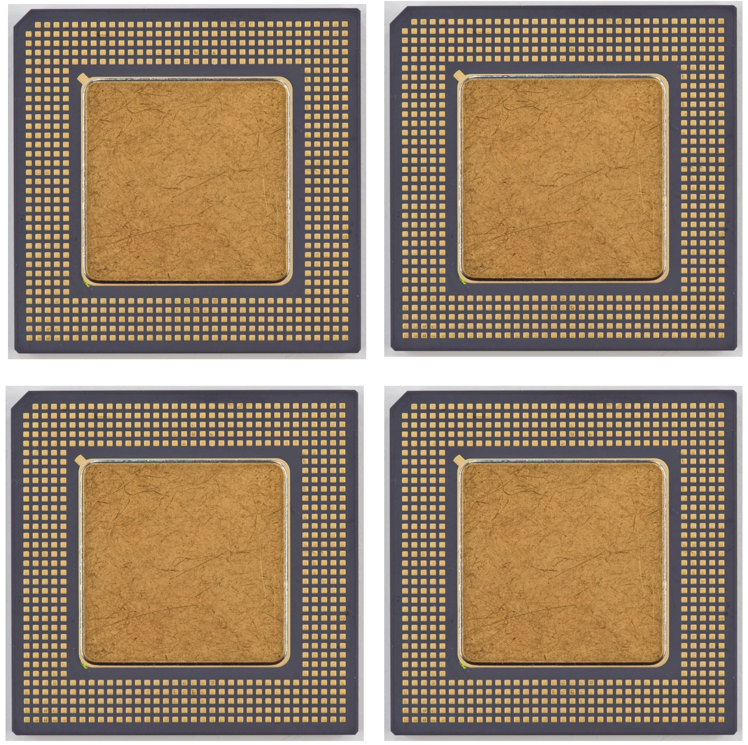
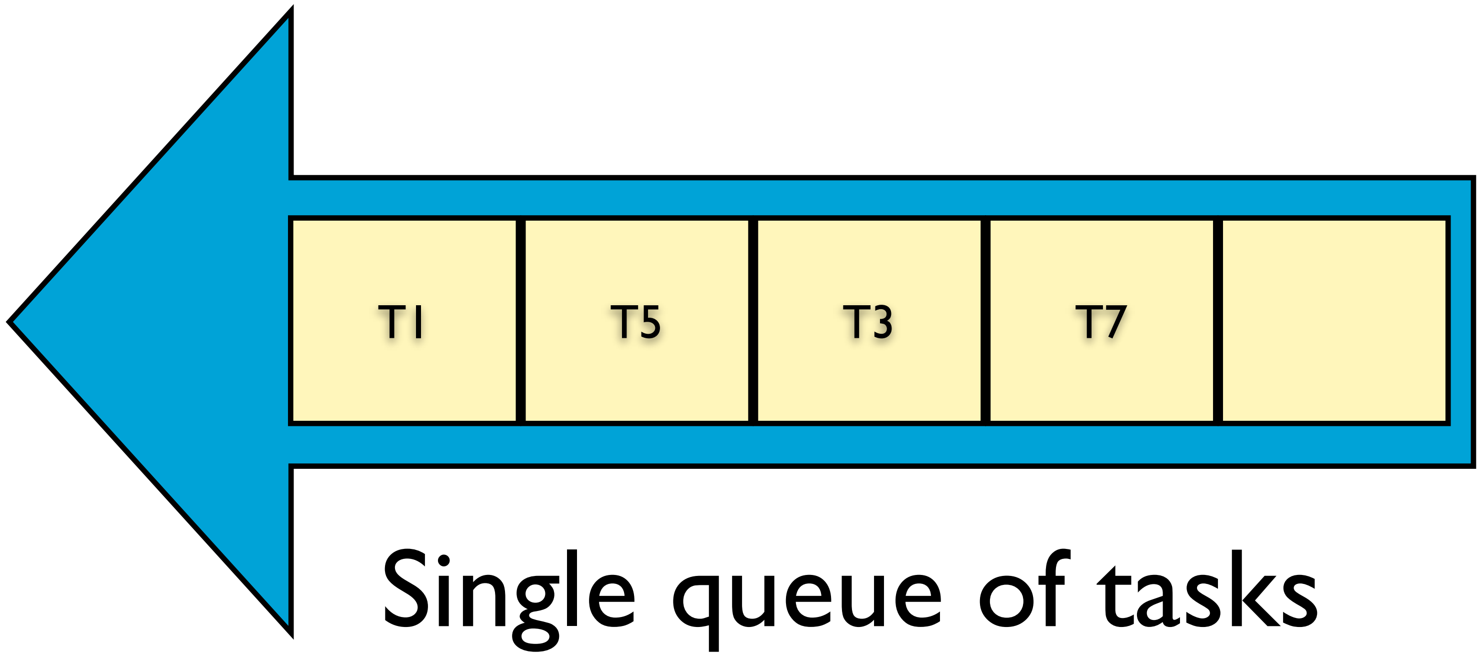
Scaling Global Scheduling with Message Passing

Max
Planck
Institute
for
Software Systems

Felipe Cerqueira
Manohar Vanga
Björn Brandenburg

Global Scheduling





Single queue of tasks
ordered by priority

Tasks can execute on any processor

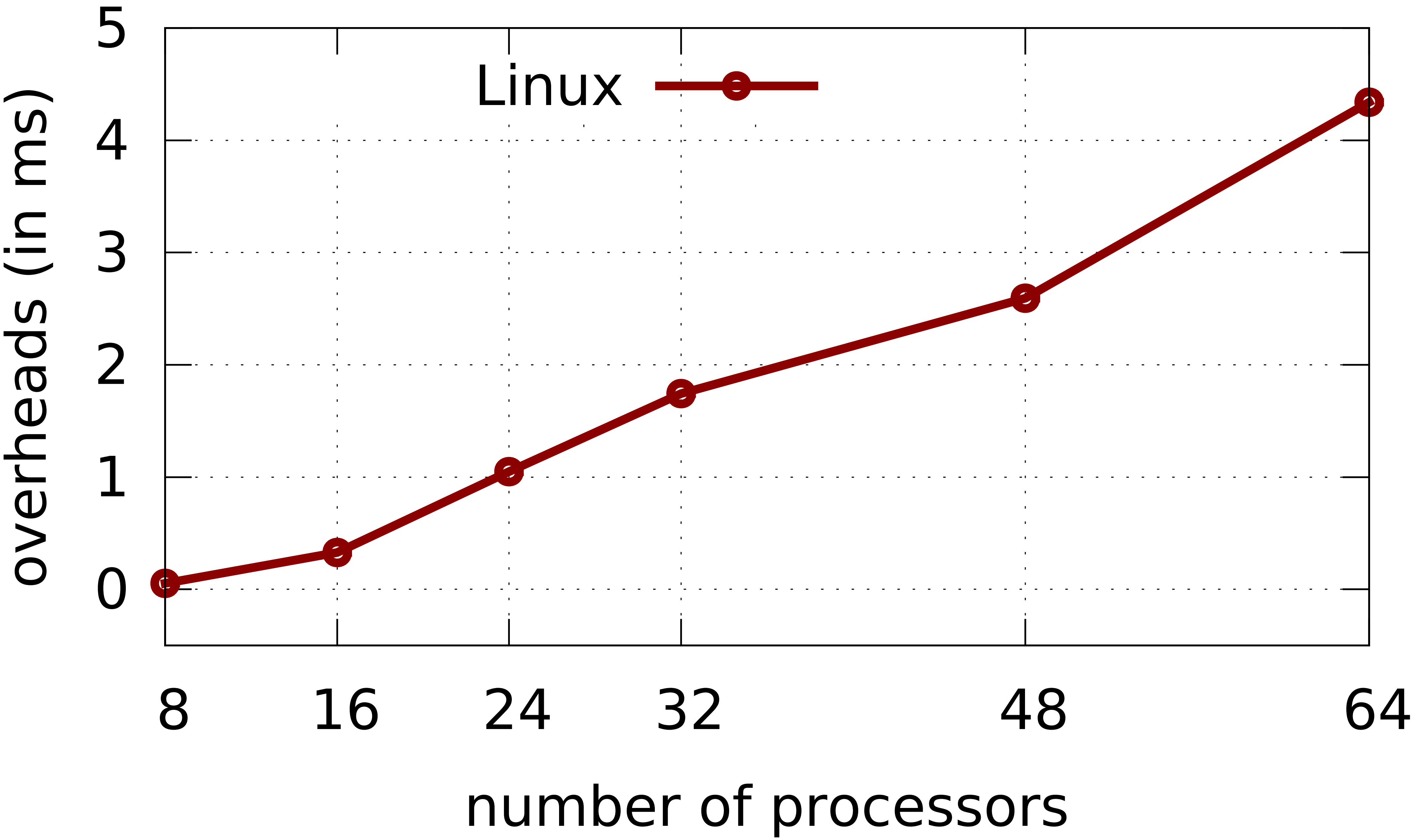
Global Scheduling

In **theory,** desirable analytical properties

In **practice**,

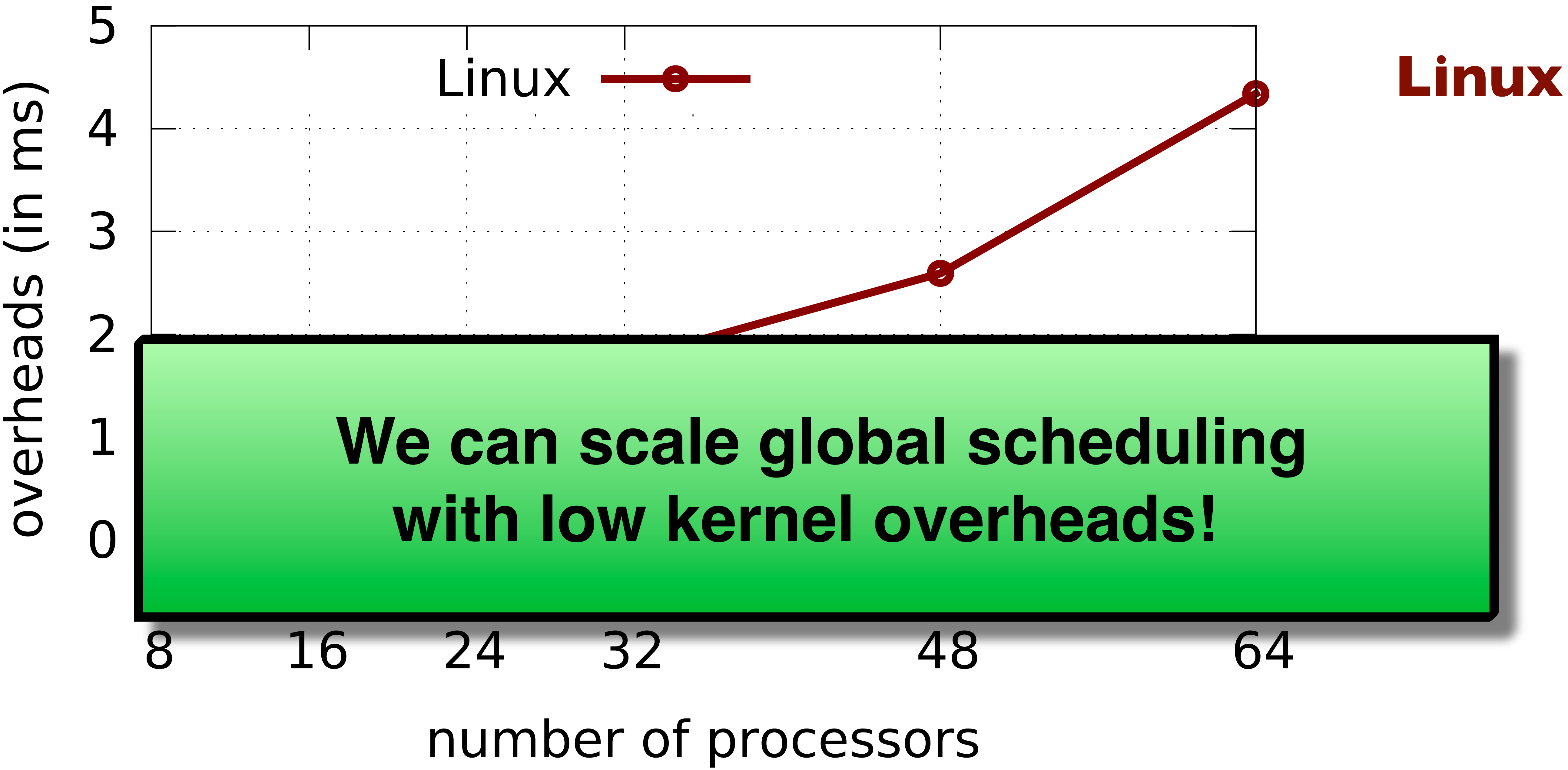
not scalable due to high overheads

Making Global Scheduling Practical



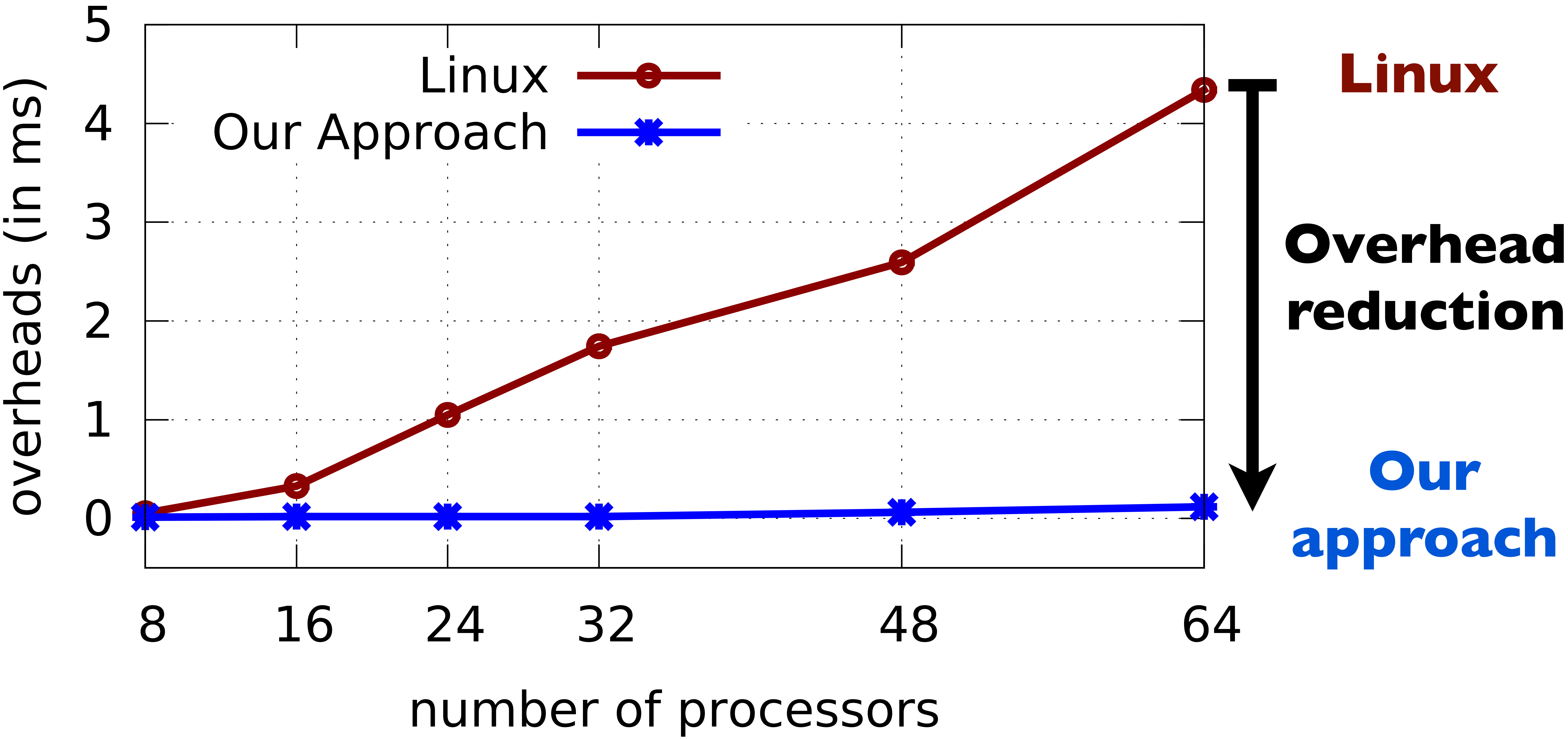
Linux

Making Global Scheduling Practical



We can scale global scheduling with low kernel overheads!

Making Global Scheduling Practical



Linux

Overhead reduction

Our approach

This Talk

1) Why global scheduling?
2) Current implementations
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation

This Talk

1) **Why global scheduling?**
2) Current implementations
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation

Why Global Scheduling?

Reasons {

Optimal schedulers
Work-conserving
Soft-real-time
and more...

Why Global Scheduling?

Reasons {

- **Optimal schedulers**
- Work-conserving
- Soft-real-time
- and more...

Optimal real-time schedulers are global

Why Global Scheduling?

Reasons {

Optimal schedulers
Work-conserving
Soft-real-time
and more...

Good for **open** and **dynamic systems**
Resilient to overloads

Why Global Scheduling?

Reasons {

- Optimal schedulers
- Work-conserving
- **Soft-real-time**
- and more...

Some global schedulers guarantee
bounded tardiness without utilization loss

Why Global Scheduling?

Reasons

- Optimal schedulers
- Work-conserving
- Soft-real-time
- **and more...**

Supports **priority inheritance**
Useful in **race-to-idle** energy conservation

Why Global Scheduling?

Reasons {

- Optimal schedulers
- Work-conserving
- Soft-real-time
- and more...

Properties not fully guaranteed by Partitioned and Clustered Scheduling!

Global Schedulers in Practice

Default scheduler for Linux, QNX and VXWorks.

This Talk

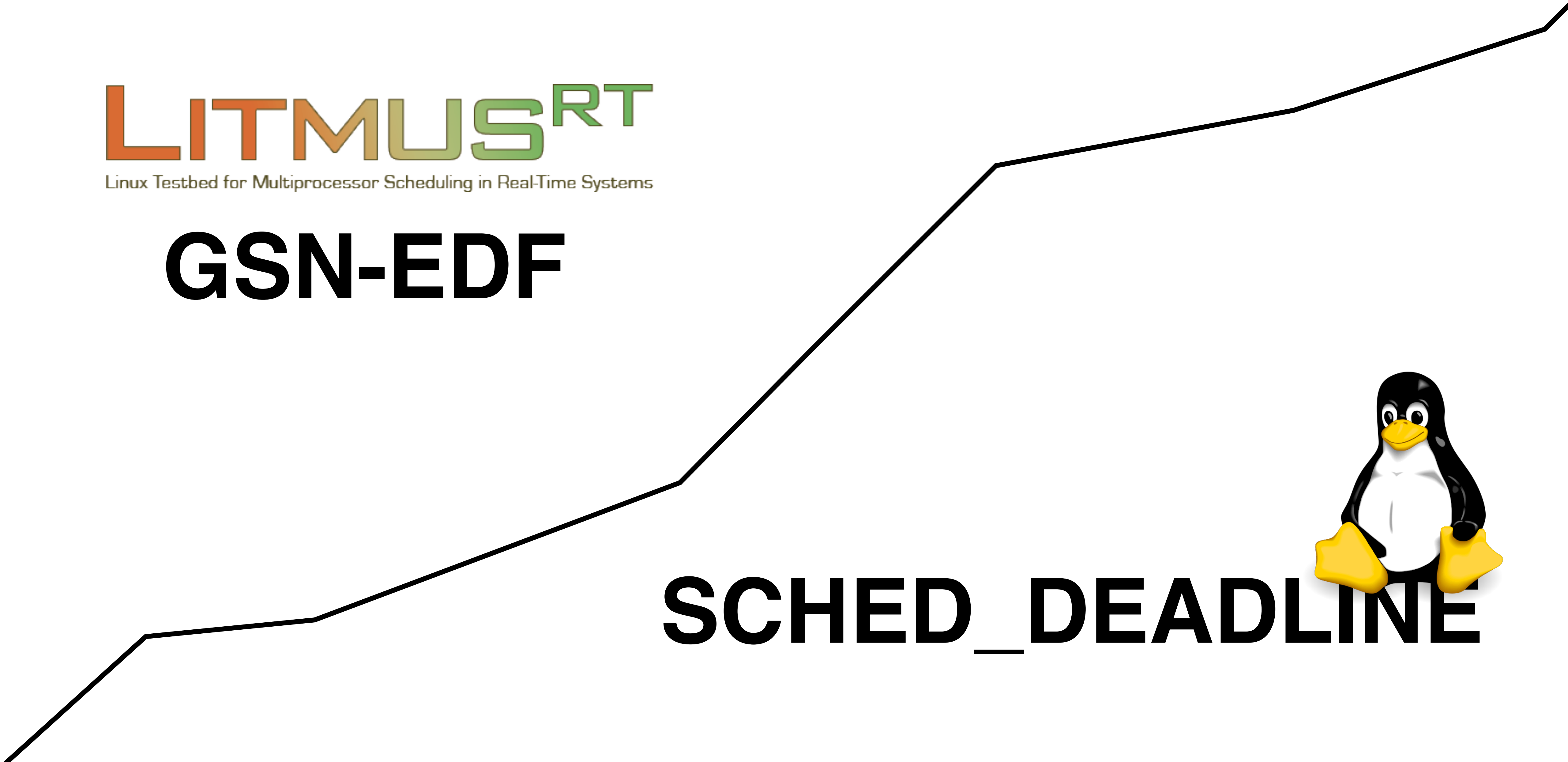
1) Why global scheduling?
2) **Current implementations**
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation

This Talk

1) Why global scheduling?
2) **Current implementations**
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation

G-EDF as a representative of global scheduling

Comparing Two Extremes



LITMUS^RT

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

GSN-EDF

SCHED_DEADLINE

Comparing Two Extremes

LITMUS RT

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

GSN-EDF

Globally shared state, single lock

Distributed state, multiple locks

SCHED_DEADLINE

Comparing Two Extremes

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

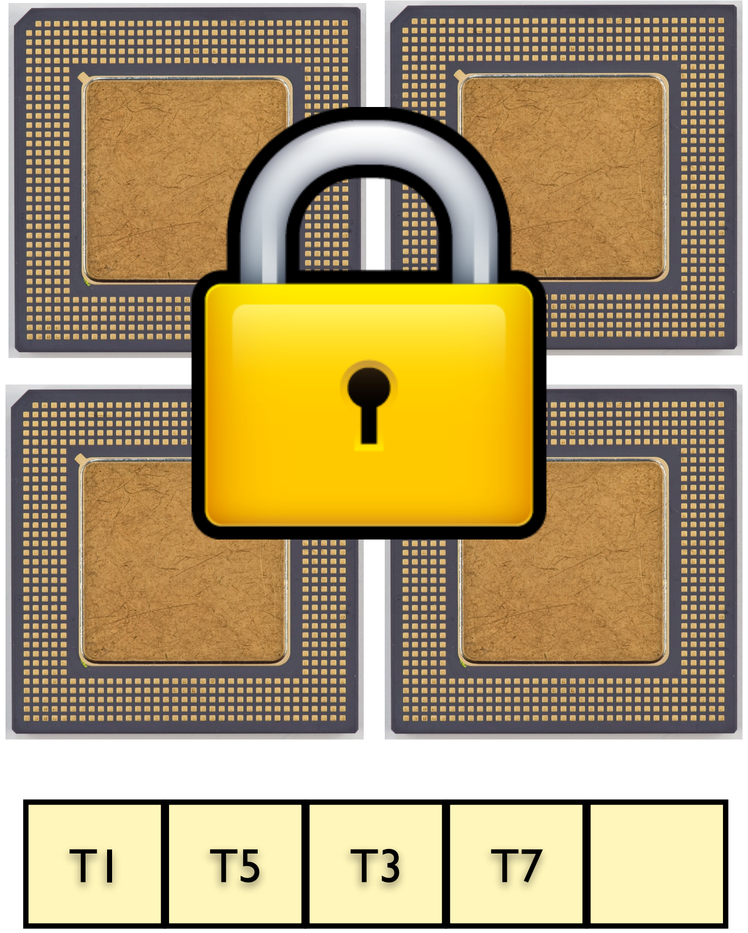
GSN-EDF

Globally shared state,
single lock

Distributed state,
multiple locks

SCHED_DEADLINE

GSN-EDF



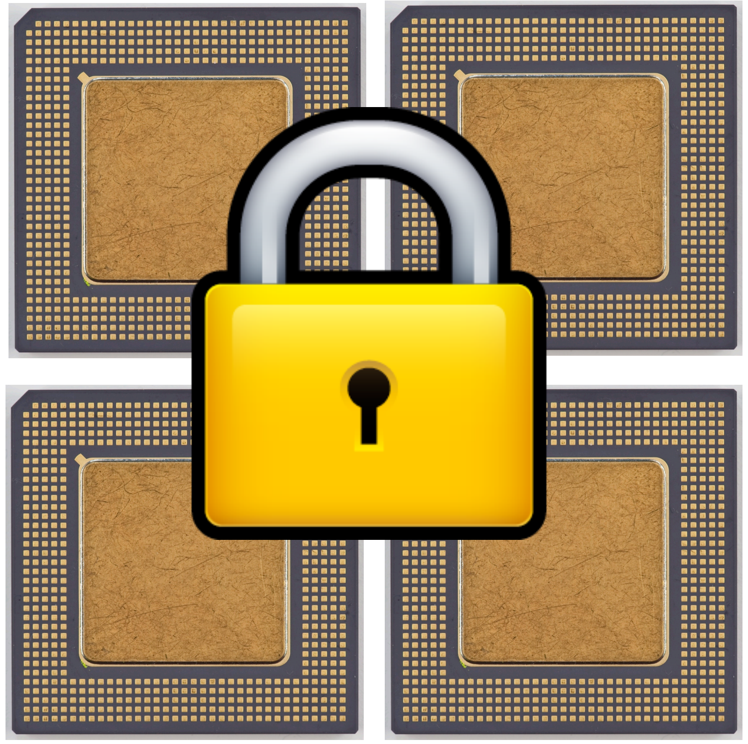
Global-EDF with support for **S**uspension-based protocols and O(1) **N**on-preemptable sections

Link-based scheduler (Block et al., 07)

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

GSN-EDF



T1	T5	T3	T7	

Global-EDF with support for **S**uspension-based protocols and O(1) **N**on-preemptable sections

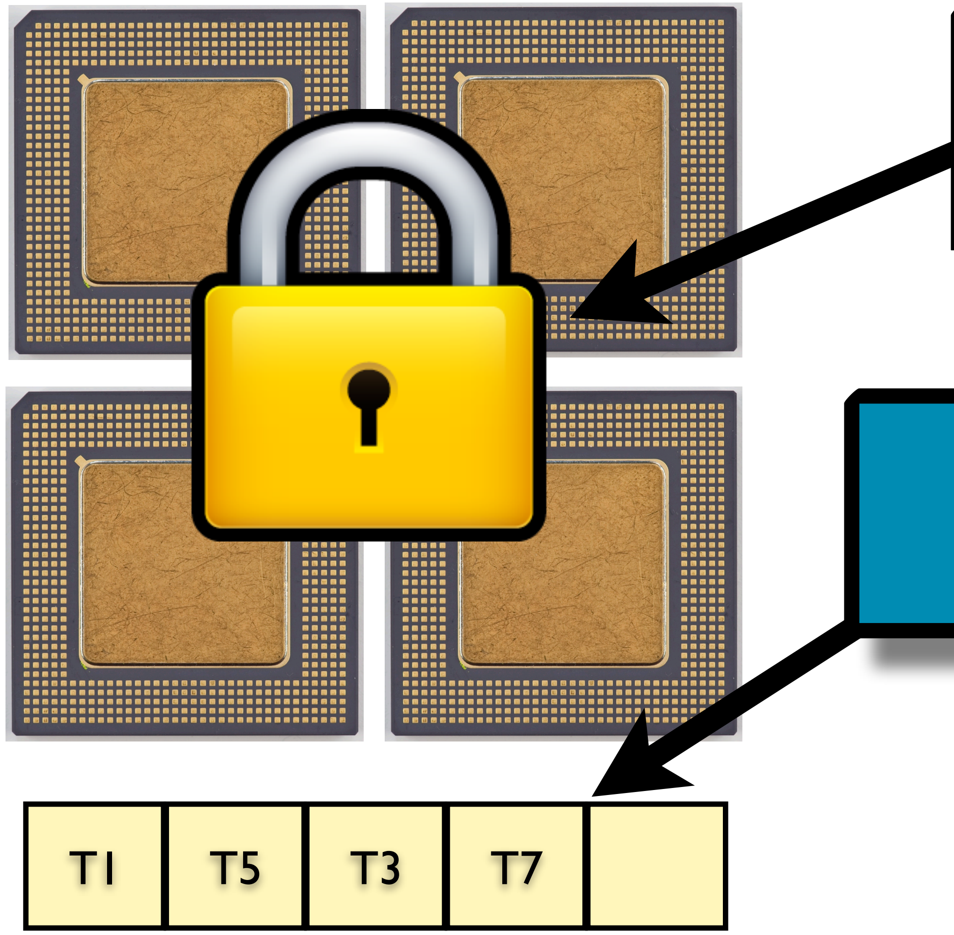
Link-based scheduler (Block et al., 07)

➡ allows simplified locking

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

GSN-EDF



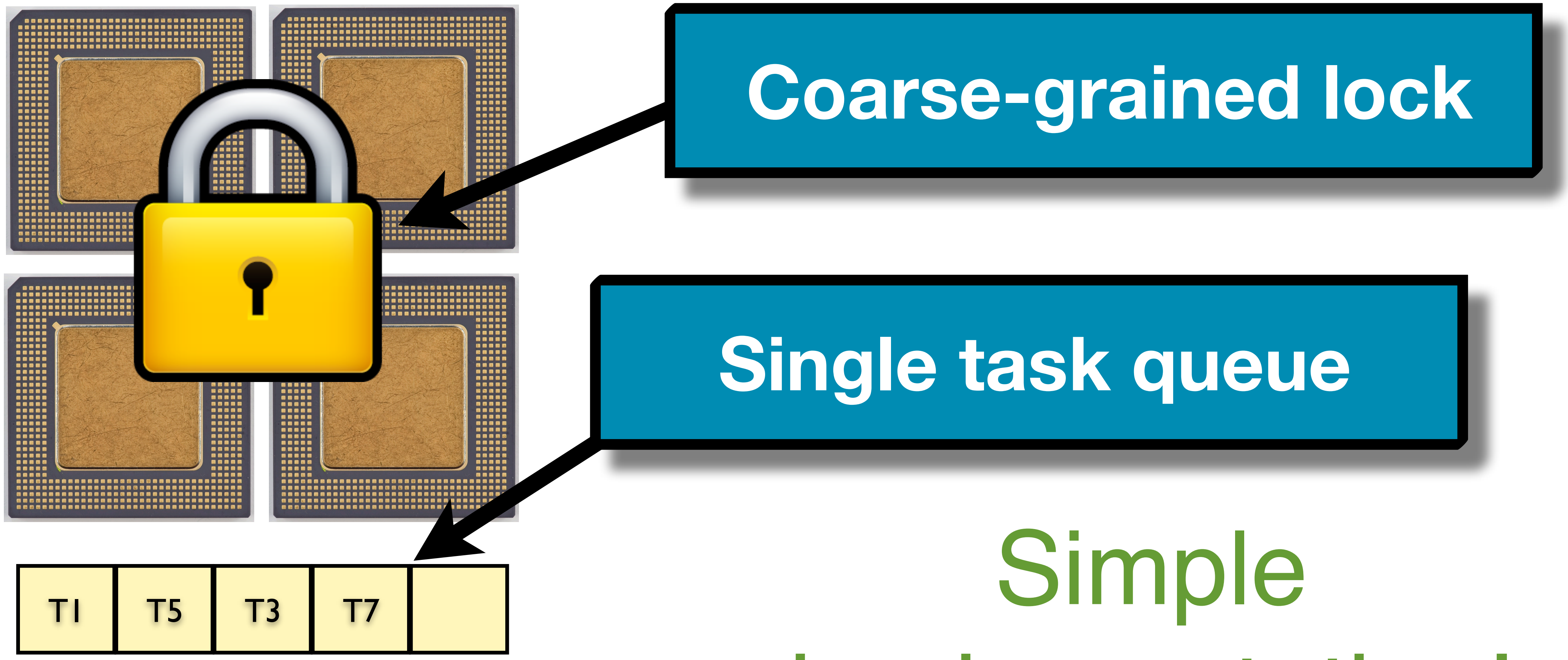
Coarse-grained lock

Single task queue

LITMUS^RT

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

GSN-EDF



Coarse-grained lock

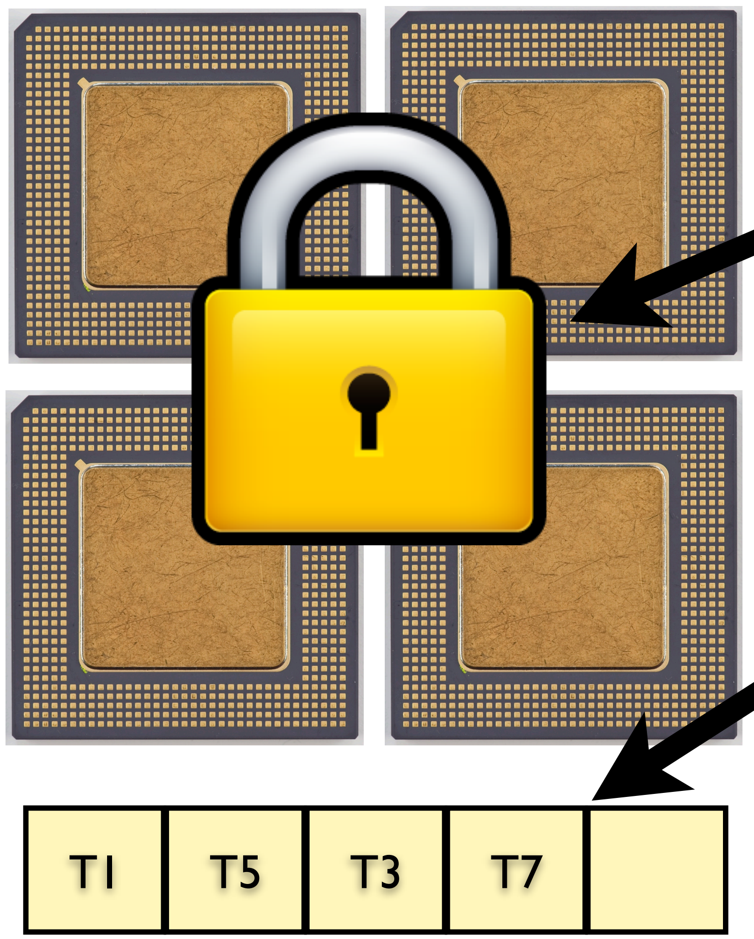
Single task queue

Simple implementation!

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

GSN-EDF



Coarse-grained lock

Single task queue

Simple implementation!

How does it scale?

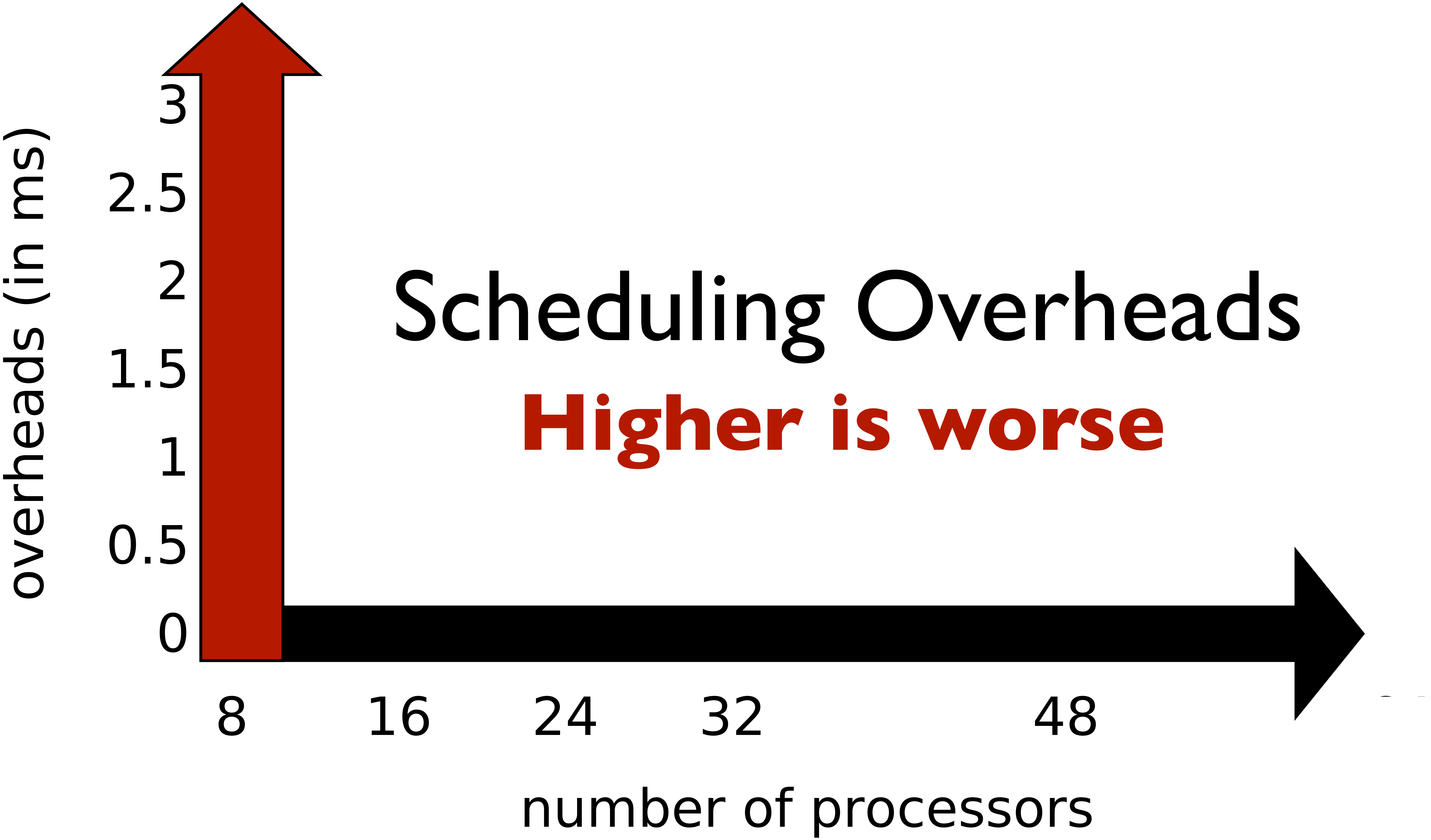
LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

Experimental Setup

- Intel Xeon X7550 @2.0GHz, with 64 cores
- Linux 3.10 with patches
  - ➡ LITMUS^RT 2013.1 and SCHED_DEADLINE v8
- Lightweight build — disabled most drivers and debugging options

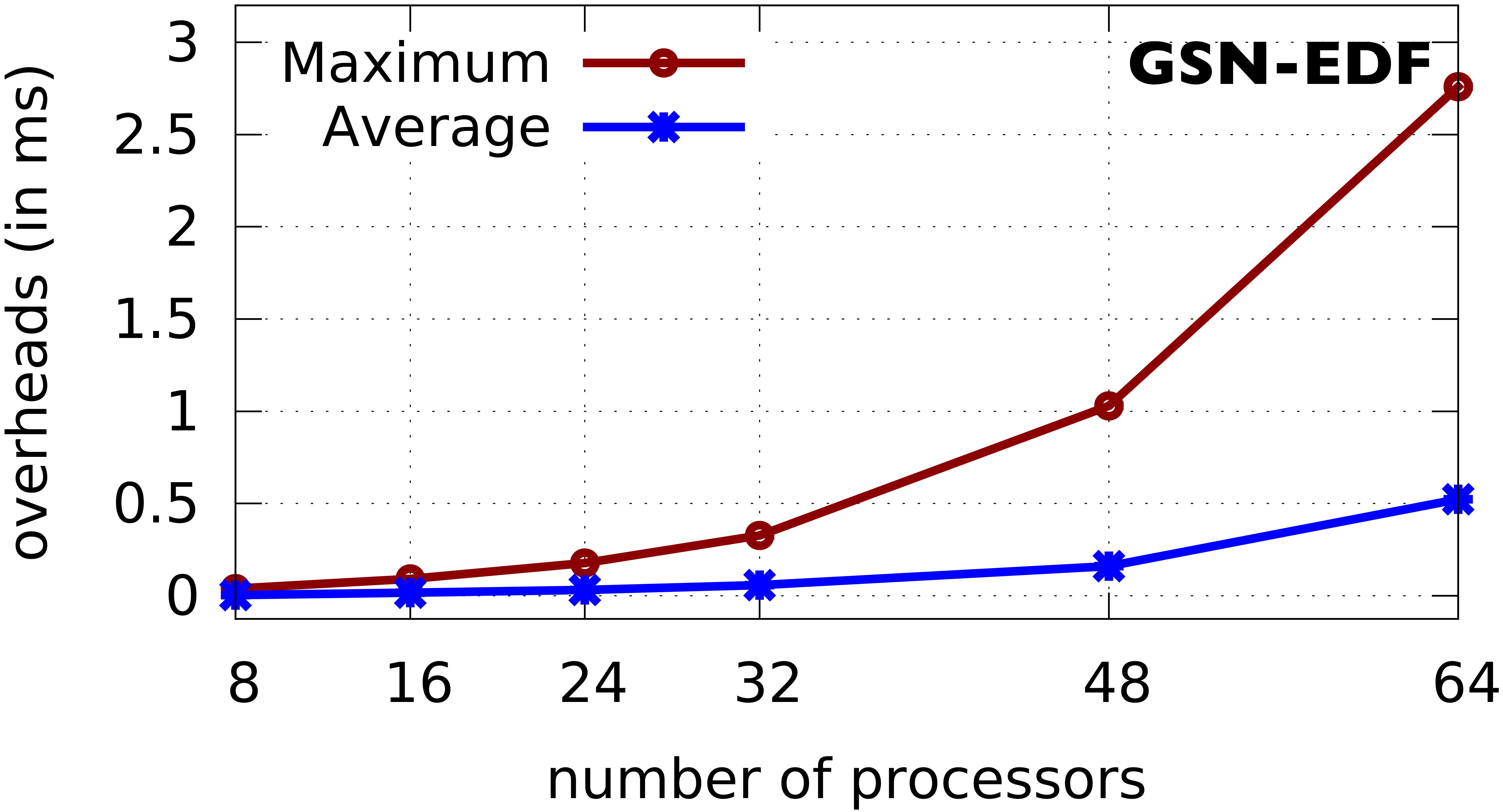
Overheads Under GSN-EDF



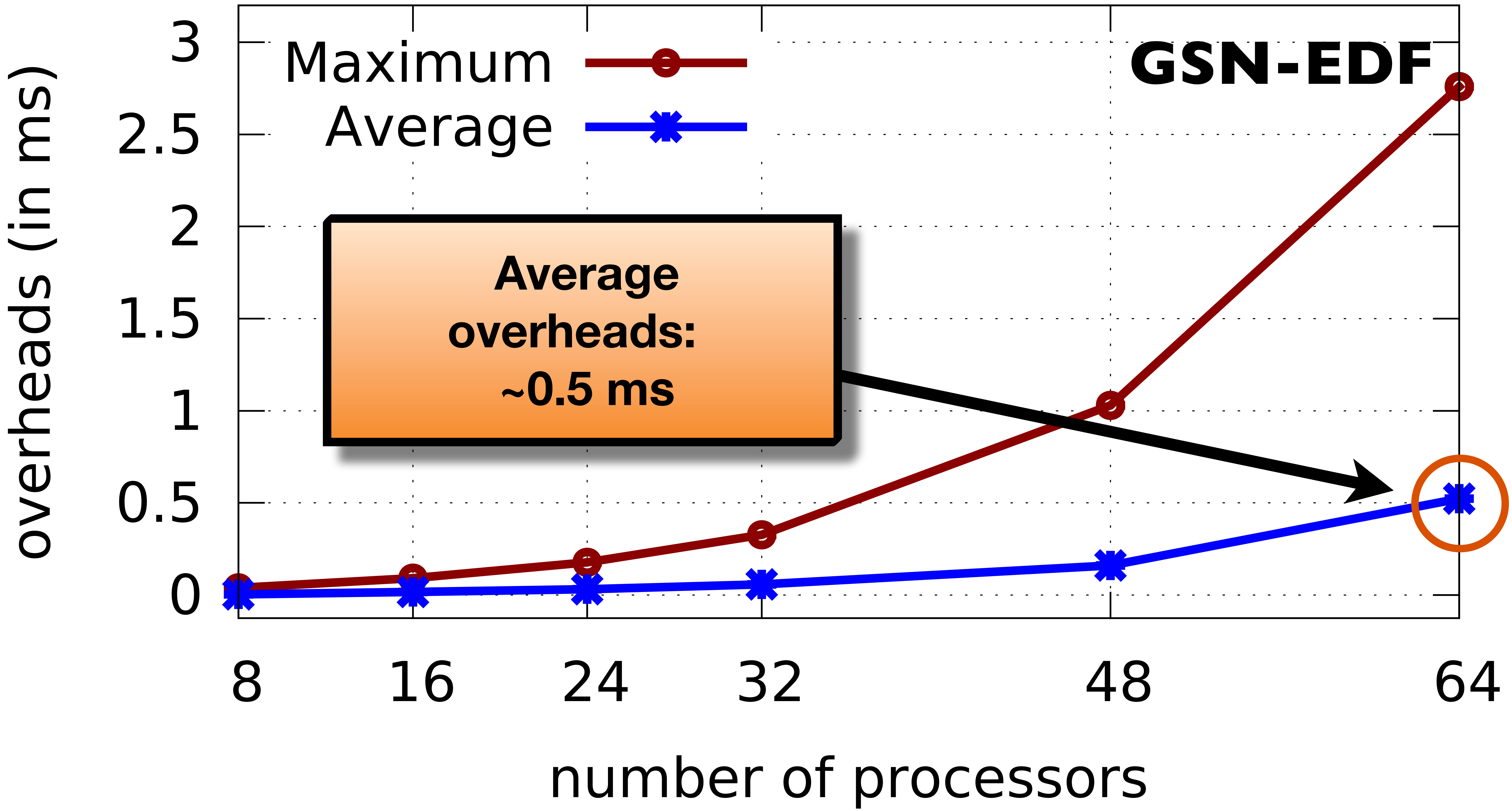
Scheduling Overheads

Higher is worse

Global Lock Does Not Scale!



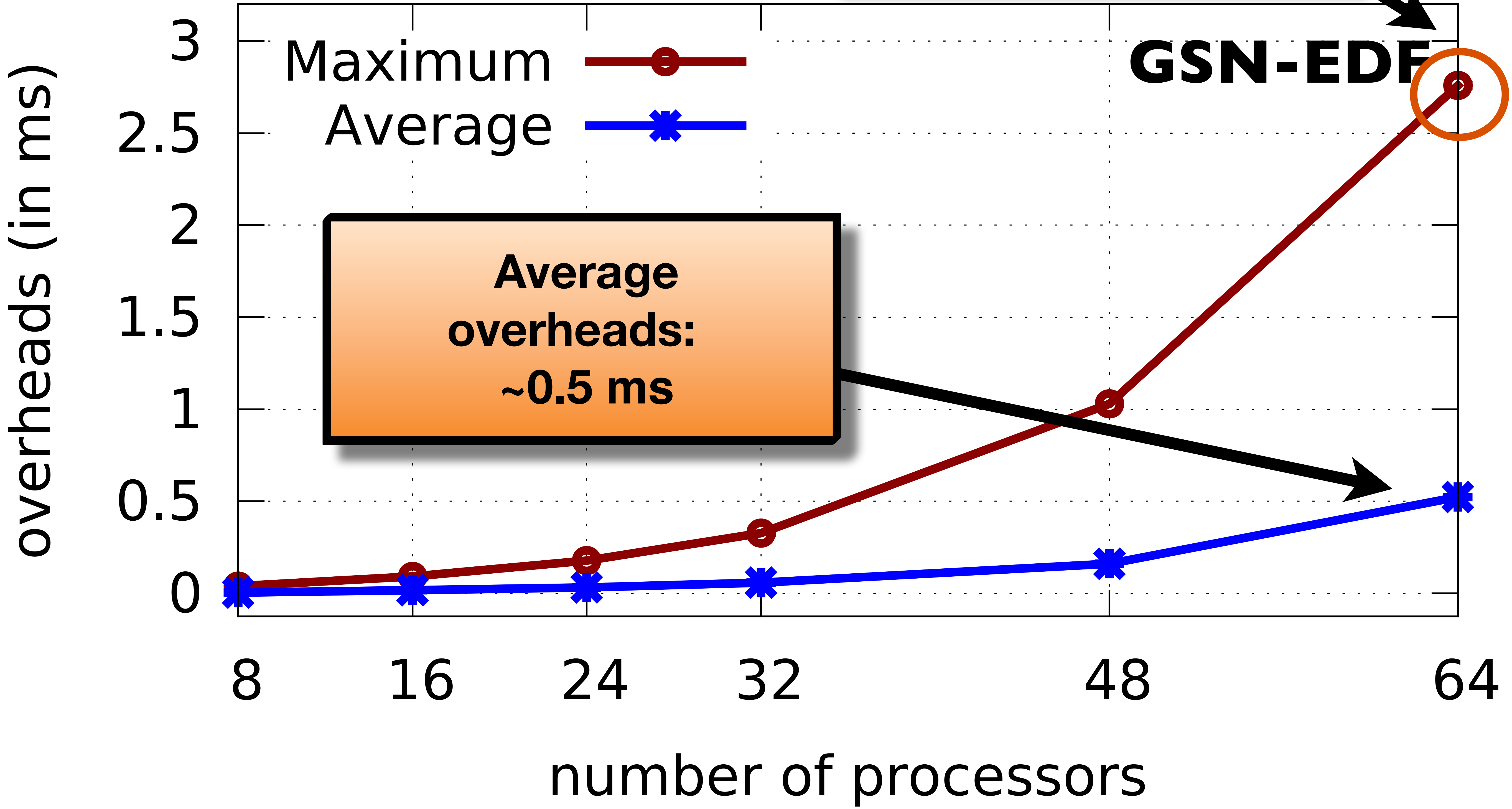
Global Lock Does Not Scale!



Average overheads: ~0.5 ms

Global Lock Do e!

For 64 CPUs, maximum overheads of ~3 ms



Average overheads: ~0.5 ms

Comparing Two Extremes

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

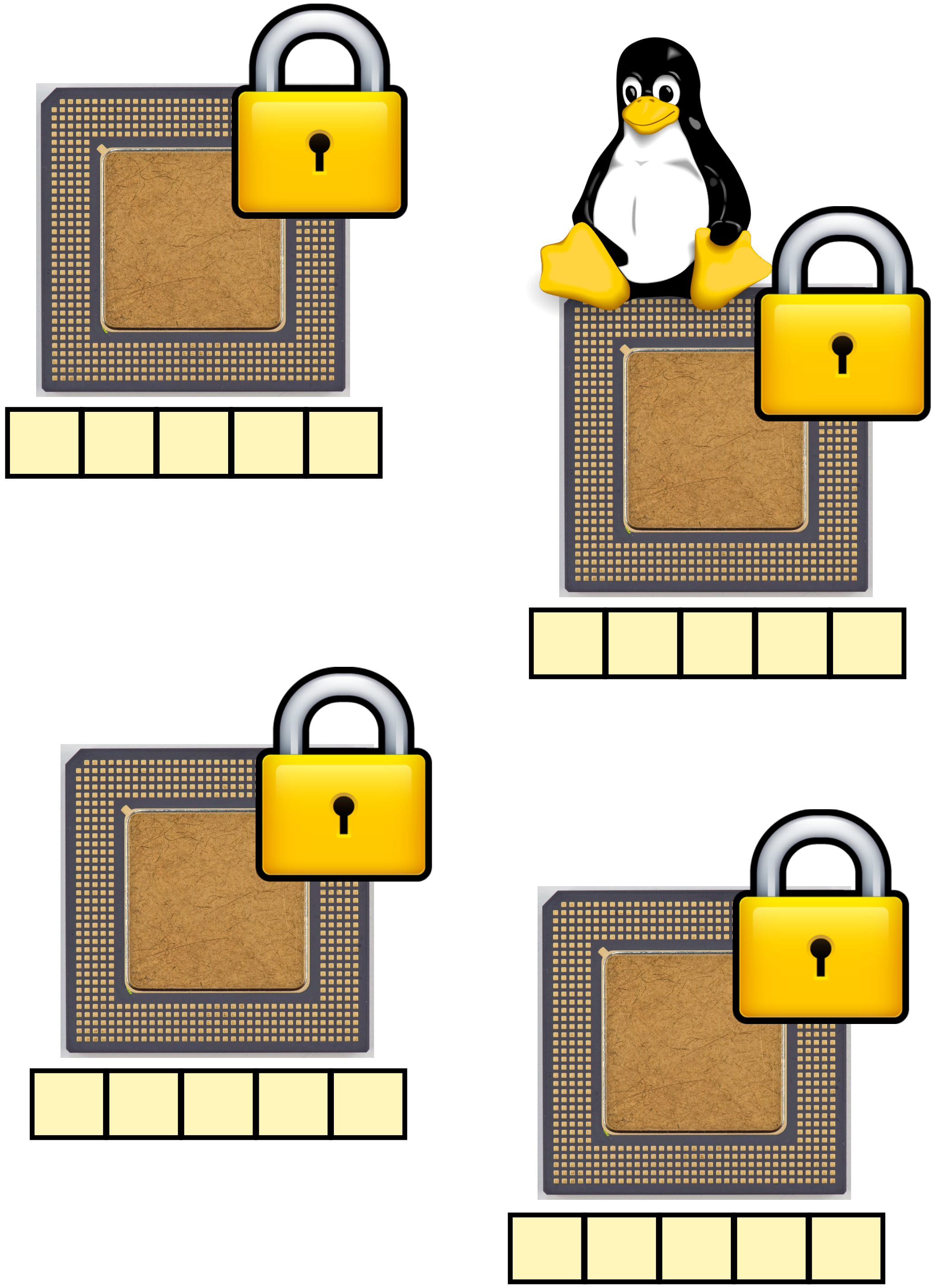
GSN-EDF

Globally share state,
single lock

Distributed state,
multiple locks

SCHED_DEADLINE

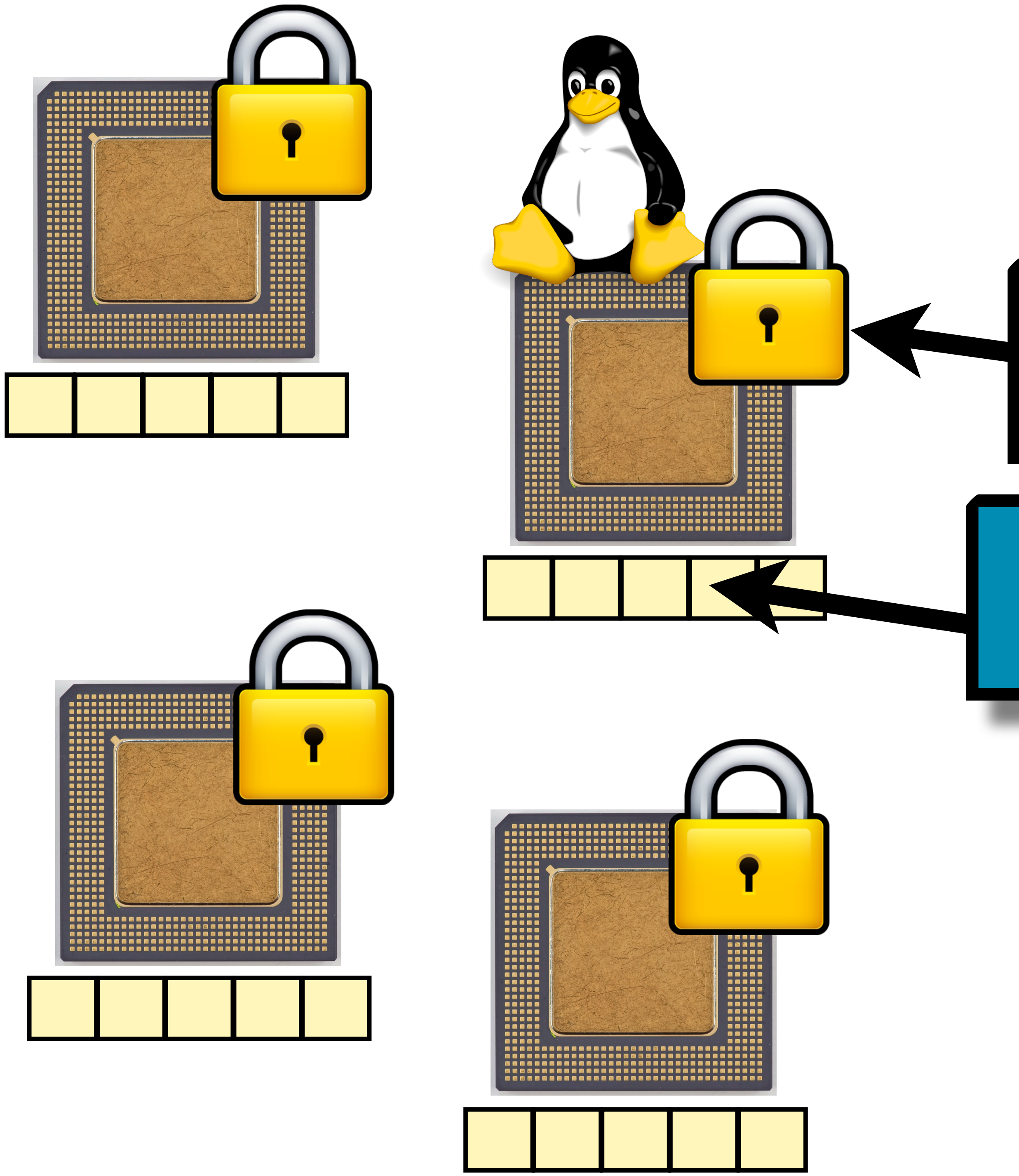
SCHED_DEADLINE



Design inherited from Linux scheduler

SCHED_DEADLINE

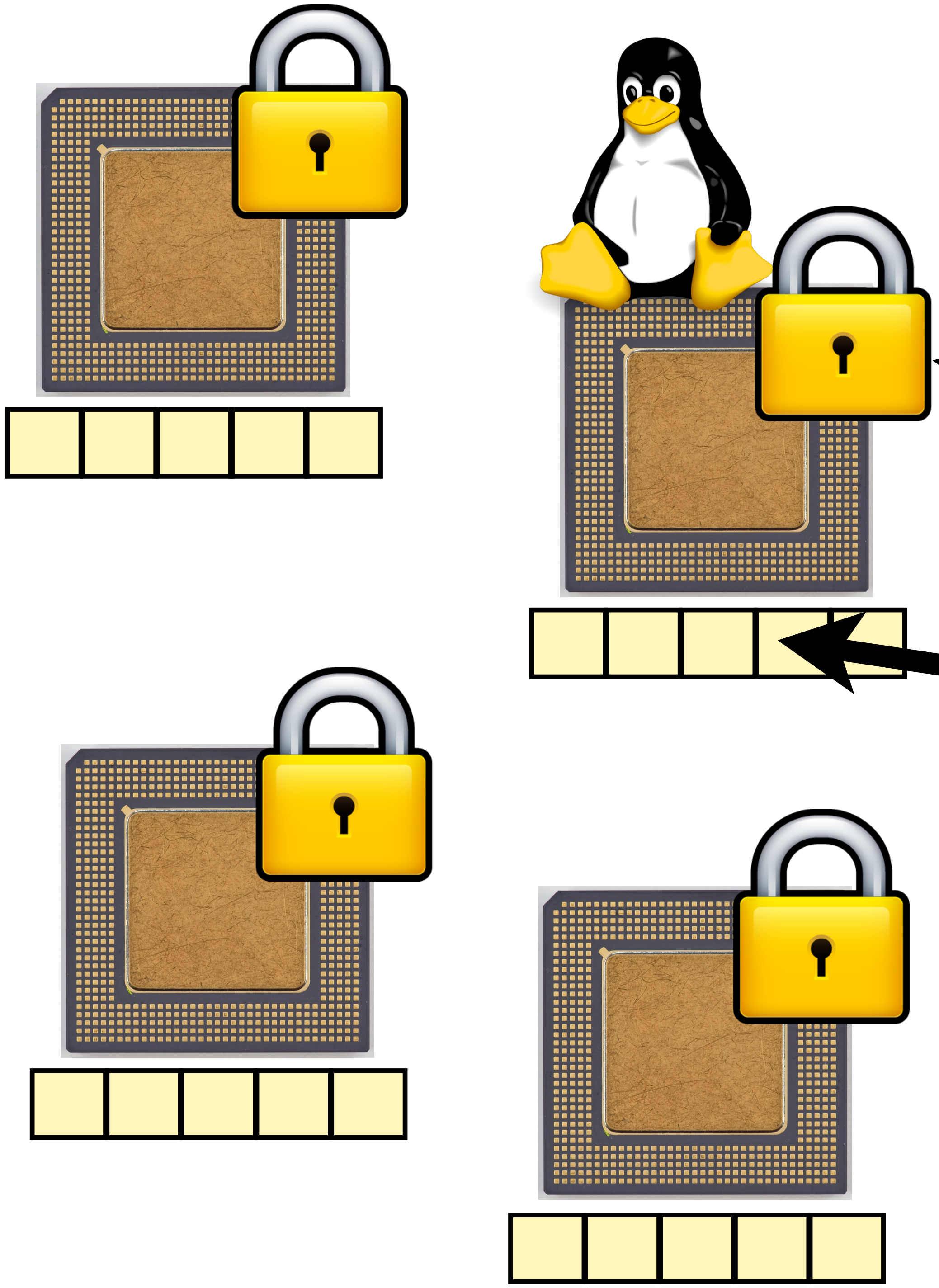
Design inherited from Linux scheduler



Per-CPU locks

Per-CPU task queues

SCHED_DEADLINE



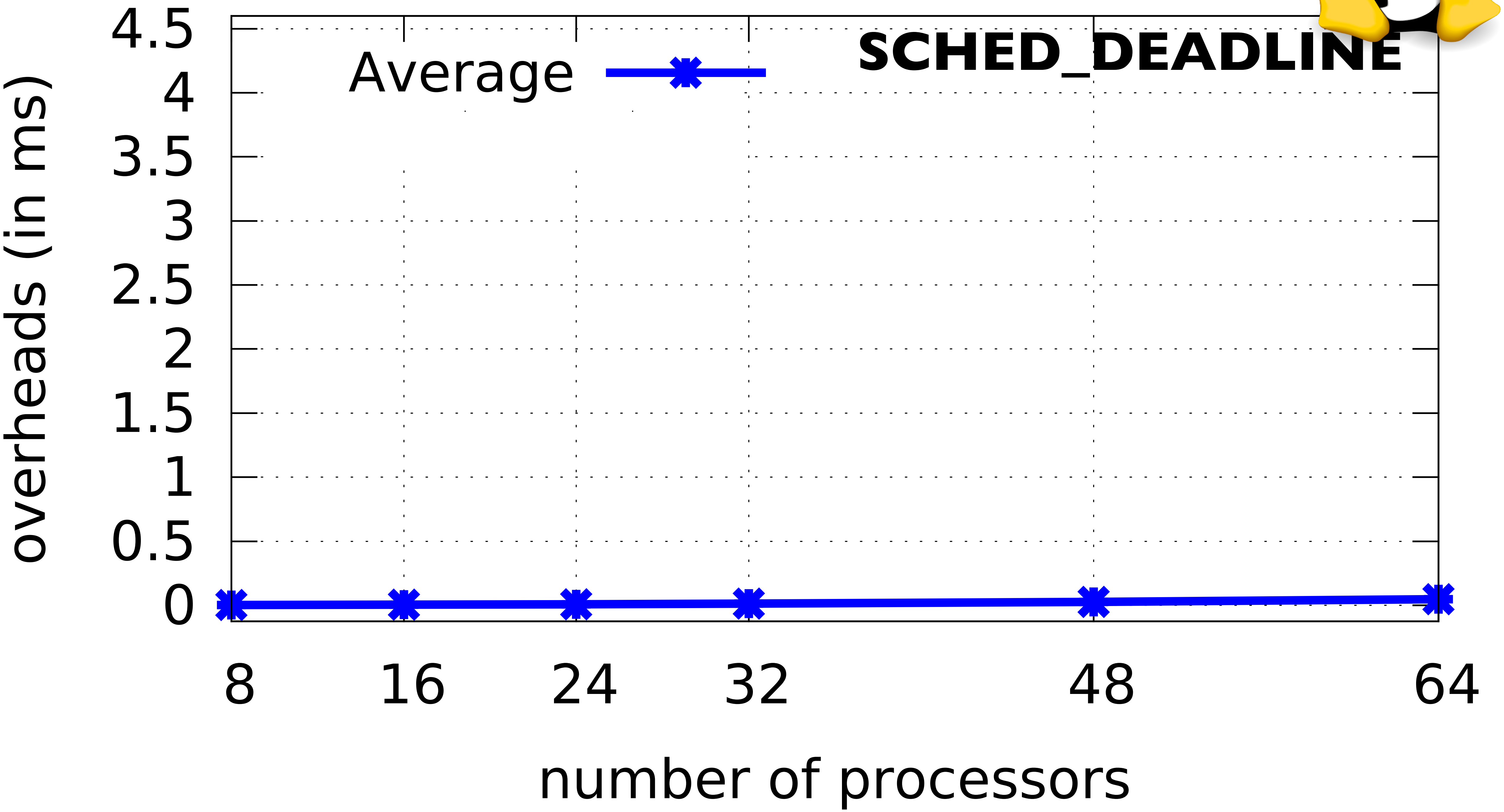
Design inherited from Linux scheduler

Per-CPU locks

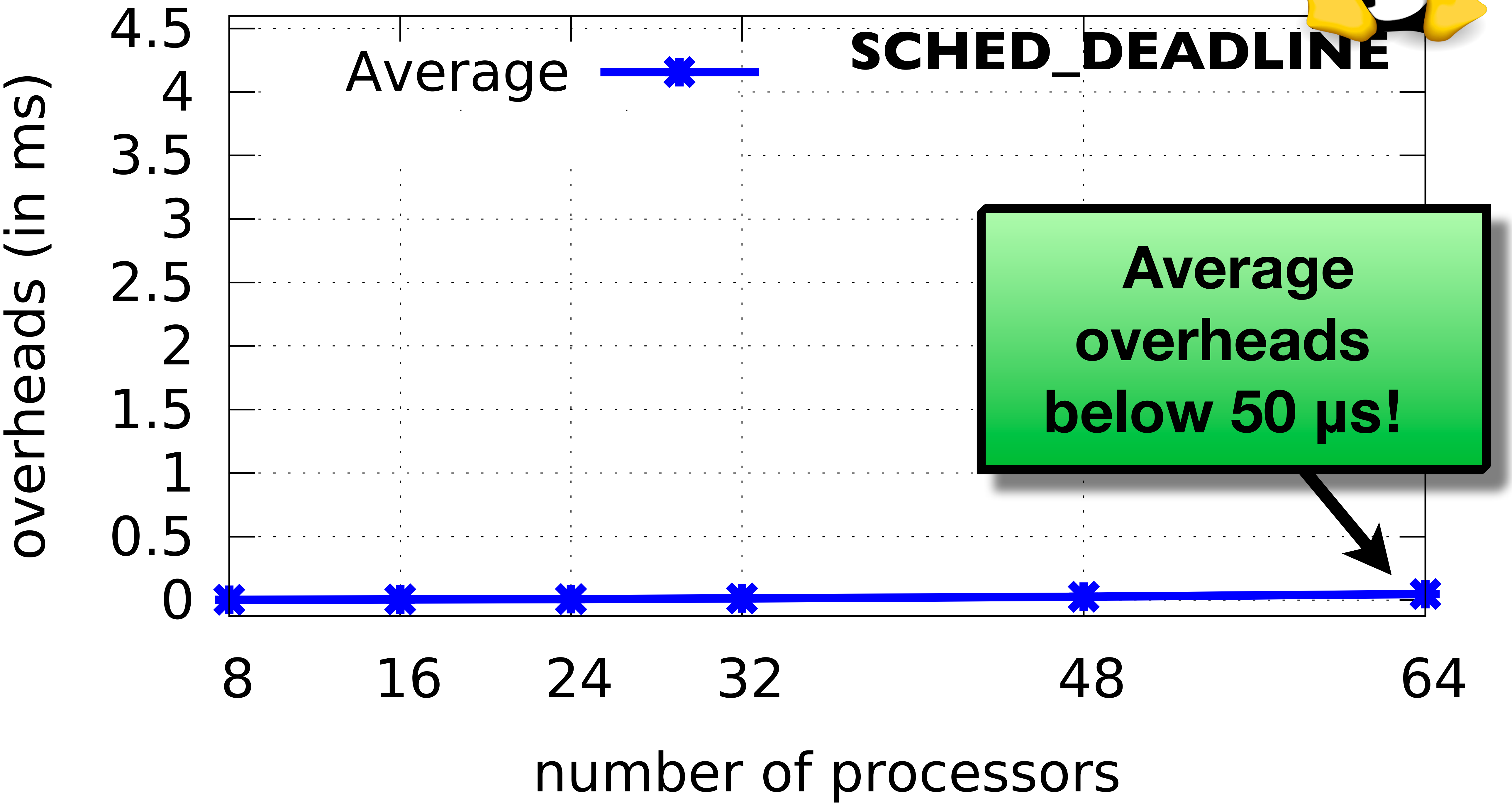
Per-CPU task queues

Intuition:
Fine-grained locking decreases contention

Benefit of Fine-Grained Locking

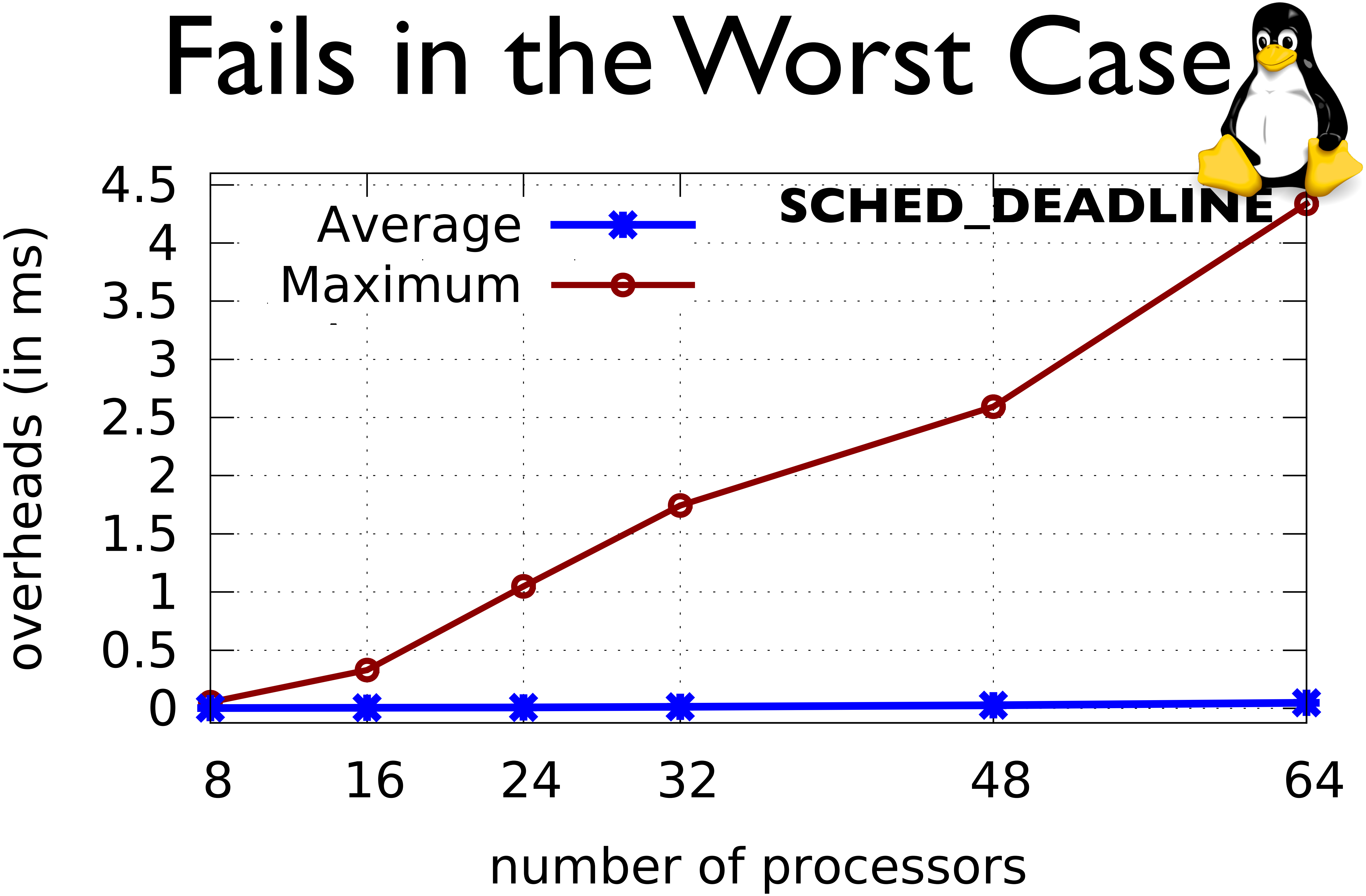


Benefit of Fine-Grained Locking

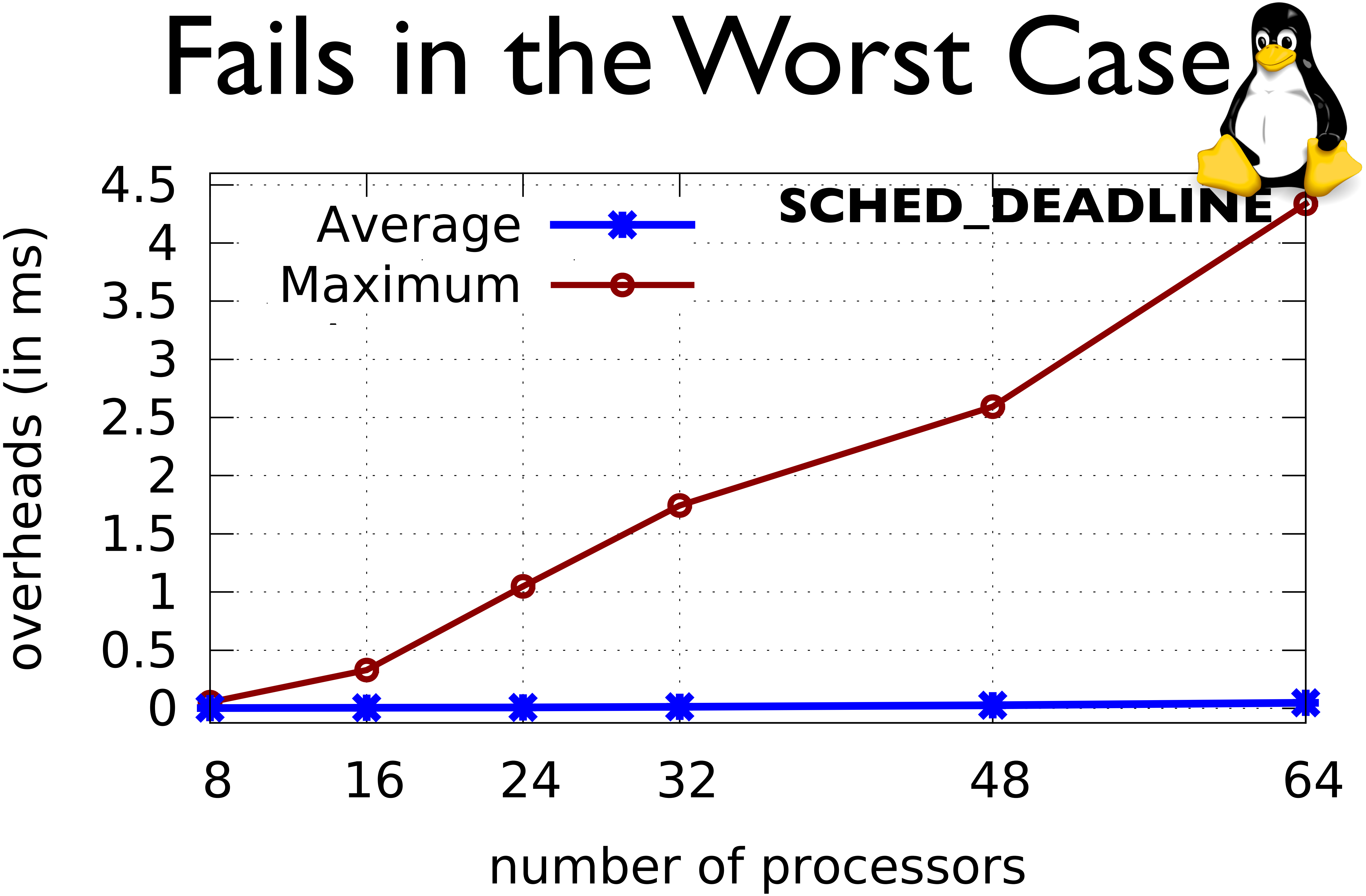


Average overheads below 50 µs!

Fine-Grained Locking Fails in the Worst Case

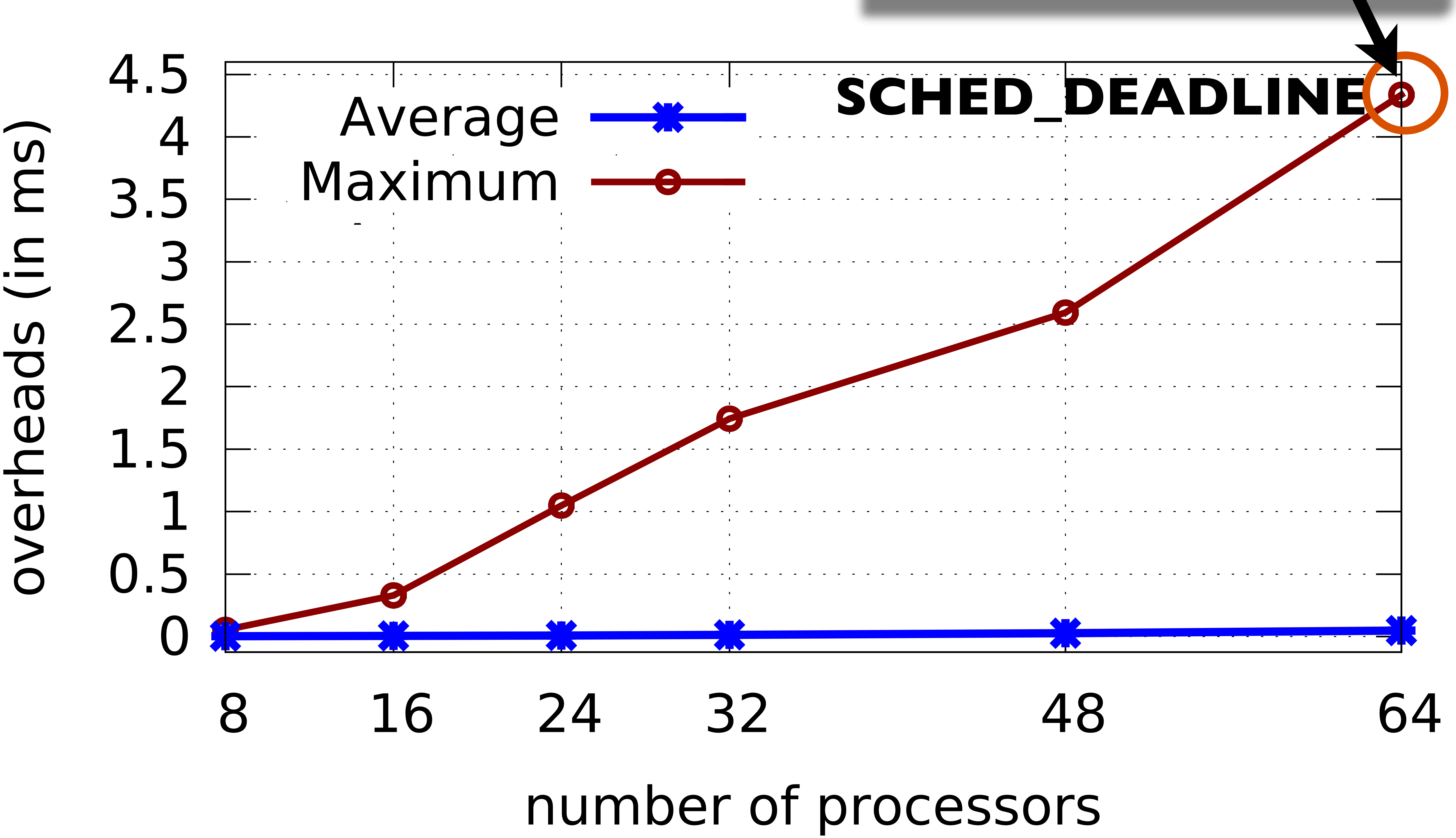


Fine-Grained Locking Fails in the Worst Case

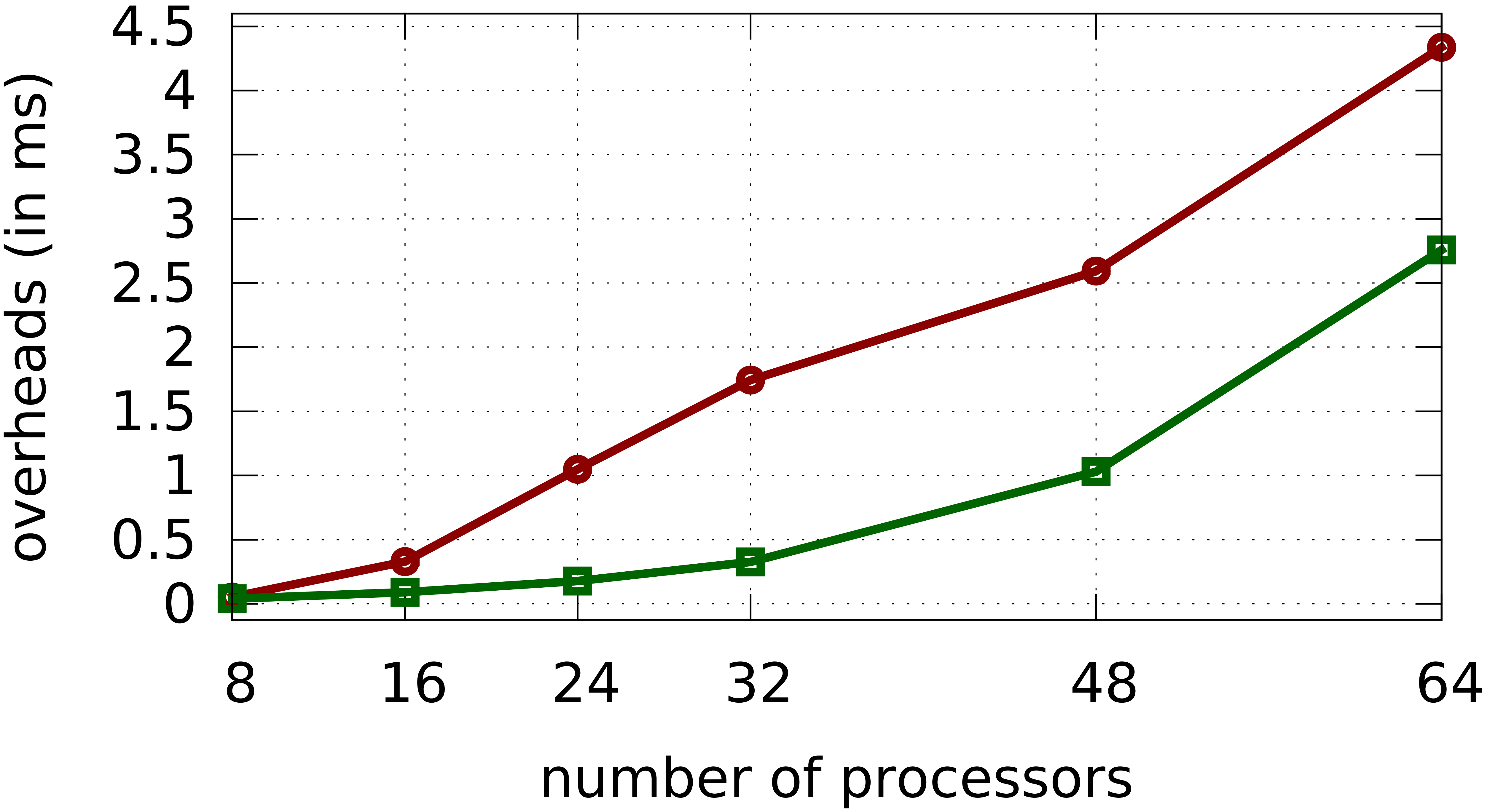


Fine-Grained
Fails in the W

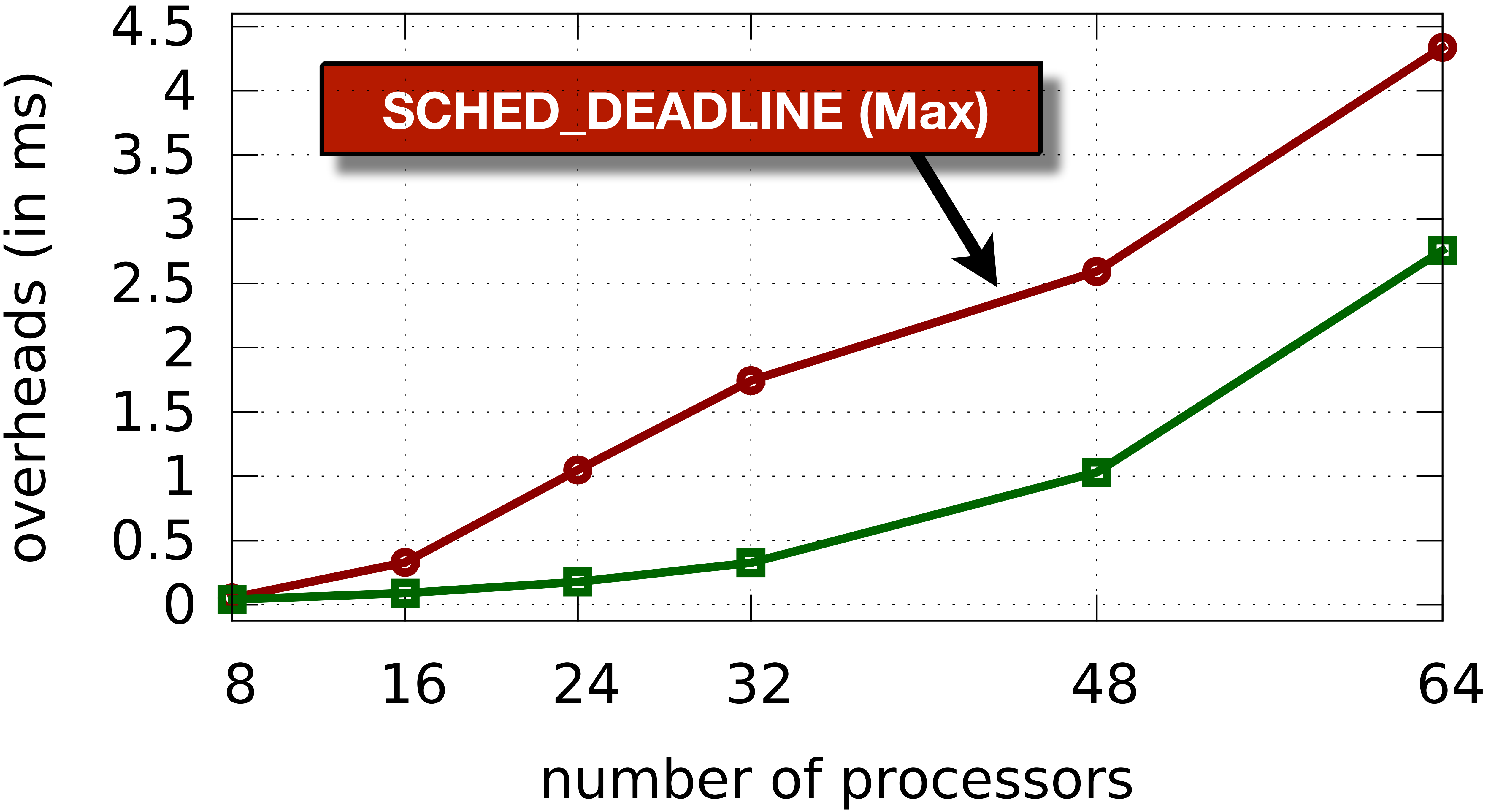
Very high overheads in the worst case!



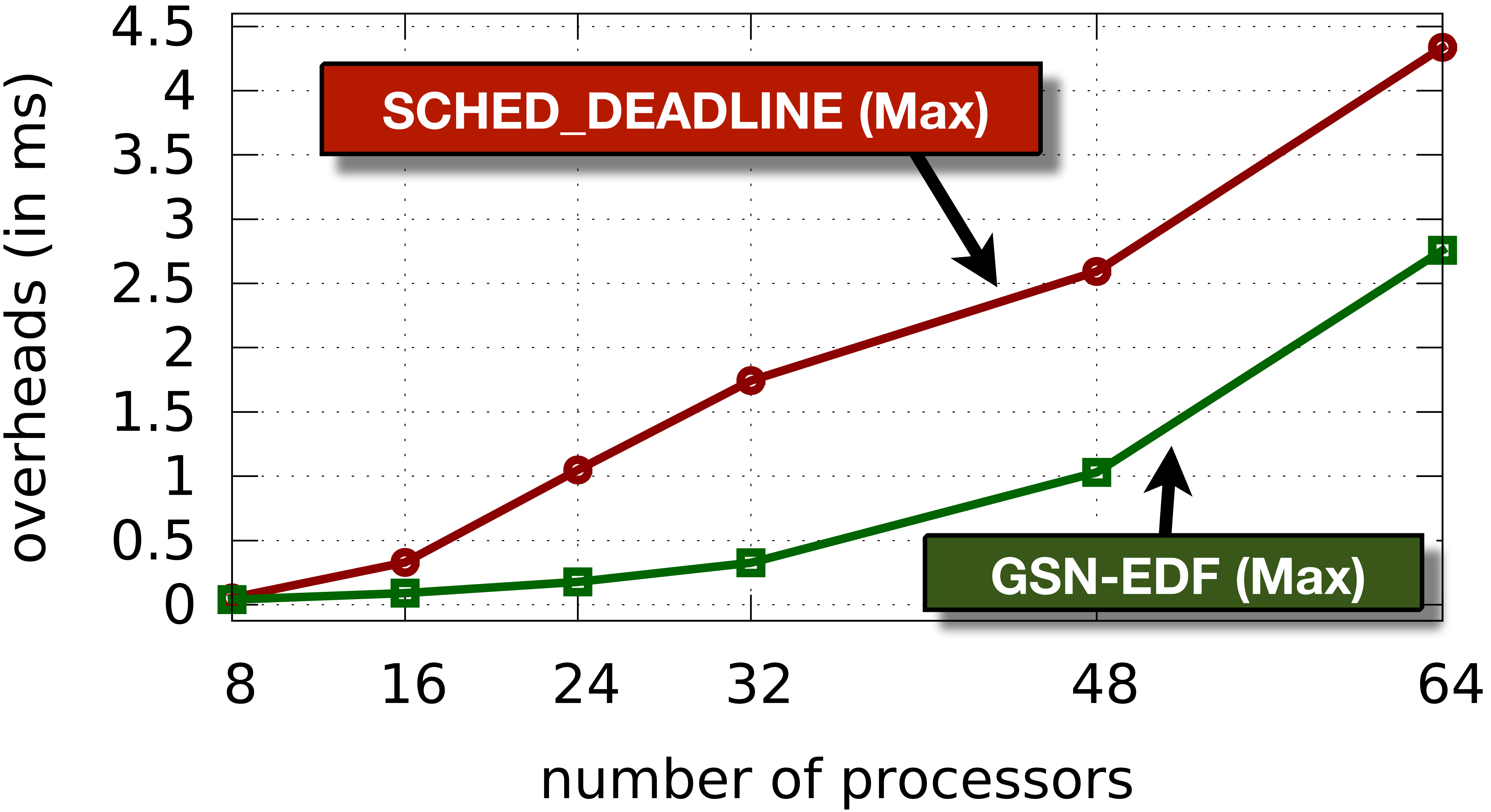
Fine-Grained vs. Coarse-Grained Locks



Fine-Grained vs. Coarse-Grained Locks

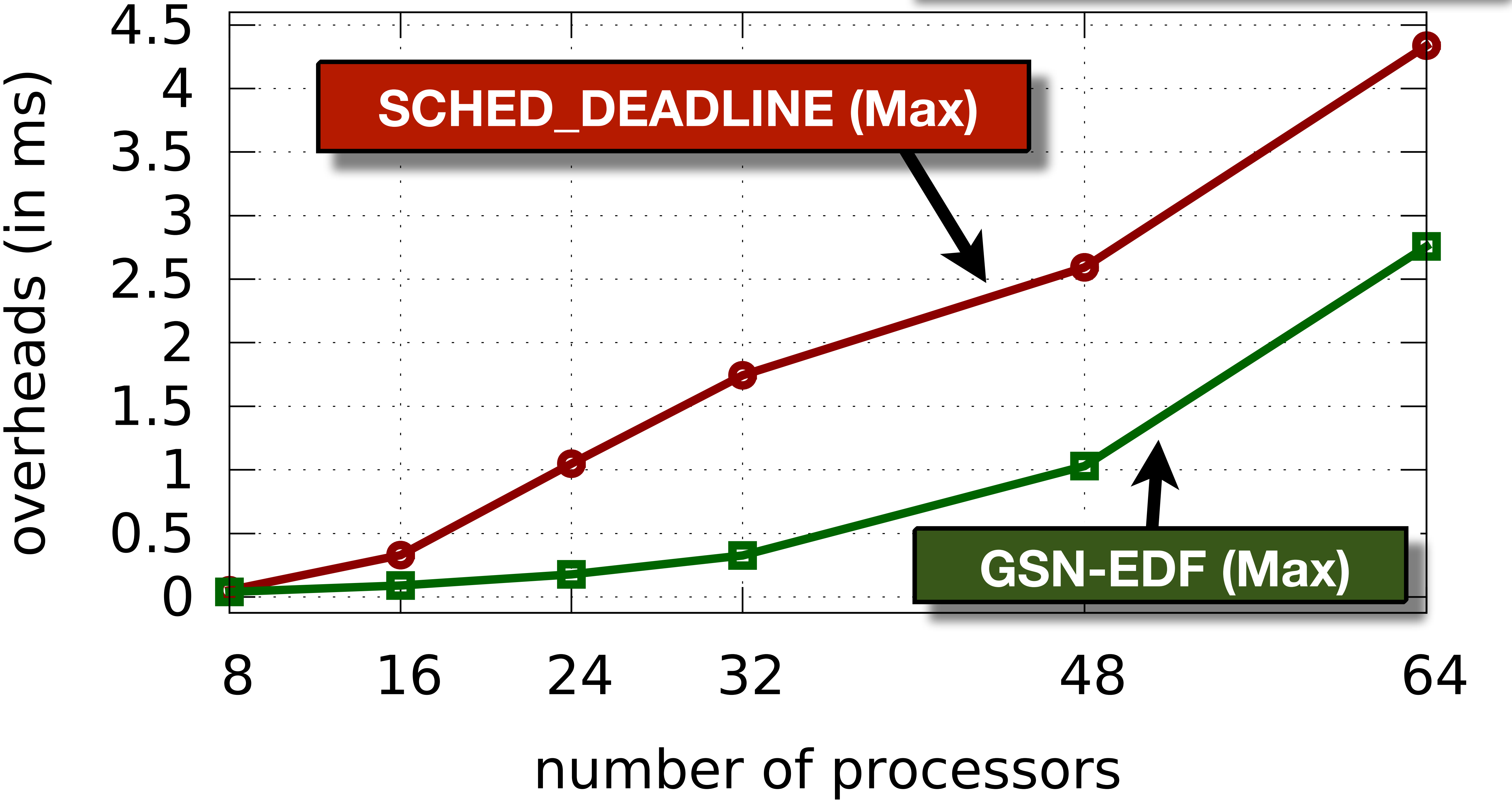


Fine-Grained vs. Coarse-Grained Locks



Fine-Grained vs Coarse-Grain

Both approaches do not scale in the worst case!



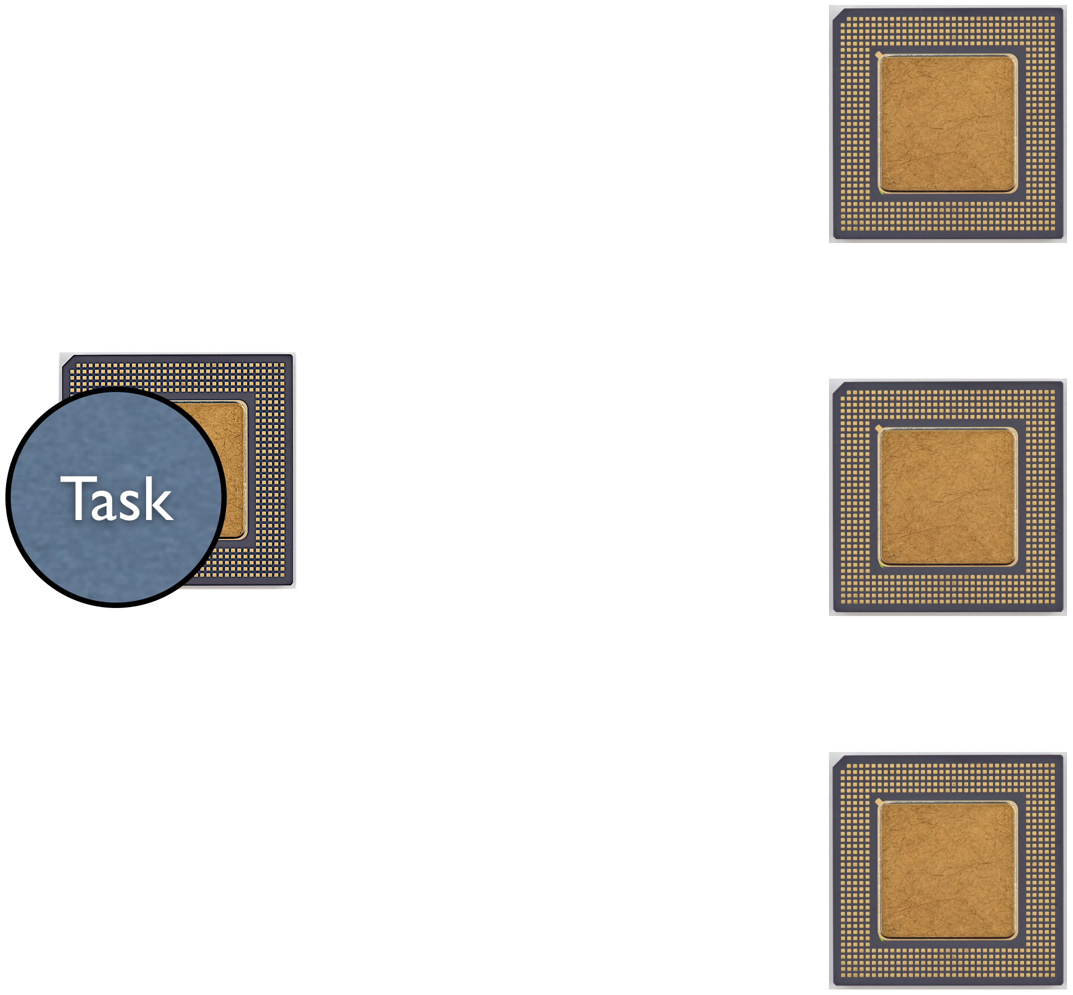
SCHED_DEADLINE (Max)

GSN-EDF (Max)

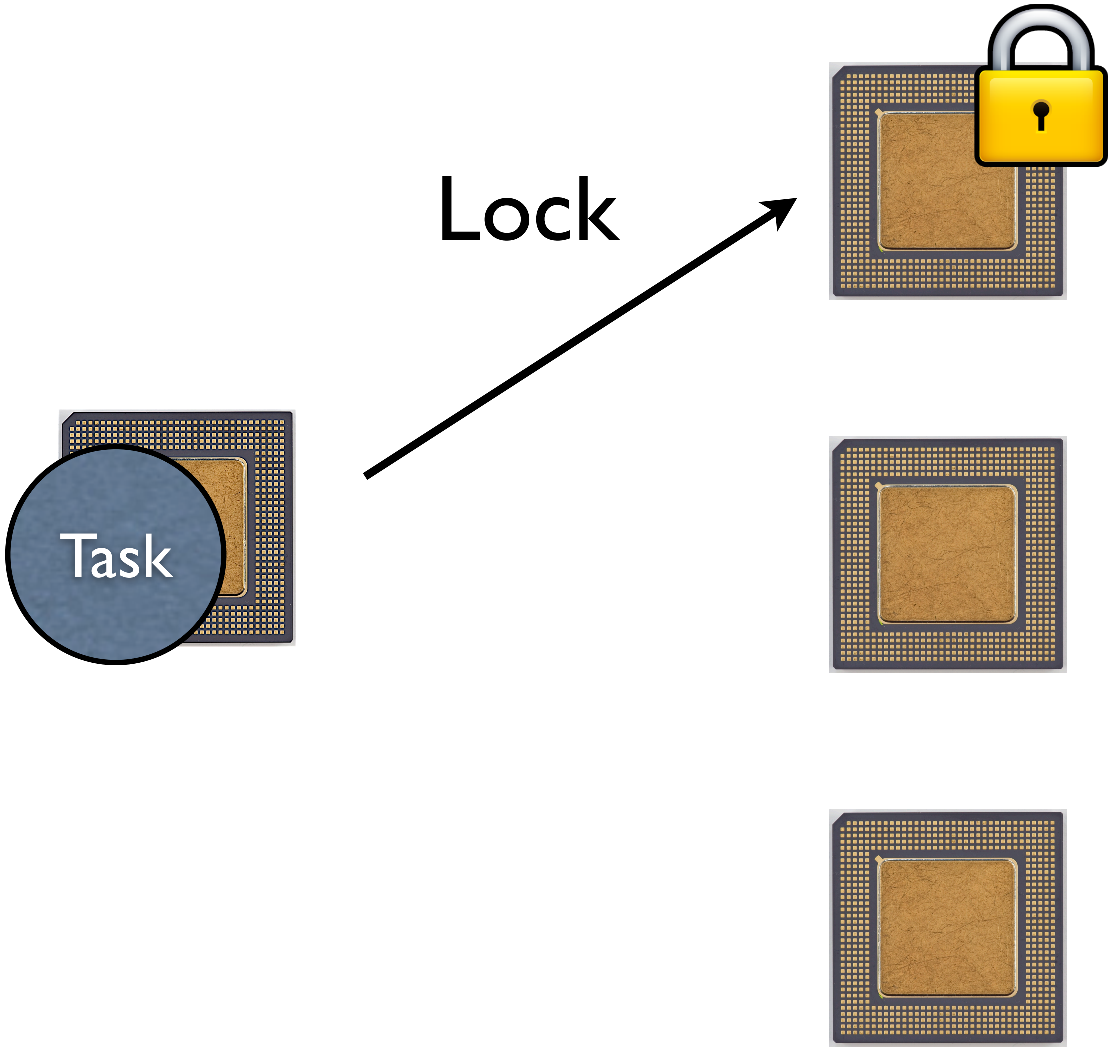
This Talk

1) Why global scheduling?
2) Current implementations
3) **Root causes of overhead**
4) How to scale global scheduling?
5) Evaluation

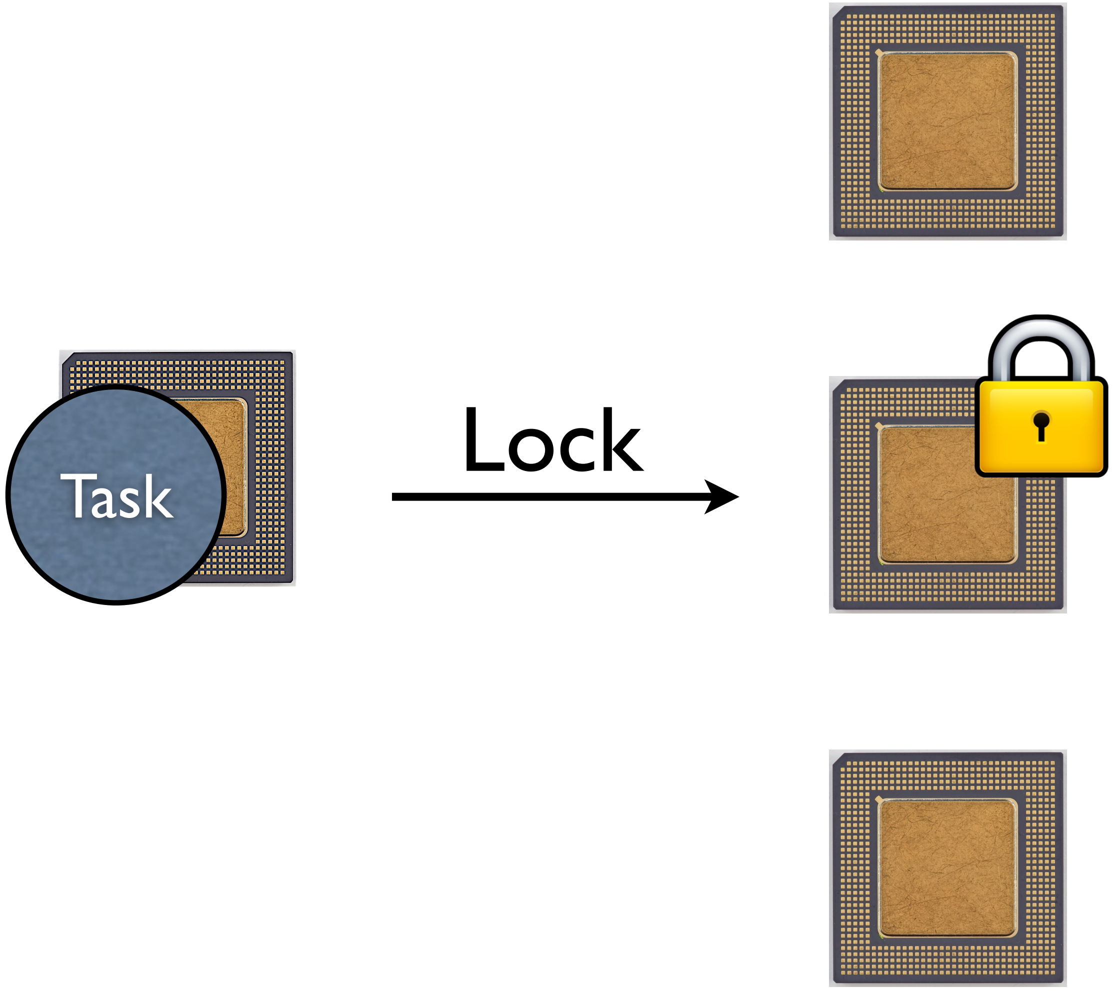
Fine-grained Locking:
Average Case



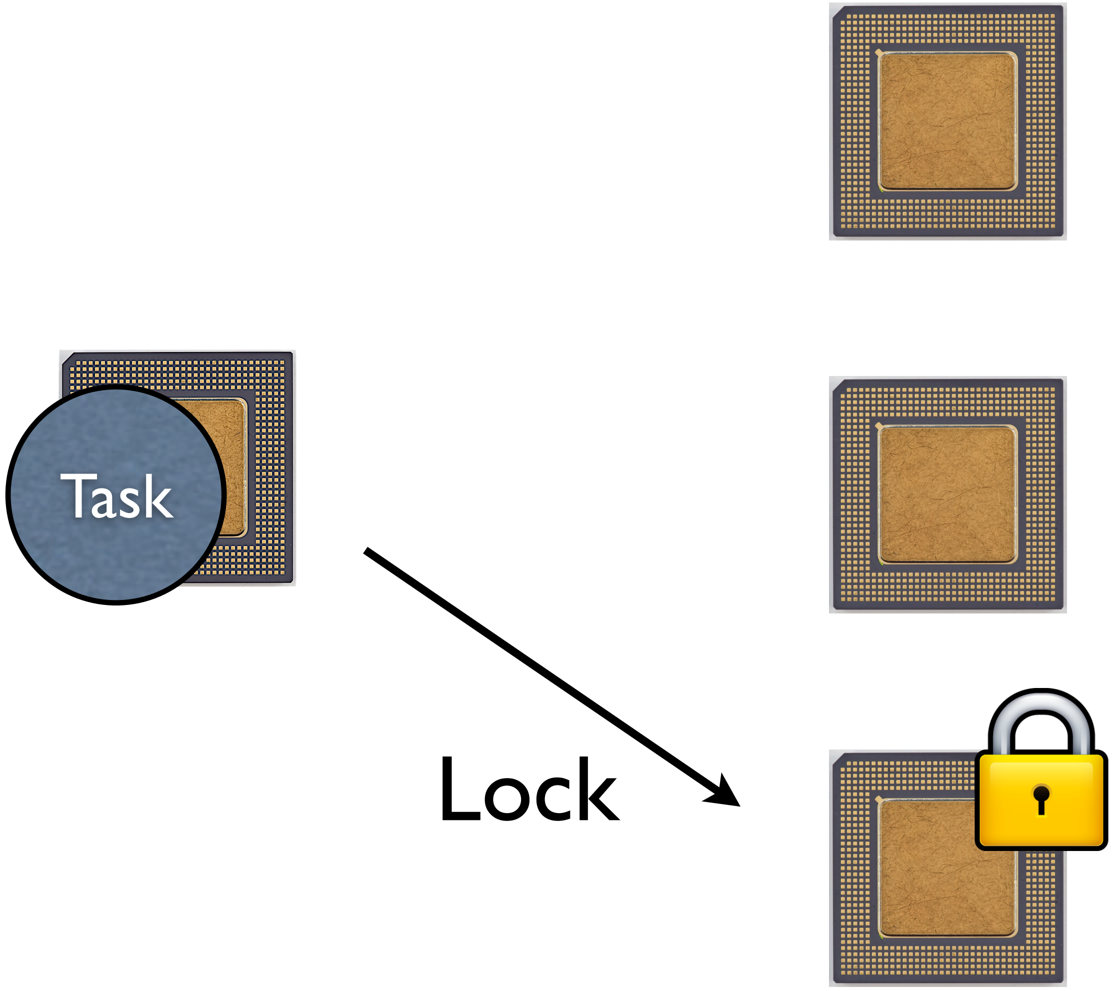
Fine-grained Locking:
Average Case



Fine-grained Locking: Average Case

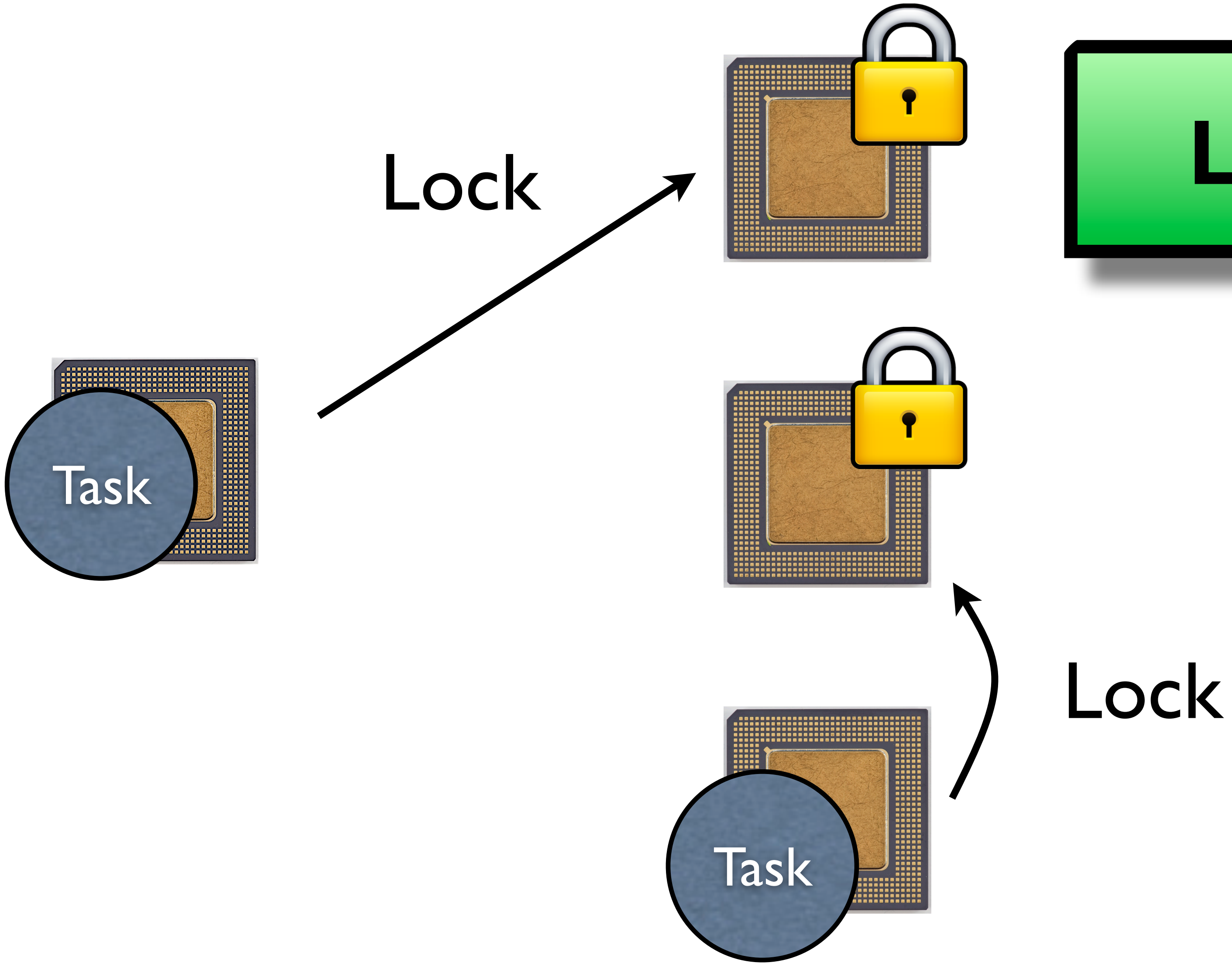


Fine-grained Locking:
Average Case



Fine-grained Locking:
Average Case

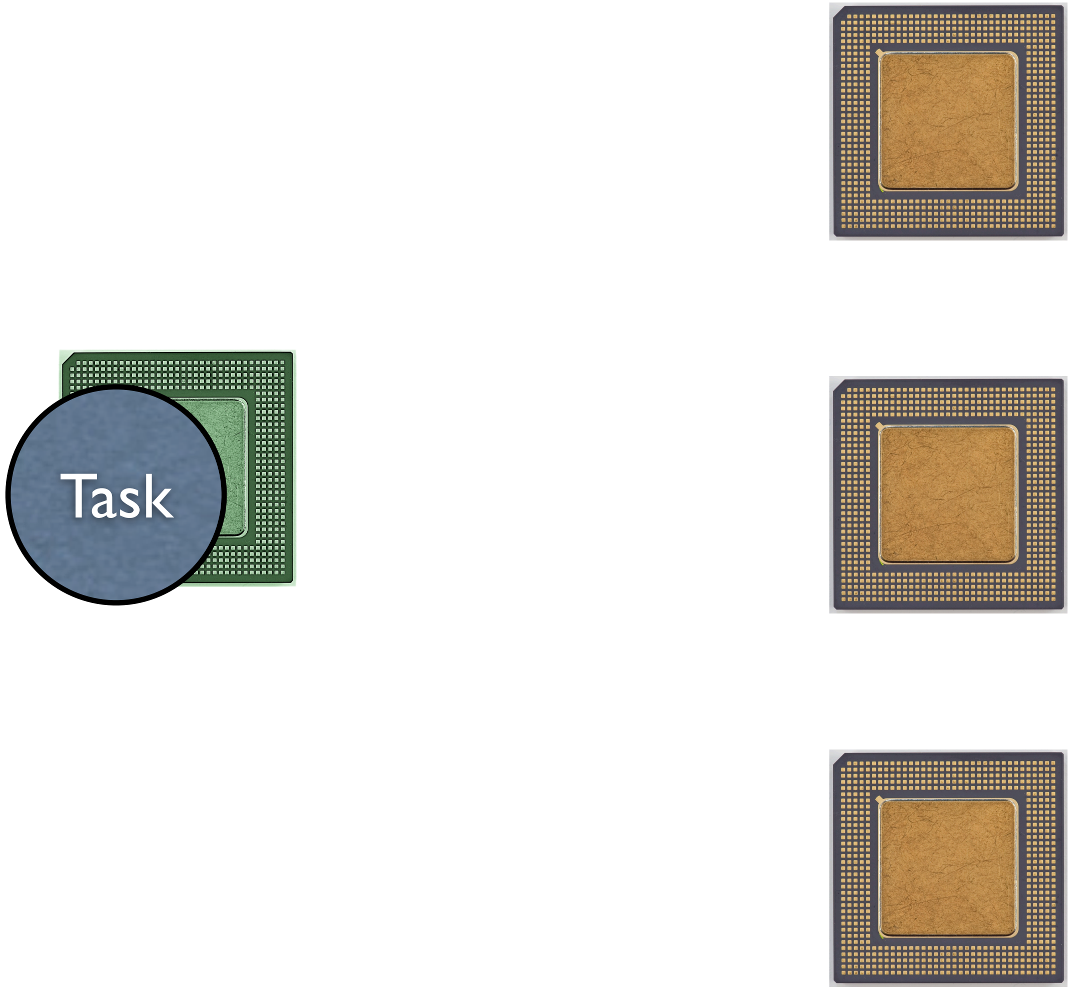
Low contention!



Fine-grained Locking:
Worst Case

Locking **every** processor:

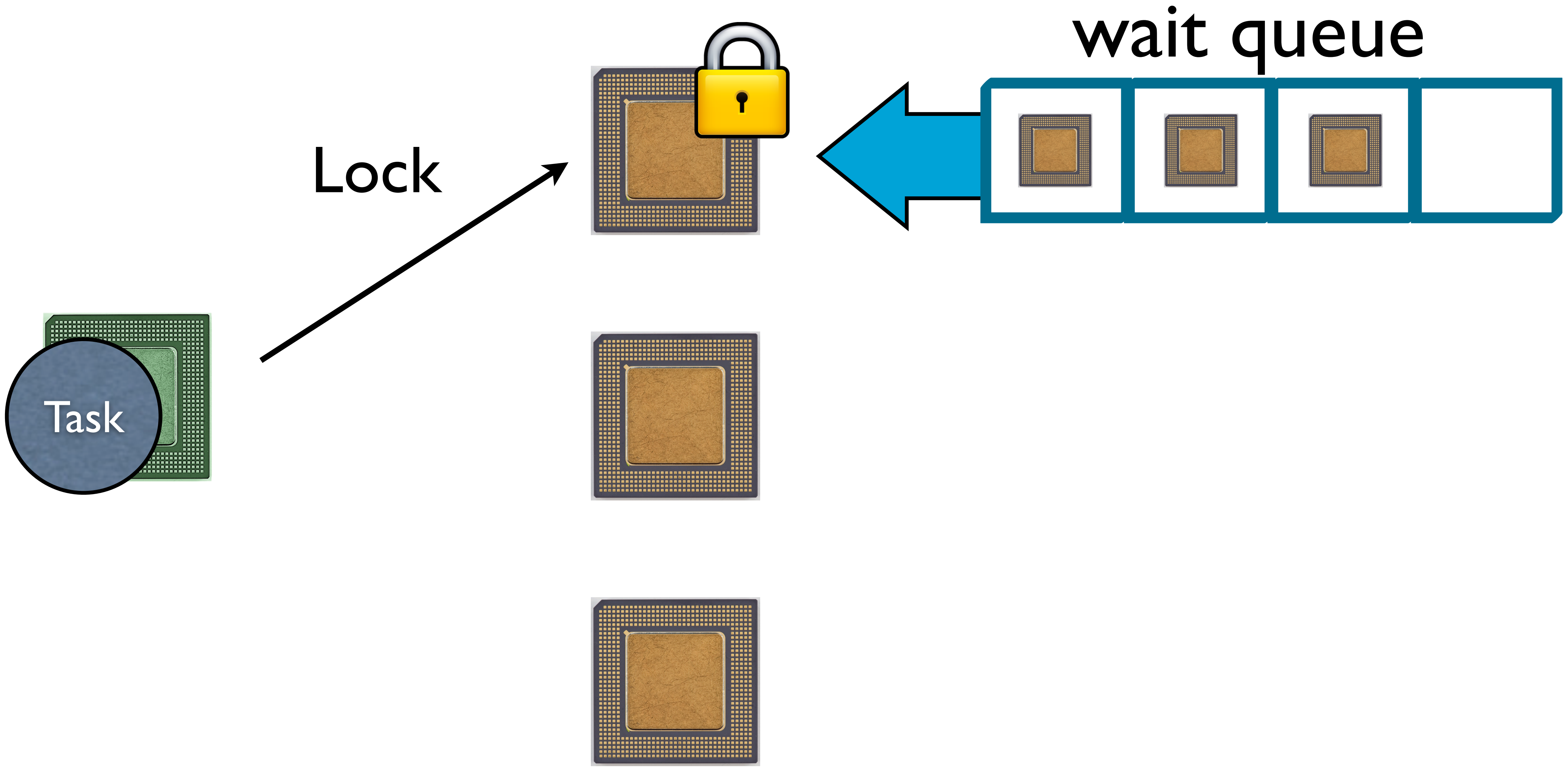
O(m) iterations



Fine-grained Locking:
Worst Case

Locking **every** processor:
O(m) iterations

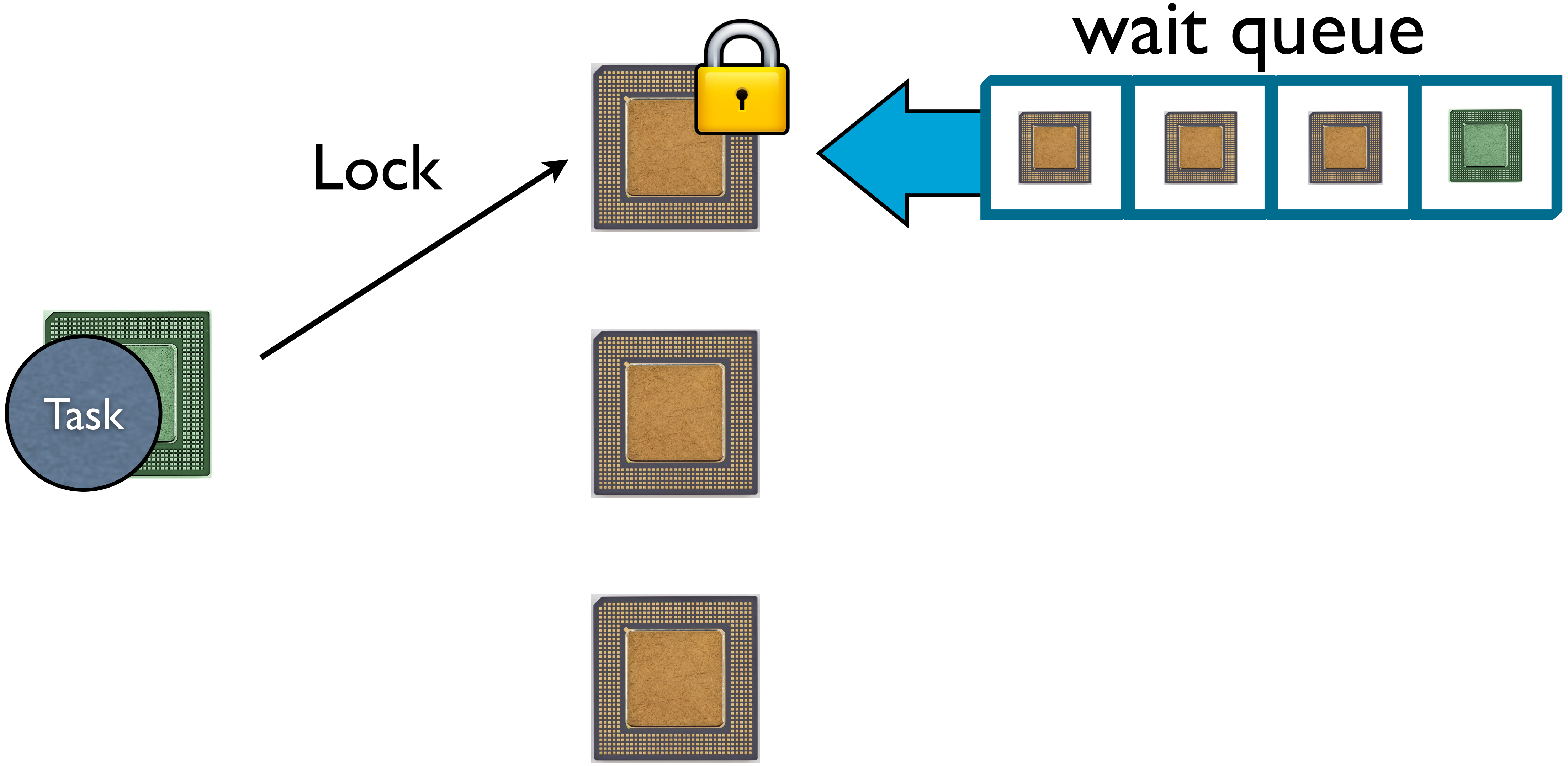
O(m) processors **already** waiting for this lock



Fine-grained Locking: Worst Case

Locking **every** processor: O(m) iterations

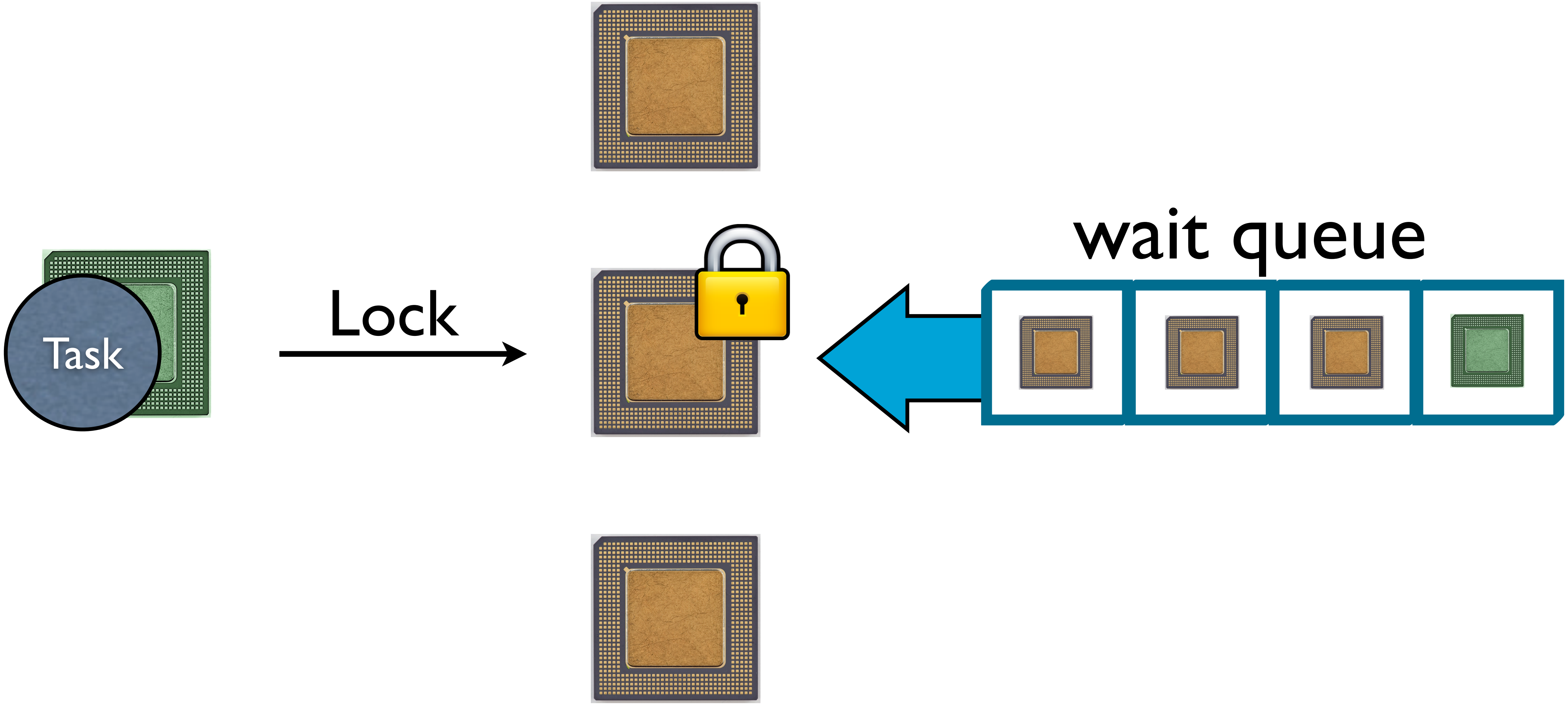
O(m) processors **already** waiting for this lock



Fine-grained Locking:
Worst Case

Locking **every** processor:
O(m) iterations

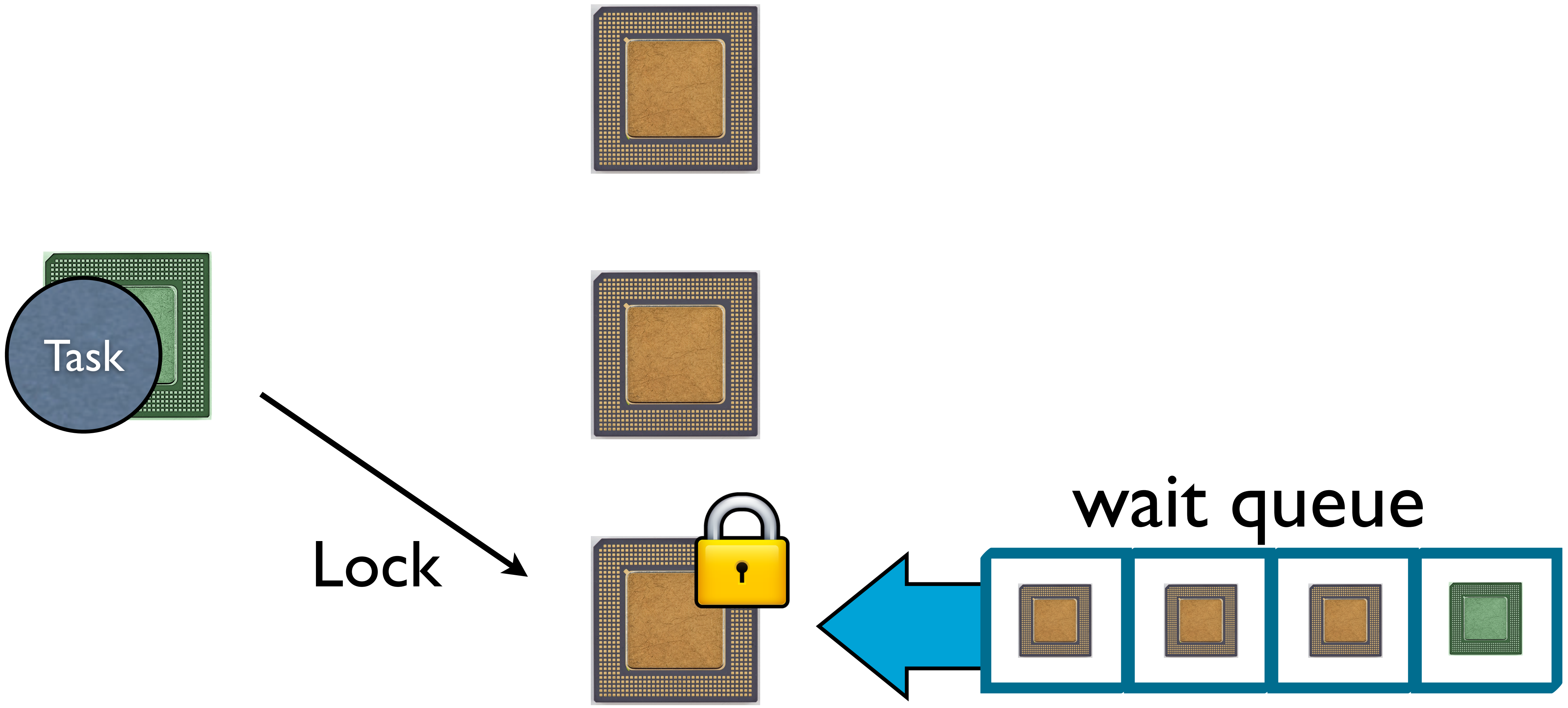
O(m) processors **already**
waiting for this lock



Fine-grained Locking: Worst Case

Locking **every** processor:
O(m) iterations

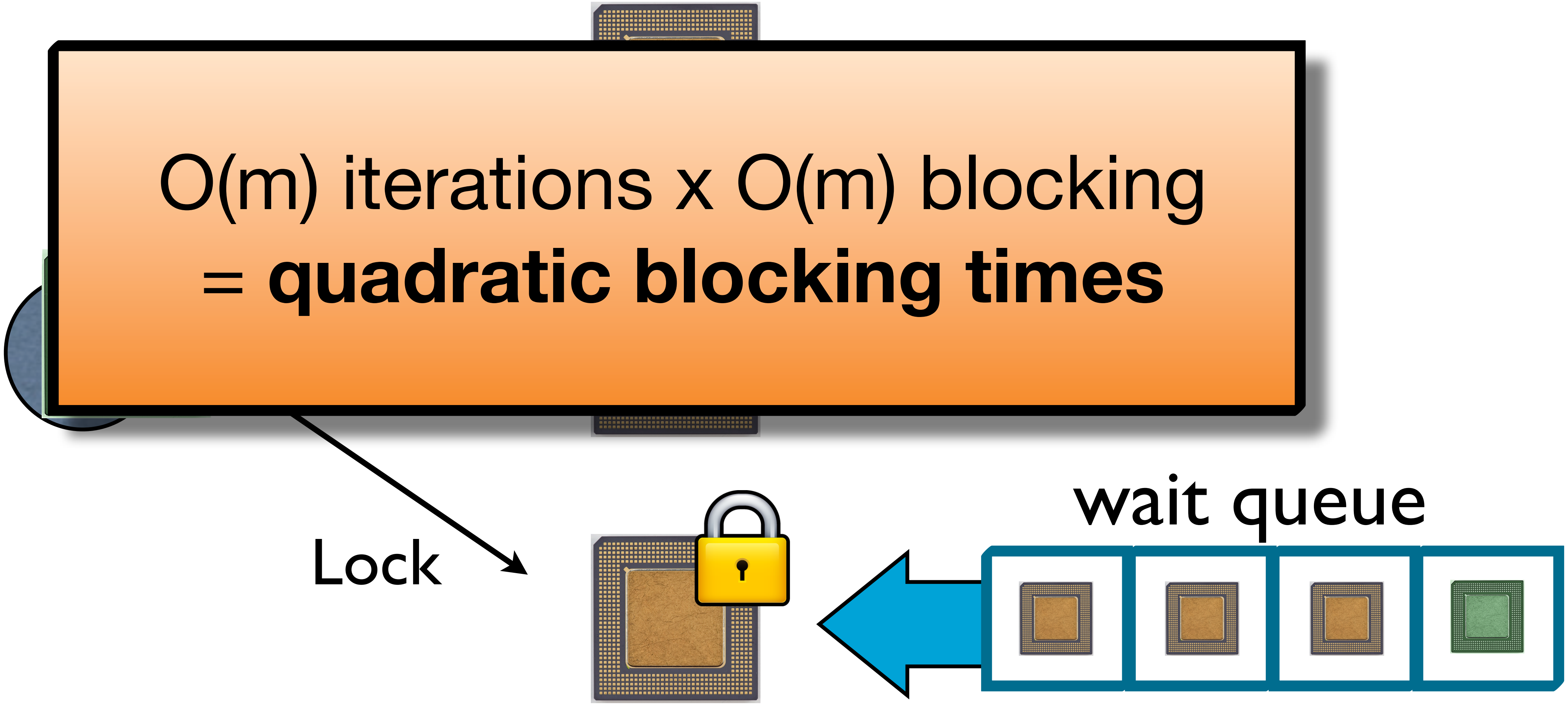
O(m) processors **already** waiting for this lock



Fine-grained Locking:
Worst Case

Locking **every** processor:
O(m) iterations

O(m) processors **already** waiting for this lock



O(m) iterations x O(m) blocking
= **quadratic blocking times**

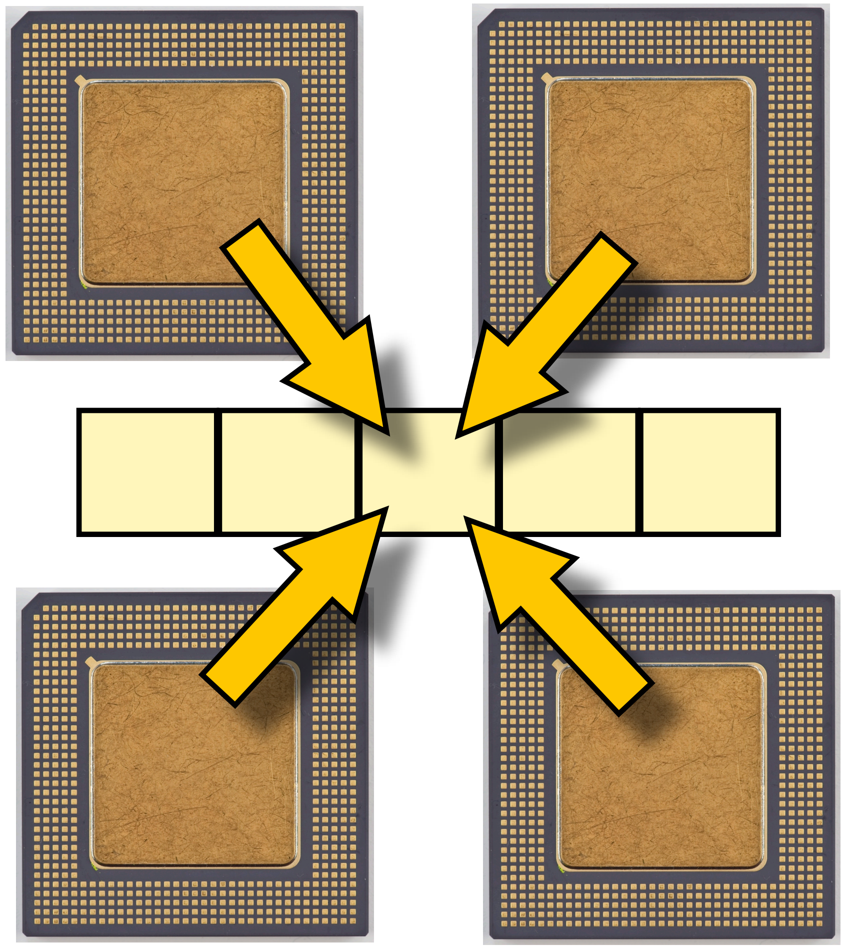
Peak Contention

Observation #1:
Peak Contention is more important than synchronization granularity with respect to **worst-case blocking.**

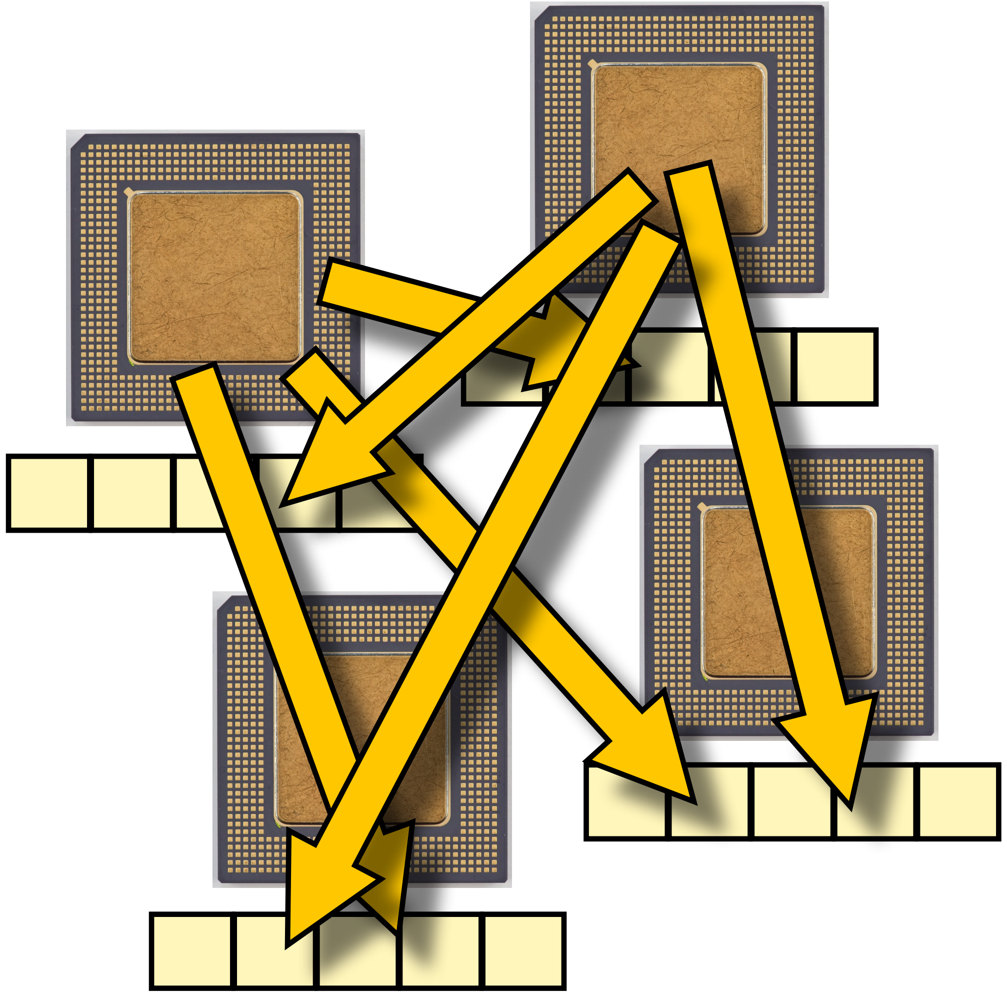
Cache-Line Bouncing

Cache-line ownership jumps from core to core

Scheduler state shared among all cores



GSN-EDF



SCHED_DEADLINE

Cache-Line Bouncing

Observation #2:
State sharing results in overheads due to **cache-line bouncing, even if it's distributed across cores**.

Root Causes of Overhead

Peak Contention

Cache-Line Bouncing

This Talk

1) Why global scheduling?
2) Current implementations
3) Root causes of overhead
4) **How to scale global scheduling?**
5) Evaluation

Candidate Solutions

Lock-free algorithms

Candidate Solutions

Lock-free algorithms

multiple CAS in the same location,
unpredictable fail-retry operations

Candidate Solutions

Lock-free algorithms

multiple CAS in the same location,
unpredictable fail-retry operations

Wait-free queue of events

Candidate Solutions

Lock-free algorithms

multiple CAS in the same location,
unpredictable fail-retry operations

Wait-free queue of events

complex garbage collection and serialization,
didn’t reduce cache-line bouncing

Candidate Solutions

Lock-free algorithms

multiple CAS in the same location,
unpredictable fail-retry operations

Wait-free queue of events

complex garbage collection and serialization,
didn’t reduce cache-line bouncing

All-to-all broadcast of events

Candidate Solutions

Lock-free algorithms

multiple CAS in the same location, unpredictable fail-retry operations

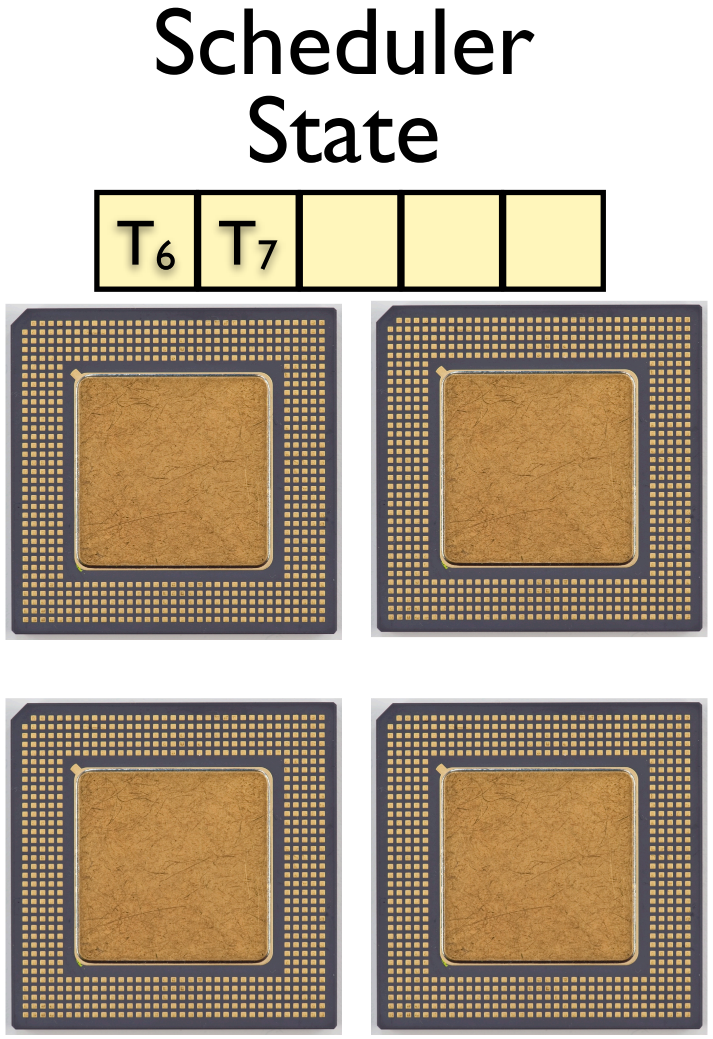
Wait-free queue of events

complex garbage collection and serialization, didn’t reduce cache-line bouncing

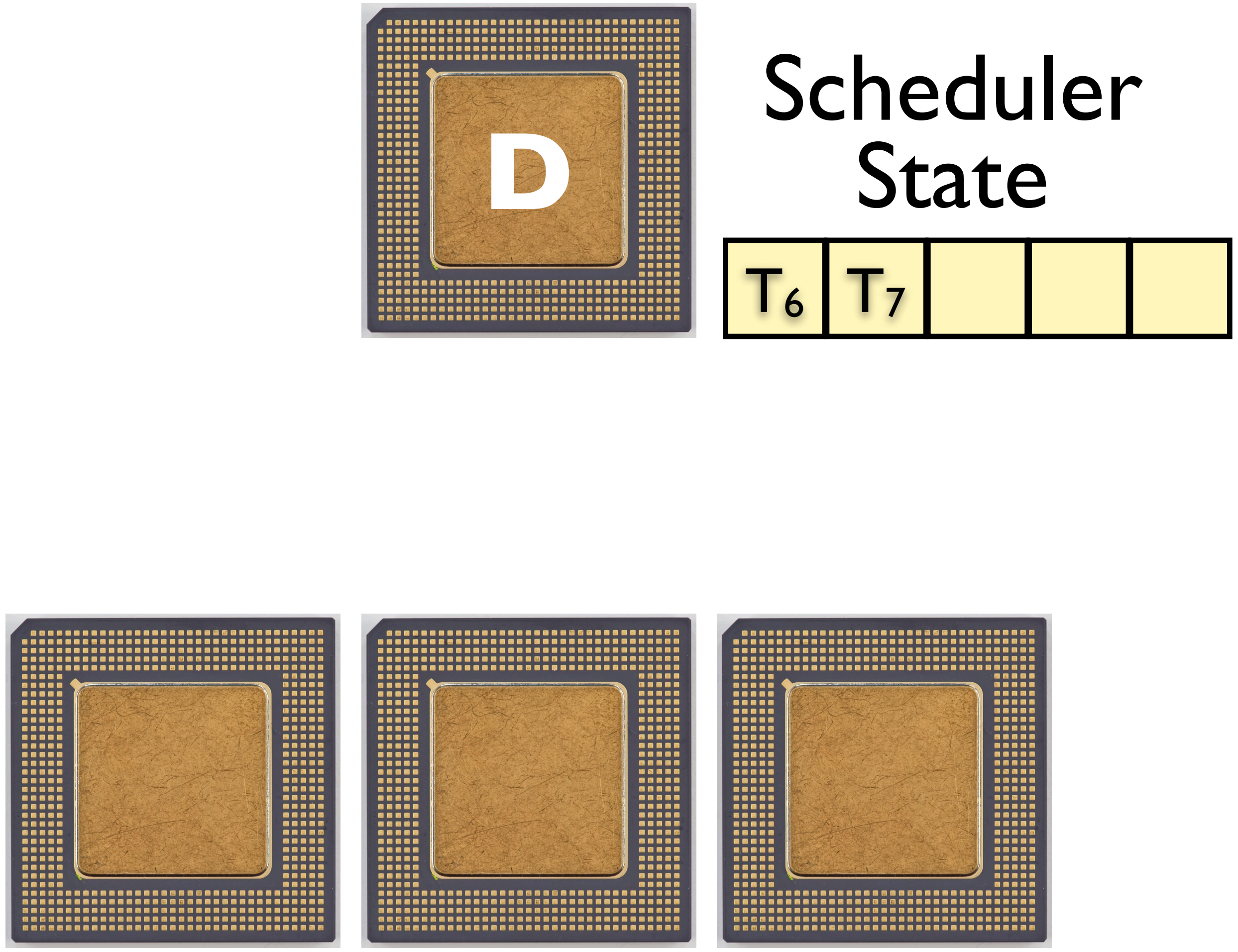
All-to-all broadcast of events

message ordering, consensus

Reducing Cache-Line Bouncing



Reducing Cache-Line Bouncing



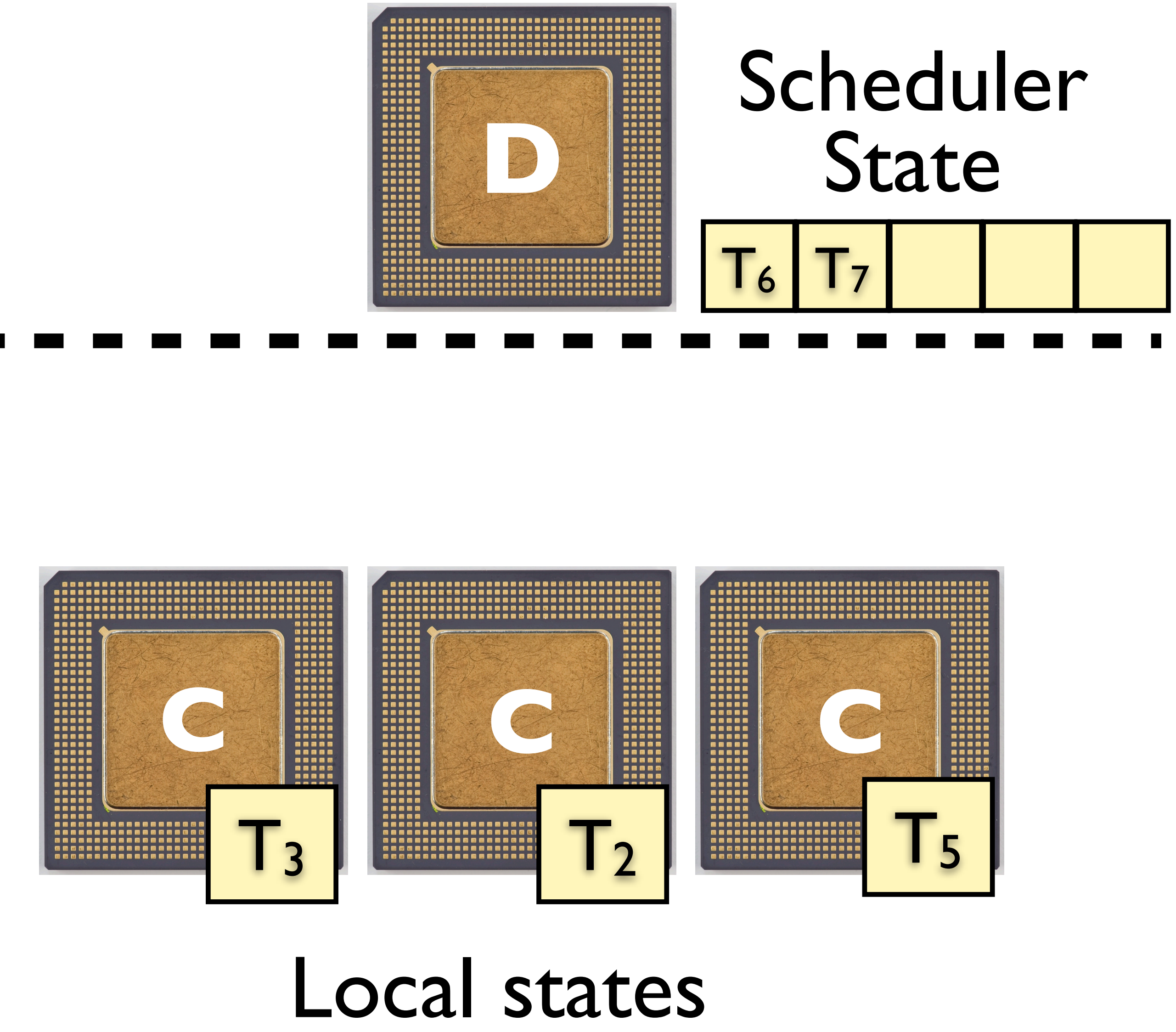
Scheduler State

T_6	T_7			

Dedicated Scheduler Processor

- Stores the full scheduler state
- Dedicated interrupt handling

Reducing Cache-Line Bouncing



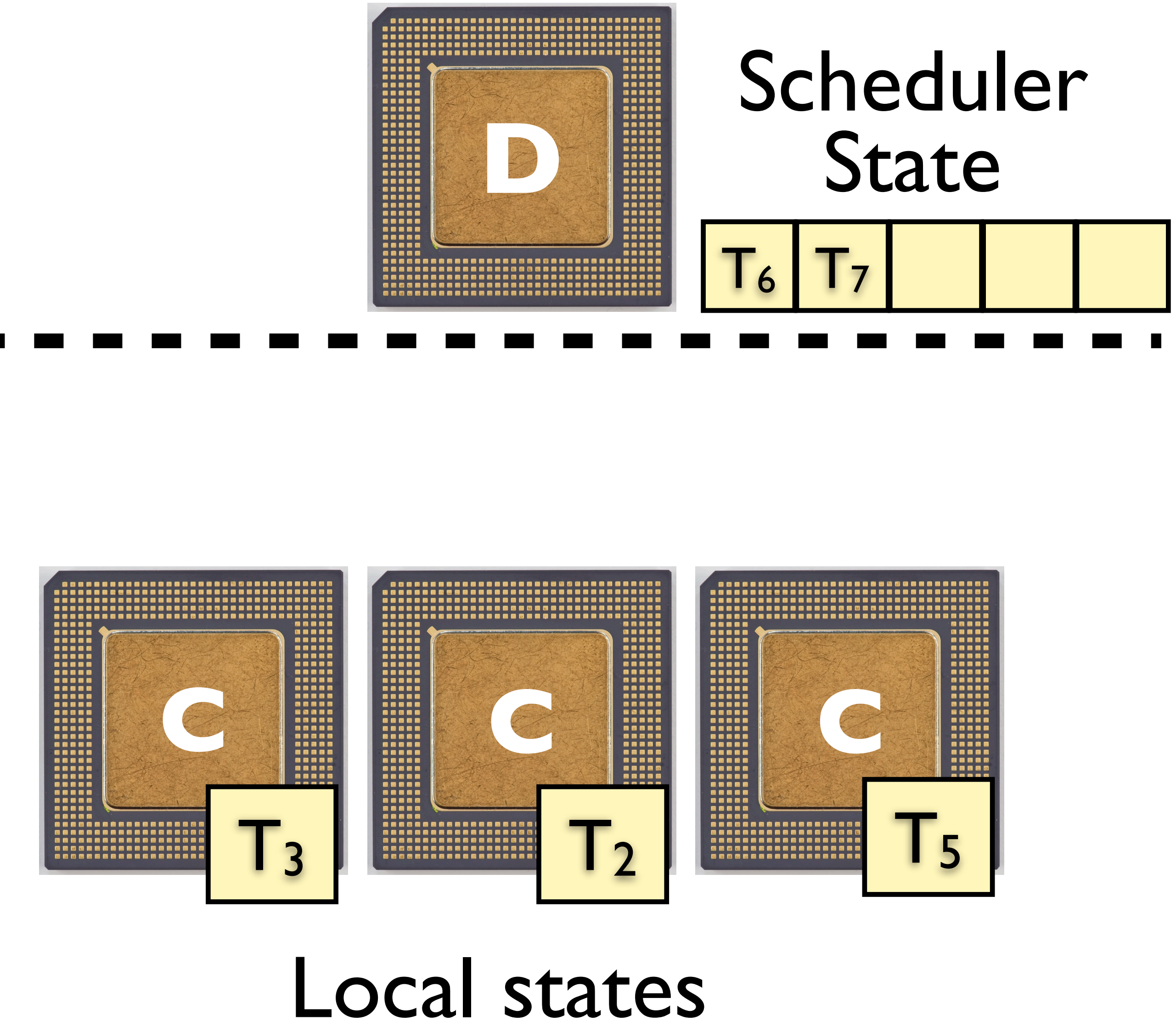
Dedicated Scheduler Processor

- Stores the full scheduler state
- Dedicated interrupt handling

Client Processors

- Only know which task they should schedule (local state)

Reducing Cache-Line Bouncing



Dedicated Scheduler Processor

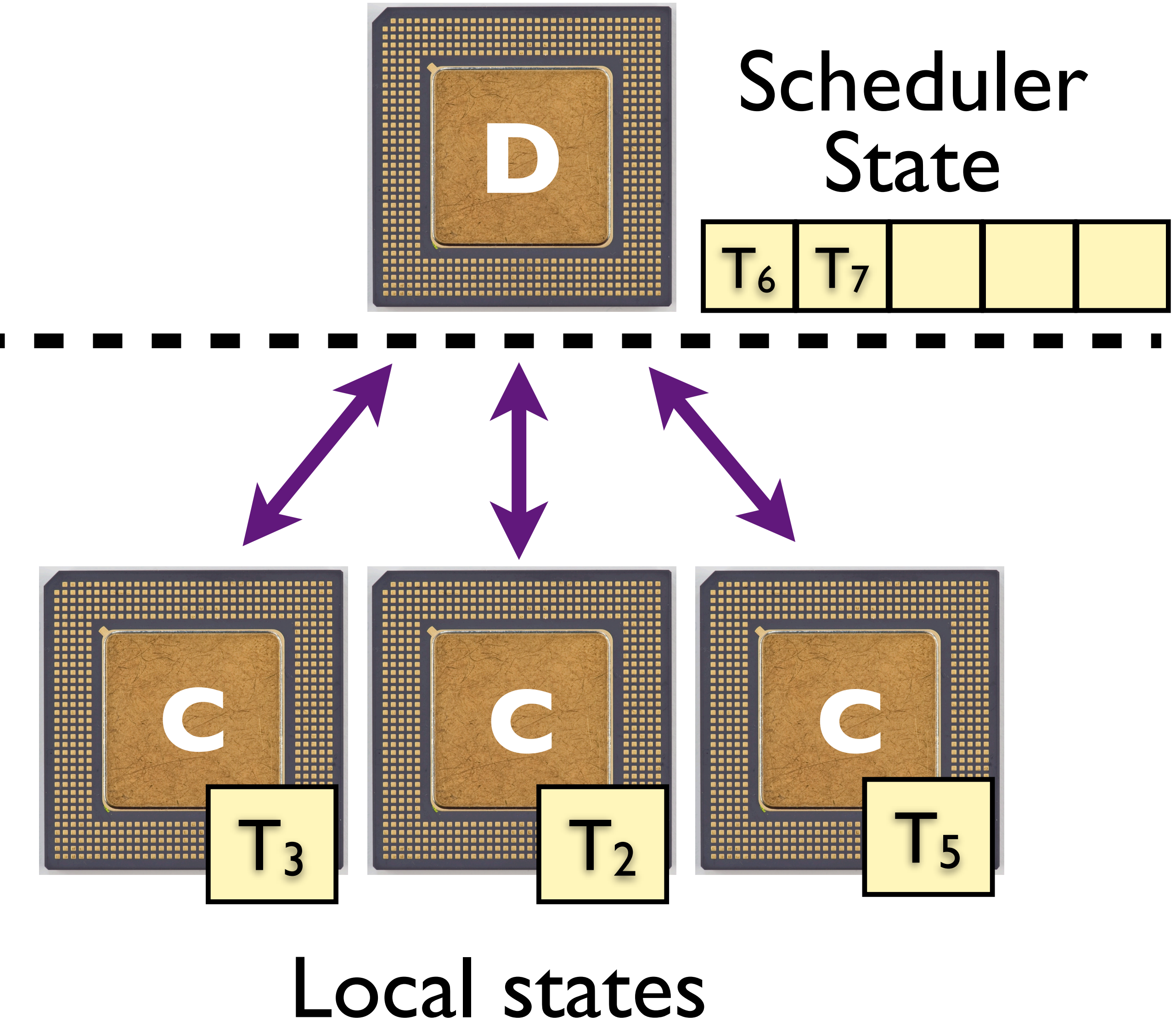
- Stores the full scheduler state
- Dedicated interrupt handling

Client Processors

- Only know which task they should schedule (local state)

Centralized state reduces sharing

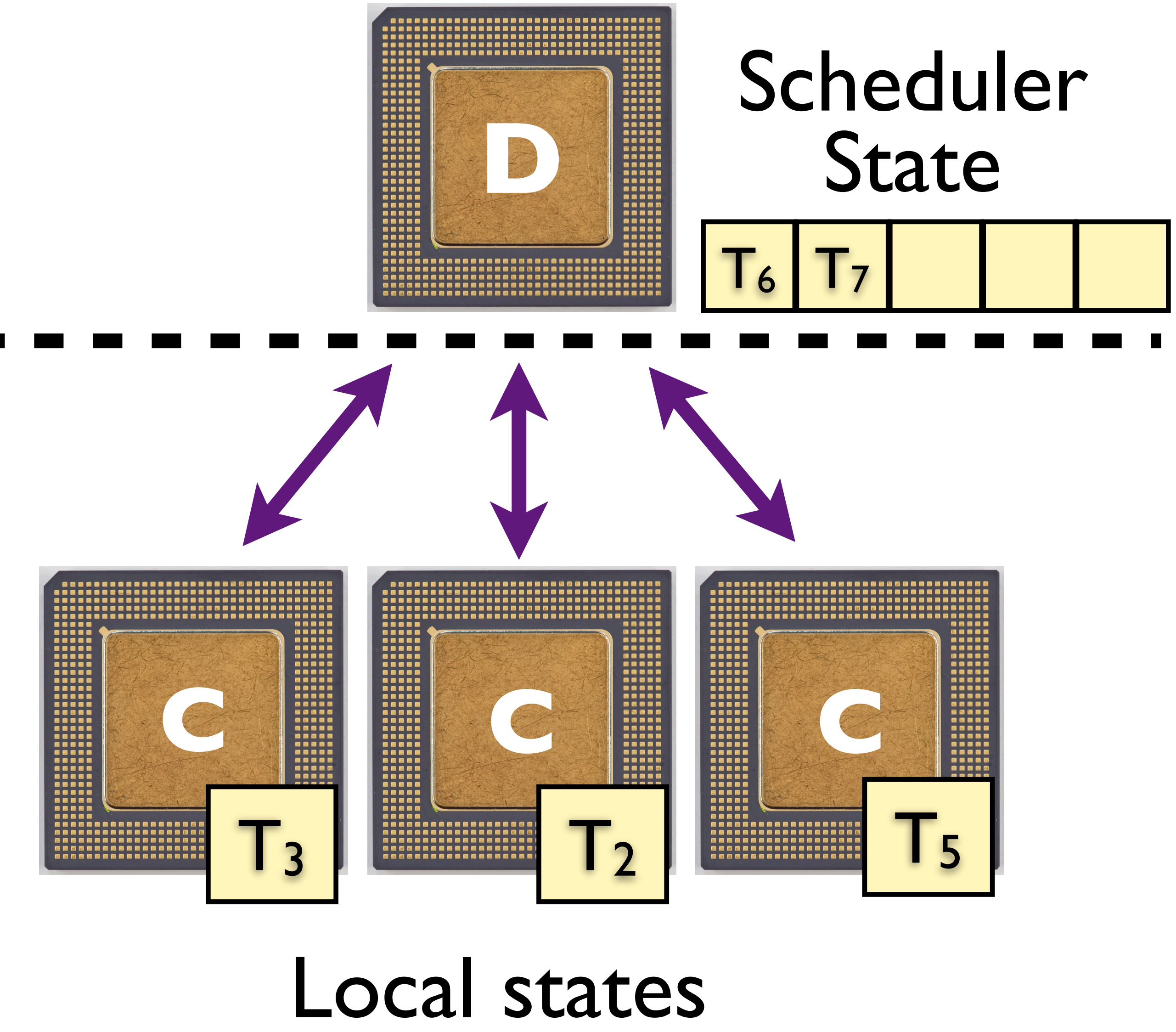
Communication with low Peak Contention



Centralized coordination

- No interaction among clients
- Low-cost communication via message passing

Communication with low Peak Contention

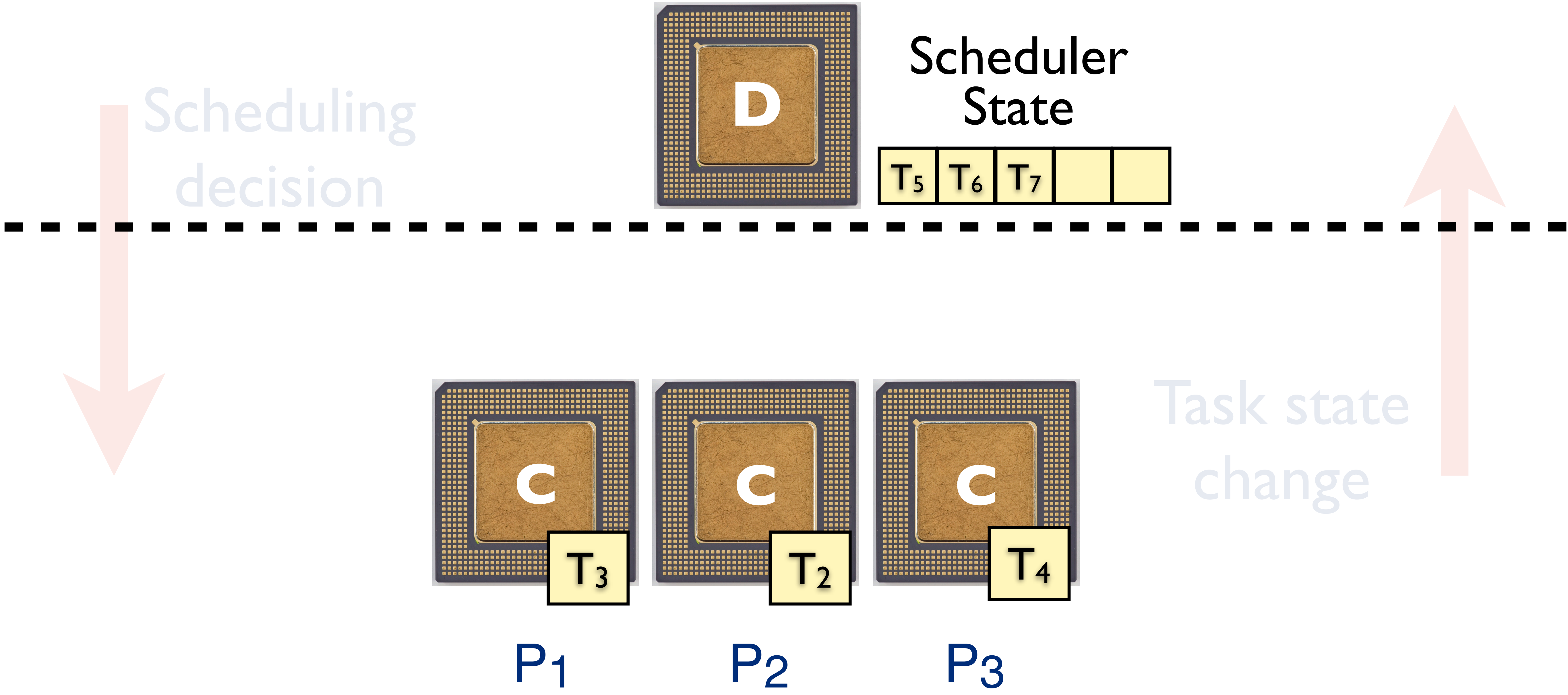


Centralized coordination

- No interaction among clients
- Low-cost communication via message passing

Contention limited to at most two processors

Message Passing



Scheduling decision

D

Scheduler State

T_5 T_6 T_7

C T_3

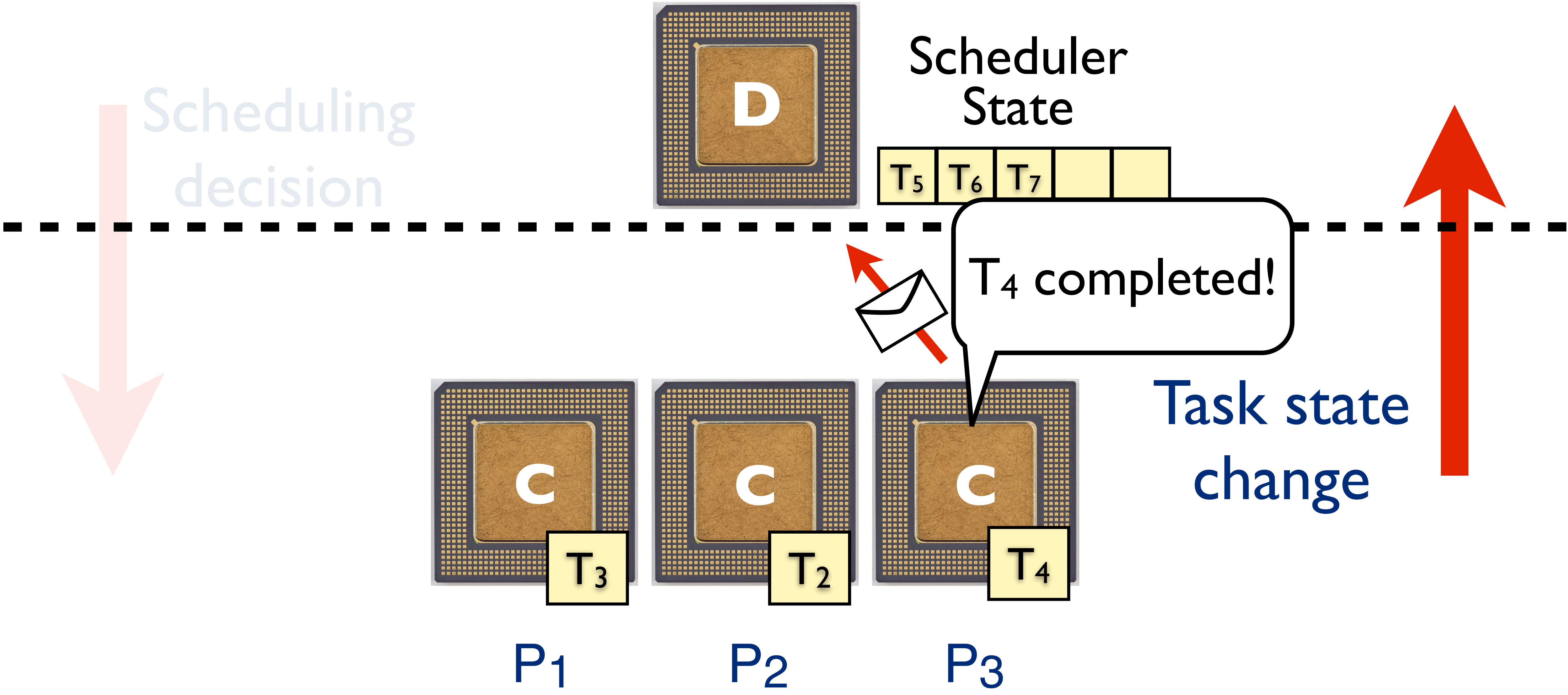
C T_2

C T_4

Task state change

P_1 P_2 P_3

Message Passing



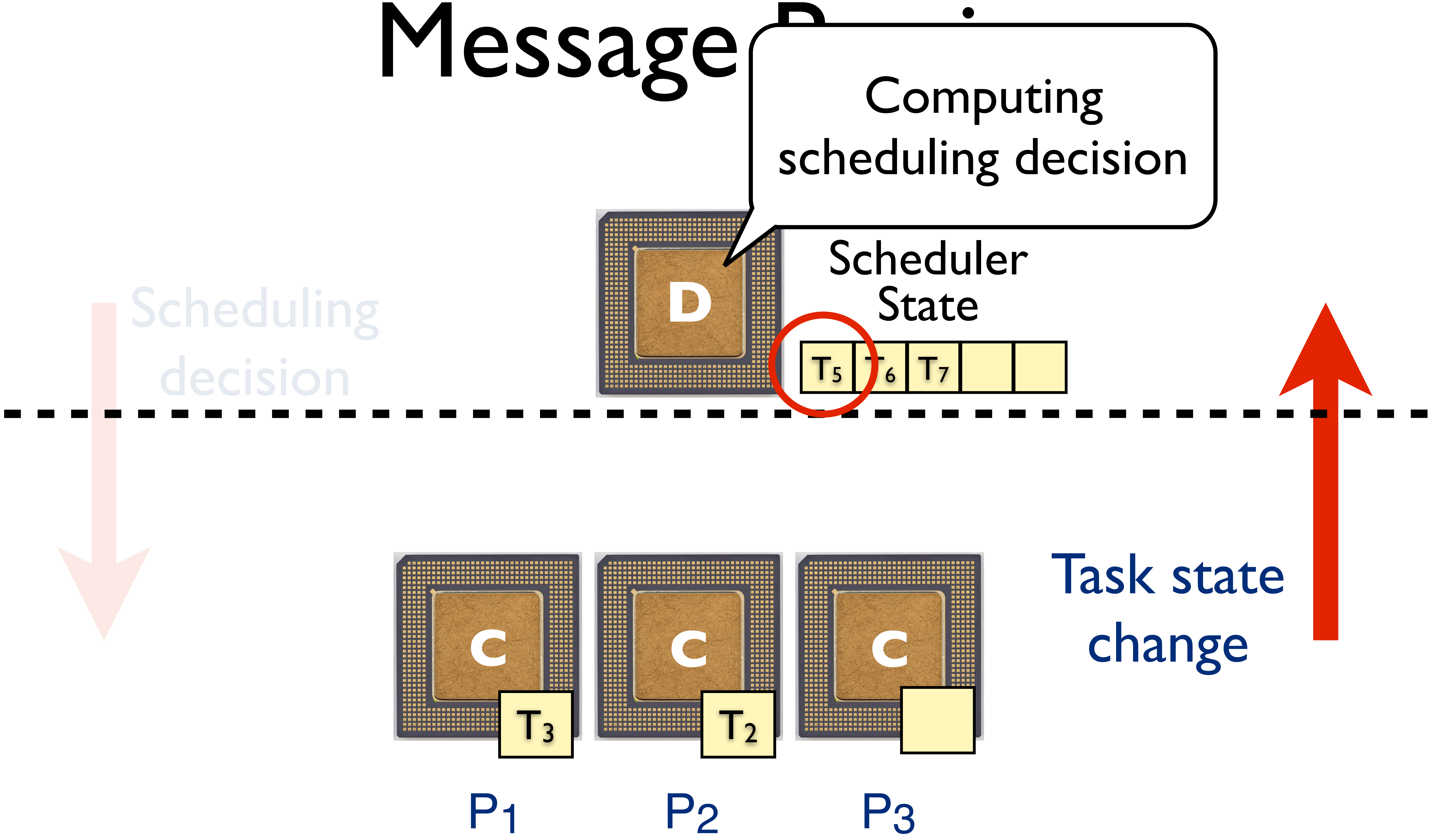
Scheduling decision

Scheduler State

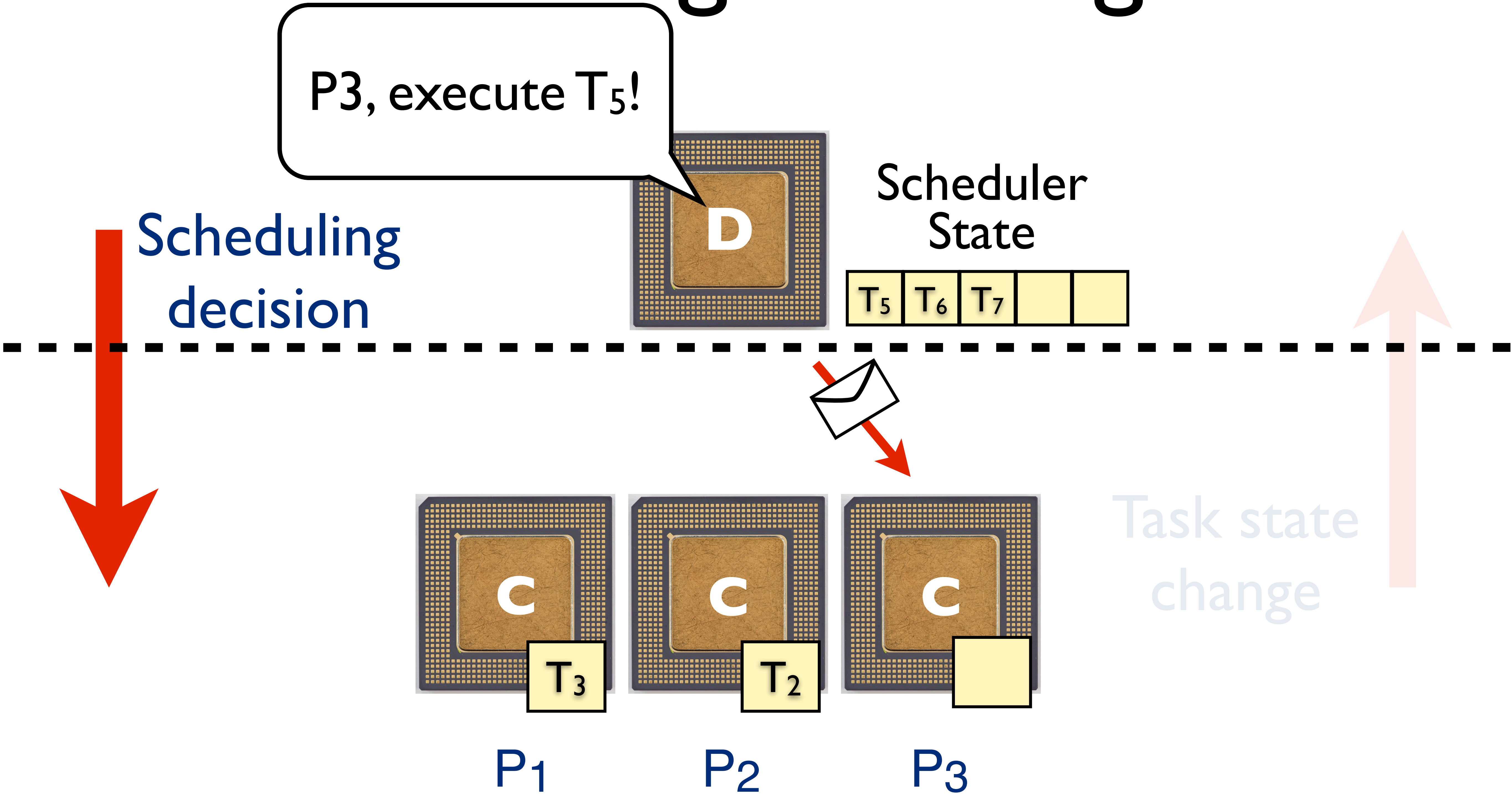
T_4 completed!

Task state change

Message P


Computing scheduling decision
Scheduler State
D
T5
T6
T7
Scheduling decision
C
C
C
T3
T2
P1
P2
P3
Task state change

Message Passing



P3, execute T_5!

D

Scheduler State

T_5 T_6 T_7

Scheduling decision

Task state change

C T_3

C T_2

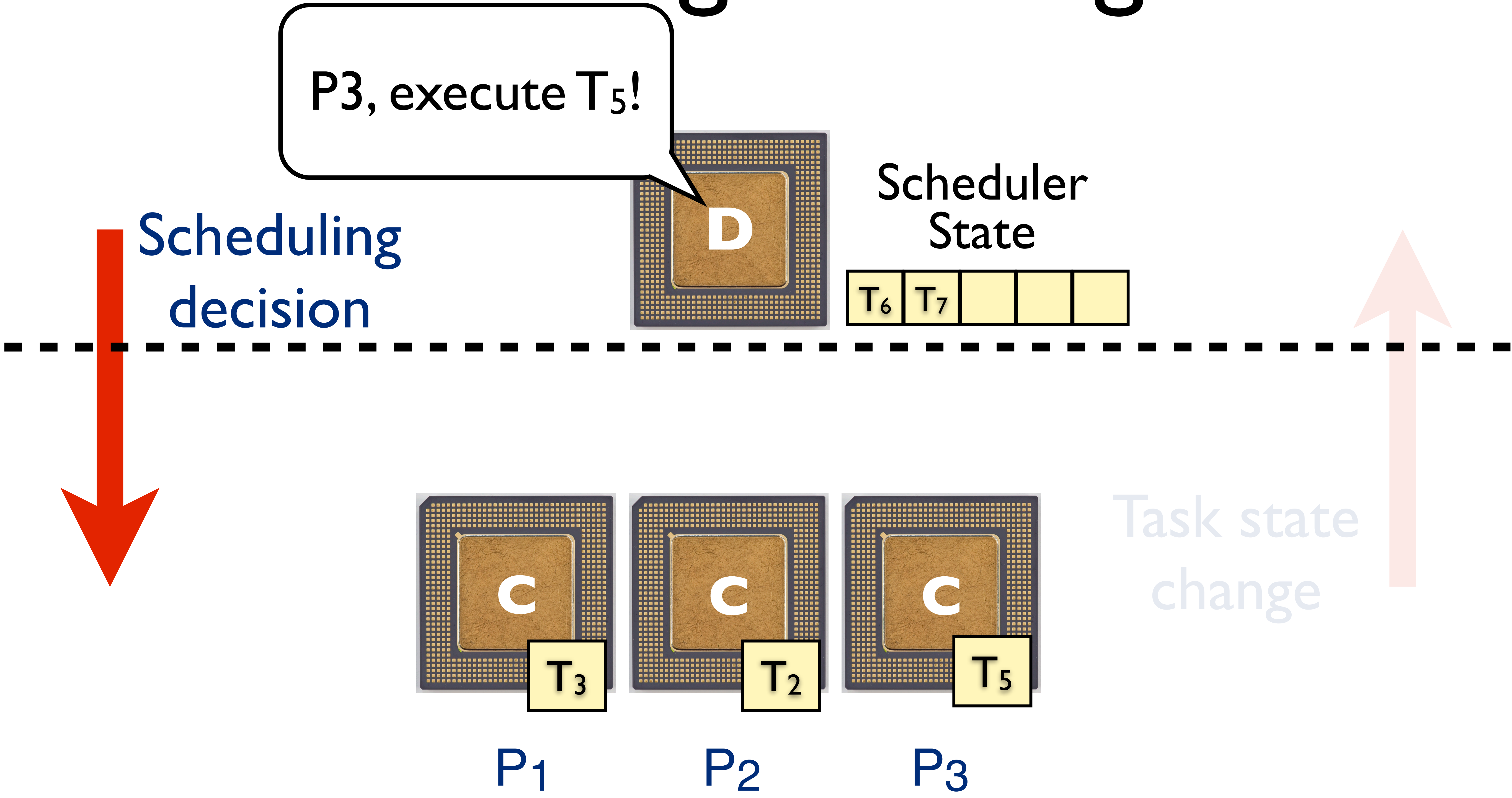
C

P_1

P_2

P_3

Message Passing



P3, execute T_5!

Scheduler State

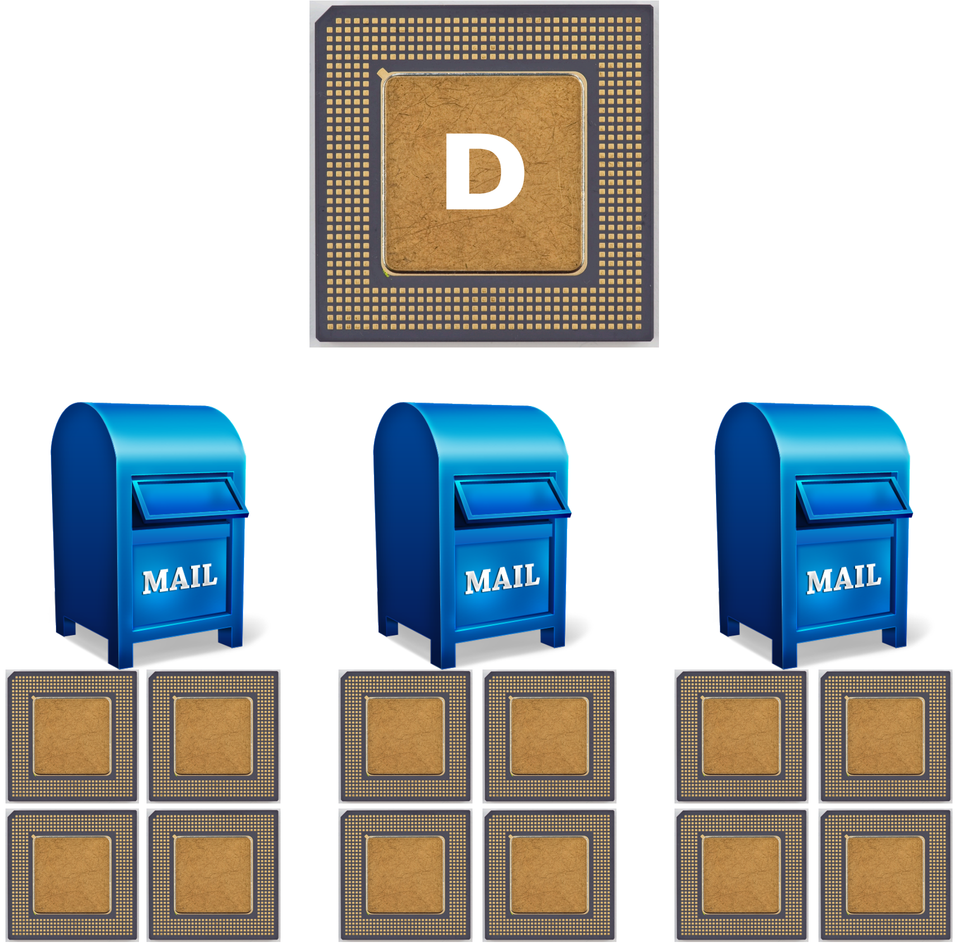
Scheduling decision

Task state change

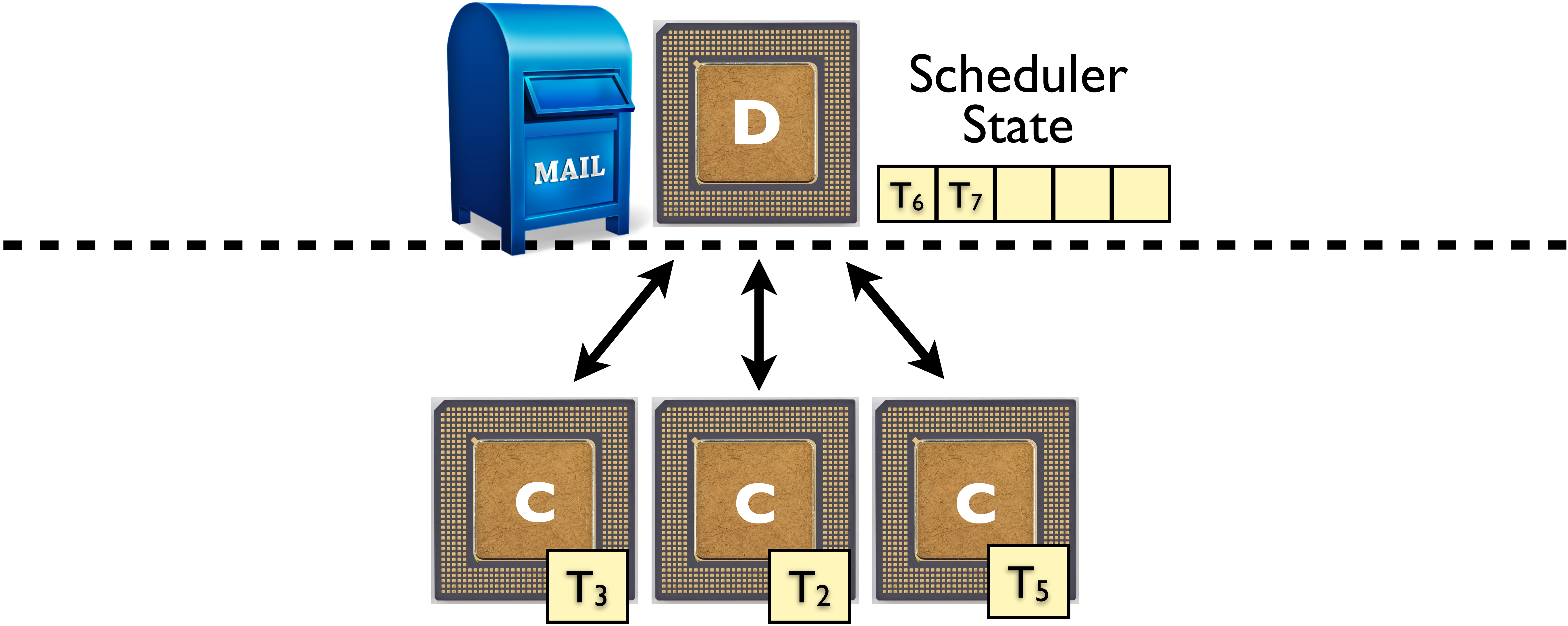
Implementing Messages Efficiently

- Message passing via per-cpu-socket mailboxes
- Shared-memory buffer with wait-free writes

Source code at
www.litmus-rt.org



G-EDF-MP



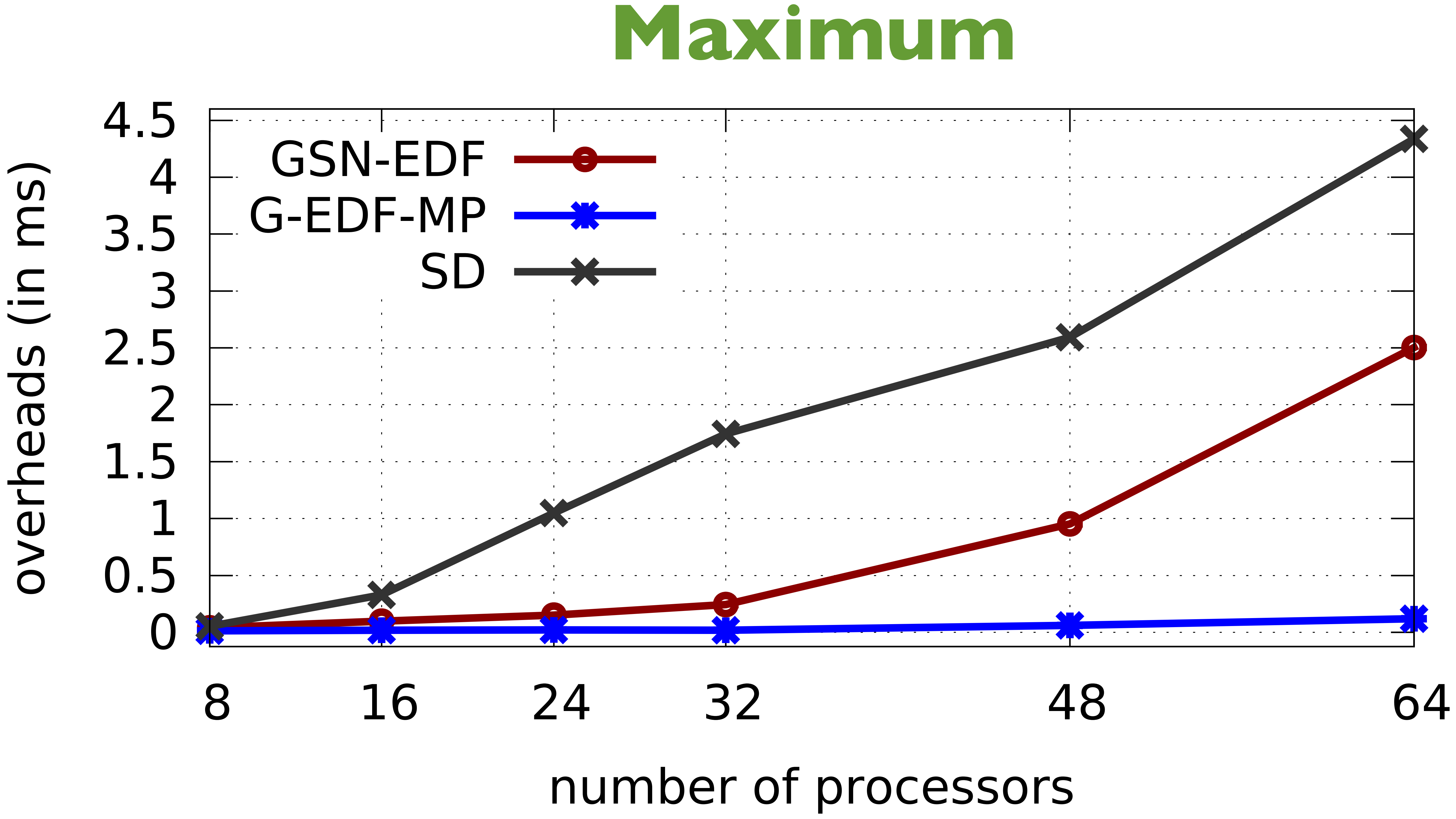
Global-EDF via **M**essage **P**assing

Centralized Scheduling with Message passing

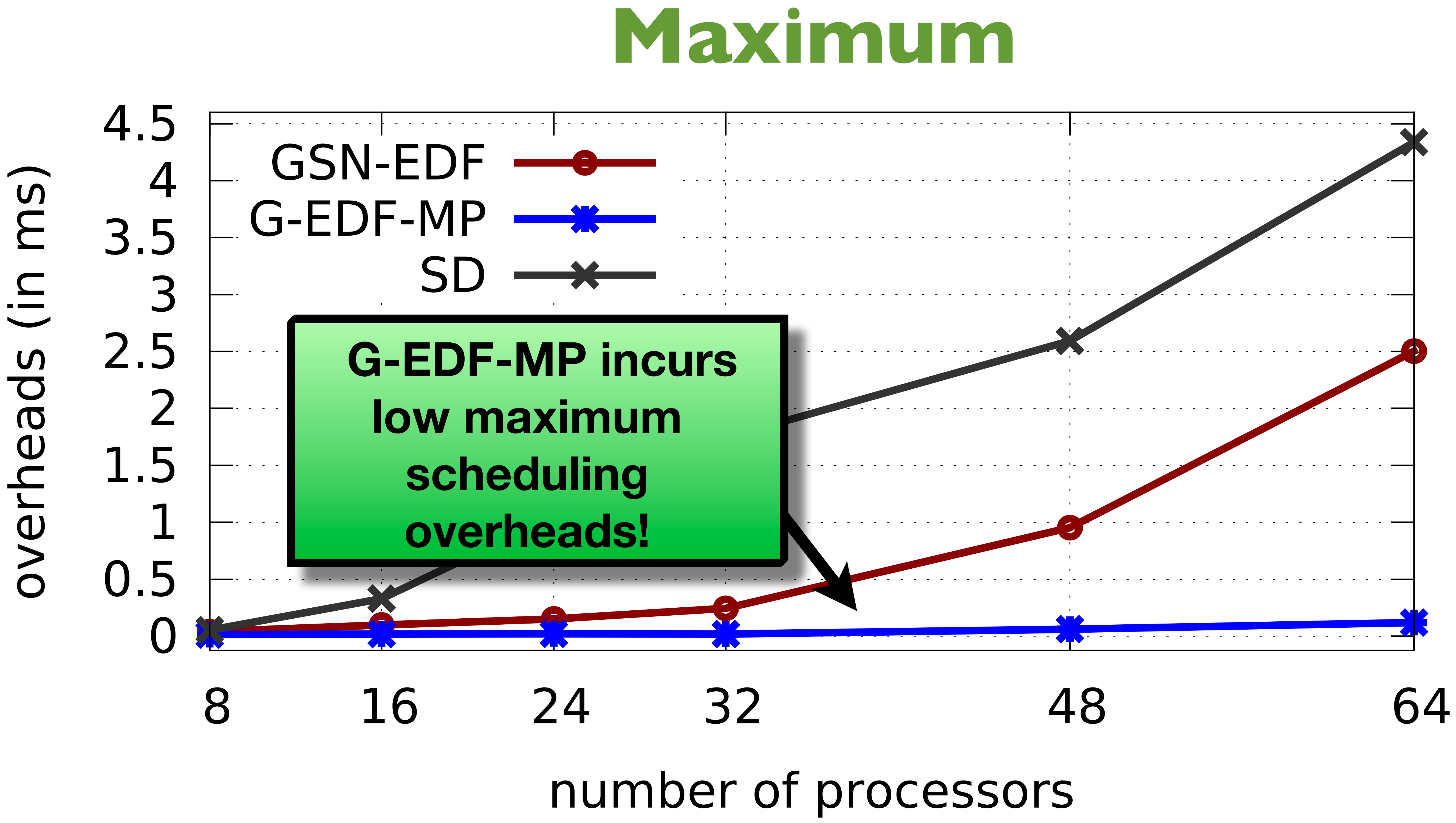
This Talk

1) Why global scheduling?
2) Current implementations
3) Root causes of overhead
4) How to scale global scheduling?
5) **Evaluation**

Low Scheduling Overheads



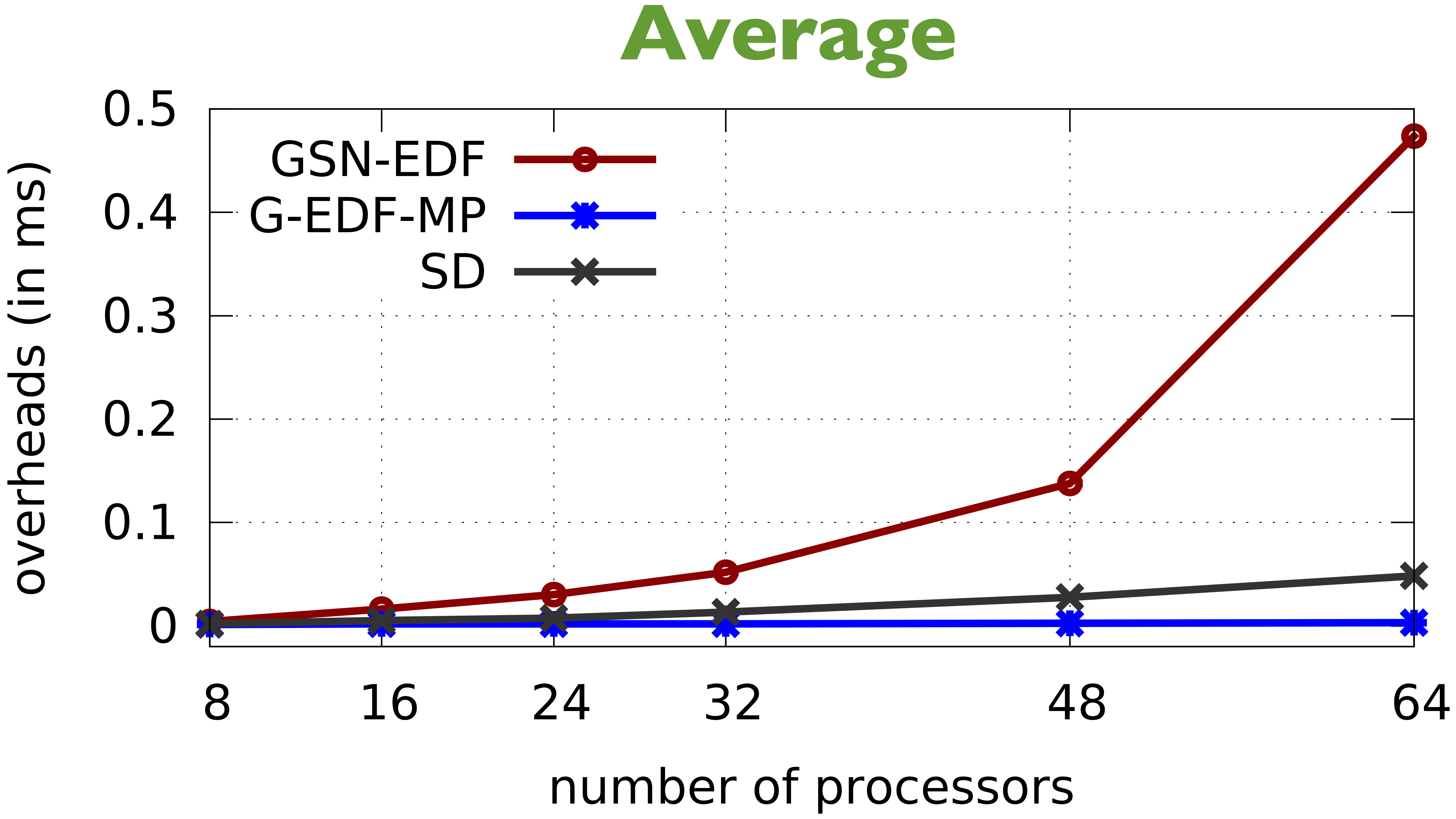
Low Scheduling Overheads



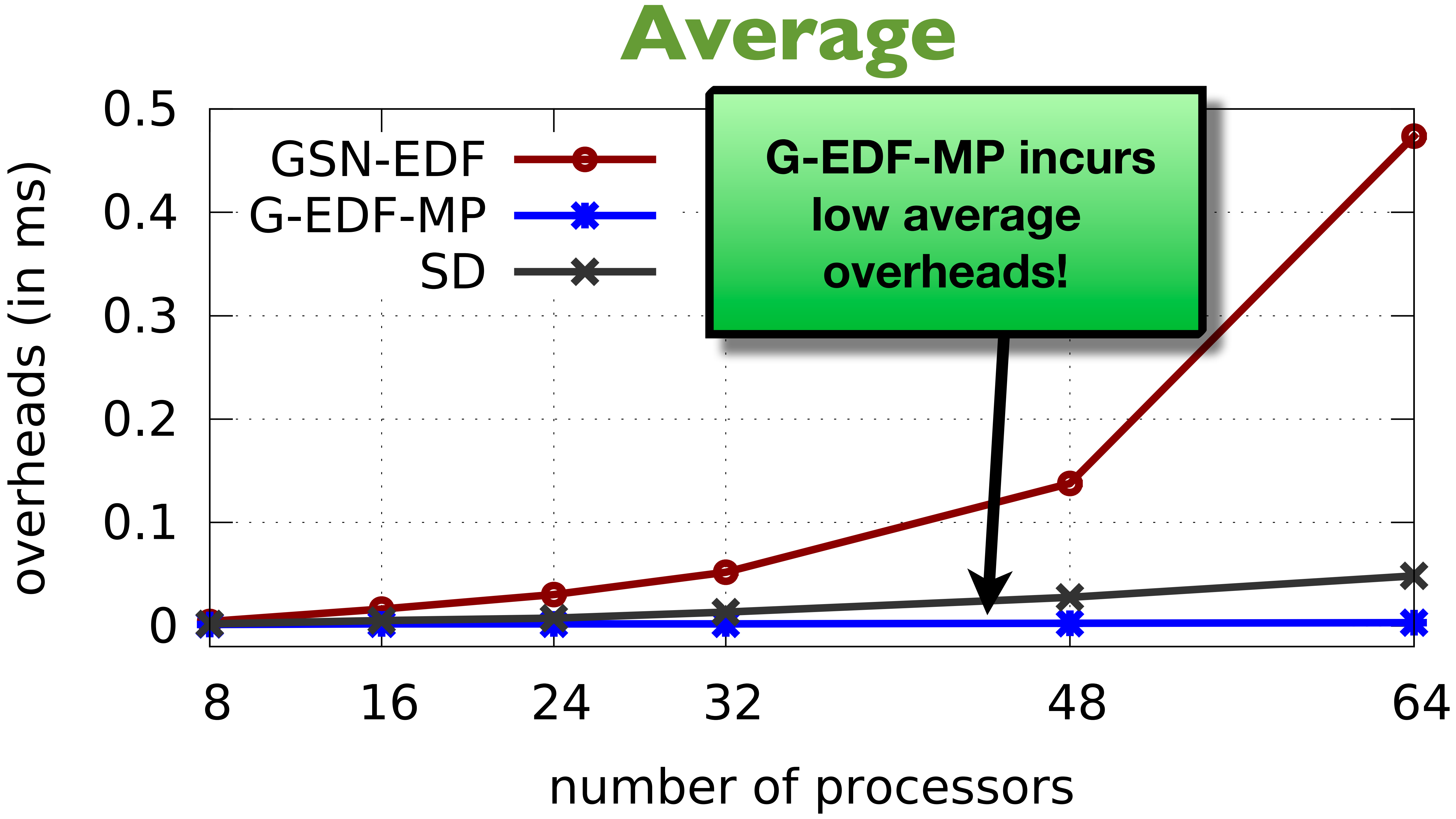
G-EDF-MP incurs low maximum scheduling overheads!

Low Scheduling Overheads

Low Scheduling Overheads

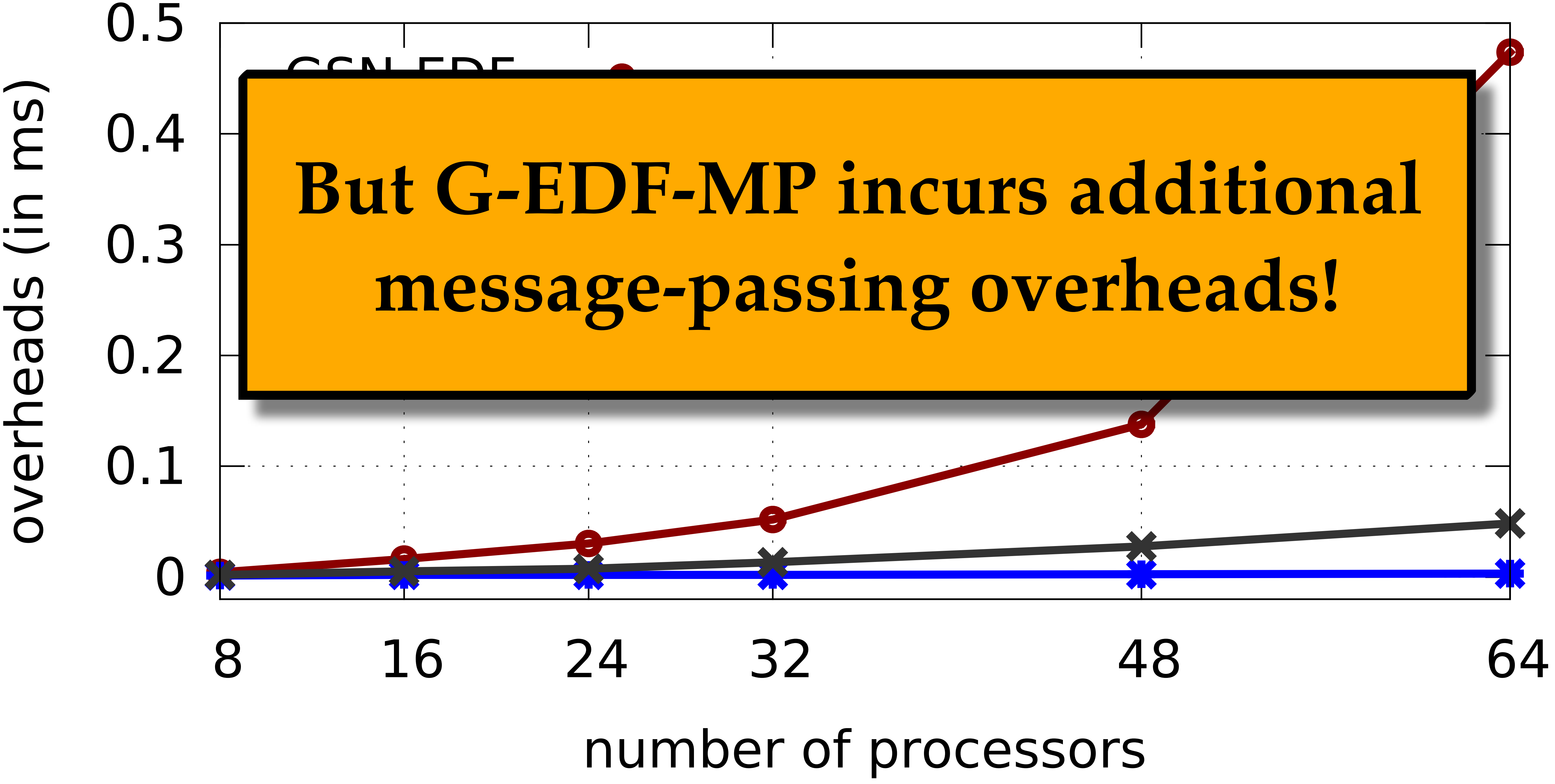


Low Scheduling Overheads



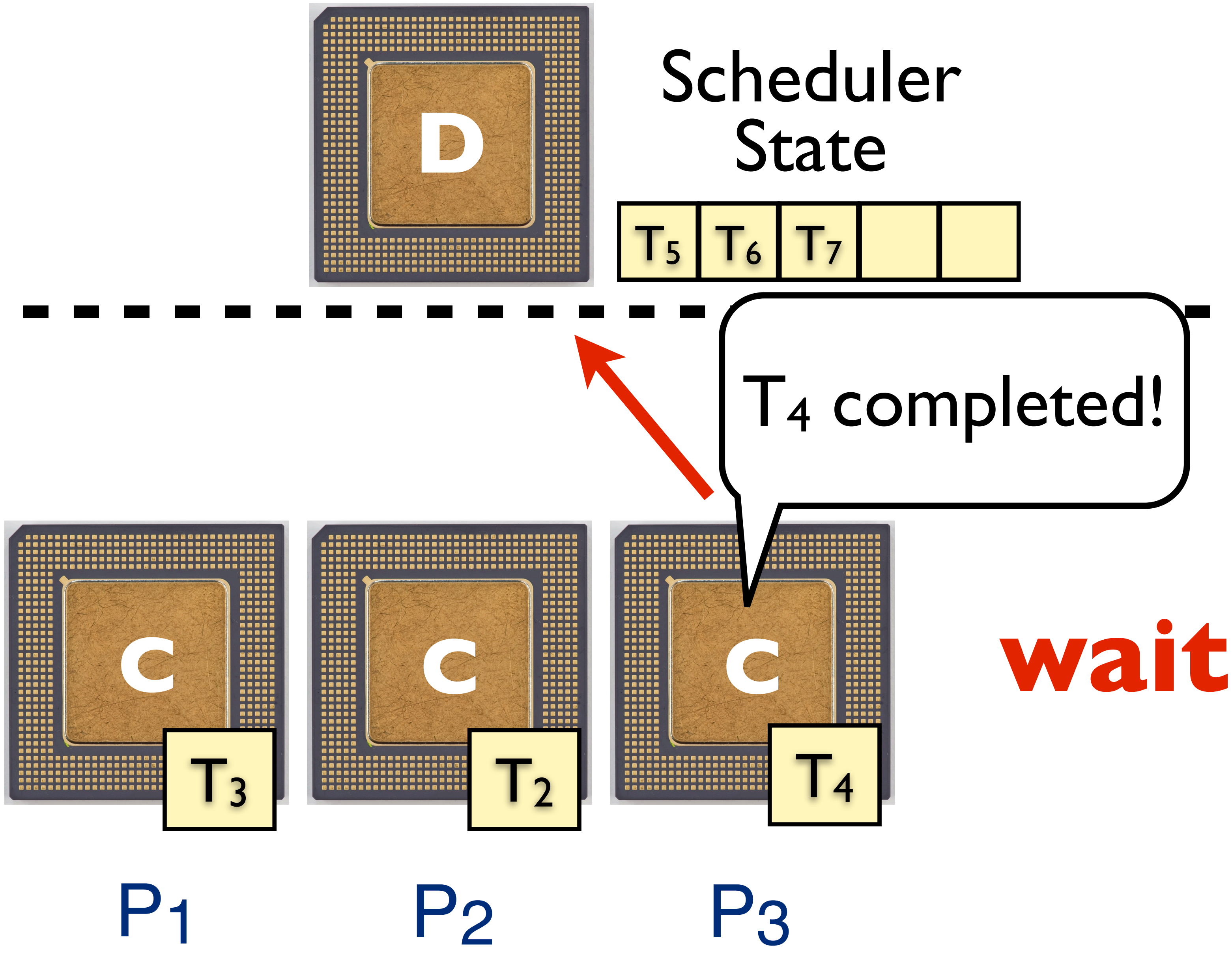
G-EDF-MP incurs low average overheads!

Low Scheduling Overheads


Average

But G-EDF-MP incurs additional message-passing overheads!

Two Sources of Overhead



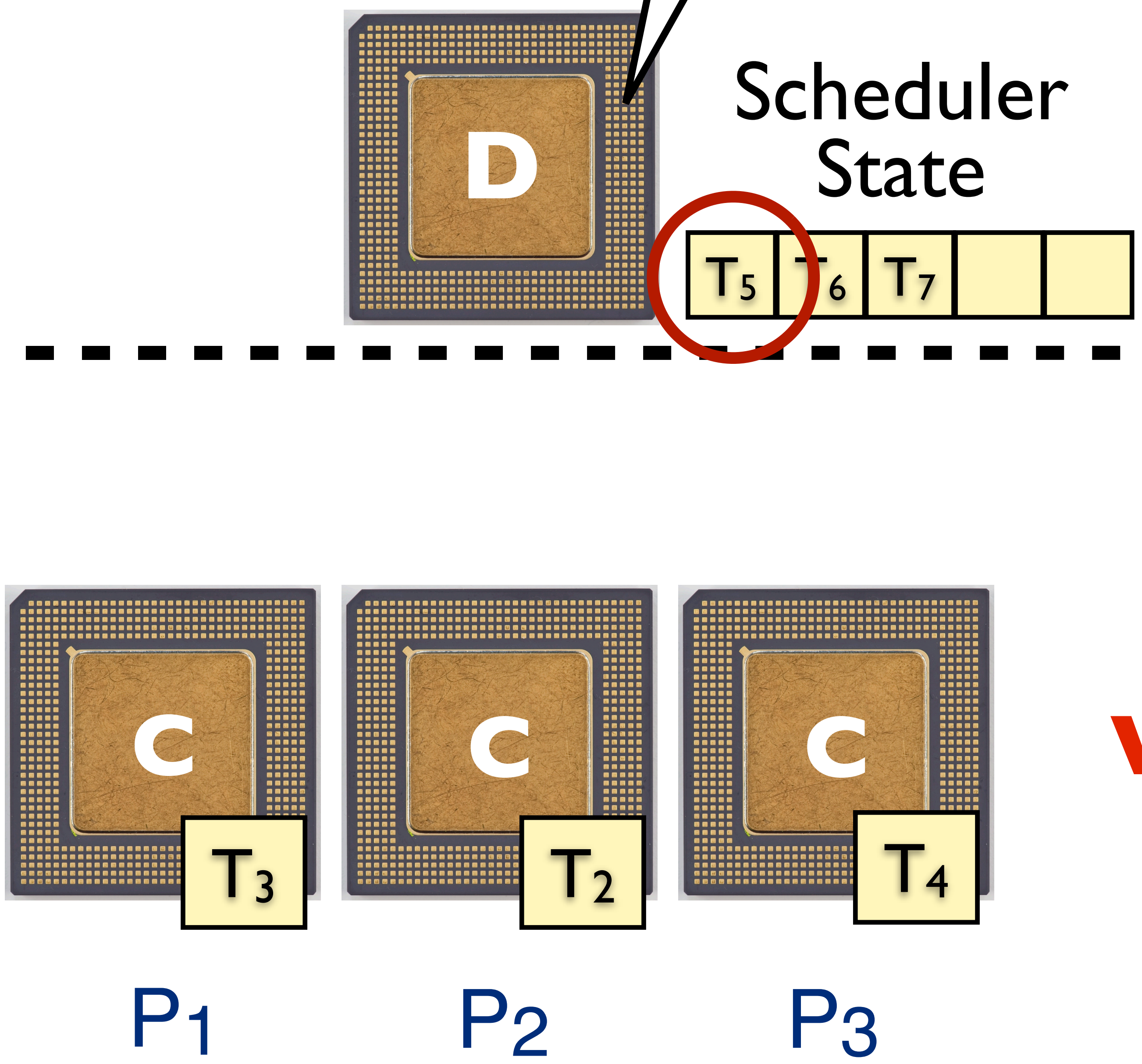
Message Latency

waiting...

Two Sources c d

Computing scheduling decision

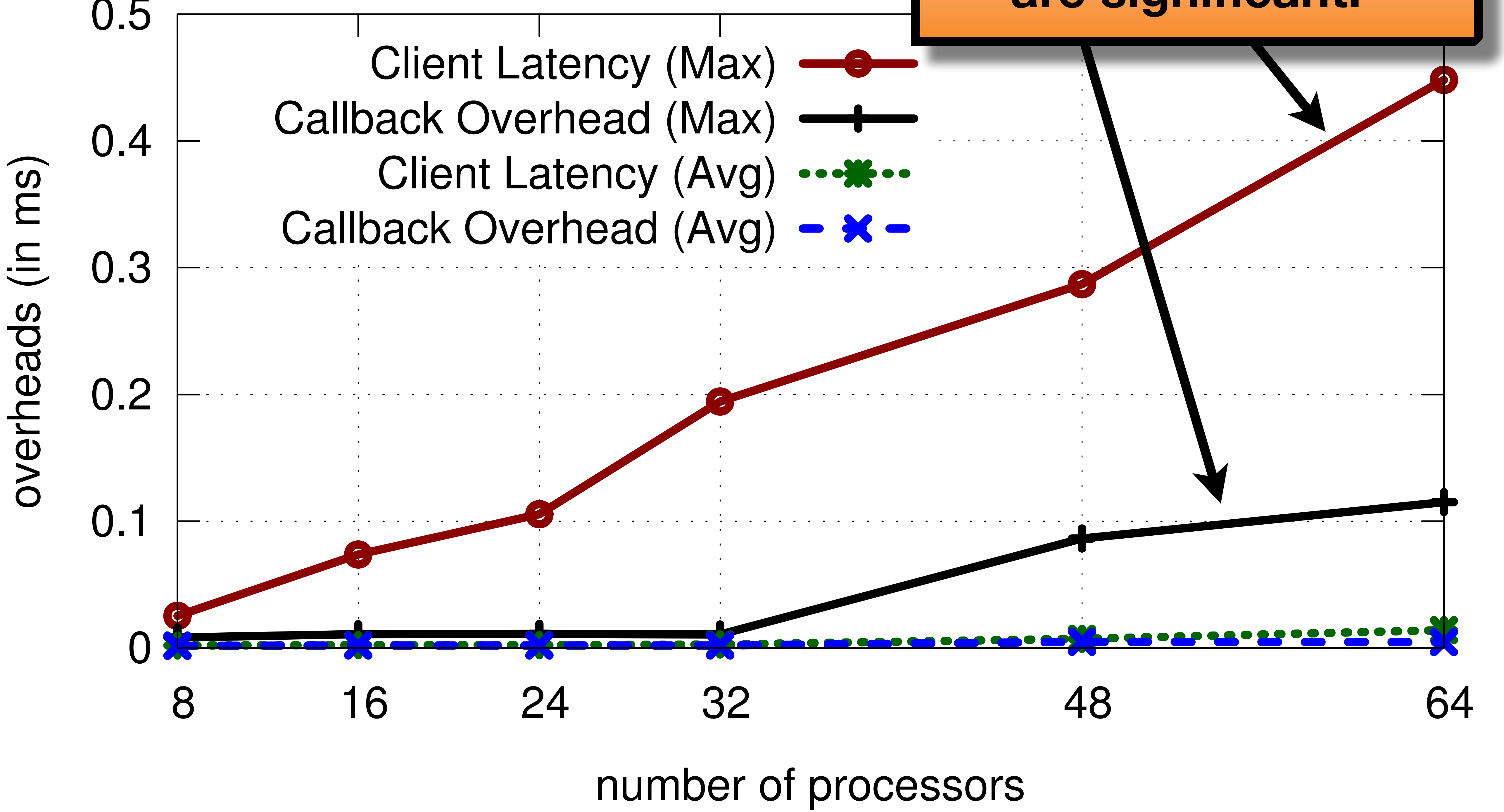
Message Callback Overhead



Message Latency

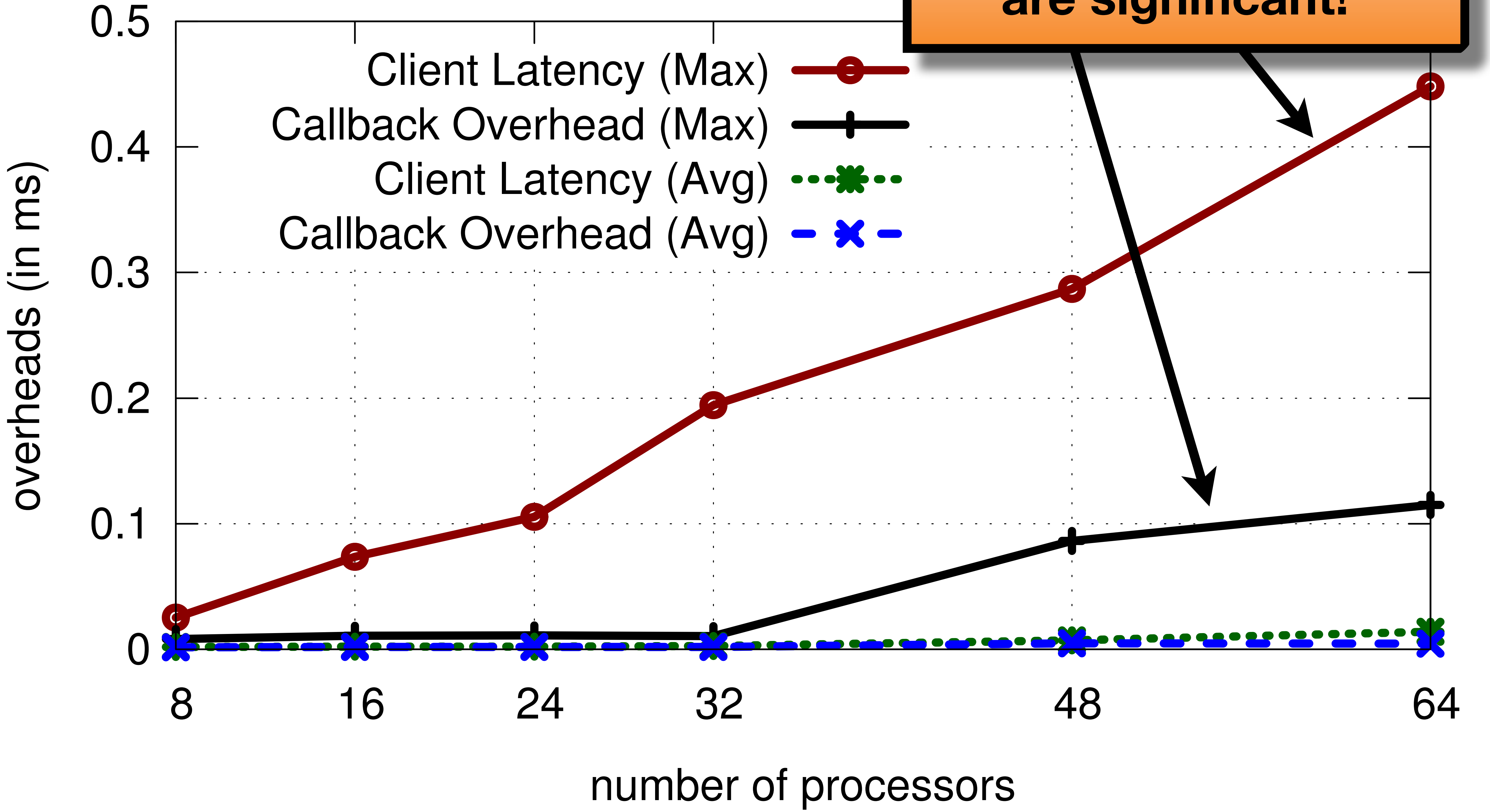
waiting...

Message-Passing



Message passing overheads are significant!

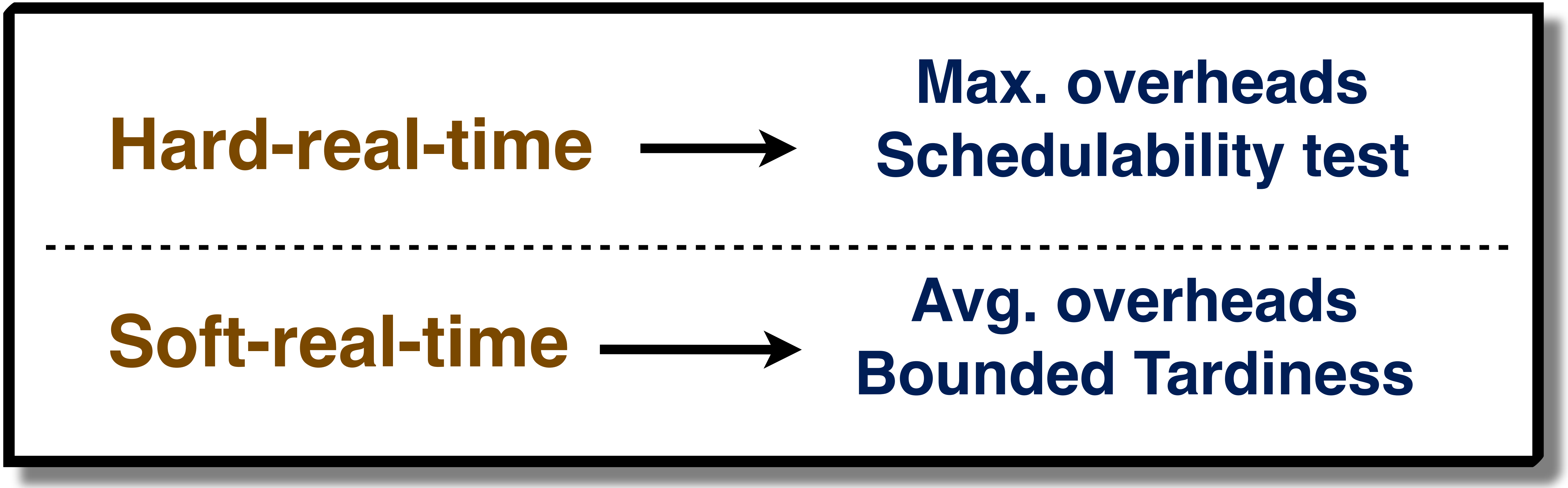
Message-Passing



Message passing overheads are significant!

What's the overall impact on schedulability?

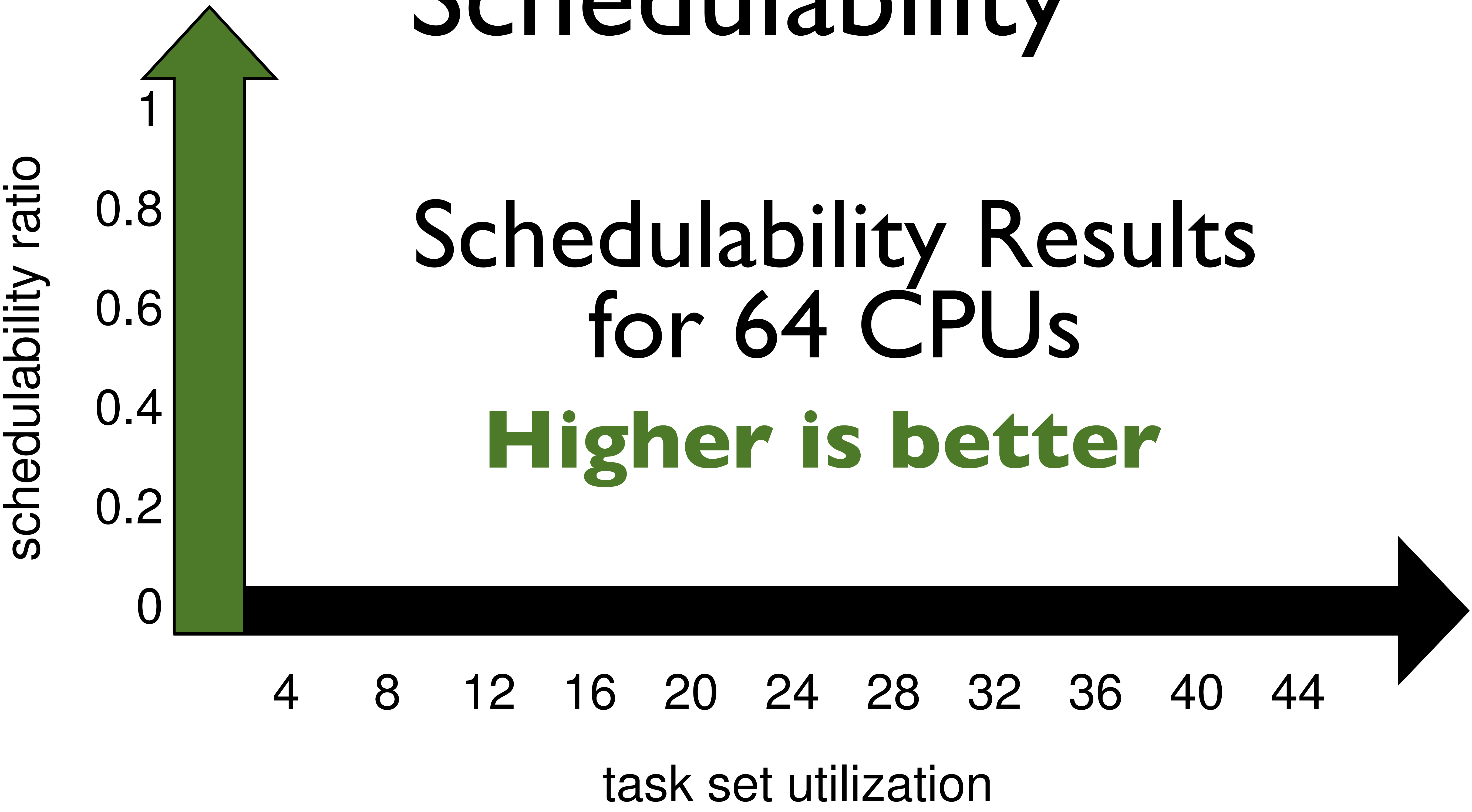
Overhead-Aware Analysis



Hard-real-time → Max. overheads
Schedulability test

Soft-real-time → Avg. overheads
Bounded Tardiness

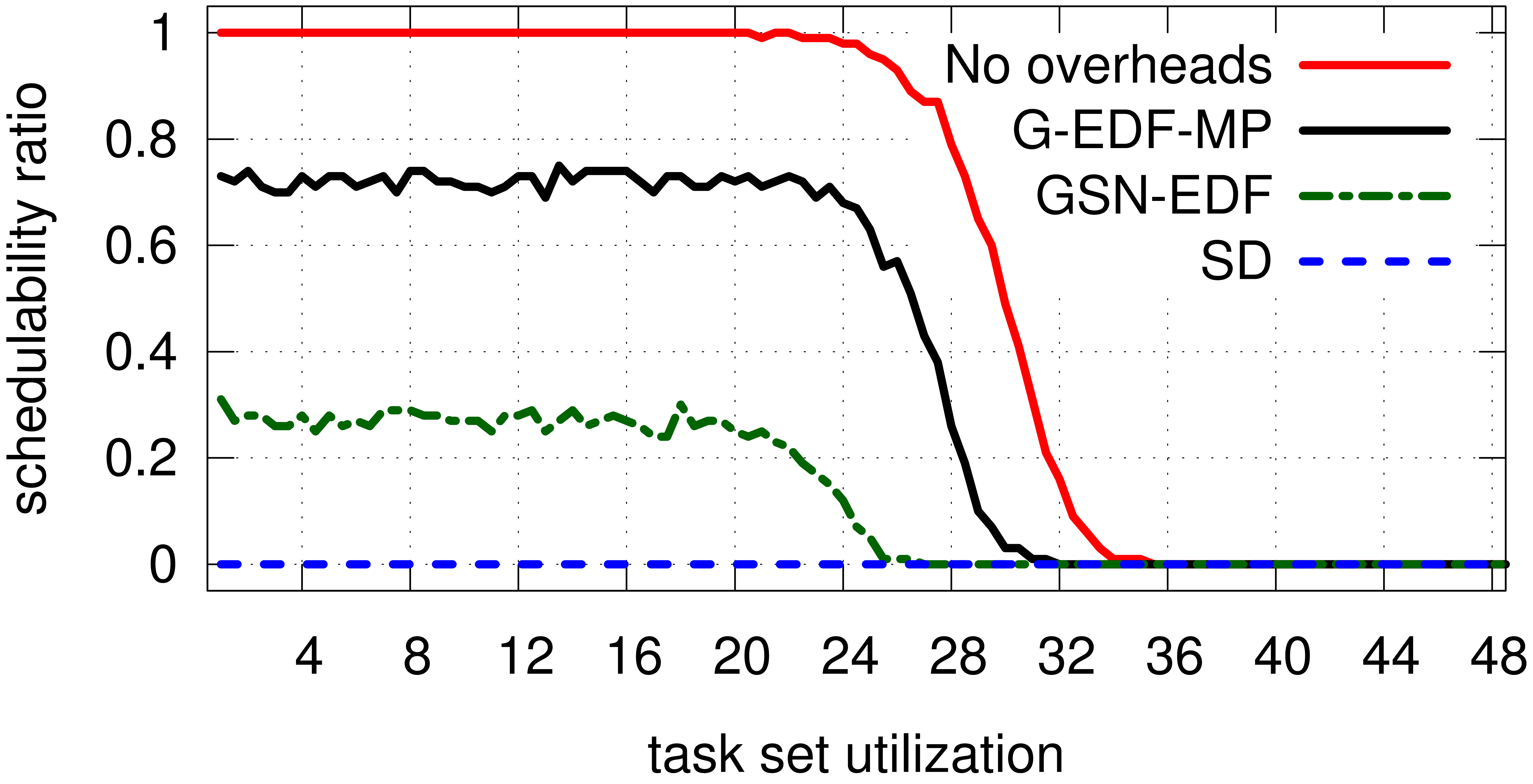
Overhead-Aware Schedulability



Schedulability Results for 64 CPUs

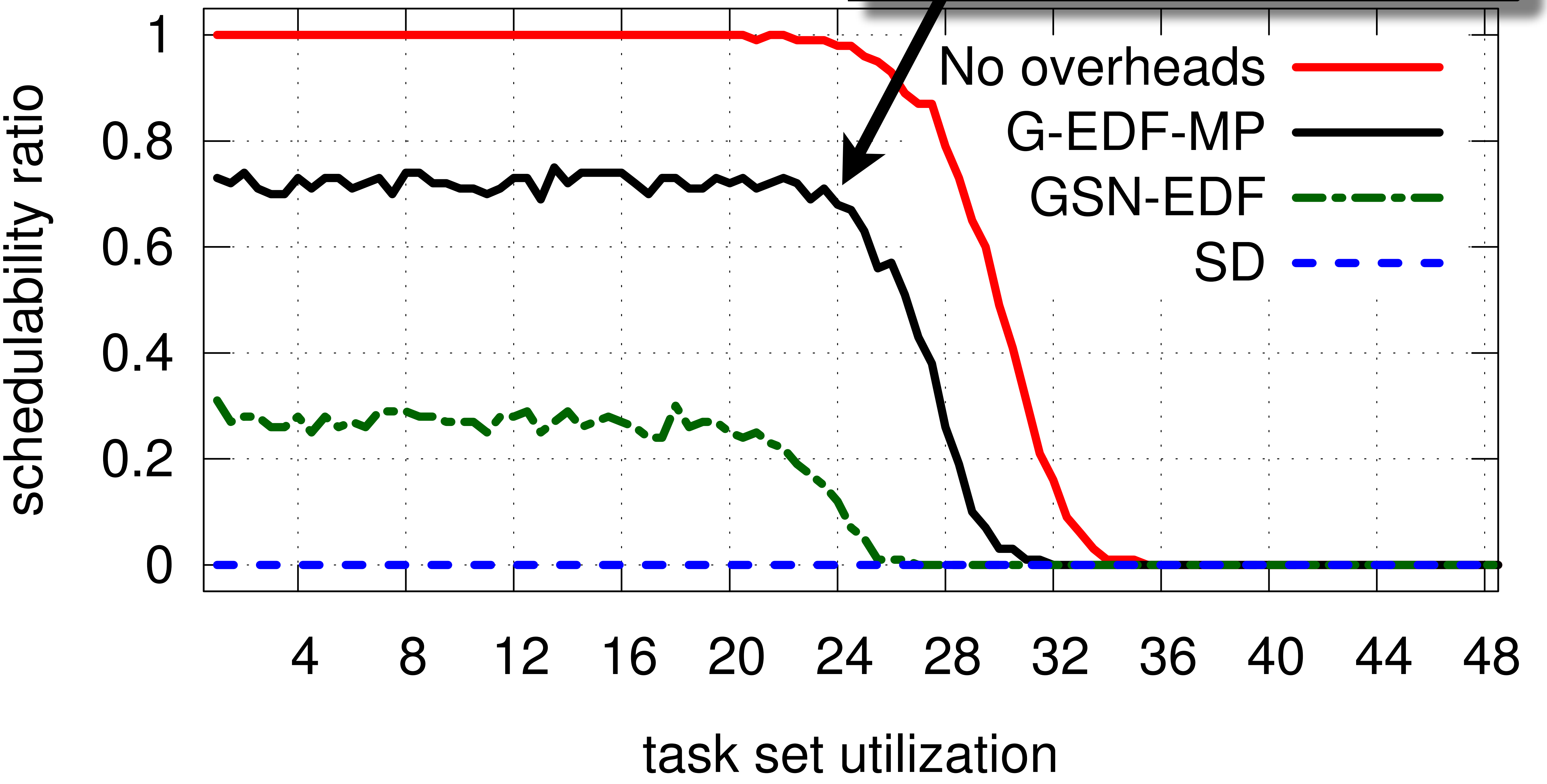
Higher is better

Hard-Real-Time Schedulability



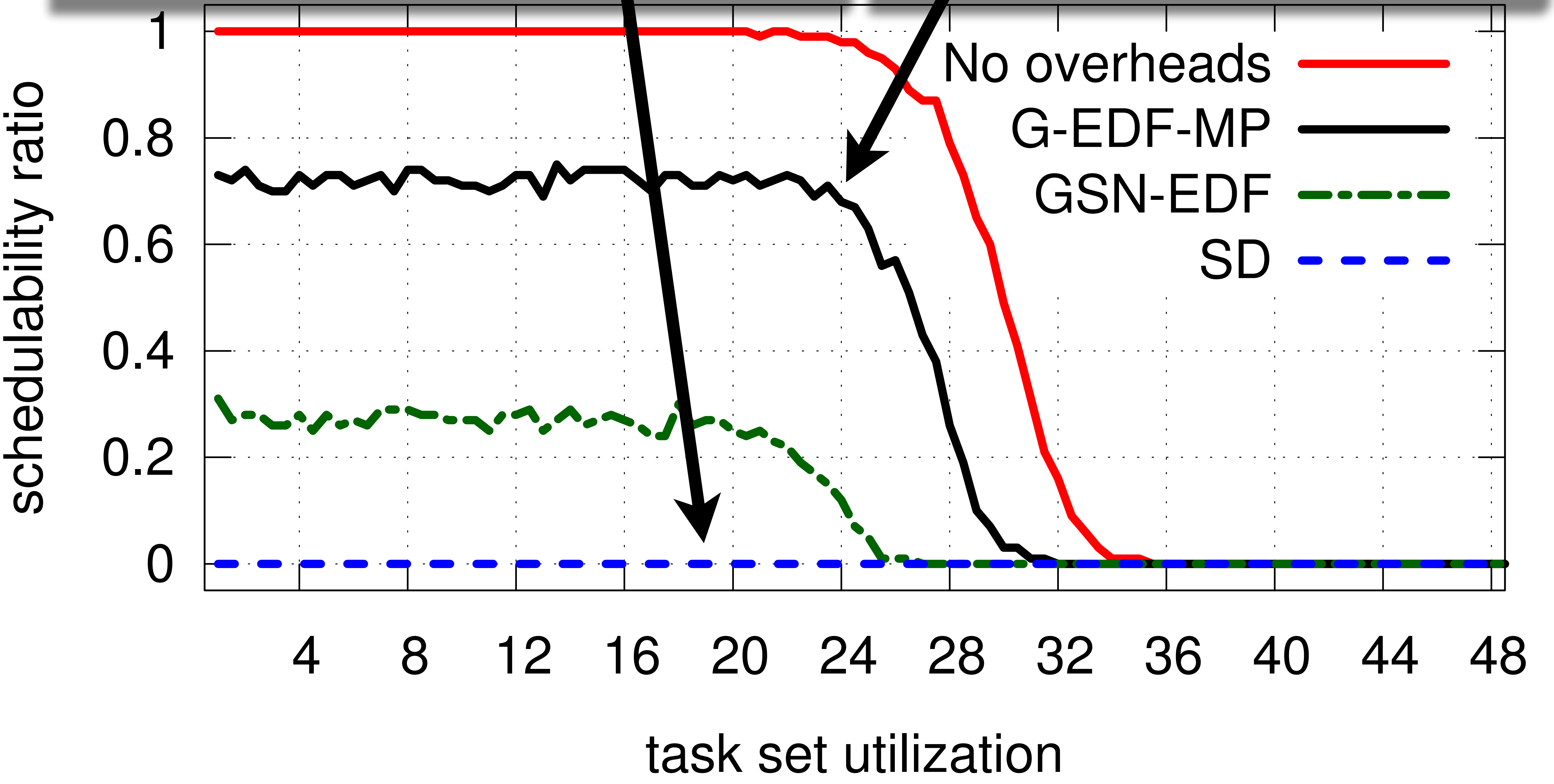
Hard-Real-Time

Higher schedulability, even with additional message-passing delays



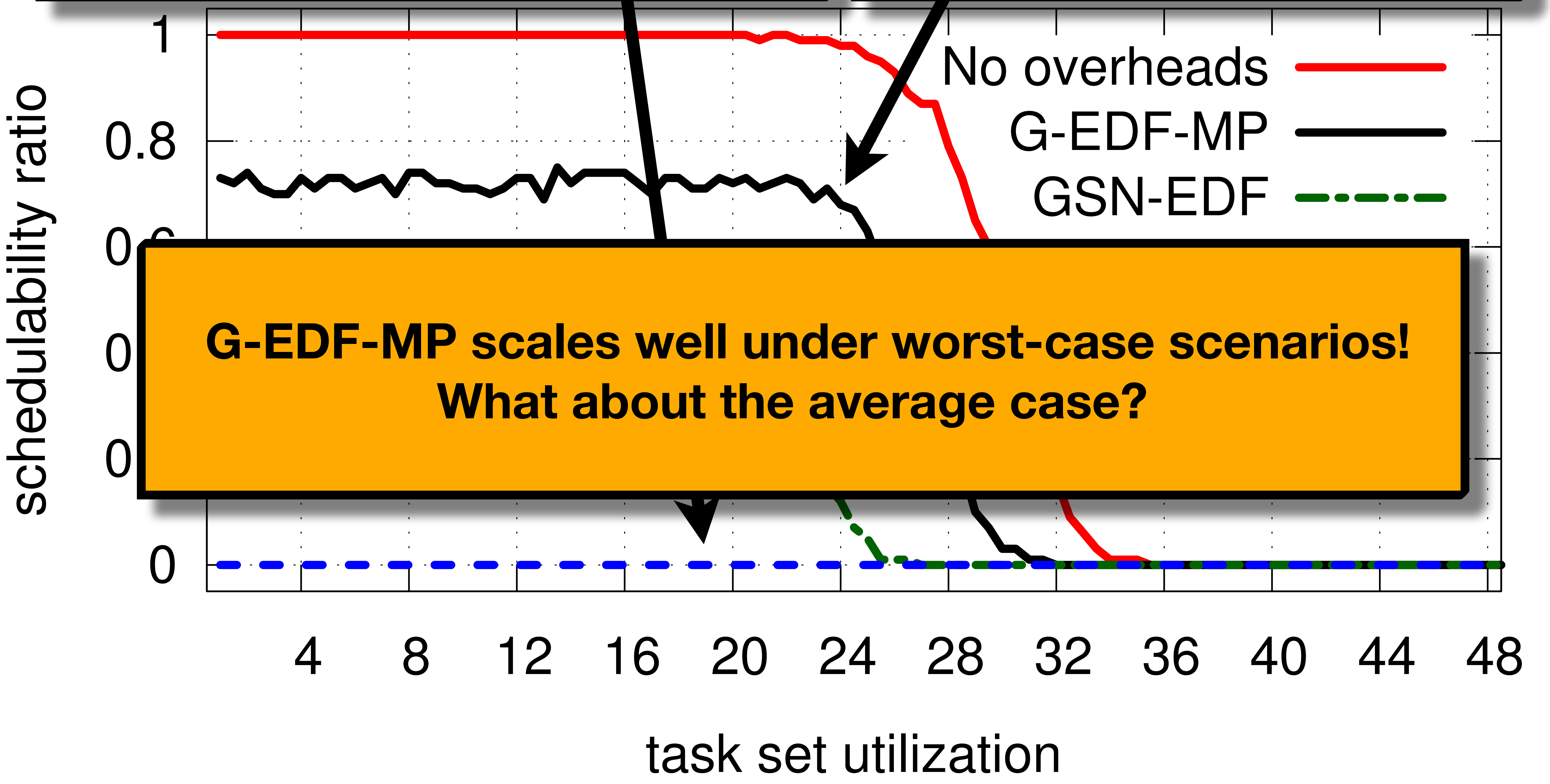
SCHED_DEADLINE does not implement dedicated interrupt handling, yielding a pessimistic analysis

Higher schedulability, even with additional message-passing delays



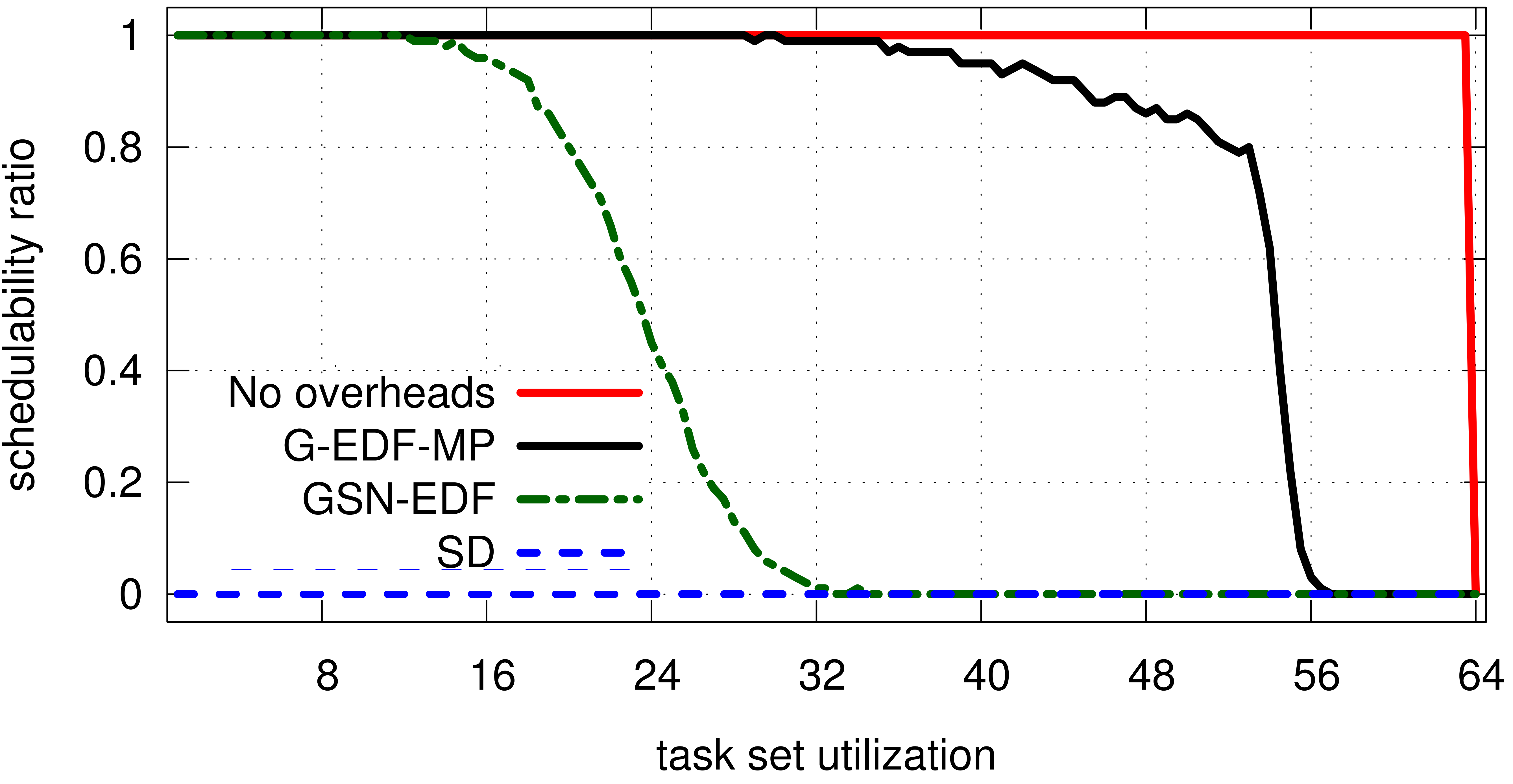
SCHED_DEADLINE does not implement dedicated interrupt handling, yielding a pessimistic analysis

Higher schedulability, even with additional message-passing delays



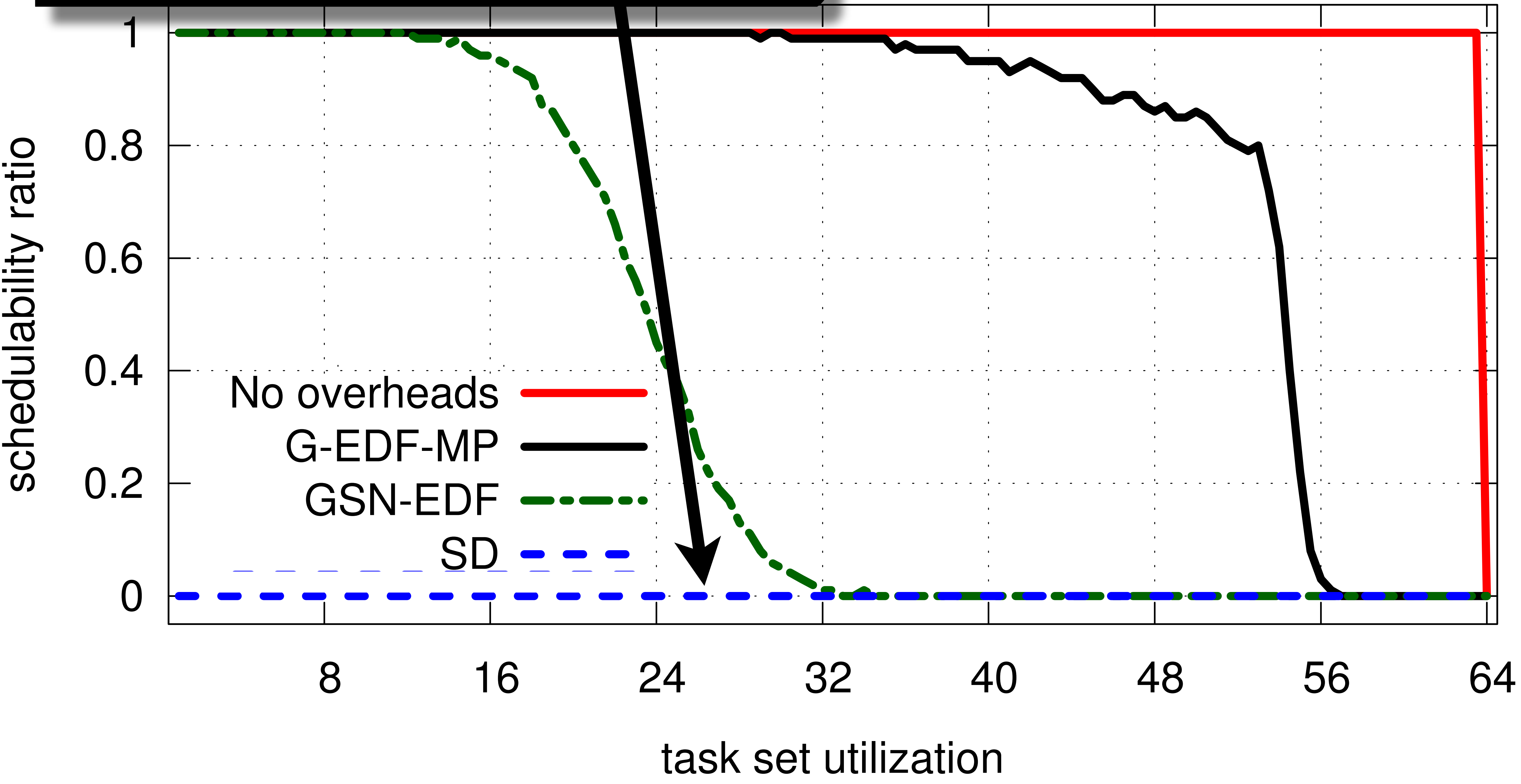
G-EDF-MP scales well under worst-case scenarios!
What about the average case?

Soft-Real-Time Schedulability



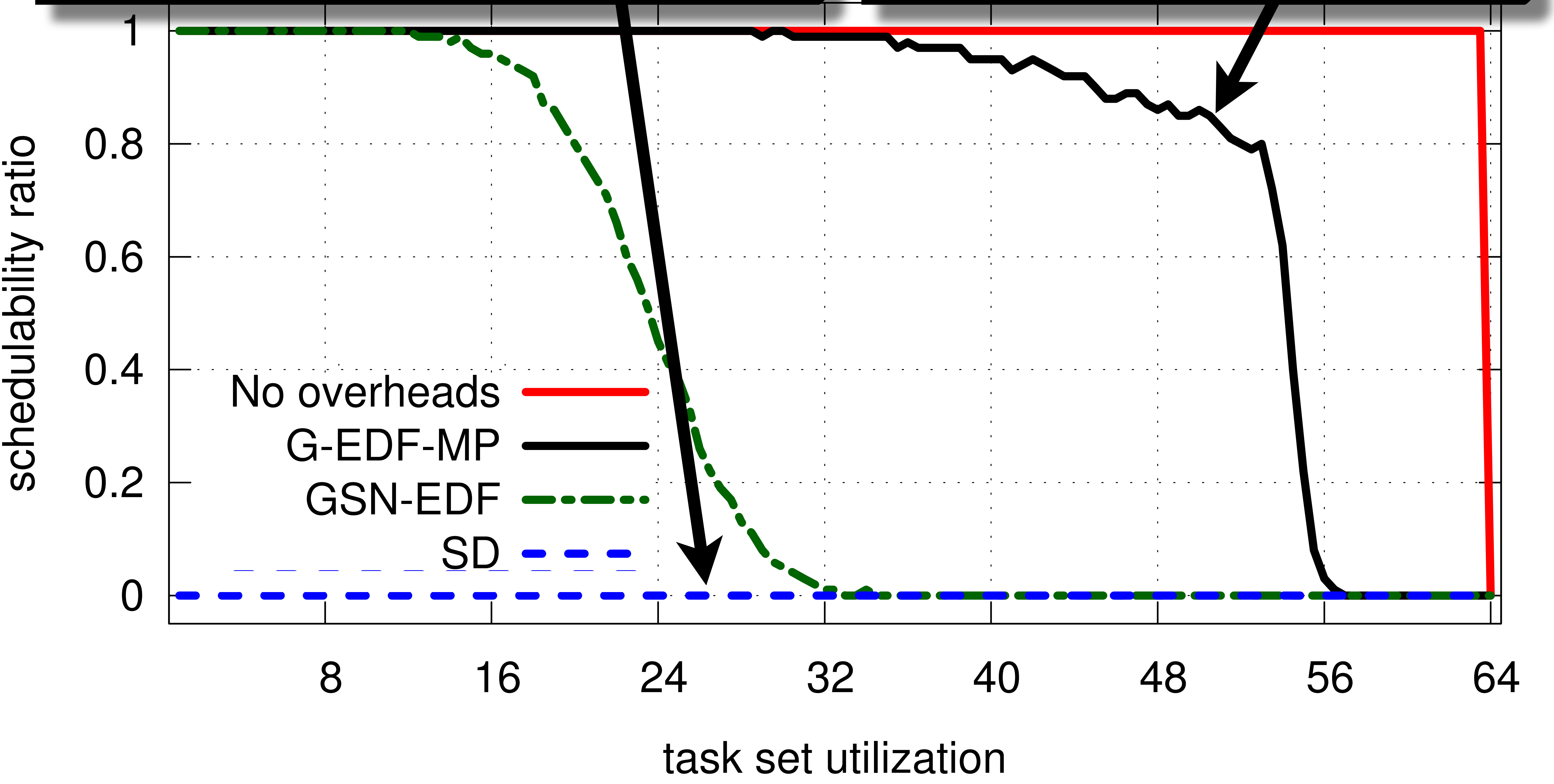
Schedulability

SCHED_DEADLINE works well in the average case, but cannot be shown to do so analytically



SCHED_DEADLINE works well in the average case, but cannot be shown to do so analytically

G-EDF-MP also performs well in the average case



Global-EDF with Low Overheads

Pair-wise coordination
+
Message passing

→

Scalable G-EDF implementation
up to 64 CPUs

Limitations

Dedicated scheduling processor is still a **scalability bottleneck** *at* ***extreme core counts*****.**

➜ G-EDF-MP scales ***much further*** than prior approaches.

G-EDF-MP is ***inappropriate*** for workloads that do not tolerate **excessive migration overheads.**

➜ Migrations are inherent to global scheduling policies, ***irrespective of implementation.***

This approach can be applied to global scheduling in general, not just G-EDF.

Conclusion

Conclusion

Fine-grained locking is not enough. Scalability of worst-case overheads requires avoiding **peak contention** and **cache-line bouncing.**

Conclusion

Fine-grained locking is not enough. Scalability of worst-case overheads requires avoiding **peak contention** and **cache-line bouncing.**

To reduce overheads, we used a **centralized scheduler** and **message passing.**

Conclusion

Fine-grained locking is not enough. Scalability of worst-case overheads requires avoiding **peak contention** and **cache-line bouncing.**

To reduce overheads, we used a **centralized scheduler** and **message passing.**

G-EDF-MP's design can be applied to other global schedulers and **extends the range of processor counts** that can be practically supported.

Thanks!

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

www.litmus-rt.org

New release 2014.1 is now available!