ORACLE®

Callisto-RTS: Fine-Grain Parallel Loops

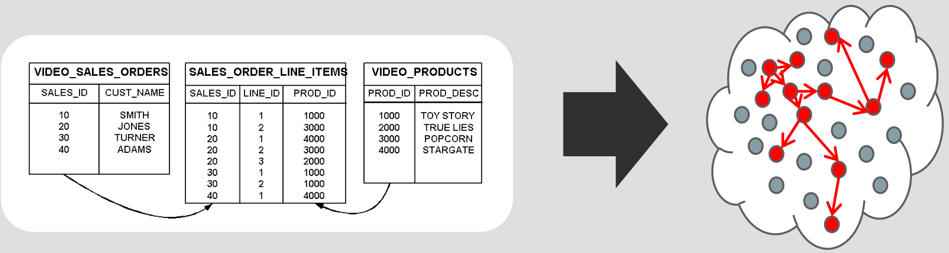
Tim Harris, Oracle Labs
Stefan Kaestle, ETH Zurich
Daniel Goodman, Oracle Labs

15 July 2016

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

In-memory graph analytics

Using a graph representation for your data



VIDEO_SALES_ORDERS

SALES_ID	CUST_NAME
10	SMITH
20	JONES
30	TURNER
40	ADAMS

SALES_ORDER_LINE_ITEMS

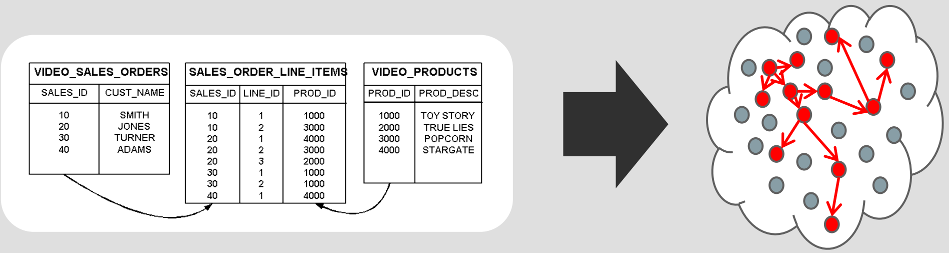
SALES_ID	LINE_ID	PROD_ID
10	1	1000
10	2	3000
20	1	4000
20	2	3000
20	3	2000
30	1	1000
30	2	1000
40	1	4000

VIDEO_PRODUCTS

PROD_ID	PROD_DESC
1000	TOY STORY
2000	TRUE LIES
3000	POPCORN
4000	STARGATE

In-memory graph analytics

Using a graph representation for your data

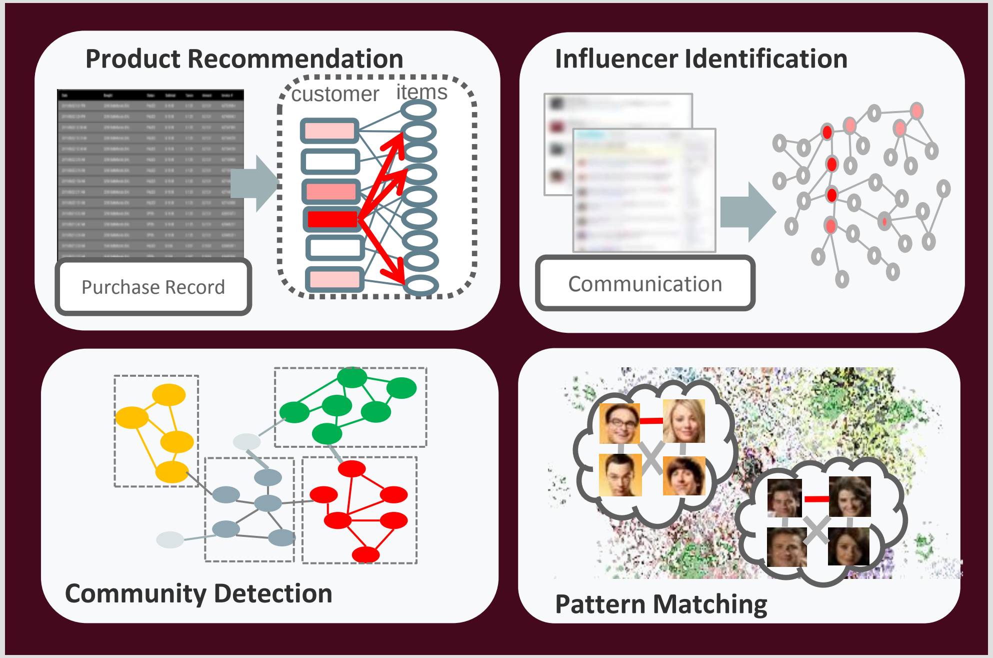


VIDEO_SALES_ORDERS	
SALES_ID	CUST_NAME
10	SMITH
20	JONES
30	TURNER
40	ADAMS

SALES_ORDER_LINE_ITEMS		
SALES_ID	LINE_ID	PROD_ID
10	1	1000
10	2	3000
20	1	4000
20	2	3000
20	3	2000
30	1	1000
30	2	1000
40	1	4000

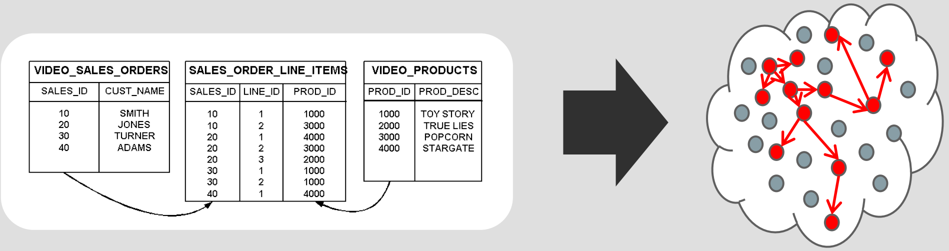
VIDEO_PRODUCTS	
PROD_ID	PROD_DESC
1000	TOY STORY
2000	TRUE LIES
3000	POPCORN
4000	STARGATE

enables many interesting new analyses



In-memory graph analytics

Using a graph representation for your data



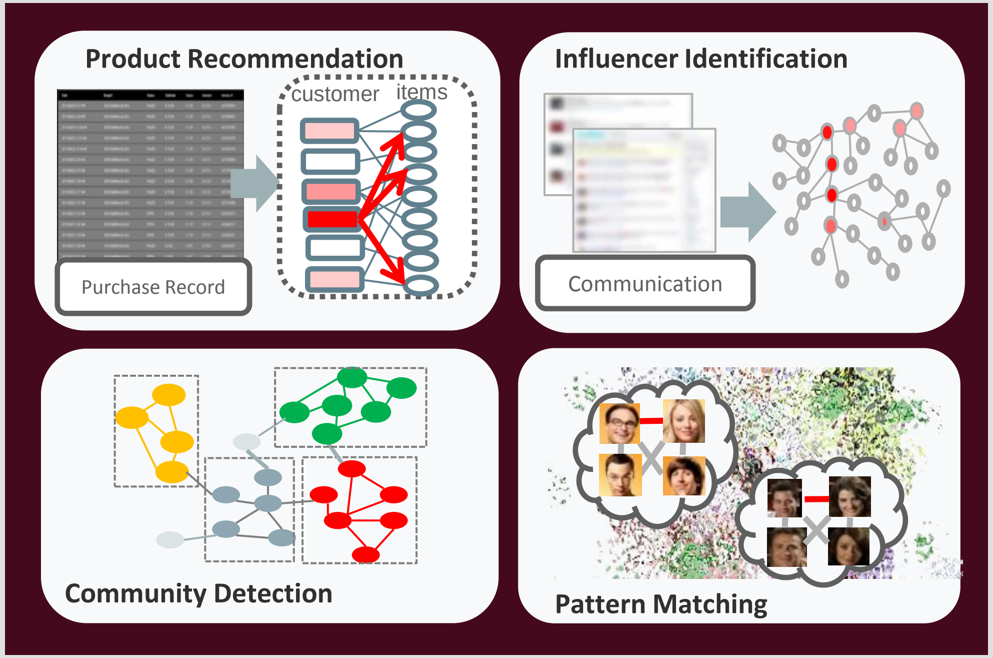
VIDEO_SALES_ORDERS	
SALES_ID	CUST_NAME
10	SMITH
20	JONES
30	TURNER
40	ADAMS

SALES_ORDER_LINE_ITEMS		
SALES_ID	LINE_ID	PROD_ID
10	1	1000
10	2	3000
20	1	4000
20	2	3000
20	3	2000
30	1	1000
30	2	1000
40	1	4000

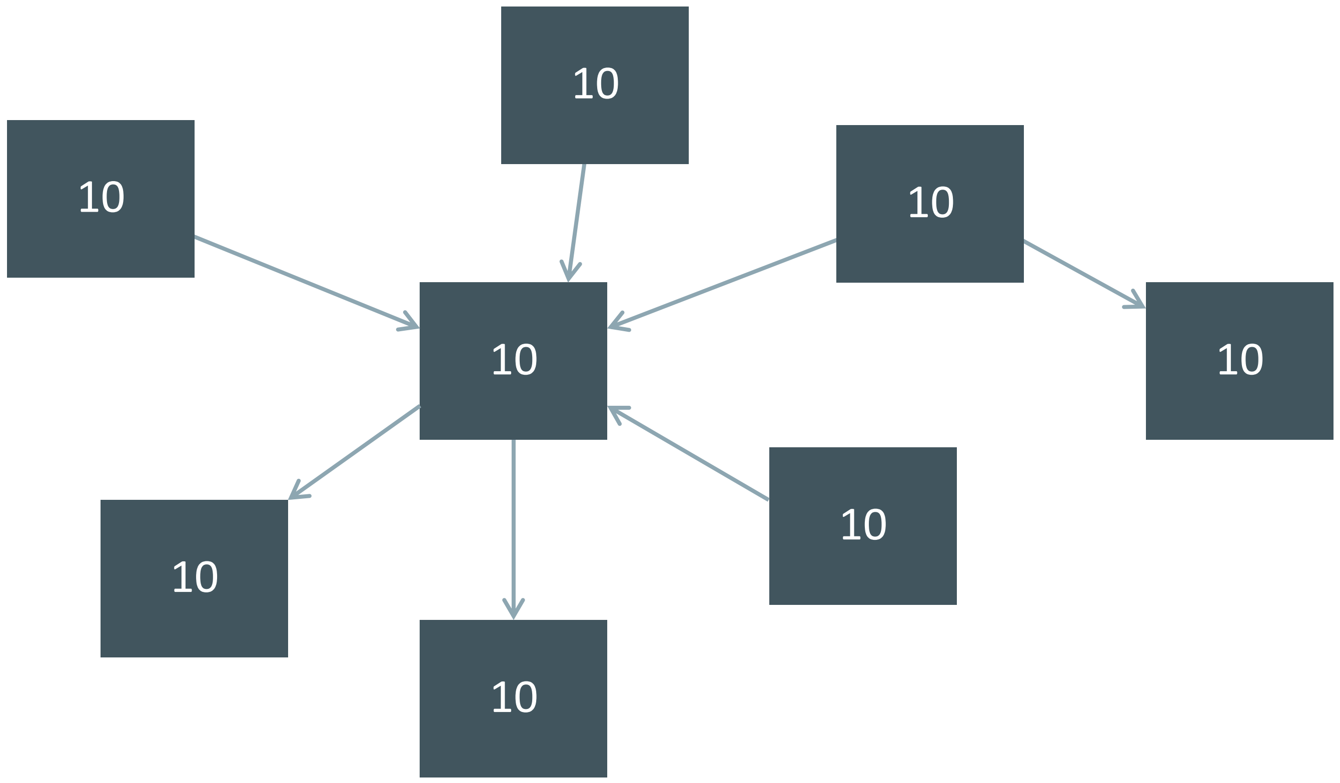
VIDEO_PRODUCTS	
PROD_ID	PROD_DESC
1000	TOY STORY
2000	TRUE LIES
3000	POPCORN
4000	STARGATE

and eliminates repeated join operations, which is much more efficient when you have a lot of relationships to traverse

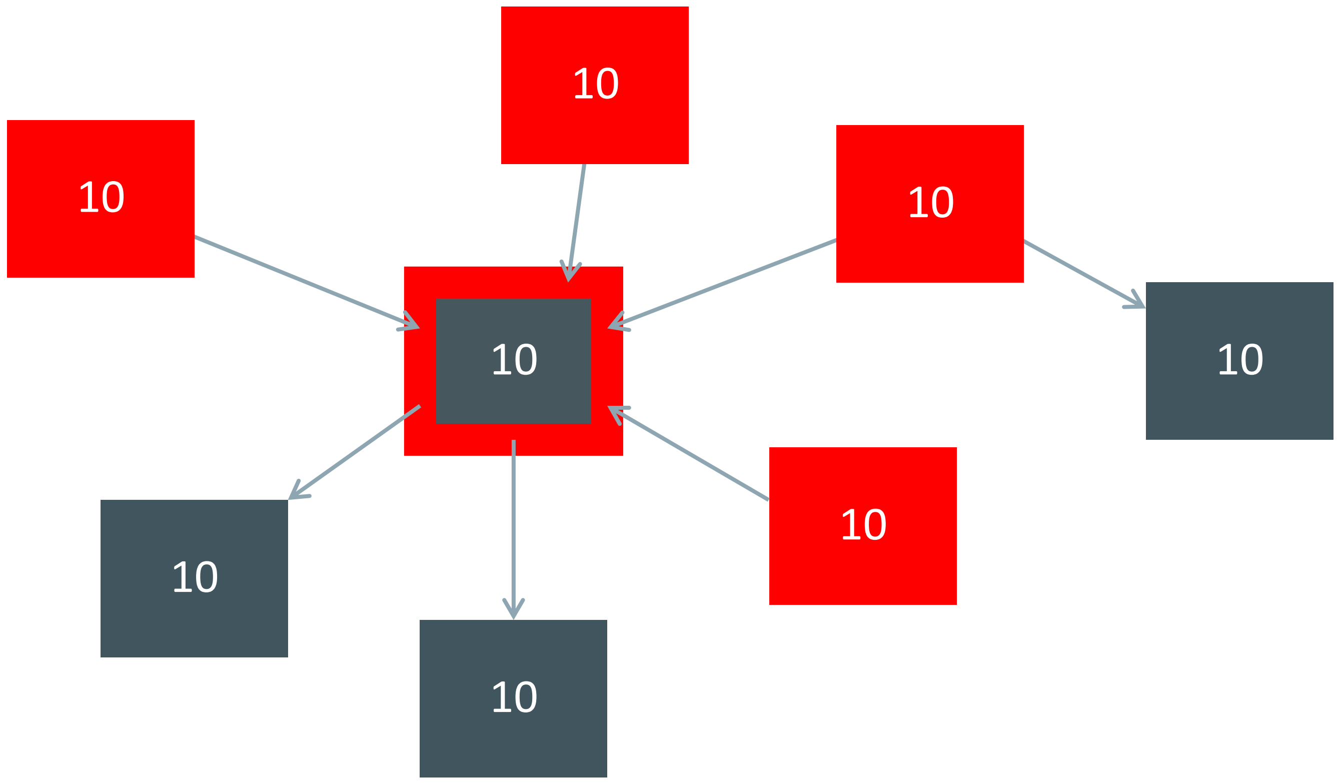
enables many interesting new analyses



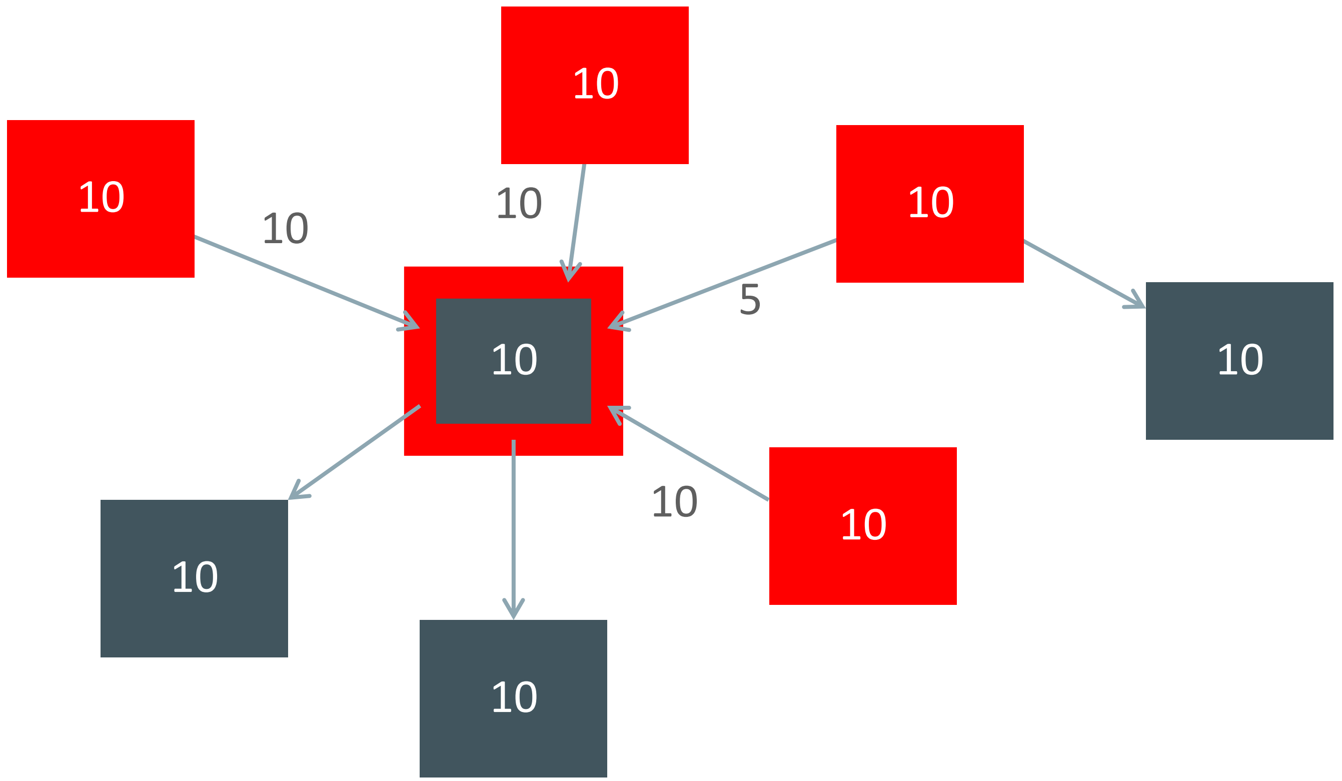
PageRank inner loop



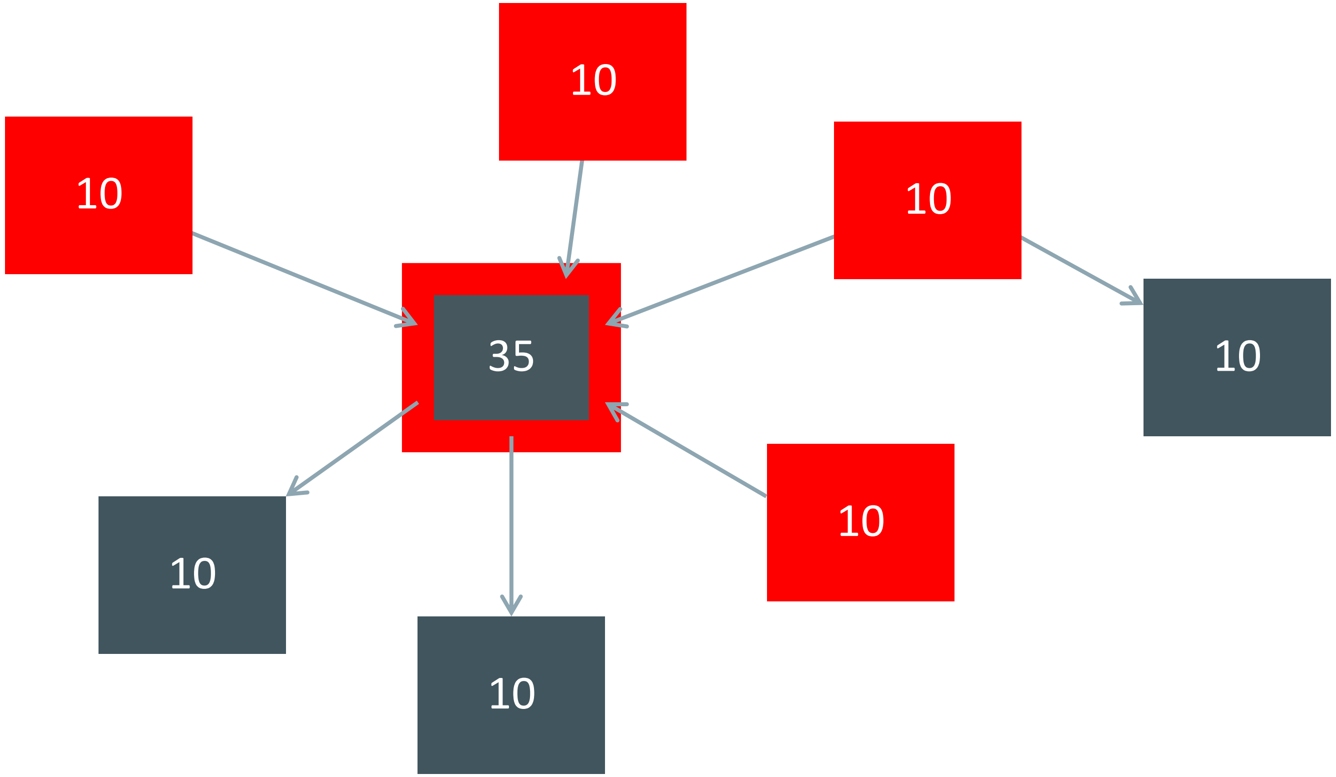
PageRank inner loop



PageRank inner loop



PageRank inner loop



ORACLE

Hardware options

Hardware options

My laptop

- Insufficient RAM
- Insufficient CPU capacity

Non-starter

Hardware options

My laptop

- Insufficient RAM
- Insufficient CPU capacity

Non-starter

Cluster

- Enough RAM
- Enough CPU capacity
- Distributed memory model

Irregular memory accesses make it hard to program

Hardware options

My laptop

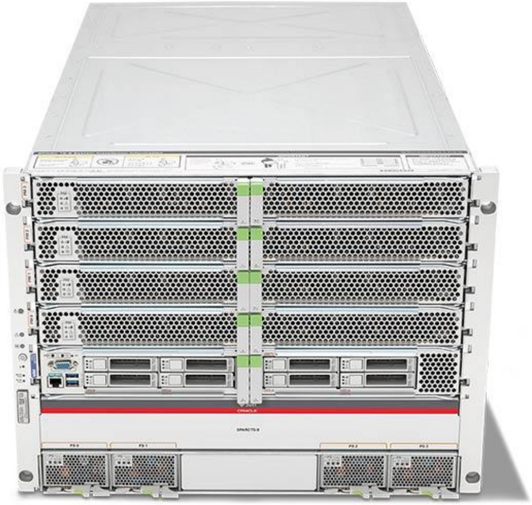
- Insufficient RAM
- Insufficient CPU capacity

Non-starter

Cluster

- Enough RAM
- Enough CPU capacity
- Distributed memory model

Irregular memory accesses
make it hard to program



Large shared memory machine

- Enough RAM
- Enough CPU capacity
- Shared memory model

In-memory graph analytics

Domain specific languages

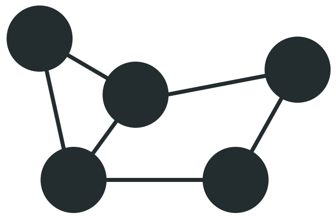
- Queries expressed in terms of graph concepts
- Tailor for different kinds of workload (e.g., sub-graph isomorphism)

Generated code

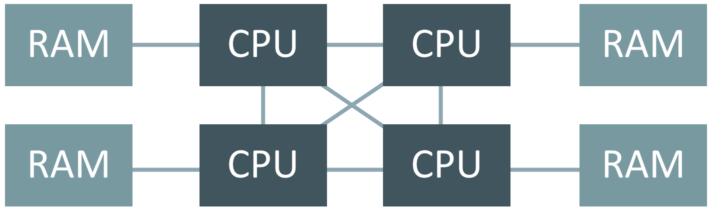
- Efficient in-memory data representations, e.g. compressed-sparse-row format
- Abundant parallelism

Runtime system

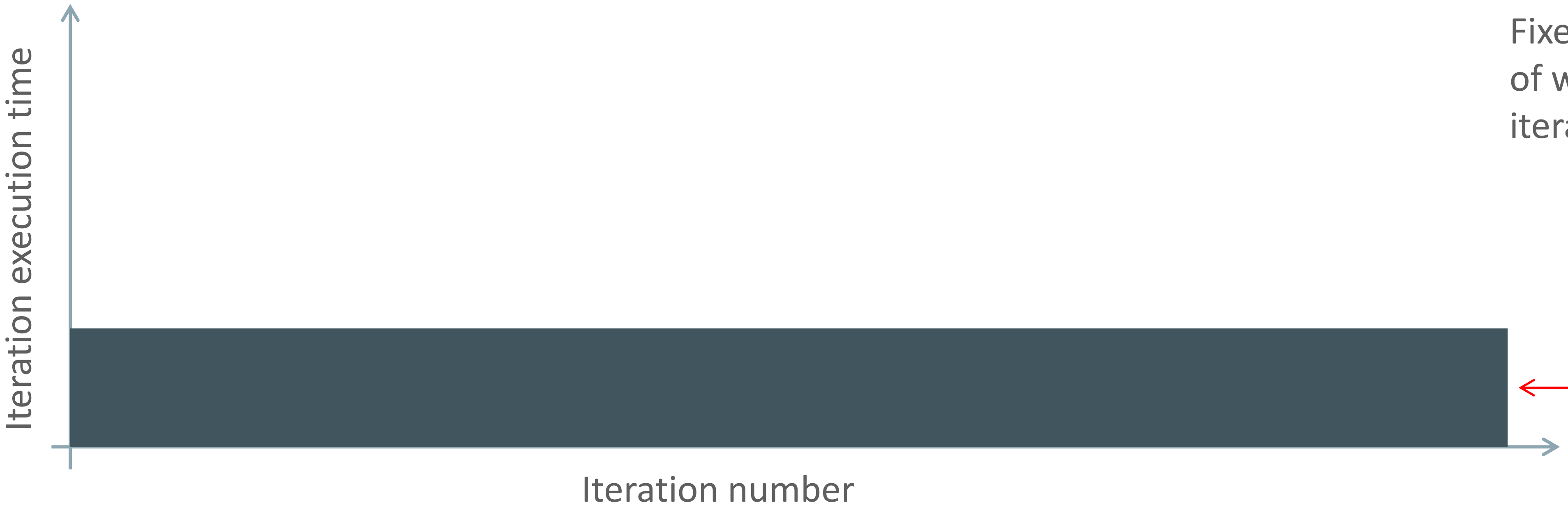
- Allocation of resources to a query
- Distribution of work and data within a machine



```
parallel_for<node_t>([&](node_t n) {
    …
});
```

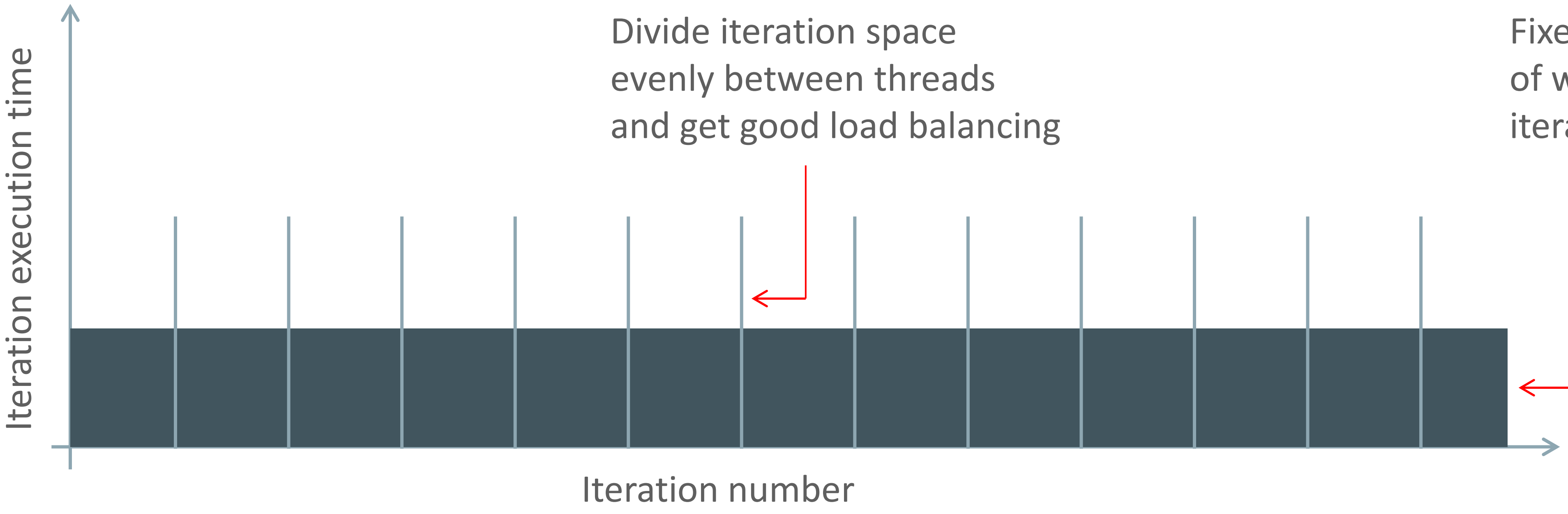


Batch size / load imbalance trade-off



Fixed amount of work in each iteration

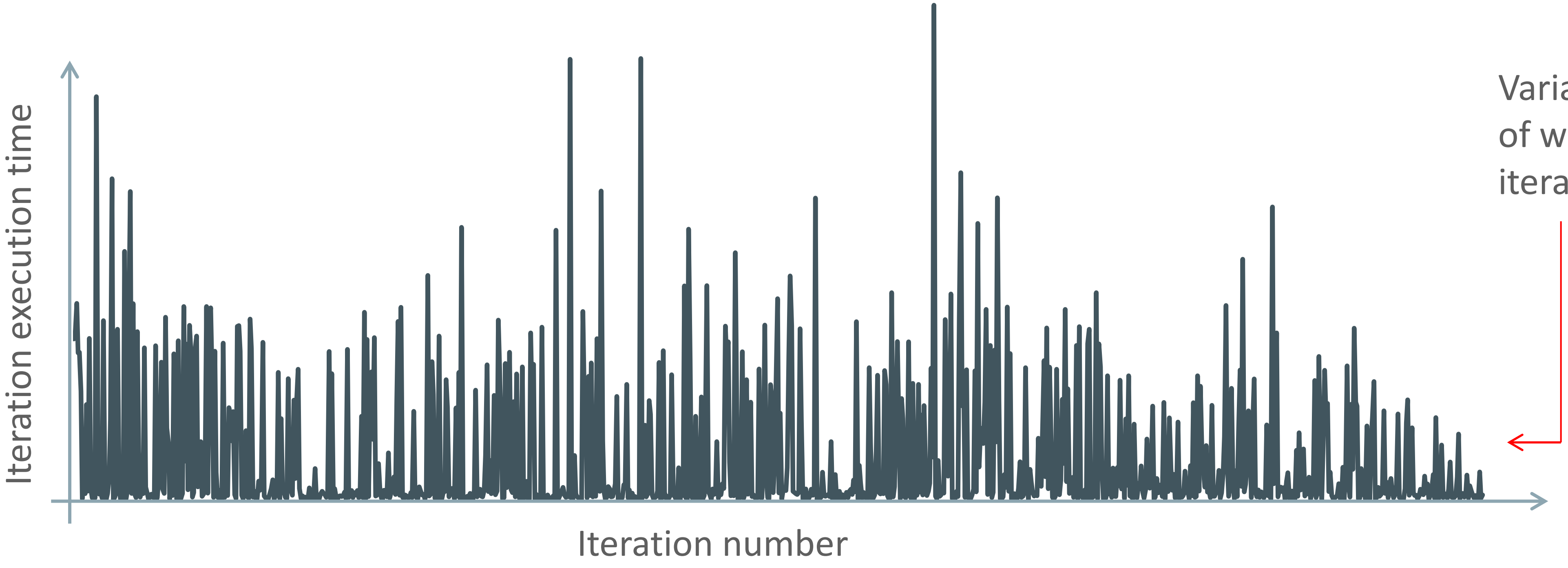
Batch size / load imbalance trade-off



Divide iteration space evenly between threads and get good load balancing

Fixed amount of work in each iteration

Batch size / load imbalance trade-off



Variable amount of work per iteration

(Actual data – #out-edges of the top 1000 nodes in the SNAP Twitter dataset)

Batch size / load imbalance trade-off

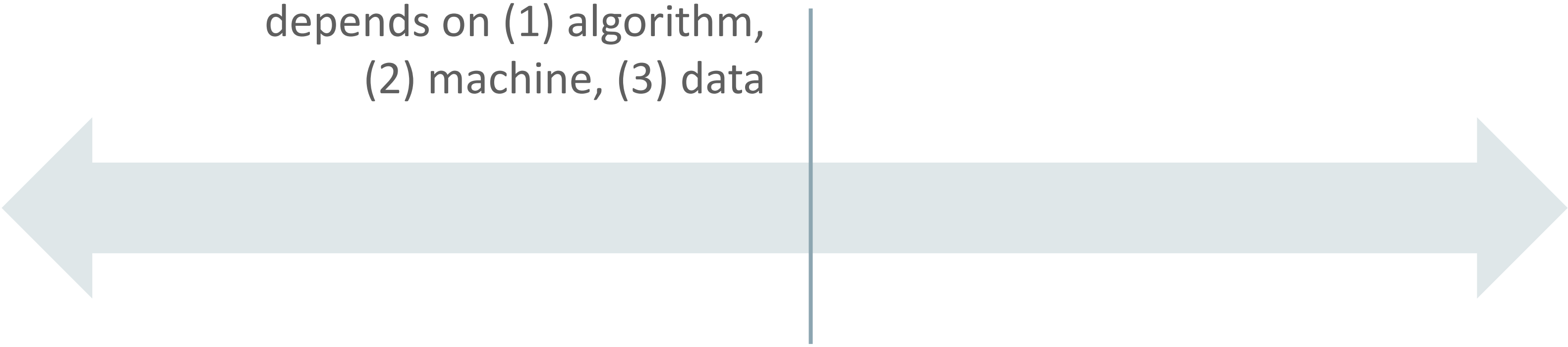
Divide into large batches

Reduce contention distributing work
Risk load imbalance

Divide into small batches

Increase contention distributing work
Achieve better load balance

Batch size / load imbalance trade-off



Typically, choose manually –
but getting this right
depends on (1) algorithm,
(2) machine, (3) data

Divide into large batches

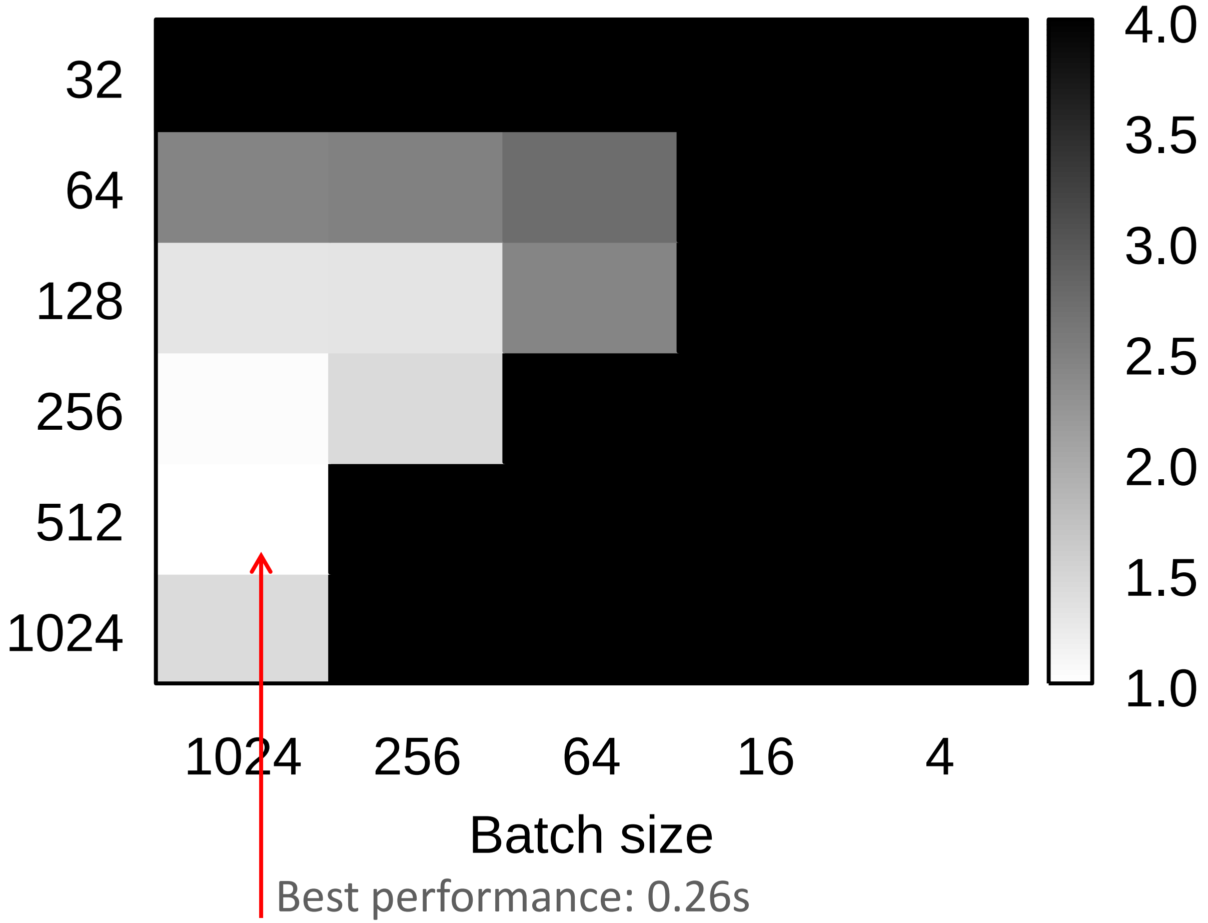
Reduce contention distributing work
Risk load imbalance

Divide into small batches

Increase contention distributing work
Achieve better load balance

PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)

OpenMP static & dynamic loops



Best performance: 0.26s

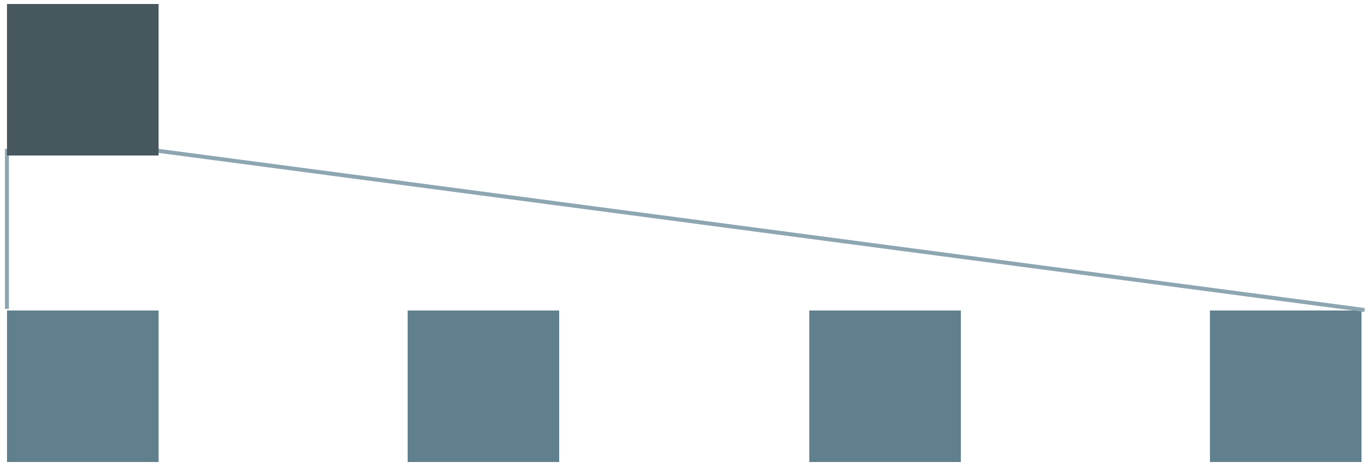
8-socket SPARC T5
16 cores per socket
8 h/w threads per core

PageRank
SNAP LiveJournal data set

My laptop

1 socket

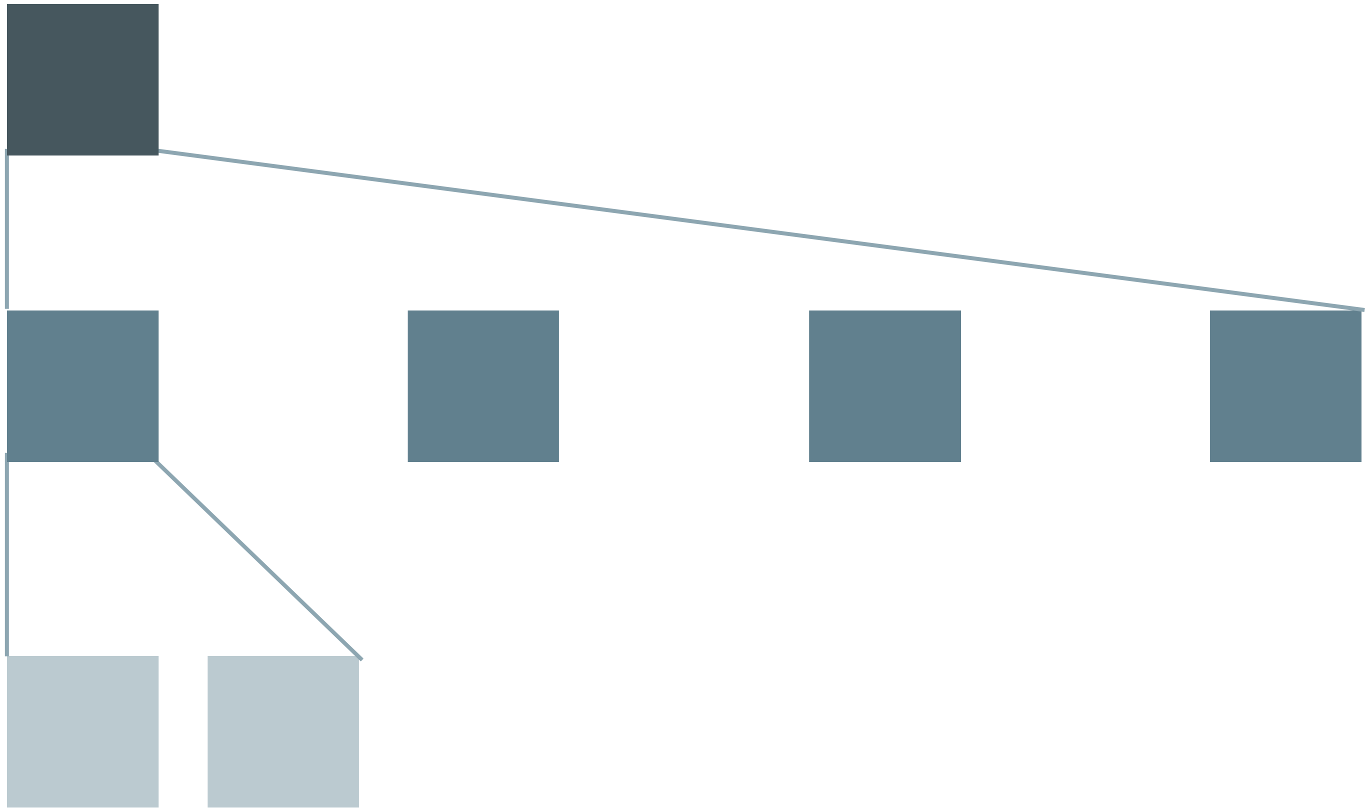
My laptop



1 socket

4 cores per socket

My laptop

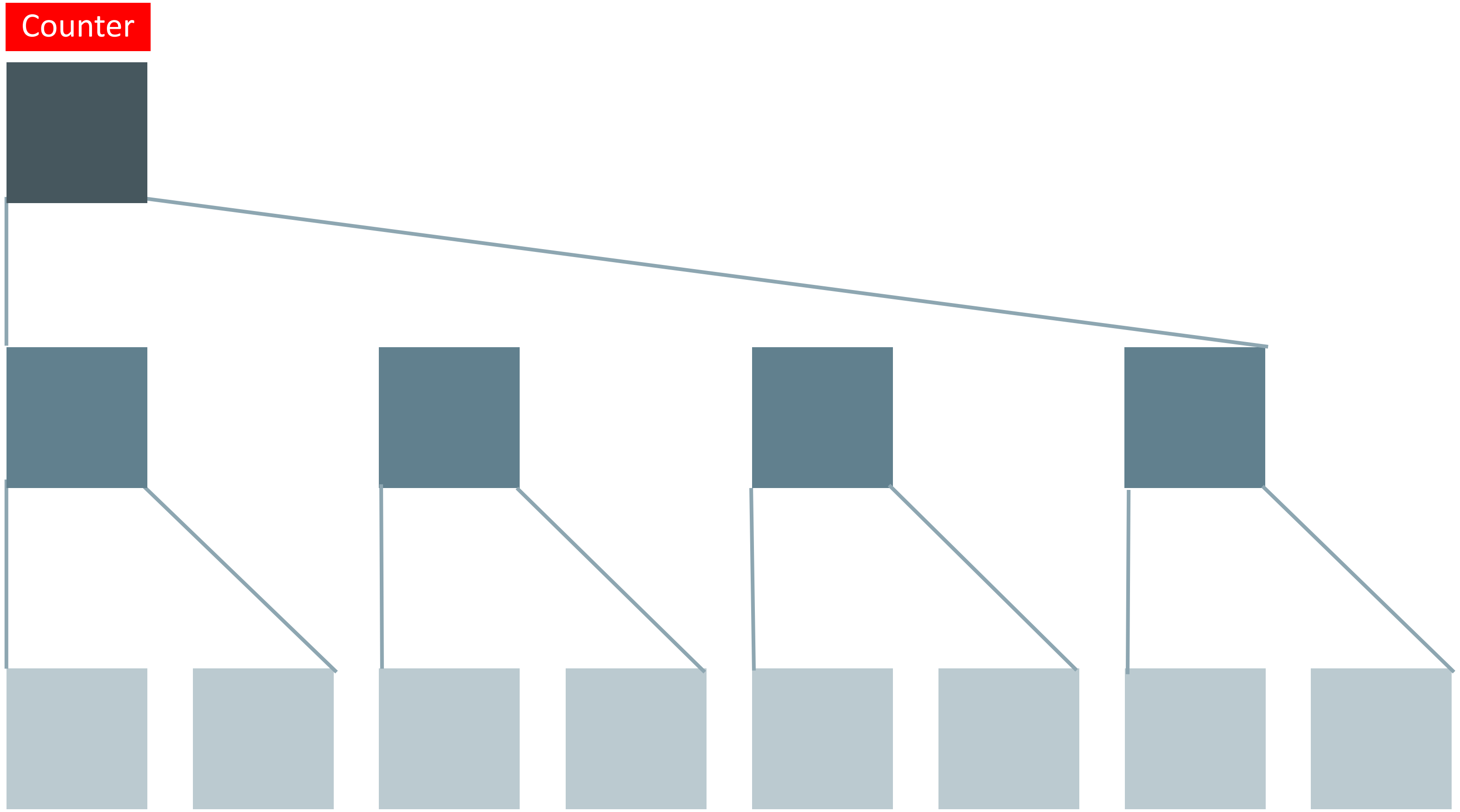


1 socket

4 cores per socket

2 h/w contexts per core

My laptop

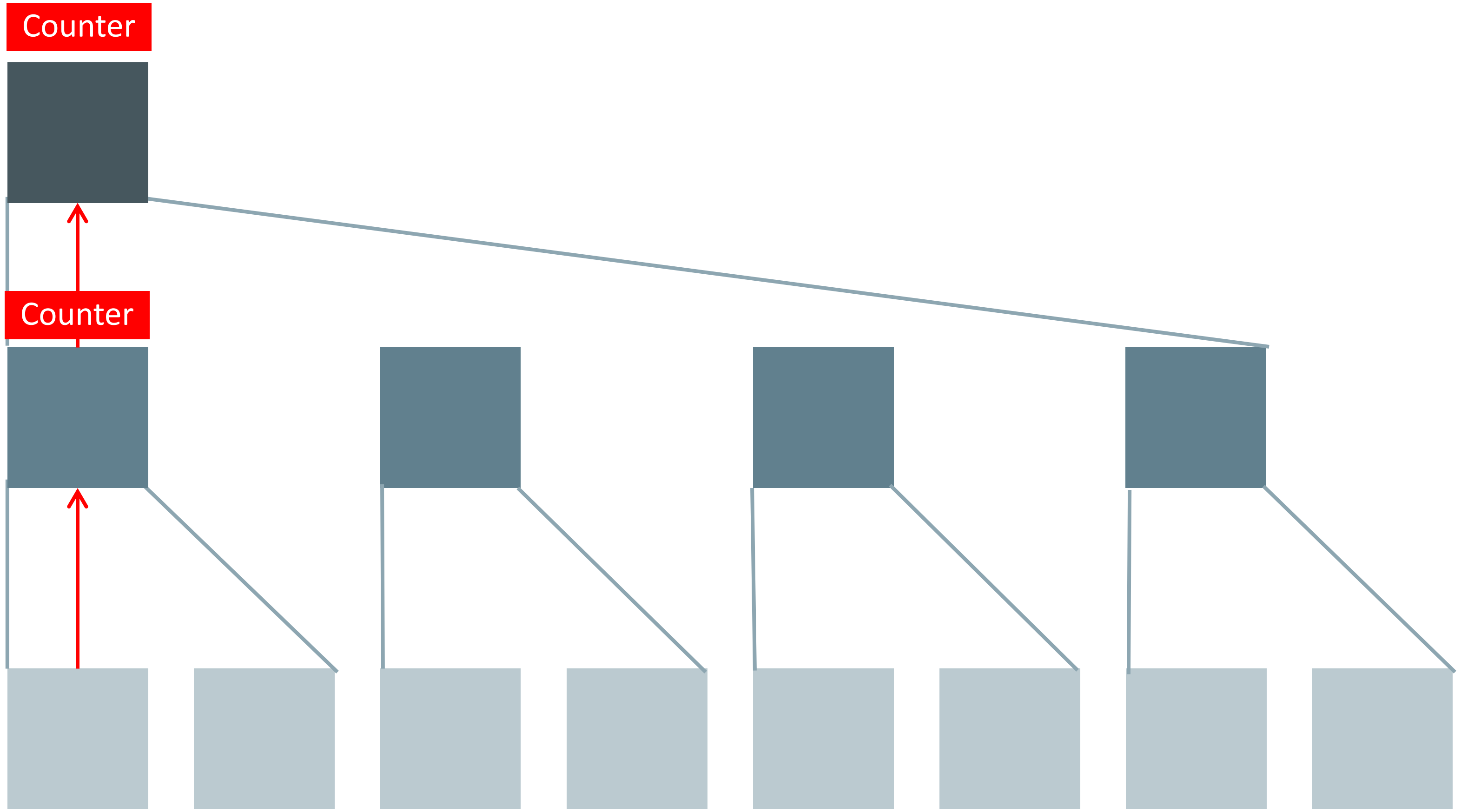


1 socket

4 cores per socket

2 h/w contexts per core

My laptop

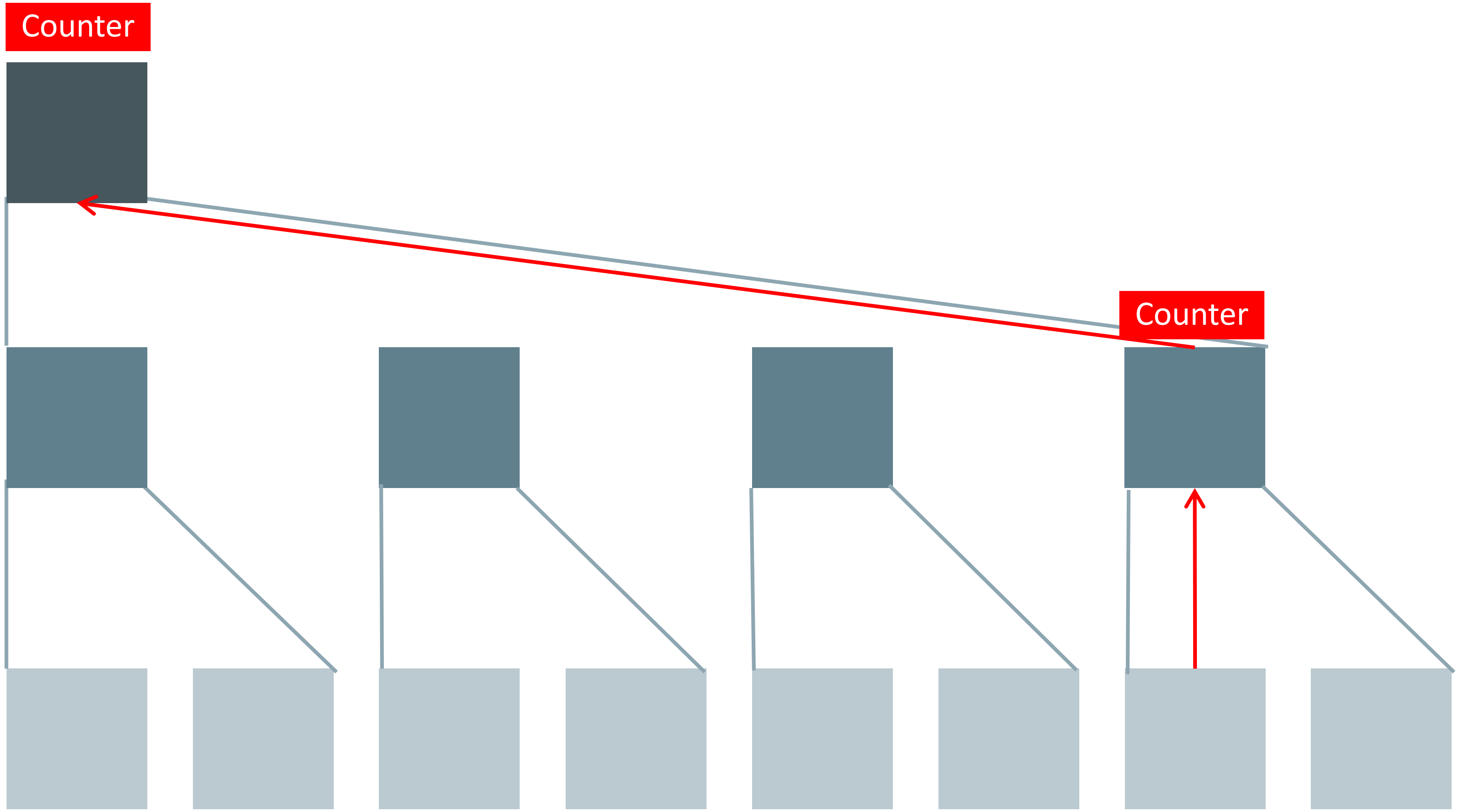


1 socket

4 cores per socket

2 h/w contexts per core

My laptop

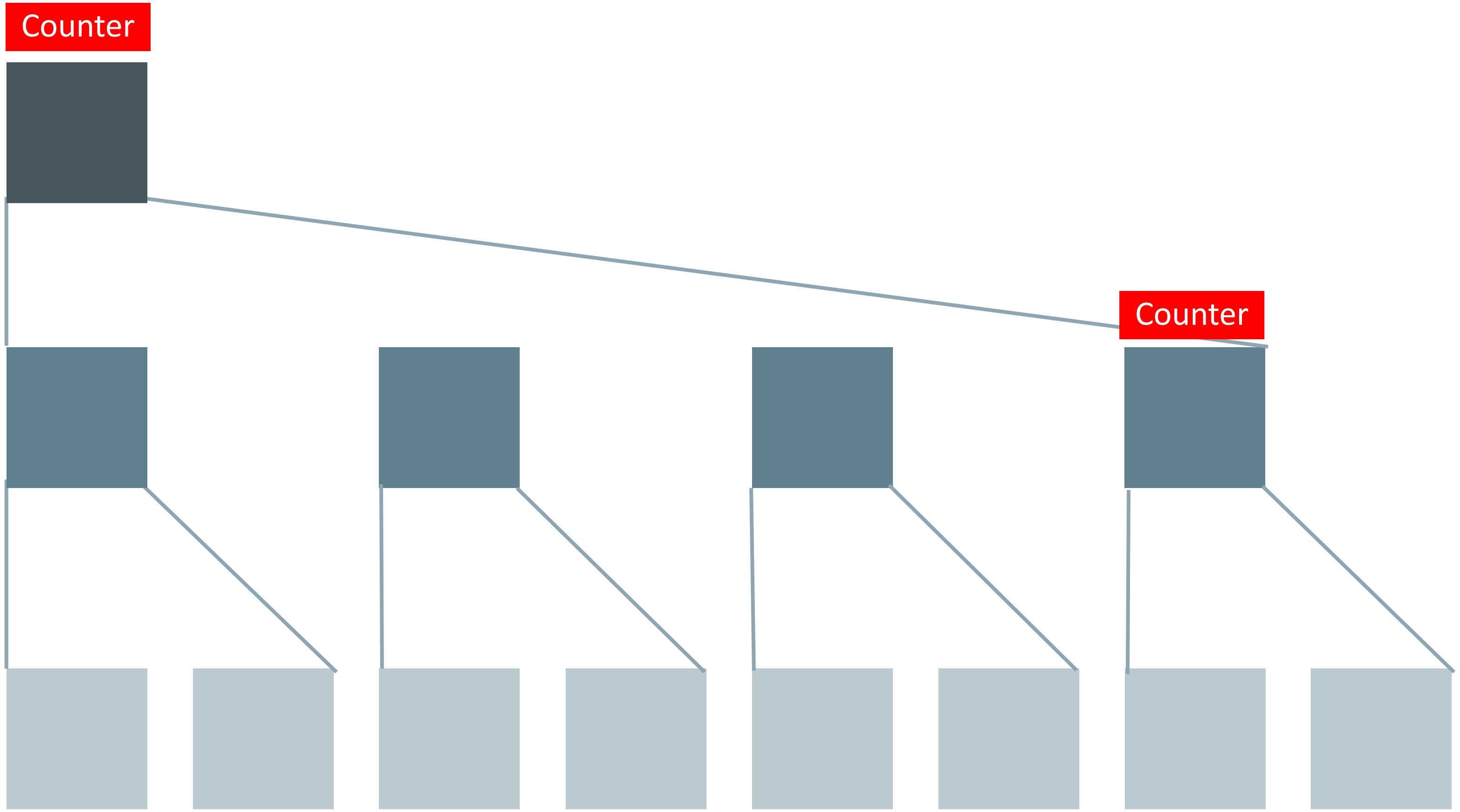


1 socket

4 cores per socket

2 h/w contexts per core

My laptop



1 socket

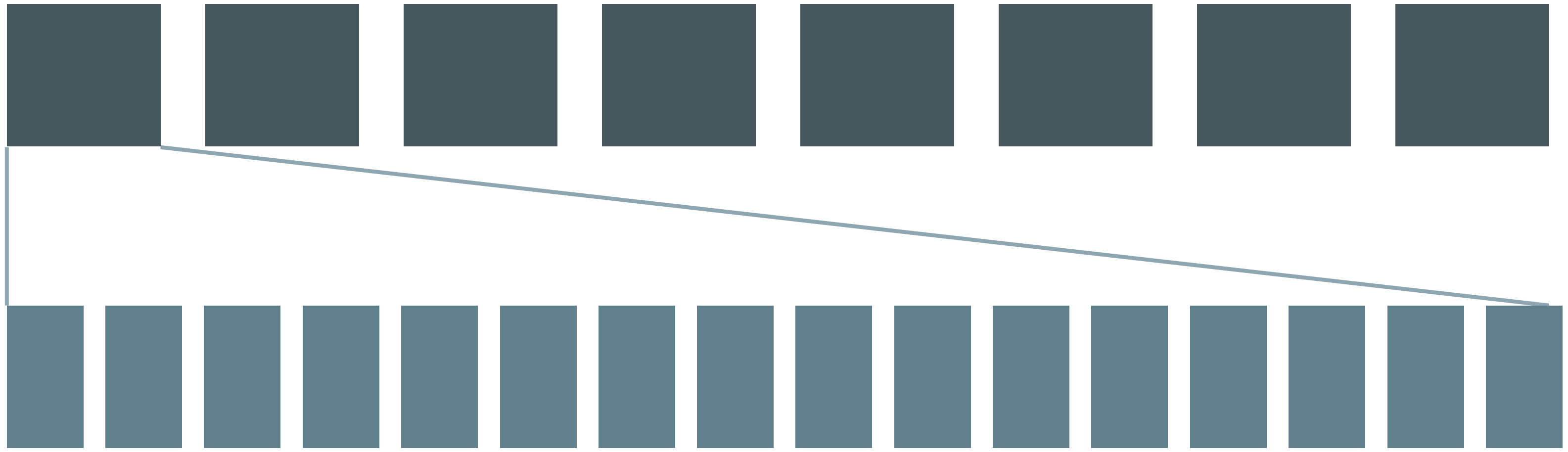
4 cores per socket

2 h/w contexts per core

T5-8

8 sockets

T5-8

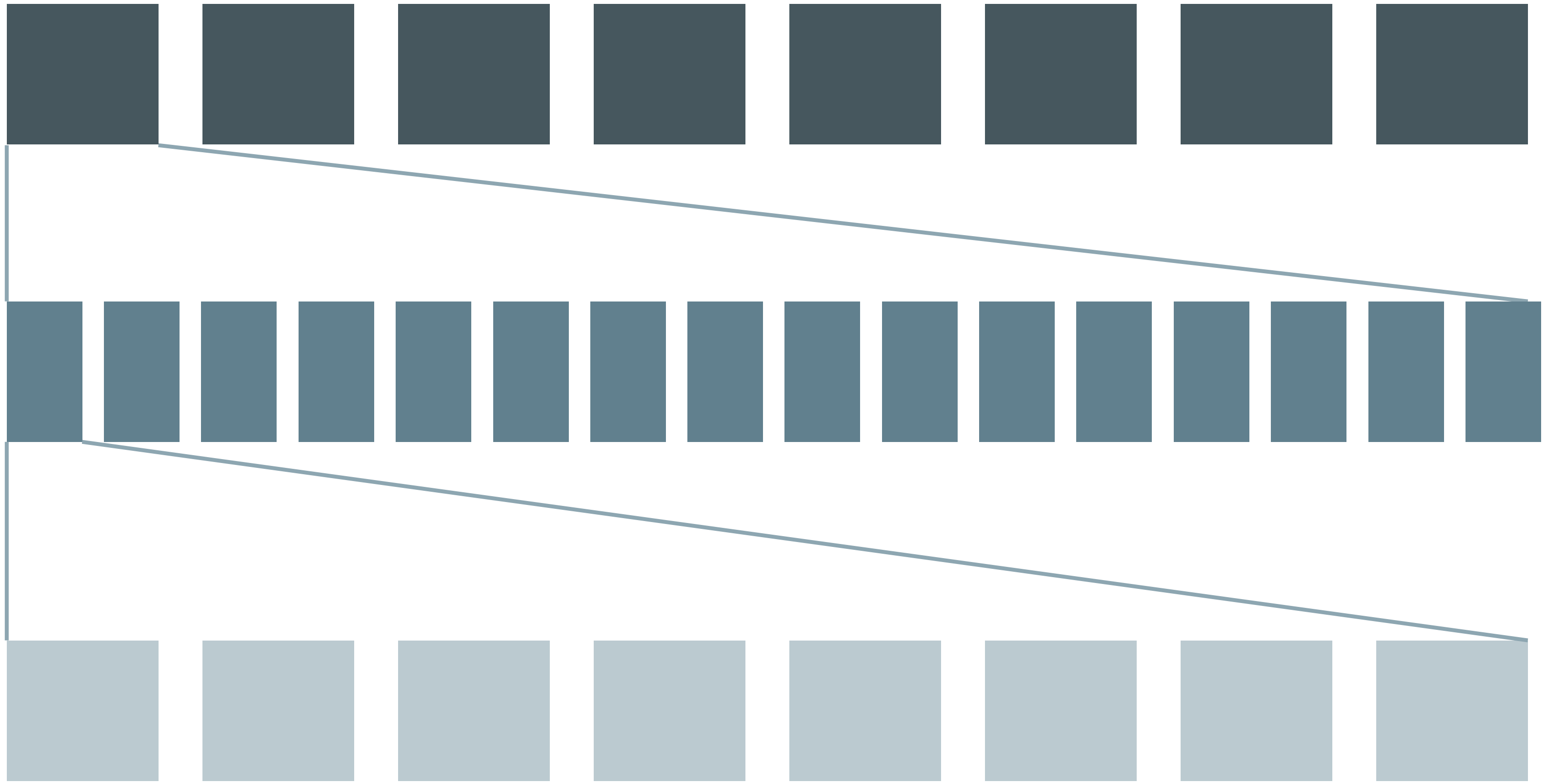


8 sockets

16 cores per socket

ORACLE

T5-8

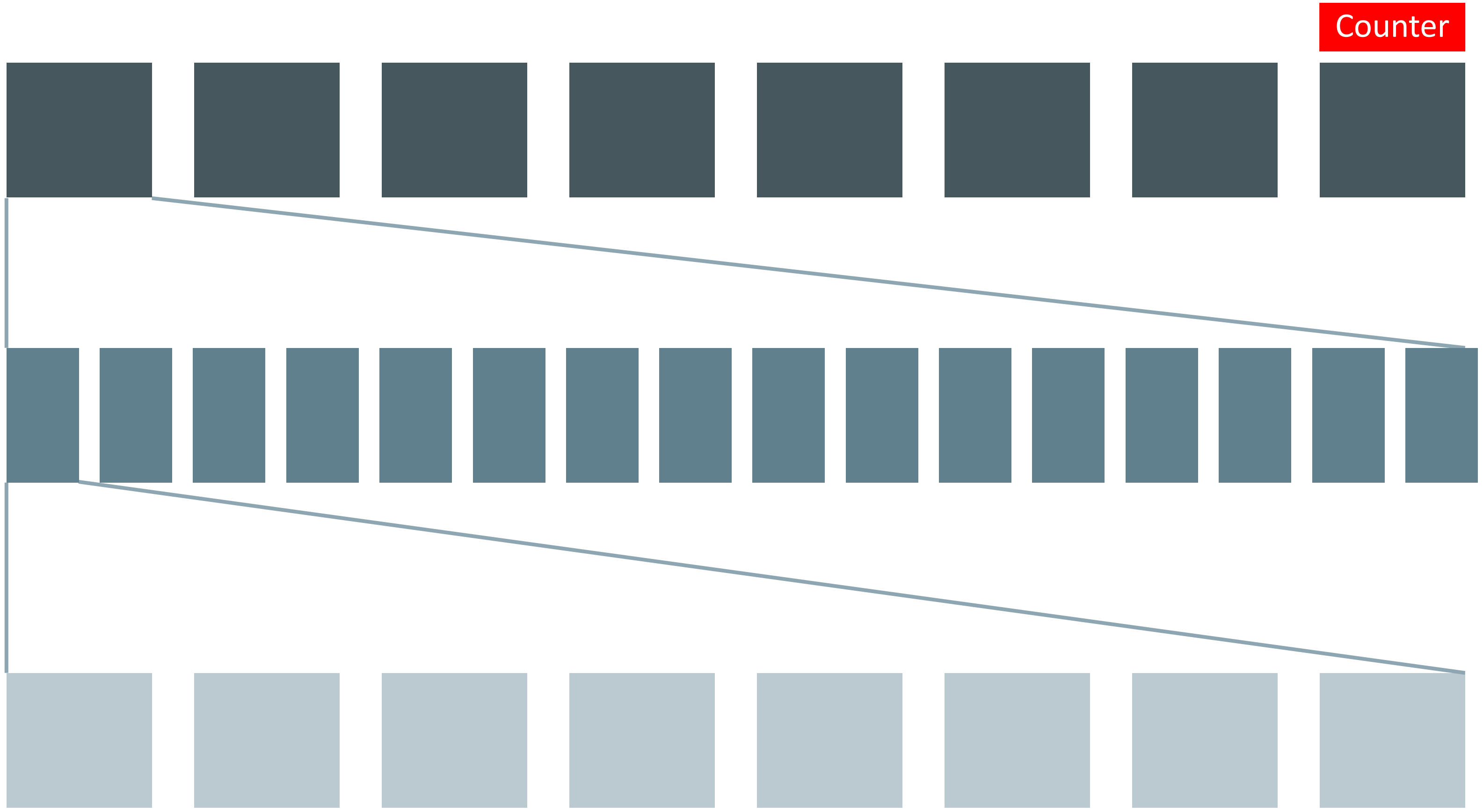


8 sockets

16 cores per socket

8 h/w contexts per core

T5-8

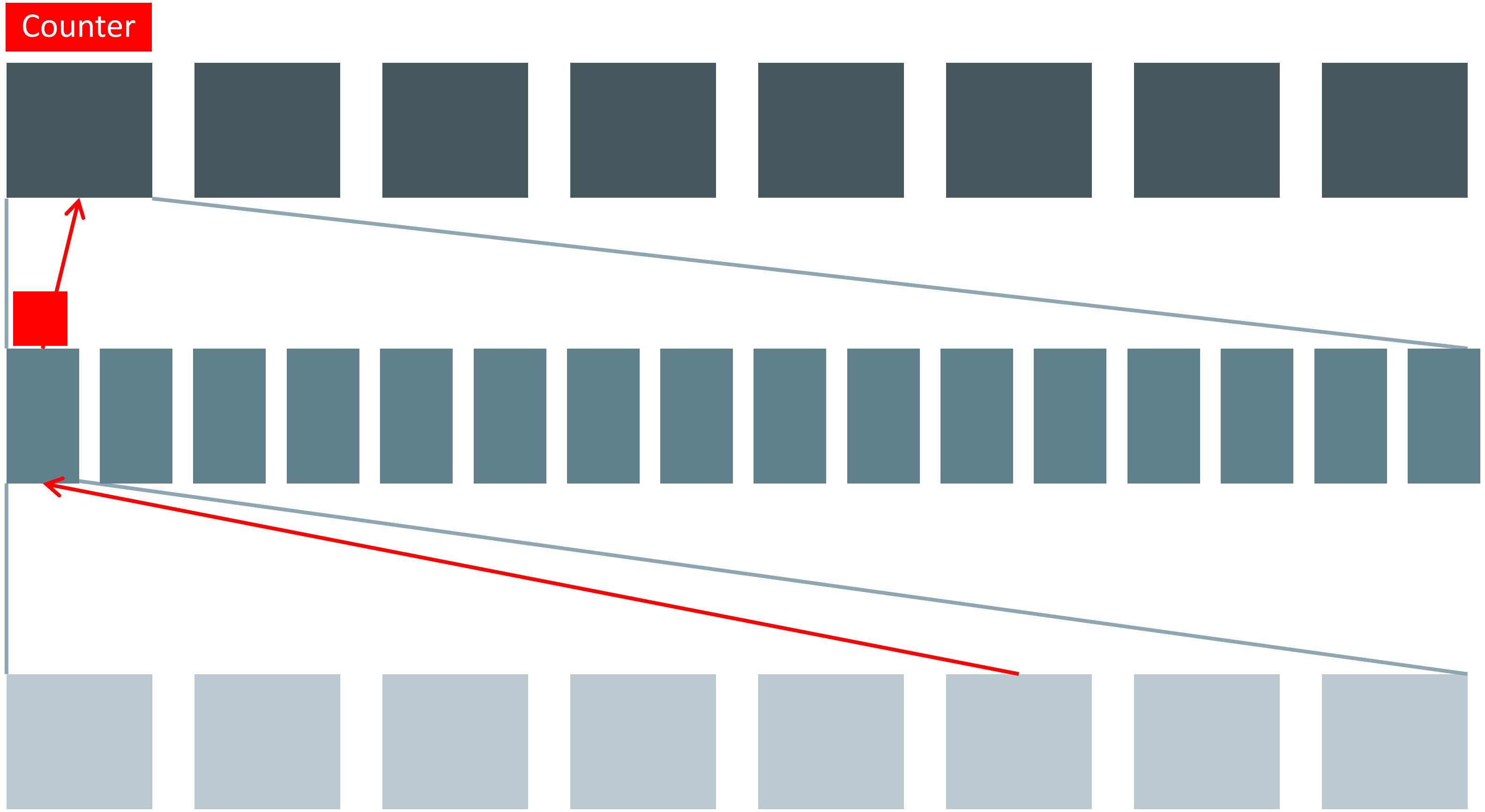


8 sockets

16 cores per socket

8 h/w contexts per core

T5-8

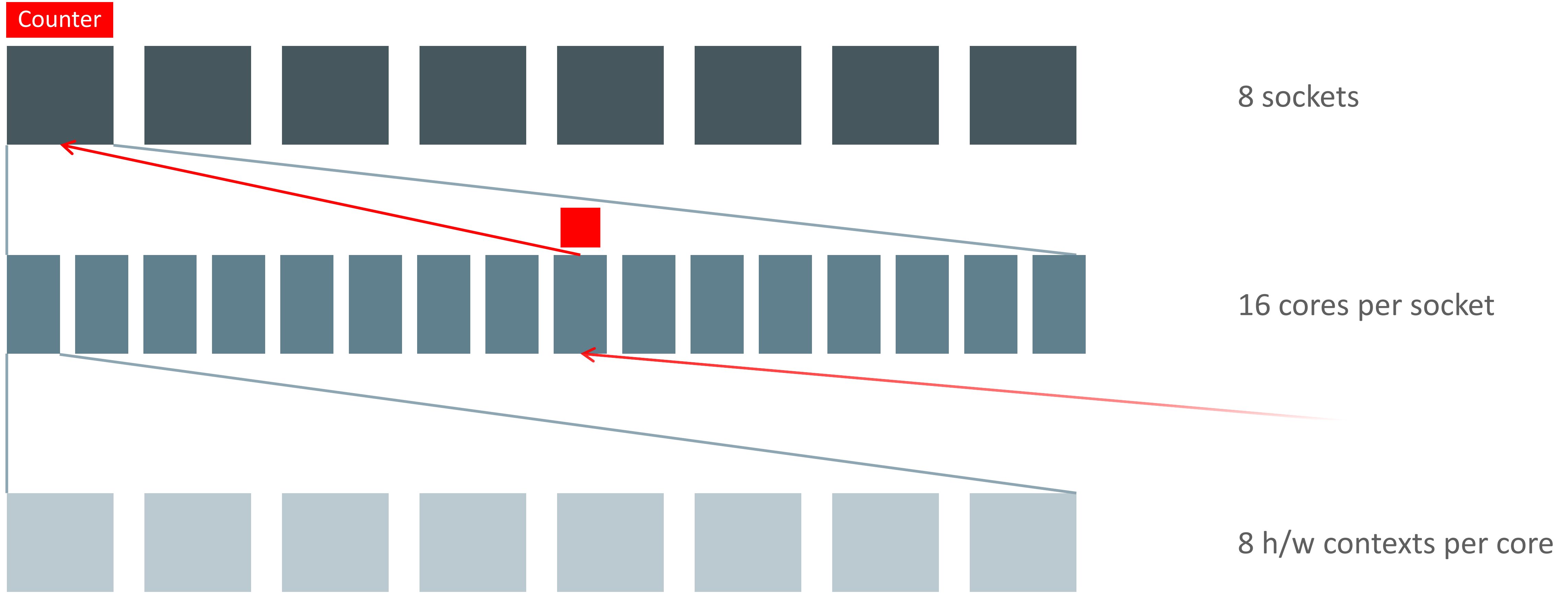


8 sockets

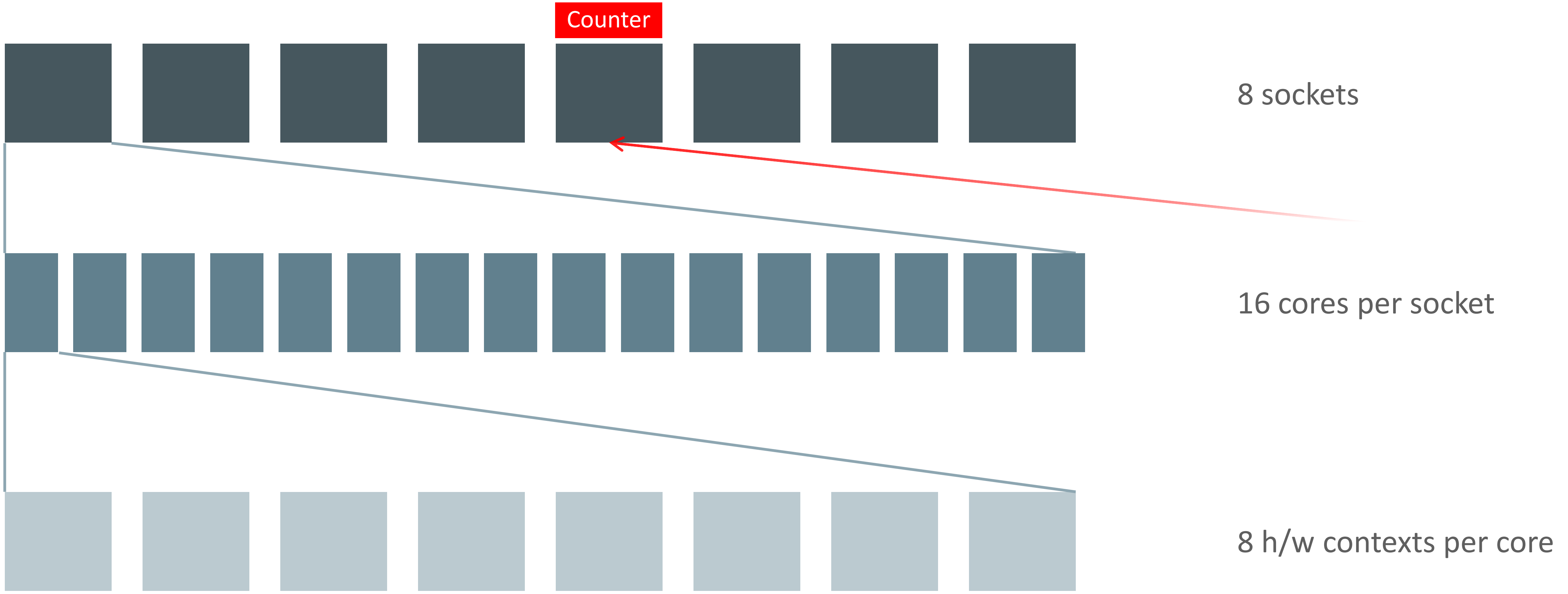
16 cores per socket

8 h/w contexts per core

T5-8



T5-8



The problem

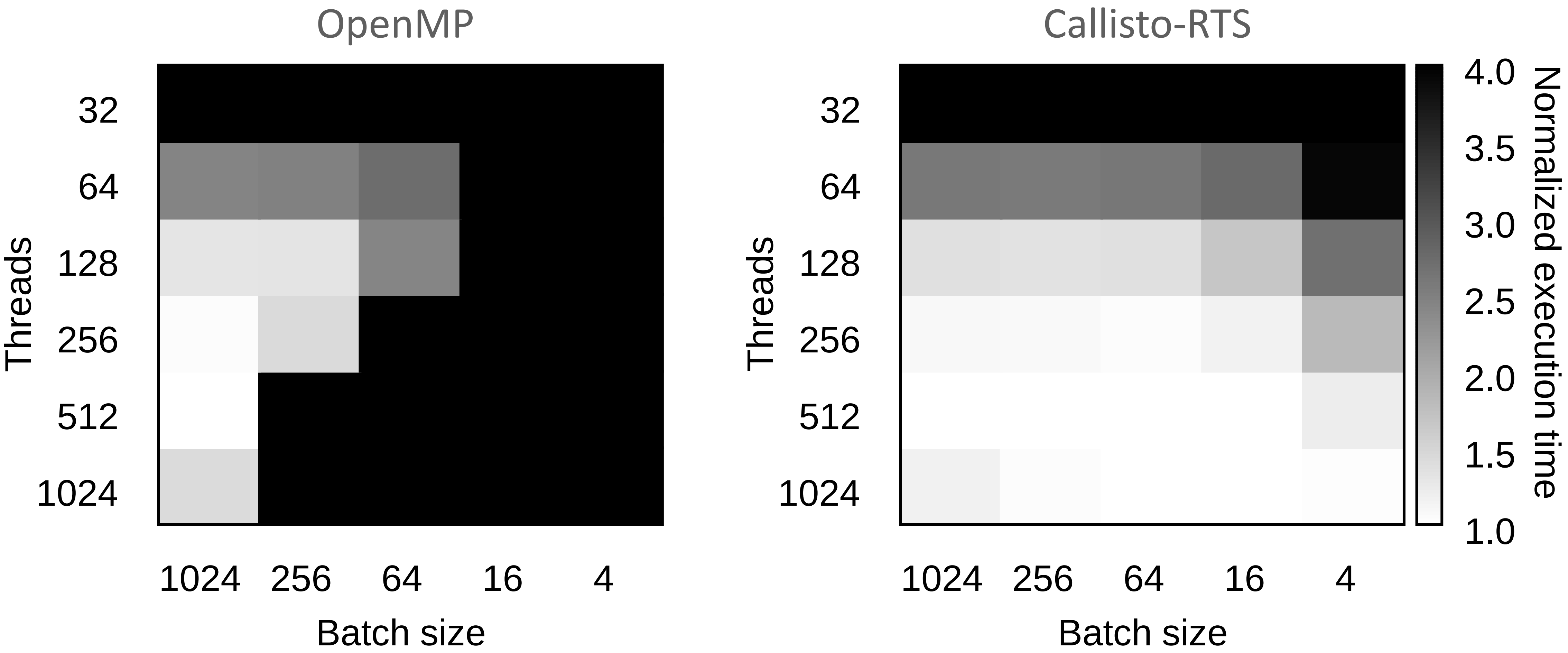
My laptop

- 8 Threads accessing the counter
- The counter is always on the required socket
- 1 time in 4 the counter is on the required core

T5-8

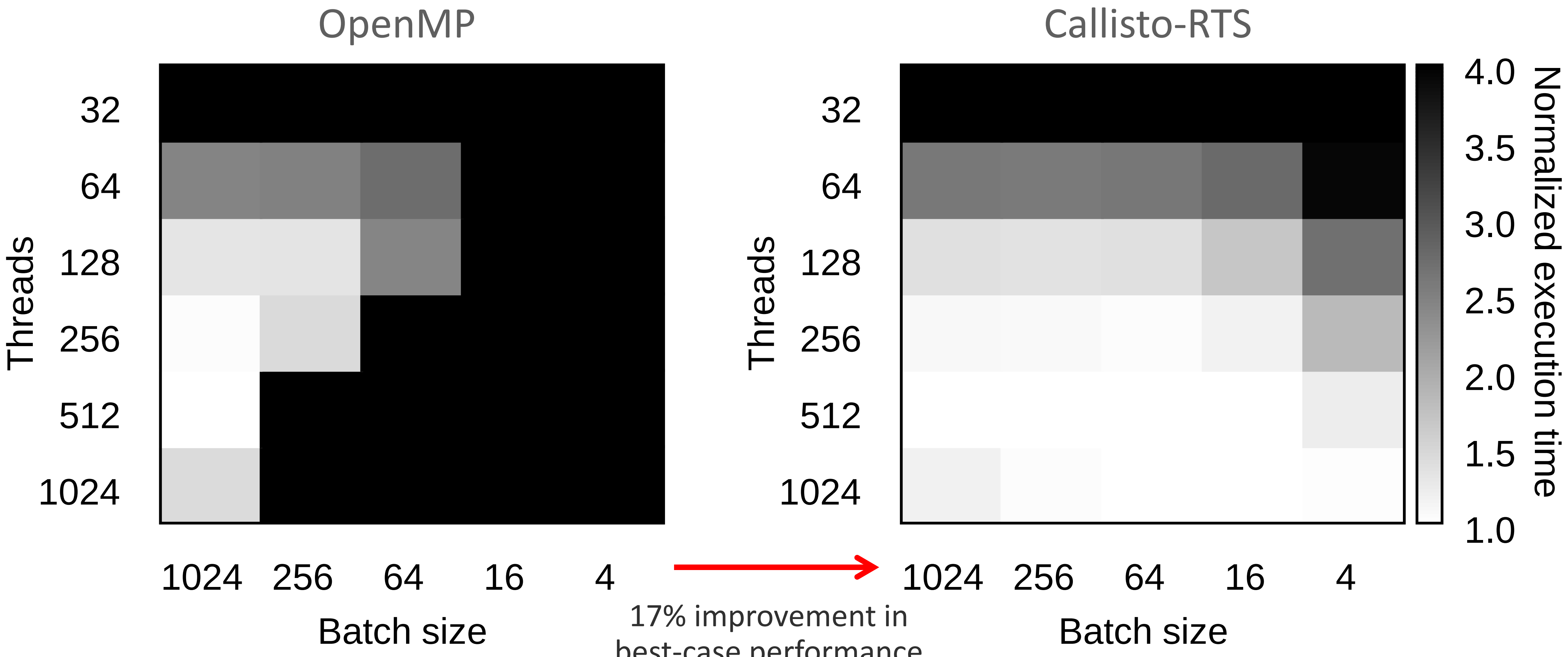
- 1024 Threads accessing the counter
- 1 time in 8 the counter is on the required socket
- 1 time in 128 the counter is on the required core

PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)



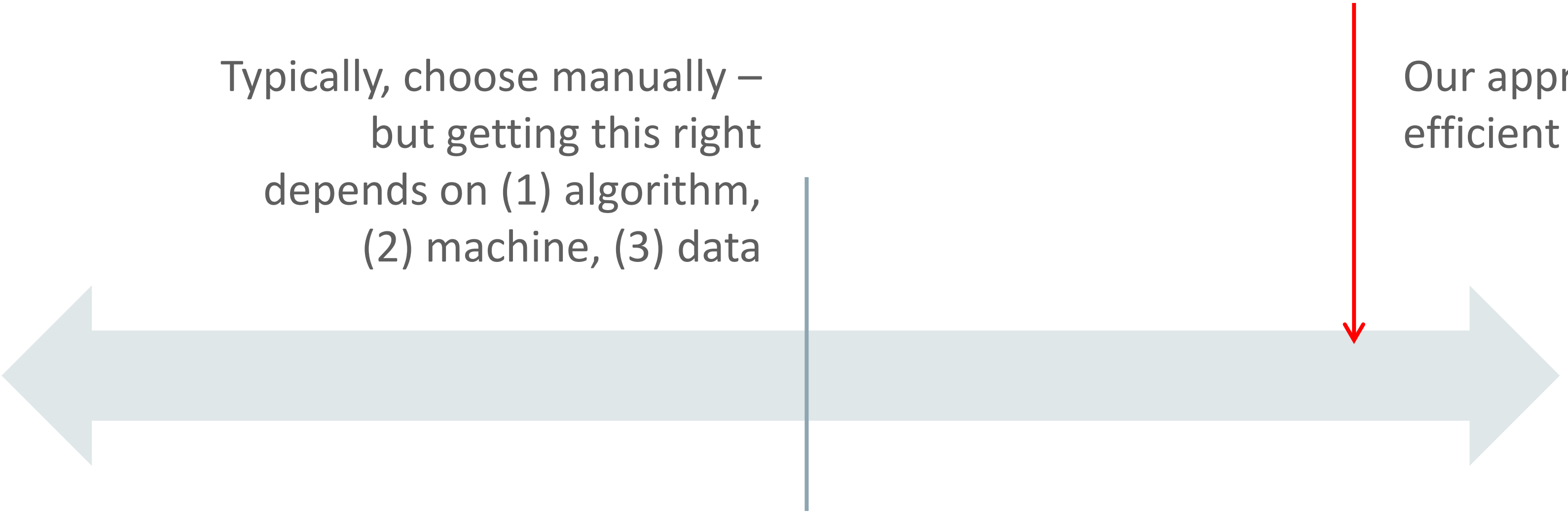
ORACLE

PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)



17% improvement in best-case performance

Batch size / load imbalance trade-off



Typically, choose manually – but getting this right depends on (1) algorithm, (2) machine, (3) data

Our approach: support efficient small batches

Divide into large batches

Reduce contention
Risk load imbalance

Divide into small batches

Increase contention distributing work
Achieve better load balance

ORACLE

Techniques

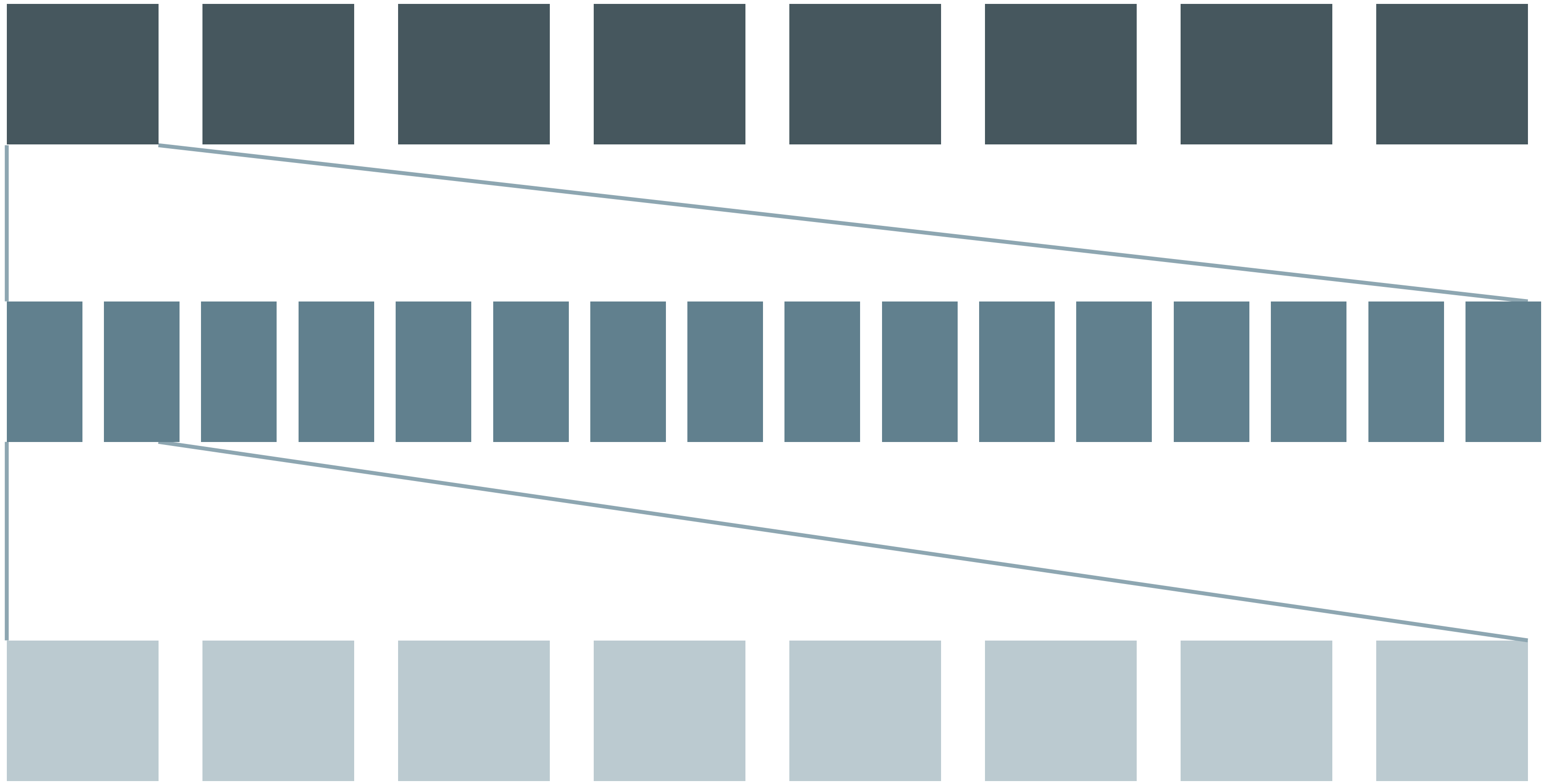
1. Request combining
2. Asynchronous work requests

Techniques

1 Request combining

2 Asynchronous work requests

Approach



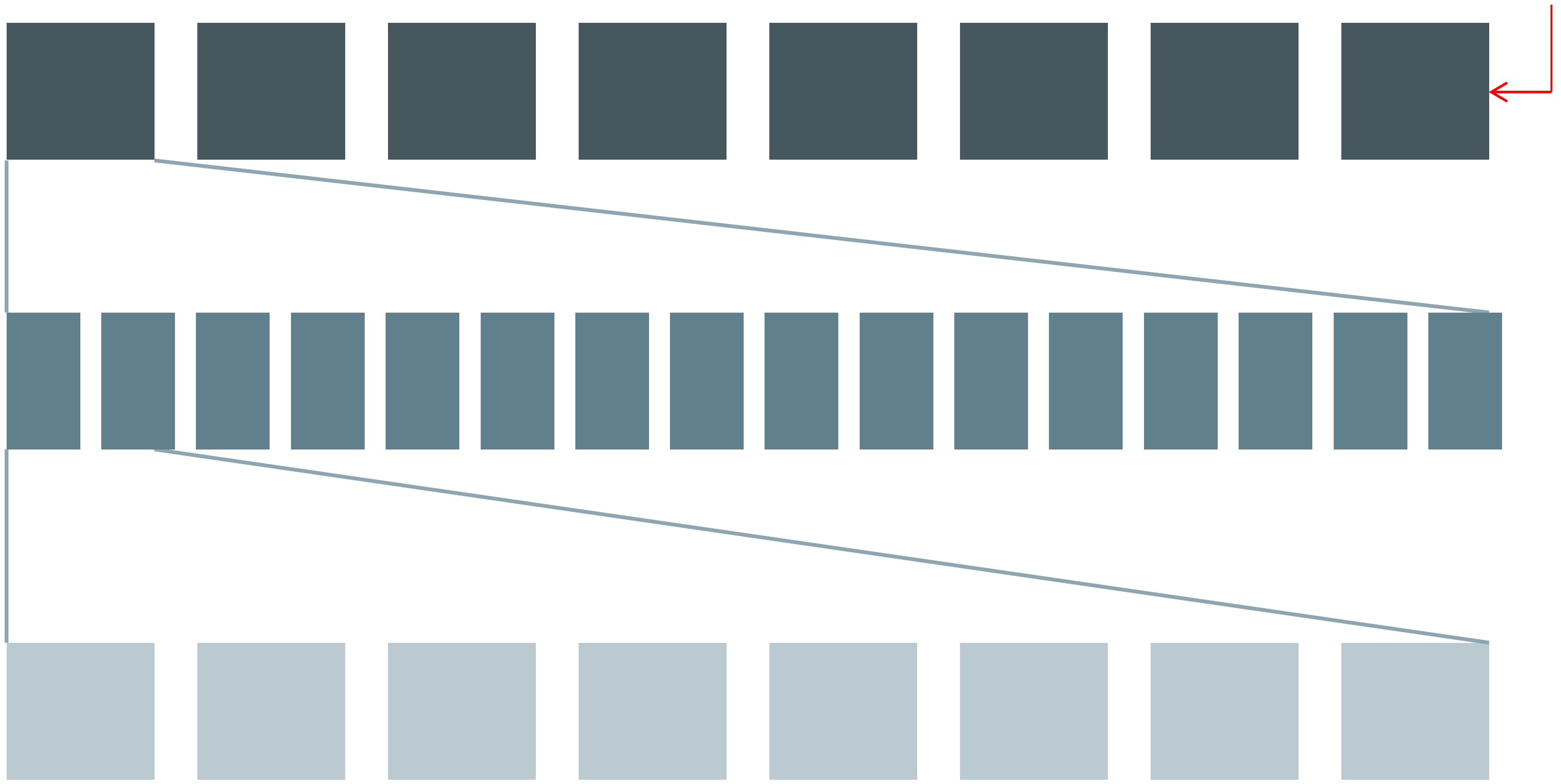
8 sockets

16 cores per socket

8 h/w contexts per core

Approach

Per-socket iteration counters, reducing communication between sockets. Steal from other sockets when own work complete.

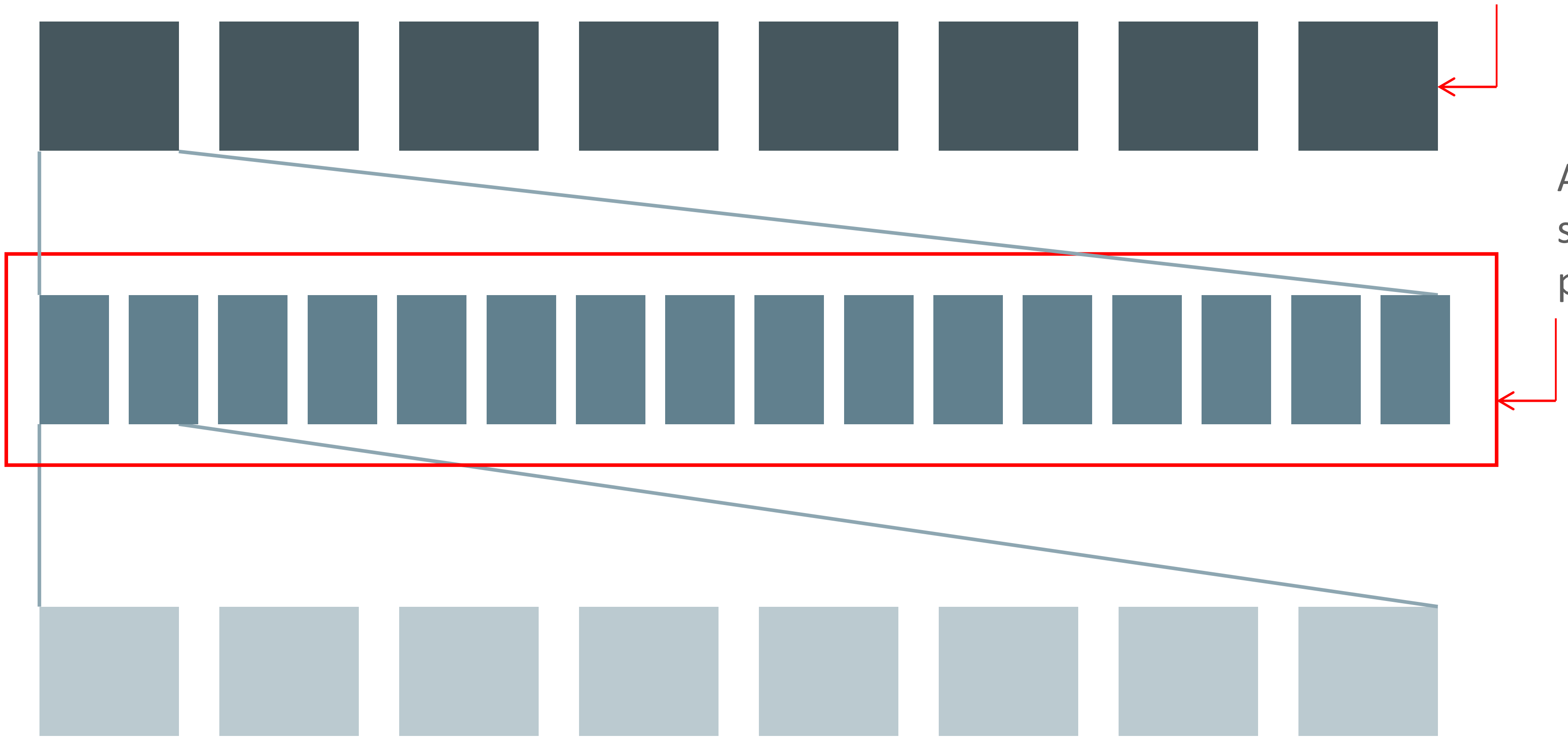


8 sockets

16 cores per socket

8 h/w contexts per core

Approach



Per-socket iteration counters, reducing communication between sockets. Steal from other sockets when own work complete.

8 sockets

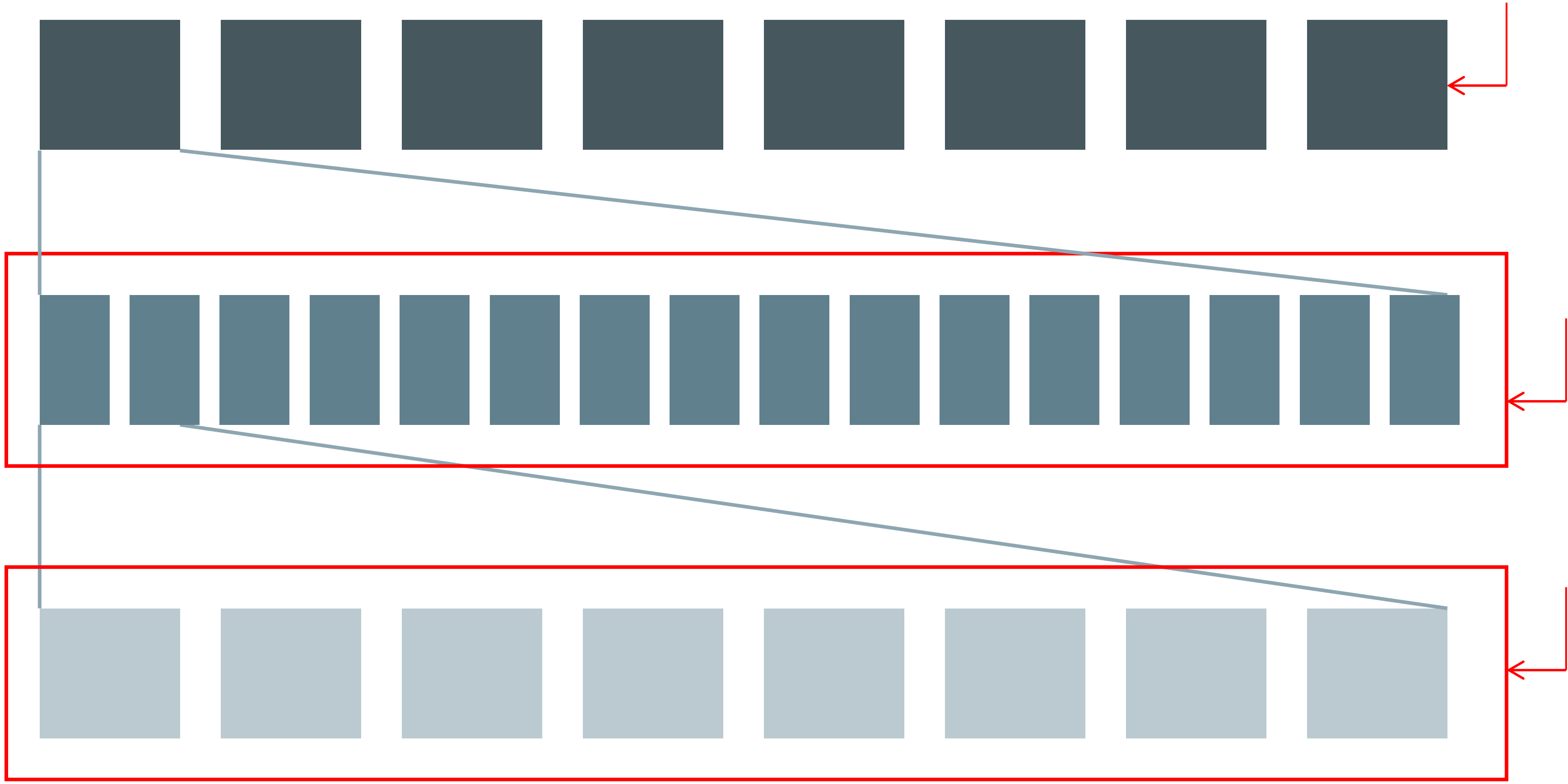
Aggregate requests within a socket, reducing contention on per-socket counter.

16 cores per socket

8 h/w contexts per core

Approach

Per-socket iteration counters, reducing communication between sockets. Steal from other sockets when own work complete.



8 sockets

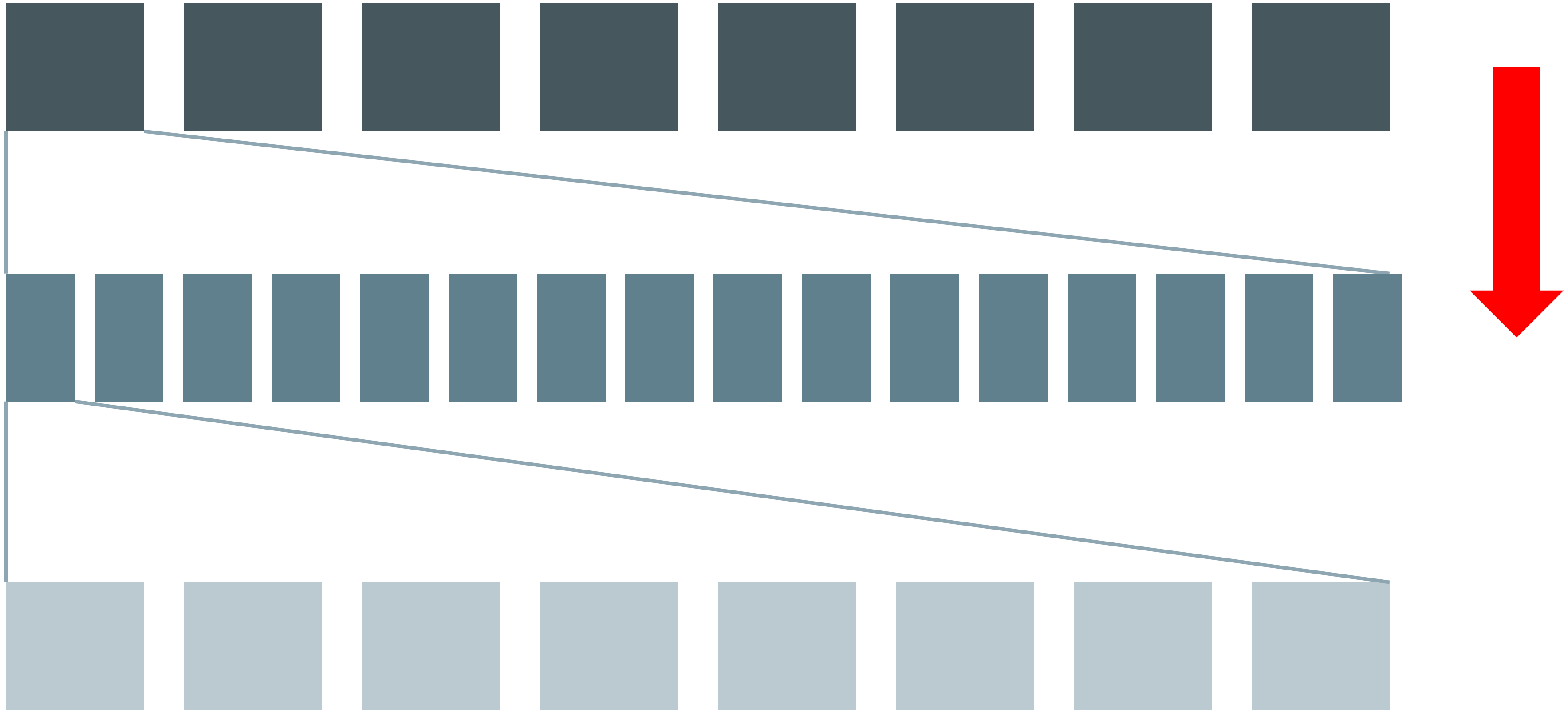
Aggregate requests within a socket, reducing contention on per-socket counter.

16 cores per socket

Further aggregation within a core, exploiting shared cache.

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



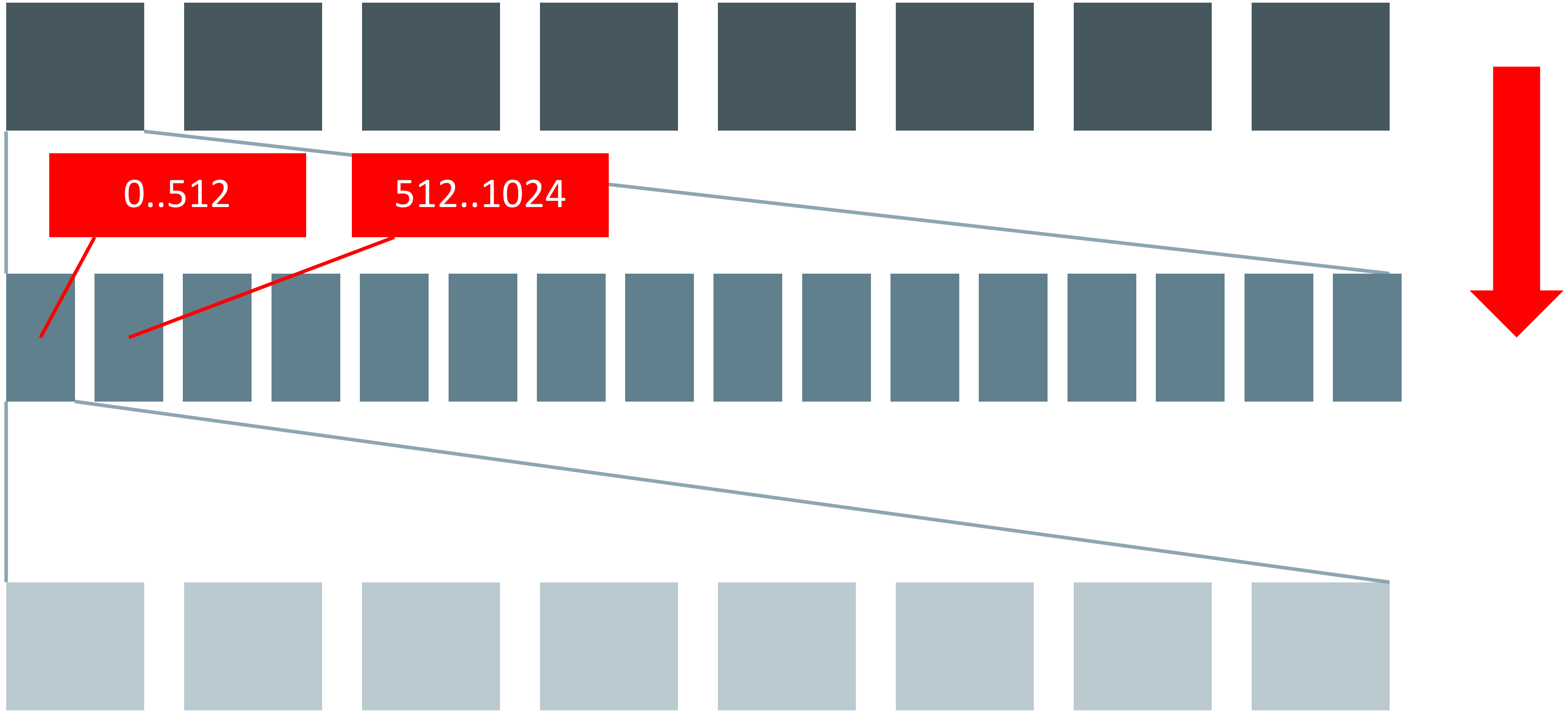
8 sockets

Distribute iterations at start of loop down to per-core counters

16 cores per socket

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



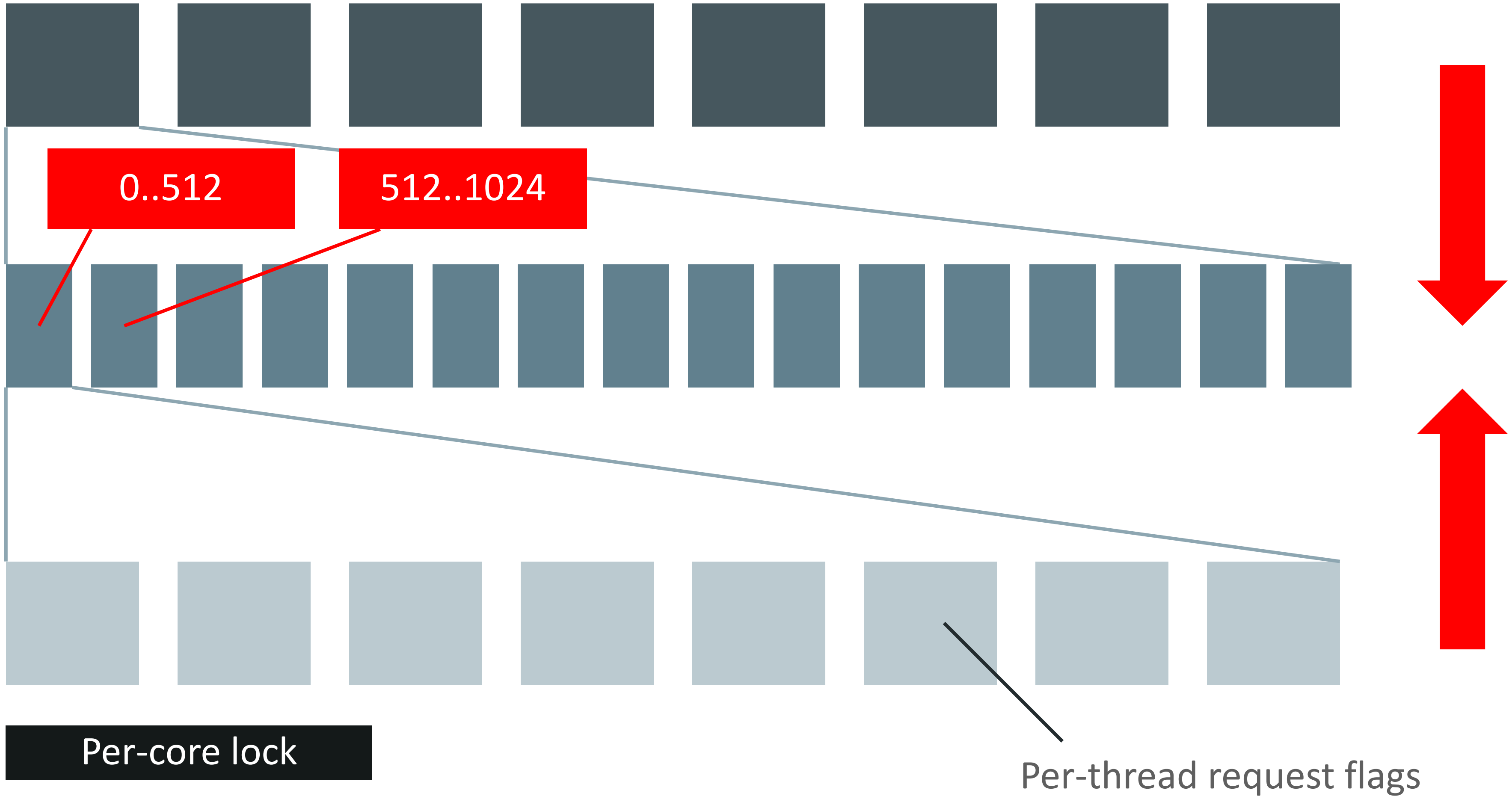
8 sockets

Distribute iterations at start of loop down to per-core counters

16 cores per socket

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



8 sockets

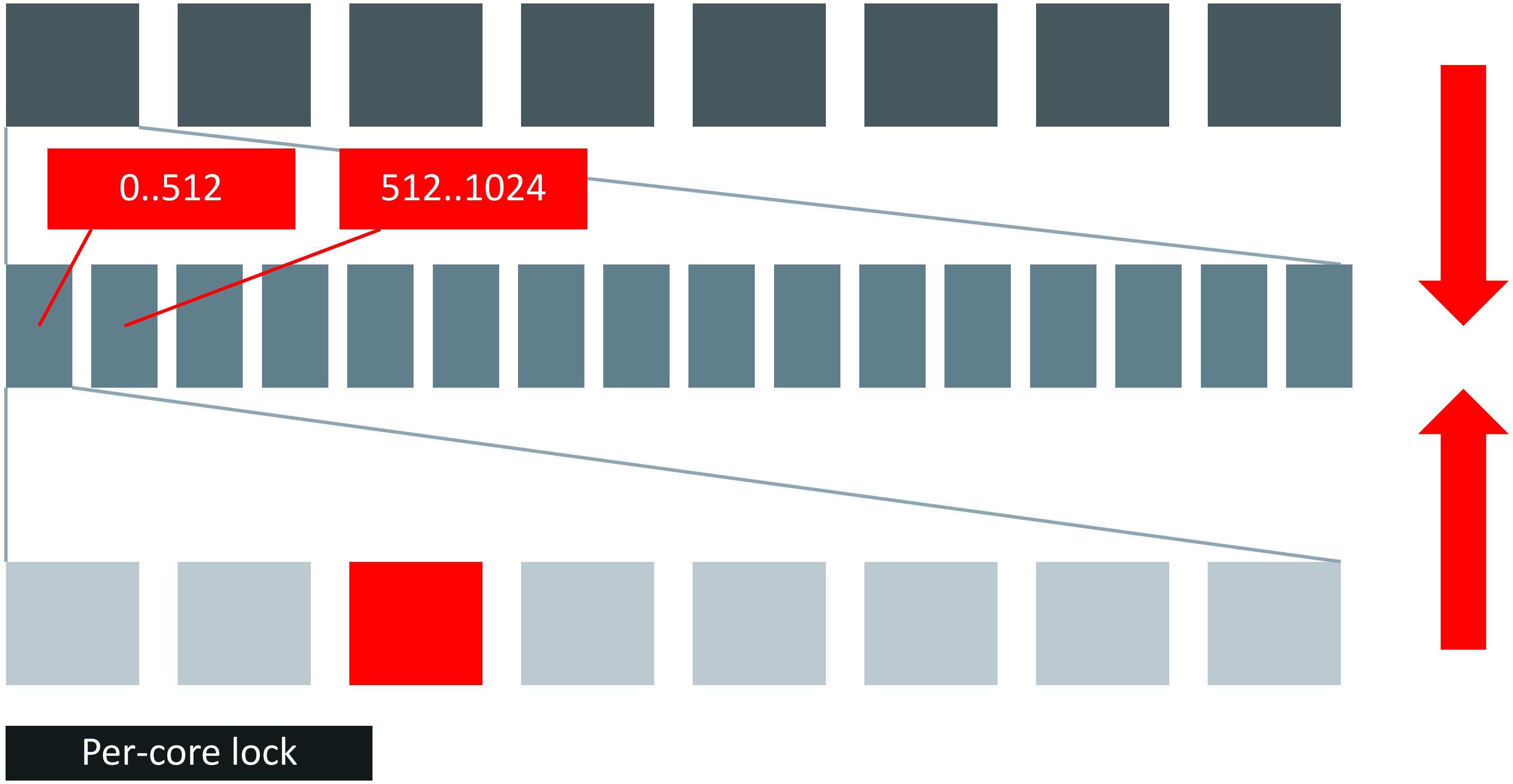
Distribute iterations at start of loop down to per-core counters

16 cores per socket

Aggregate requests upwards within a core

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



8 sockets

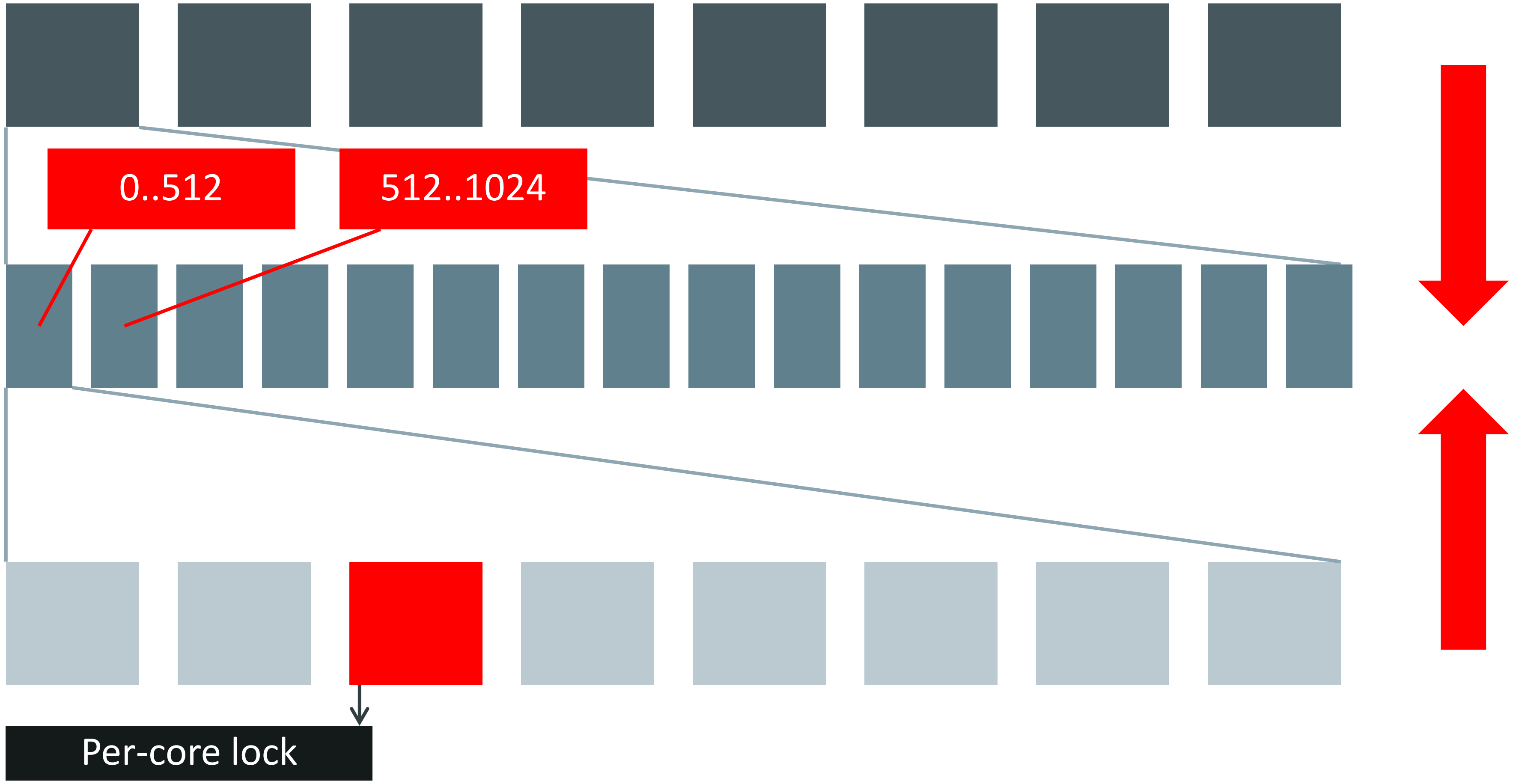
Distribute iterations at start of loop down to per-core counters

16 cores per socket

Aggregate requests upwards within a core

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



8 sockets

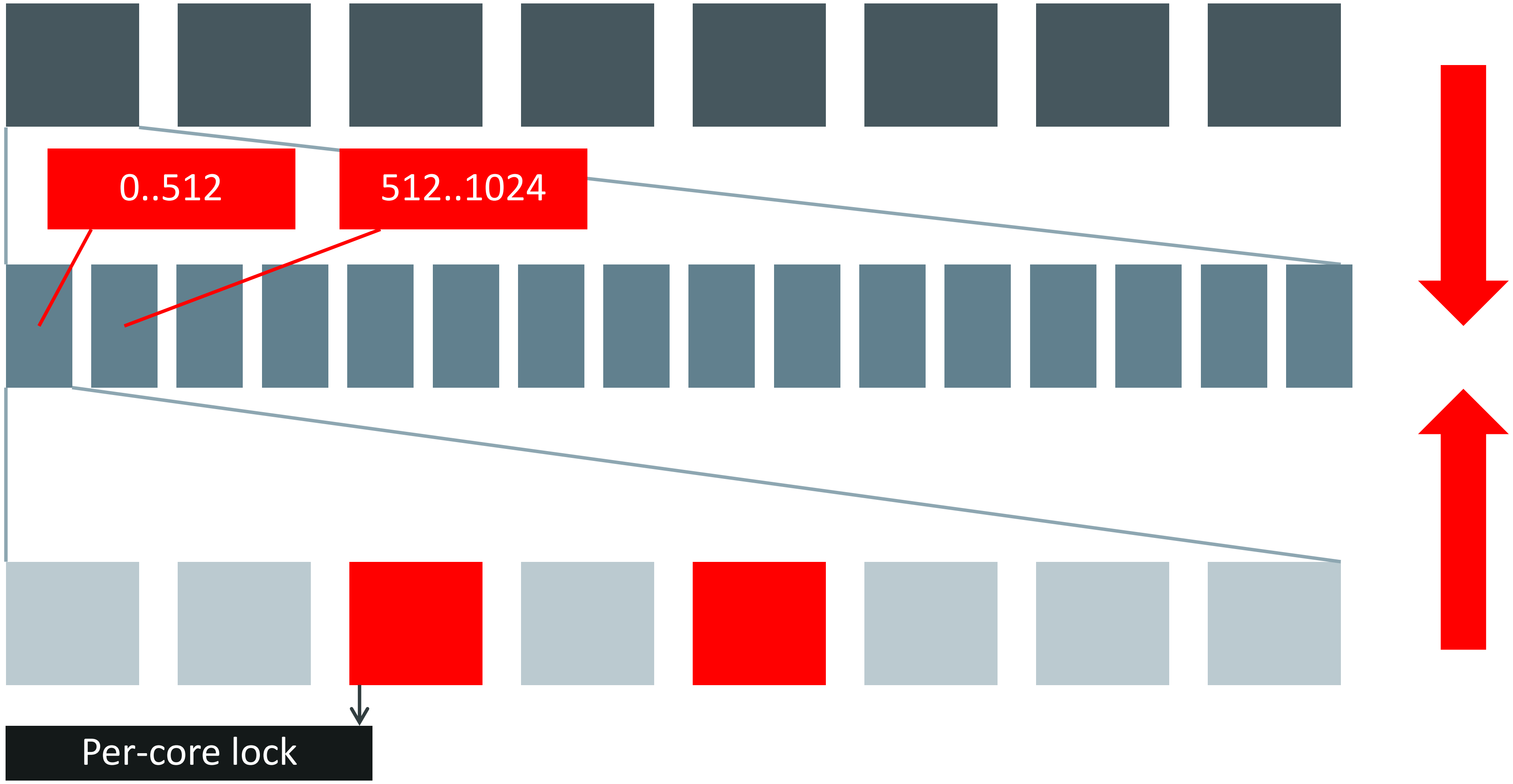
Distribute iterations at start of loop down to per-core counters

16 cores per socket

Aggregate requests upwards within a core

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



8 sockets

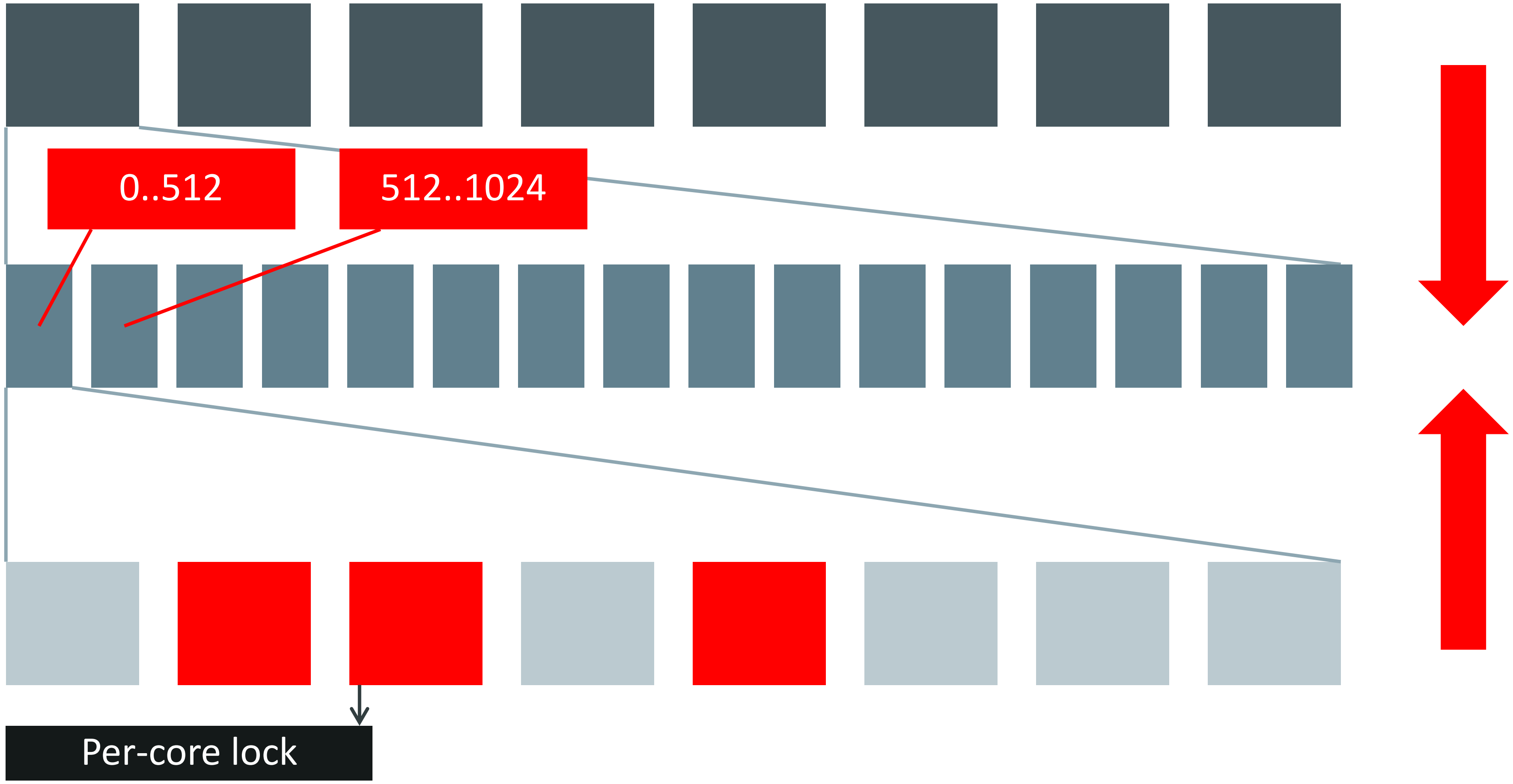
Distribute iterations at start of loop down to per-core counters

16 cores per socket

Aggregate requests upwards within a core

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



8 sockets

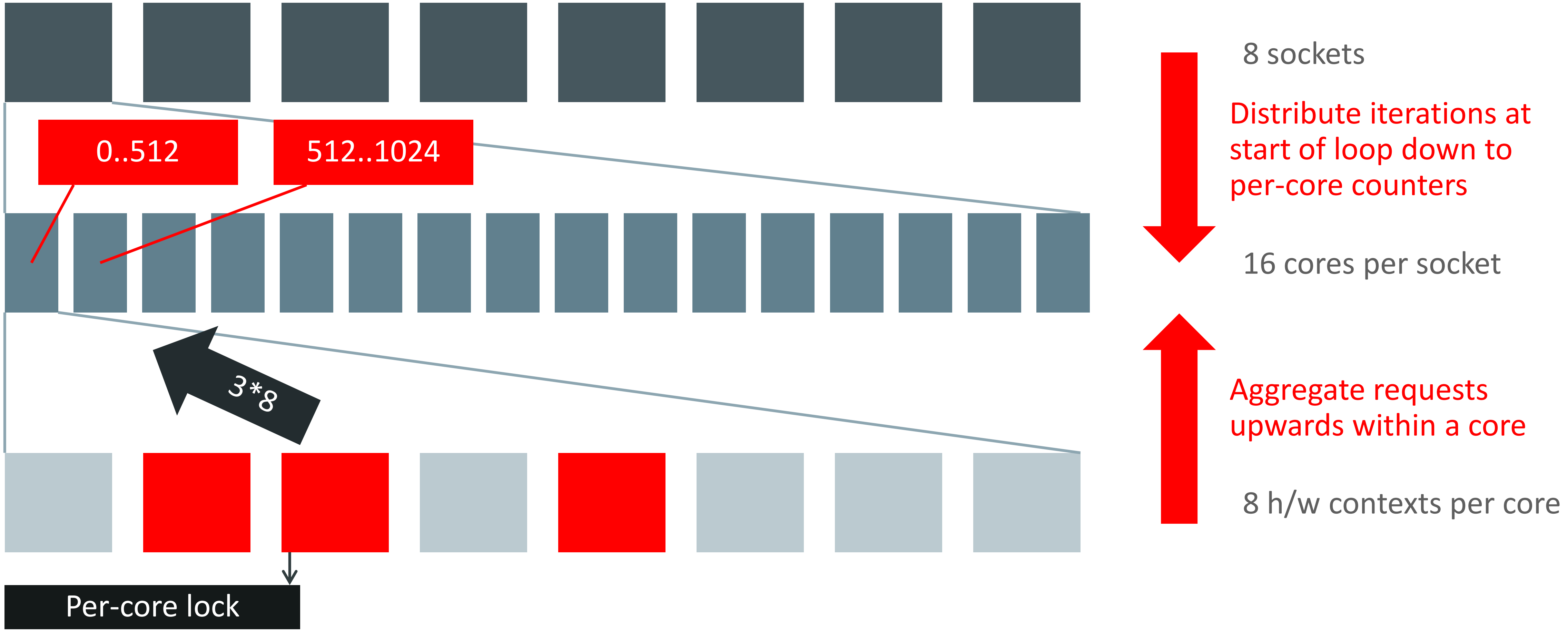
Distribute iterations at start of loop down to per-core counters

16 cores per socket

Aggregate requests upwards within a core

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



8 sockets

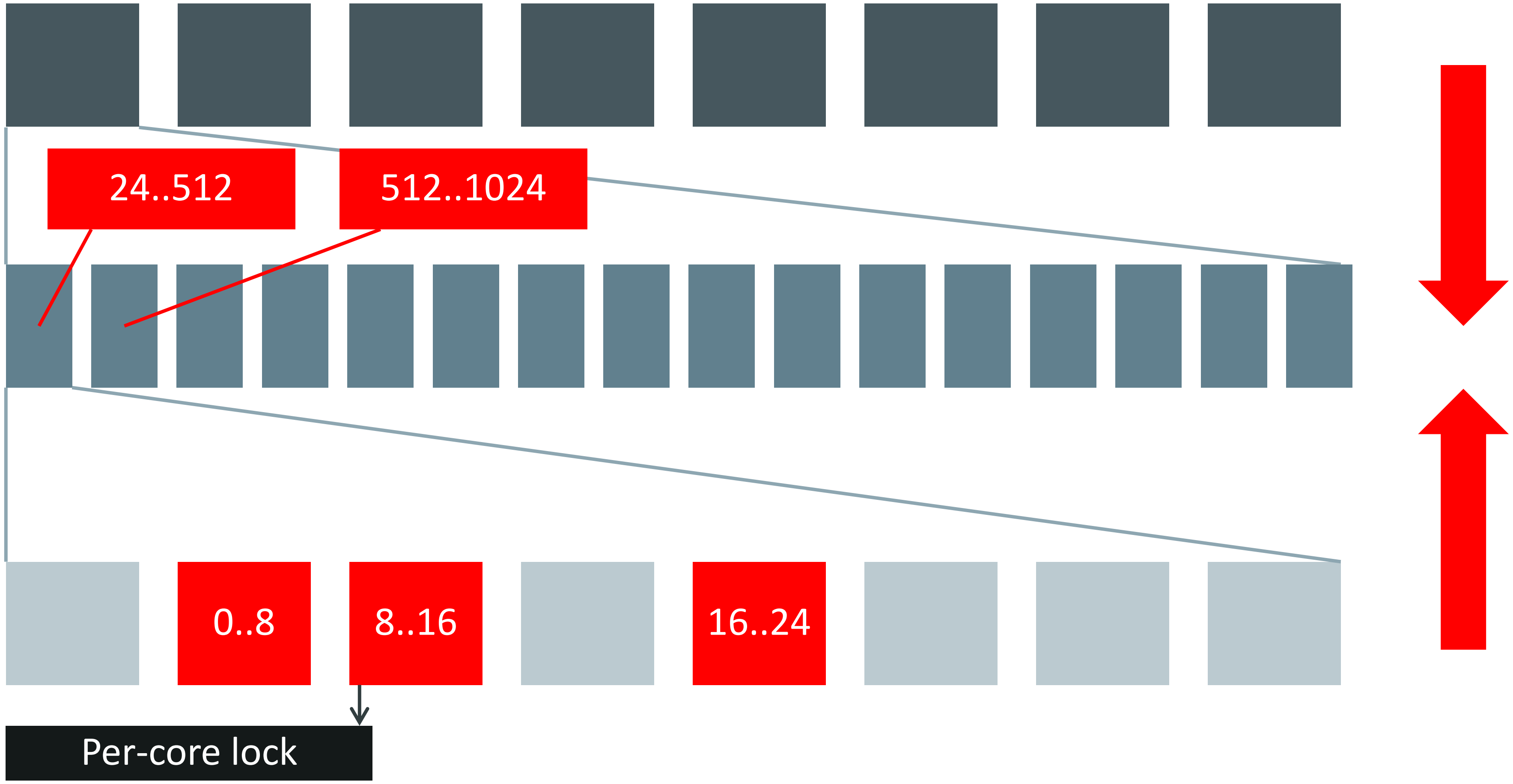
Distribute iterations at start of loop down to per-core counters

16 cores per socket

Aggregate requests upwards within a core

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



8 sockets

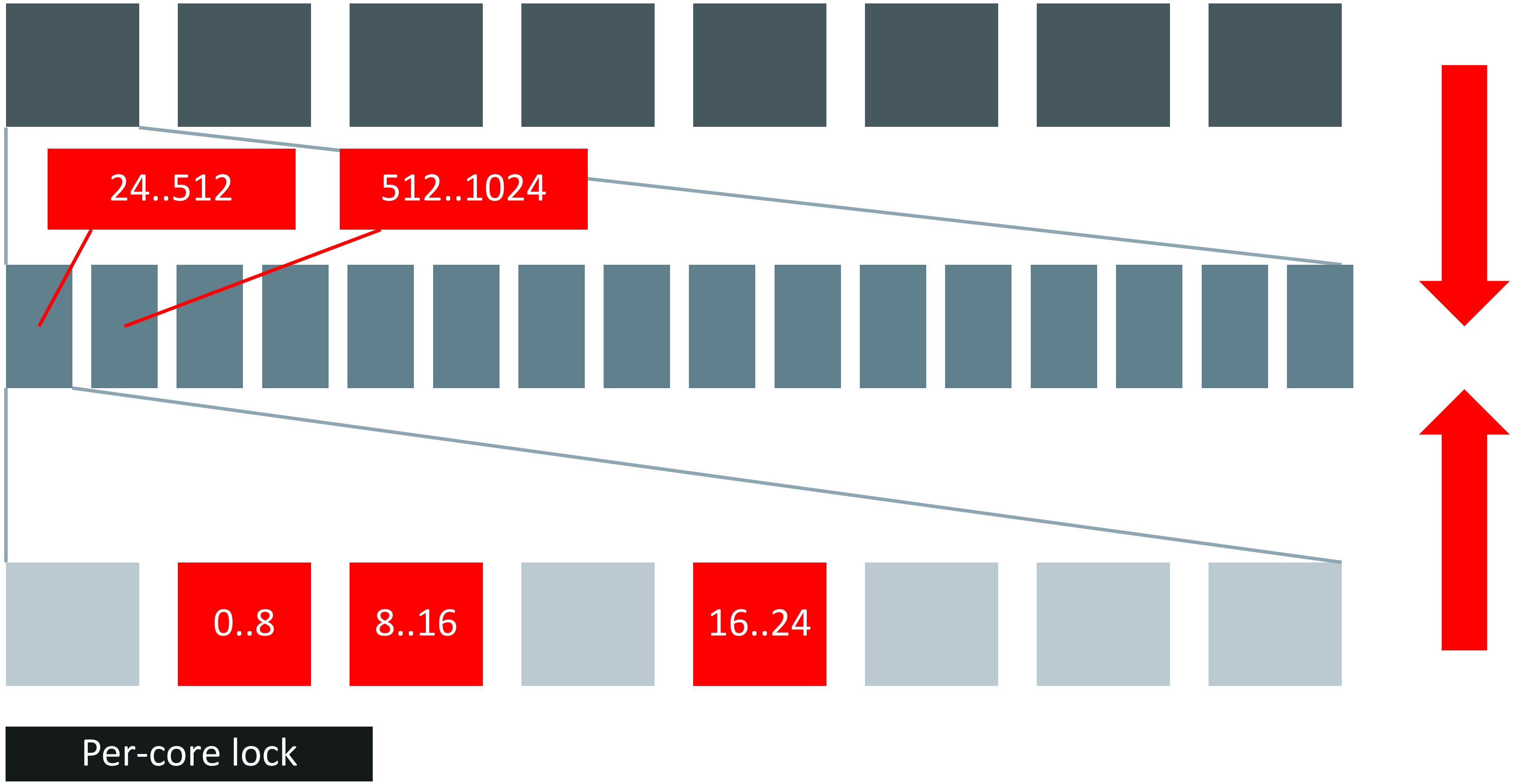
Distribute iterations at start of loop down to per-core counters

16 cores per socket

Aggregate requests upwards within a core

8 h/w contexts per core

Approach, consider a loop 0..65536, batch size 8



8 sockets

Distribute iterations at start of loop down to per-core counters

16 cores per socket

Aggregate requests upwards within a core

8 h/w contexts per core

Hierarchical distribution with request combining

- Combining implemented over flags in a single line in the shared L1 D$
- On TSO: no memory fences
- Synchronization remains core-local if work is evenly distributed
- Threads waiting for combining can use mwait

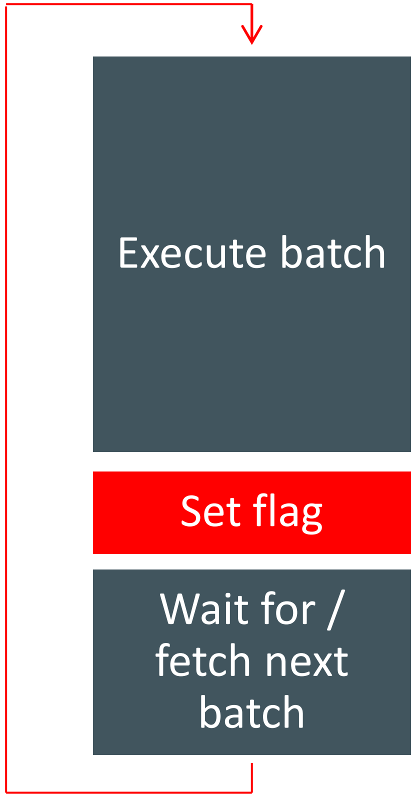
Techniques

1 Request combining

2 Asynchronous work requests

Asynchronous combining of requests

Synchronous

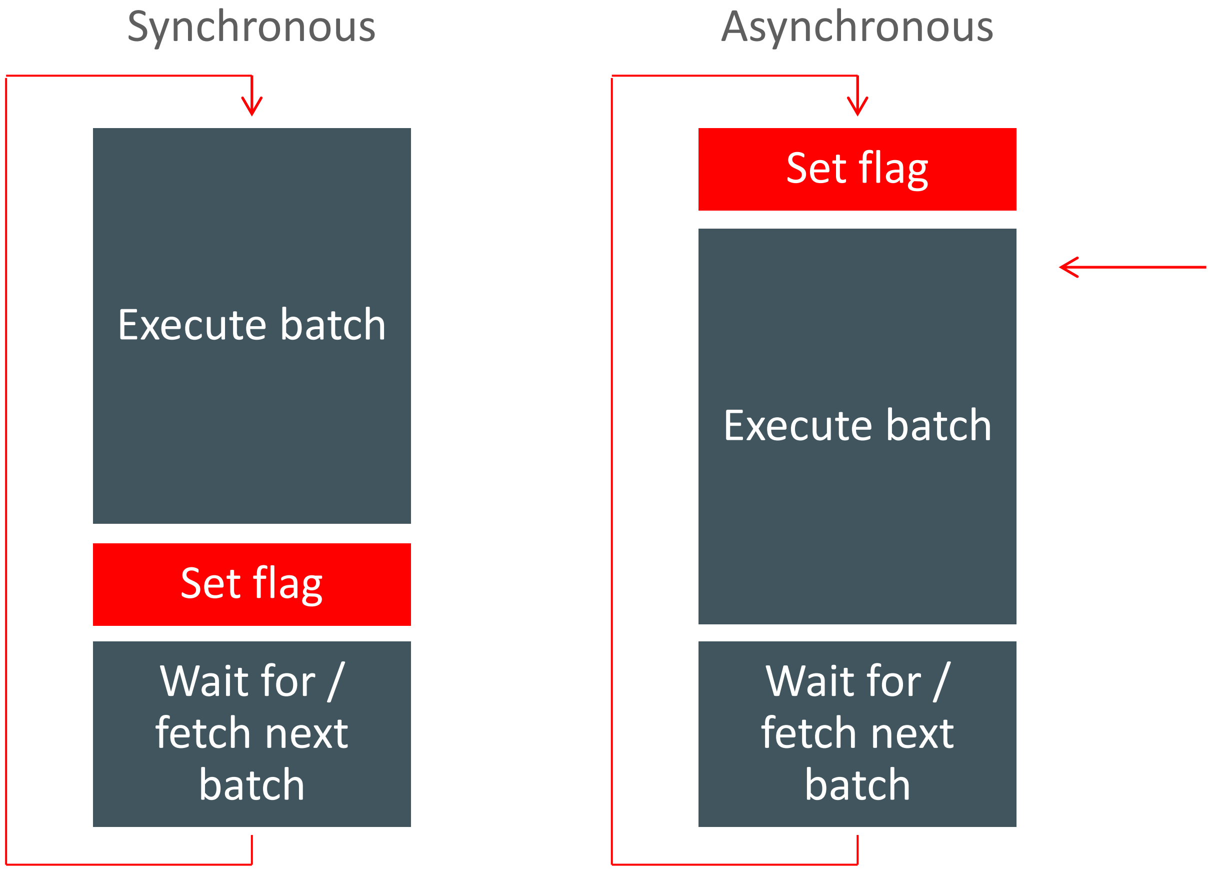


Execute batch

Set flag

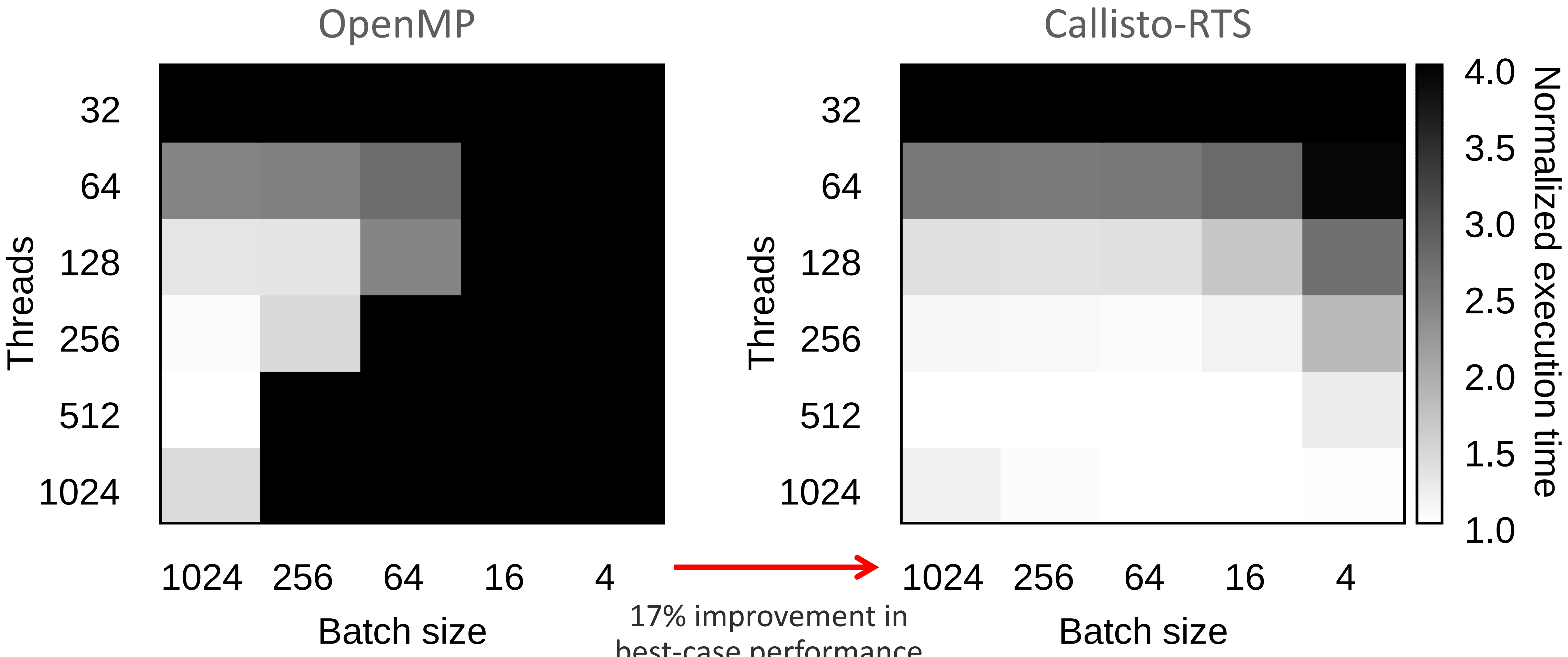
Wait for / fetch next batch

Asynchronous combining of requests



Intuition: the time taken to execute the current batch provides an opportunity for other cores to service our request without us needing to wait, and the number of requests batched together will be larger.

PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)

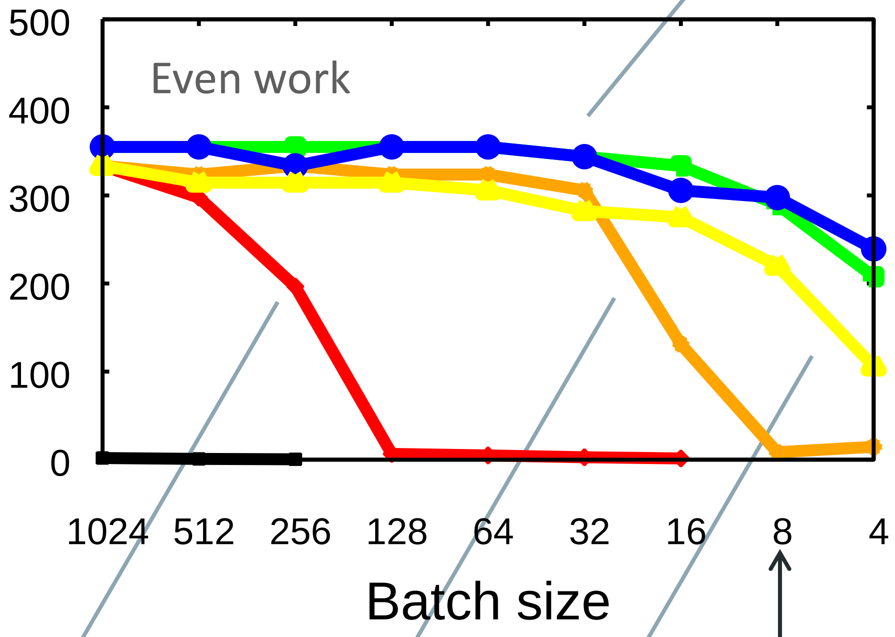


17% improvement in best-case performance

Microbenchmark results

SPARC T5-8, 1024 threads

Per-core + asynchronous combining (blue)
Per-core + synchronous combining (green)



(Approx
1k cycles)

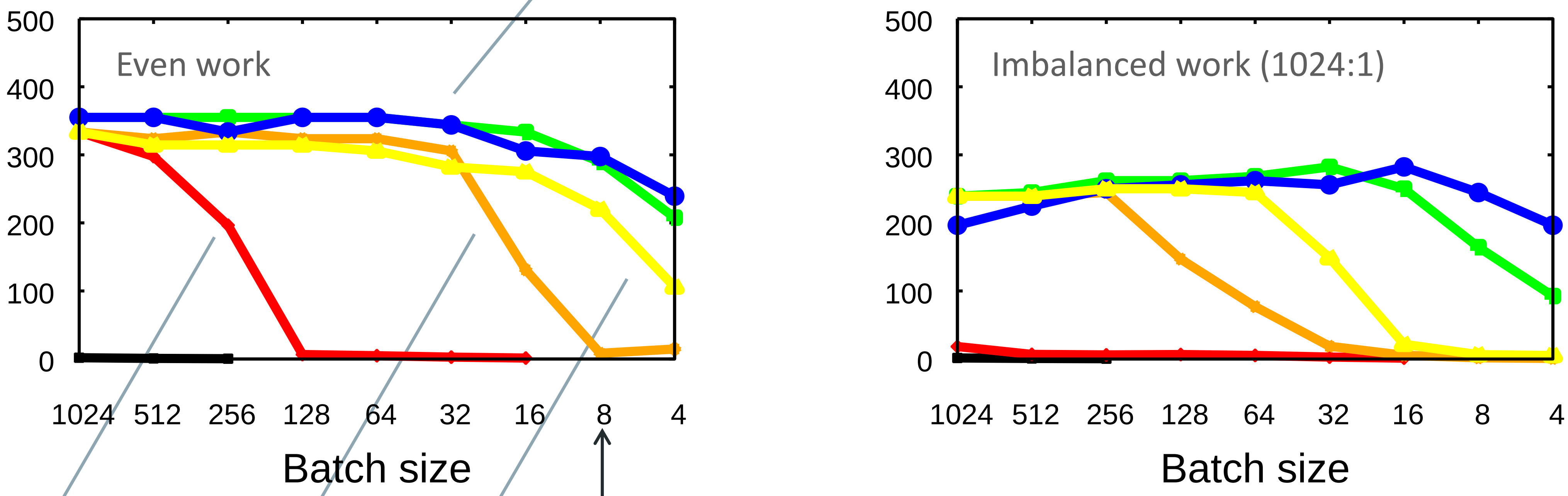
Per-socket counters

Per-core counters

Per-thread
counters

Microbenchmark results

SPARC T5-8, 1024 threads



Per-core + asynchronous combining (blue)
Per-core + synchronous combining (green)

Per-socket counters

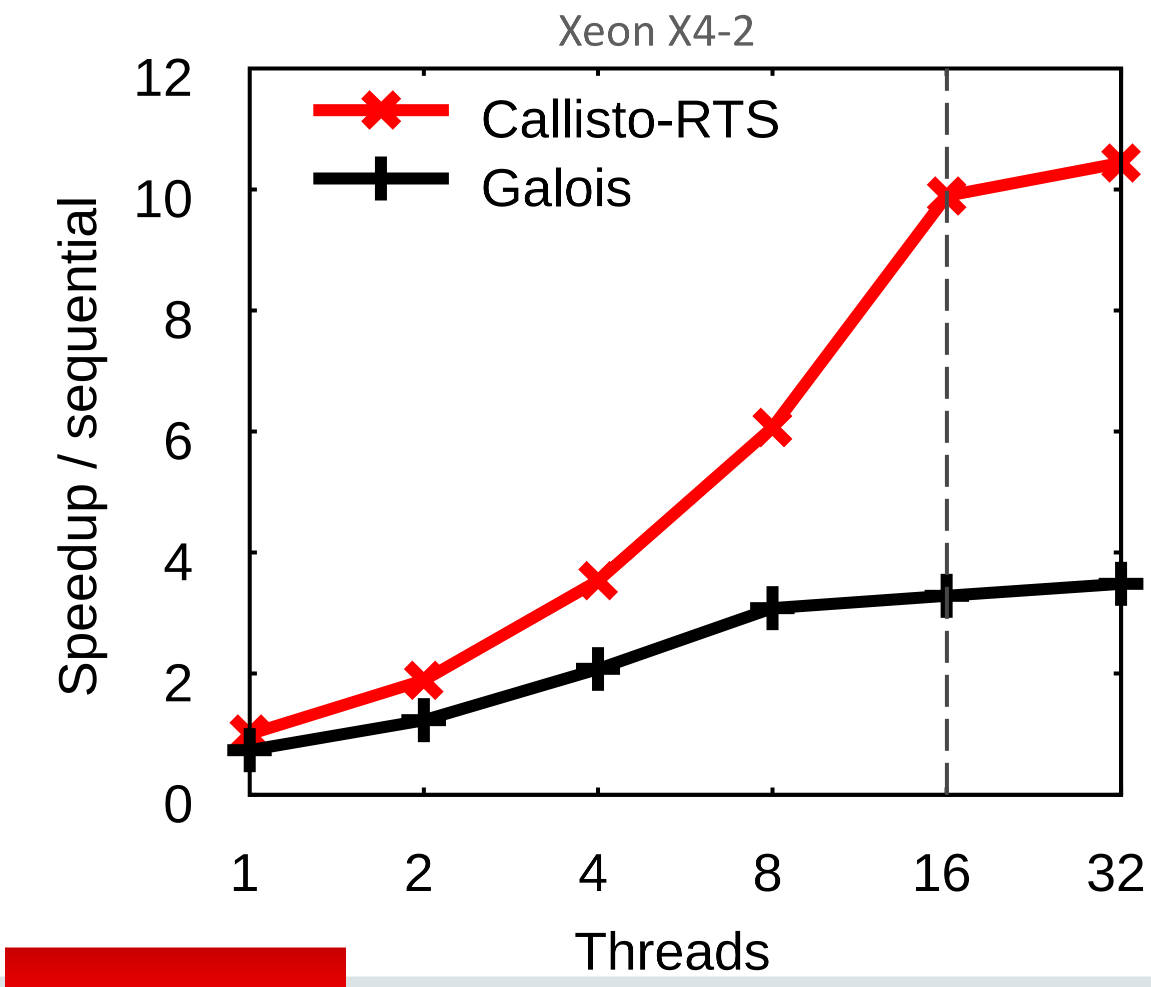
Per-core counters

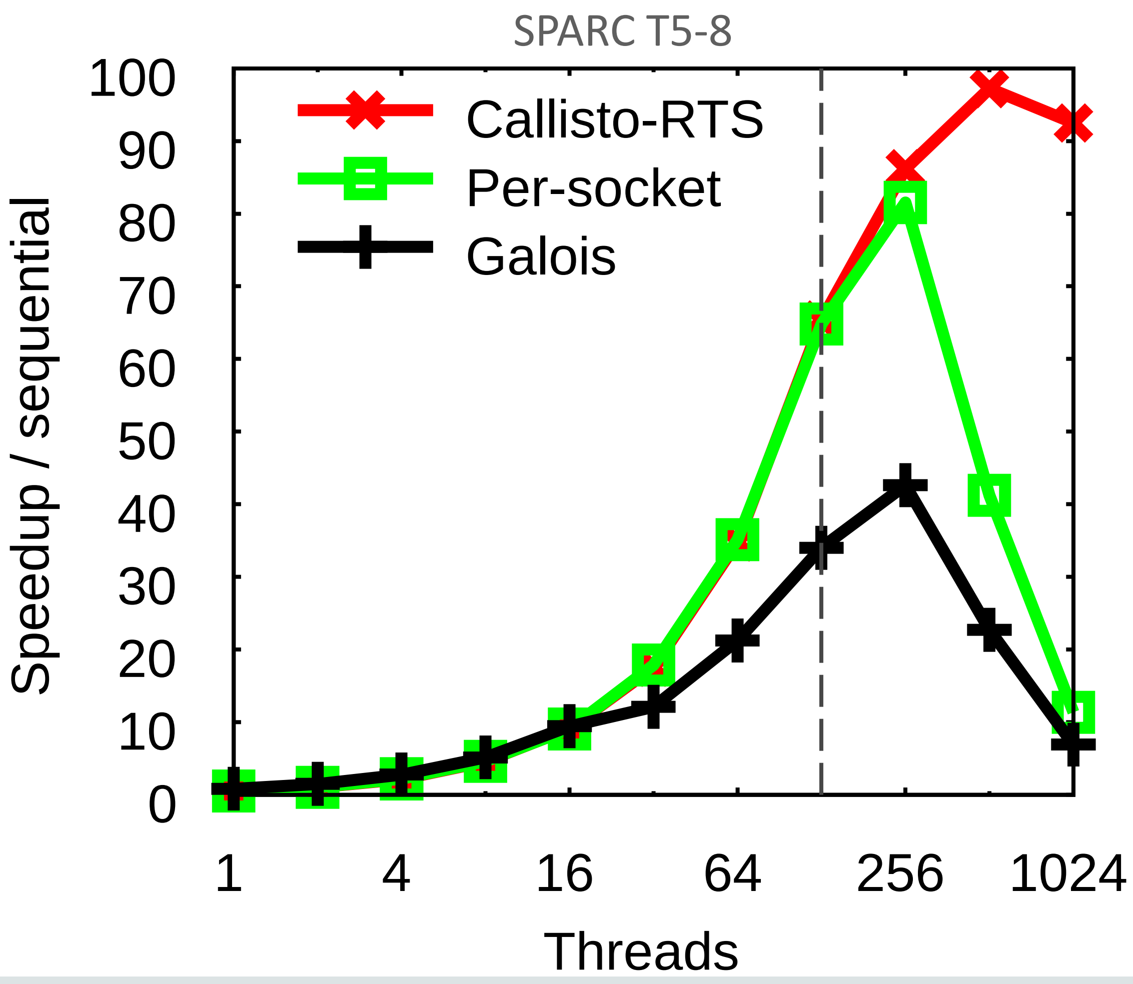
Per-thread counters

(Approx 1k cycles)

Comparison with Galois

SNAP LiveJournal data set





Nested loops

Nested loops

- Abundant parallelism, why use nesting?

Nested loops

- Abundant parallelism, why use nesting?
- Contention between iterations of an outer loop
- E.g., betweenness-centrality:
  - Iterate over vertices
  - BFS traversal from each vertex (plus additional work)

Nested loops

- Abundant parallelism, why use nesting?
- Contention between iterations of an outer loop
- E.g., betweenness-centrality:
  - Iterate over vertices
  - BFS traversal from each vertex (plus additional work)

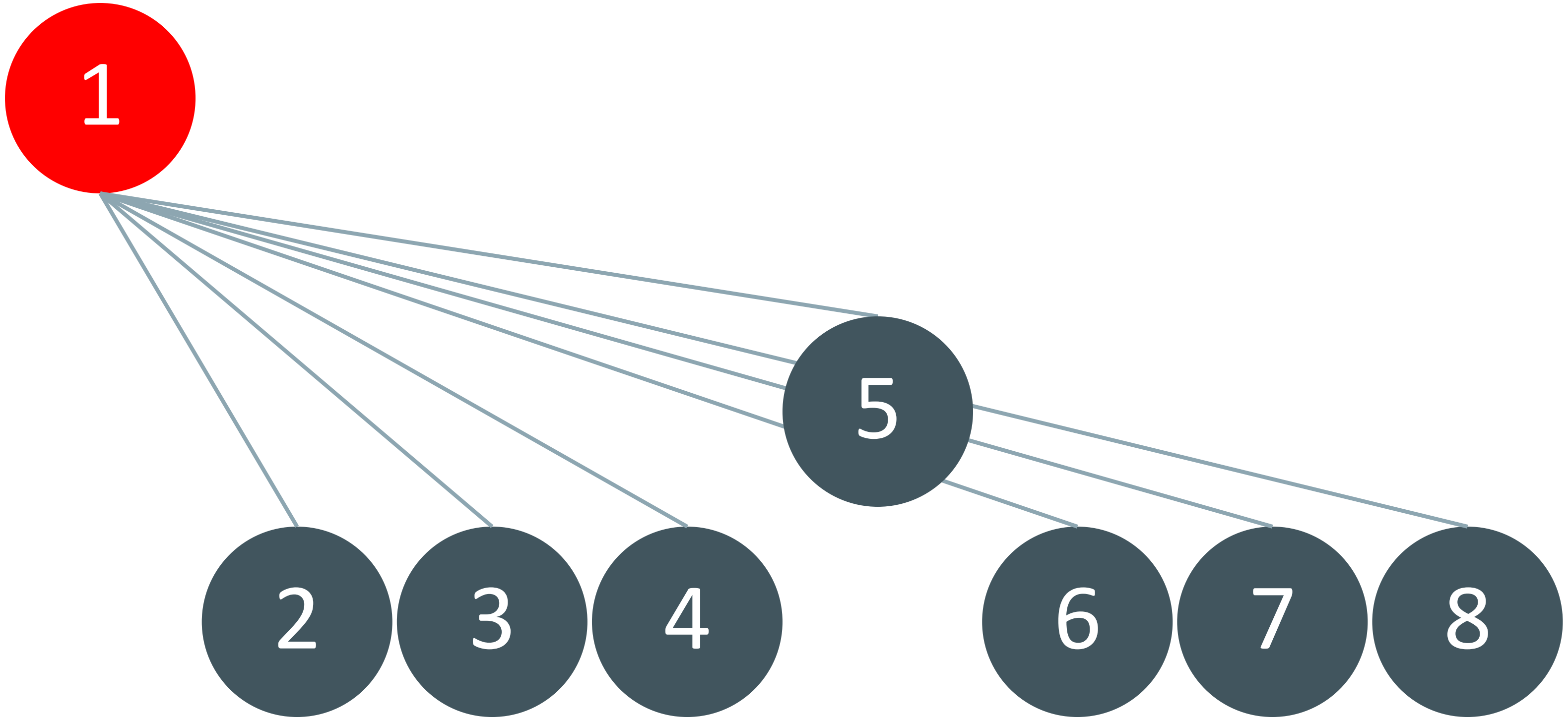
Better cache locality within each traversal
than between (unrelated) traversals

Nested loops

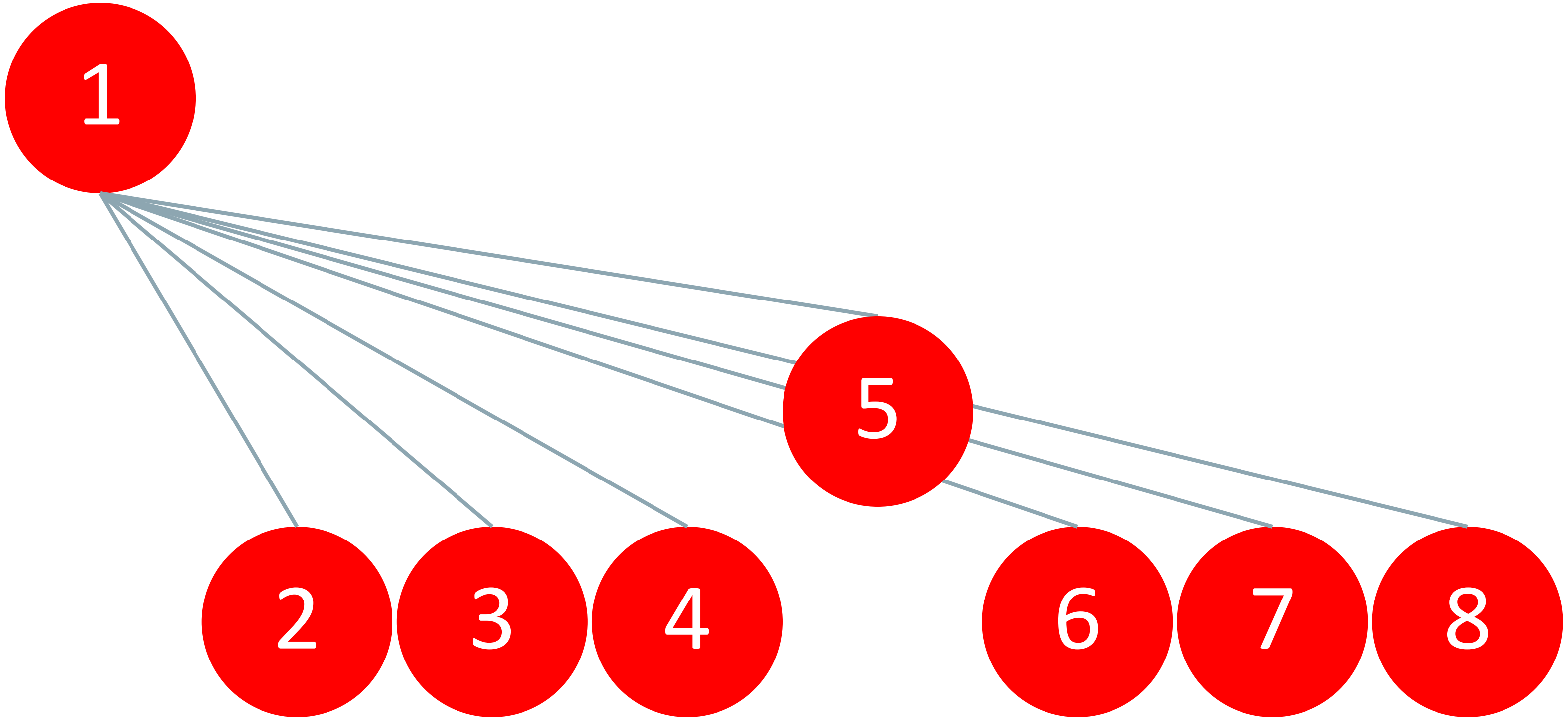
- Abundant parallelism, why use nesting?
- Contention between iterations of an outer loop
- E.g., betweenness-centrality:
  - Iterate over vertices
  - BFS traversal from each vertex (plus additional work)

Better cache locality within each traversal than between (unrelated) traversals → Run at most one of these per L2 D$

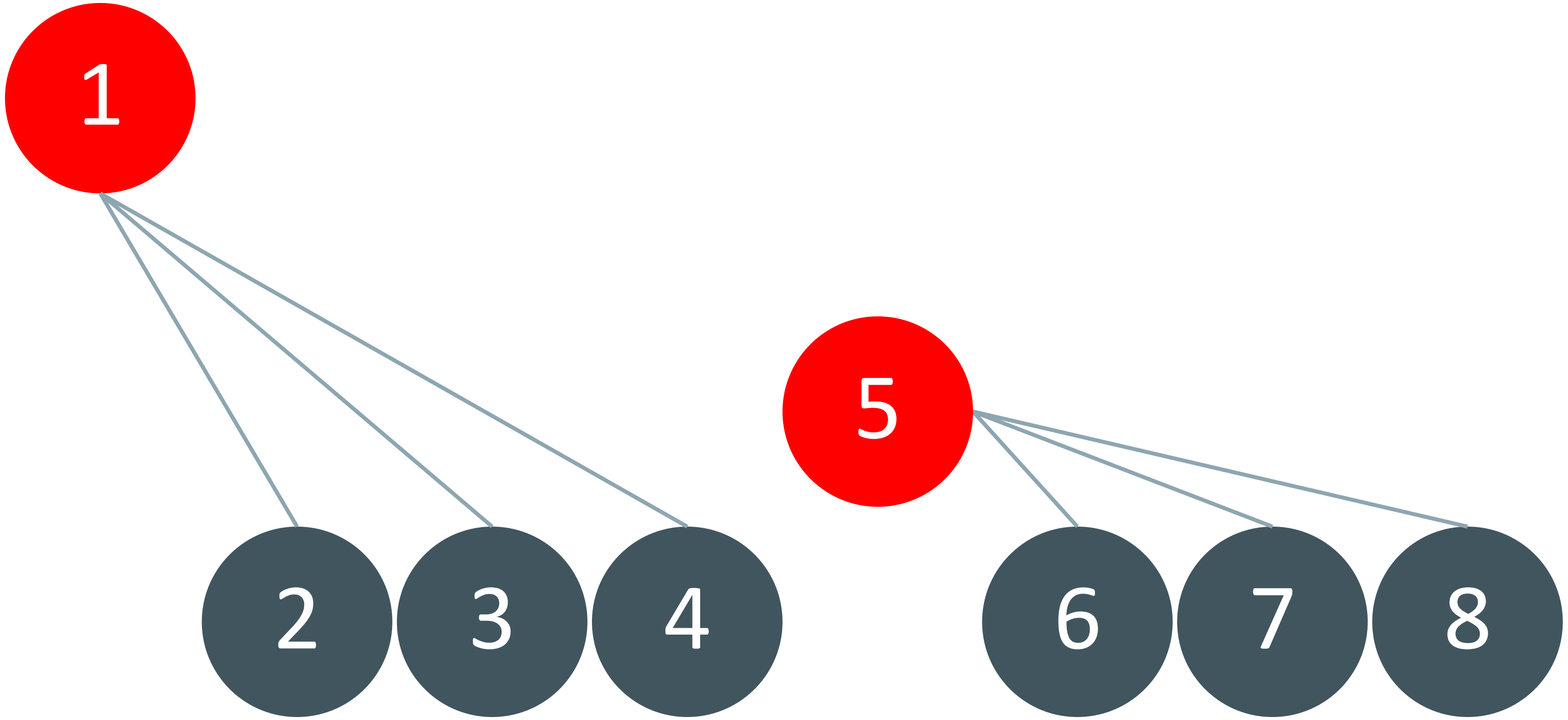
Nested loops



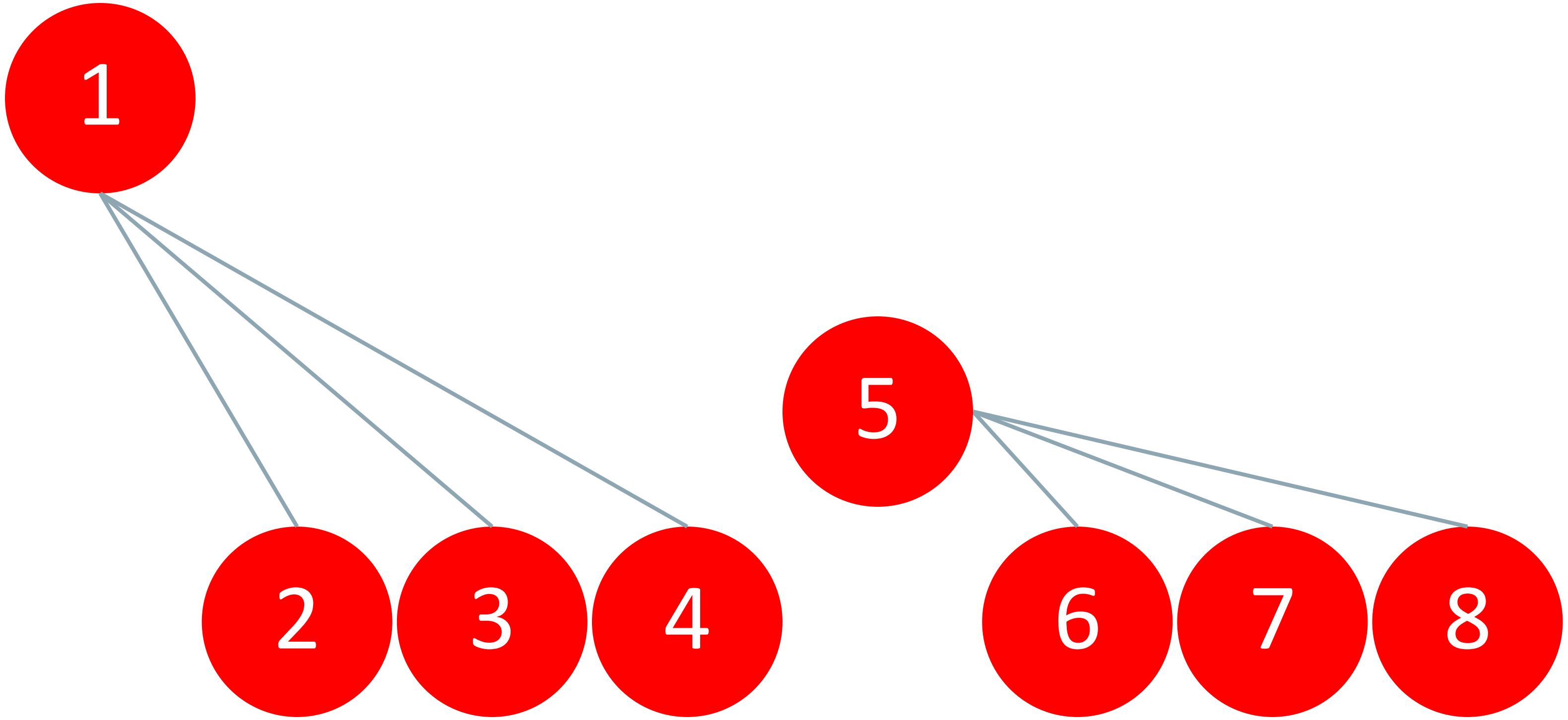
Nested loops: default, all threads participate



Nested loops: outer level – just 1+5 participate

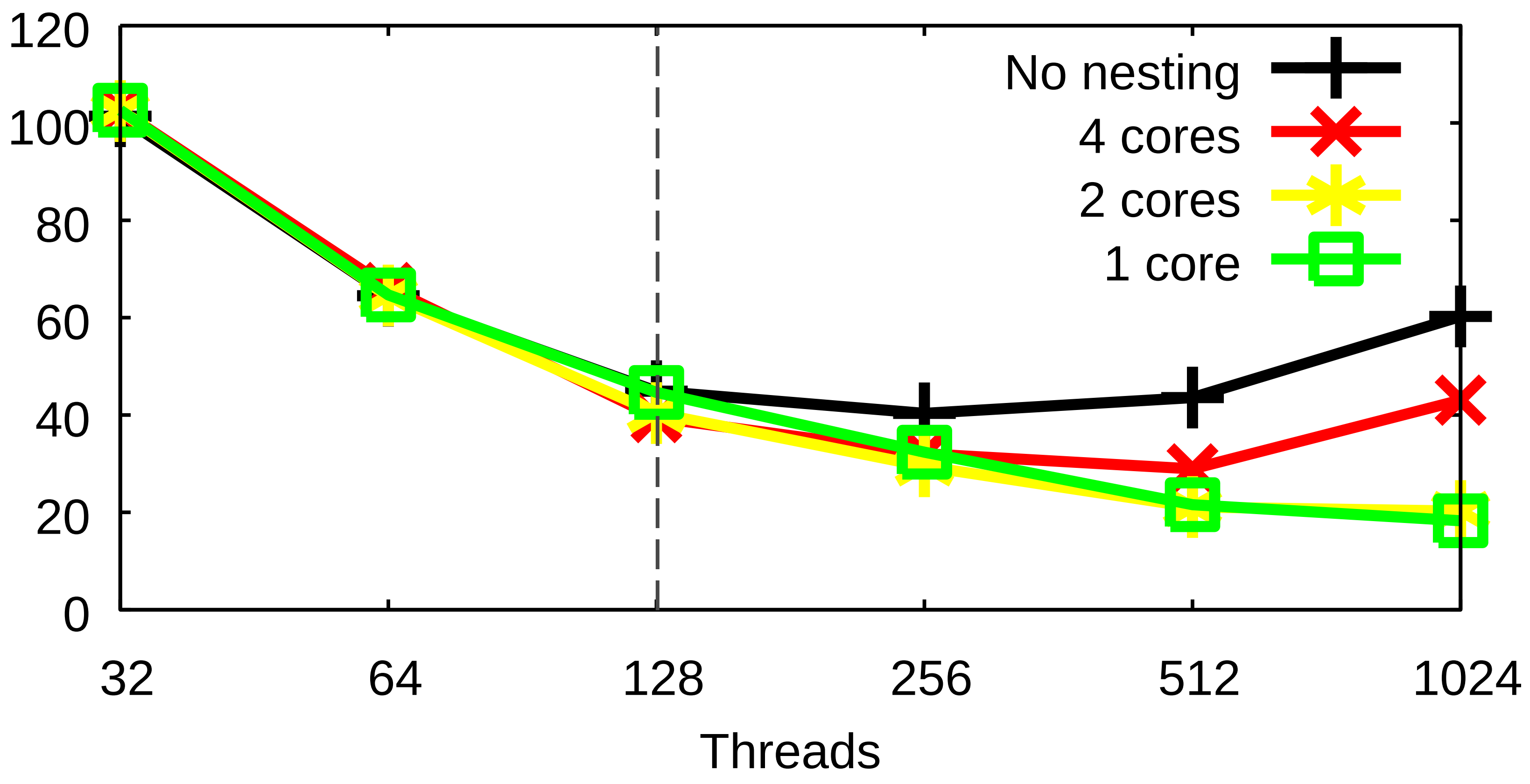


Nested loops: inner level –help respective leaders



Betweenness-centrality

SNAP Slashdot data set (82.1K nodes, 948K edges), T5-8



In-memory graph analytics

Domain specific languages

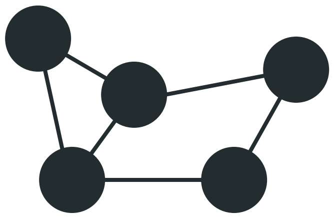
- Queries expressed in terms of graph concepts
- Tailor for different kinds of workload (e.g., sub-graph isomorphism)

Generated code

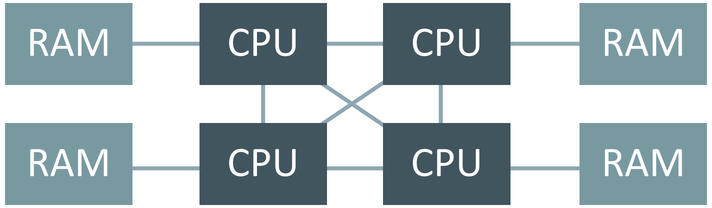
- Efficient in-memory data representations, e.g. compressed-sparse-row format
- Abundant parallelism

Runtime system

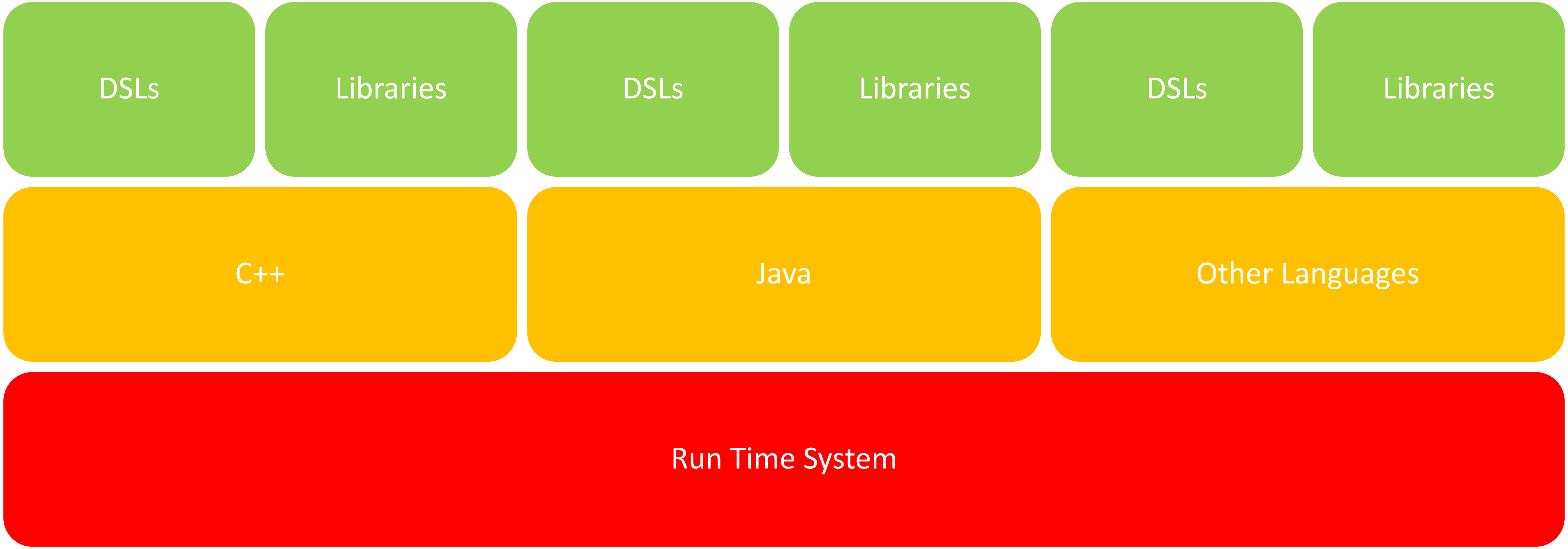
- Allocation of resources to a query
- Distribution of work and data within a machine



```
parallel_for<node_t>([&](node_t n) {
    …
});
```



RTS use cases



DSLs

Libraries

DSLs

Libraries

DSLs

Libraries

C++

Java

Other Languages

Run Time System

Future work

- Continuing development of the programming model
- Control over data placement as well as threads
  - Initial examples from graph workloads generally have random accesses: spread data and threads widely in the machine
  - (See “Shoal”, USENIX ATC 2015)
- Interactions between multiple parallel workloads
  - OS/runtime system interaction (ref our prior work at EuroSys 2014)
  - Placement in the machine
  - Control over degree of parallelism

Integrated Cloud

Applications & Platform Services

ORACLE®