Callisto-RTS: Fine-Grain Parallel Loops

Tim Harris, Oracle Labs
Stefan Kaestle, ETH Zurich

7 July 2015

Setting: parallel loops on shared-memory machines

```
for (uint64_t node = 0; node < G.num_nodes(); node++) {
    double val = 0.0;
    for (edge_t w_idx = G.r_begin[node];
          w_idx < G.r_begin[node+1];
          w_idx ++)  {
       node_t w = G.r_node_idx [w_idx];
       val += G_pg_rank[w] / (G.begin[w+1] - G.begin[w]);
    }
    G_pg_rank_nxt[node] = (1 - d) / N + d * val;
}
```

ORACLE

Setting: parallel loops on shared-memory machines

```
parallel_for<uint64_t>(0, G.num_nodes(),
  [&](uint64_t node) {
    double val = 0.0;
    for (edge_t w_idx = G.r_begin[node];
          w_idx < G.r_begin[node+1];
          w_idx ++)  {
       node_t w = G.r_node_idx [w_idx];
       val += G_pg_rank[w] / (G.begin[w+1] - G.begin[w]);
    }
    G_pg_rank_nxt[node] = (1 - d) / N + d * val;
  });
```

ORACLE

Setting: parallel loops on shared-memory machines

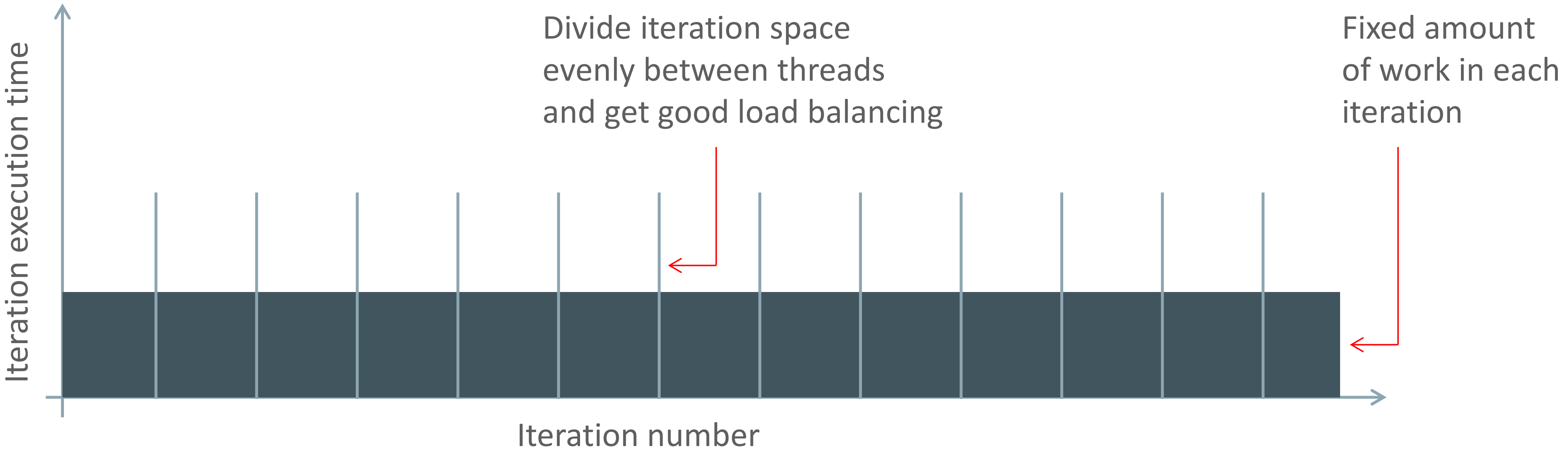
```
parallel_for<uint64_t>(0, G.num_nodes(),
  [&](uint64_t node) {
    double val = 0.0;
    for (edge_t w_idx = G.r_begin[node];
          w_idx < G.r_begin[node+1];
          w_idx ++)  {
      node_t w = G.r_node_idx [w_idx];
      val += G_pg_rank[w] / (G.begin[w+1] - G.begin[w]);
    }
    G_pg_rank_nxt[node] = (1 - d) / N + d * val;
});
```

Loop index type and bounds

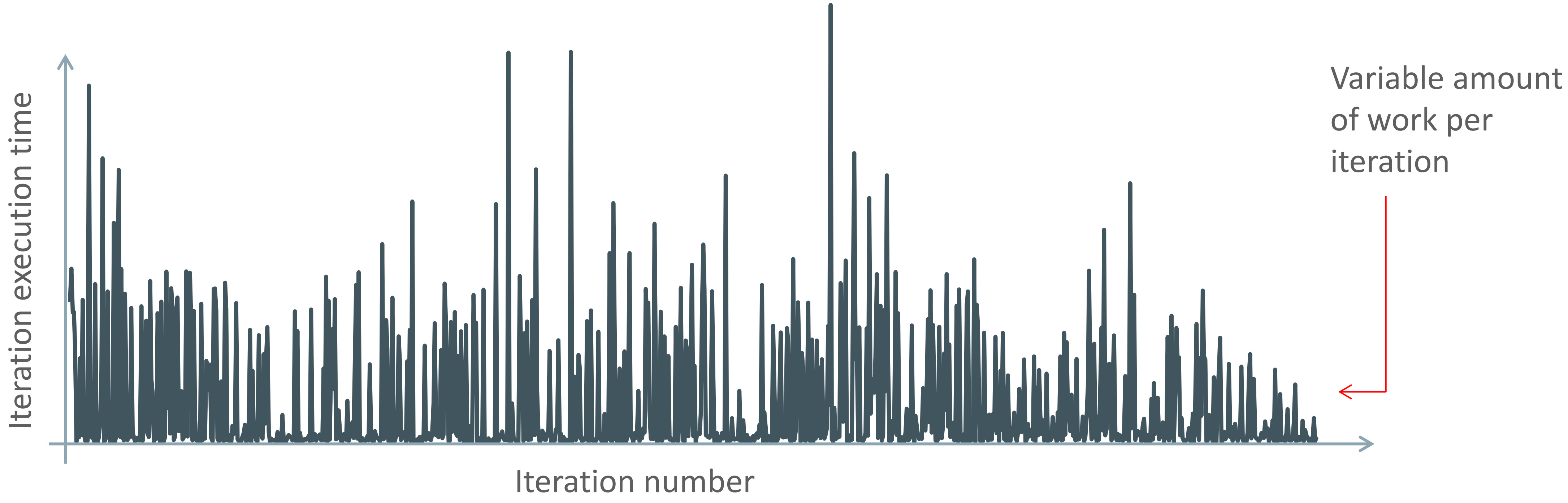
Loop body (C++ lambda)

Batch size / load imbalance trade-off

Batch size / load imbalance trade-off



Batch size / load imbalance trade-off



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

Batch size / load imbalance trade-off

Divide into large batches

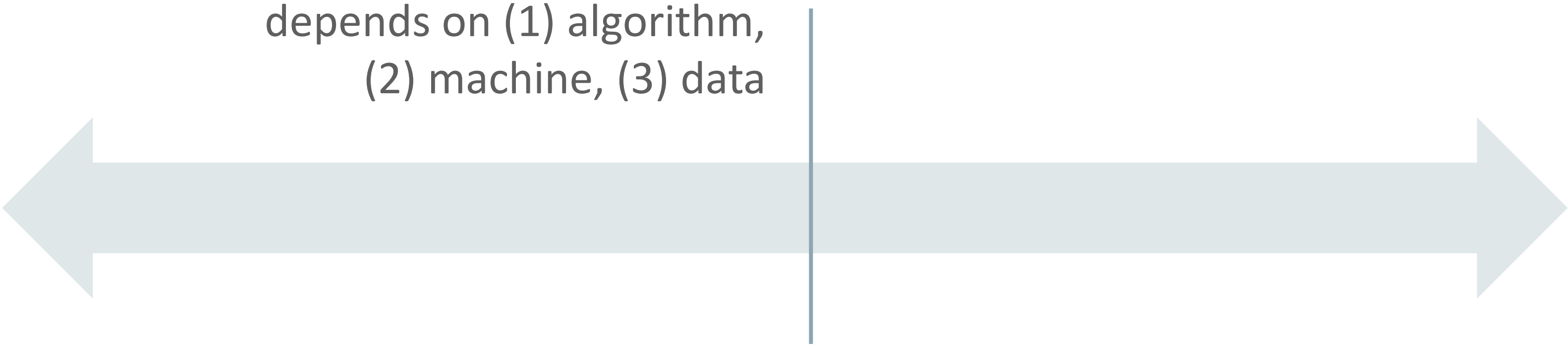
Reduce contention distributing work
Risk load imbalance

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

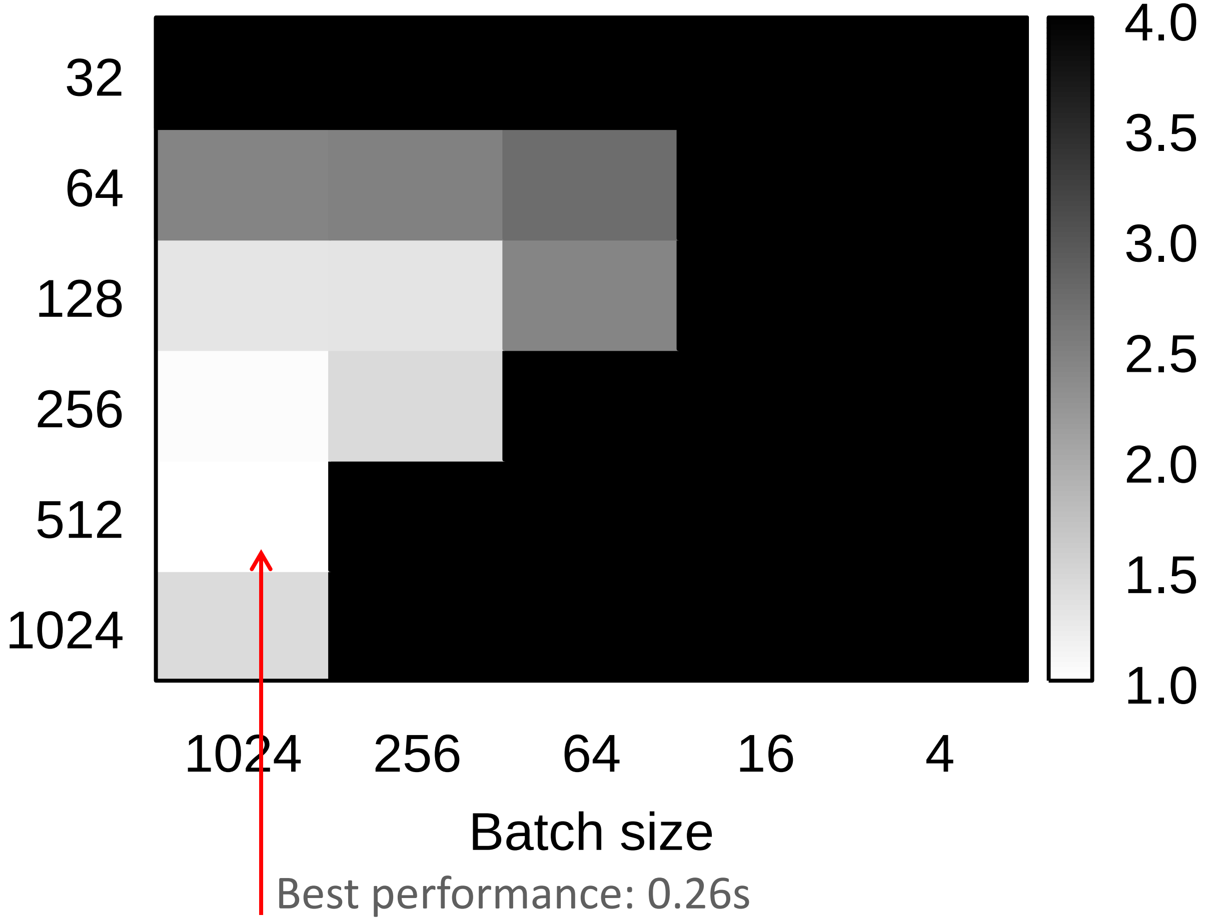
Divide into small batches

Increase contention distributing work
Achieve better load balance

ORACLE

Example performance

OpenMP static & dynamic loops



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

PageRank
SNAP LiveJournal data set

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

Overview

1. Request combining
2. Asynchronous work requests
3. Non-work-conserving nested loops
4. Results

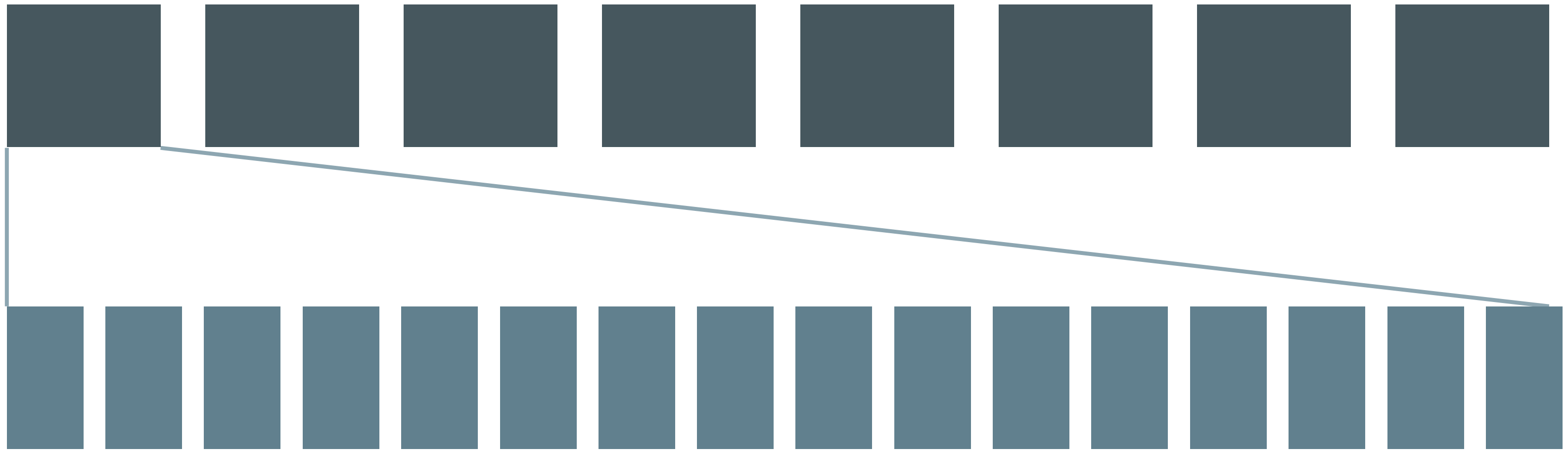
Overview

1 Request combining
2 Asynchronous work requests
3 Non-work-conserving nested loops
4 Results

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

8 sockets

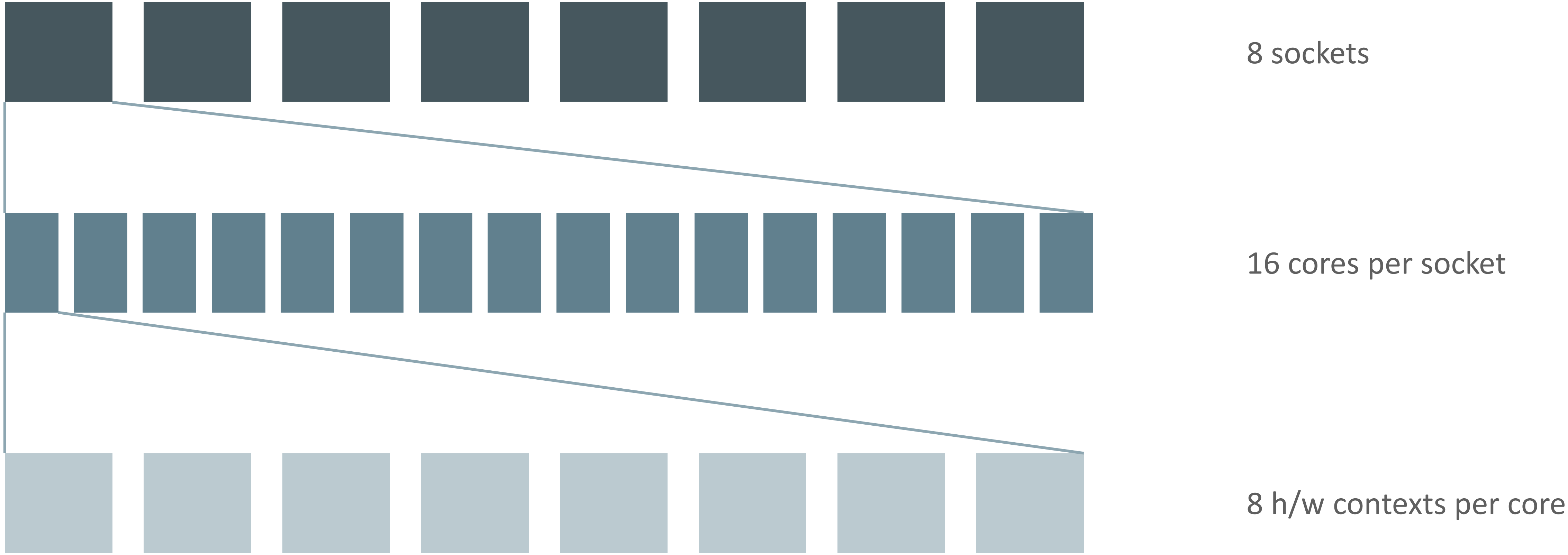
Approach, consider a loop 0..65536, batch size 8



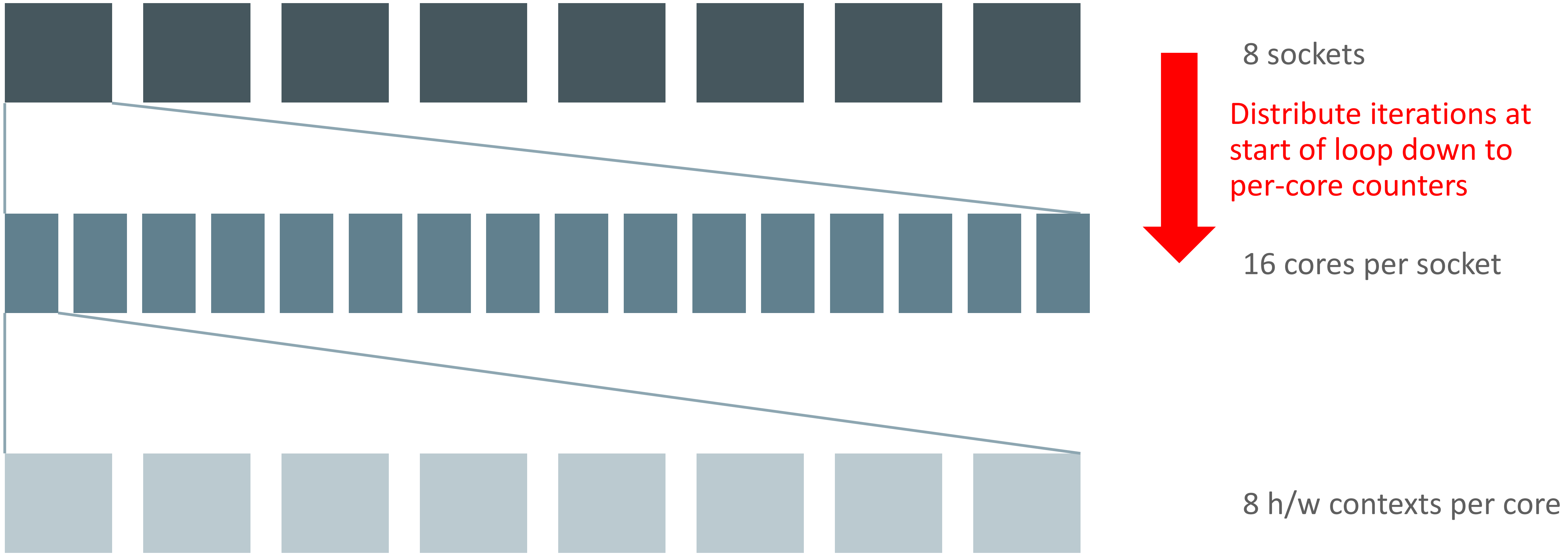
8 sockets

16 cores per socket

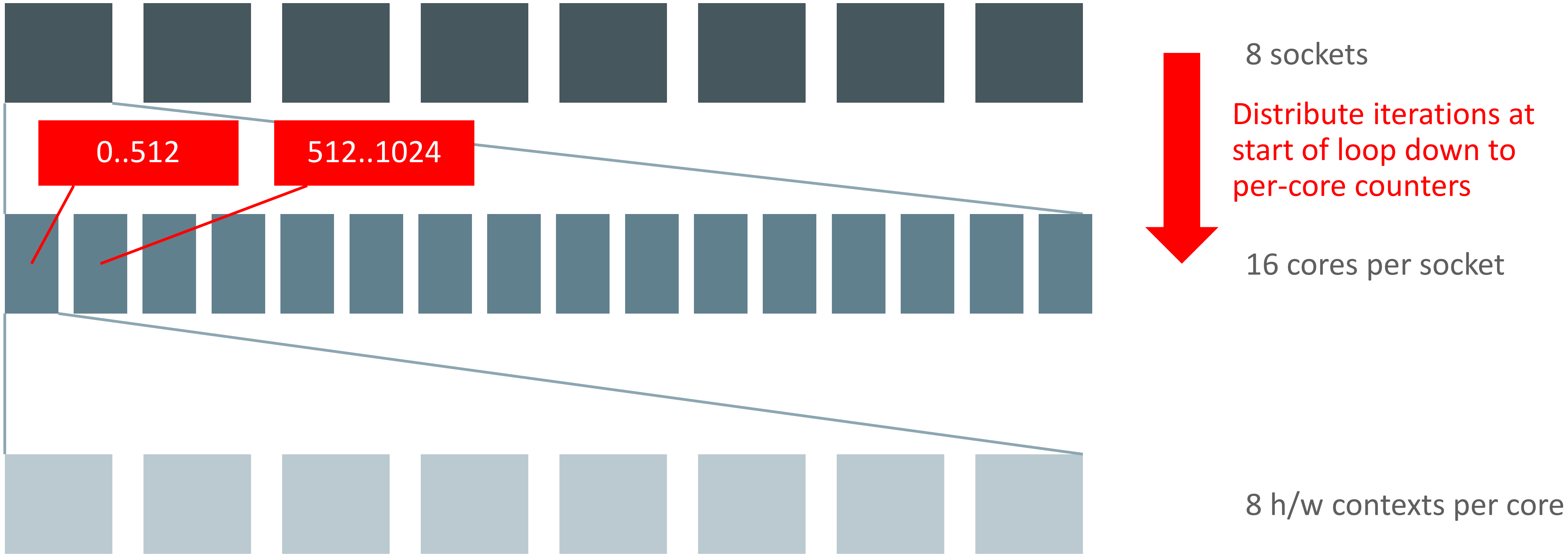
Approach, consider a loop 0..65536, batch size 8



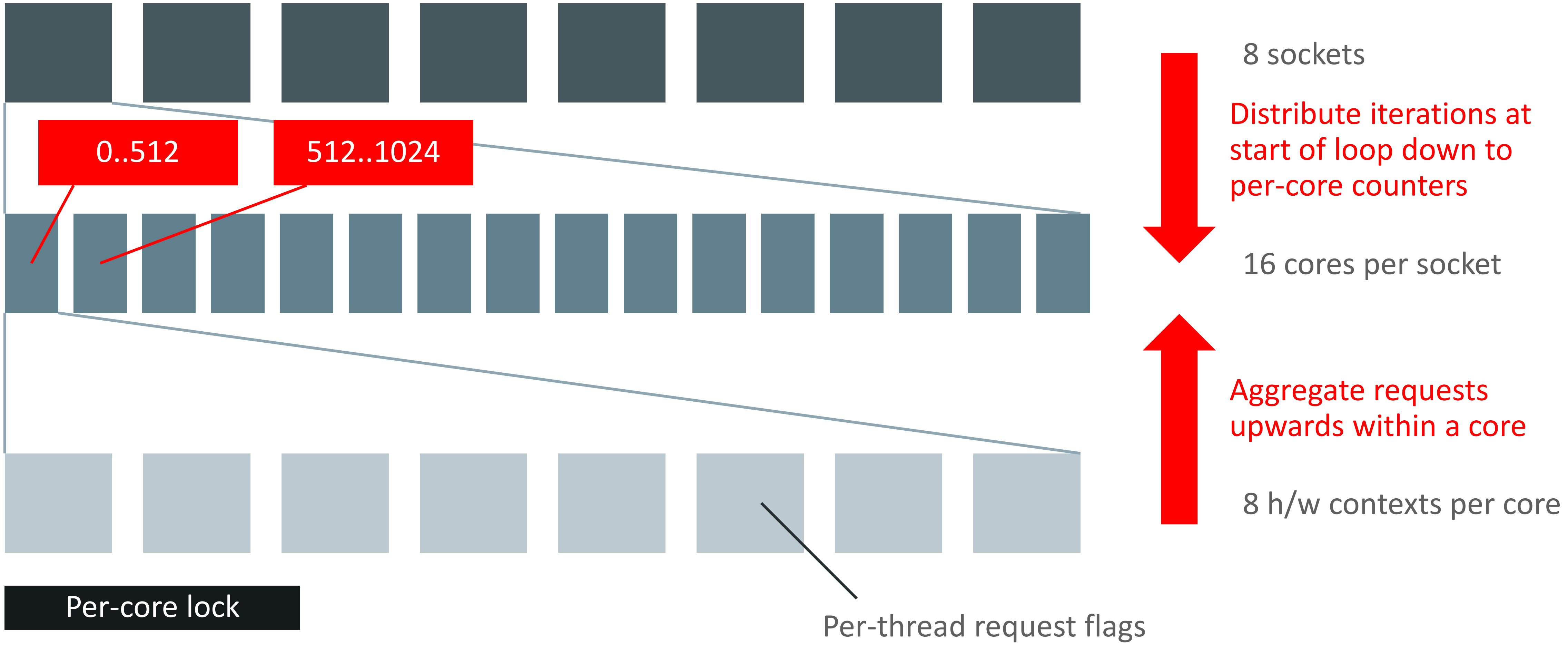
Approach, consider a loop 0..65536, batch size 8



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

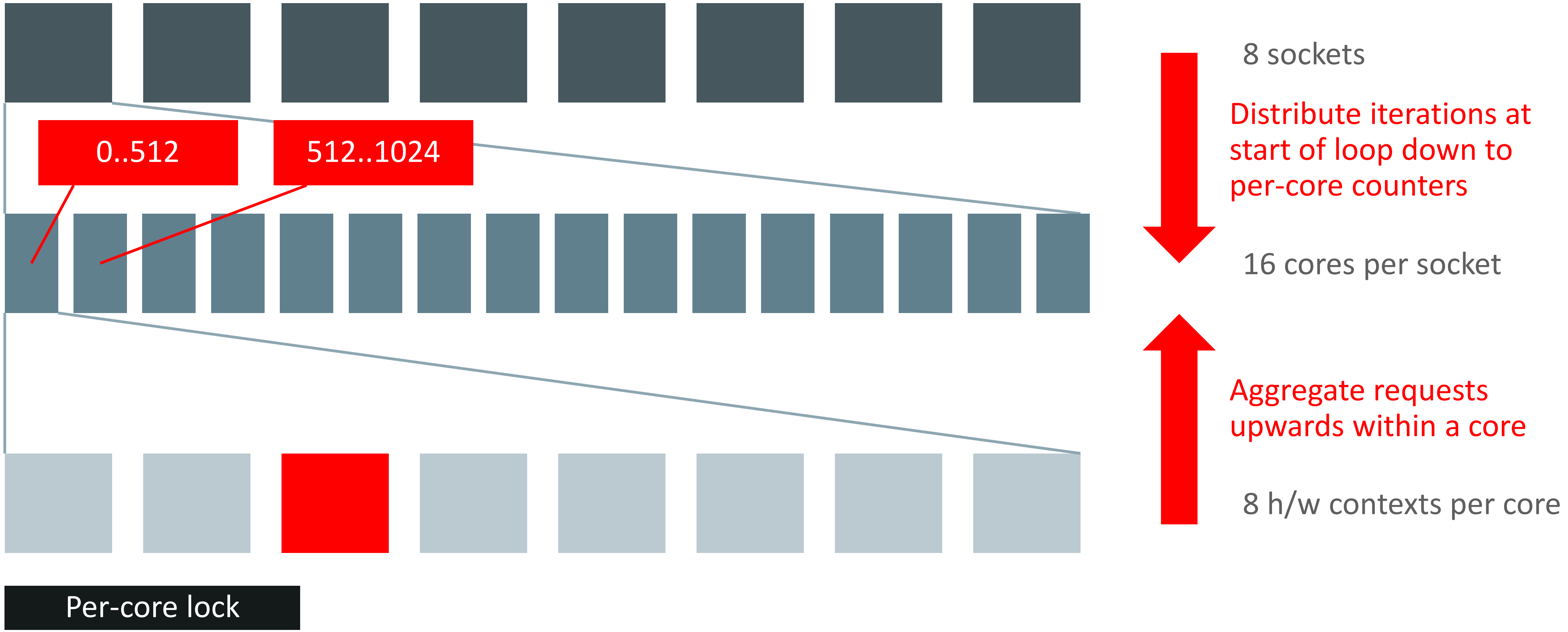


Approach, consider a loop 0..65536, batch size 8

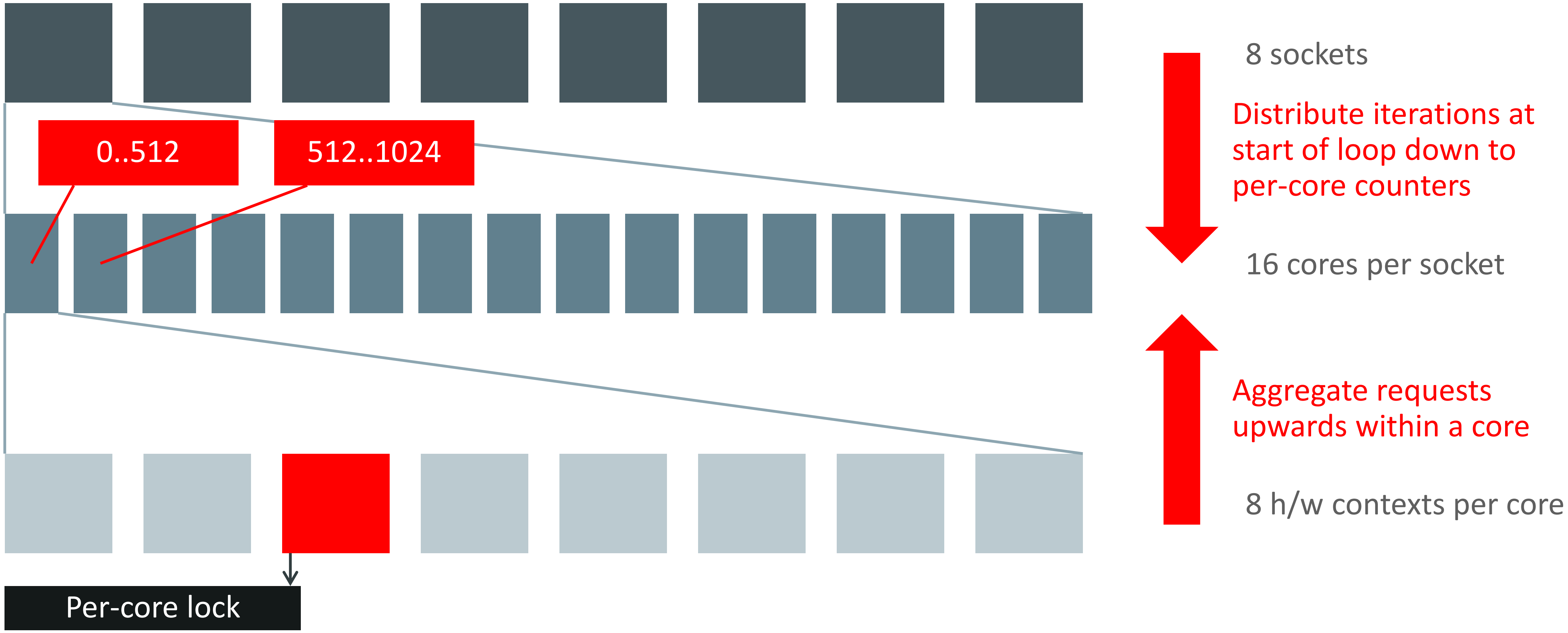


ORACLE

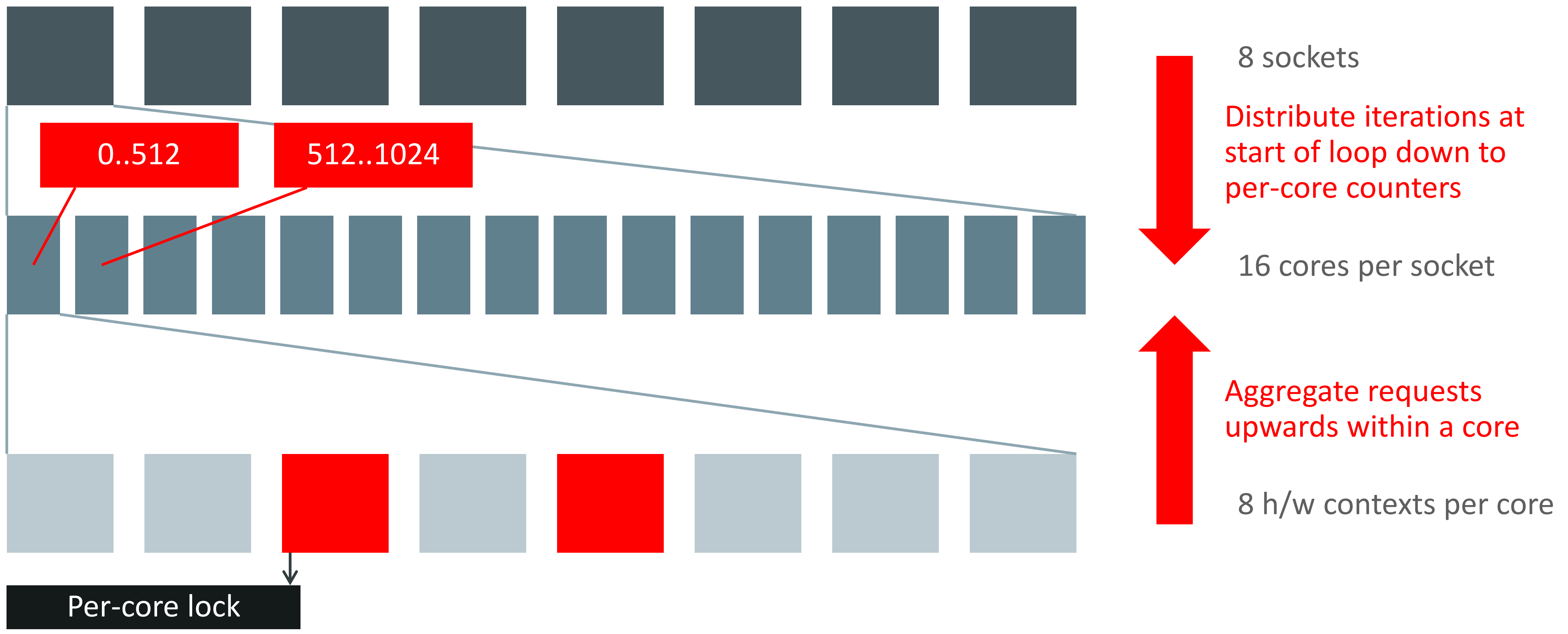
Approach, consider a loop 0..65536, batch size 8



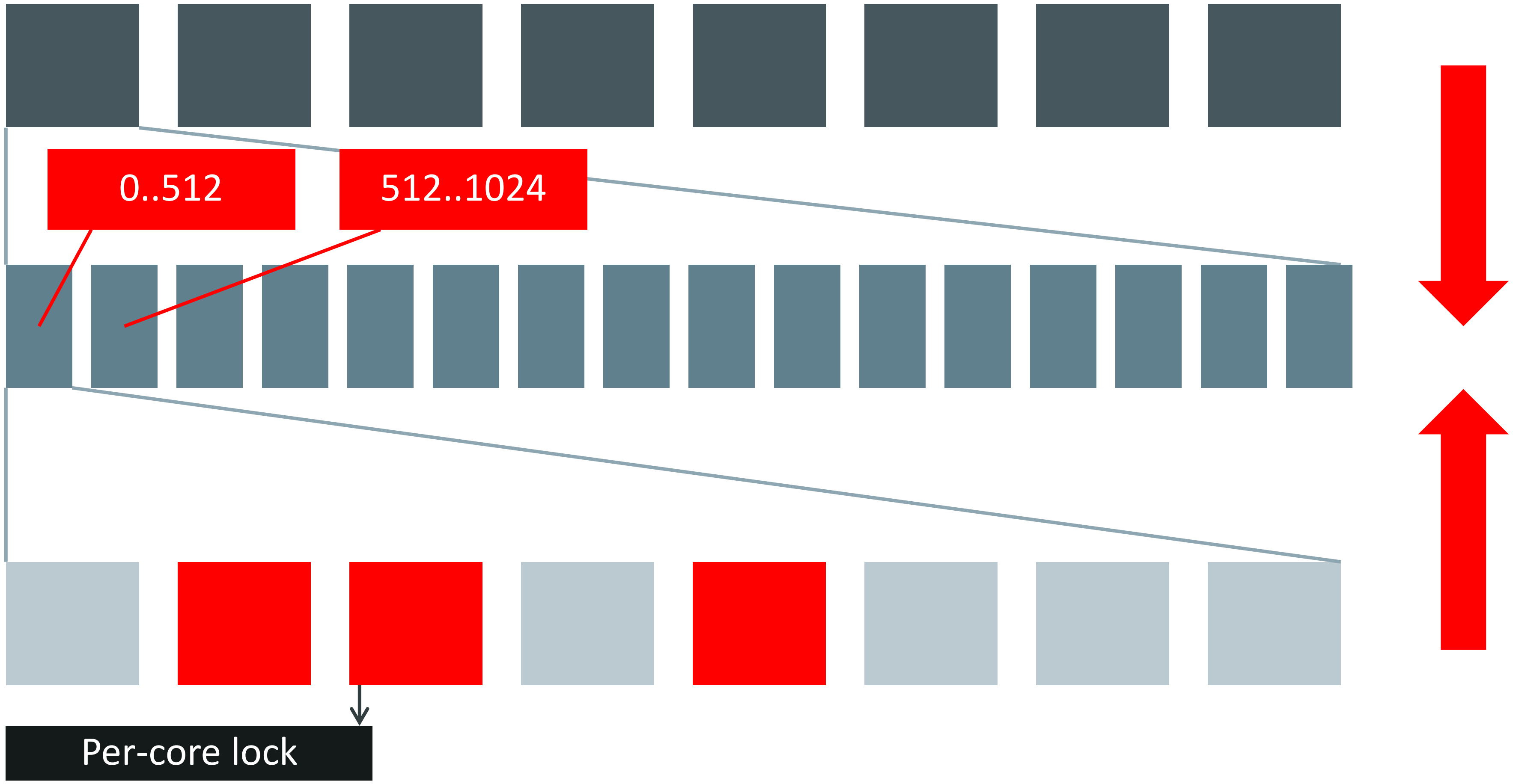
Approach, consider a loop 0..65536, batch size 8



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



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



8 sockets

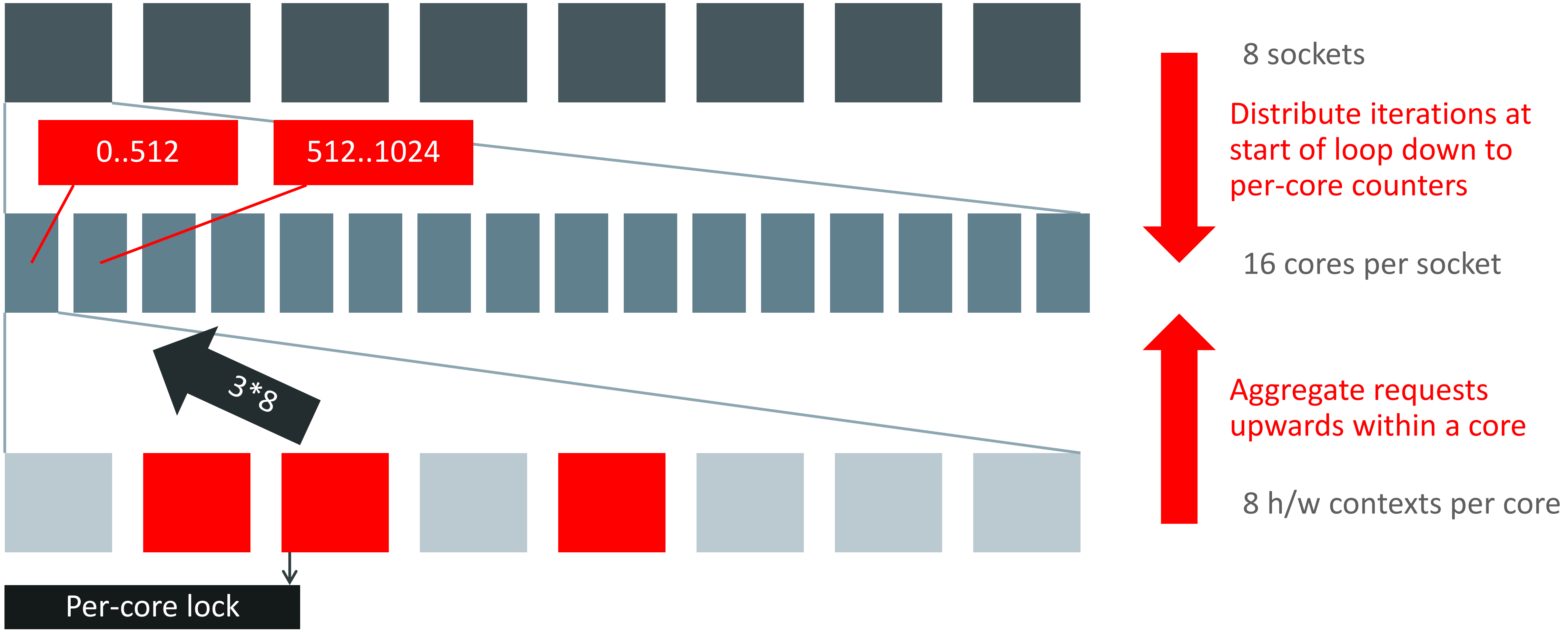
Distribute iterations at start of loop down to per-core counters

16 cores per socket

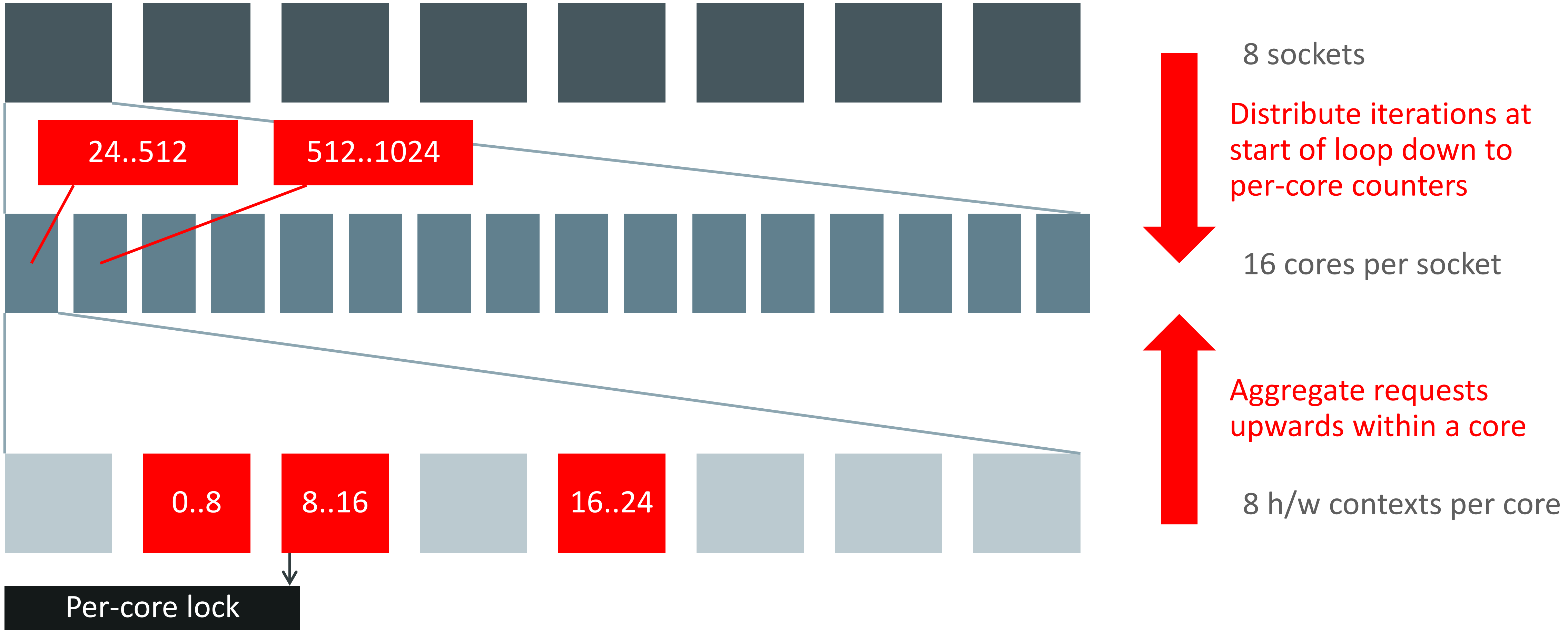
Aggregate requests upwards within a core

8 h/w contexts per core

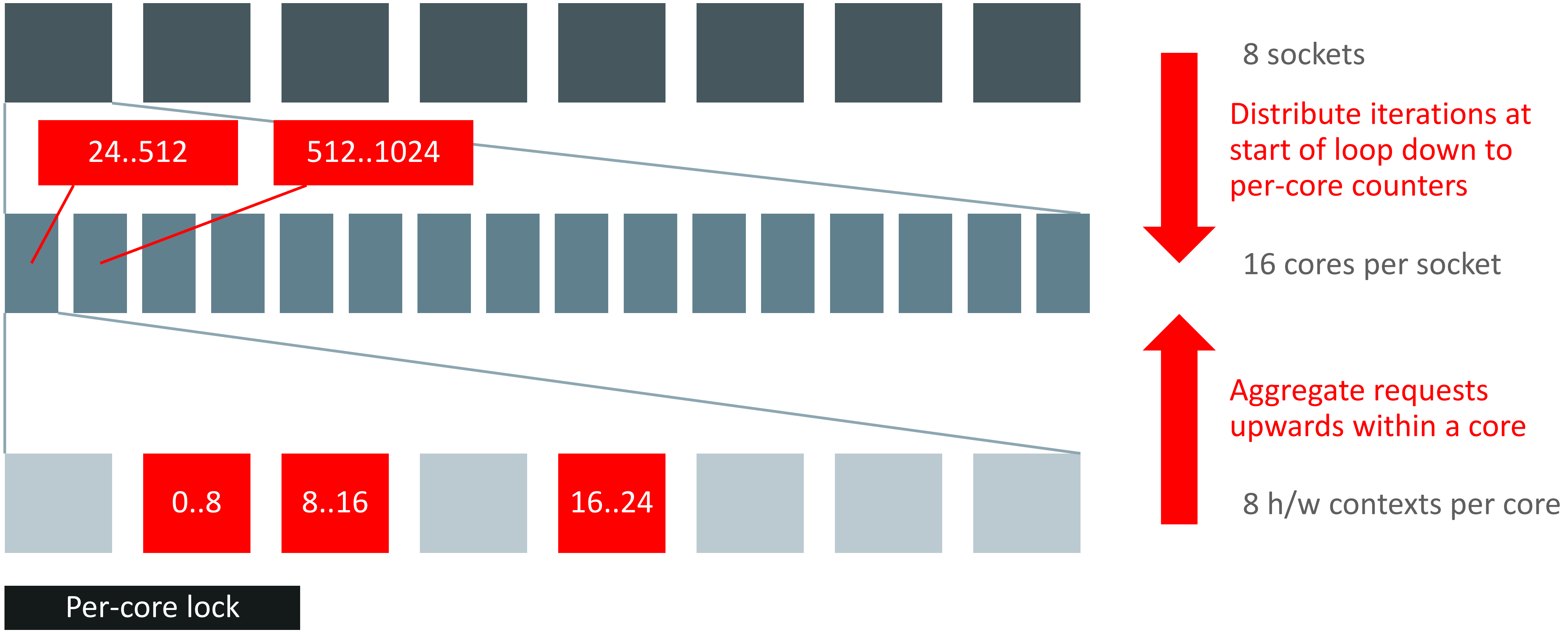
Approach, consider a loop 0..65536, batch size 8



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



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



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

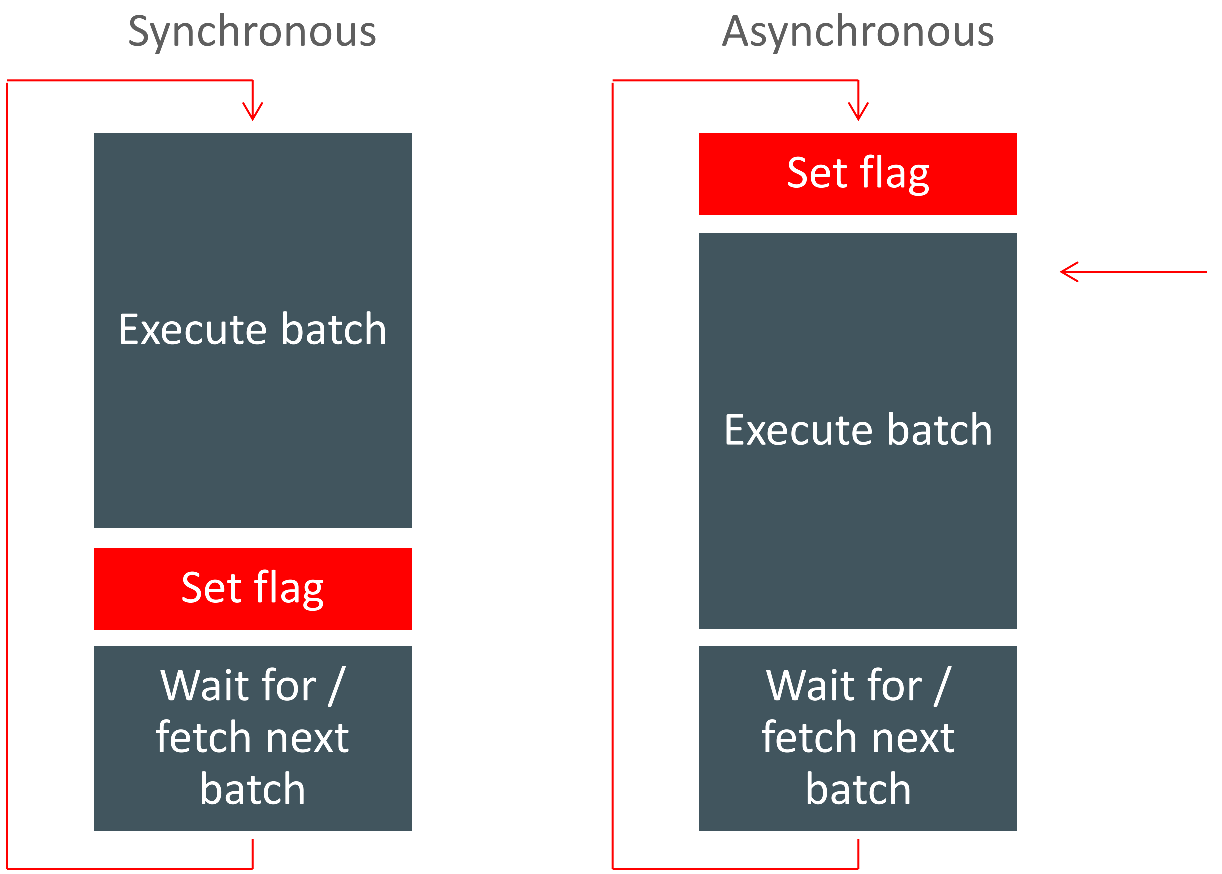
ORACLE

Overview

1. Request combining
2. Asynchronous work requests
3. Non-work-conserving nested loops
4. Results

Asynchronous combining of requests

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

ORACLE

Overview

1. Request combining
2. Asynchronous work requests
3. Non-work-conserving nested loops
4. Results

Nested loops

- Abundant parallelism, why use nesting?

ORACLE

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)

ORACLE

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

ORACLE

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$

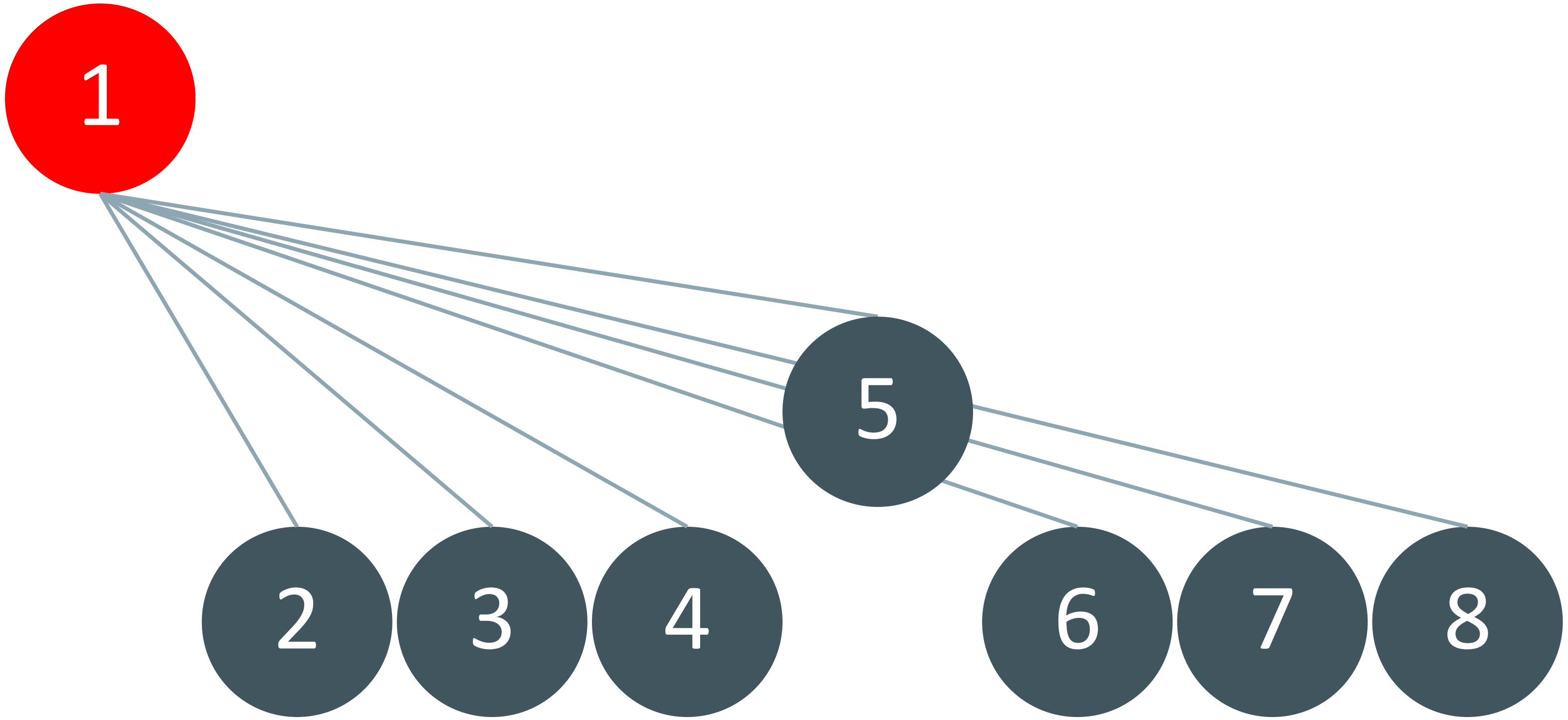
Nested loops

Controlling thread -> loop allocation

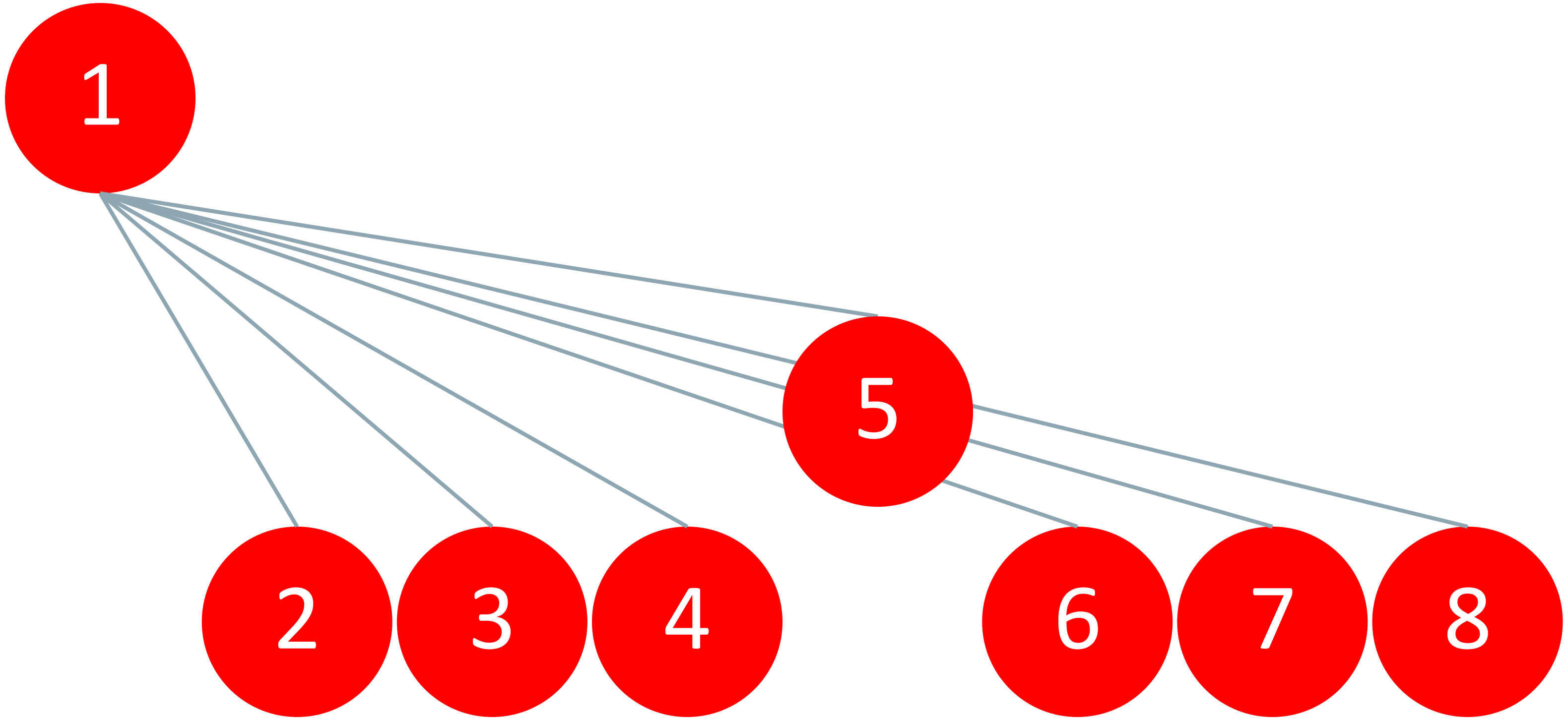
- Number loops “inside out”
  - Level 0 => innermost
  - Level 1 => may contain a level-0 loop
- Each thread also has a level
  - It will execute iterations <= its own level
  - Level 0 thread: only executes inner-most loop iterations
  - …

ORACLE

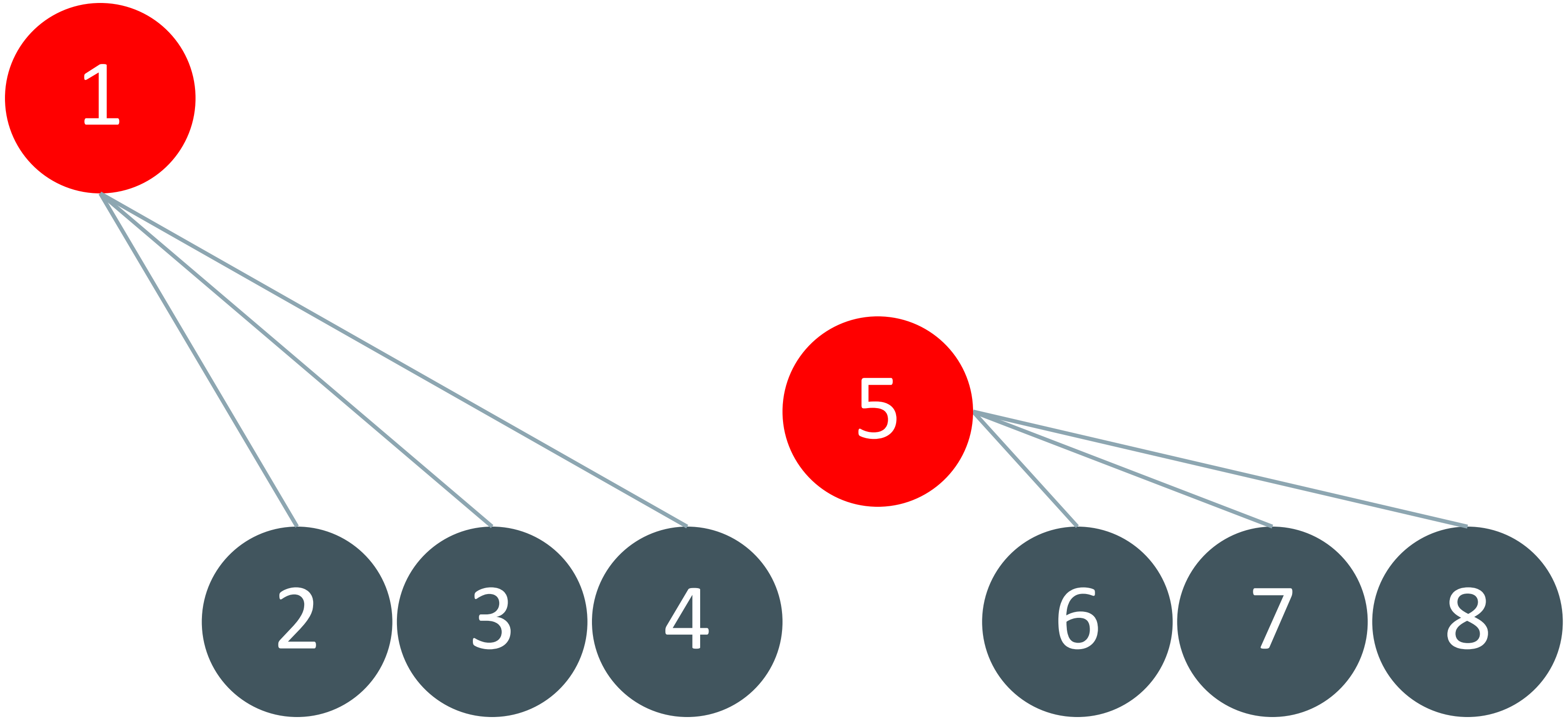
Nested loops



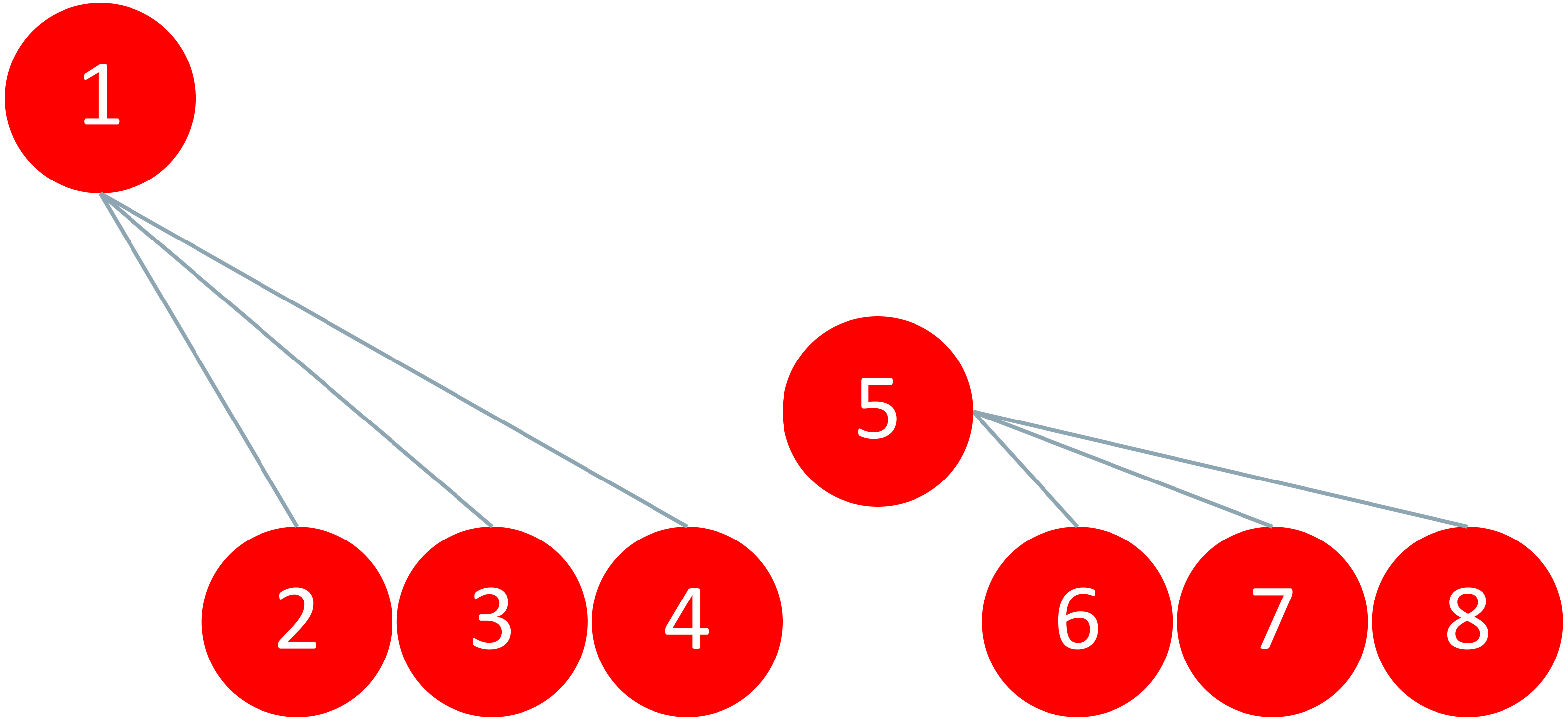
Nested loops: non-nested level 0 – all threads participate



Nested loops: outer (level 1) – just 1+5 participate



Nested loops: inner (level 0) –help respective leaders



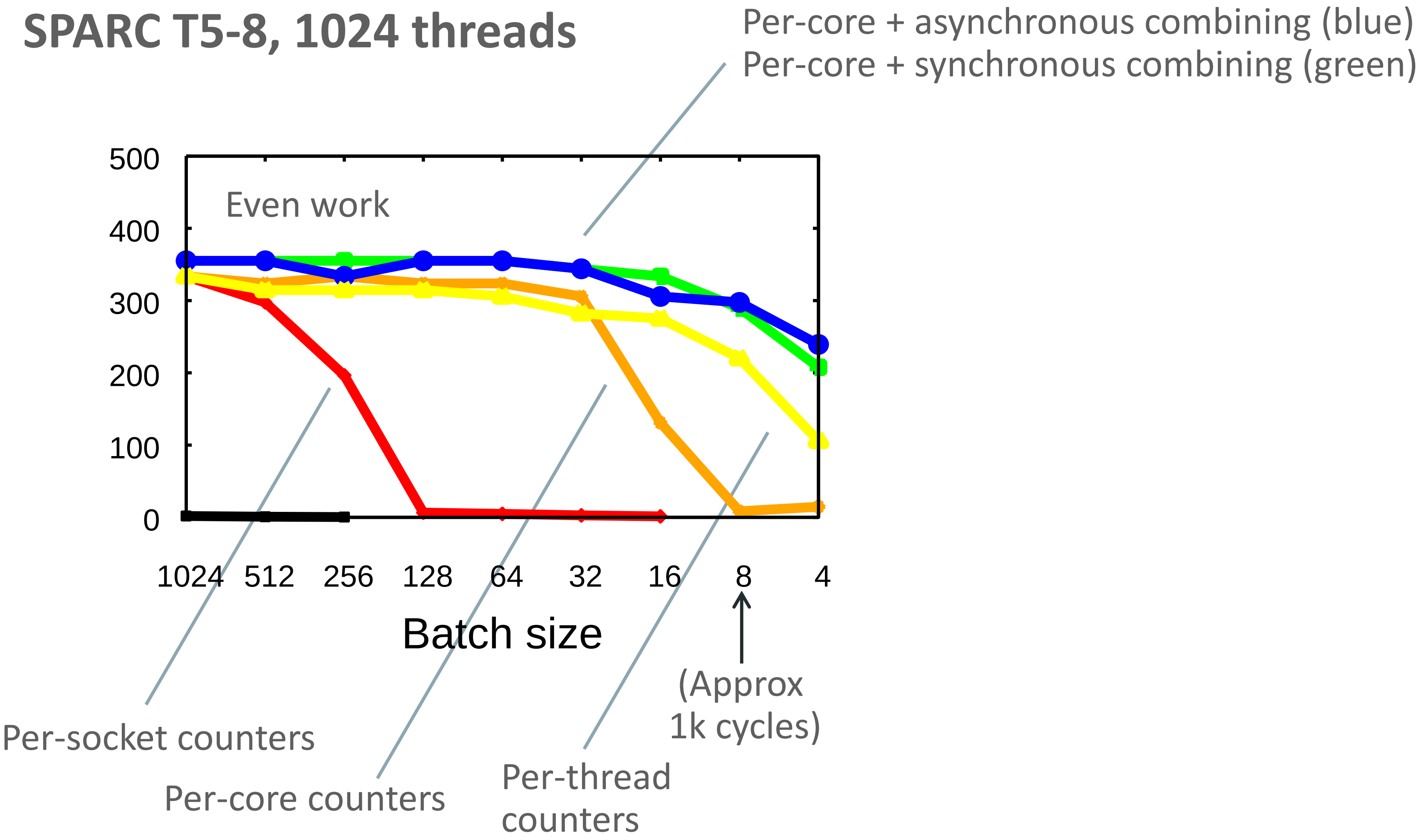
Overview

1. Request combining
2. Asynchronous work requests
3. Non-work-conserving nested loops
4. Results

ORACLE

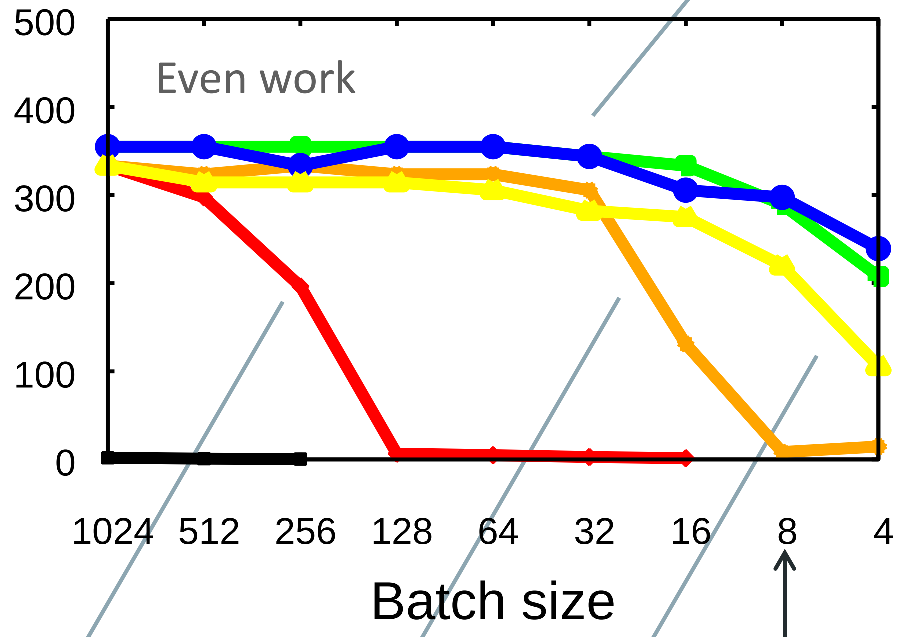
Microbenchmark results

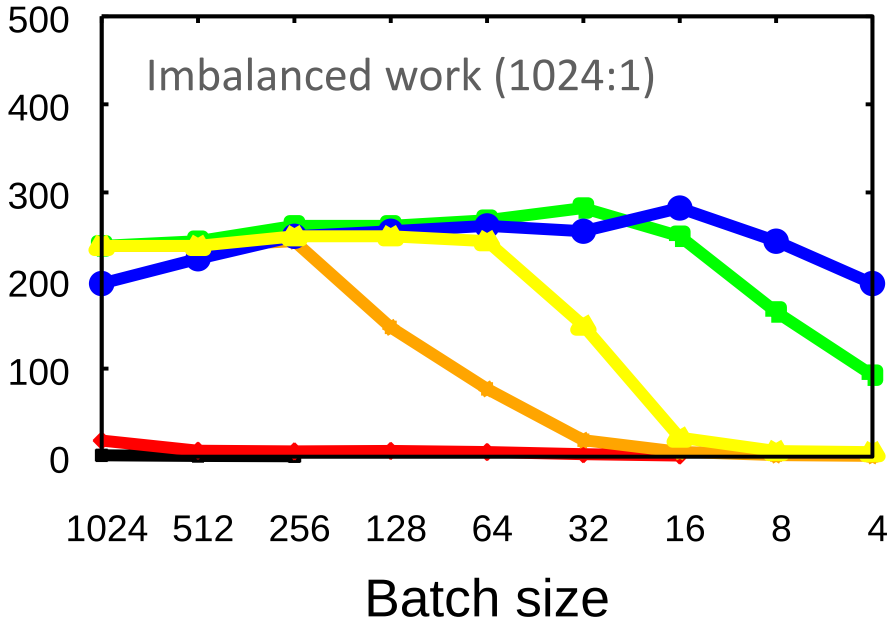
SPARC T5-8, 1024 threads



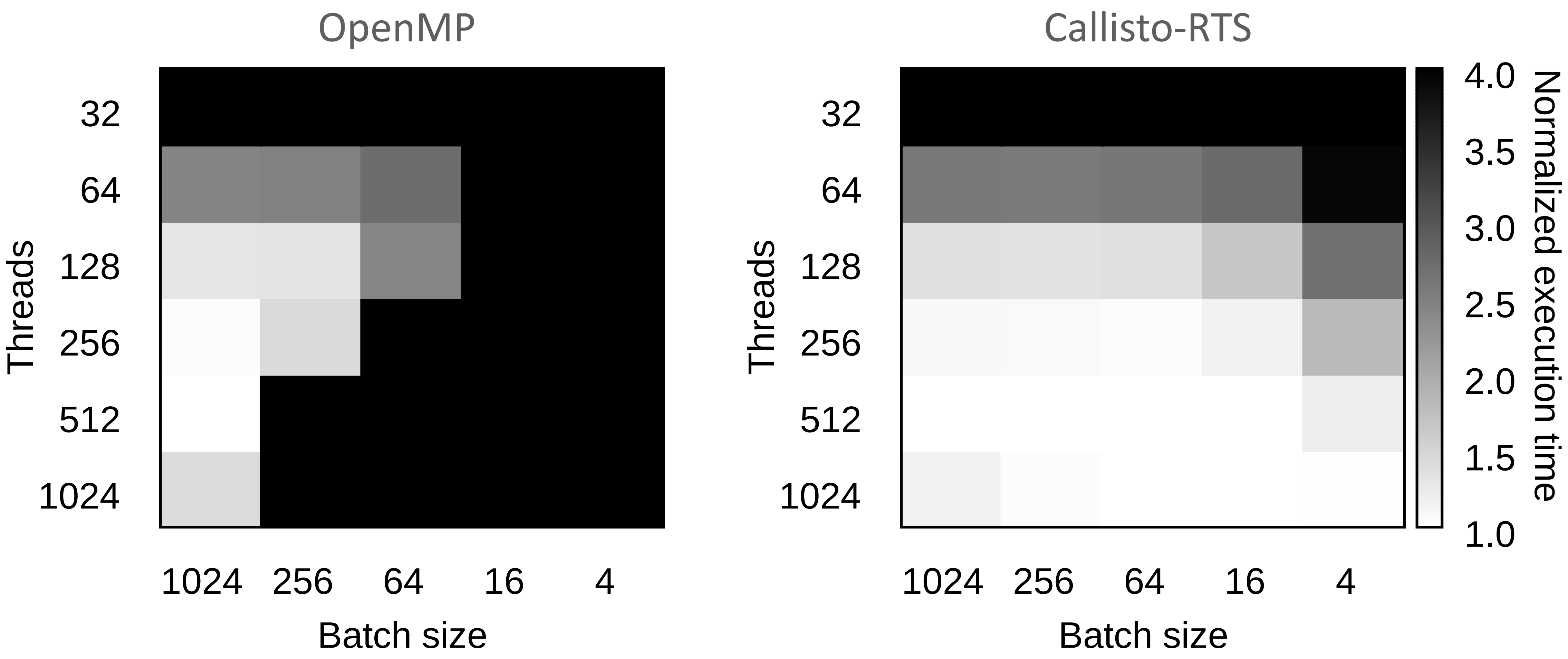
Microbenchmark results

SPARC T5-8, 1024 threads

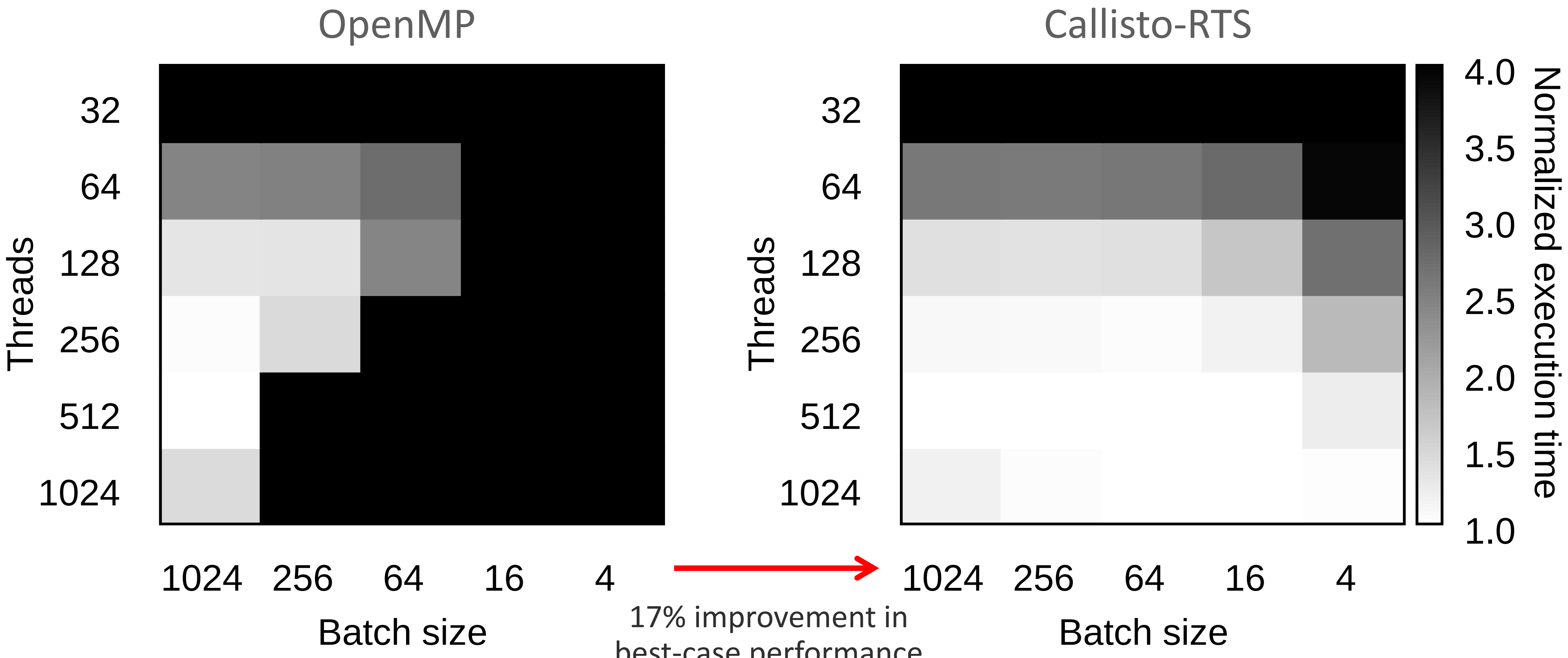




PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)



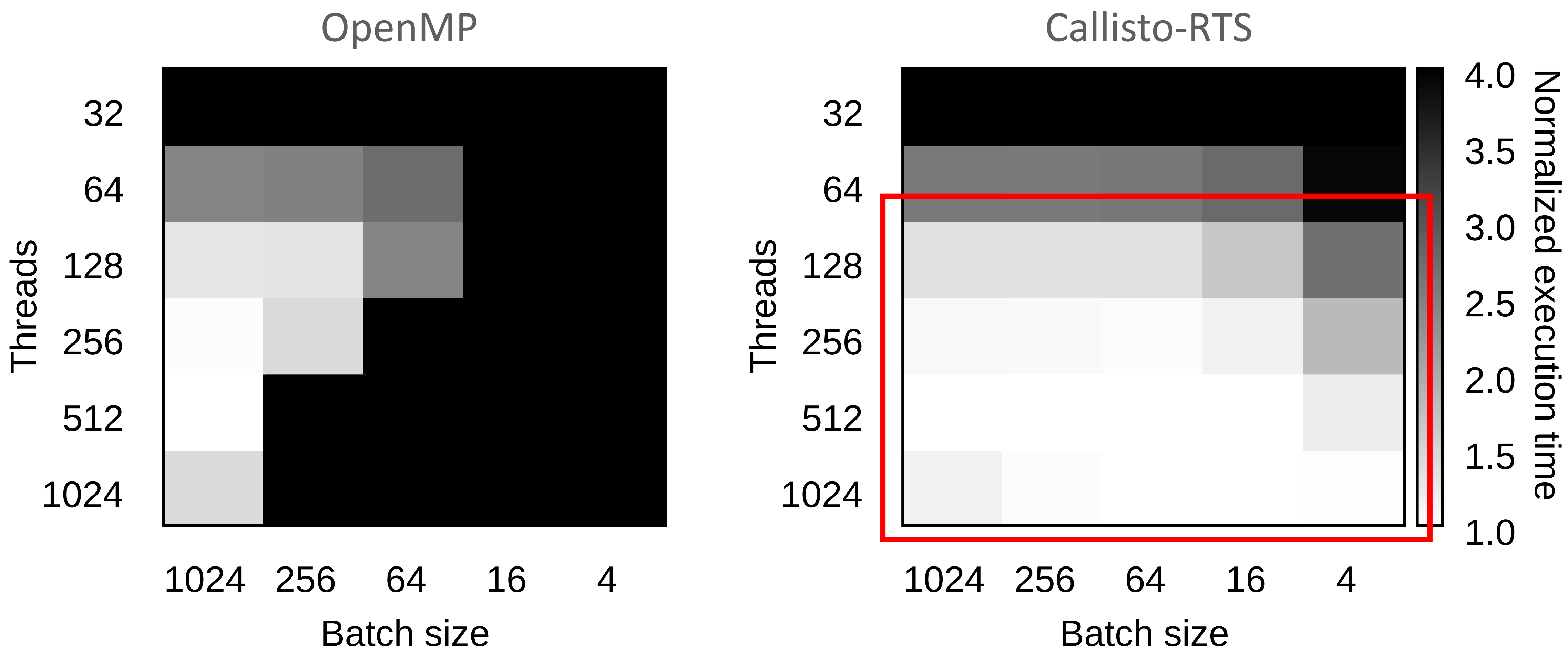
PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)



17% improvement in best-case performance

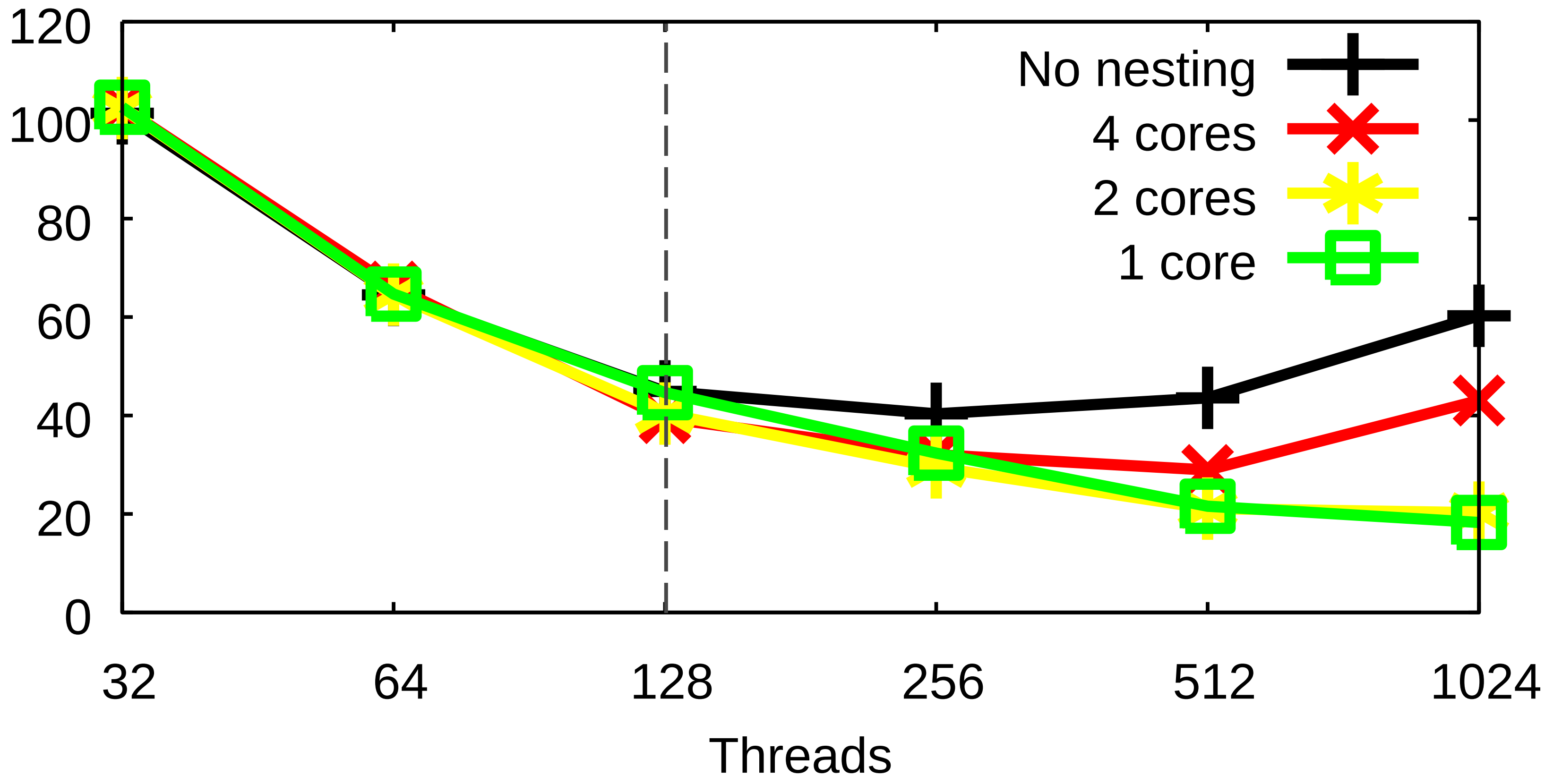
ORACLE

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



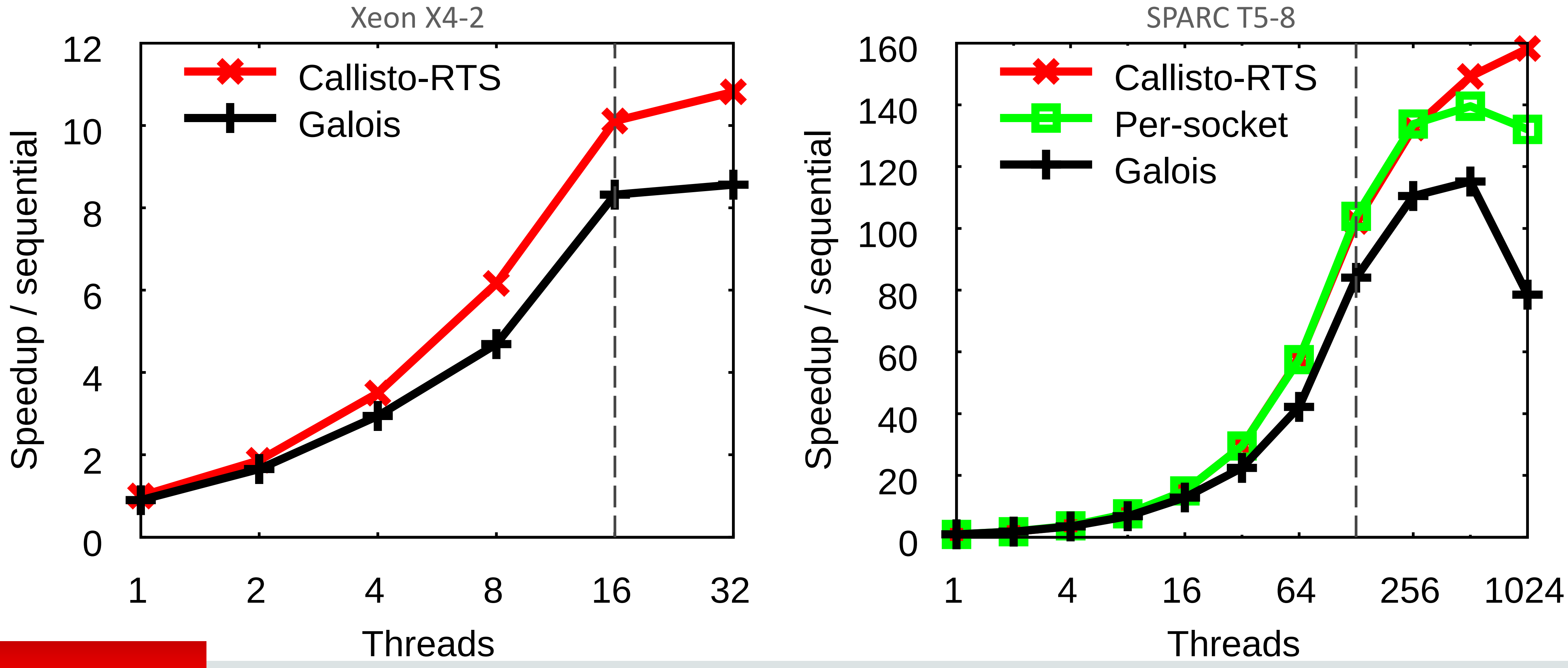
Betweenness-centrality

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



Comparison with Galois

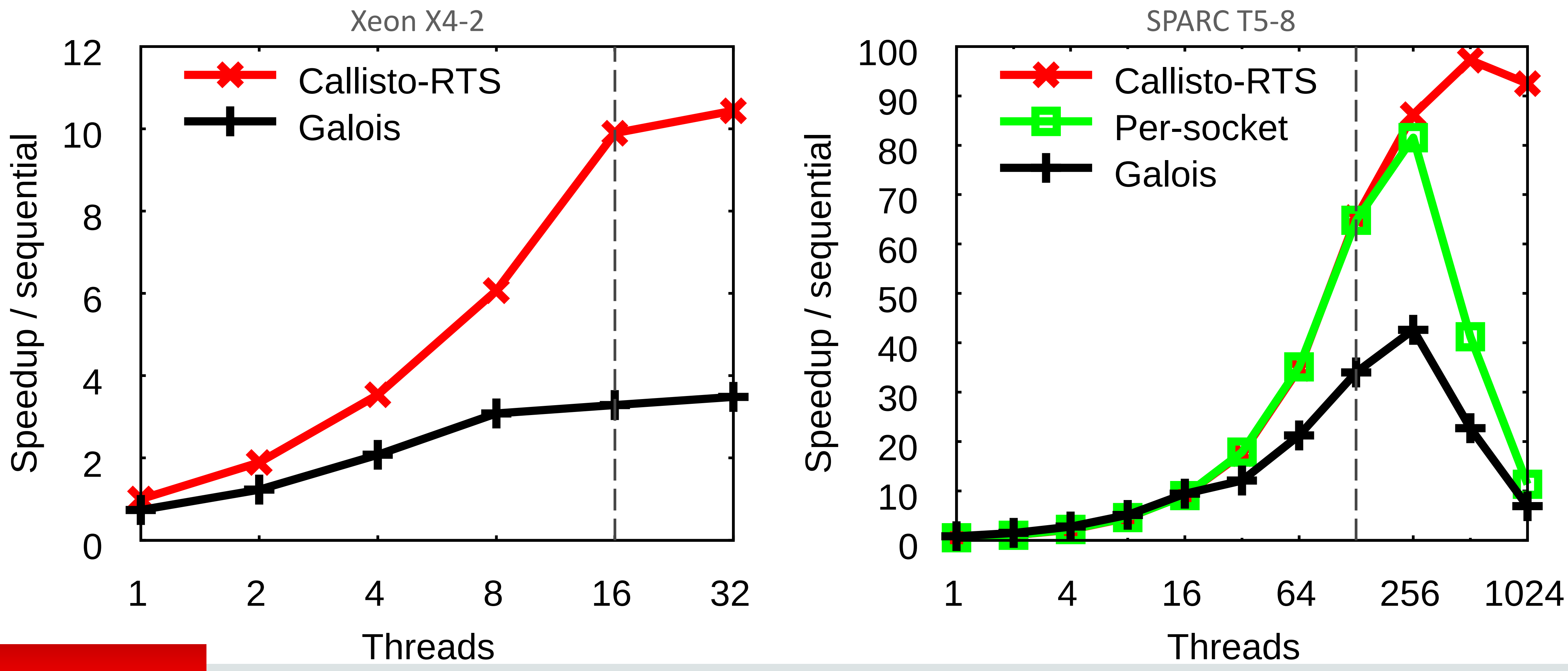
SNAP Twitter data set



ORACLE

Comparison with Galois

SNAP LiveJournal data set



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

ORACLE

Integrated Cloud

Applications & Platform Services