Fine-Grained Locks for Multithreaded Grid Operations

Santiago Ospina De Los Ríos, Prof. Peter Bastian

DUNE User Meeting 2023, Dresden
18.09.2023

Interdisciplinary Center for Scientific Computing

Federal Ministry of Education and Research

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Contents

- Motivation
- Assembly of Finite Elements
- Grid Partition & Scheduling
- Mask Shared Region
- Fine-Grained Locks
- Shared Memory vs Private Memory
- Conclusions

Motivation

Desktop Environments

Generic Binary

It’s very hard to bundle MPI in a generic binary for usage distribution. (e.g., multi-platform GUI)

Multi-Tasking

Program is shared with other unknown tasks that may need higher priority.

High Performance Computing

Surface-to-Volume Ratio

High dimension or high polynomial degree FE problems suffer from a high surface-to-volume ratio. This translates on higher communication overhead.

Node Level Load Balancing

Sharing memory between processes allows the use of fair work stealing algorithms

Assembly of Finite Elements

Volume Integrals

```
def residual(test, trial, coeff):
    ltest = test.localView()
    ltrial = trial.localView()
    residual = [0., ..., 0.]
    for entity in grid_view:
        bind(entity, ltest, ltrial)
        lcoeff = localVector(coeff, ltrial)
        lresidual = localResidual(ltest, ltrial, lcoeff)
        accumulateVector(residual, ltest, lresidual)
    return residual
```

```
def localVector(vector, lspace):
    lvector = [0., ..., 0.]
    for dof in range(lspace.size):
        lvector[dof] = vector[lspace.index(dof)]
    return lvector
```

$$\int_T \alpha_V(u_h, v_h) - \lambda_V(v_h)$$

```
def accumulateVector(vector, lspace, lvector):
    for dof in range(lspace.size):
        vector[lspace.index(dof)] += lvector[dof]
```

Assembly of Finite Elements

Data Storage

Data is

- **...*physically organized* arbitrarily in memory**
- **...*temporally accessed* differently depending on the numerics**
- **...*semantically attached* to the topology of the grid**

Assembly of Finite Elements

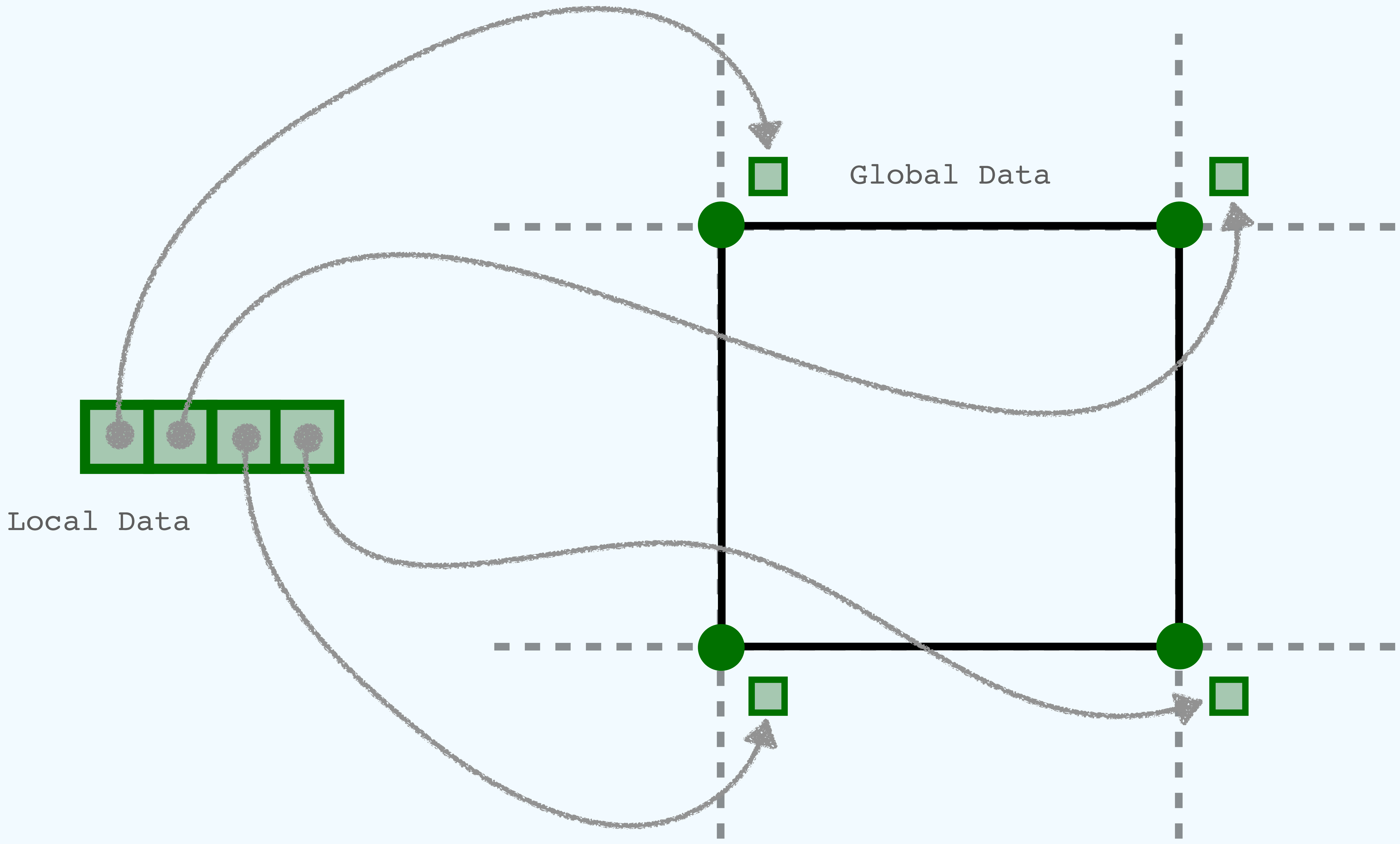
Data Storage

Data is

- …*physically organized* arbitrarily in memory
- …*temporally accessed* differently depending on the numerics
- …*semantically attached* to the topology of the grid

Assembly of Finite Elements

Scatter Data: Local to Global



Assembly of Finite Elements

Grid Partition & Work Scheduling

Mask Shared Region

Fine-Grained Locks

Grid Partition & Scheduling

Where to parallelize?

```
def residual(test, trial, coeff):
    ltest = test.localView()
    ltrial = trial.localView()
    residual = [0., ..., 0.]
    for entity in grid_view:
        bind(entity, ltest, ltrial)
        lcoeff = localVector(coeff, ltrial)
        lresidual = localResidual(ltest, ltrial, lcoeff)
        accumulateVector(residual, ltest, lresidual)
    return residual
```

Grid Partition & Scheduling

Where to parallelize?

```
def residual(test, trial, coeff):
    ltest = test.localView()
    ltrial = trial.localView()
    residual = [0., ..., 0.]
    for entity in grid_view:
        bind(entity, ltest, ltrial)
        lcoeff = localVector(coeff, ltrial)
        lresidual = localResidual(ltest, ltrial, lcoeff)
        accumulateVector(residual, ltest, lresidual)
    return residual
```

Grid Partition & Scheduling

Grid Partition

```
def partition(grid_view, n):
    begin_it = grid_view.begin()
    chunk = grid_view.size(0) / n
    remainder = grid_view.size(0) % n
    ranges = []
    for i in range(n-1):
        next_end = begin_it + (chunk + (remainder ? 1 : 0))
        ranges.append([begin_it, next_end])
        begin_it = next_end
        if reminder:
          reminder = reminder - 1
    ranges.append([begin_it, grid_view.end()])
    return ranges
```

Naive Partition

- \+ Easy: Split iterators in equal chunks
- \+ Generic to any grid
- \+ Enables same cache use as original grid
- \- Unknown size of shared region
- \- Maybe unbalanced

Load Balanced Naive Partition

- \+ Add many (naive) partitions to TBB
- \+ Generic to any grid
- \+ Enables same cache use as original grid
- \+ Automatically balanced
- \- Shared region is bigger than Naive Partition

Critical Section

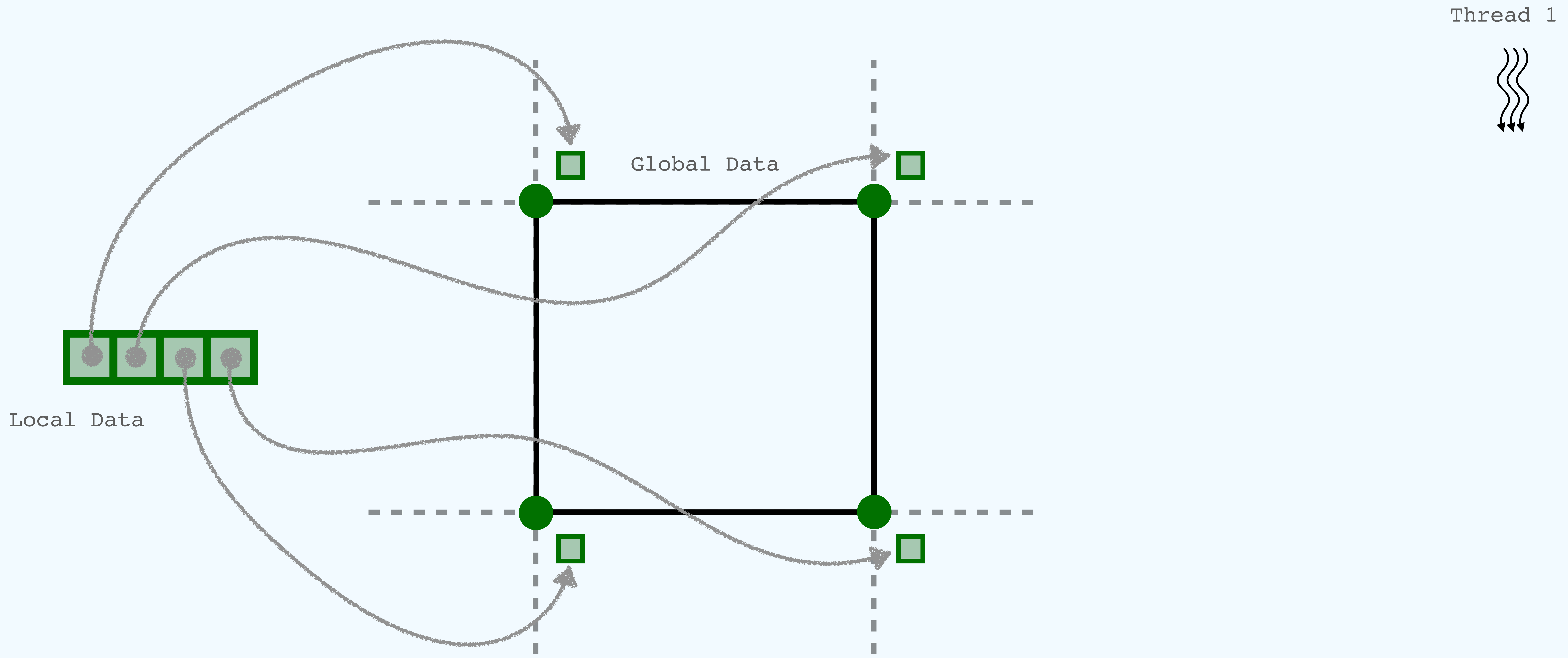
Two or more threads may race to access the same global data

```
def residual(test, trial, coeff):
    ltest = test.localView()
    ltrial = trial.localView()
    residual = [0., ..., 0.]
    for entity in grid_view: # multi-threaded
        bind(entity, ltest, ltrial)
        lcoeff = localVector(coeff, ltrial)
        lresidual = localResidual(ltest, ltrial, lcoeff)
        accumulateVector(residual, ltest, lresidual)
    return residual
```

```
def accumulateVector(vector, lspace, lvector):
    for dof in range(lspace.size):
        vector[lspace.index(dof)] += lvector[dof]
```

Critical Section

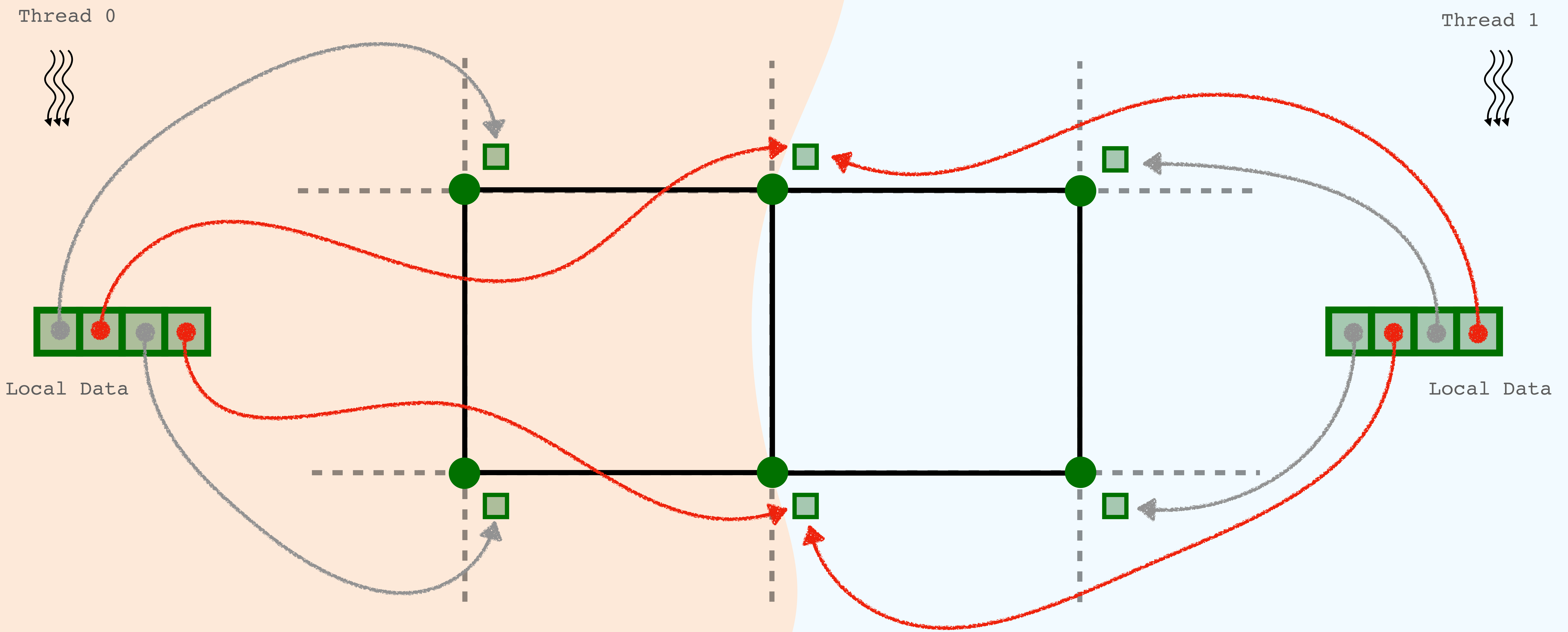
Thread access data as in the sequential case



vector[lspace.index(dof)] += lvector[dof]

Critical Section

Two or more threads may race to access the same global data




```
vector[lspace.index(dof)] += lvector[dof]
```

```
vector[lspace.index(dof)] += lvector[dof]
```

Assembly of Finite Elements

Grid Partition & Work Scheduling

Mask Shared Region

Fine-Grained Locks

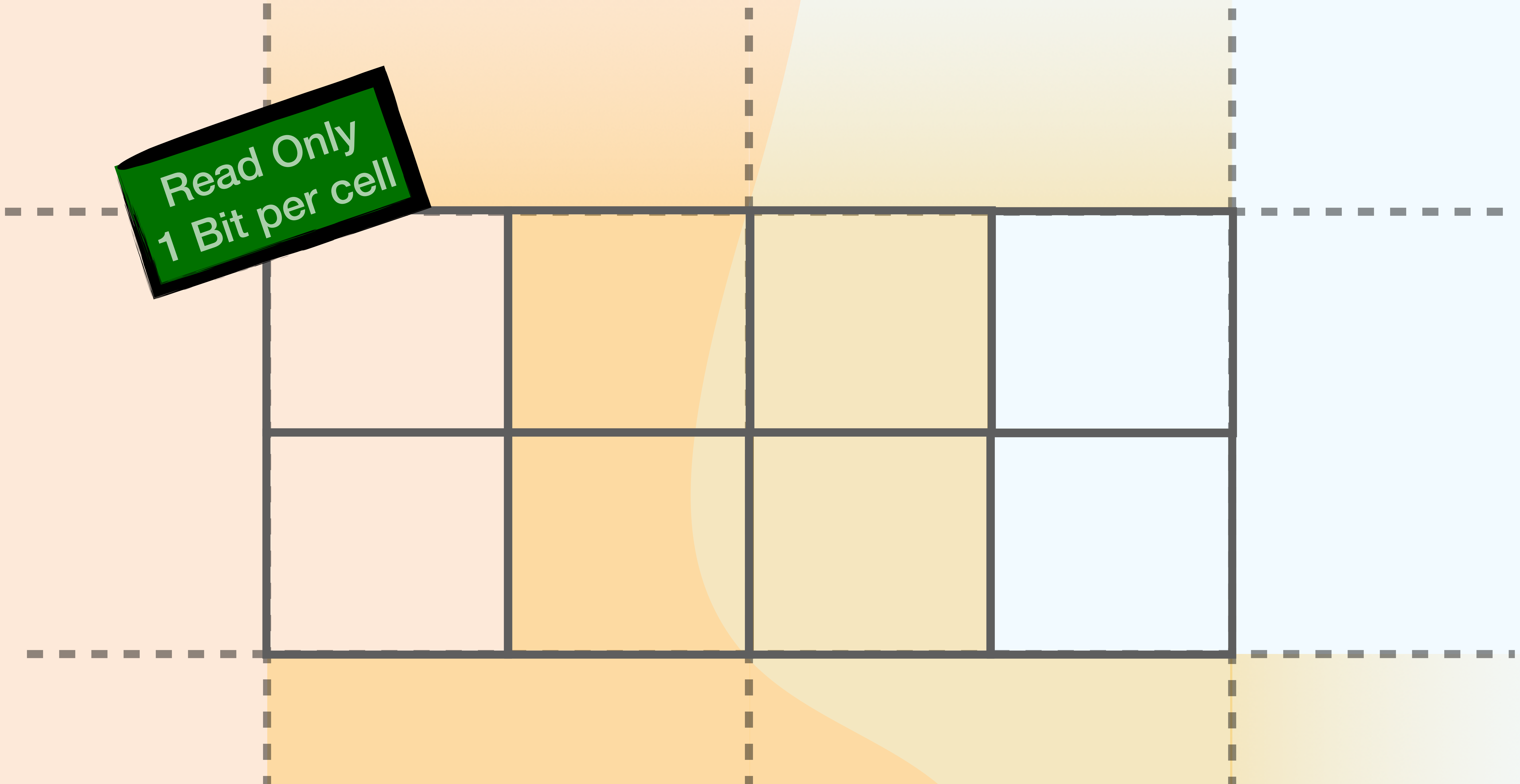
Masking of Critical Section

Shared Region on Grid Partitions

Partition 0

Partition 1

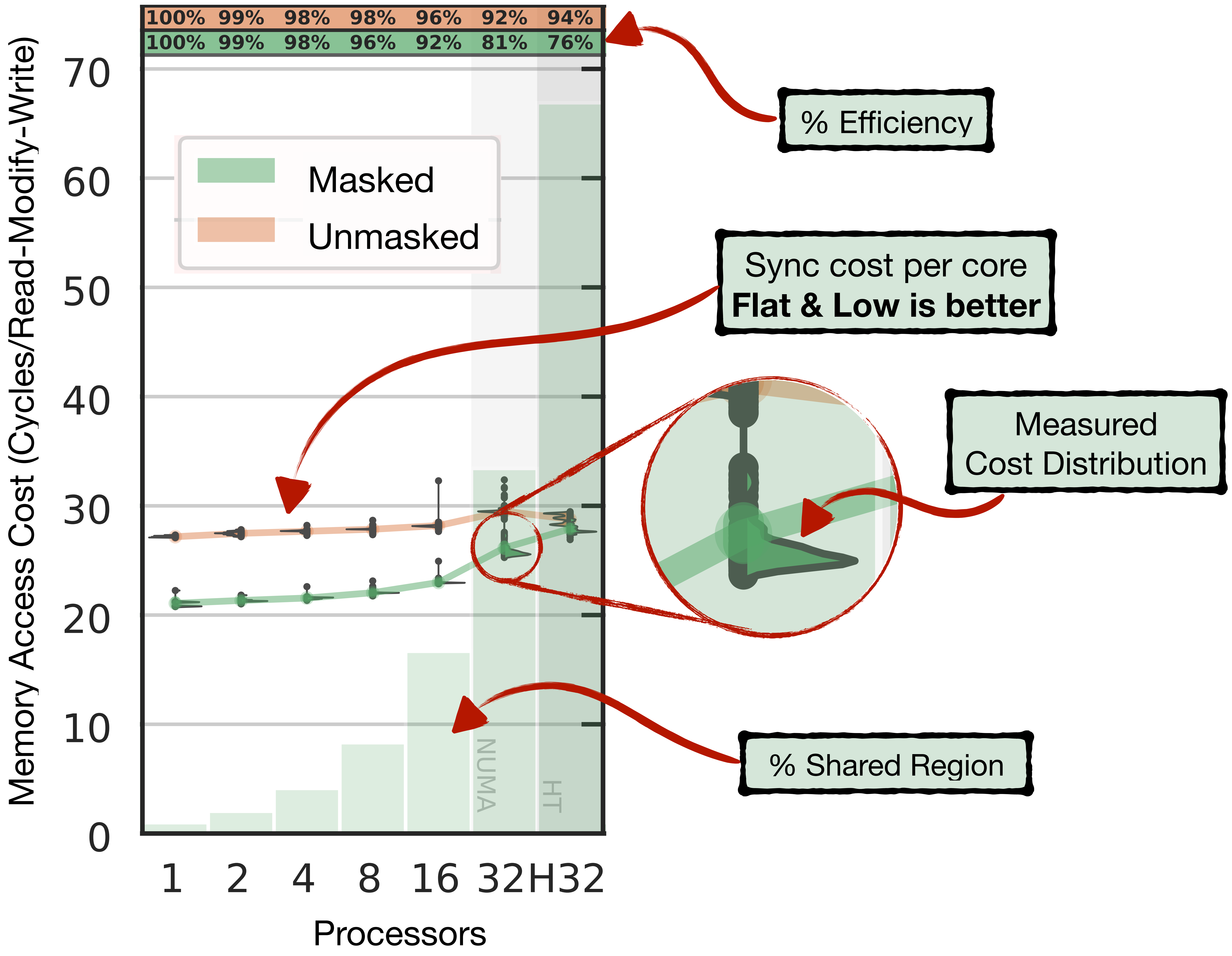
1. Assign a unique owner to each sub-entity
2. Find the shared region on all sub-entities
3. Collect the shared region into back into the cell



Critical Section Micro-Benchmark

A proxy for synchronization cost

```
def benchmark(test, trial, vector):
    for entity in grid_view: # multi-threaded
        bind(entity, ltest, ltrial)
        accumulateVector(vector, ltest)
        unbind(ltest, ltrial)
```



Assembly of Finite Elements

Grid Partition & Work Scheduling

Mask Shared Region

Fine-Grained Locks

Fine-Grained Locks

Same task, different exclusivity modes to access memory

Amount of data exclusive to the lock

`Mutex (std::mutex)`

`Mutex & Batched Buffer`

`Batched Data Lock (std::mutex/N)`

`Grid Entity Locks`

`Atomic Lock (std::atomic & std::atomic_ref)`

Syncronization cost

Fine-Grained Locks

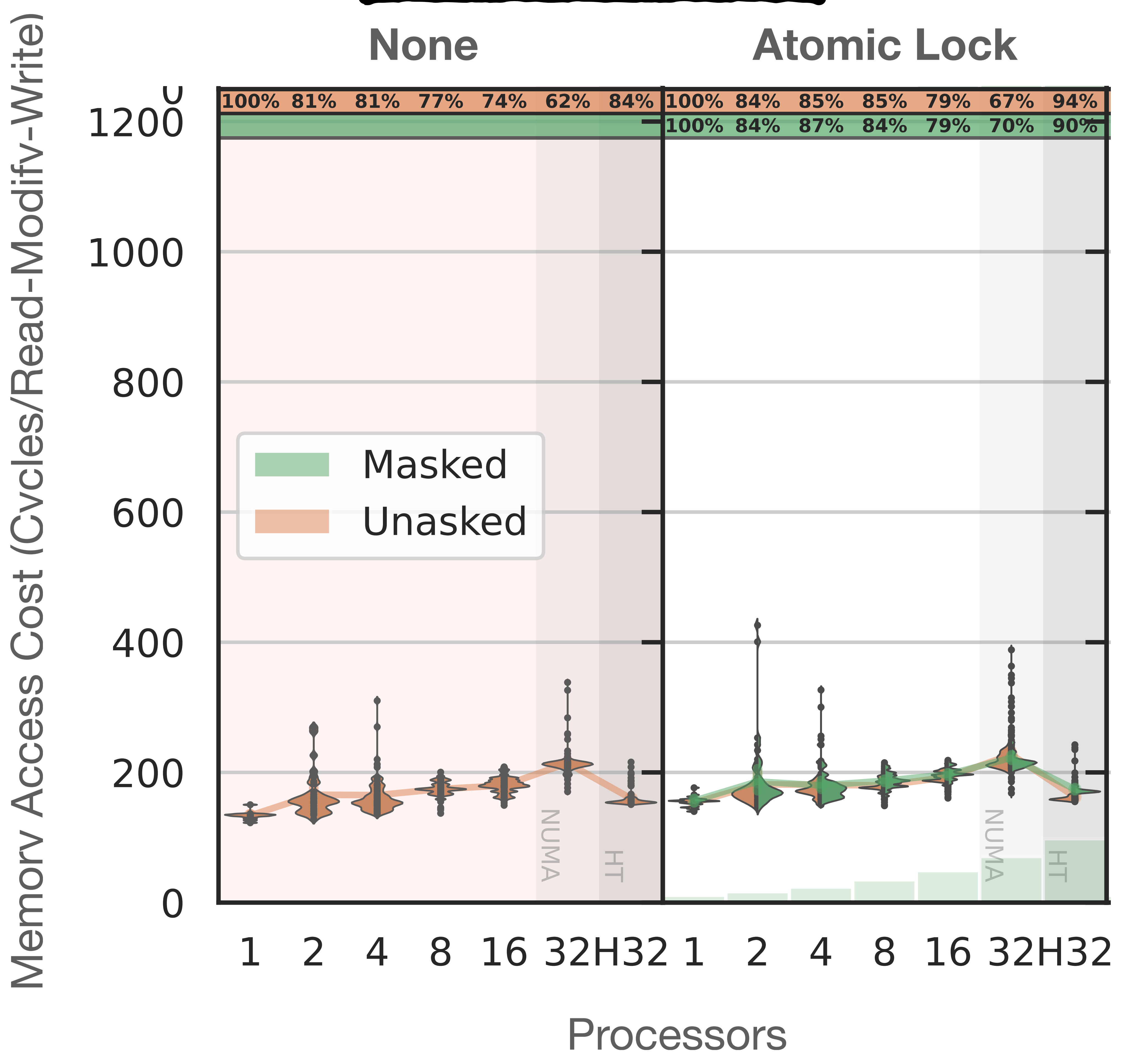
Atomic Lock (std::atomic & std::atomic_ref)

Each data entry has its own Compare & Swap lock

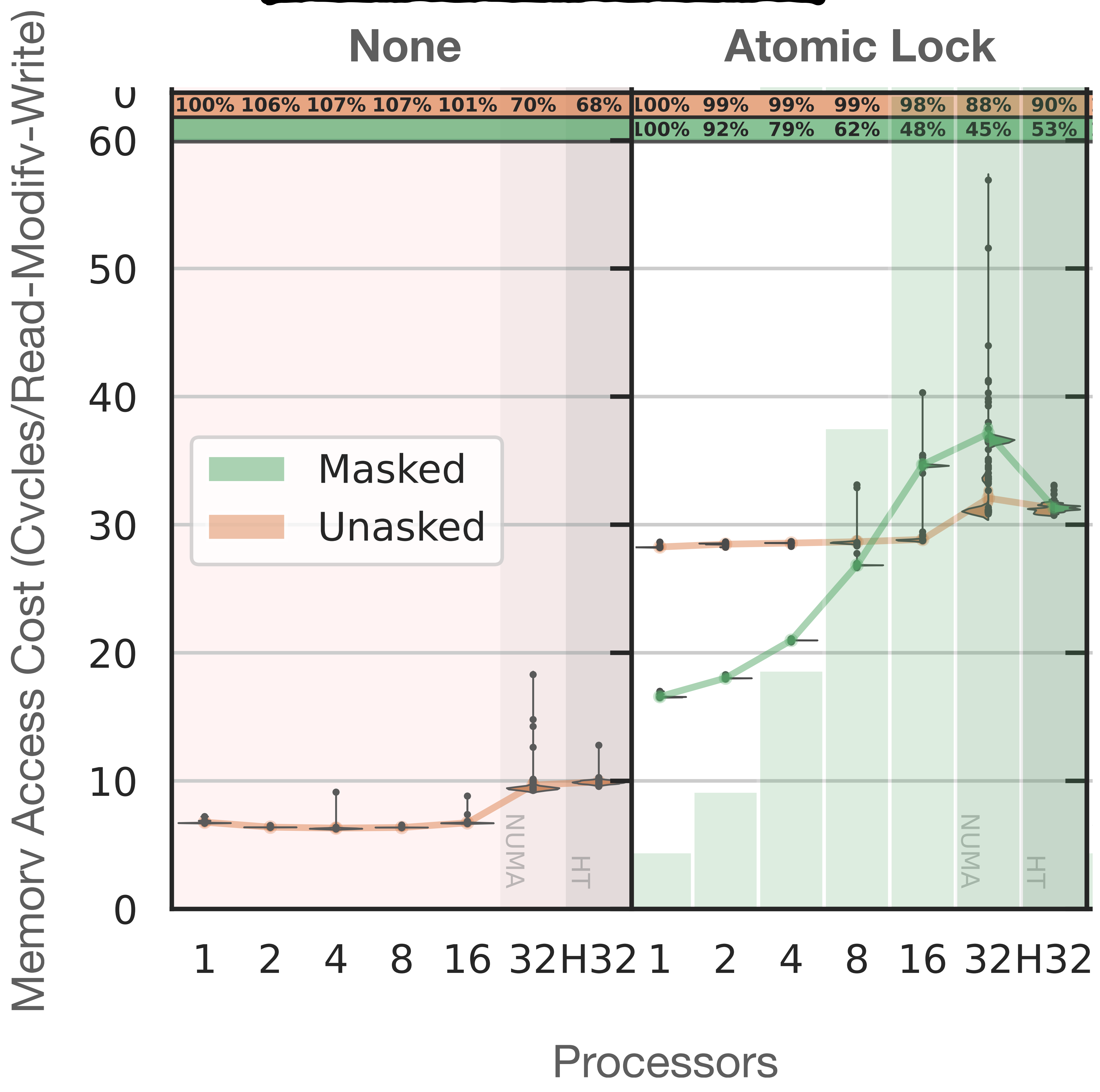
Fine-Grained Locks

`Atomic Lock (std::atomic & std::atomic_ref)`

P1 elements in 2D



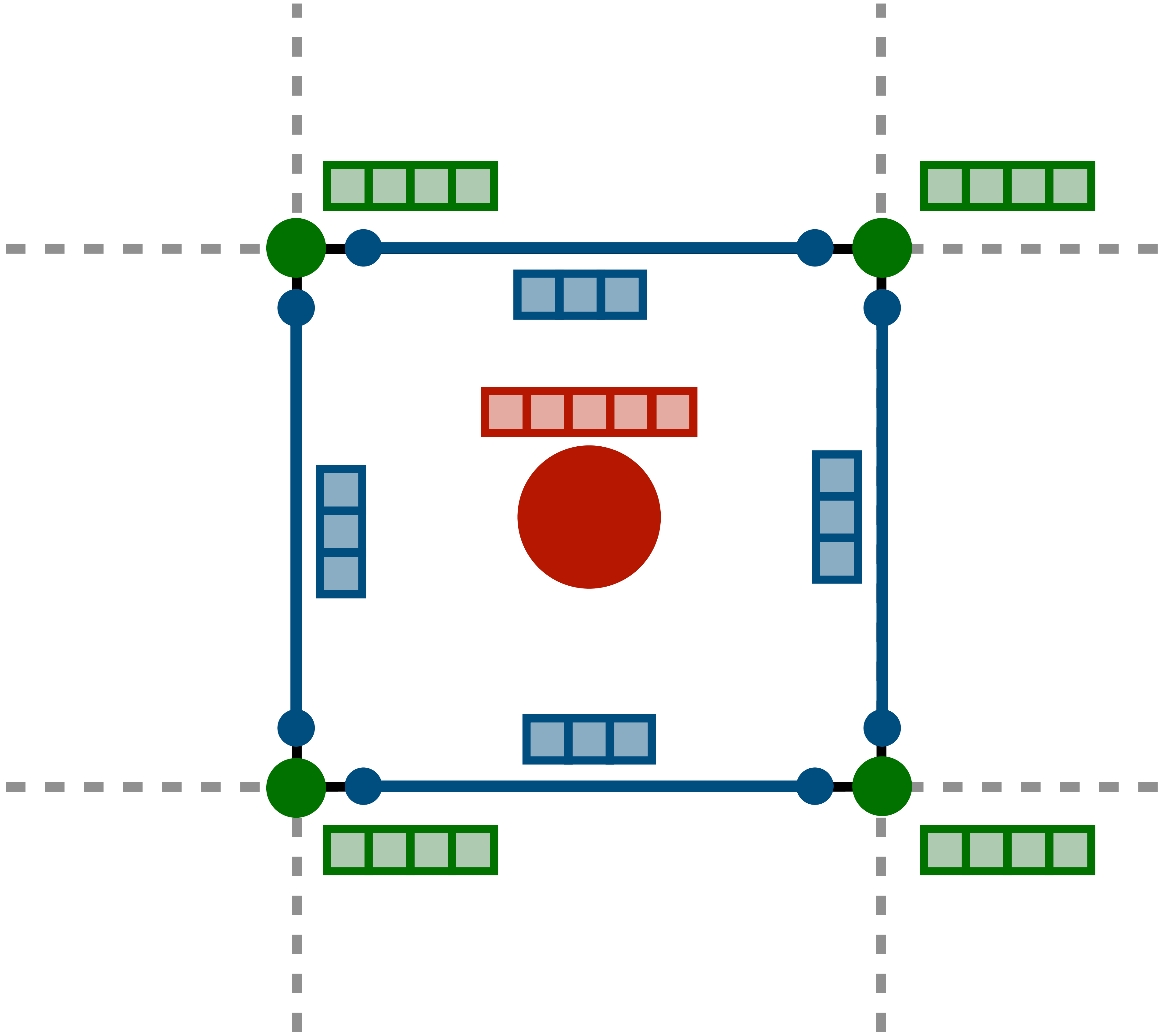
Q5DG elements in 2D



Fine-Grained Locks

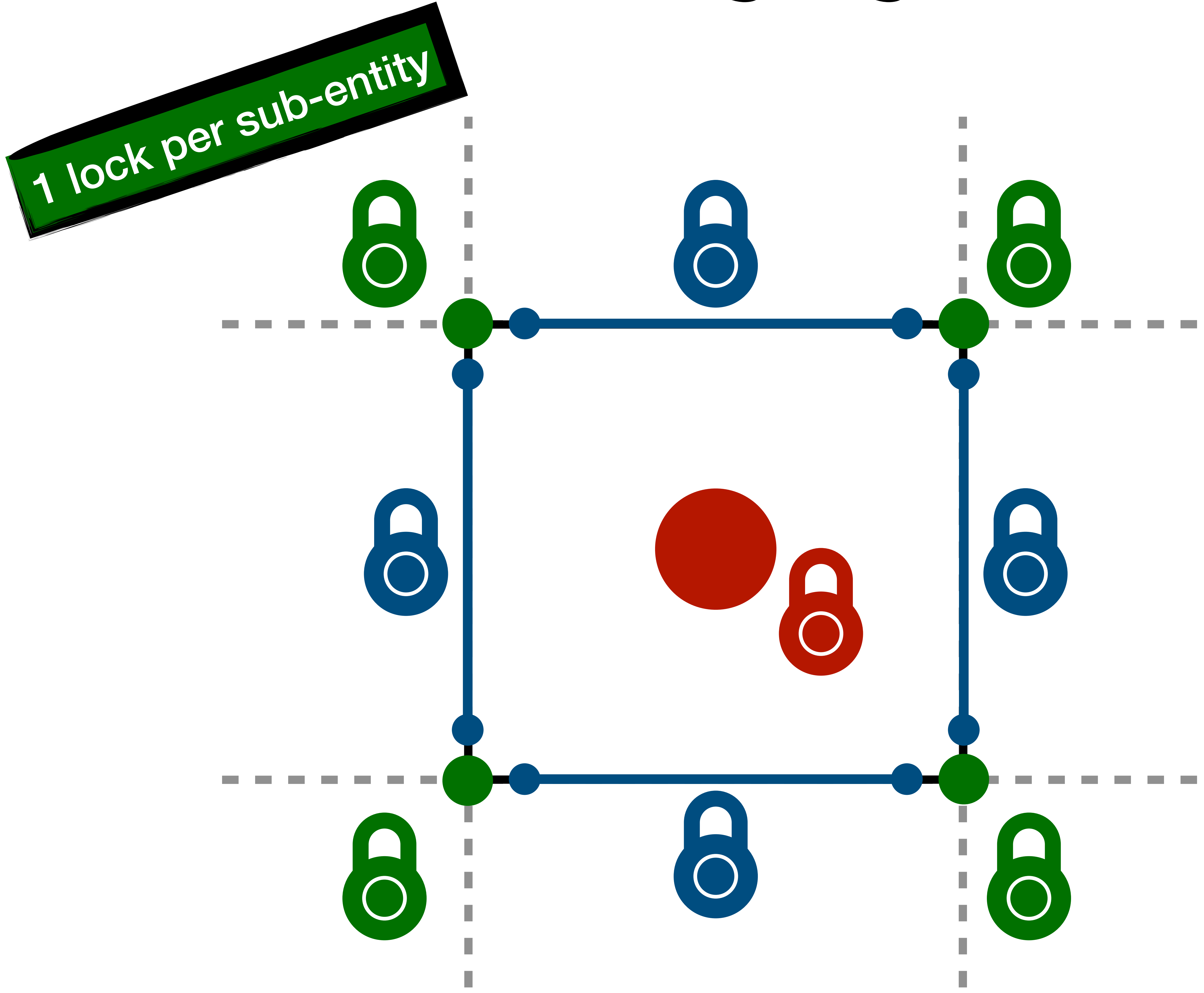
Let's solve the consistency problem locally

```
def accumulateVector(vector, lspace, lvector):
    lspace.lock()
    for dof in range(lspace.size):
        vector[lspace.index(dof)] += lvector[dof]
    lspace.unlock()
```



Fine-Grained Locks

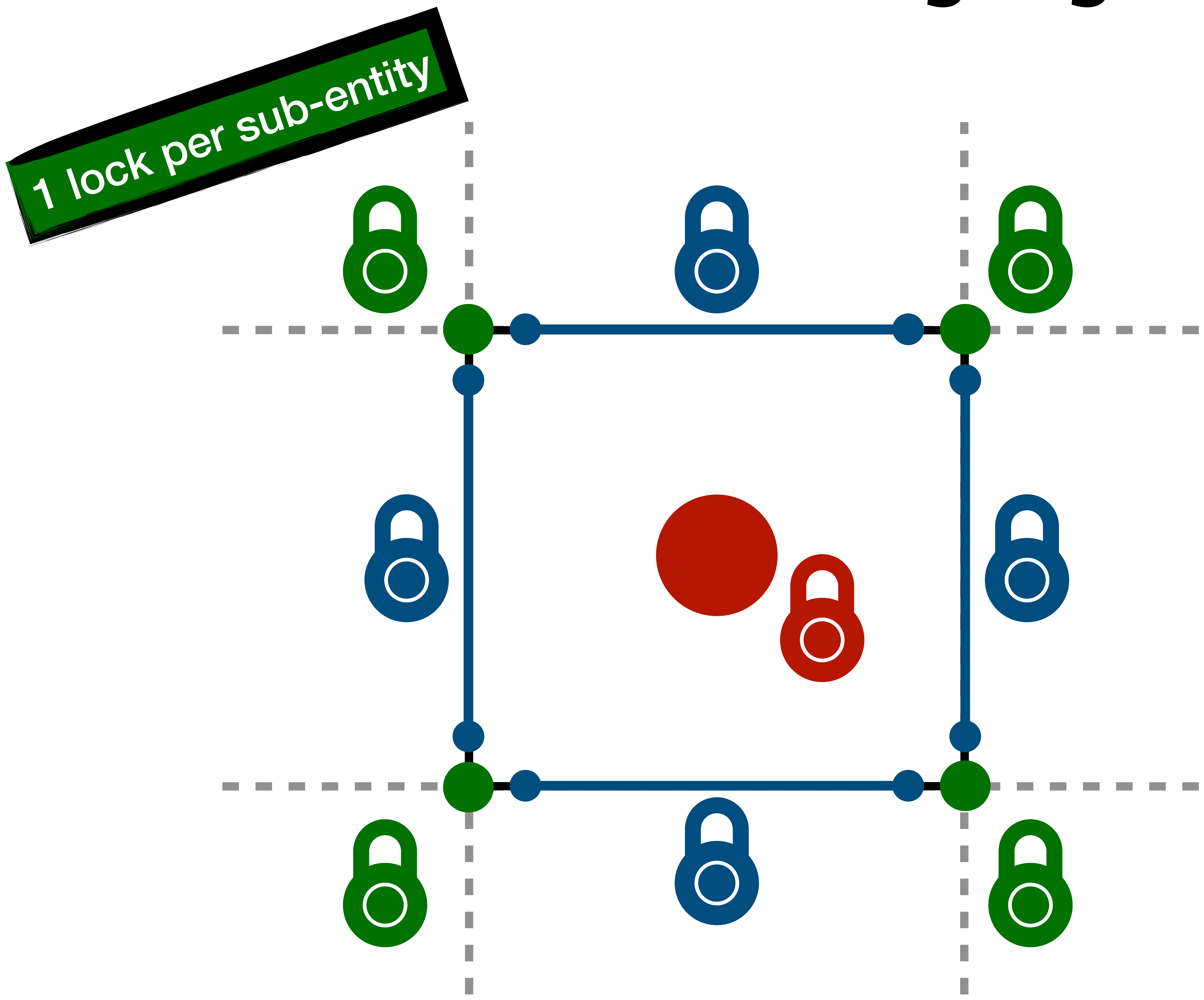
Locking algorithm: Avoiding deadlocks




```
class LocalSpace:
    # true: successful! we have the lock
    # false: failed! another thread has the lock
    def try_lock(self):
        # list of sub-entity padlocks
        padlocks = [🔒,🔒,...,🔒,🔒,...🔒]
        size = len(padlocks)
        # try to lock all the padlocks
        for i in range(size):
            if not padlocks[i].try_lock():
                # release all our locked padlocks
                for j in range(size-i-1):
                    padlocks[j].unlock()
                return False
        return True
```

Fine-Grained Locks

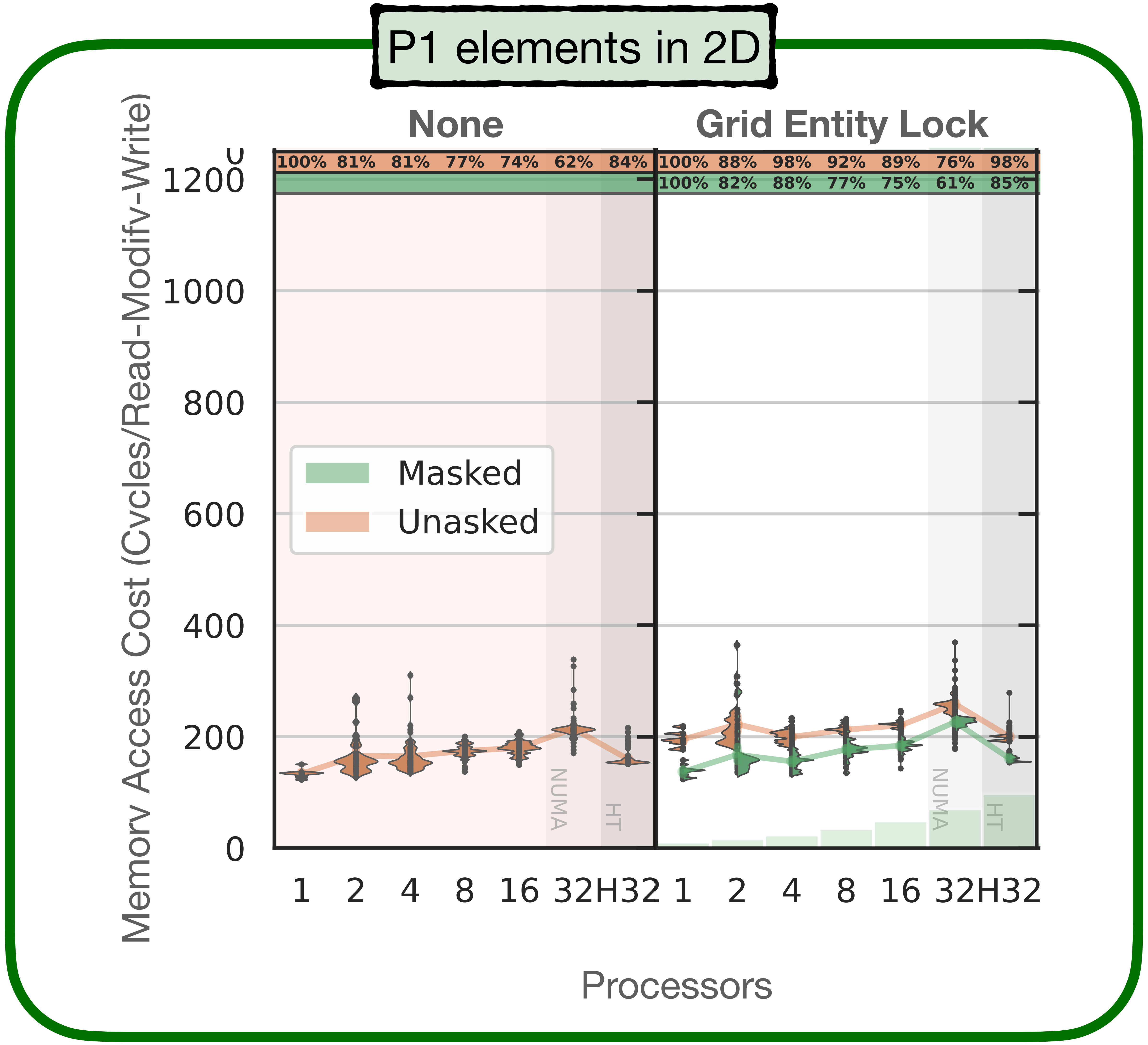
Locking algorithm: Spin Lock

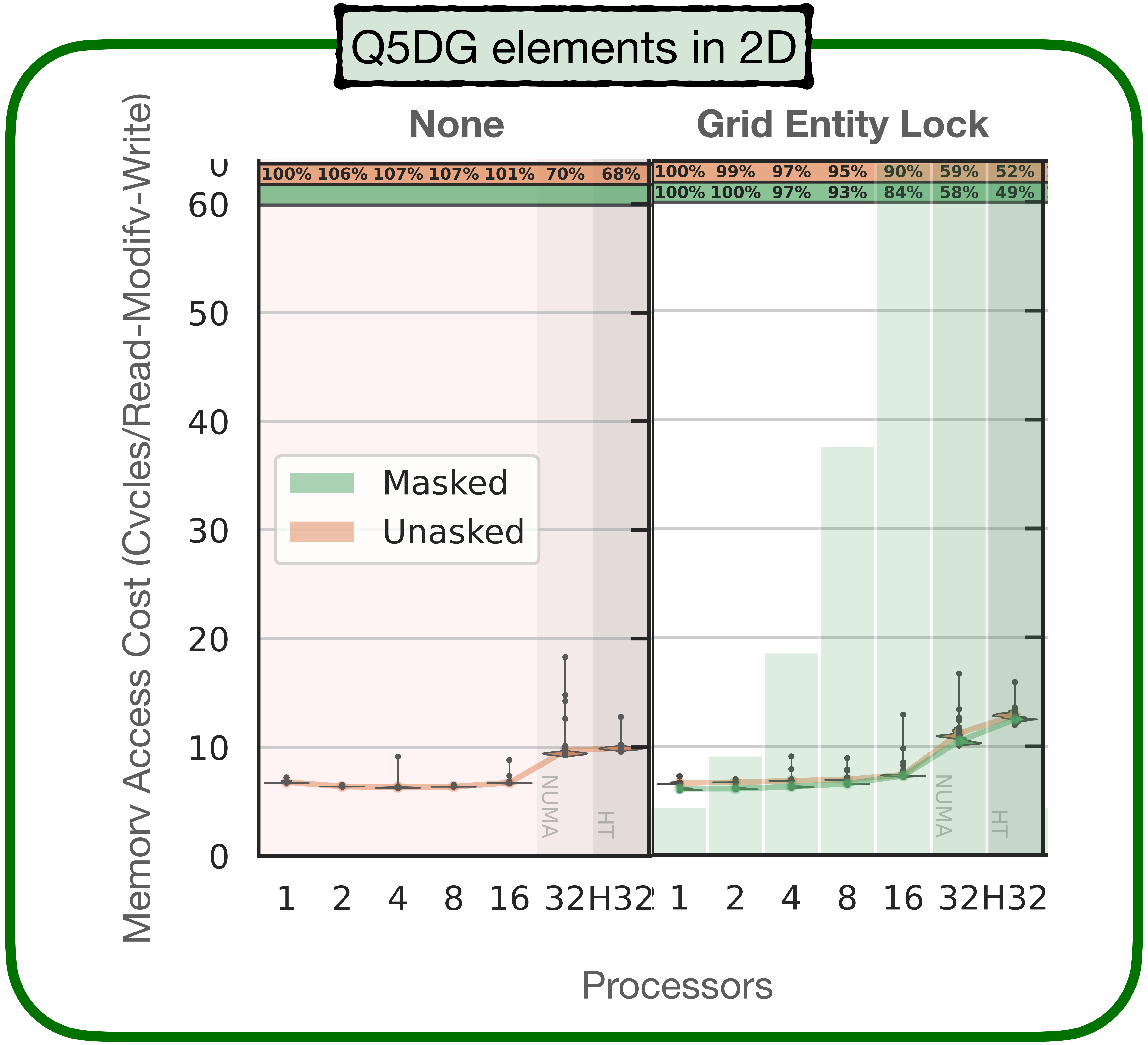



```
class LocalSpace:
    def lock(self):
        # spin until we acquire all the padlocks
        while(not self.try_lock()):
            pass
```

Fine-Grained Locks

Grid Entity Locks





Shared Memory vs Private Memory

Same task, different spatial modes to access memory

Benchmark of a more realistic HPC case

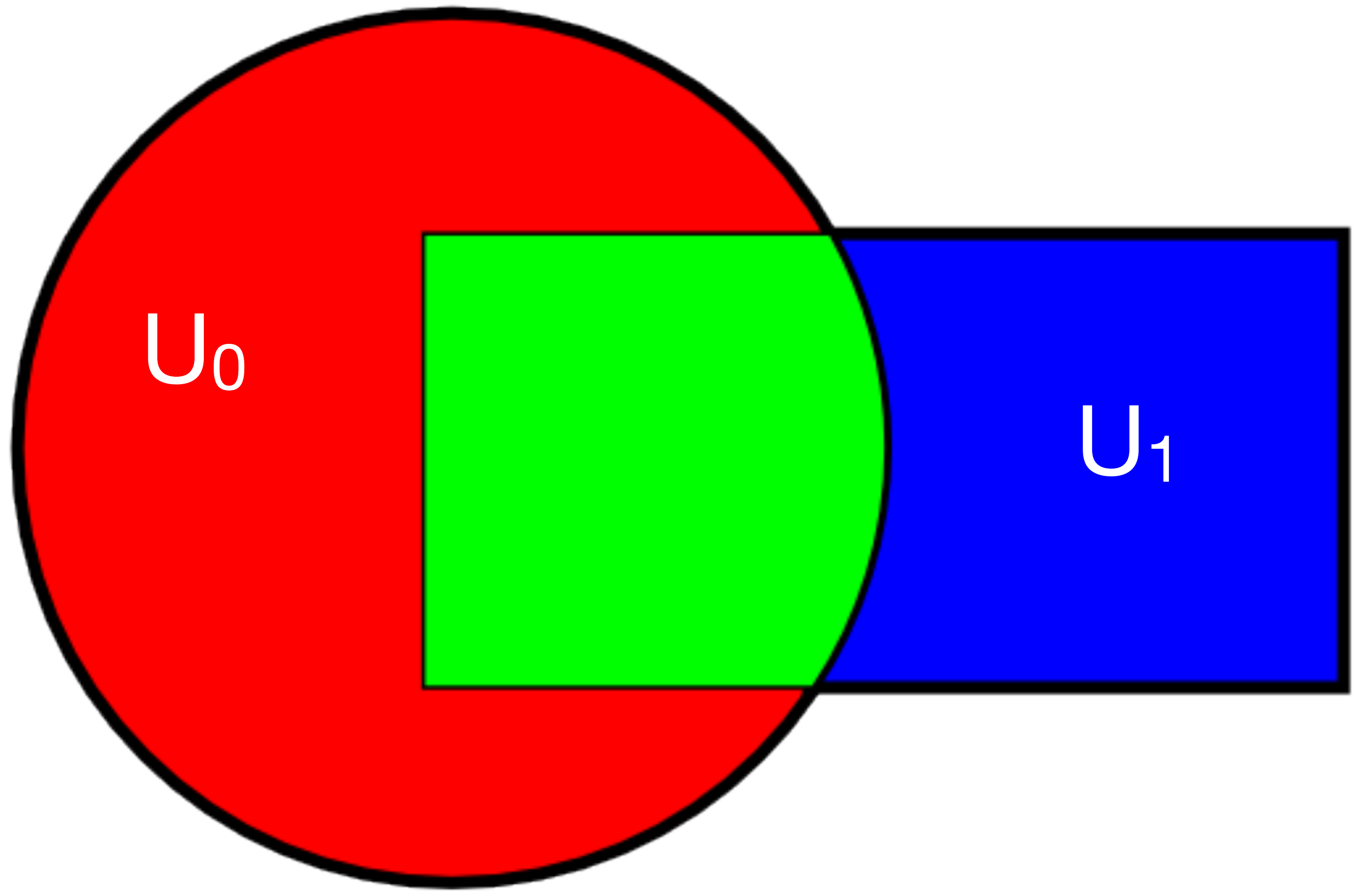
VectorBitSpinLock+TBB+Unmasked

- Reaction-Diffusion Equation
  - Structured grid in 3D
  - Discontinuous Galerkin with Interior Penalty
  - Assembly of a Residual Operator (Representative of Matrix-Free workload)
- AMD EPYC 7713 Milan
  - 64 Cores, 1 Socket
  - 1 Numa Node Per Socket (NPS=1)
- SIMD Vectorized Kernel
  - ~60% of Peak Performance
  - ~15 Arithmetic Intensity

Shared Memory vs Private Memory

How to measure throughput?

- **Issue**: Private and Shared memory approaches may not need the same amount of DOFs to solve the *same* problem.



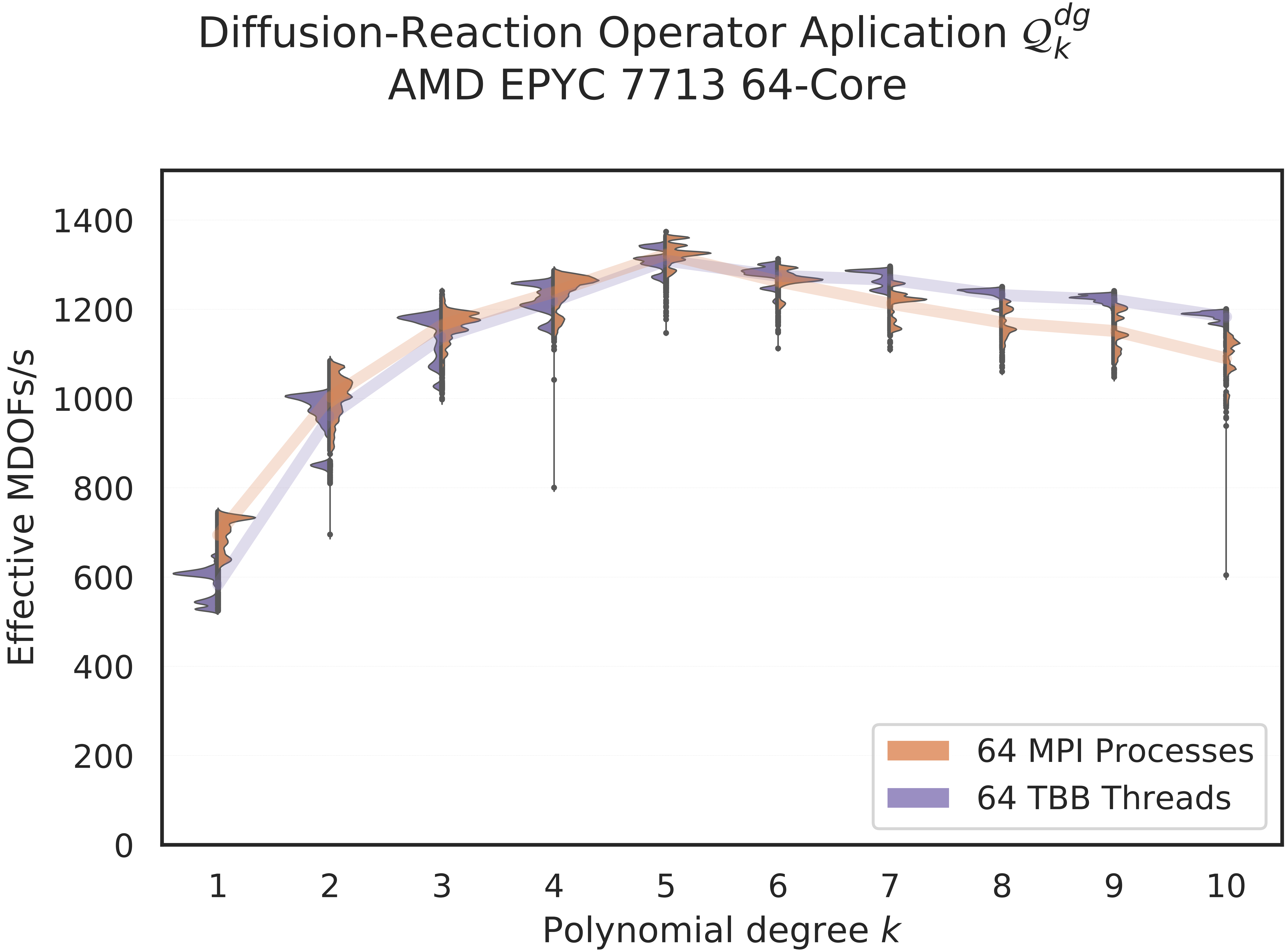
Total Degrees of Freedom

$$\overbrace{DOFs}^{\text{Total Degrees of Freedom}} := \sum_{p=0}^{P} dim(U_p)$$

$$\underbrace{EDOFs}_{\text{Effective Degrees of Freedom}} := dim\left(\cup_{p=0}^{P} U_p \right)$$

Effective Degrees of Freedom

Shared Memory vs Private Memory



Conclusions

- Entity level mutual exclusive locks are robust and scalable for Finite Elements.
- Shared region on grid partitions can amortize synchronization costs effectively.
- TBB work stealing can hides latency and unbalance issues on high core counts.

Thanks for your Attention

Question?