Scalable Domain-decomposed Monte Carlo Neutral Transport for Nuclear Fusion

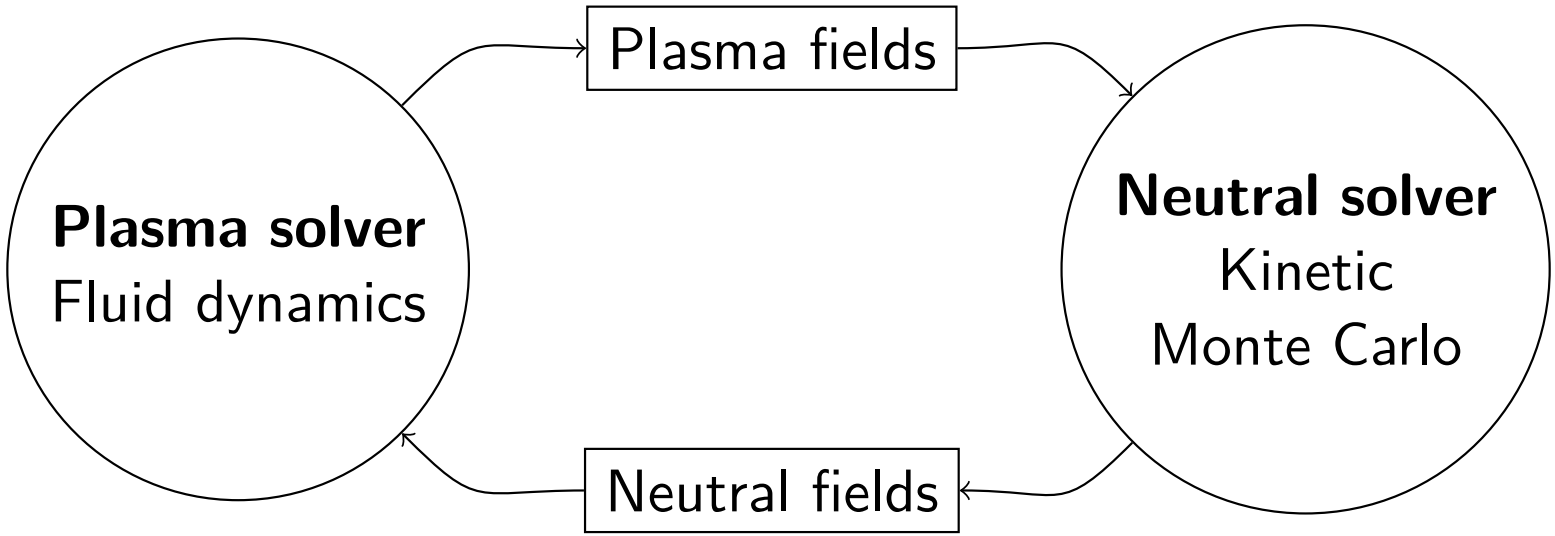
Oskar Lappi
ACH-VTT

Nov 2025

Presentation Outline

1. EIRENE needs domain-decomposed Monte Carlo (DDMC)
2. Monte Carlo transport in Eiron (a new MC code)
3. EIRENE's algorithms
4. Eiron's DDMC algorithm visually explained
5. Strong and weak scaling results

Fusion plasmas in the edge and scrape-off-layer (SOL)



Simulating the edge and SOL regions is done with a fluid-kinetic model

- Plasma has a short mean free path (mfp), fluid model valid
- Neutral gas surrounding it has a long mfp, fluid model invalid

---

- Kinetic Monte Carlo (neutrals) is more complex than CFD (plasma)
- EIRENE is the kinetic neutral solver of choice in EUROfusion

EIRENE has issues with large grids

"The use of EIRENE as a neutrals solver is also **possible** *in 3D but* **only at low resolutions** *due to technical* **limitations** *associated* **with memory requirements**, *making it incompatible for the moment with turbulent simulations."*

V. Quadri et al. "Edge plasma turbulence simulations in detached regimes with the SOLEDGE3X code". In: Nuclear Materials and Energy 41 (2024), Section 2.3

"For **large 3D grids** *used for transport calculations (e.g. EMC3-EIRENE) on large machines or for turbulent calculations on present day devices* **the memory requirements** *to store the 100 input and output tallies of EIRENE become* **large enough** *so as to* **preclude running with one MPI process per core on a node**."

D.V. Borodin et al. "Fluid, kinetic and hybrid approaches for neutral and trace ion edge transport modelling in fusion devices". In: Nuclear Fusion 62.8 (Aug. 1, 2022), Section 3.

Why? Because it is not domain-decomposed

"Overcoming this limitation would **require** *implementing a* **domain decomposition** *scheme in EIRENE and/or coarsening the grid used for neutrals, which is currently identical to the plasma grid."*

V. Quadri et al. "Edge plasma turbulence simulations in detached regimes with the SOLEDGE3X code". In: Nuclear Materials and Energy 41 (2024), Section 2.3

"On the longer term, further relaxing memory constraints will **require** *implementing a* **domain decomposition** *approach, with nodes being allocated chunks of the full grid, and particles distributed among threads on the node"*

D.V. Borodin et al. "Fluid, kinetic and hybrid approaches for neutral and trace ion edge transport modelling in fusion devices". In: Nuclear Fusion 62.8 (Aug. 1, 2022), Section 3.

Lack of domain decomposition holding back EIRENE

Issues

- The grid must fit inside the memory on one compute node
- Performance slows down without DD, worse cache efficiency

Performance improvements in the plasma codes don't matter when EIRENE takes 90-99% of the runtime [1].

[1] EIRENE uses ">90% of computing time in ITER mean-field simulations (e.g., >99% in SOLEDGE3X-EIRENE)", Patrick Tamain, TSVV-5 presentation slides

Eiron, a reduced EIRENE for domain decomposition

Domain decomposition is a big change

Domain decomposition is quite complicated, so we did not implement it straight into EIRENE.

Eiron is a new open source C++ Monte Carlo code built for testing domain decomposition and other new schemes[3][4].

[3] Oskar Lappi et al. *Scalable Domain-decomposed Monte Carlo Neutral Transport for Nuclear Fusion*. Nov. 6, 2025. arXiv: 2511.04489

[4] Oskar Lappi et al. *2D implementation of Kinetic-diffusion Monte Carlo in Eiron*. Sept. 23, 2025. arXiv: 2509.19140

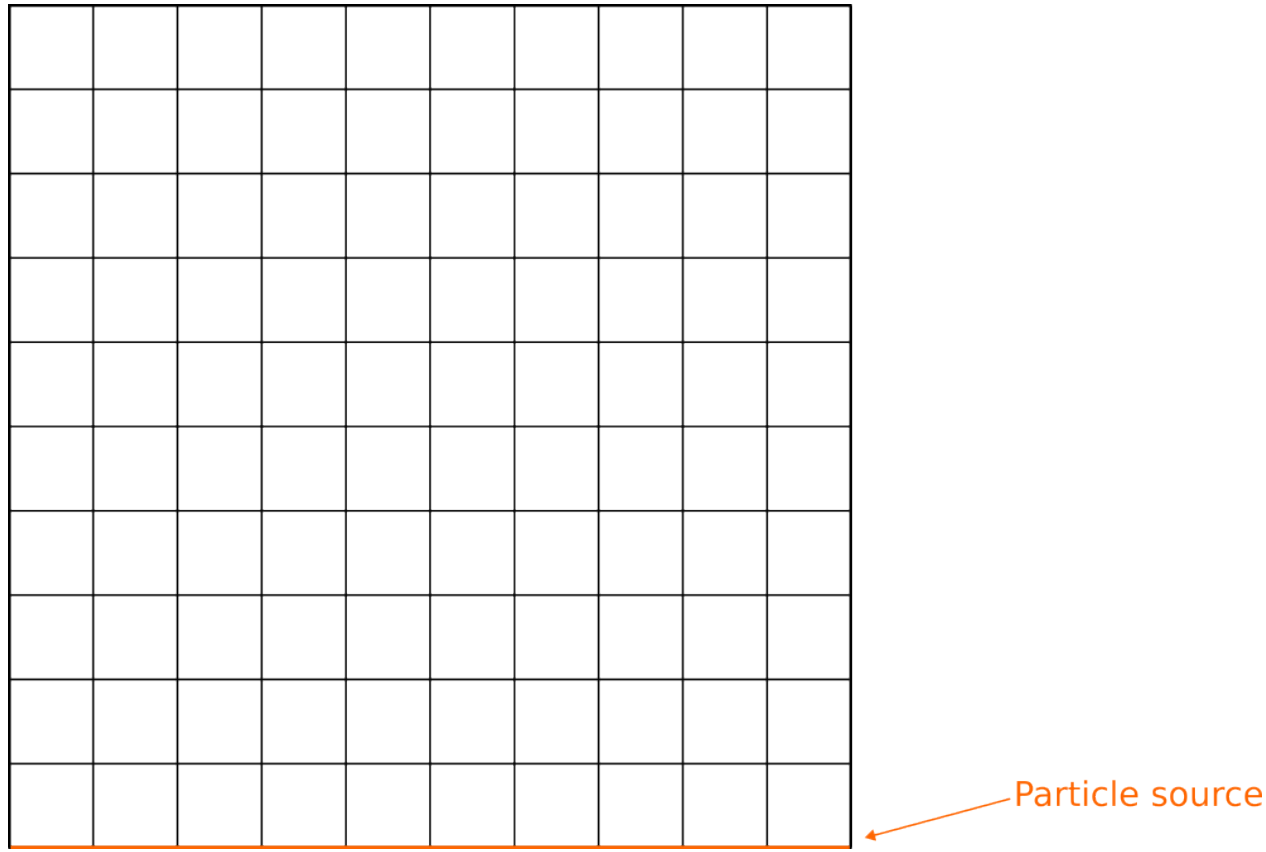
Monte Carlo transport in Eiron

Monte Carlo transport

0.5	1.5	0.4	3.1	0.3	7.2	0.4	0.1	0.2	0.1
2.4	0.6	0.1	2.4	0.6	0.2	1.2	0.6	0.1	2.4
0.3	3.1	3.1	0.3	3.1	3.1	0.7	3.1	3.1	0.3
3.3	0.4	2.1	3.3	0.4	2.1	0.4	0.4	2.1	3.3
2.0	0.3	4.1	2.0	0.3	4.1	3.2	0.3	4.1	2.0
0.0	3.1	2.1	0.0	3.1	2.1	1.6	3.1	2.1	0.0
1.2	2.1	1.1	1.2	2.1	1.1	0.9	2.1	1.1	1.2
1.6	0.4	3.1	1.6	0.4	3.1	0.3	0.4	3.1	1.6
5.0	1.1	0.1	5.0	1.1	0.2	1.2	1.1	0.3	5.0
2.1	5.1	0.1	2.1	5.4	1.6	0.5	5.2	0.1	2.1

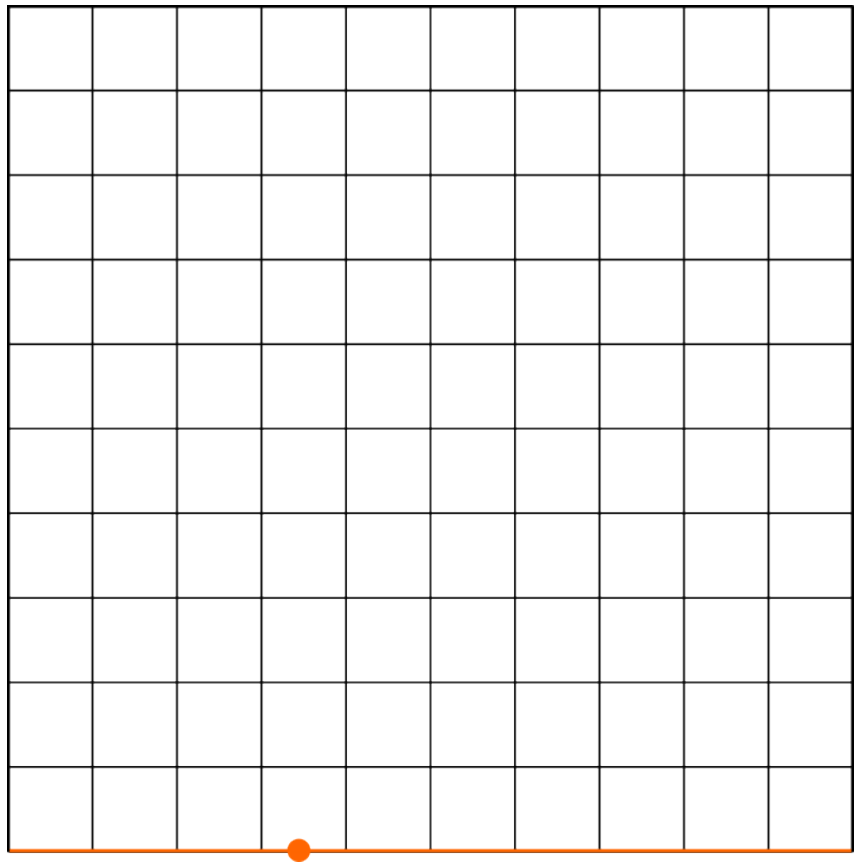
In Eiron, we're working with 2D, square-celled grids. There are static plasma fields stored on the grid.

Monte Carlo transport



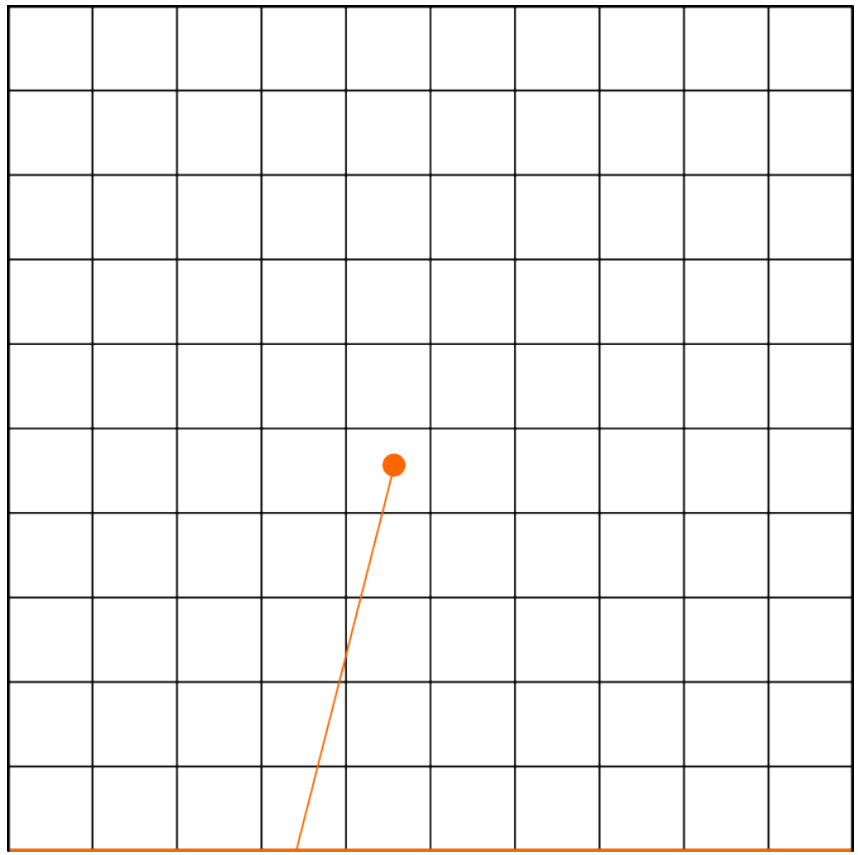
There are particle sources which can be placed in the space.
This is a wall source, "volume sources" in 2D space also possible.

Monte Carlo transport



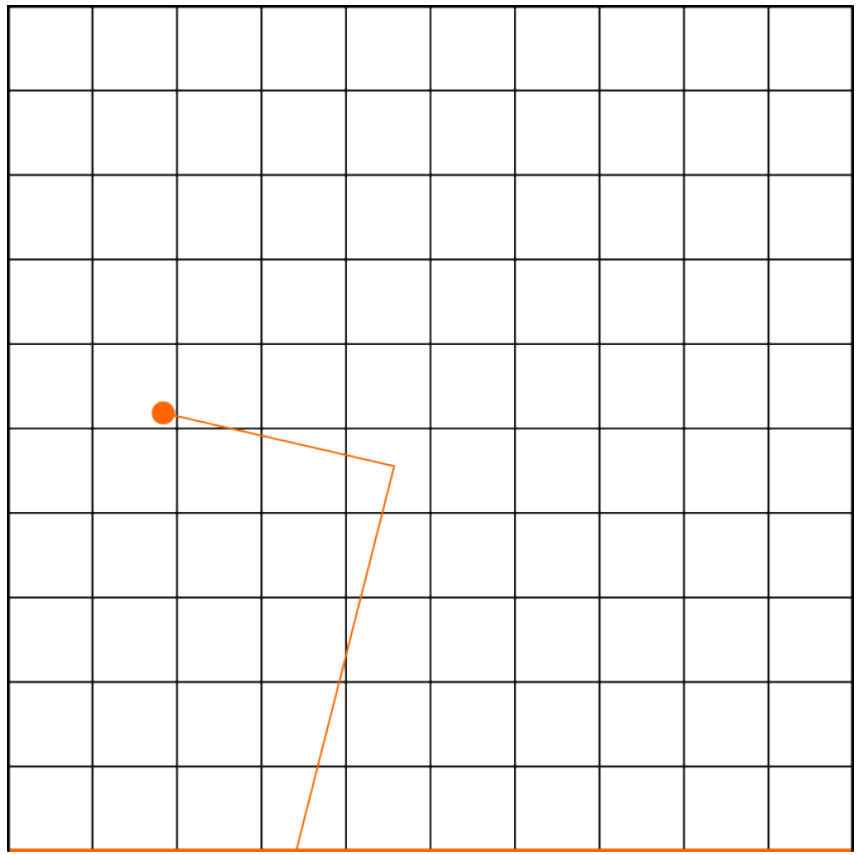
Neutral particles are randomly generated at the sources.

Monte Carlo transport



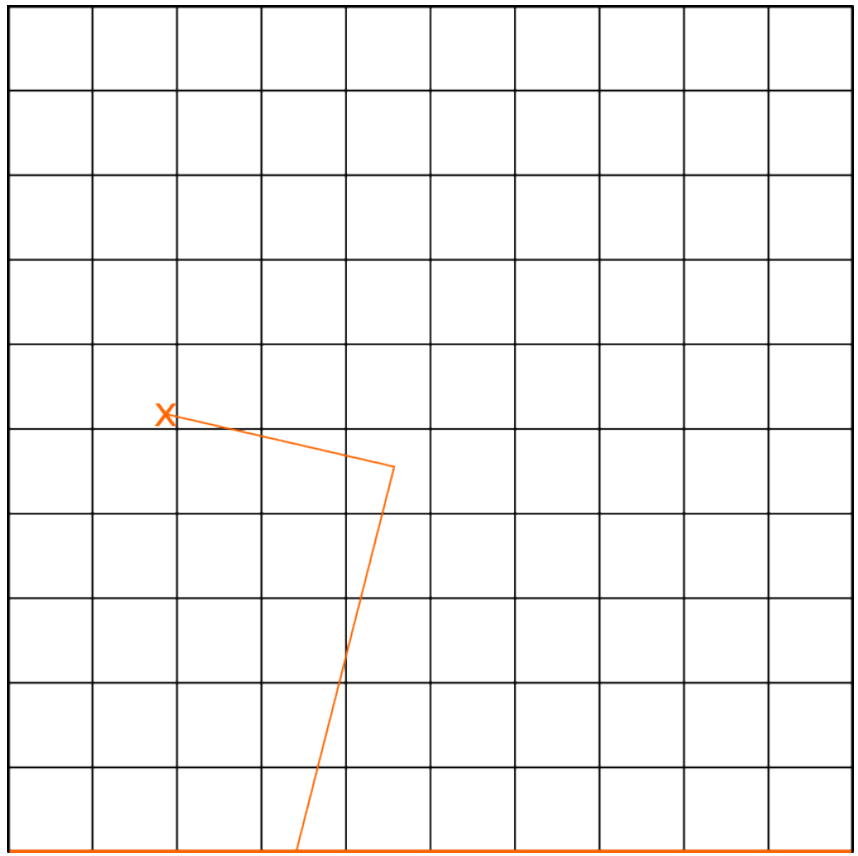
We integrate the *total macroscopic cross section* Σ_t along the particle's linear trajectory until the integral is equal to a sampled value X, and a collision occurs.

Monte Carlo transport



After *scattering* collisions, particles continue with a new velocity.

Monte Carlo transport



After *absorption* events, particles are terminated.

Parallel algorithms in Eiron

Domain replication (DR)

Traditional way to parallelize Monte Carlo.
EIRENE's MPI parallellization.
Requires a lot of memory.

Shared memory (SM)

Same as EIRENE's OpenMP parallelization.
Partially alleviates memory issue. Scales worse than DR.

Domain decomposition (DD)

Novel asynchronous domain decomposition algorithm.
Solves the memory issue.

Algorithm Domain replication (DR)

```
In an OpenMP parallel region
Create thread-private copy of estimation grid
for each source in sources do
    for particles in source, in parallel over threads do
        Simulate particle, estimate into private estimation grid
Reduce estimation grids
```

Memory usage grows

Memory usage scales linearly with grid resolution $\times$ #CPU cores.

Grid res. or #CPU cores grows $\implies$ worse resource allocation

At some point, simulations become too big to run.

Algorithm Shared memory (SM)

```
In an OpenMP parallel region
for each source in sources do
    for particles in source, in parallel over threads do
        Simulate particle, estimate into shared estimation grid
```

Less memory usage, but memory usage still grows

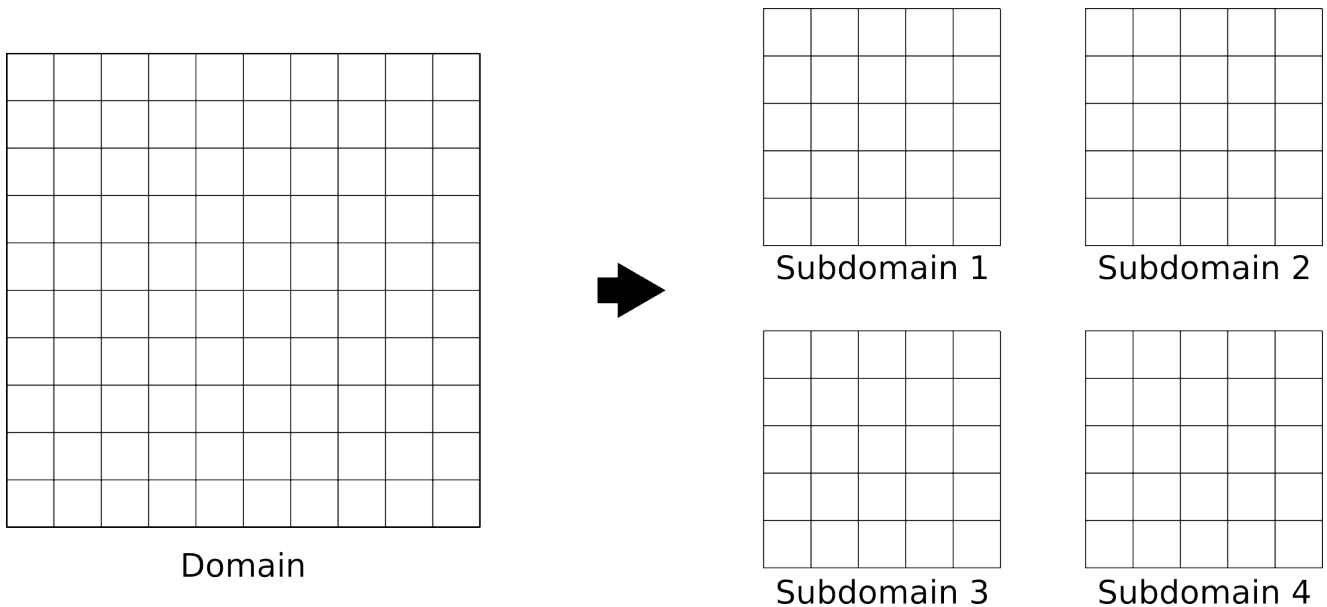
Memory usage scales linearly with grid resolution. Adding CPU cores on a single node does not increase memory usage.

However, shared writeable grid $\implies$ *cache thrashing*.

At some point, simulations still become too big to run.

Domain decomposition (DD)

Domain decomposition is what you reach for when your simulation space is too big to efficiently simulate in a single process.



Eiron's domain-decomposed Monte Carlo algorithm is fully asynchronous. As far as we know it's the first fully asynchronous algorithm.

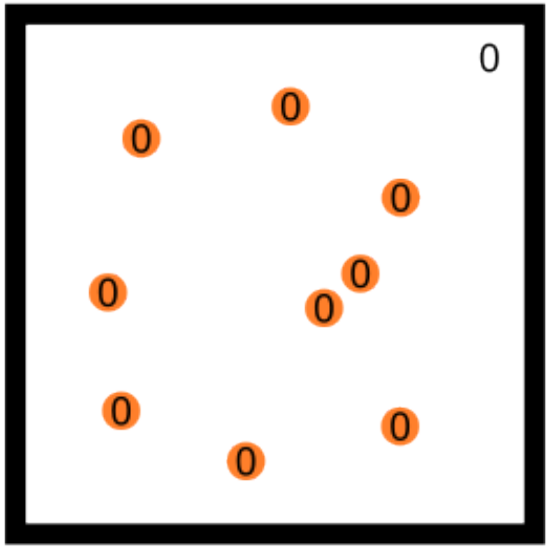
DDMC algorithm visualized example

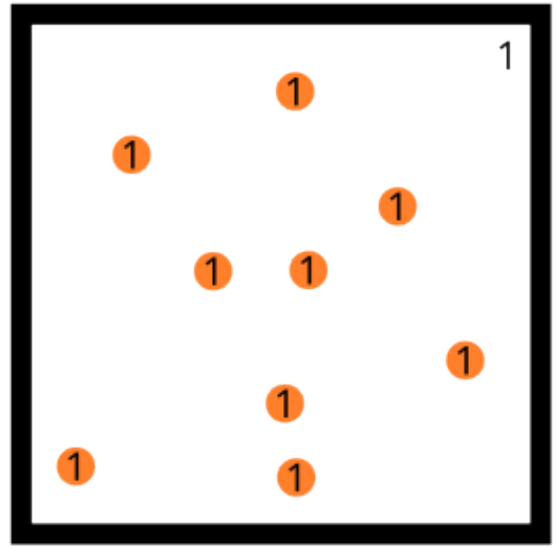
4 MPI ranks, $\texttt{BUFFER\_SIZE} = 4$, $\texttt{SEND\_PERIOD} = 4$

iteration=?

State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:0	1:0
2:0	3:0



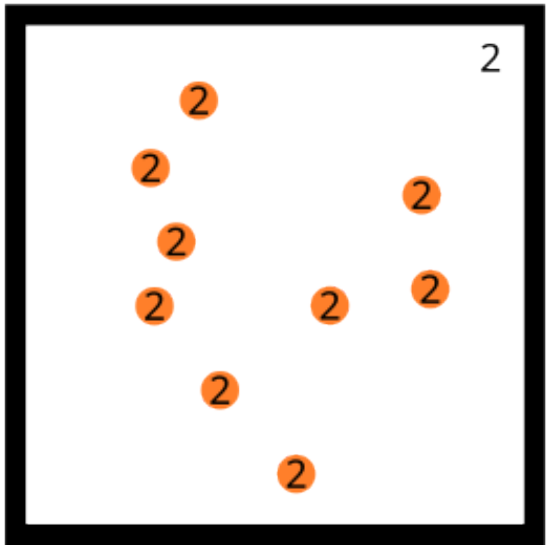


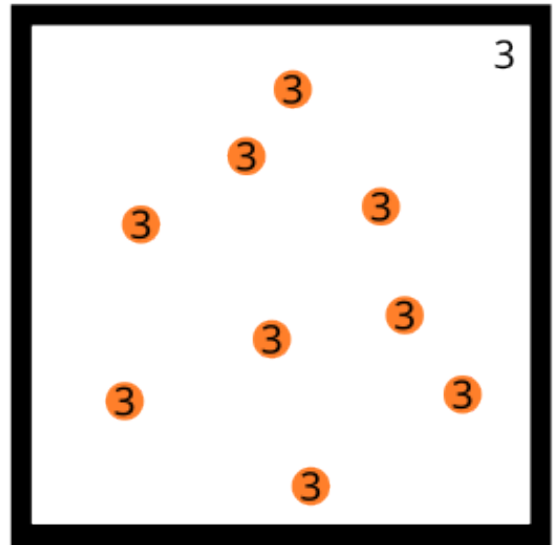
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?

State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?



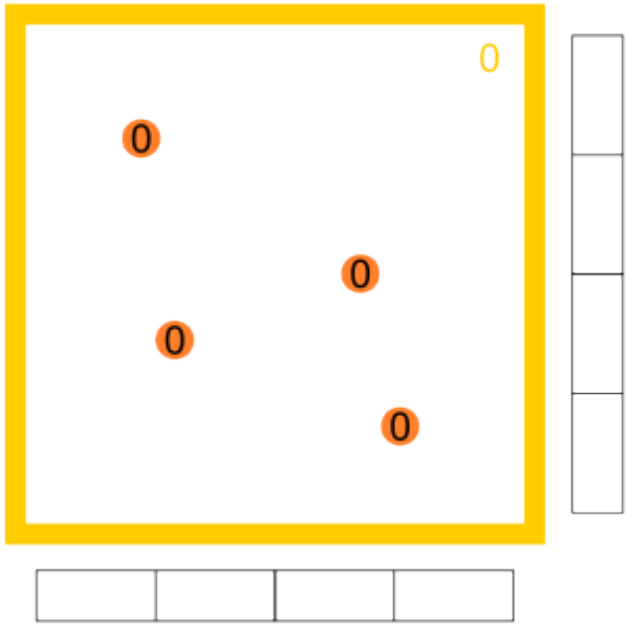


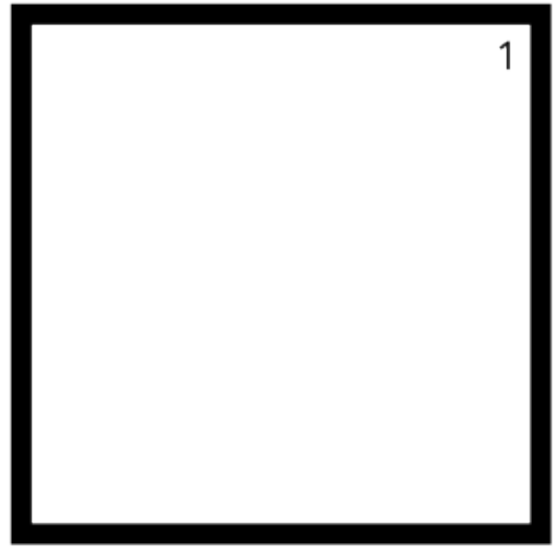
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?

iteration=1

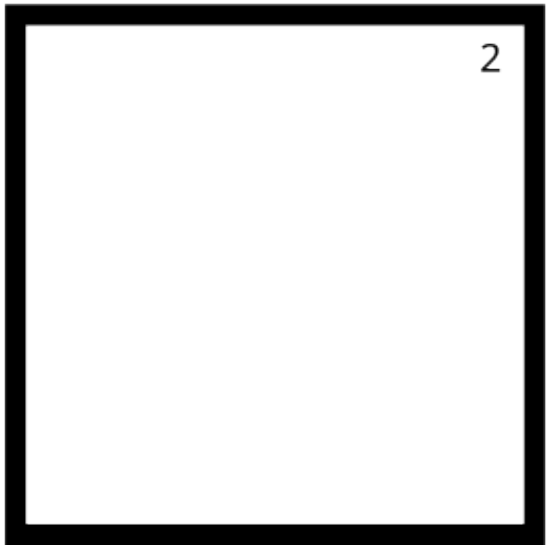
State	
live_parts: 4	
live_ranks: 0,1,2,3	
dead_parts:	
0:0	1:0
2:0	3:0

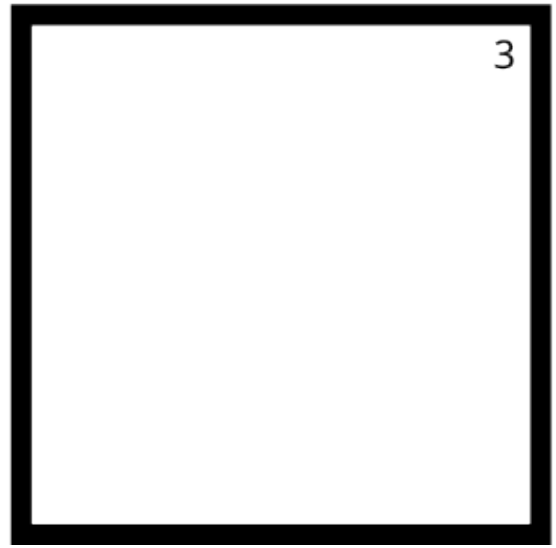




State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?



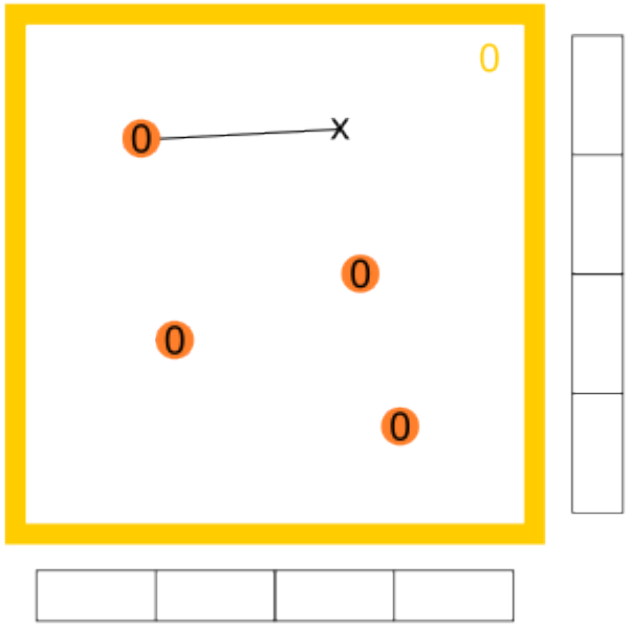


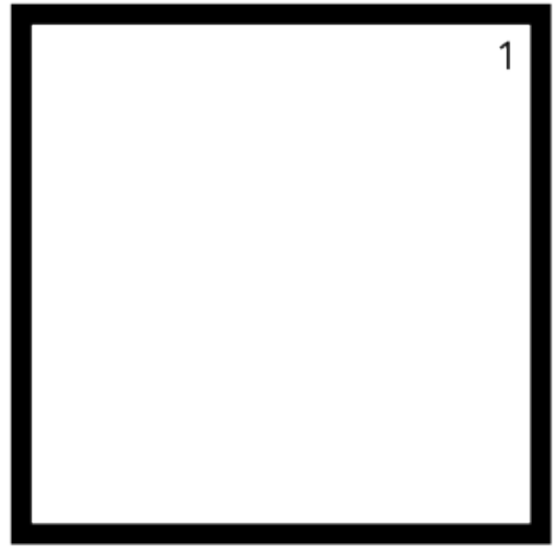
State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

iteration=1

State
live_parts: 4
live_ranks: 0,1,2,3
dead_parts:

0:1	1:0
2:0	3:0



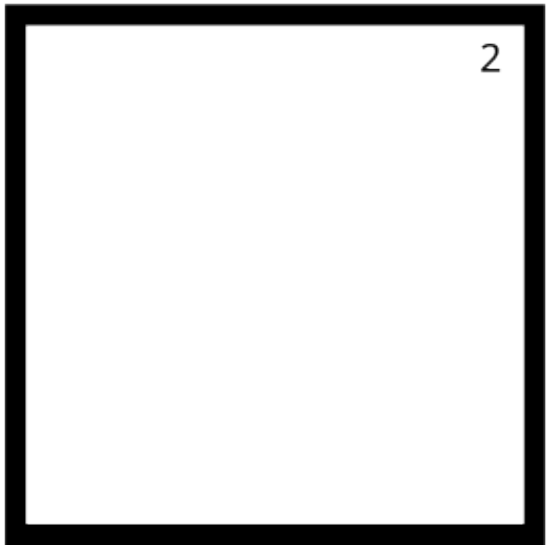


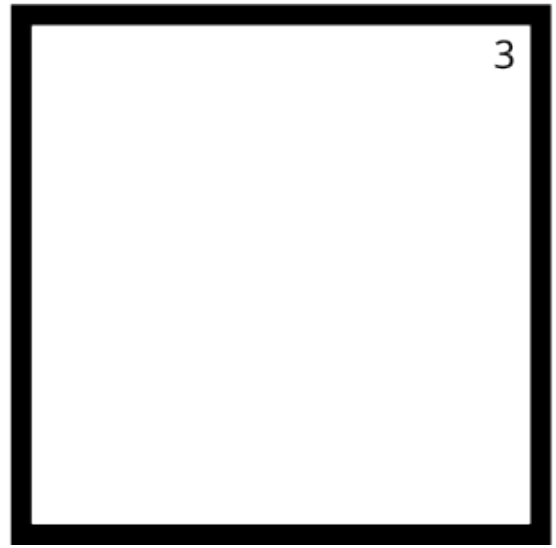
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?

State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?





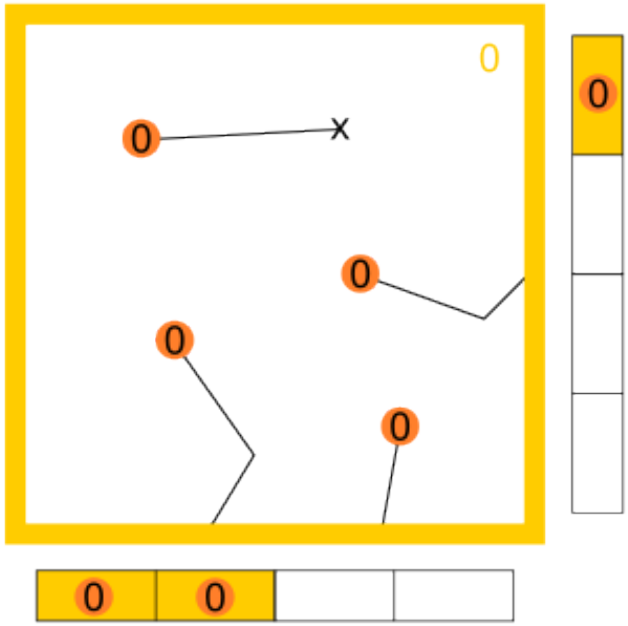
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?

iteration=1

State
live_parts: 4 live_ranks: 0,1,2,3 dead_parts:

0:1	1:0
2:0	3:0



1

State
live_parts: ? live_ranks: 0,1,2,3 dead_parts:

0:?	1:?
2:?	3:?

State
live_parts: ? live_ranks: 0,1,2,3 dead_parts:

0:?	1:?
2:?	3:?

2

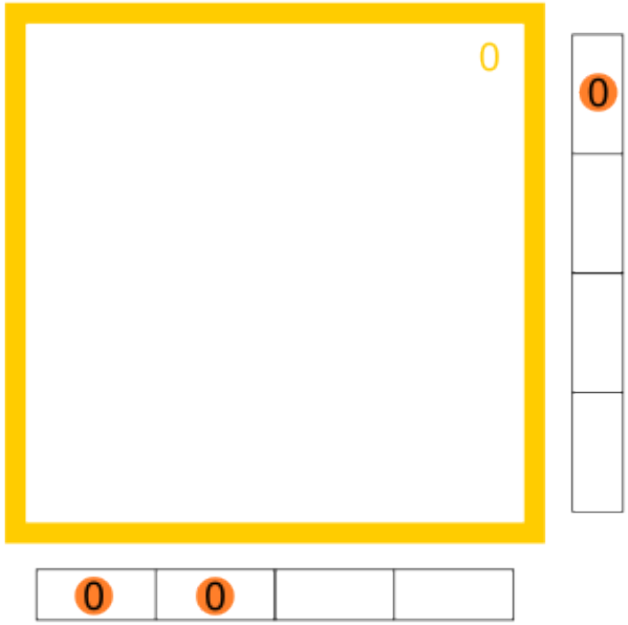
3

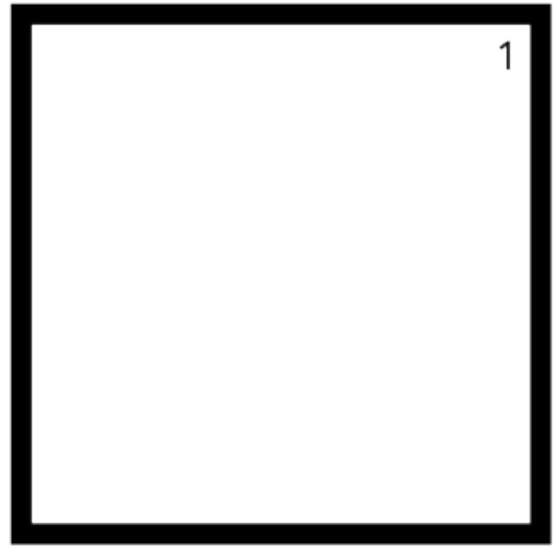
State
live_parts: ? live_ranks: 0,1,2,3 dead_parts:

0:?	1:?
2:?	3:?

iteration=1

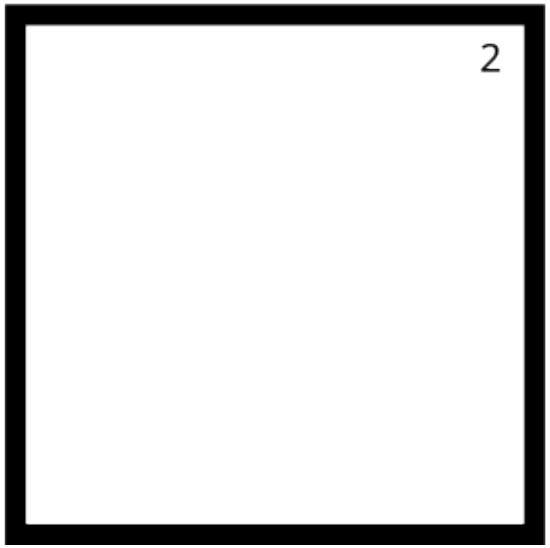
State
live_parts: 3
live_ranks: 0,1,2,3
dead_parts:
0:0 | 1:0
2:0 | 3:0

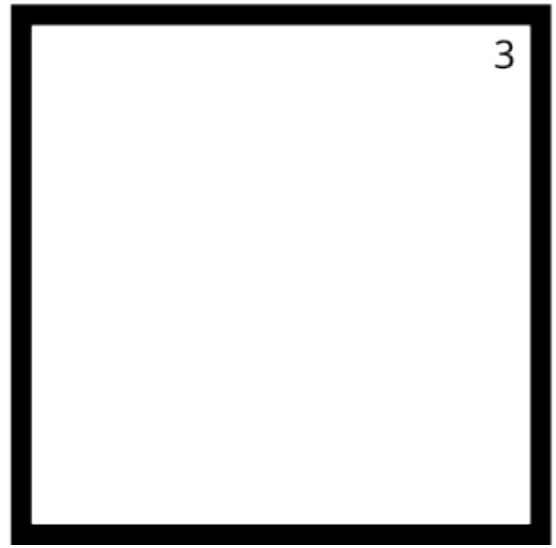




State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

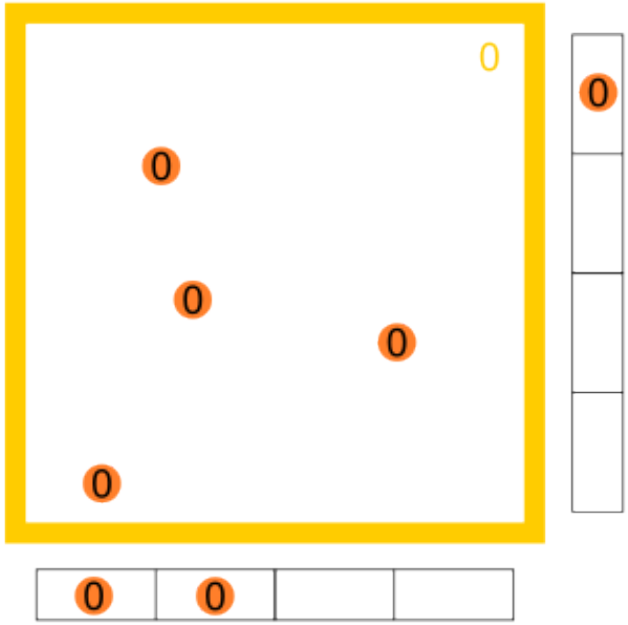


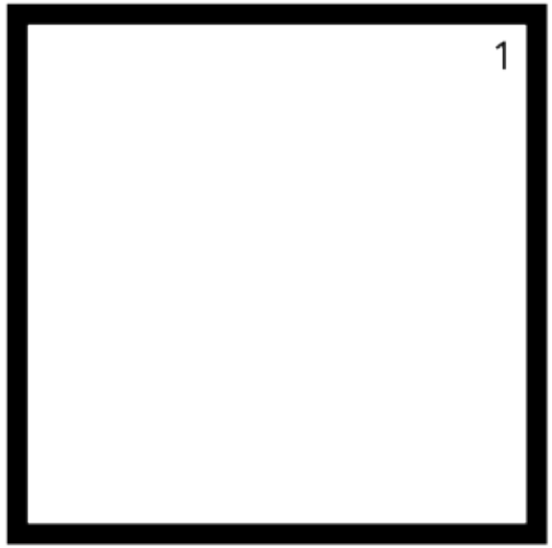


State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

iteration=2

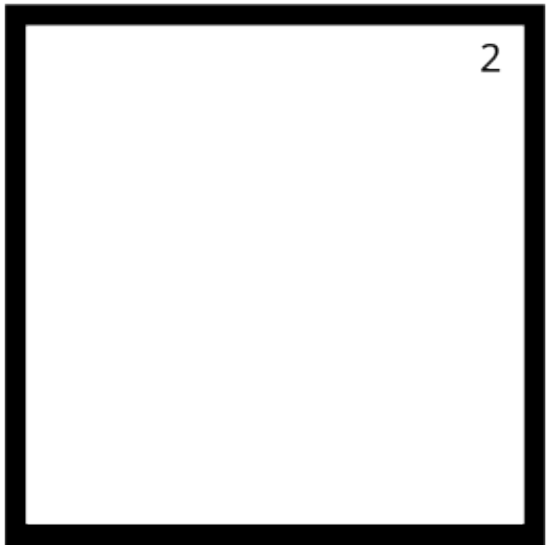
State	
live_parts: 7	
live_ranks: 0,1,2,3	
dead_parts:	
0:0	1:0
2:0	3:0

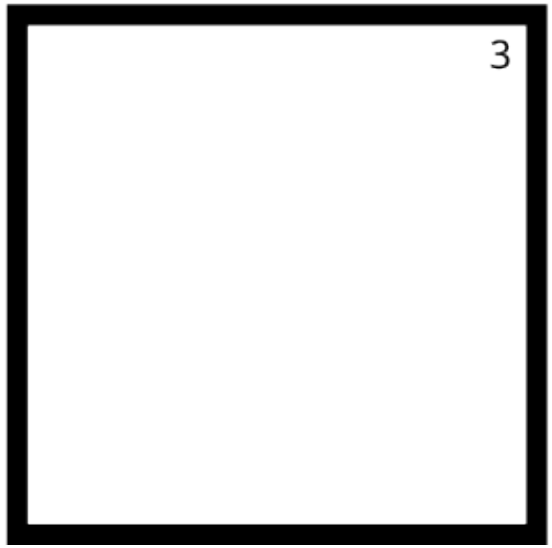




State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

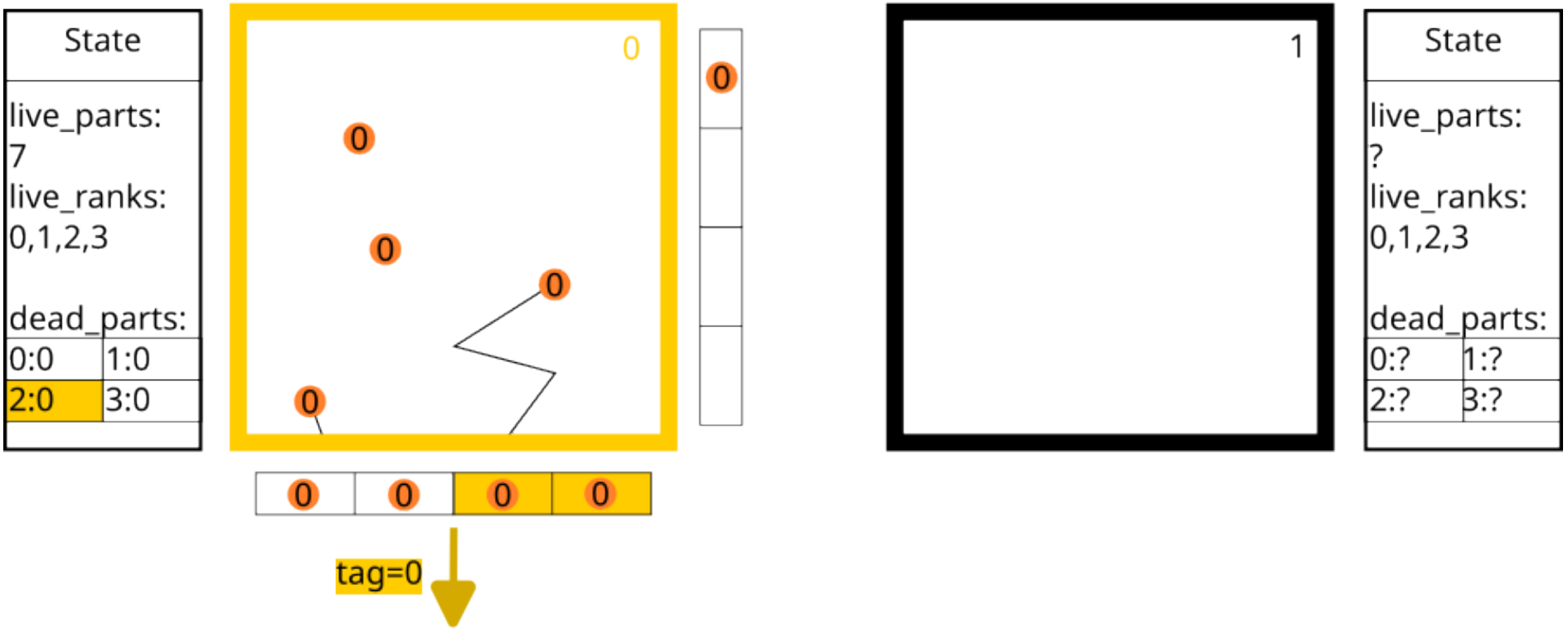
State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?





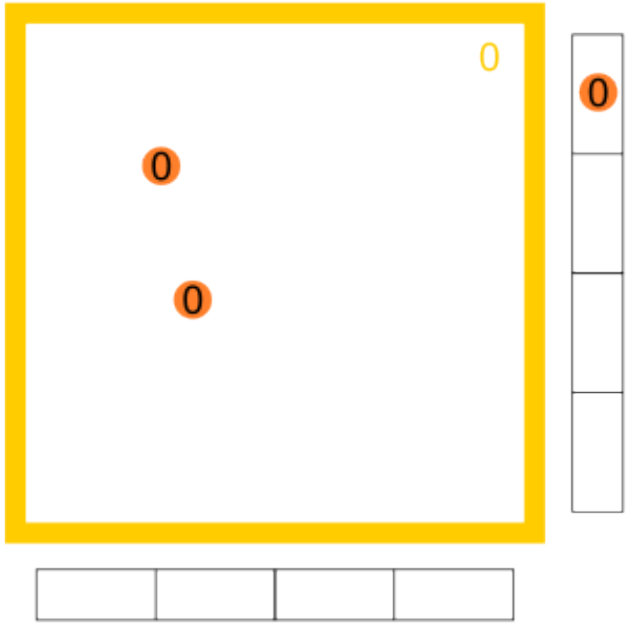
State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

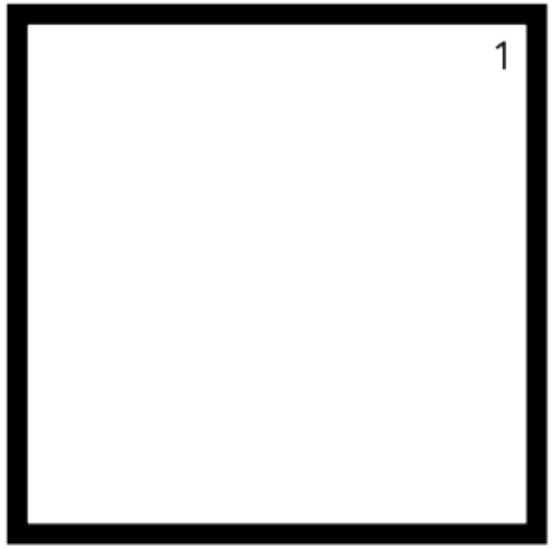
iteration=2


State
live_parts:
7
live_ranks:
0,1,2,3
dead_parts:
0:0
1:0
2:0
3:0
0
0
0
0
0
0
0
0
0
0
tag=0
1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
2
3
State
live_parts:
4
live_ranks:
0,1,2,3
dead_parts:
0:0
1:0
2:0
3:0

iteration=2

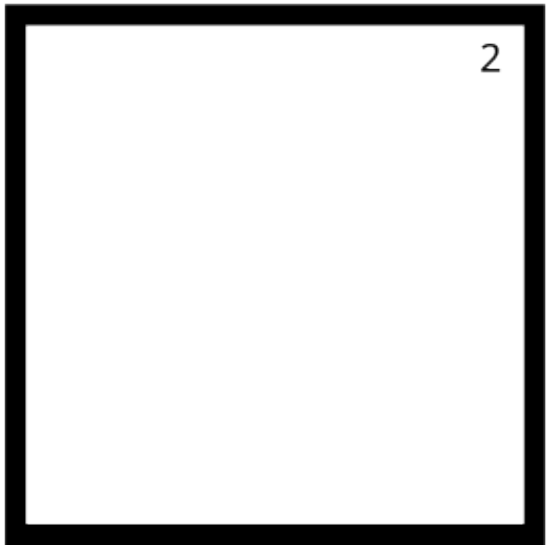
State
live_parts: 7
live_ranks: 0,1,2,3
dead_parts:
0:0 | 1:0
2:0 | 3:0

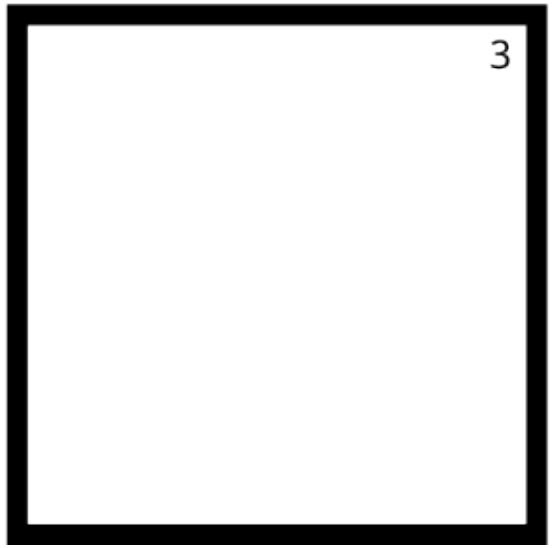




State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

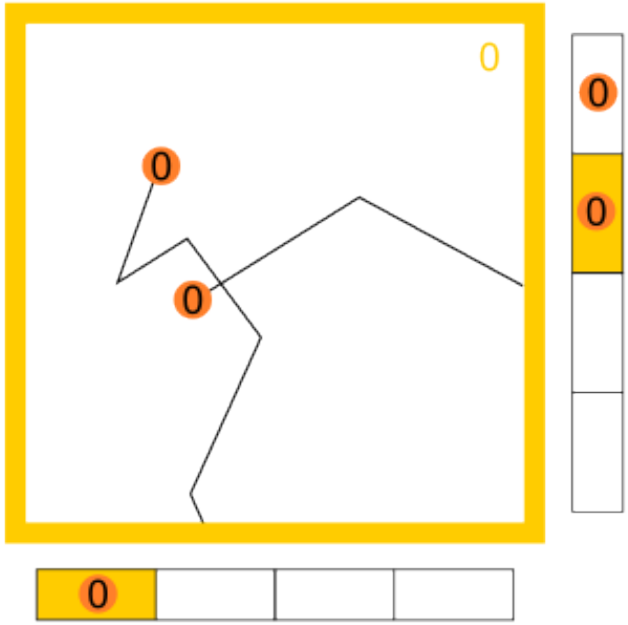


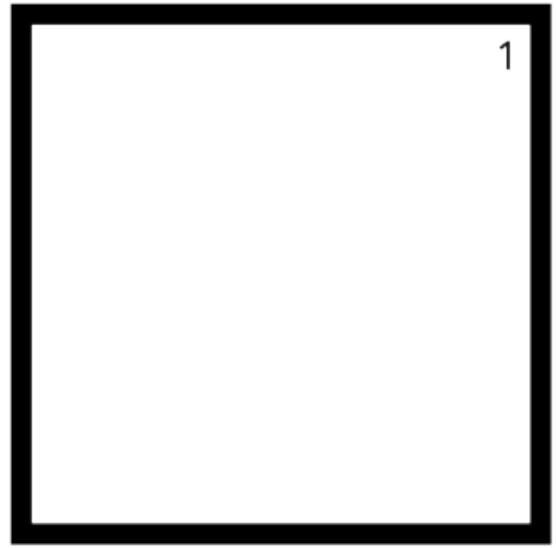


State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

iteration=2

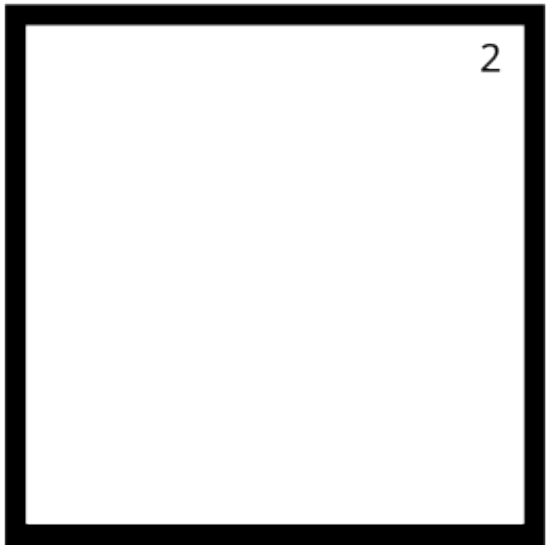
State
live_parts: 7
live_ranks: 0,1,2,3
dead_parts:
0:0 | 1:0
2:0 | 3:0

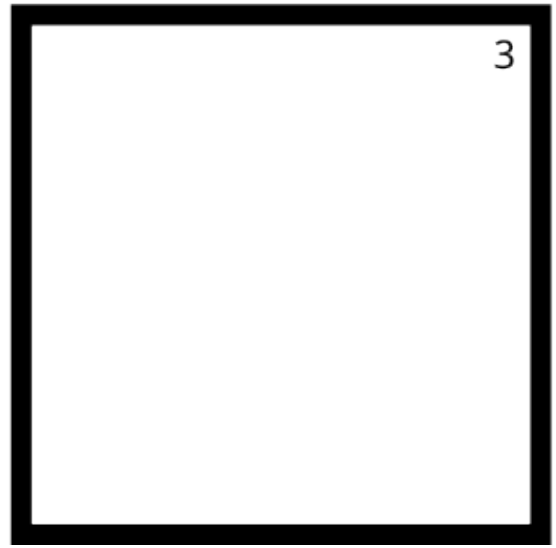




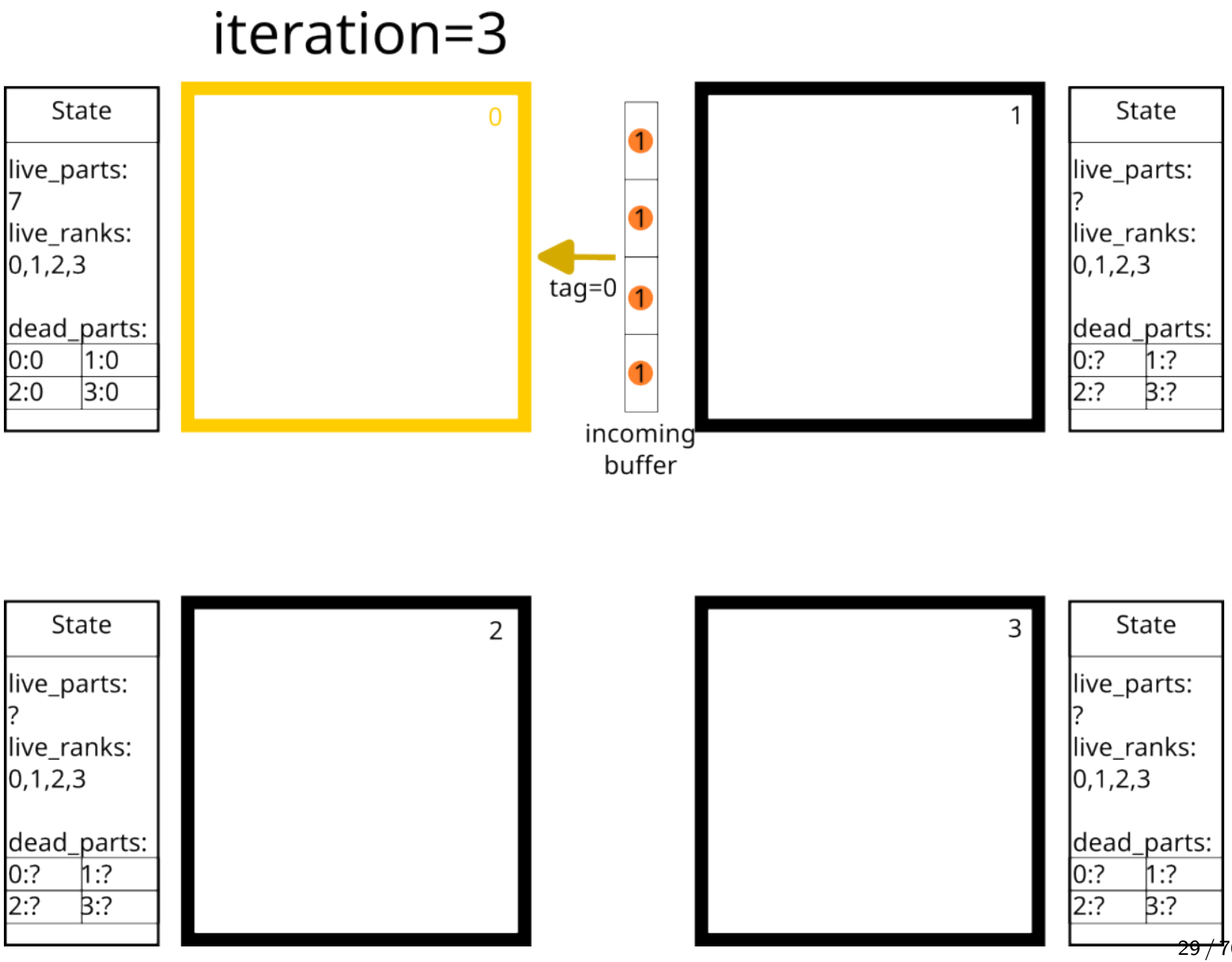
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?





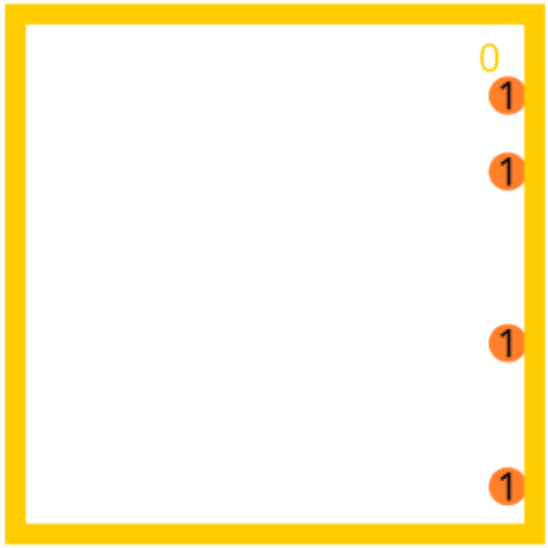
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

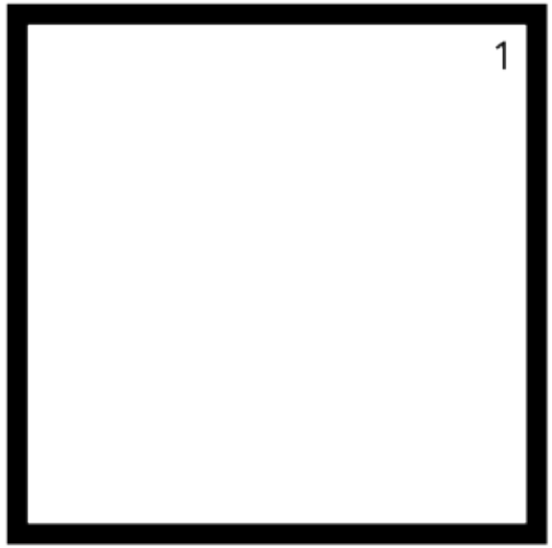

iteration=3
State
live_parts:
7
live_ranks:
0,1,2,3
dead_parts:
0:0
1:0
2:0
3:0
0
tag=0
1
1
1
1
incoming
buffer
1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
2
3
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?

iteration=3

State
live_parts: 7
live_ranks: 0,1,2,3
dead_parts:

0:0	1:0
2:0	3:0



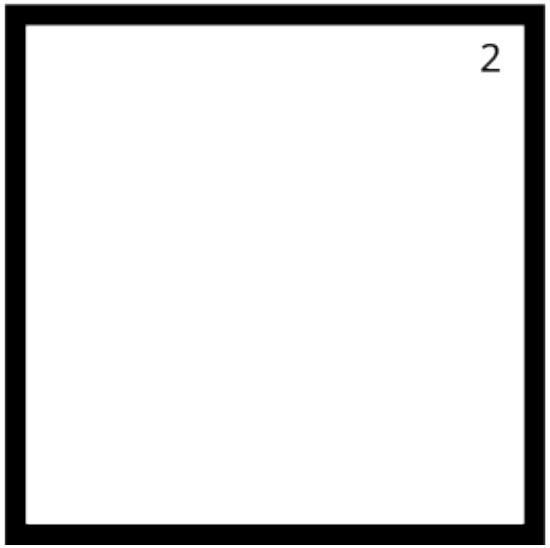


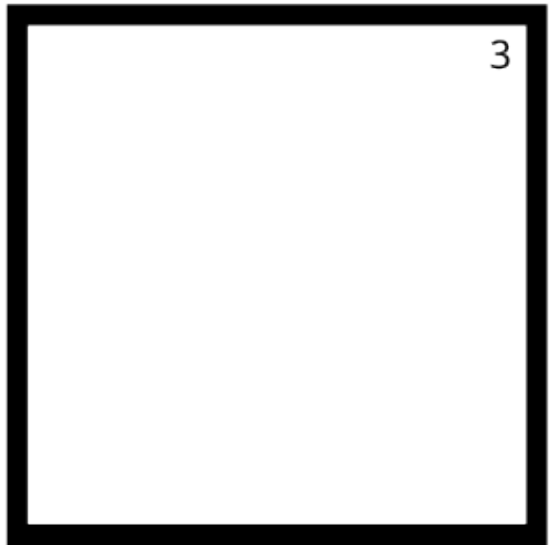
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?

State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

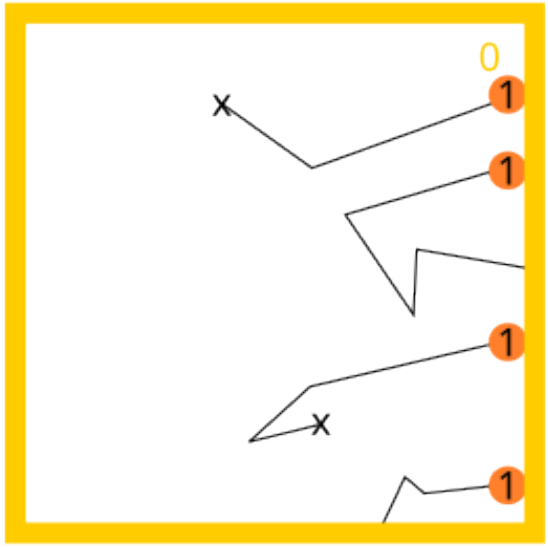
0:?	1:?
2:?	3:?

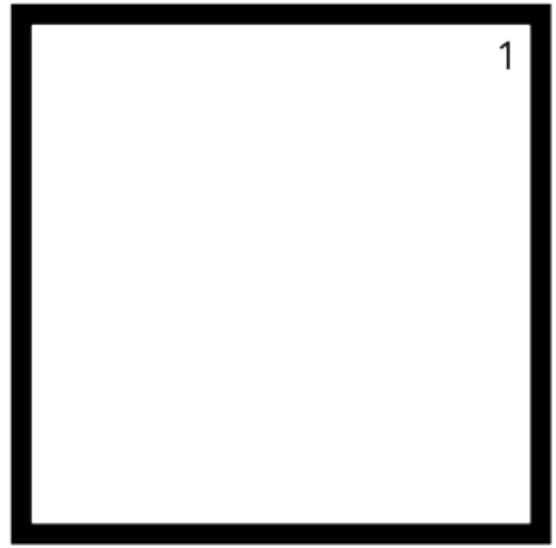




iteration=3

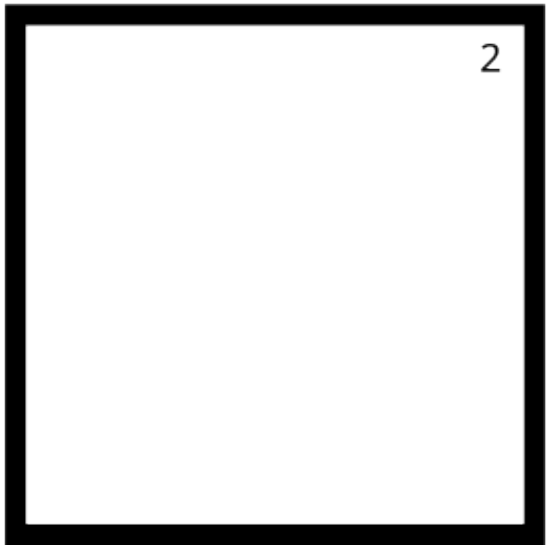
State	
live_parts: 7	
live_ranks: 0,1,2,3	
dead_parts:	
0:0	1:2
2:0	3:0

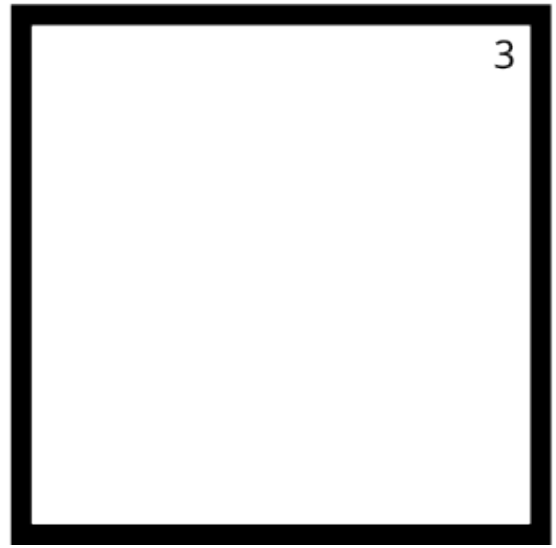




State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

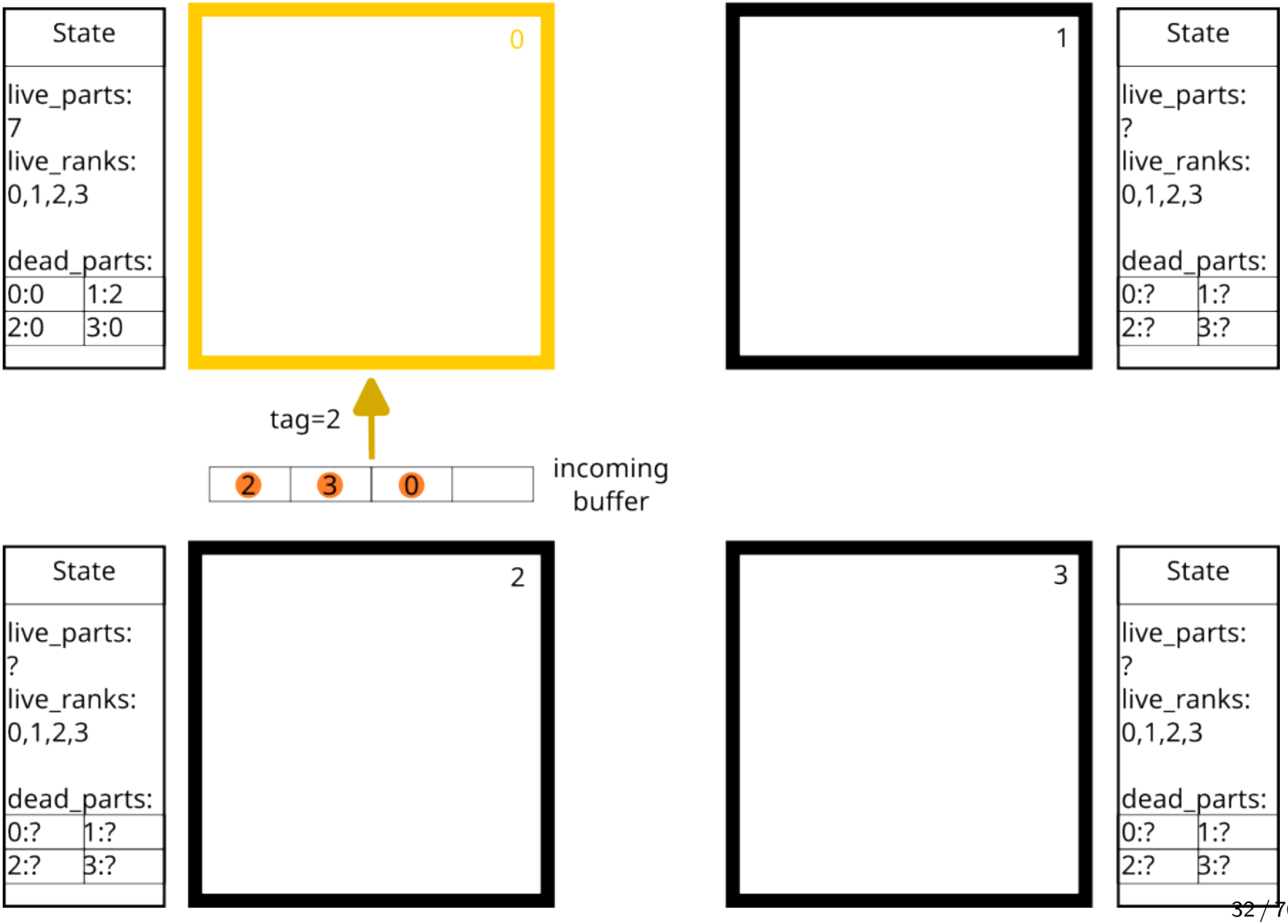
State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?



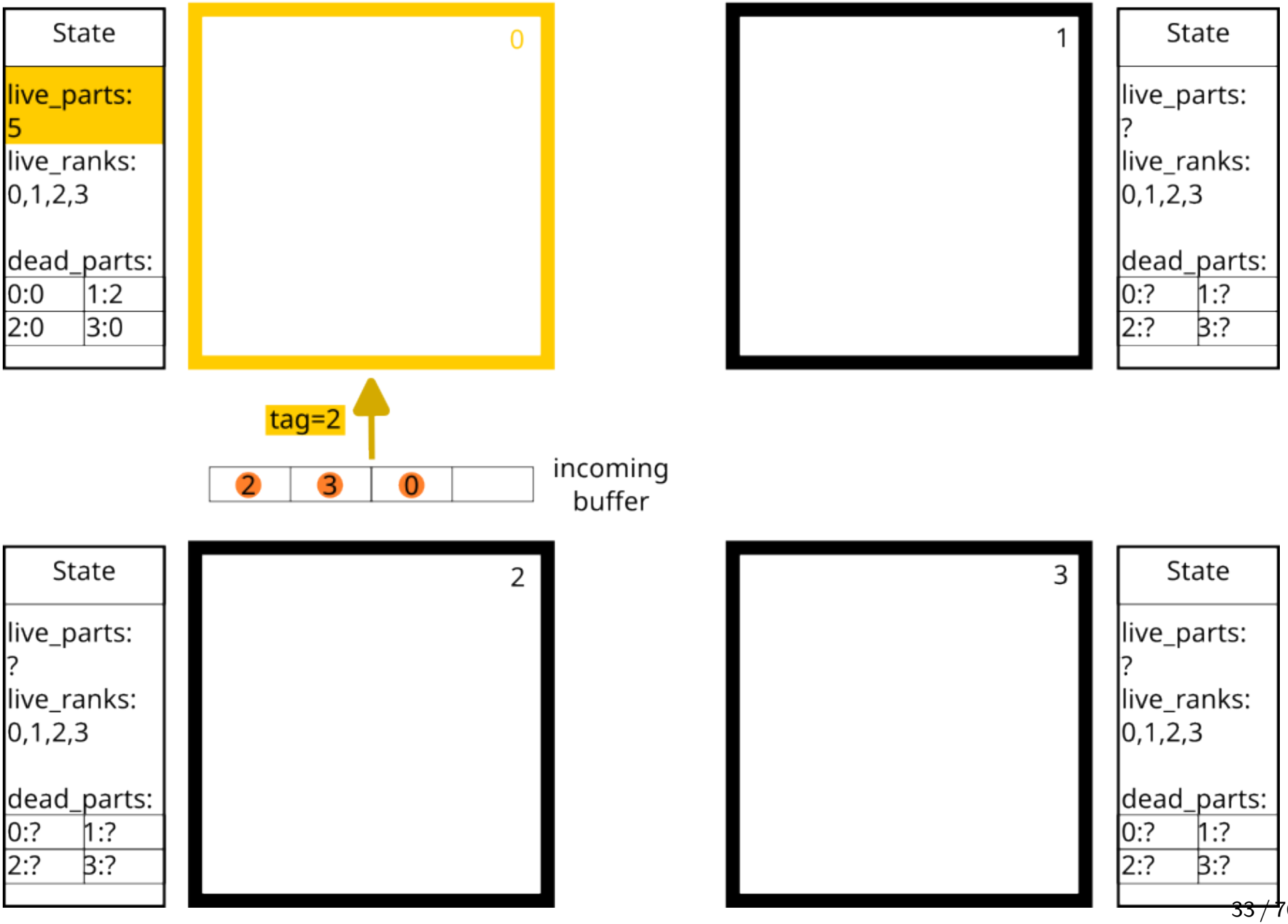


State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

iteration=4

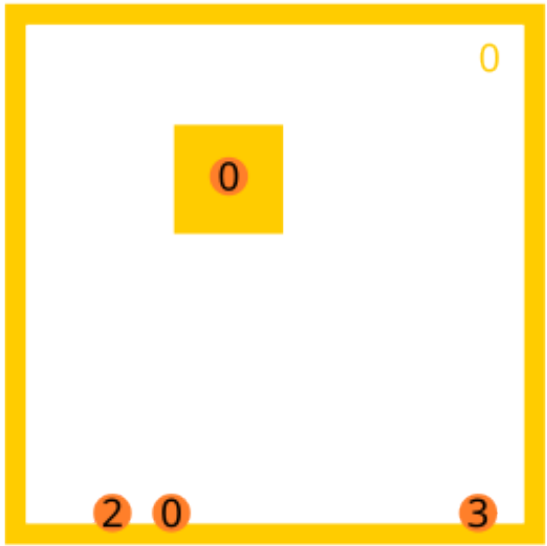

State
live_parts:
7
live_ranks:
0,1,2,3
dead_parts:
0:0
1:2
2:0
3:0
0
1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
tag=2
2
3
0
incoming
buffer
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
2
3
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?

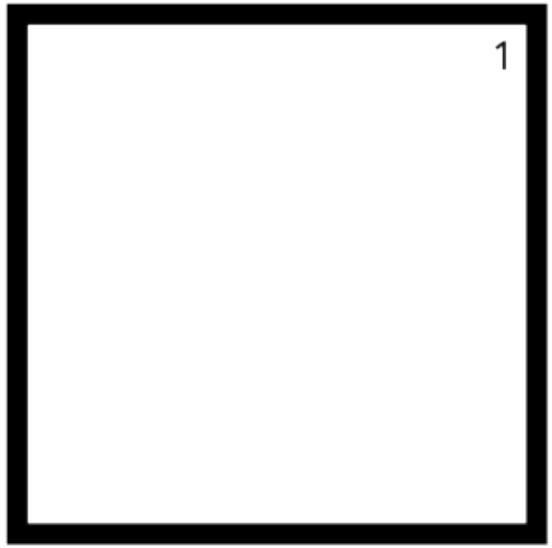
iteration=4


State
live_parts: 5
live_ranks: 0,1,2,3
dead_parts:
0:0 1:2
2:0 3:0
0
1
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? 1:?
2:? 3:?
tag=2
2
3
0
incoming buffer
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? 1:?
2:? 3:?
2
3
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? 1:?
2:? 3:?

iteration=4

State
live_parts: 6
live_ranks: 0,1,2,3
dead_parts:
0:0 | 1:2
2:0 | 3:0

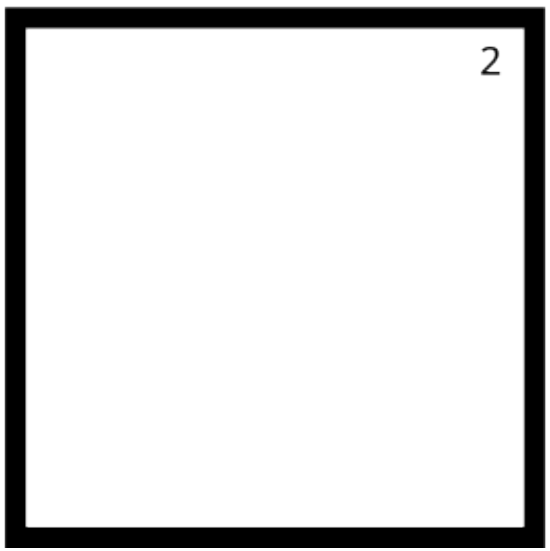


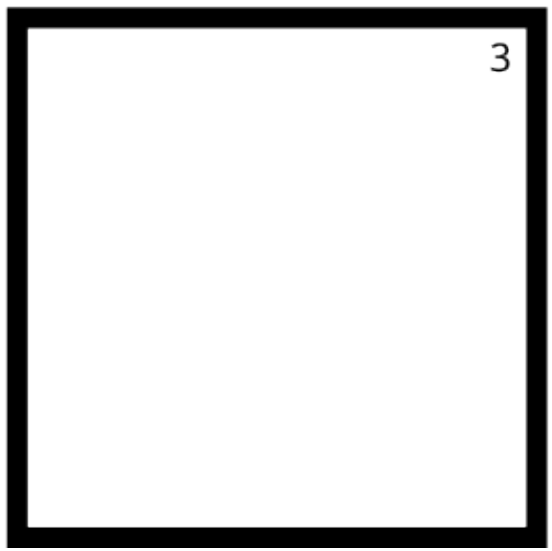


State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

Generate 1 particle to fill buffer

State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

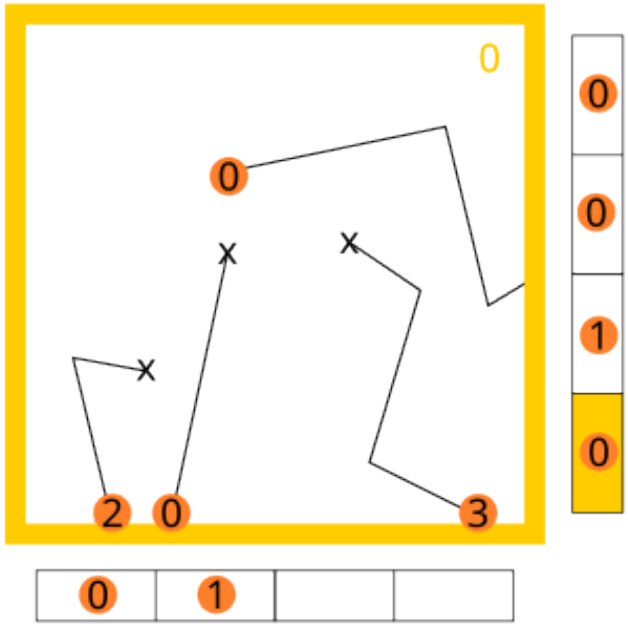




State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:
0:? | 1:?
2:? | 3:?

iteration=4

State	
live_parts: 6	
live_ranks: 0,1,2,3	
dead_parts:	
0:1	1:2
2:1	3:1



State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

1

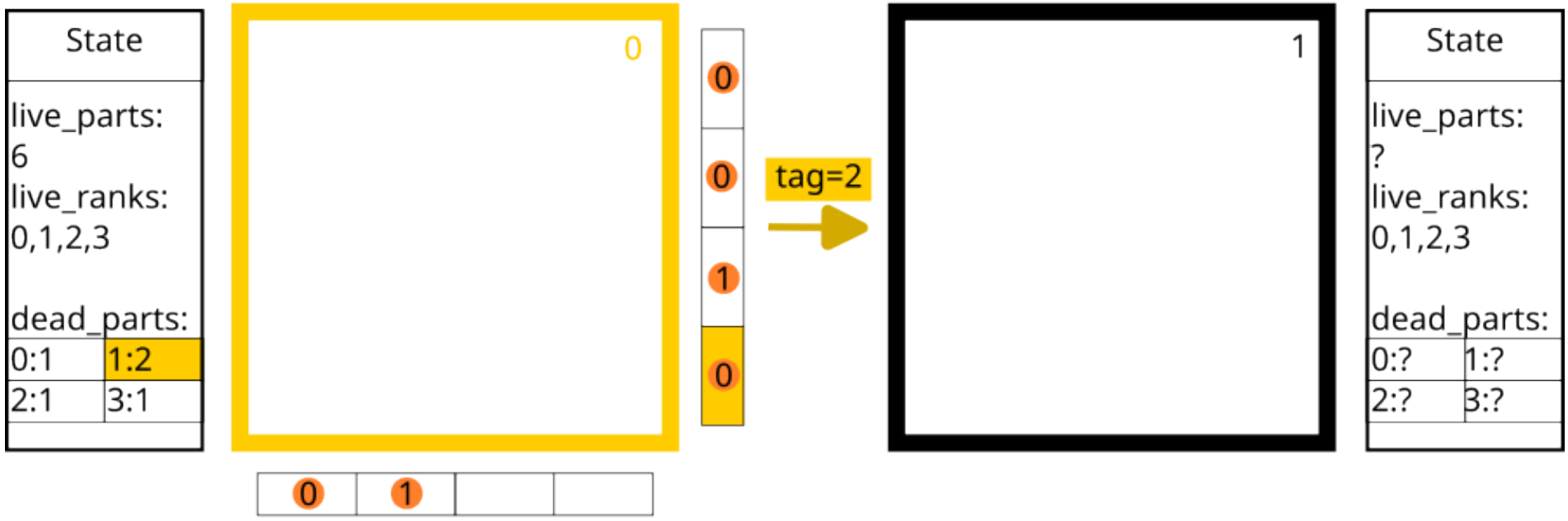
State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

2

State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

3

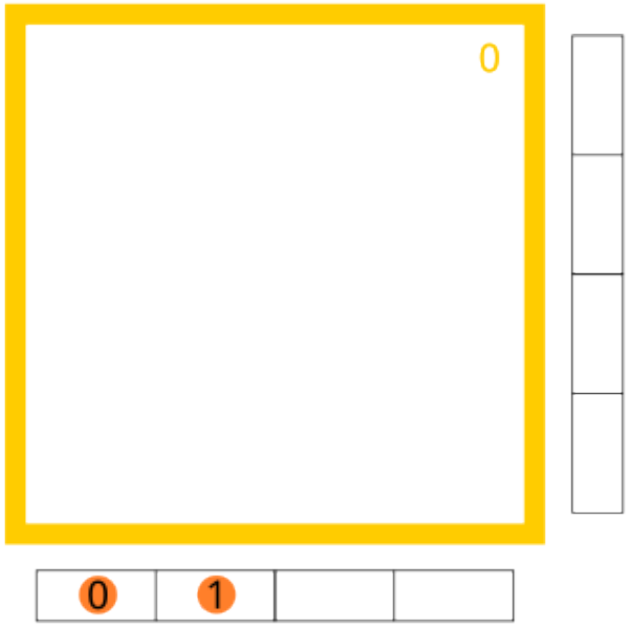
iteration=4

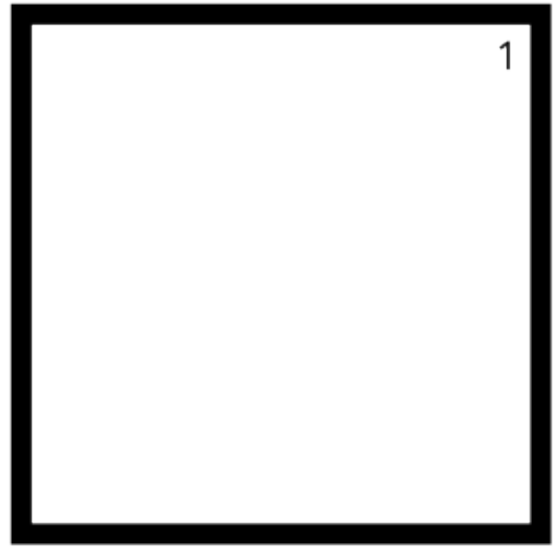

State
live_parts:
6
live_ranks:
0,1,2,3
dead_parts:
0:1
1:2
2:1
3:1
0
0
0
1
0
tag=2
1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
0
1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
2
3
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?

iteration=4

State
live_parts: 6
live_ranks: 0,1,2,3
dead_parts:

0:1	1:0
2:1	3:1



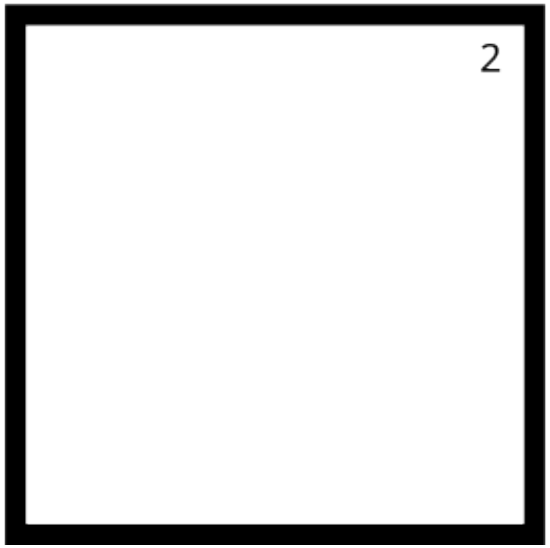


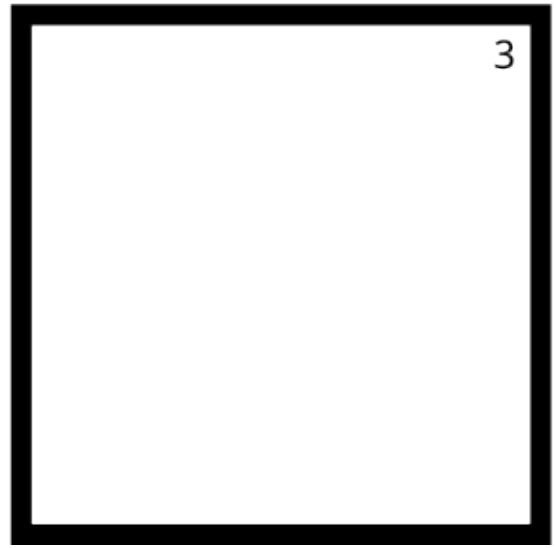
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?

State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?

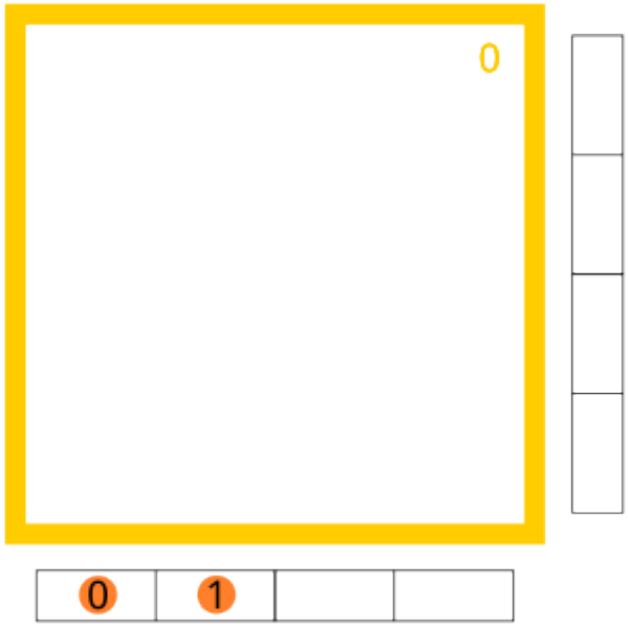


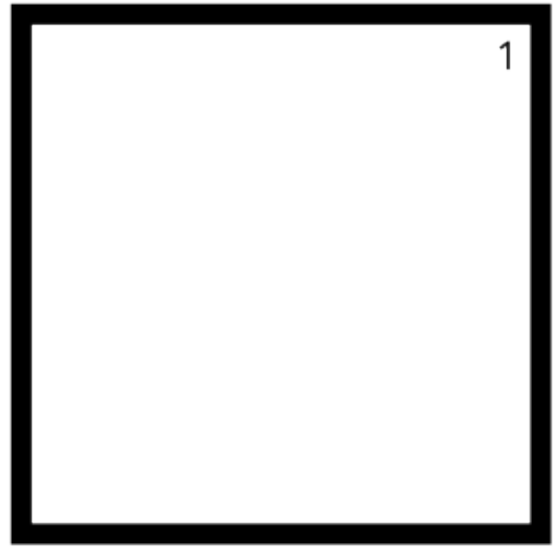


iteration=4

State
live_parts: 5
live_ranks: 0,1,2,3
dead_parts:

0:0	1:0
2:1	3:1



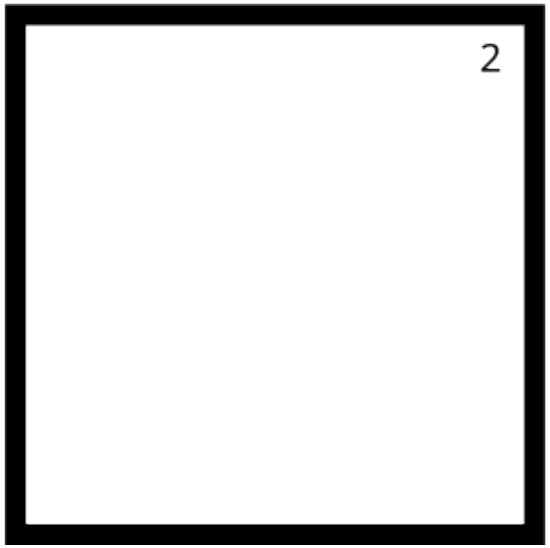


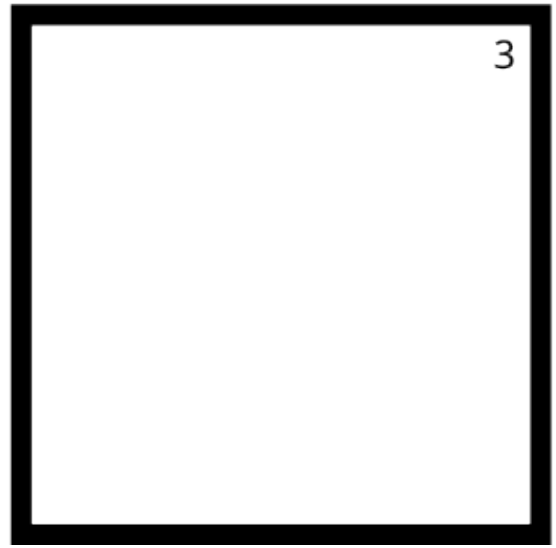
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?

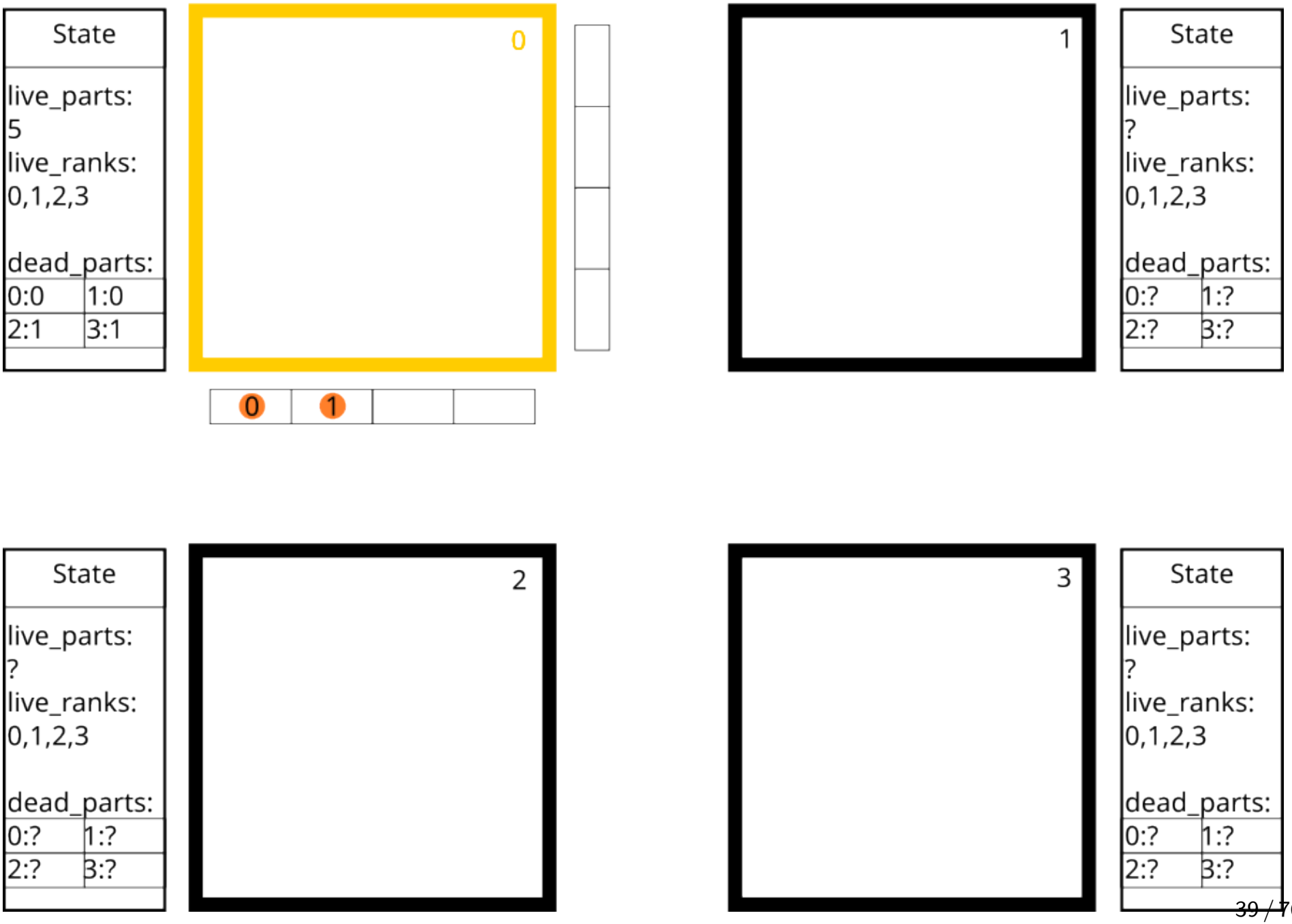
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?

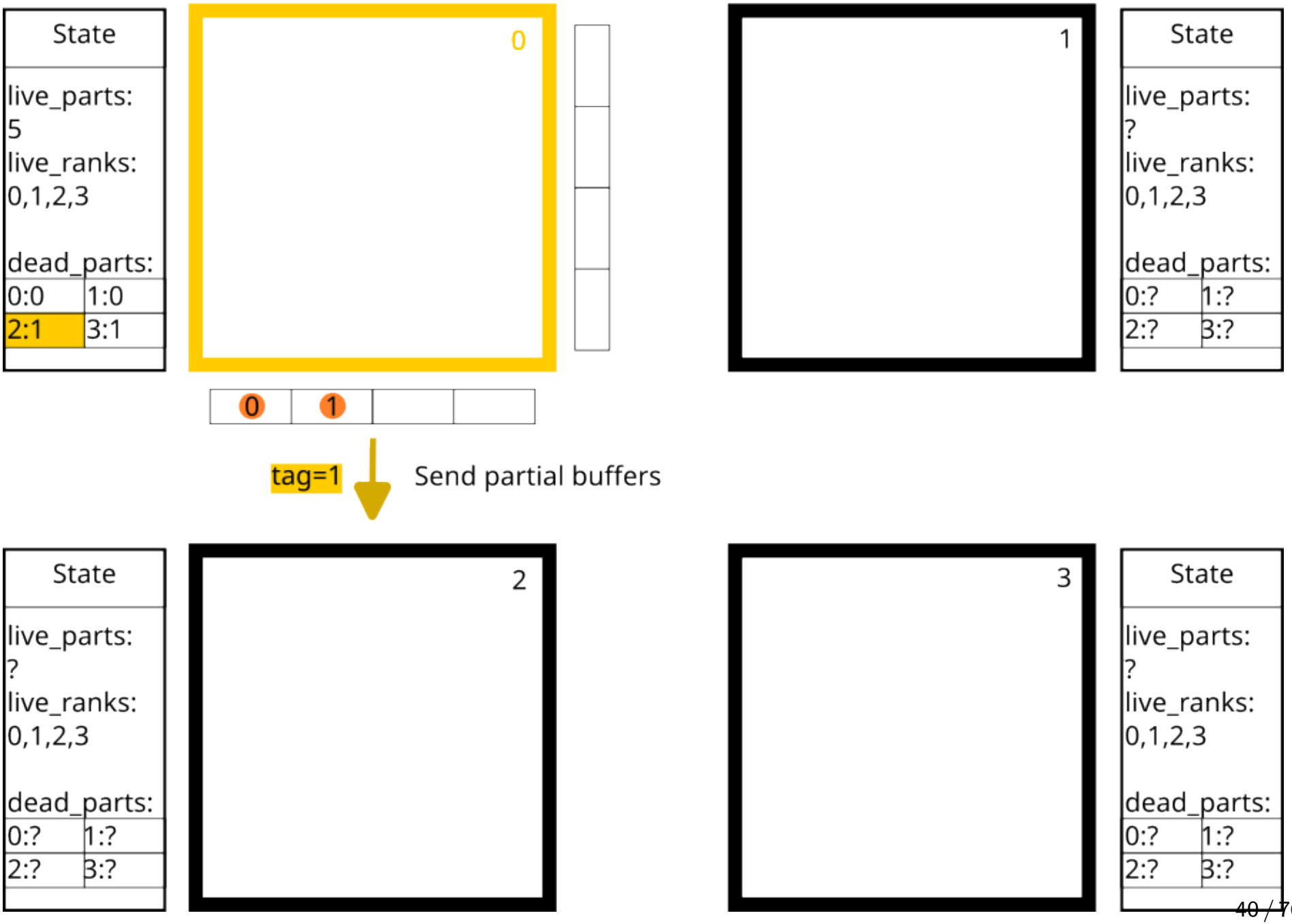




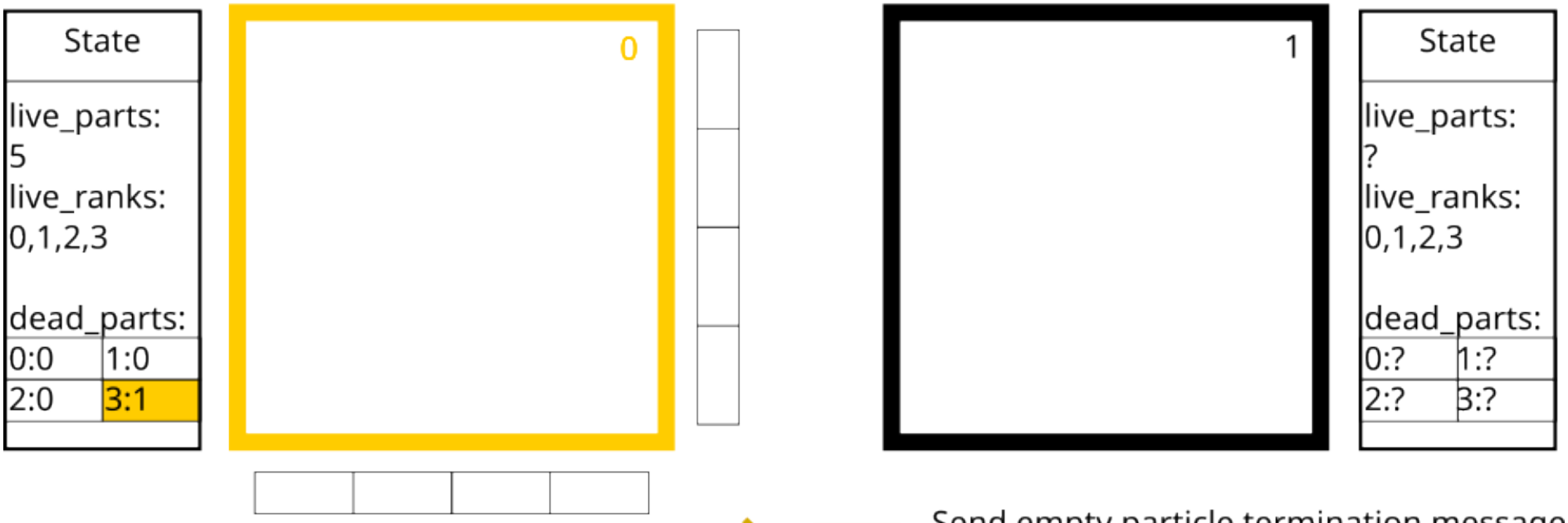
iteration=4 == SEND_PERIOD*k



iteration=4 == SEND_PERIOD*k

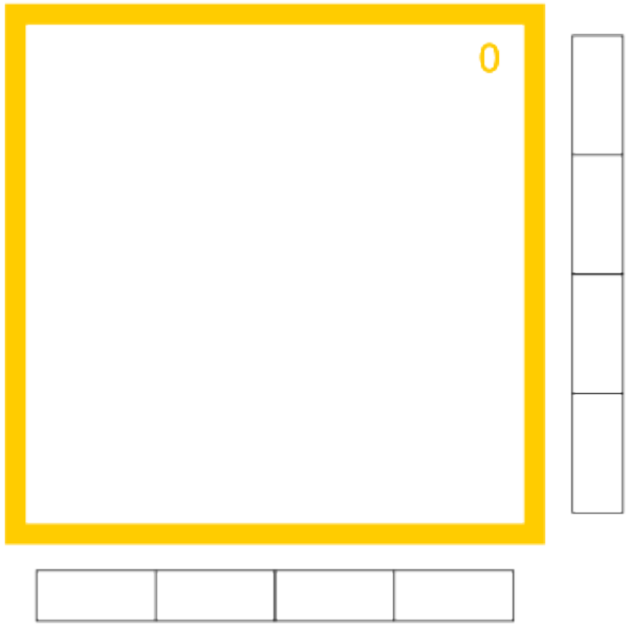


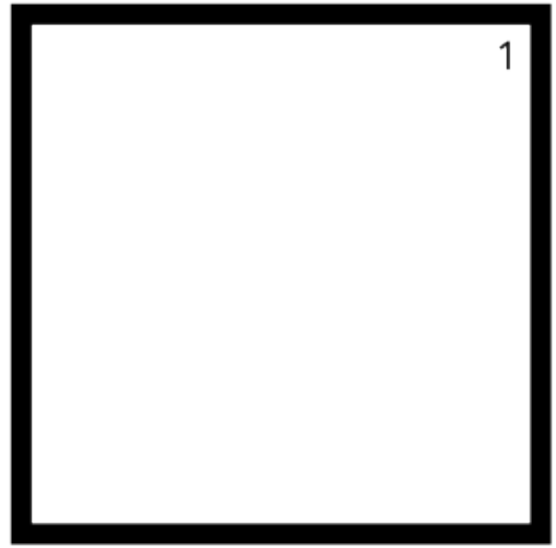
iteration=4 == SEND_PERIOD*k



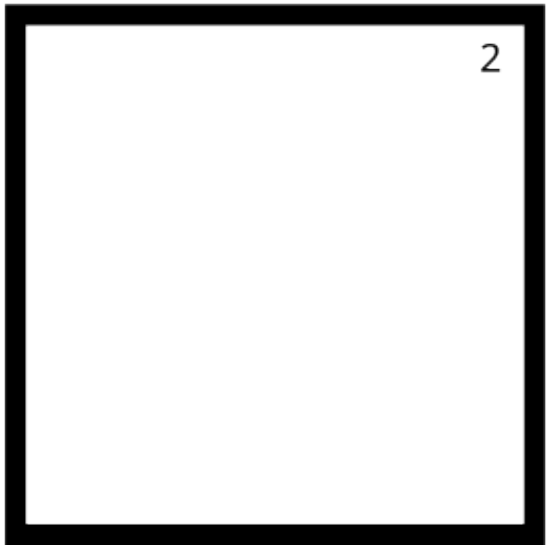
iteration=4

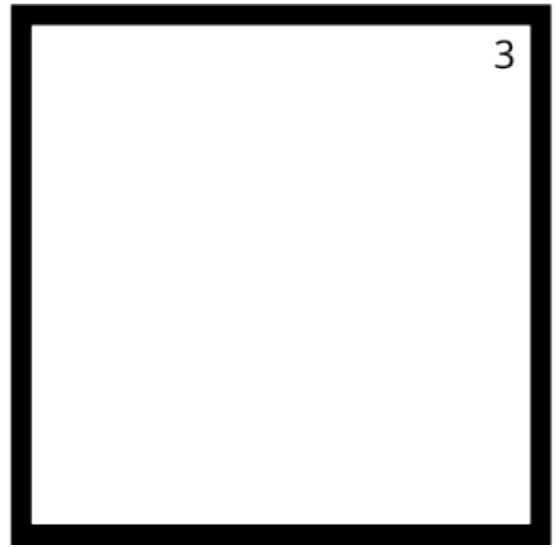
State	
live_parts: 5	
live_ranks: 0,1,2,3	
dead_parts:	
0:0	1:0
2:0	3:0



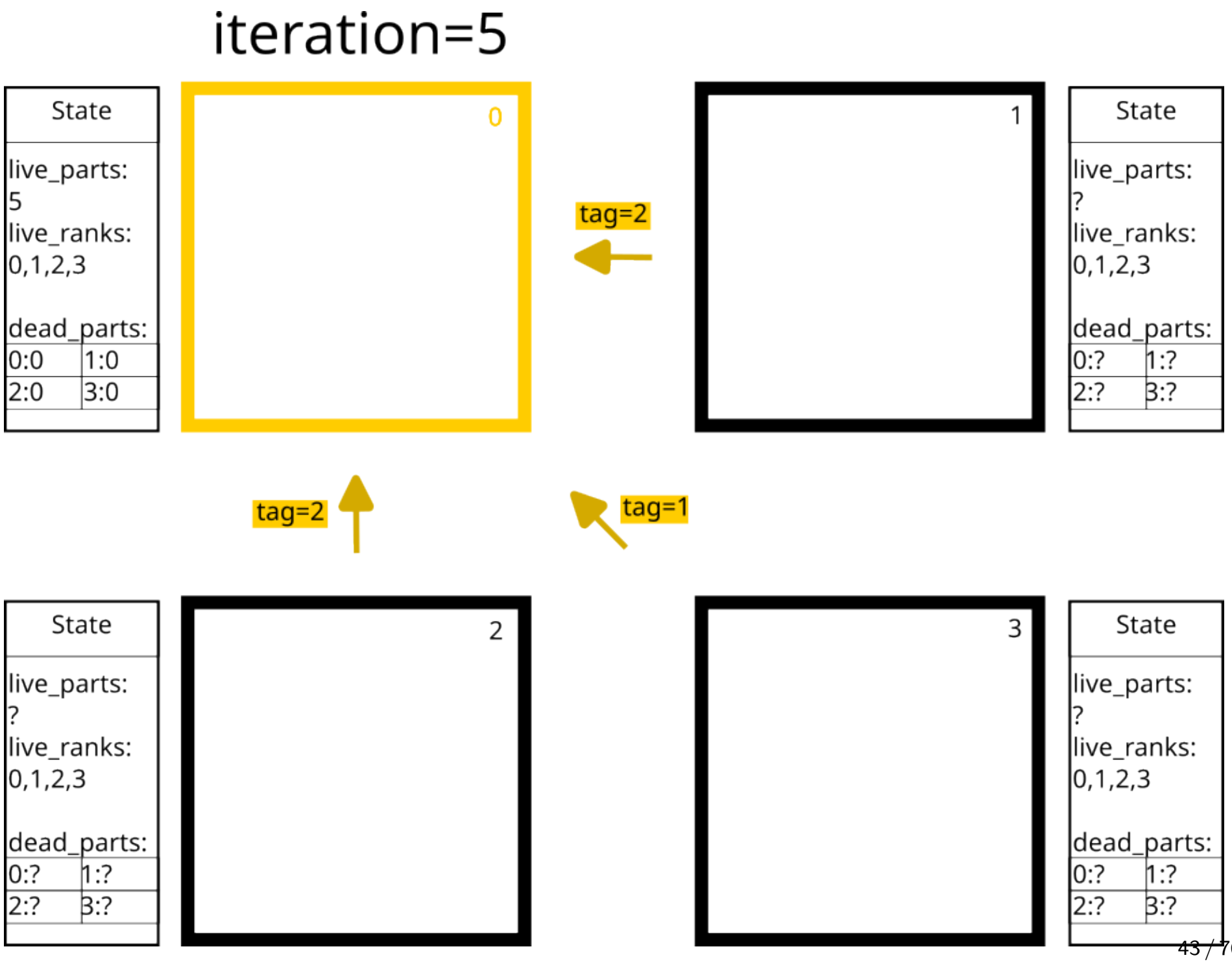


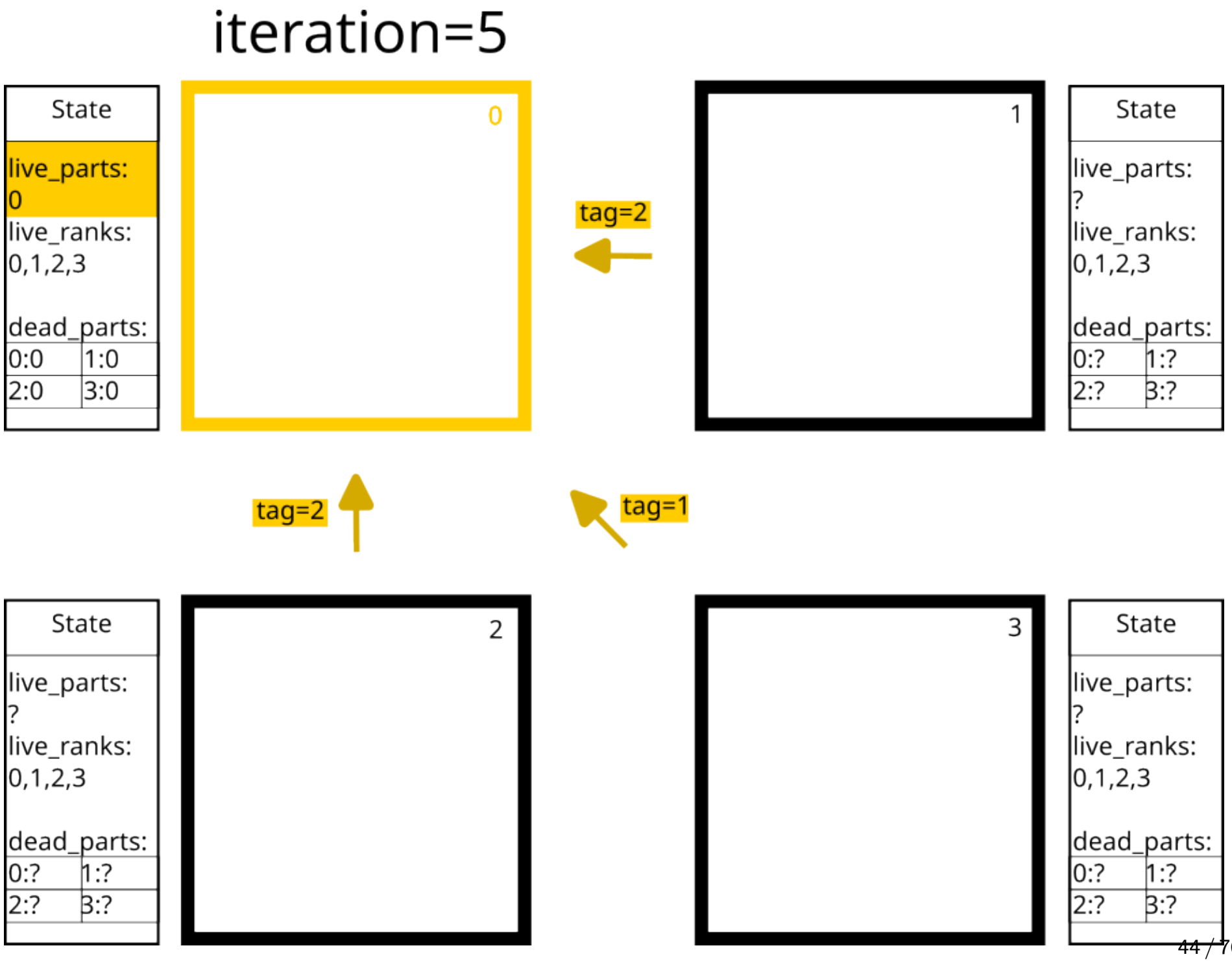
State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?





State	
live_parts: ?	
live_ranks: 0,1,2,3	
dead_parts:	
0:?	1:?
2:?	3:?

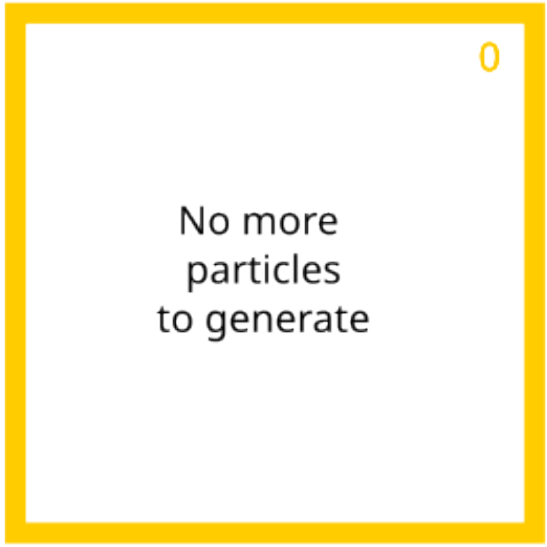

iteration=5
State
live_parts:
5
live_ranks:
0,1,2,3
dead_parts:
0:0
1:0
2:0
3:0
0
1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
tag=2
tag=2
tag=1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
2
3
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?

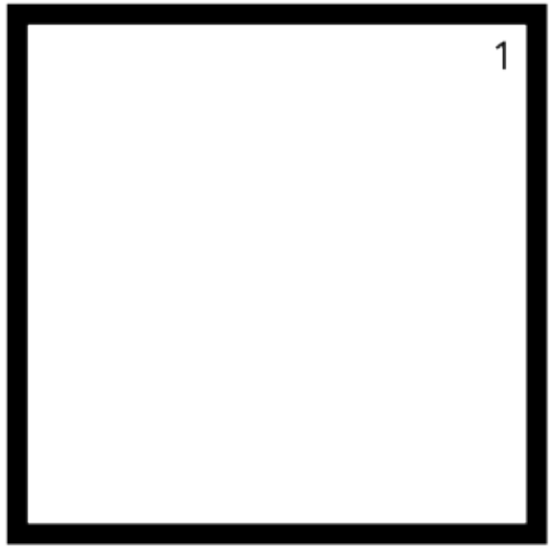

iteration=5
State
live_parts:
0
live_ranks:
0,1,2,3
dead_parts:
0:0
1:0
2:0
3:0
0
tag=2
1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
tag=2
tag=1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
2
3
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?

iteration=5

State
live_parts: 0
live_ranks: 0,1,2,3
dead_parts:

0:0	1:0
2:0	3:0



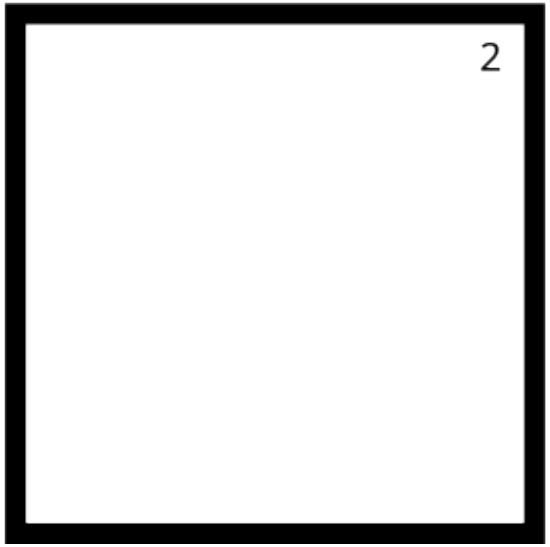


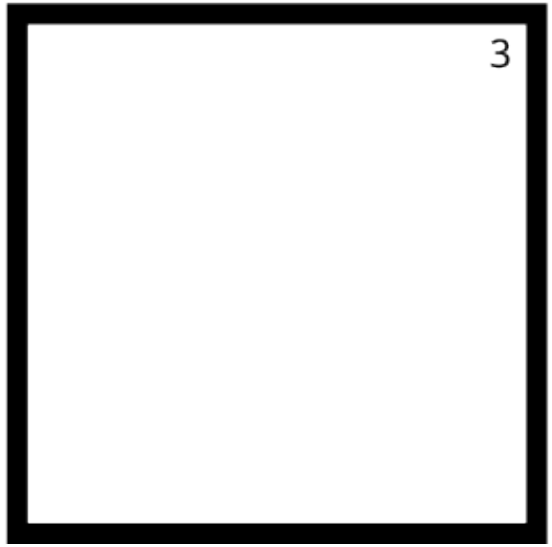
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

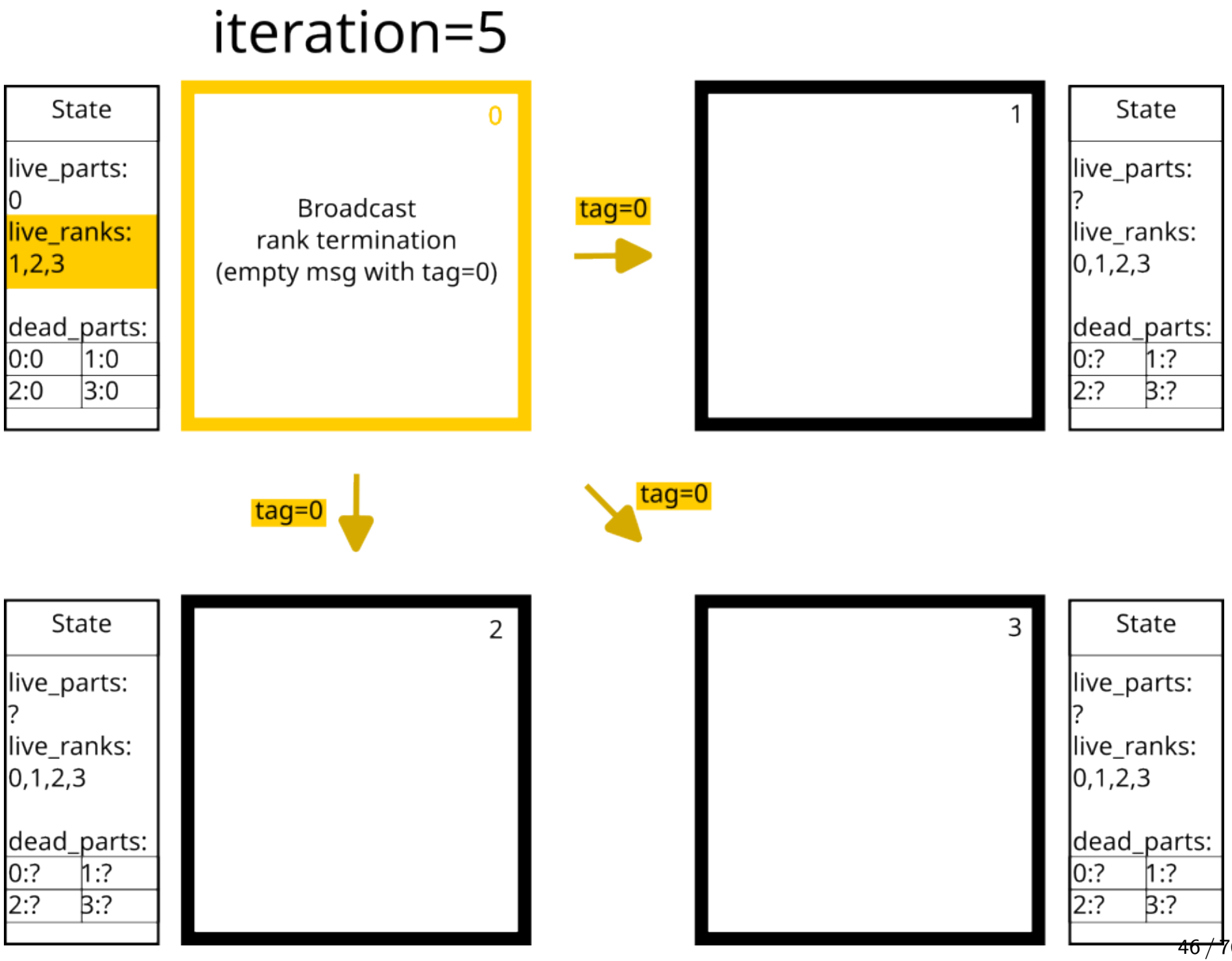
0:?	1:?
2:?	3:?

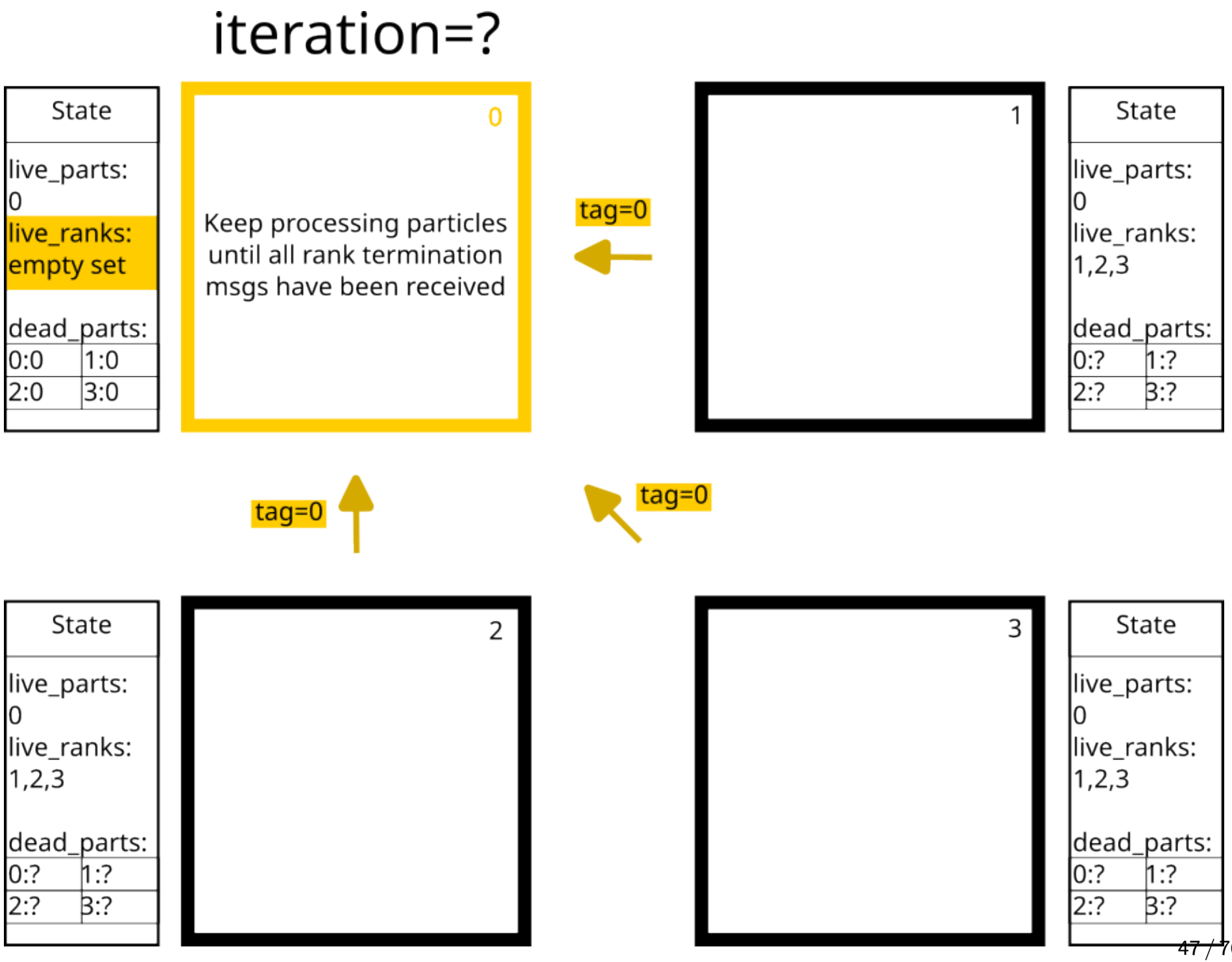
State
live_parts: ?
live_ranks: 0,1,2,3
dead_parts:

0:?	1:?
2:?	3:?






iteration=5
State
live_parts:
0
live_ranks:
1,2,3
dead_parts:
0:0
1:0
2:0
3:0
0
Broadcast
rank termination
(empty msg with tag=0)
tag=0
1
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
tag=0
tag=0
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?
2
3
State
live_parts:
?
live_ranks:
0,1,2,3
dead_parts:
0:?
1:?
2:?
3:?


iteration=?
State
live_parts:
0
live_ranks:
empty set
dead_parts:
0:0
1:0
2:0
3:0
0
Keep processing particles until all rank termination msgs have been received
tag=0
1
State
live_parts:
0
live_ranks:
1,2,3
dead_parts:
0:?
1:?
2:?
3:?
tag=0
tag=0
State
live_parts:
0
live_ranks:
1,2,3
dead_parts:
0:?
1:?
2:?
3:?
2
3
State
live_parts:
0
live_ranks:
1,2,3
dead_parts:
0:?
1:?
2:?
3:?

iteration=?

State
live_parts: 0
live_ranks: empty set
dead_parts:

0:0	1:0
2:0	3:0

0

When there are no live ranks, the simulation has terminated

1

State

live_parts: 0
live_ranks: empty set

dead_parts:

0:0	1:0
2:0	3:0

State

live_parts: 0
live_ranks: empty set

dead_parts:

0:0	1:0
2:0	3:0

2

3

State

live_parts: 0
live_ranks: empty set

dead_parts:

0:0	1:0
2:0	3:0

Algorithm, for completeness

Algorithm Domain decomposition

Partition grid and sources
live_ranks ← {*all ranks*}, *live_parts* ← 0, *dead_parts*[*all ranks*] ← 0
buffers ←one buffer per neighbor
iteration ←1
while there are active ranks **do**
    sim_buffer ← TRY_RECV(*live_ranks*, *live_parts*)
    while |*sim_buffer*| < BUFFER_SIZE **do**
        sim_buffer ⇐ try to generate particle from local sources
        Increment *live_parts* for each generated particle
    for each *particle* **in** *sim_buffer* **do**
        if *particle* crossed subdomain boundary **then**
            Add *particle* to *buffer* with *sid* equal to the particle's next subdomain id
            dead_parts ← SEND_BUFFER(*buffer*,*buffer.sid*,*dead_parts*) (if full)
        if particle terminated **then**
            Increment *dead_parts*[*particle.gen_rank*]
    if *iteration* ≡ *0* (*mod* SEND_PERIOD) **then**
        for each *buffer* **in** *buffers* **do**
            dead_parts ← SEND_BUFFER(*buffer*,*buffer.sid*, *dead_parts*)
        dead_parts ← SEND_DEAD_PARTICLE_COUNT(*dead_parts*)
    if local sources depleted and *live_parts* = 0 **then**
        Mark local rank as dead, broadcast empty msg with tag 0
    Increment *iteration*

Algorithm, for completeness

Algorithm Domain decomposition communication routines

function TRY_RECV(*live_ranks*, *live_parts*)
- **if** a message *msg* has arrived **then**
  - *live_parts* ← *live_parts* − *msg.tag*
  - **if** *msg.buffer* is not empty **then**
    - **return** (*msg.buffer*, *live_ranks*, *live_parts*)
  - **if** *msg.buffer* is empty ∧ *msg.tag* = 0 **then**
    - Remove *msg.source* from *live_ranks*
- **return** (empty buffer, *live_ranks*, *live_parts*)

function SEND_BUFFER(*buffer*, *sid*, *dead_parts*)
- *rank* ← Select an MPI rank that is assigned to subdomain *sid*
- *t* ←MIN(*dead_parts*[*rank*], MPI_TAG_UB)
- Asynchronously send *buffer* to *rank* with tag=*t*
- *dead_parts*[*rank*] ← *dead_parts*[*rank*] − *t*
- **return** *dead_parts*

function SEND_DEAD_PARTICLE_COUNT(*dead_parts*)
- **for each** (*rank*, *count*) **in** *dead_parts* **do**
  - **if** *count* > 0 **then**
    - *t* ←MIN(*count*, MPI_TAG_UB)
    - Asynchronously send empty message to *rank* with tag=*t*
    - *dead_parts*[*rank*] ← *dead_parts*[*rank*] − *t*
- **return** *dead_parts*

Asynchronous termination control

Novel asynchronous termination control

Previous algorithms have relied on global sums of generated vs terminated particles. The novelty is that the sums are partitioned by generating rank and done fully asynchronously.

Piggy-backing on the particle buffer messages we also eliminate a portion of the termination messages.

Performance and scalability results

Scaling experiments

We define two collision processes

- scattering (isotropic rotation)
- absorption (path termination)

Setting mean speed $= 1$, and we vary

- λ, the mean free path
- ν_s/ν_t, the ratio of scattering collisions to total collisions

We use two collision regimes

- $\lambda = 0.25, \nu_s/\nu_t = 0.01$ (*low-collisional*)
- $\lambda = 0.05, \nu_s/\nu_t = 0.99$ (*high-collisional*)

Strong scaling experiment

Definition

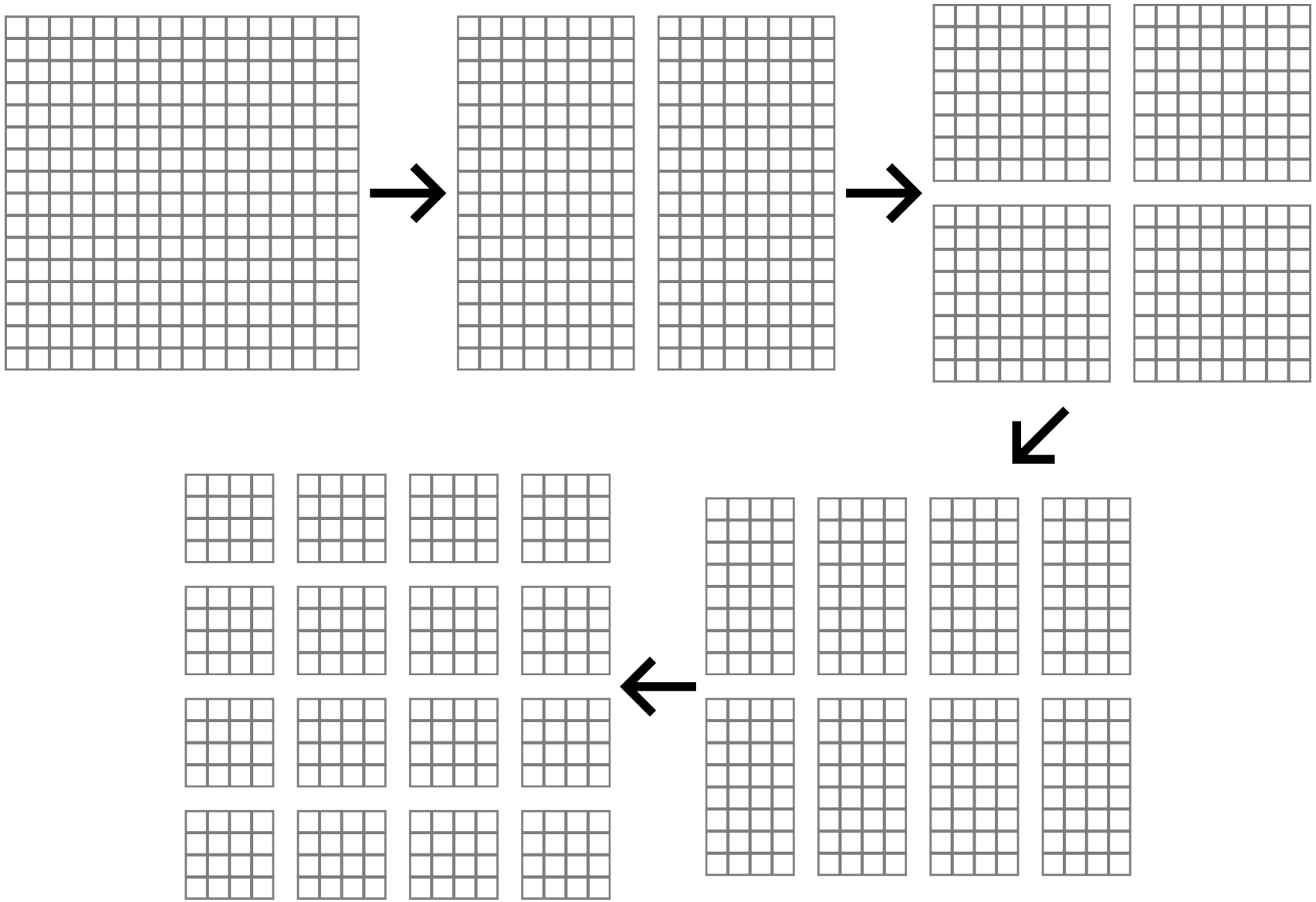
In a strong scaling experiment, we measure the speedup as we add more compute resources to a constant overall workload.

Scale

We scale all three algorithms to a full node (128 cores).

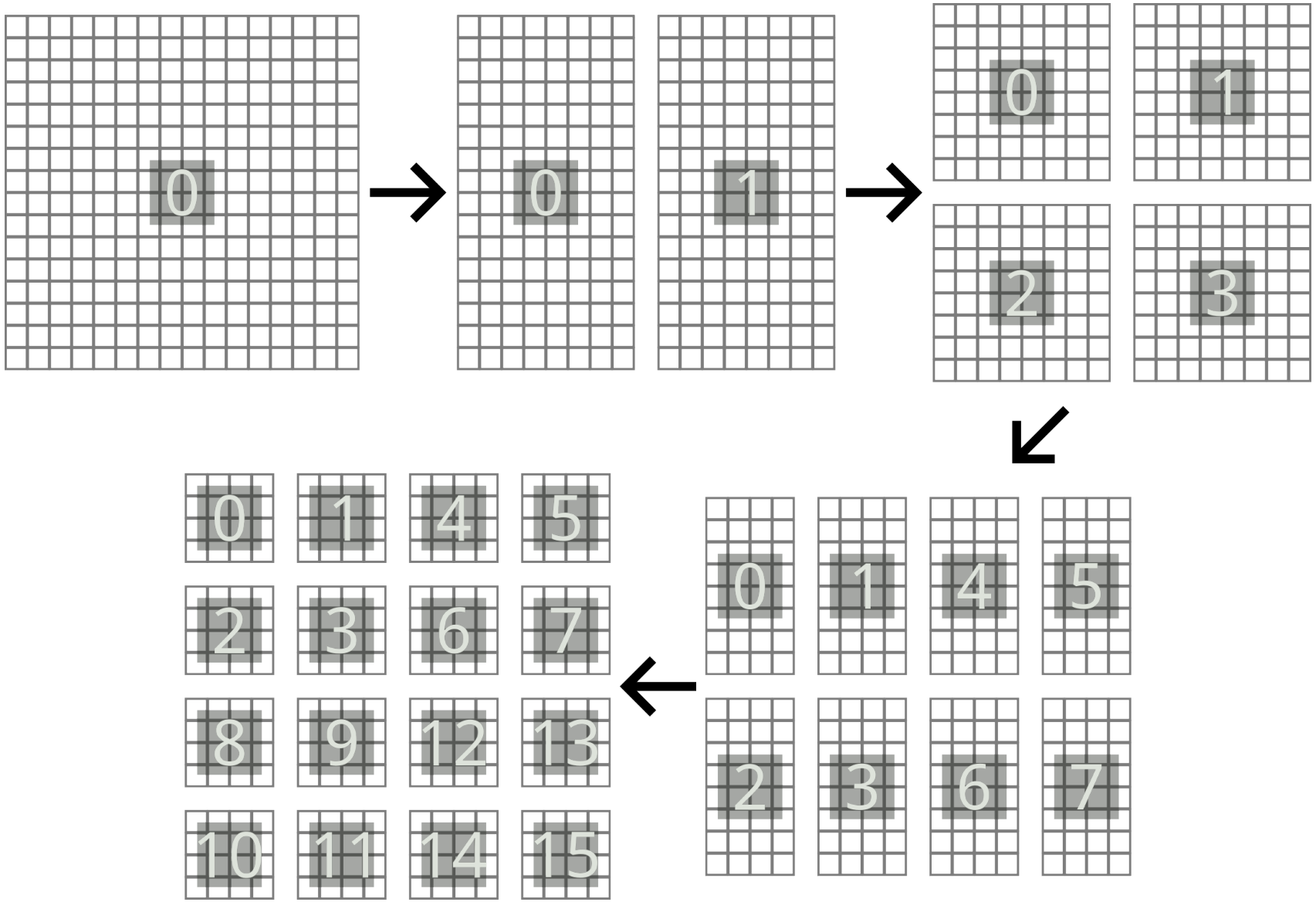
For DR and SM algorithms, we add an OpenMP thread for each new core.

Strong scaling experiment



For DDMC, we recursively bisect the domain as we add MPI ranks.

Strong scaling experiment



We use z-order indexing to assign nearby subdomains to nearby CPU cores.

Strong scaling experiment

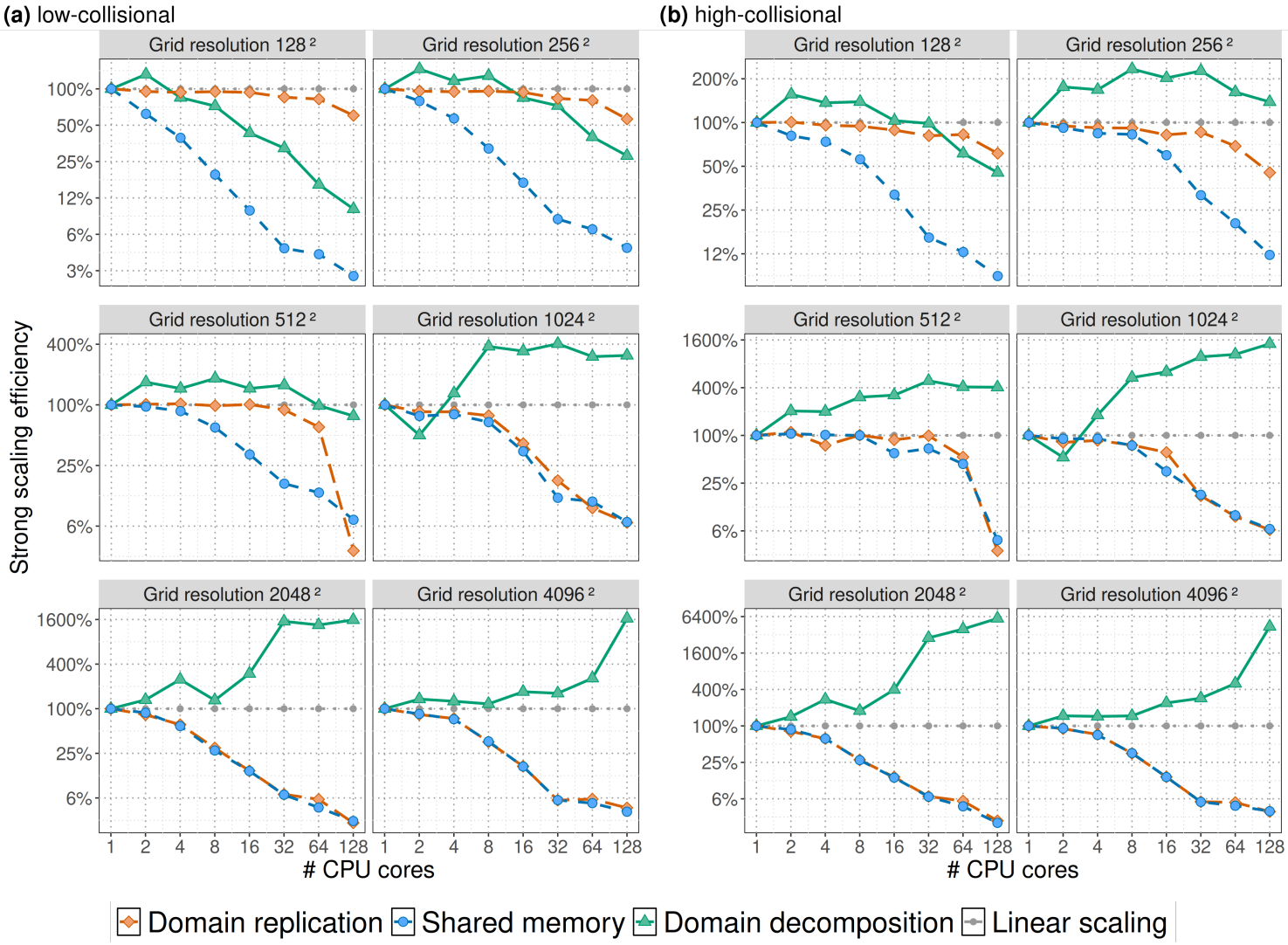
Strong scaling efficiency

The figure on the next slide plots the *strong scaling efficiency* SSE_n as it varies when we increase the CPU core count n:

$$SSE_n = \frac{t_1}{t_n \times n}$$

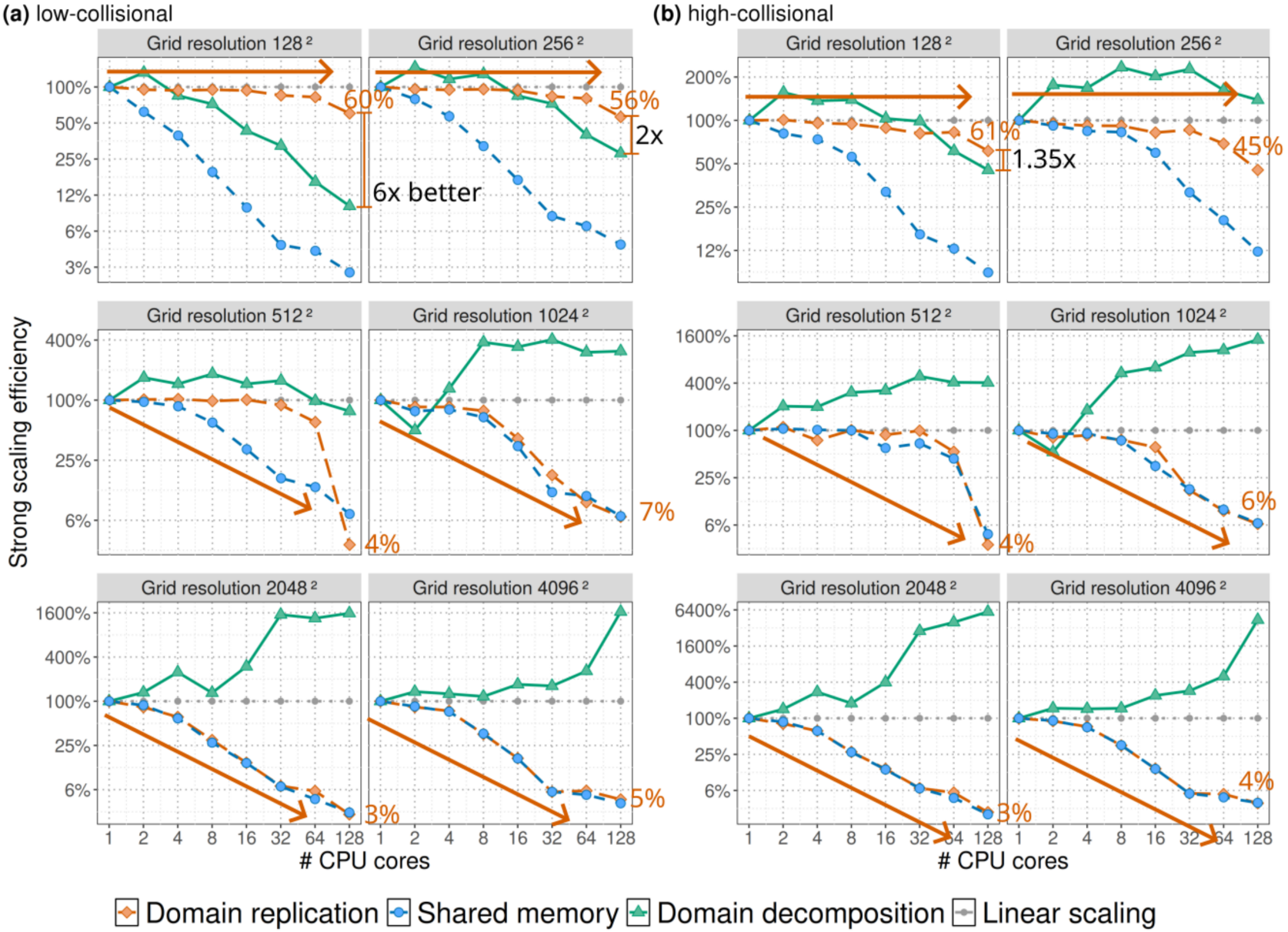
where t_n is the runtime of the simulation for n CPU cores.

You may be used to seeing the *strong scaling speedup*, this number is the speedup divided by the number of cores.

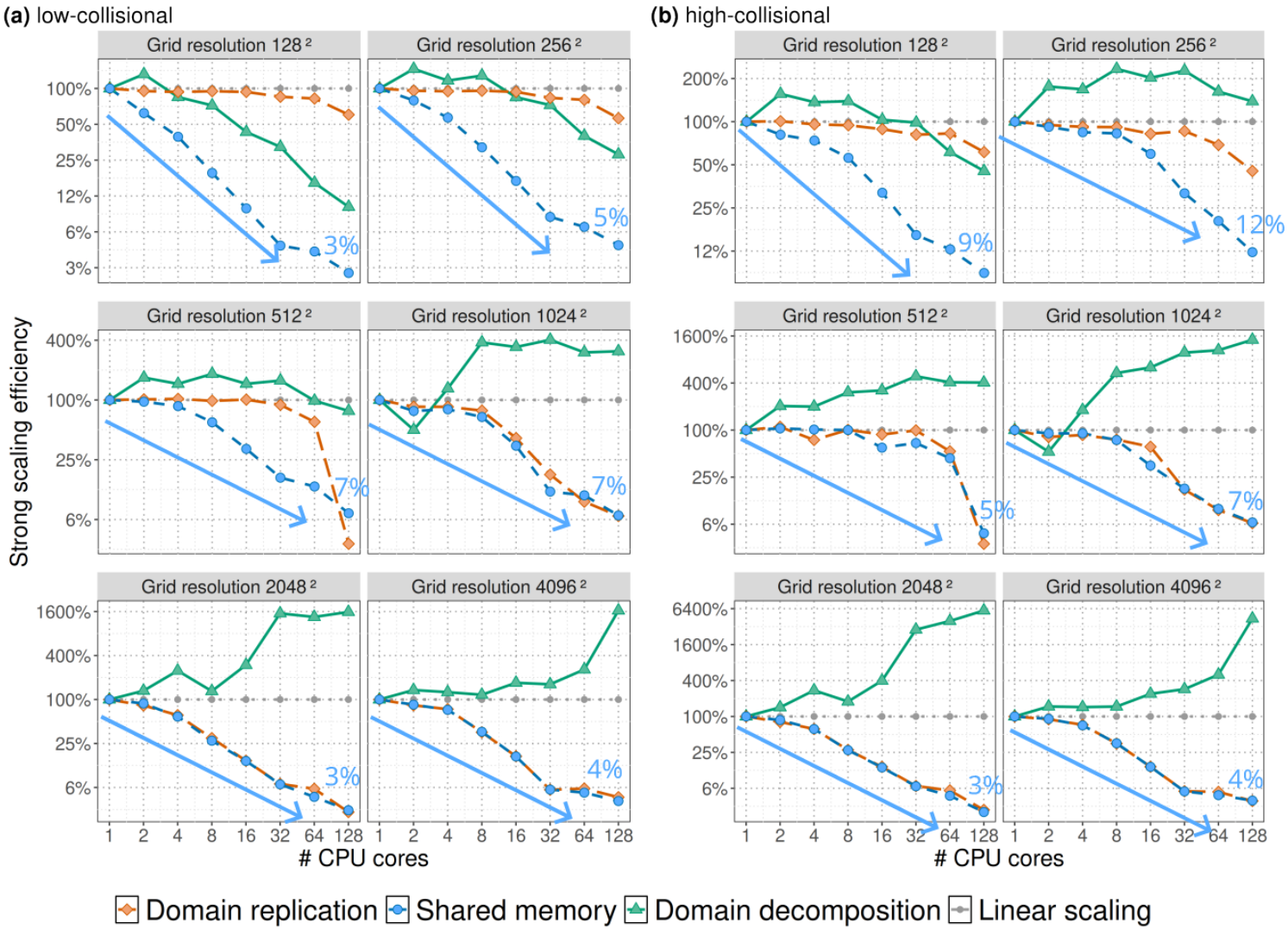


In the above log-log figure, SSE is shown across grid resolutions and collision regimes. Small grids on top, large grids on bottom.

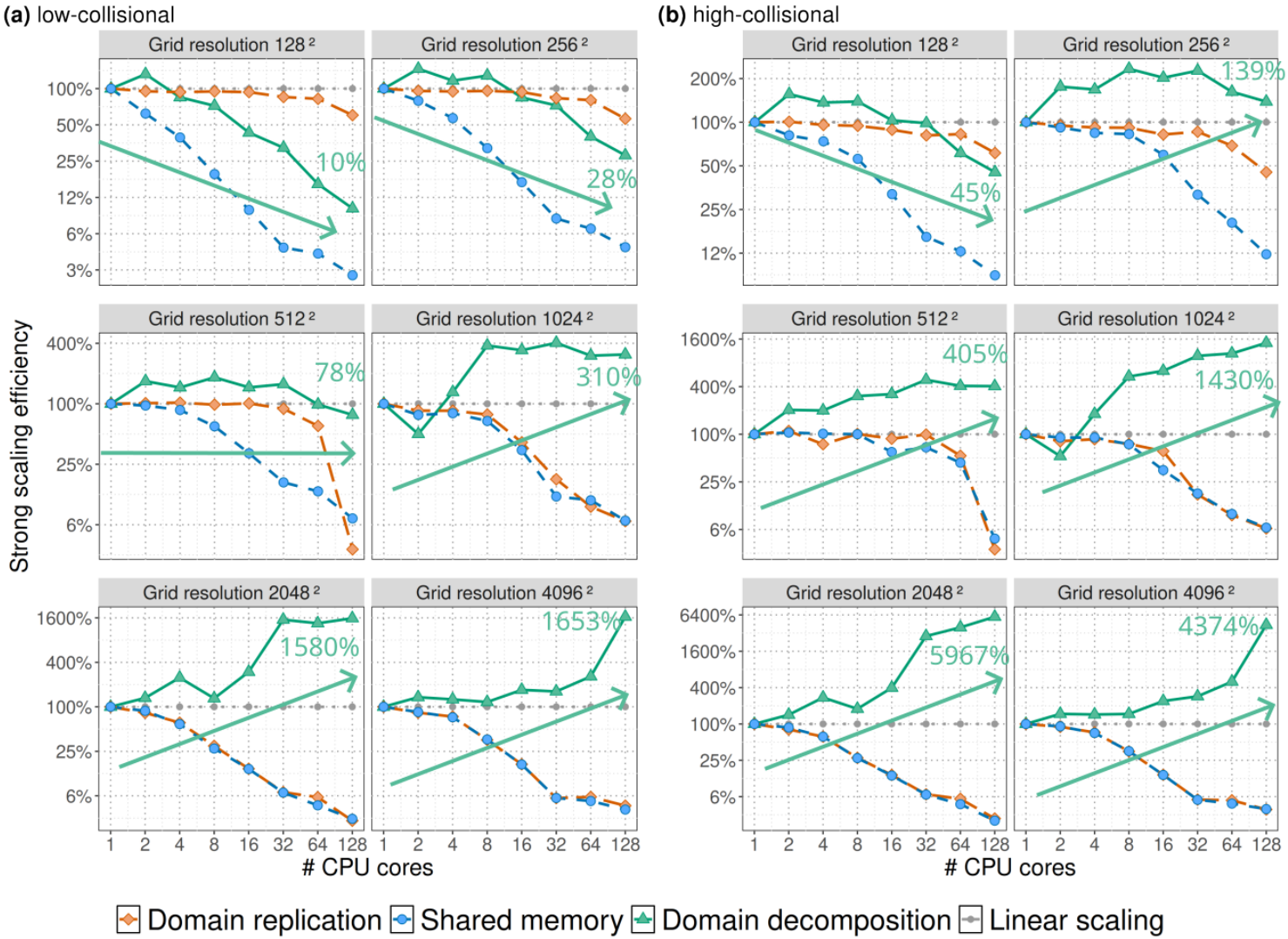
Note: due to variance in superlinear scaling, y-axis is specific to subfigure pairs.



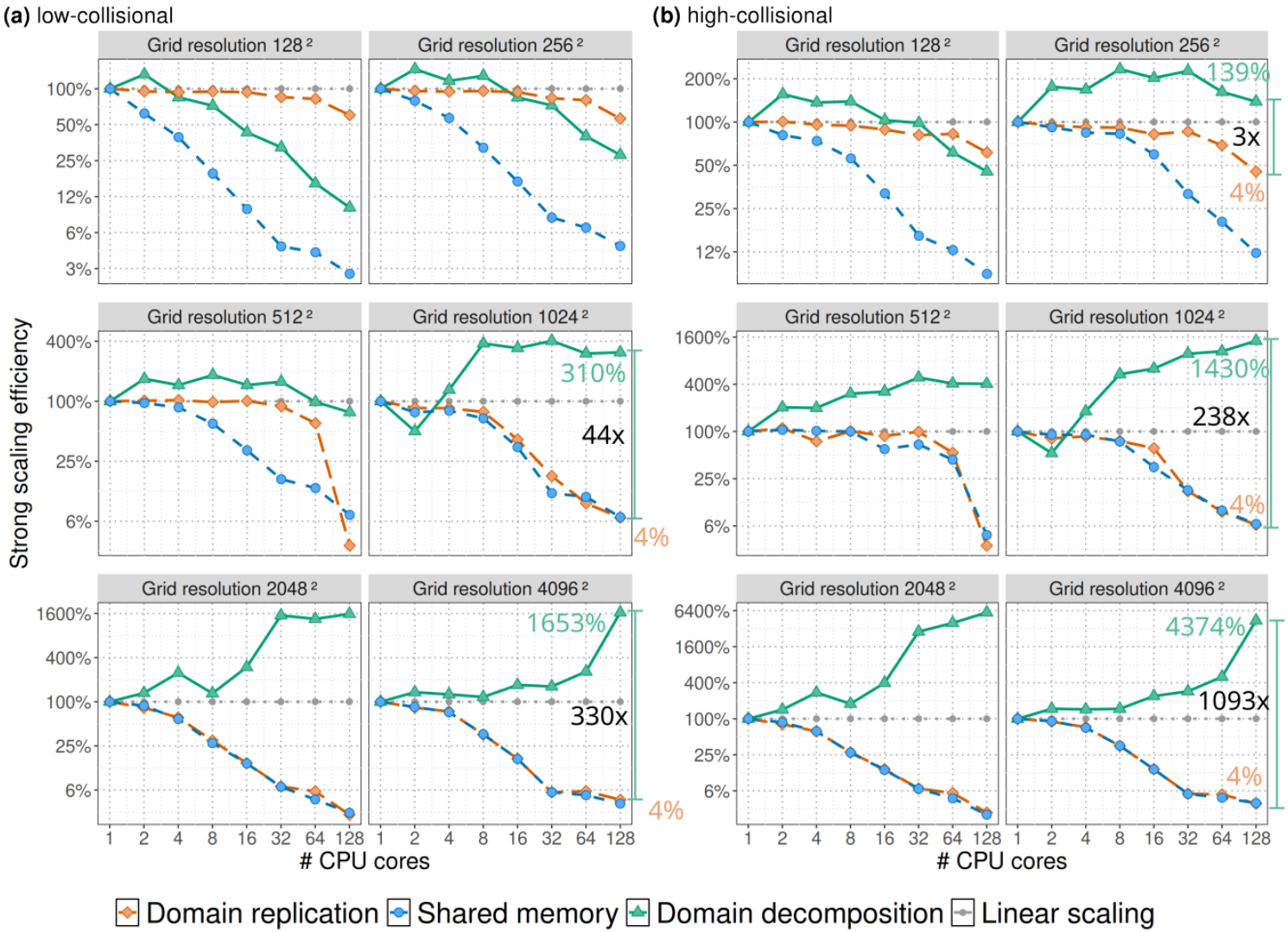
- For small grids domain replication scales well. Better than DDMC
- Domain replication is not really sensitive to collisionality



- Shared memory scalability is never good, except for small core counts
- We used `OMP_PROC_BIND=spread`, but `OMP_PROC_BIND=close` improves performance if optimizing for a handful of cores



- Domain decomposition scales well except for really small grids and low-collisionality (high communication overhead)
- DDMC is superlinear for grids larger than the L3 cache (and sometimes for smaller ones)



- If we compare domain decomposition to domain replication we see that the advantage grows as we move from small grids to larger ones
- For big grids, the performance advantage is on the order of 100× or 1000×

Strong scaling, cache efficiency

The grid size, or more precisely, the memory footprint of the grid, has a strong impact on the performance. This is due a higher cache efficiency.

Grid resolution	Grid memory size	L3 miss rate	L3 slice miss rate
128^2	512 KiB	0%	0%
256^2	2 MiB	0%	0%
512^2	8 MiB	0%	50%
1024^2	32 MiB	50%	87.5%
2048^2	128 MiB	87.5%	98.4%
4096^2	512 MiB	98.4%	99.6%

Table 1: Cache miss rate model, $\textit{miss rate} = 1 - \textit{cache size}/\textit{grid memory size}$. The Zen 2 L3 cache size is 16 MiB of which each core has a 4 MiB L3 slice attached to it. The model assumes that Monte Carlo transport simulation is characterized by random memory accesses to grid memory, and that other memory accesses are negligible.

Weak scaling experiment

Definition

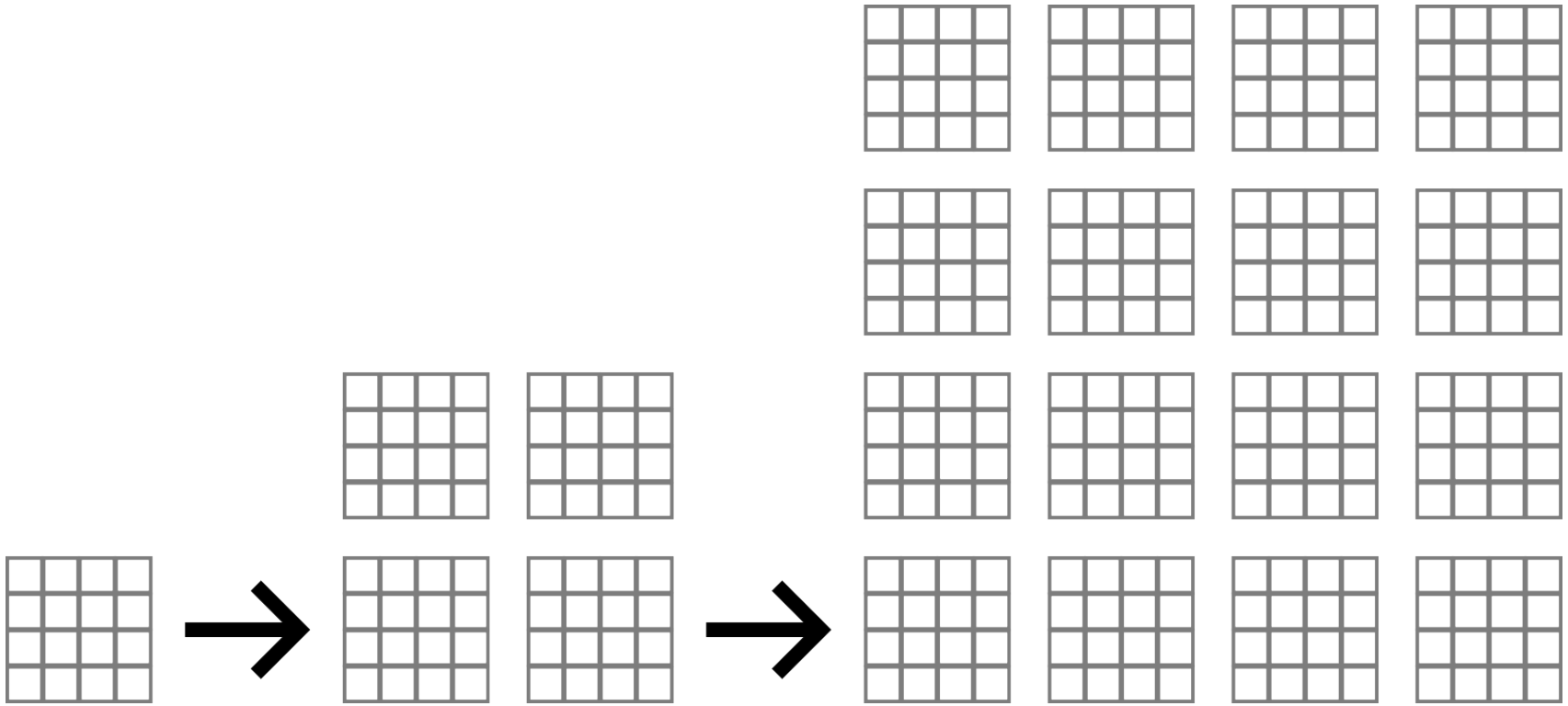
In a weak scaling experiment, we measure how the runtime changes as we add more compute resources, keeping the workload per compute resource constant.

Scale

We scale DDMC to a 128 nodes (16384 cores). This is the largest compute allocation we could get on Mahti.

The other algorithms run out of memory, so they were not tested.

Weak scaling experiment



Each MPI rank in the experiment always gets a subdomain of the same size, 256^2. The boundary conditions are periodic, and we generate the same number of particles in each subdomain.

NOTE: The mean free paths are now relative to the subdomain size, not the full domain size.

Weak scaling experiment

Weak scaling efficiency

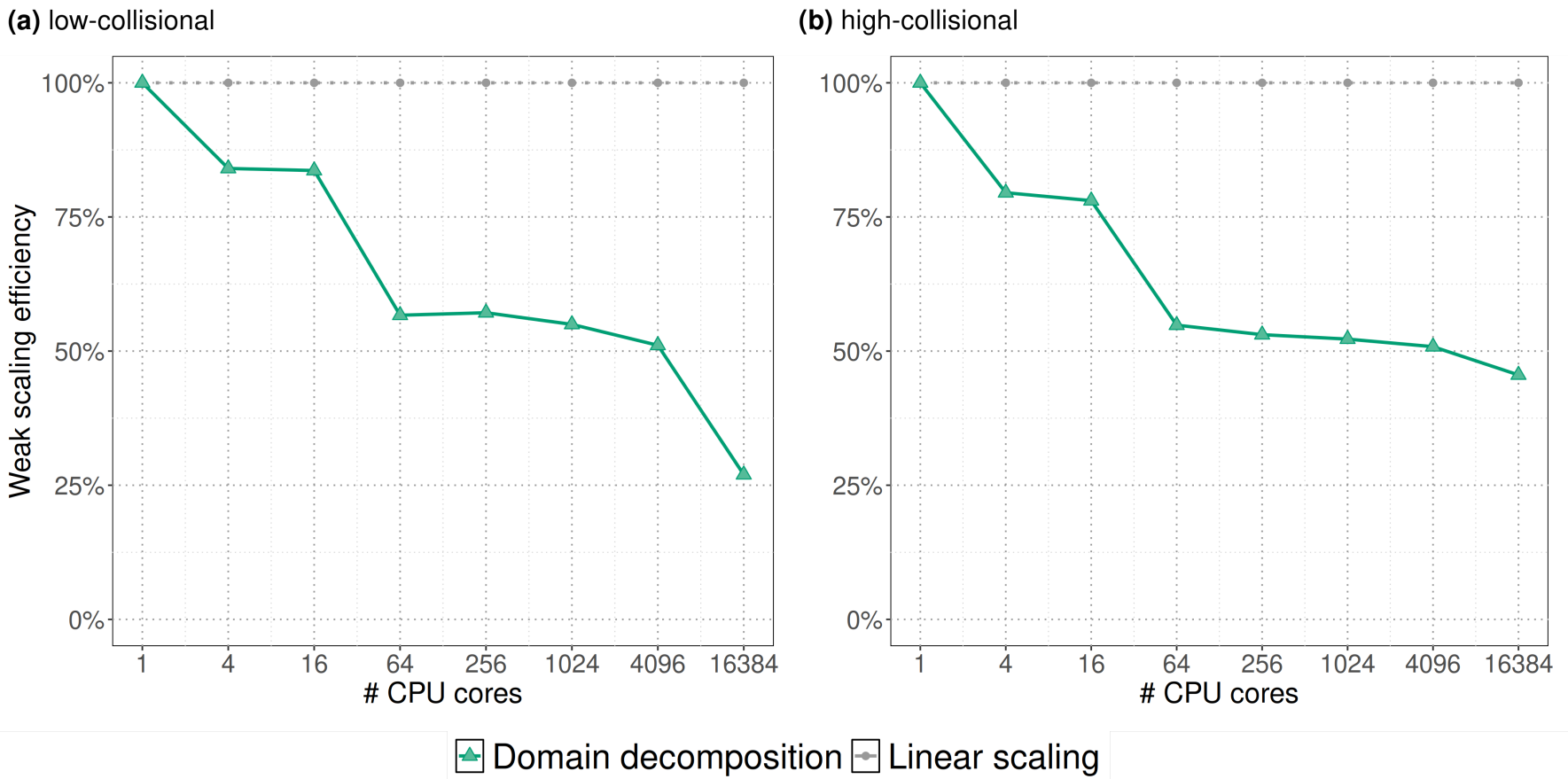
The figure on the next slide plots the *weak scaling efficiency* WSE_n as it varies when we increase the CPU core count n:

$$WSE_n = \frac{t_1}{t_n}$$

where t_n is the runtime of the simulation for n CPU cores.

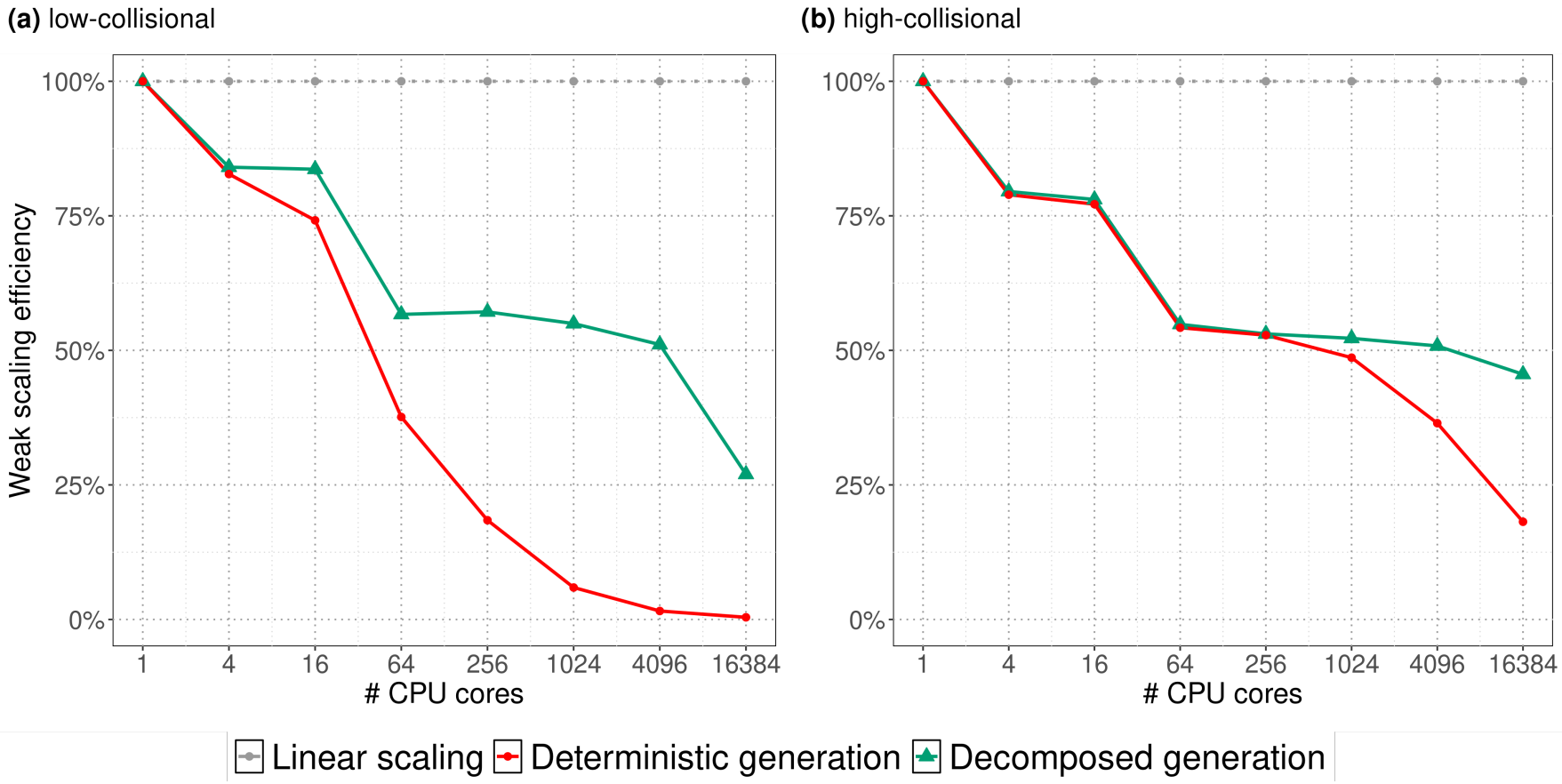
Domain decomposition weak scaling results

The weak scaling of domain decomposition is good, at 16384 CPU cores it is around 50% for high-collisional cases, 25% for low-collisional cases.



In other words, 16384 cores do 16384 times as much work as a single core in twice the time (high-col) or four times the time (low-col).

Particle generation bottleneck



Filtering particles

The deterministic generation method is for each rank to generate all potential particles, and then filter out the ones that don't.

Particle generation bottleneck

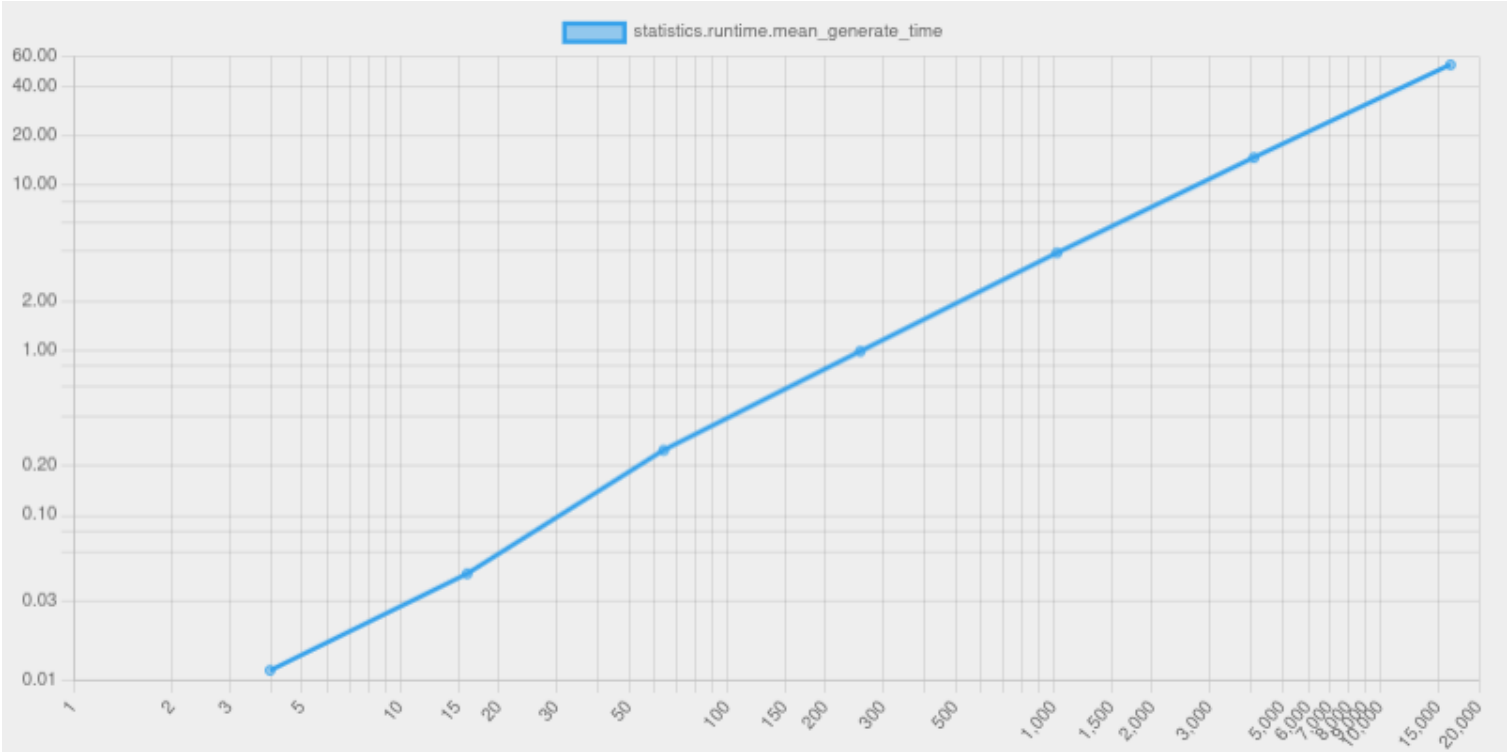


Figure: log-log plot with # cores on x-axis, part. generation time on y-axis (deterministic method). Weak scaling experiment.

Linearly growing cost

The cost to generate a particle is roughly $0.6us$. But for 16384 cores, each core generates $81,920,000$ particles, which costs **53 seconds**.

Particle generation bottleneck

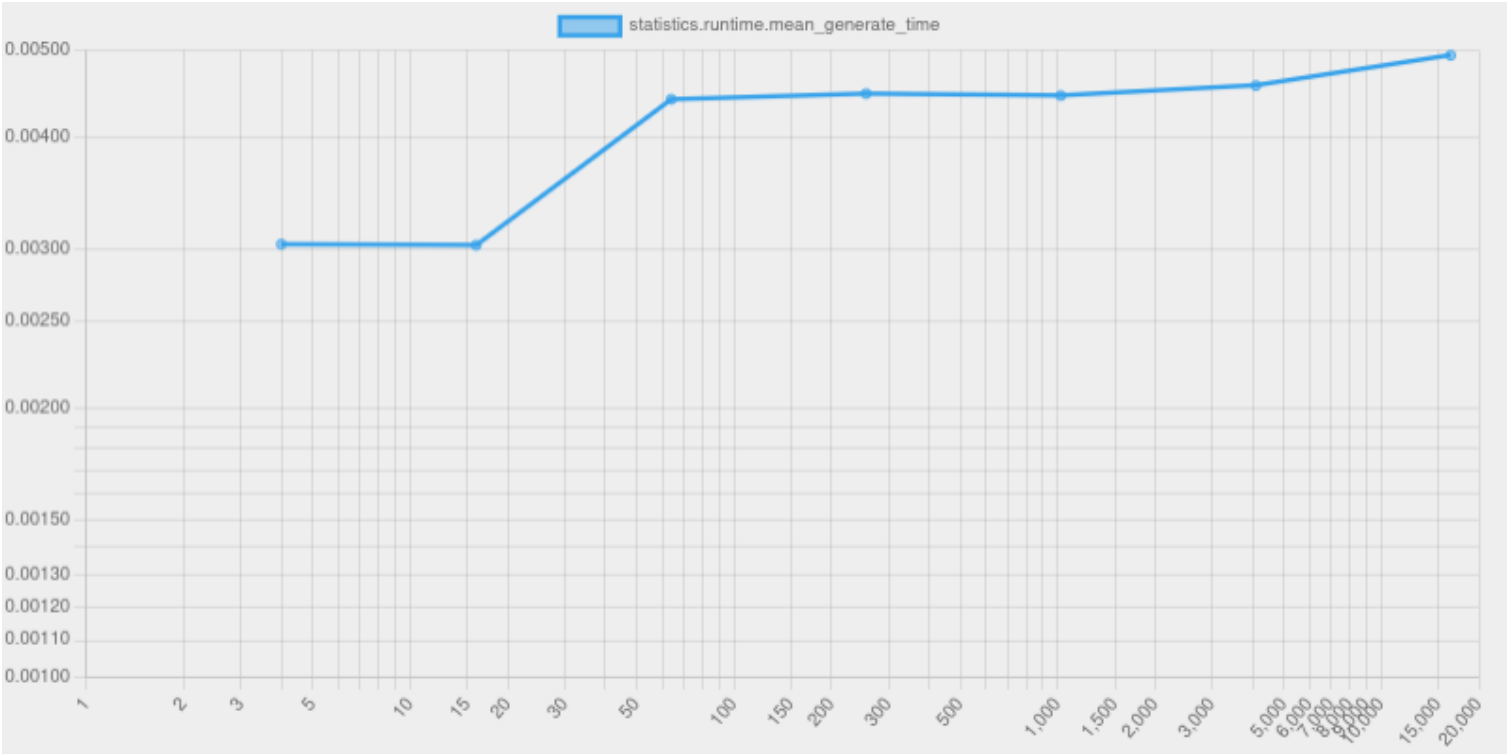
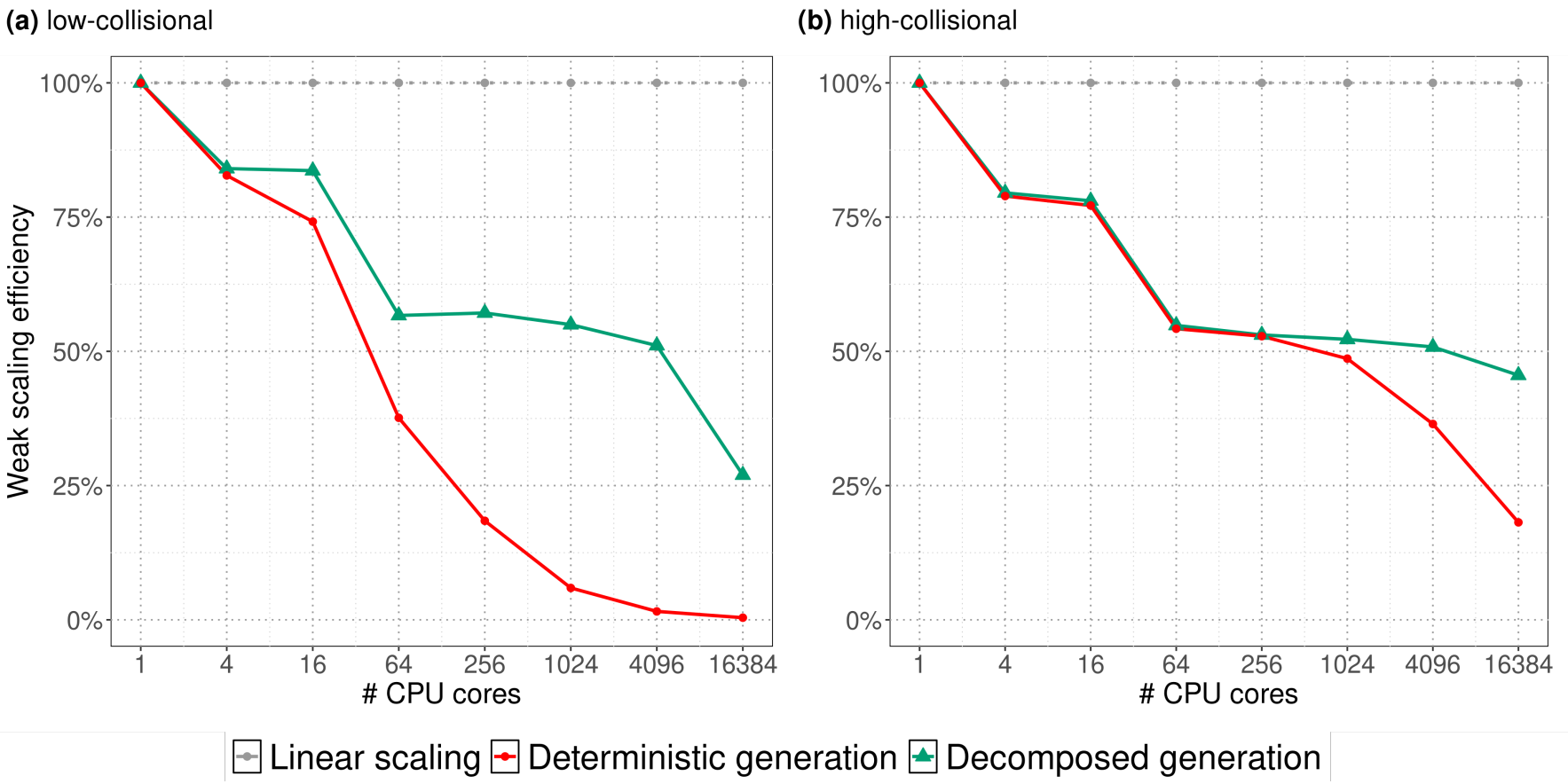


Figure: log-log plot with # cores on x-axis, part. generation time on y-axis (decomposed method). Weak scaling experiment.

Decomposed method

What we ended up doing is to decompose the sources into independent subsources, which keeps the generation cost low. What particle trajectories that are generated now depends on the source decomposition.

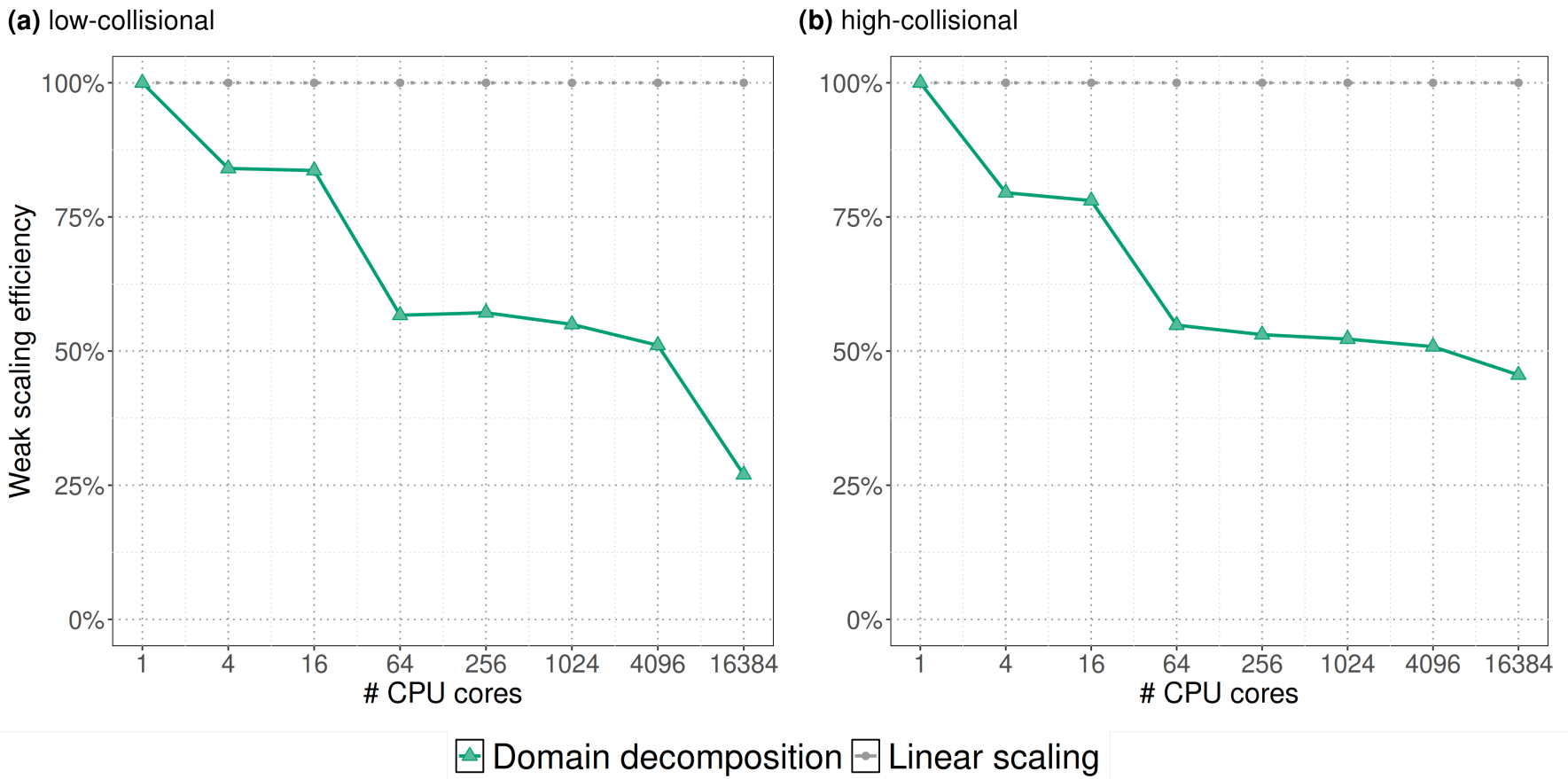
Particle generation bottleneck



Filtering particles

Decomposition by subdomain works, as we can see here. A Quasi-Monte Carlo method could potentially be better, allowing for both determinism and performance.

Scalability drop to 50%



Simulation cost

The two big drops in efficiency at $1 \rightarrow 4$ cores and $16 \rightarrow 64$ are not due to comm. overhead, but simulation becoming slower. Suspect cache effects. The last low-collisional drop *is* due to communication overhead.

Model limitations

1) The cross section Σ_t is based on constant collision rates ν_k, one scalar per collision process k:

$$\Sigma_t = \sum_k \frac{\nu_k}{|v|}$$

---

2) Eiron has two simple scattering collisions: isotropic rotations (dummy collision) and a globally constant BGK distribution.

---

3) Eiron only supports 2D square grids

The plan is to improve on the situation

1) implement more collision rate models, starting with AMJUEL
2) implement elastic collisions and cell-wise BGK collisions
3) go to 3D (lowest priority)

Future work

TSVV-5/TSVV-K wants to continue the project.

1. Begin implementing the algorithm in EIRENE
2. Add physics discussed on previous slide
3. Investigate the compute performance drop at 4 and 64 cores
4. Begin porting the algorithm to GPUs

Thank you

Questions?

`https://version.helsinki.fi/lapposka/eiron`

[1] V. Quadri et al. "Edge plasma turbulence simulations in detached regimes with the SOLEDGE3X code". In: *Nuclear Materials and Energy* 41 (2024), p. 101756. ISSN: 2352-1791. DOI: https://doi.org/10.1016/j.nme.2024.101756.

[2] D.V. Borodin et al. "Fluid, kinetic and hybrid approaches for neutral and trace ion edge transport modelling in fusion devices". In: *Nuclear Fusion* 62.8 (Aug. 1, 2022), p. 086051. ISSN: 0029-5515, 1741-4326. DOI: 10.1088/1741-4326/ac3fe8.

[3] Oskar Lappi et al. *Scalable Domain-decomposed Monte Carlo Neutral Transport for Nuclear Fusion*. Nov. 6, 2025. DOI: 10.48550/arXiv.2511.04489. arXiv: 2511.04489. URL: http://arxiv.org/abs/2511.04489.

[4] Oskar Lappi et al. *2D implementation of Kinetic-diffusion Monte Carlo in Eiron*. Sept. 23, 2025. DOI: 10.48550/arXiv.2509.19140. arXiv: 2509.19140. URL: http://arxiv.org/abs/2509.19140.