Turning Contention Into Cooperation:

Reducing the cost of synchronized global data structures in Grappa

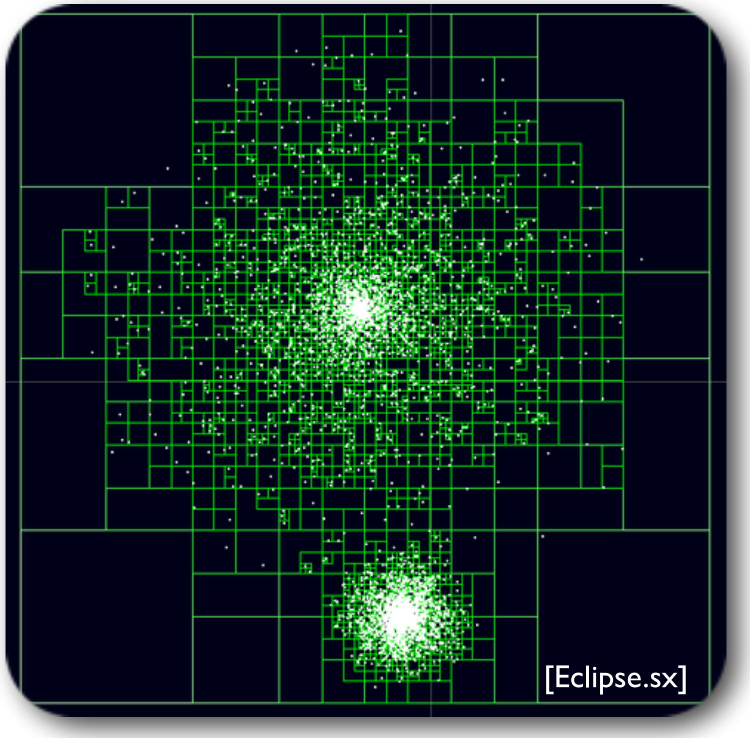
© Disney, Inc. *Fantasia (The Pastoral Symphony)*

simple, distributed, batched synchronization

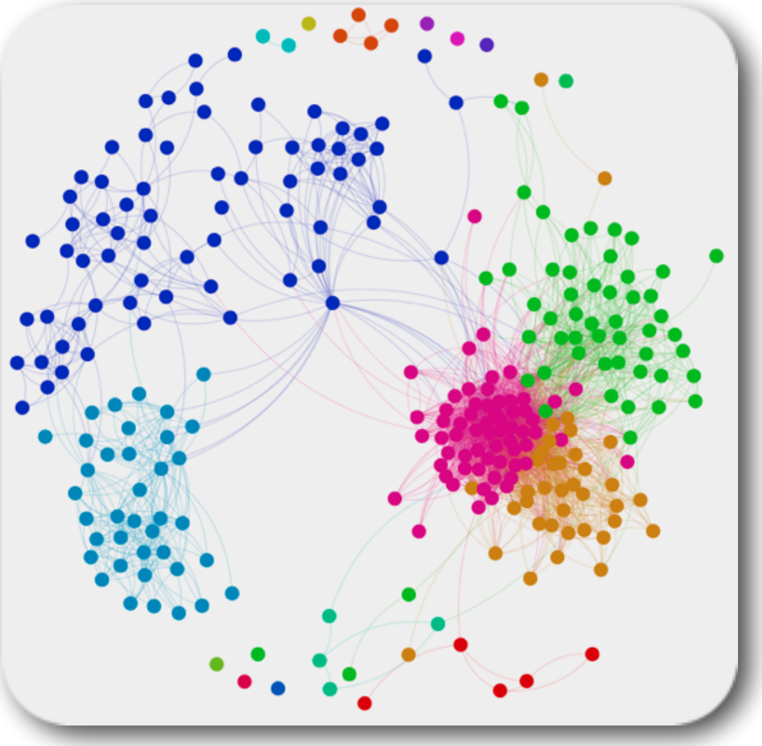
sequential consistency at cluster scale

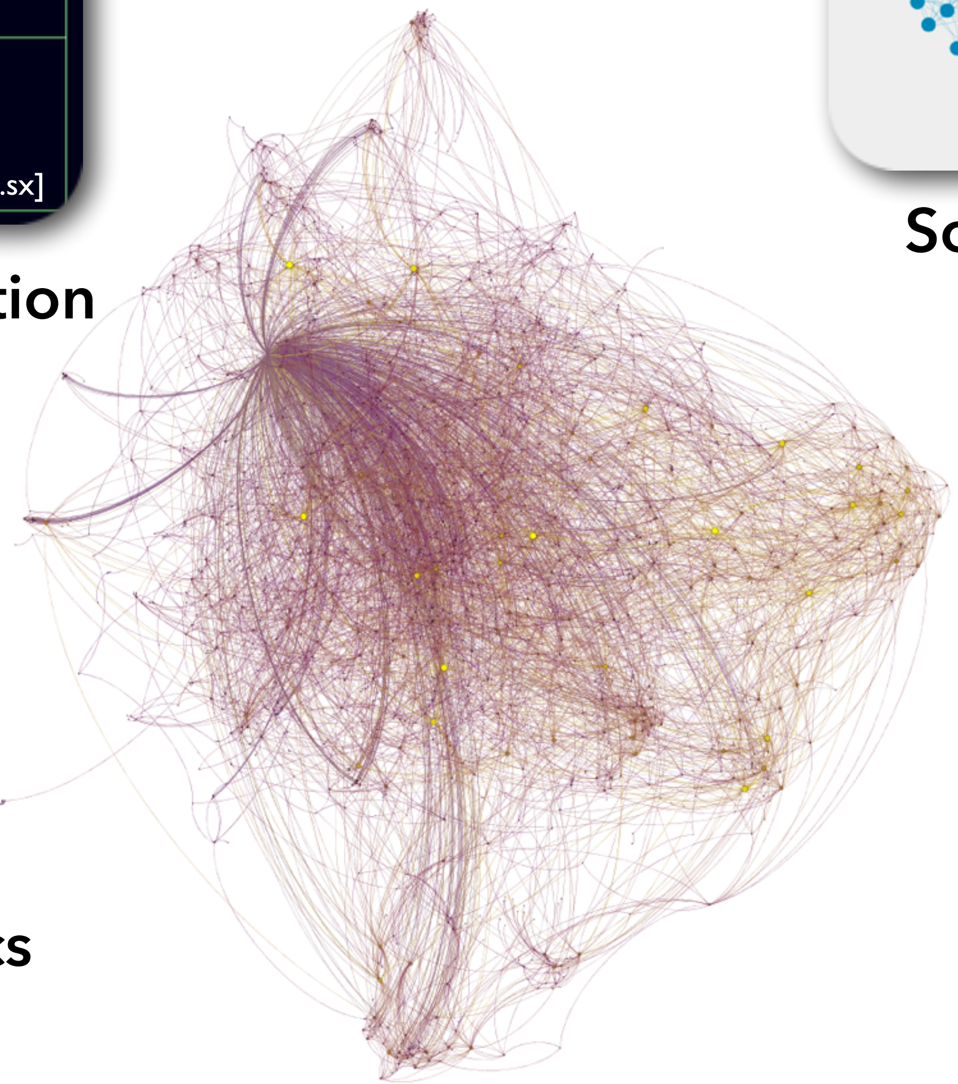
Brandon Holt, Jacob Nelson, Brandon Myers,
Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin

Irregular Applications

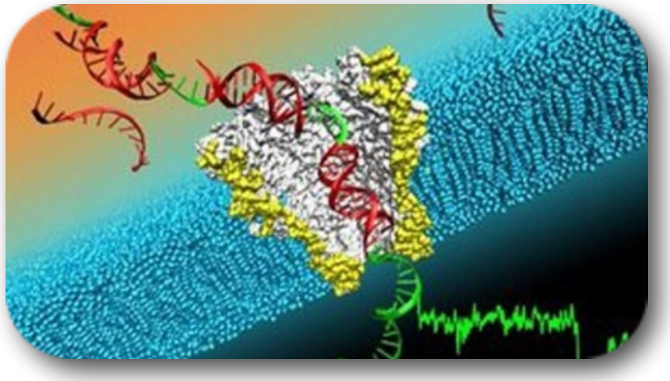


Barnes-Hut n-body simulation


Social network analysis



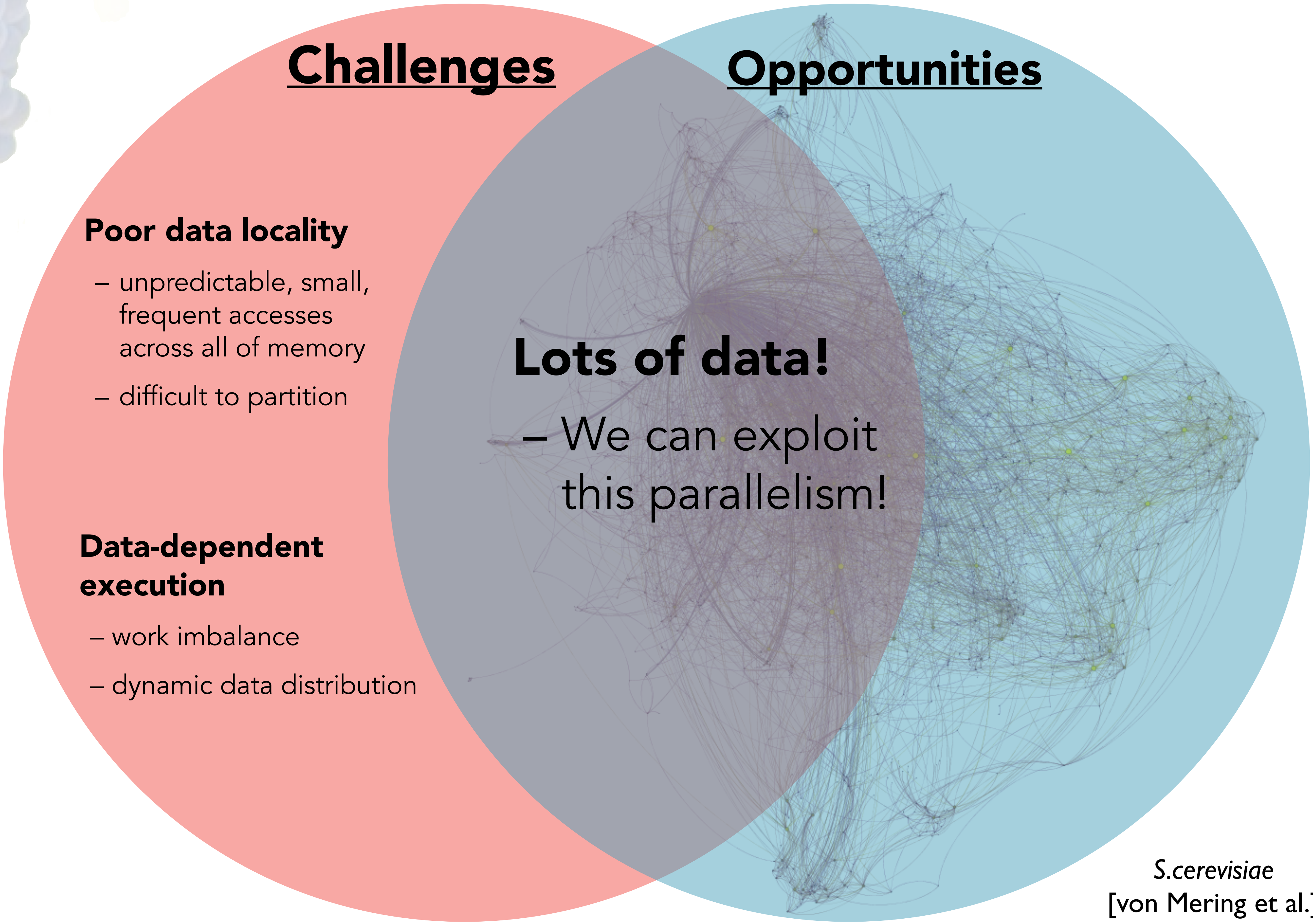
Fraud detection


Bioinformatics

Machine learning

Clustering

Irregular Applications



S.cerevisiae
[von Mering et al.]

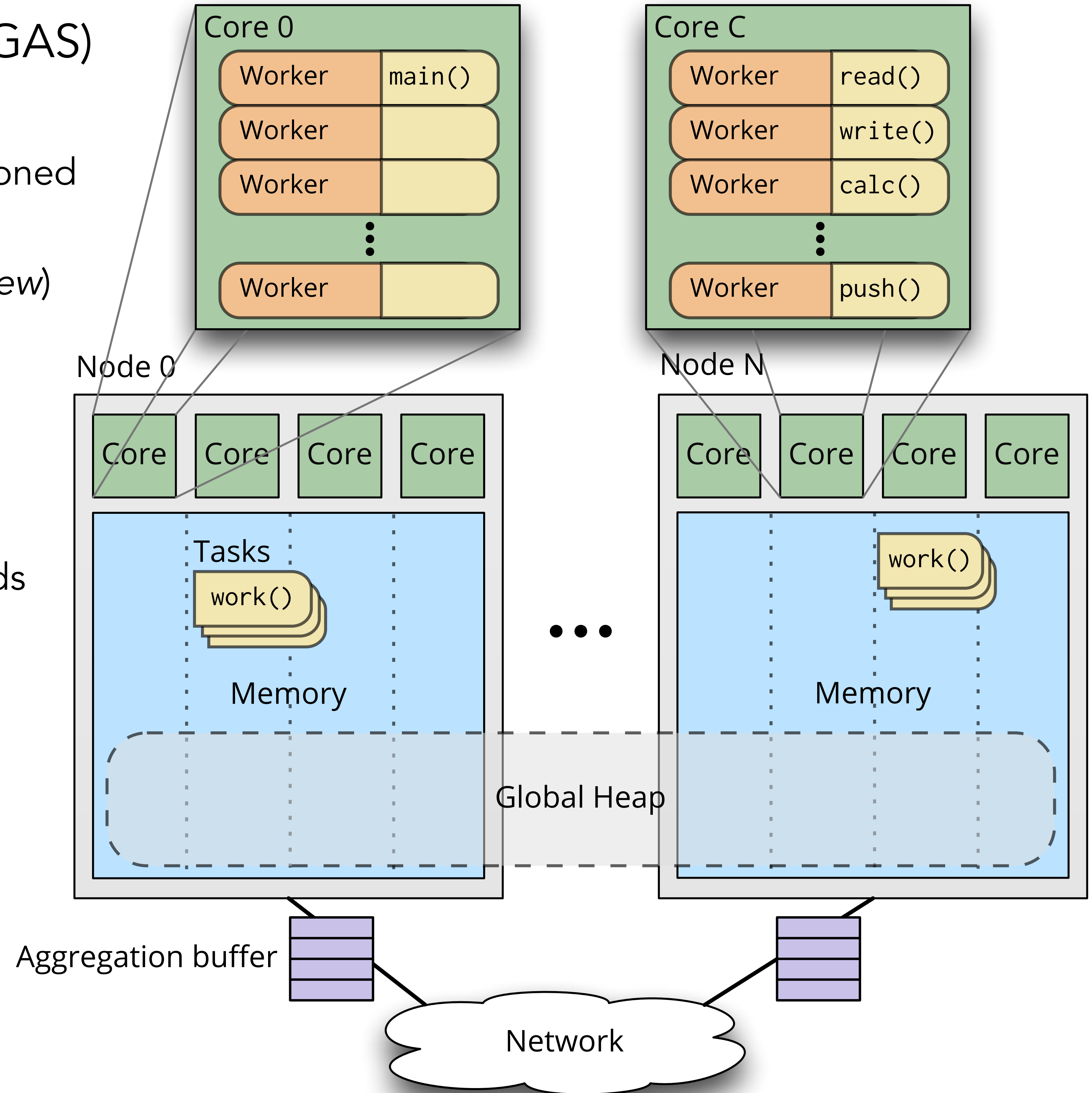
Grappa: a latency-tolerant PGAS runtime

Partitioned Global Address Space (PGAS) programming model

- memory distributed over cluster and partitioned among cores
- programmed as a single machine (*global view*)
- C++11 library interface

Runtime capabilities:

- **Aggregated** communication
- Cooperatively-scheduled lightweight threads for **latency tolerance**
- Access other cores' data *only* via delegate operations
- Sequential consistency



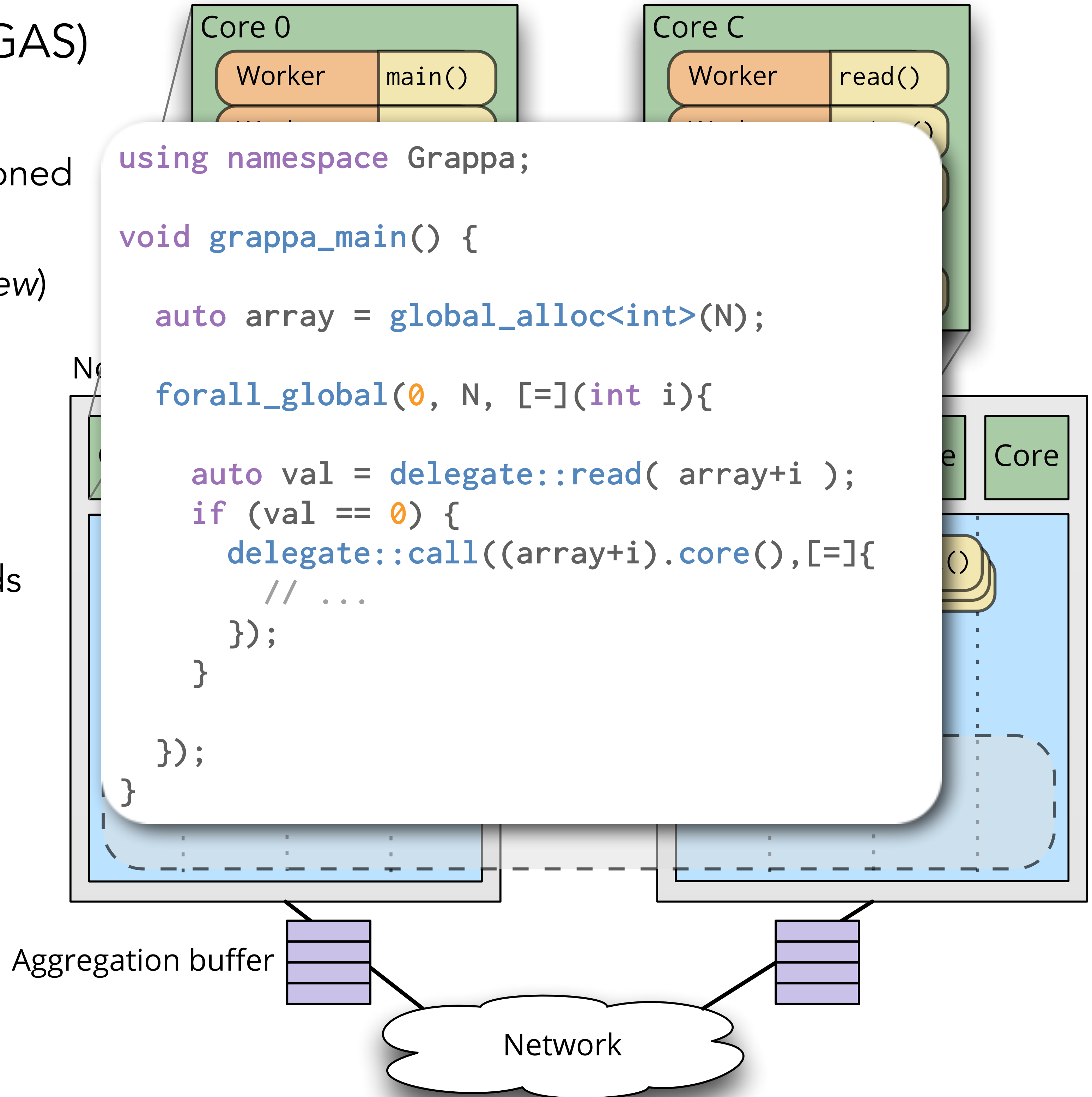
Grappa: a latency-tolerant PGAS runtime

Partitioned Global Address Space (PGAS) programming model

- memory distributed over cluster and partitioned among cores
- programmed as a single machine (*global view*)
- C++11 library interface

Runtime capabilities:

- **Aggregated** communication
- Cooperatively-scheduled lightweight threads for **latency tolerance**
- Access other cores' data *only* via delegate operations
- Sequential consistency




```
using namespace Grappa;

void grappa_main() {

  auto array = global_alloc<int>(N);

  forall_global(0, N, [=](int i){

    auto val = delegate::read( array+i );
    if (val == 0) {
      delegate::call((array+i).core(),[=]{
        // ...
      });
    }

  });
}
```

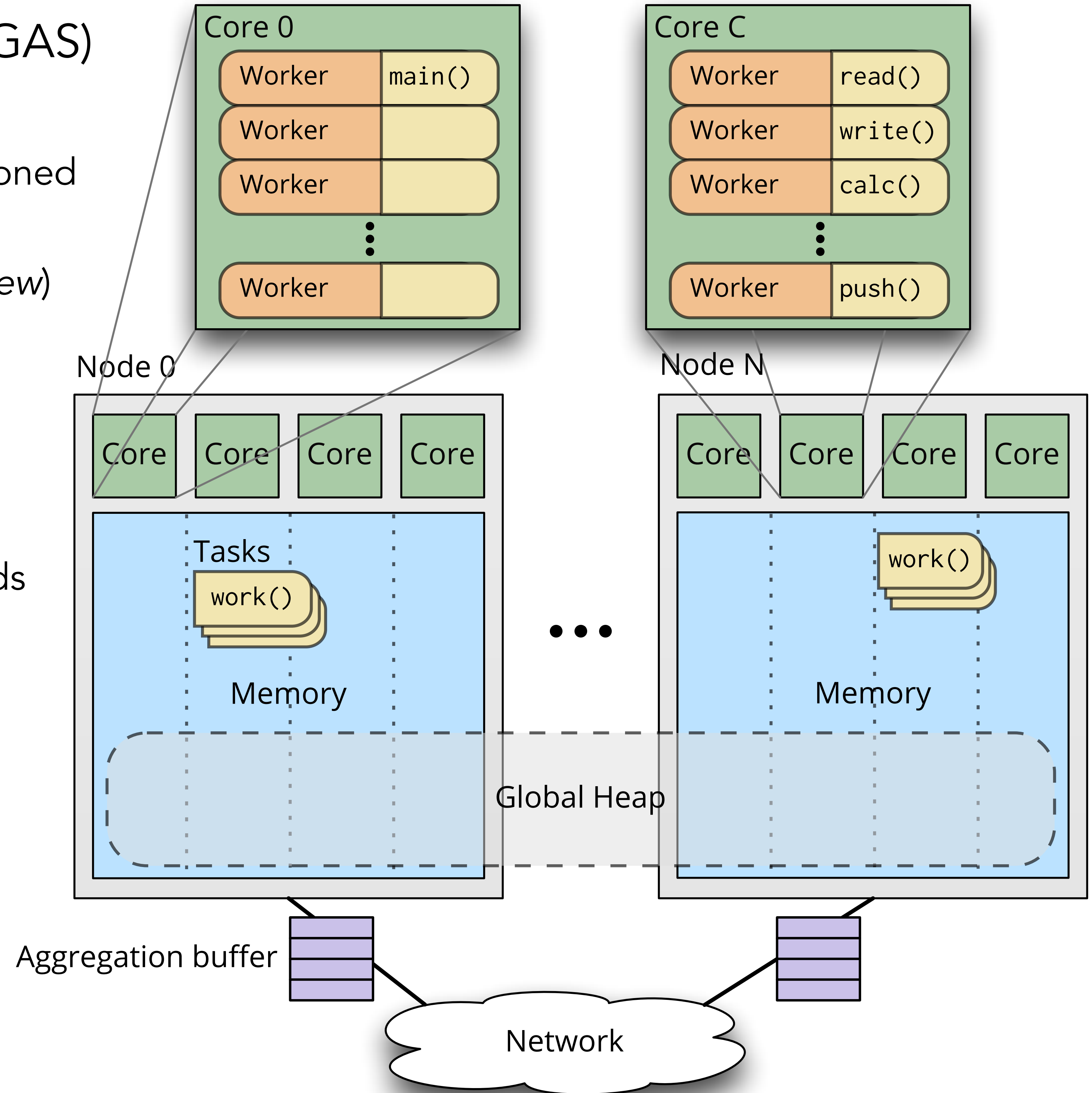
Grappa: a latency-tolerant PGAS runtime

Partitioned Global Address Space (PGAS) programming model

- memory distributed over cluster and partitioned among cores
- programmed as a single machine (*global view*)
- C++11 library interface

Runtime capabilities:

- **Aggregated** communication
- Cooperatively-scheduled lightweight threads for **latency tolerance**
- Access other cores' data *only* via delegate operations
- Sequential consistency



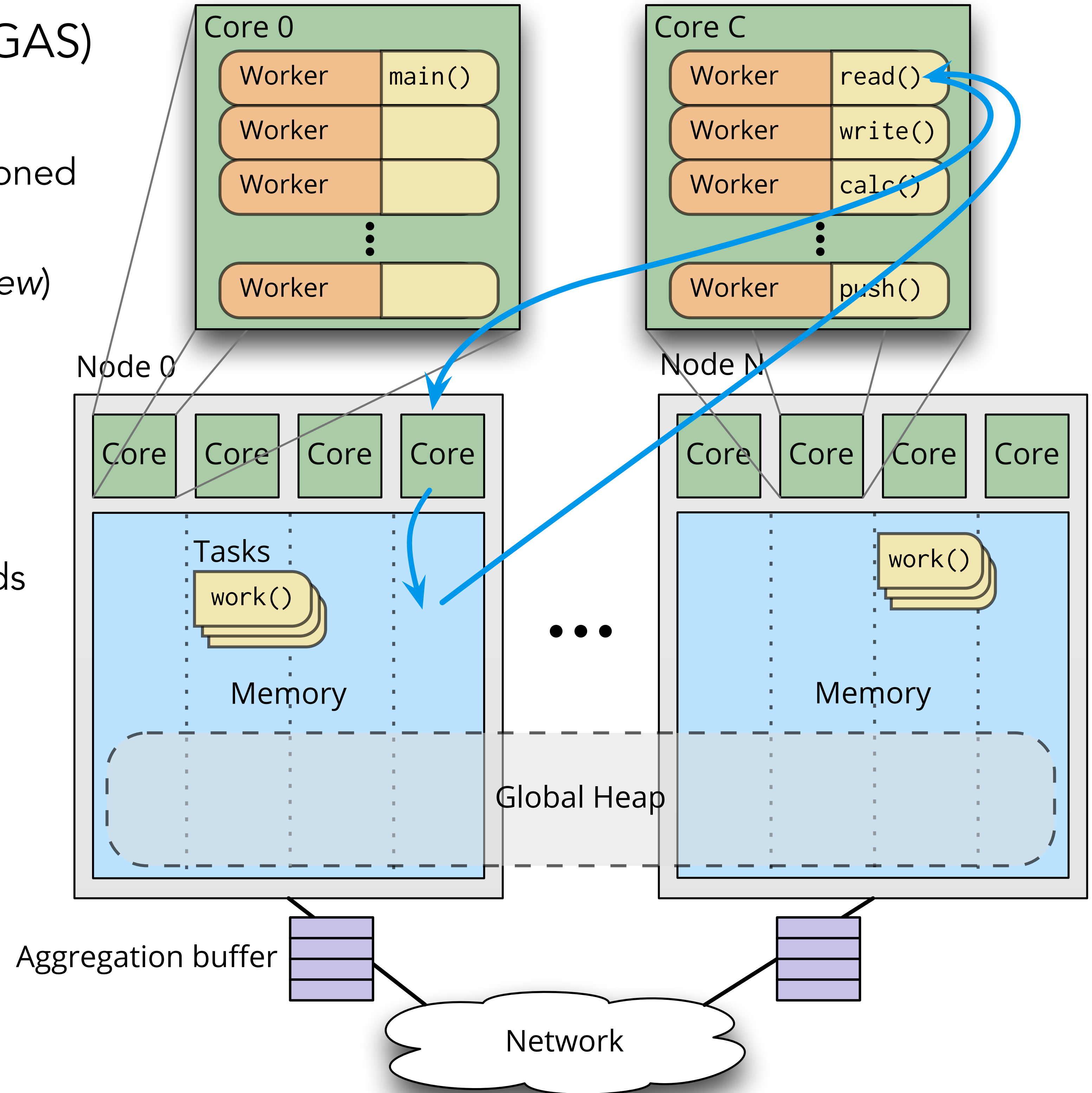
Grappa: a latency-tolerant PGAS runtime

Partitioned Global Address Space (PGAS) programming model

- memory distributed over cluster and partitioned among cores
- programmed as a single machine (*global view*)
- C++11 library interface

Runtime capabilities:

- **Aggregated** communication
- Cooperatively-scheduled lightweight threads for **latency tolerance**
- Access other cores' data *only* via delegate operations
- Sequential consistency



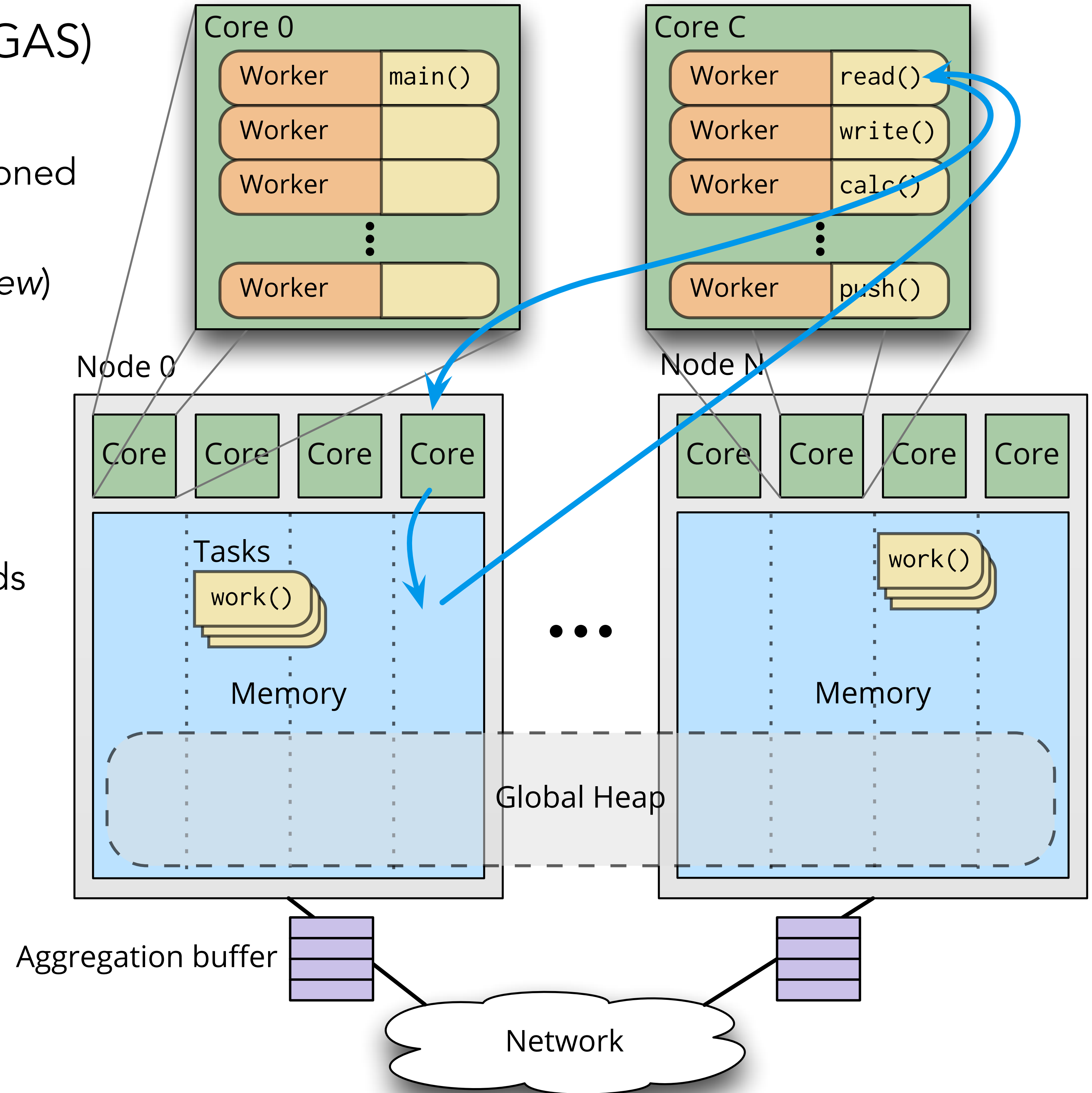
Grappa: a latency-tolerant PGAS runtime

Partitioned Global Address Space (PGAS) programming model

- memory distributed over cluster and partitioned among cores
- programmed as a single machine (*global view*)
- C++11 library interface

Runtime capabilities:

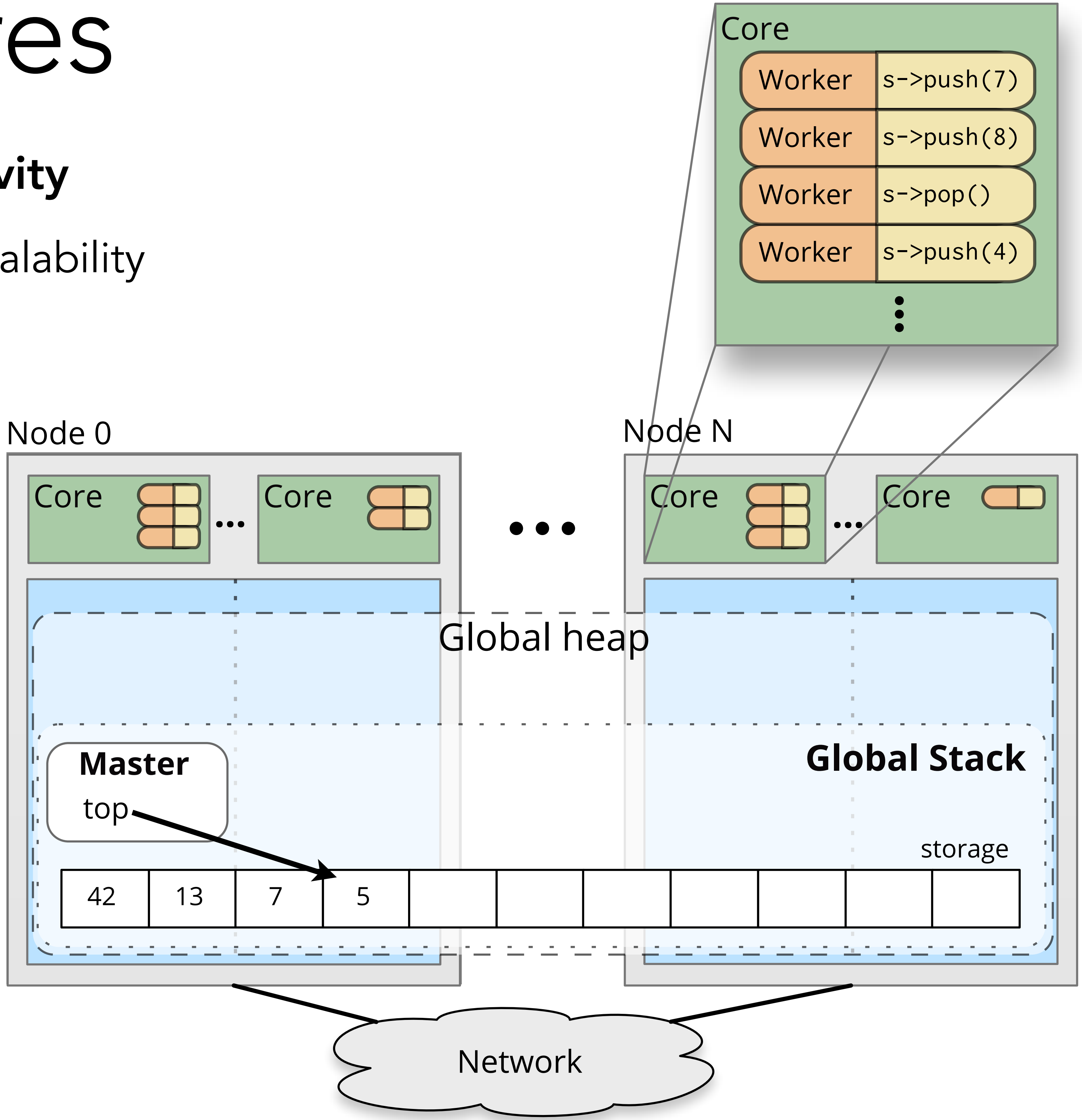
- **Aggregated** communication
- Cooperatively-scheduled lightweight threads for **latency tolerance**
- Access other cores' data *only* via delegate operations
- Sequential consistency



synchronized shared data structures

Standard library aids **productivity**

Generality costs performance/scalability

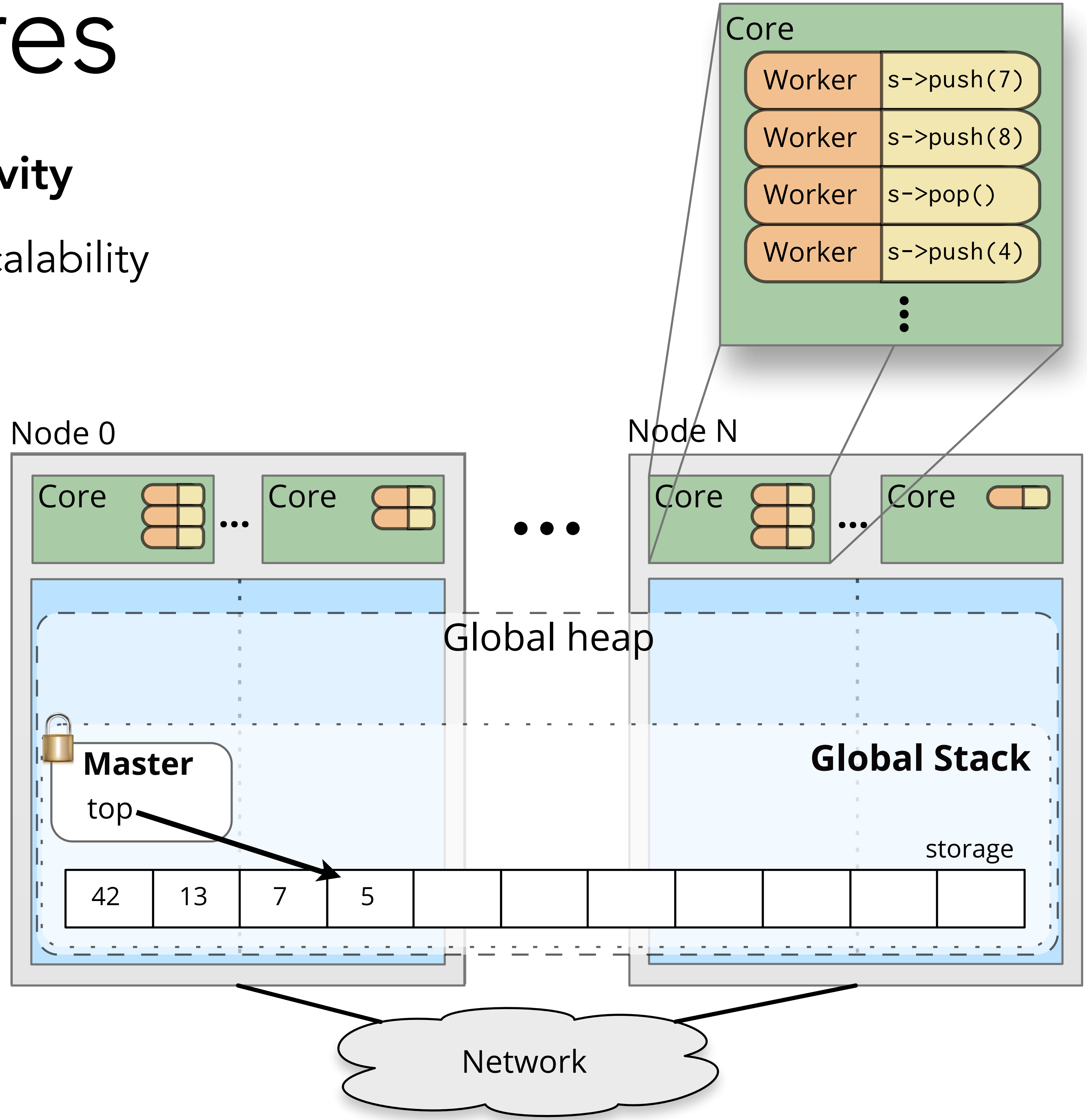


synchronized shared data structures

Standard library aids **productivity**

Generality costs performance/scalability

Must maintain **consistency**

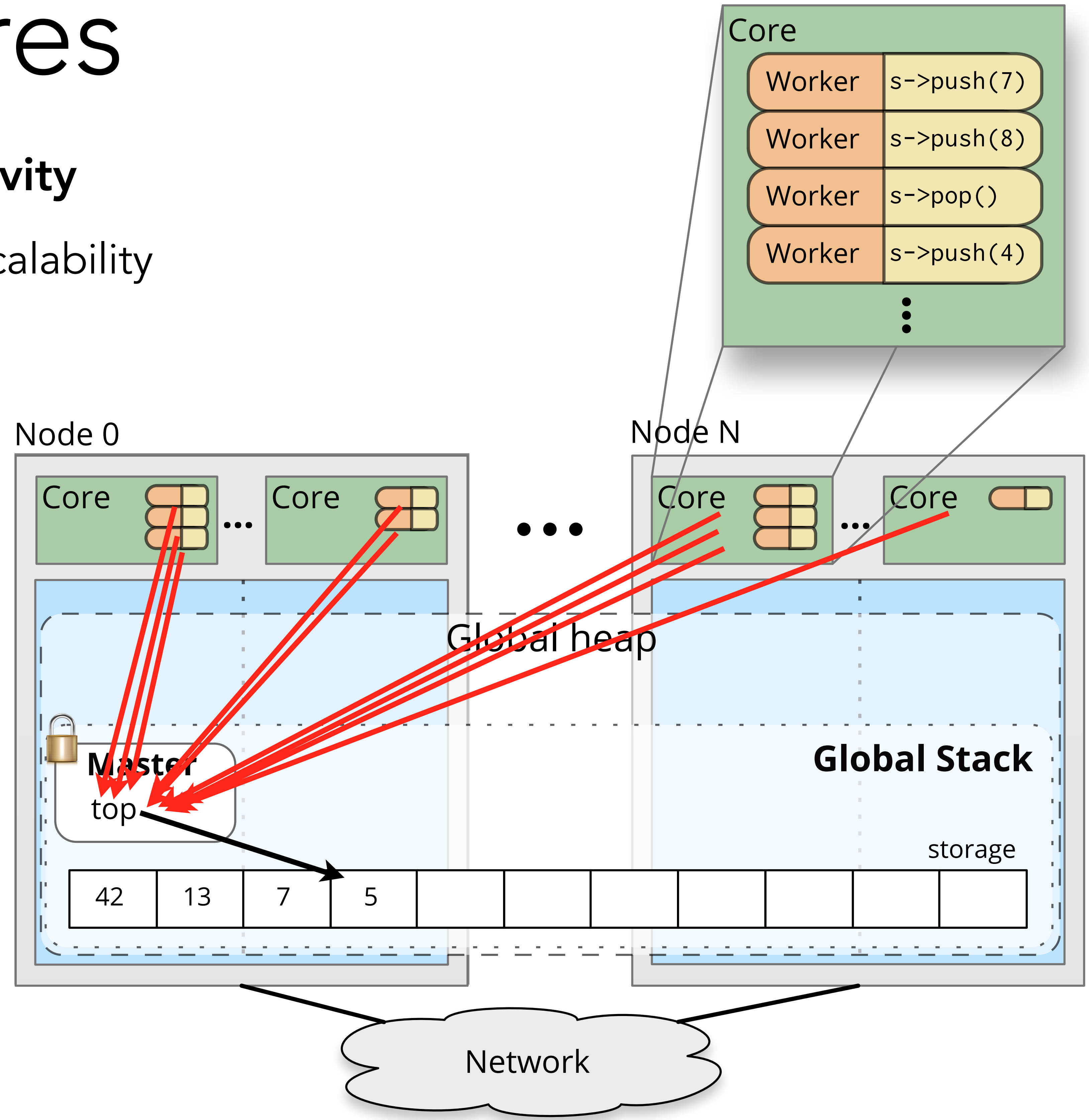


synchronized shared data structures

Standard library aids **productivity**

Generality costs performance/scalability

Must maintain **consistency**

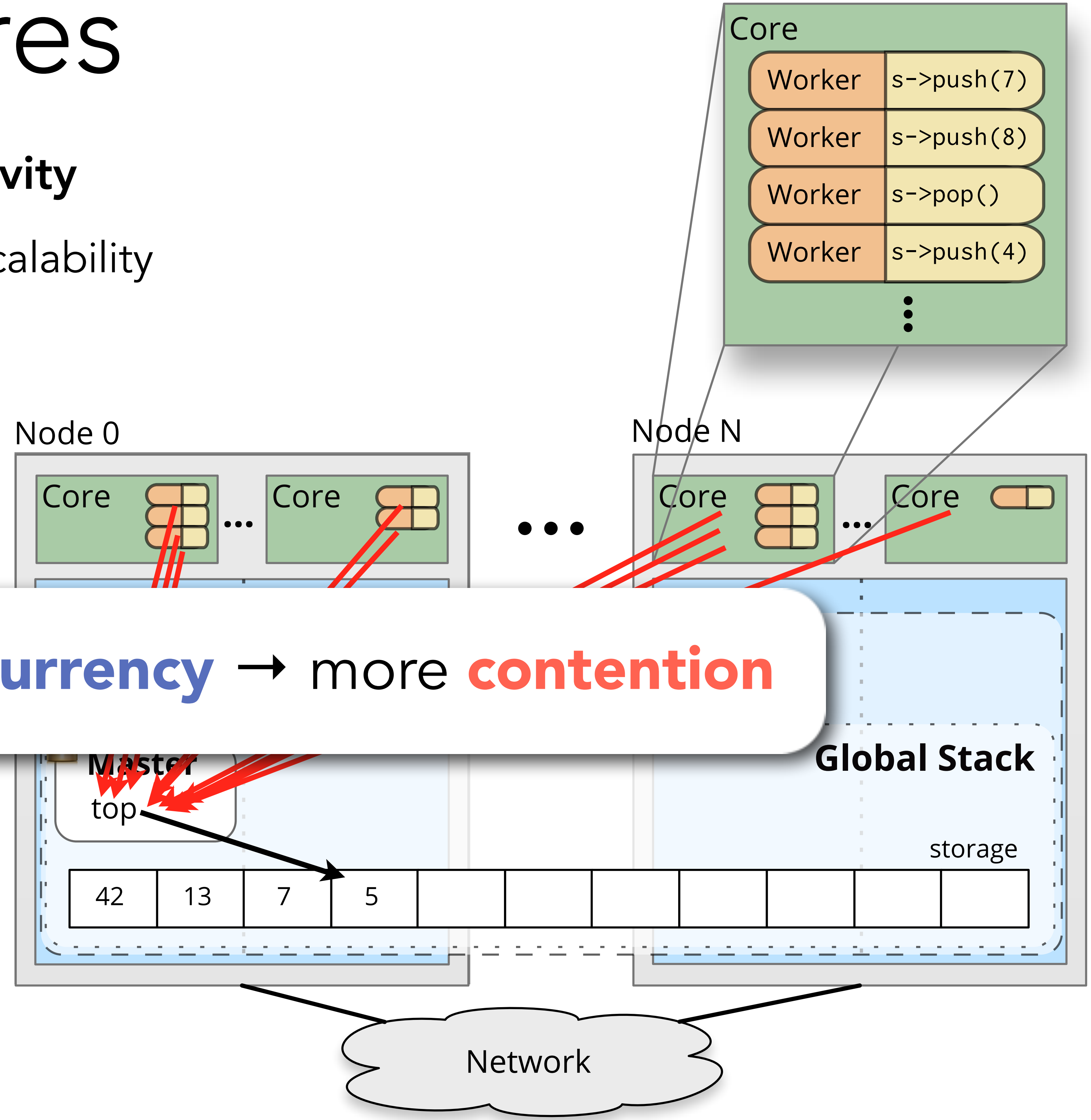


synchronized shared data structures

Standard library aids **productivity**

Generality costs performance/scalability

Must maintain **consistency**



more **concurrency** → more **contention**

contention → cooperation

contention → cooperation

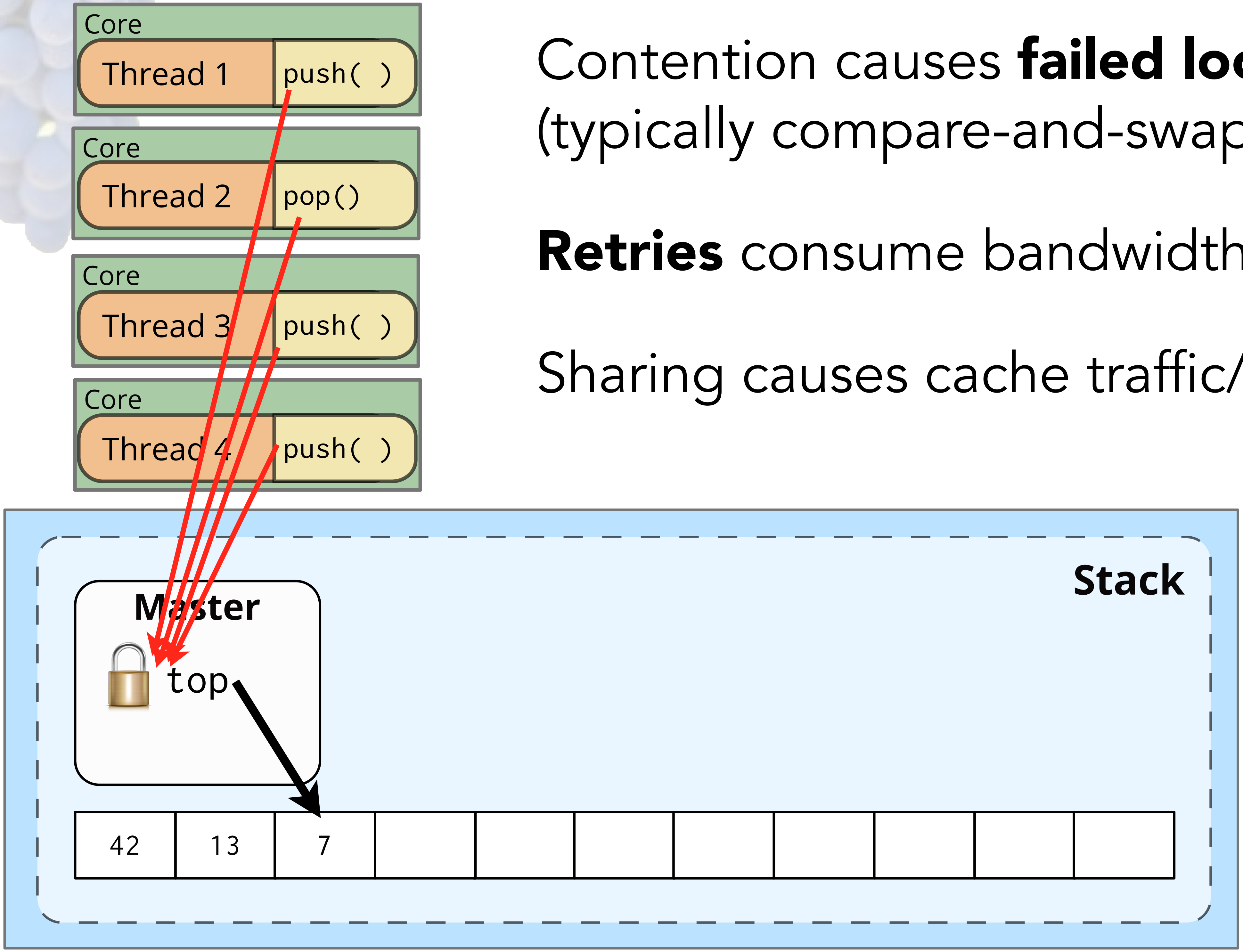
contention → cooperation

contention → cooperation

contention → cooperation

contention → cooperation

contention: global lock

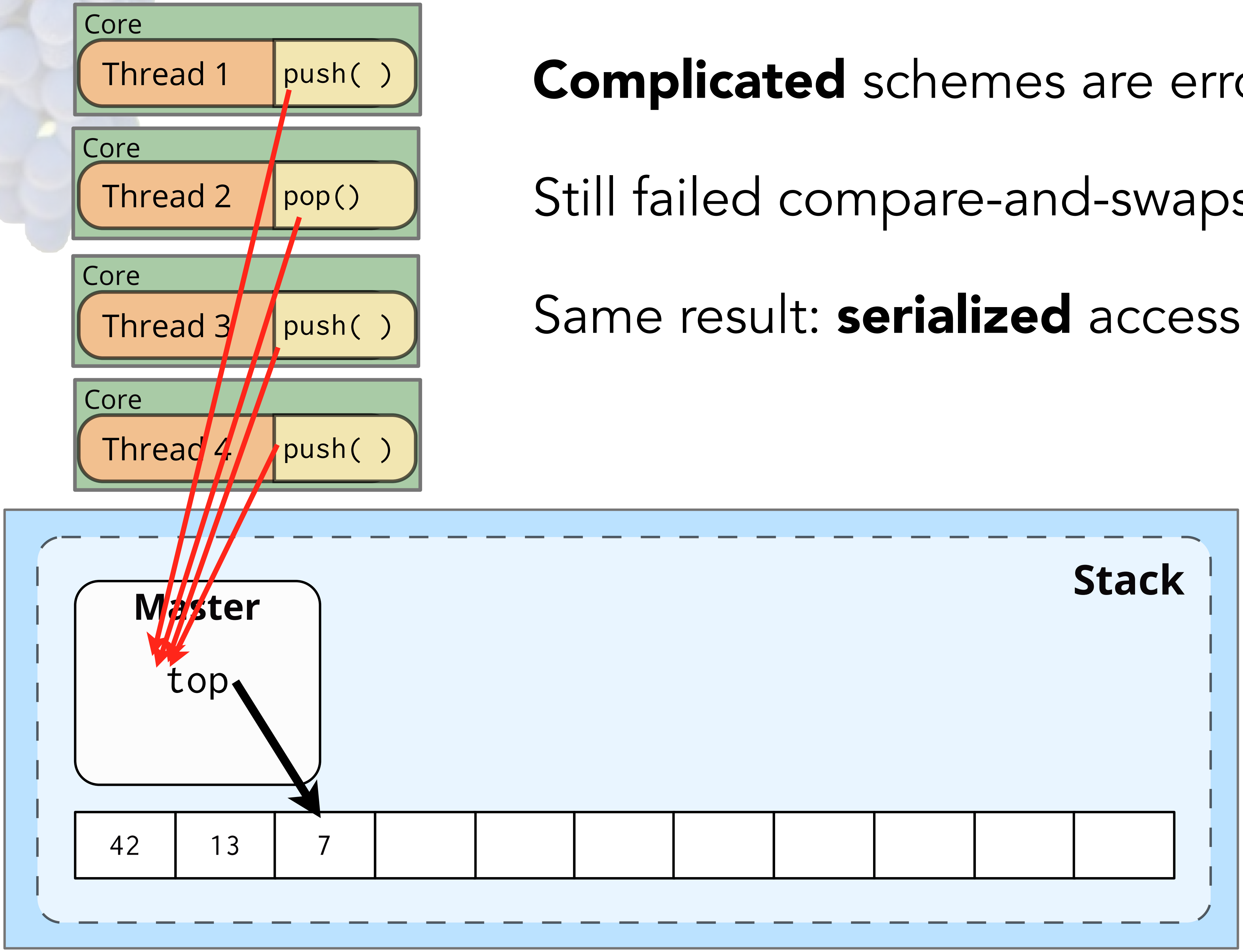


Contention causes **failed lock acquires** (typically compare-and-swaps)

Retries consume bandwidth

Sharing causes cache traffic/**thrashing**

contention: fine-grained sync

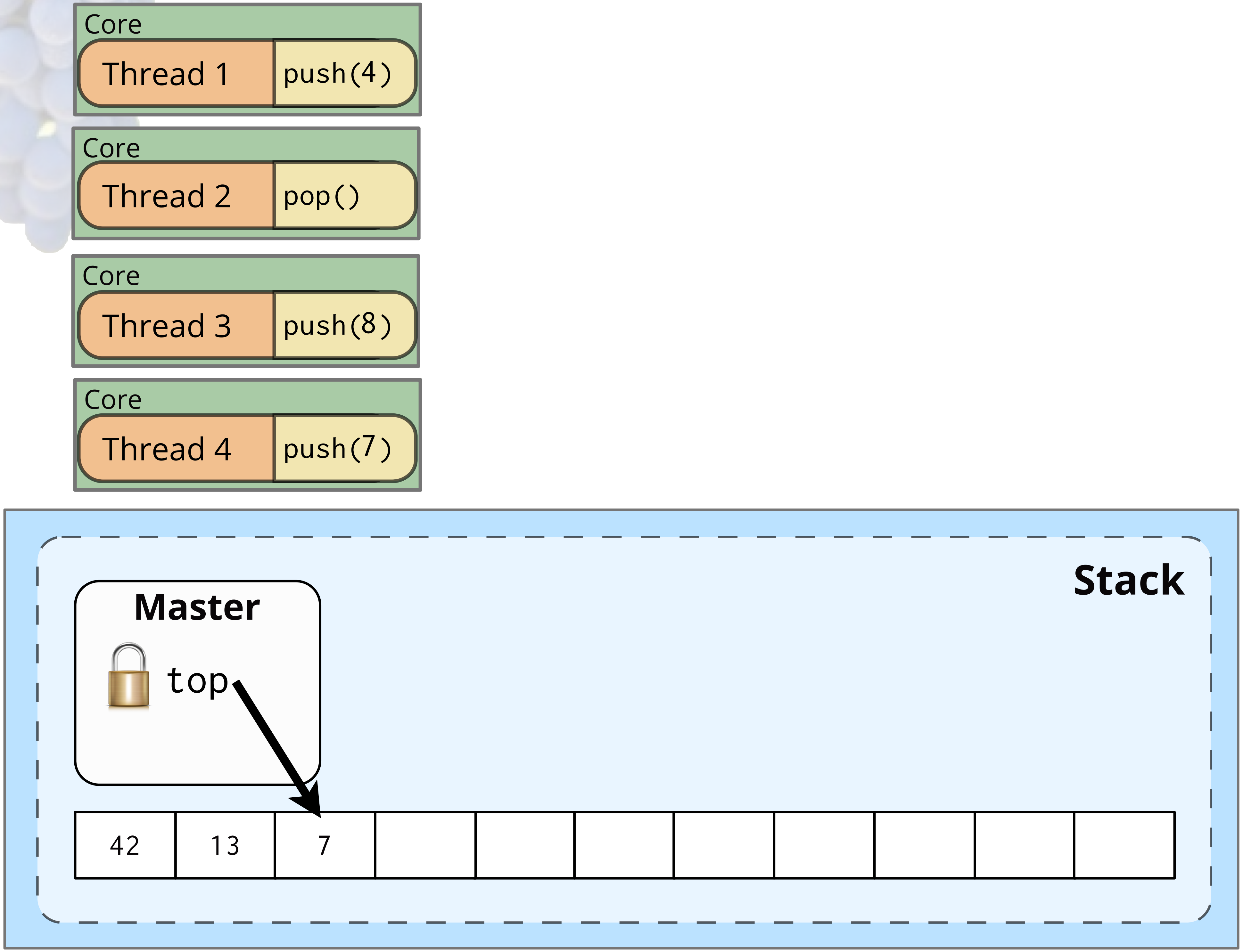


Complicated schemes are error-prone

Still failed compare-and-swaps and **retries**

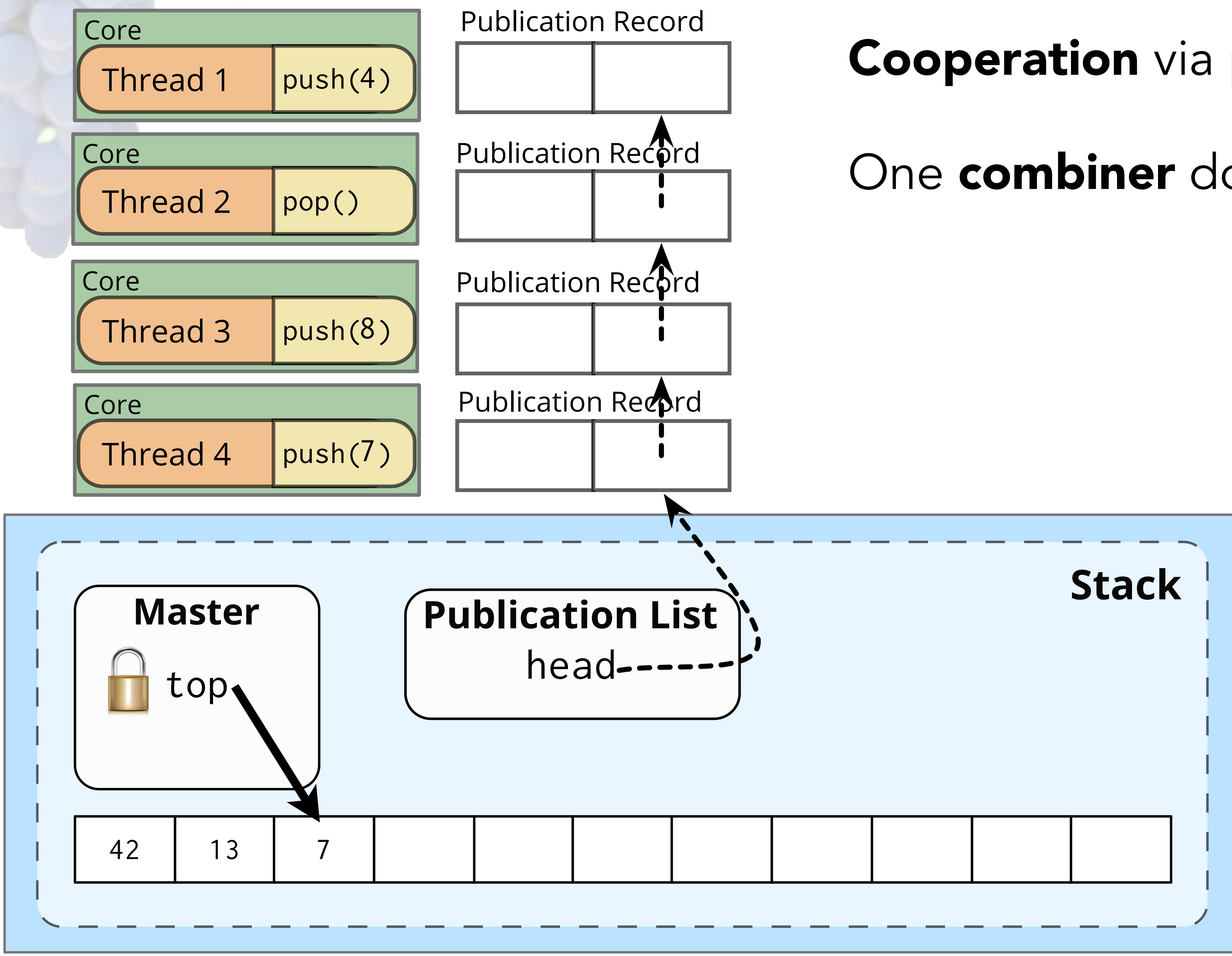
Same result: **serialized** access

cooperation: flat combining [1]



[1] *"Flat Combining and the Synchronization-Parallelism Tradeoff"*
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir
(SPAA '10)

cooperation: flat combining [1]

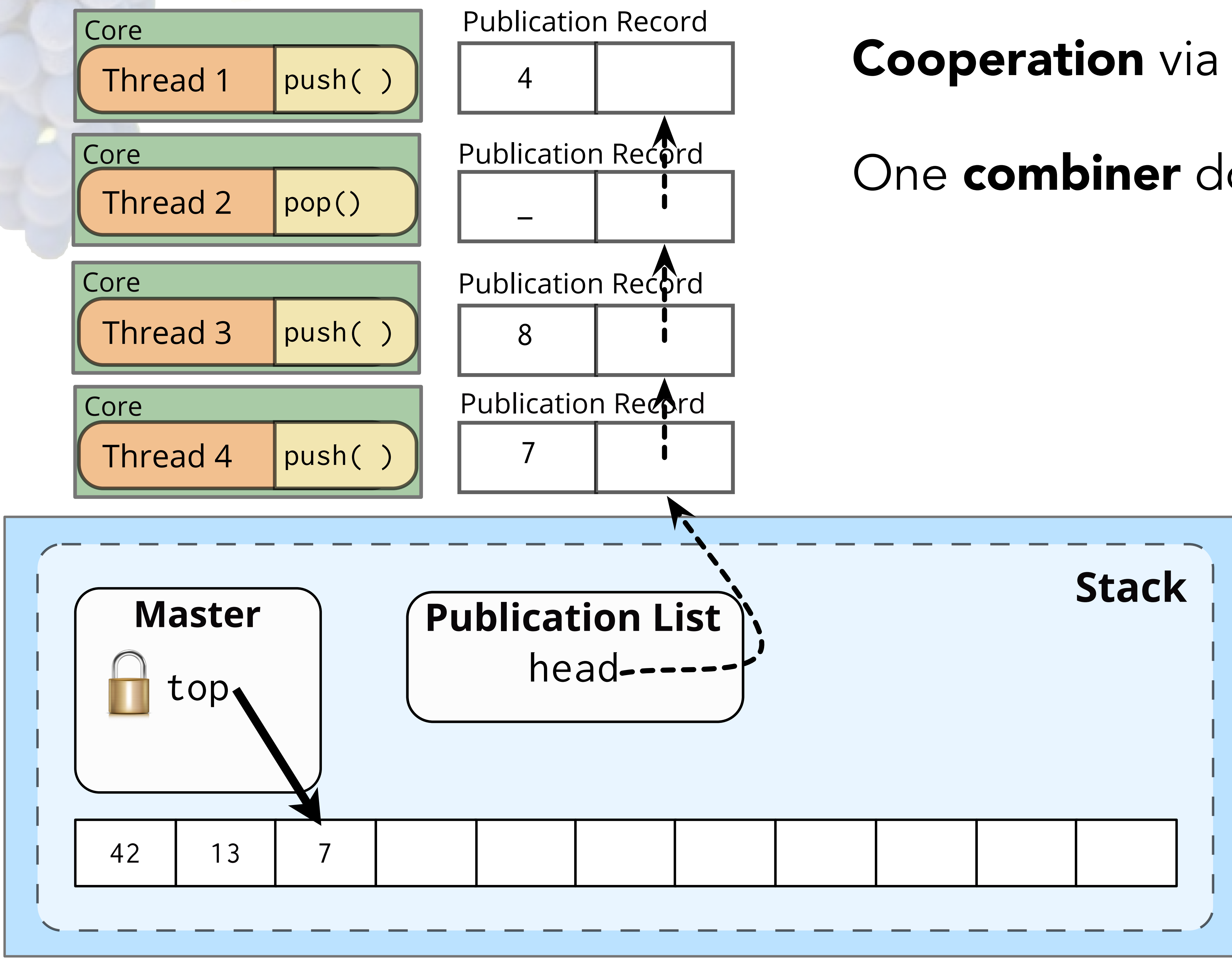


Cooperation via publication list

One **combiner** does all the work

[1] *"Flat Combining and the Synchronization-Parallelism Tradeoff"* Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA '10)

cooperation: flat combining [1]

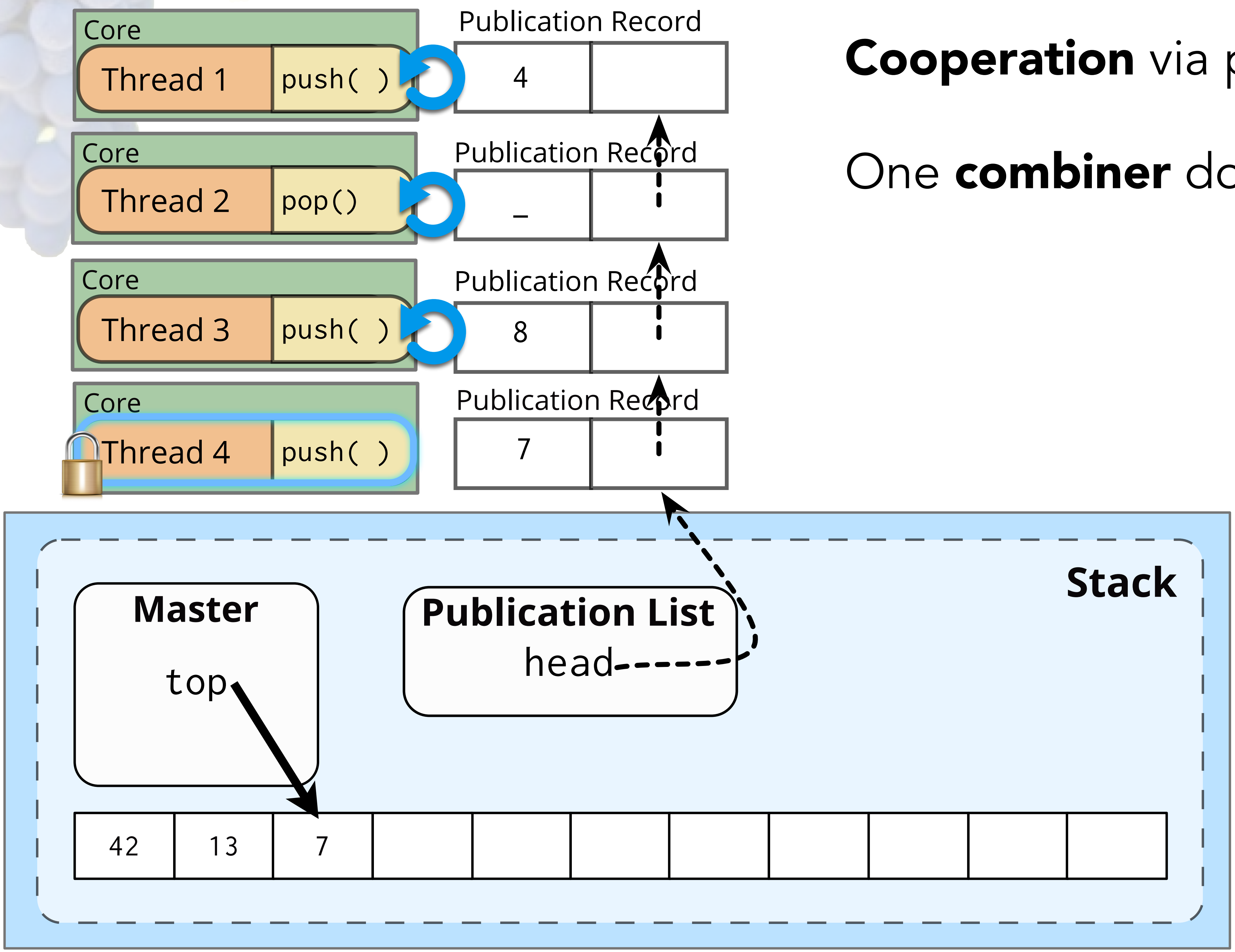


Cooperation via publication list

One **combiner** does all the work

[1] *"Flat Combining and the Synchronization-Parallelism Tradeoff"*
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA '10)

cooperation: flat combining [1]

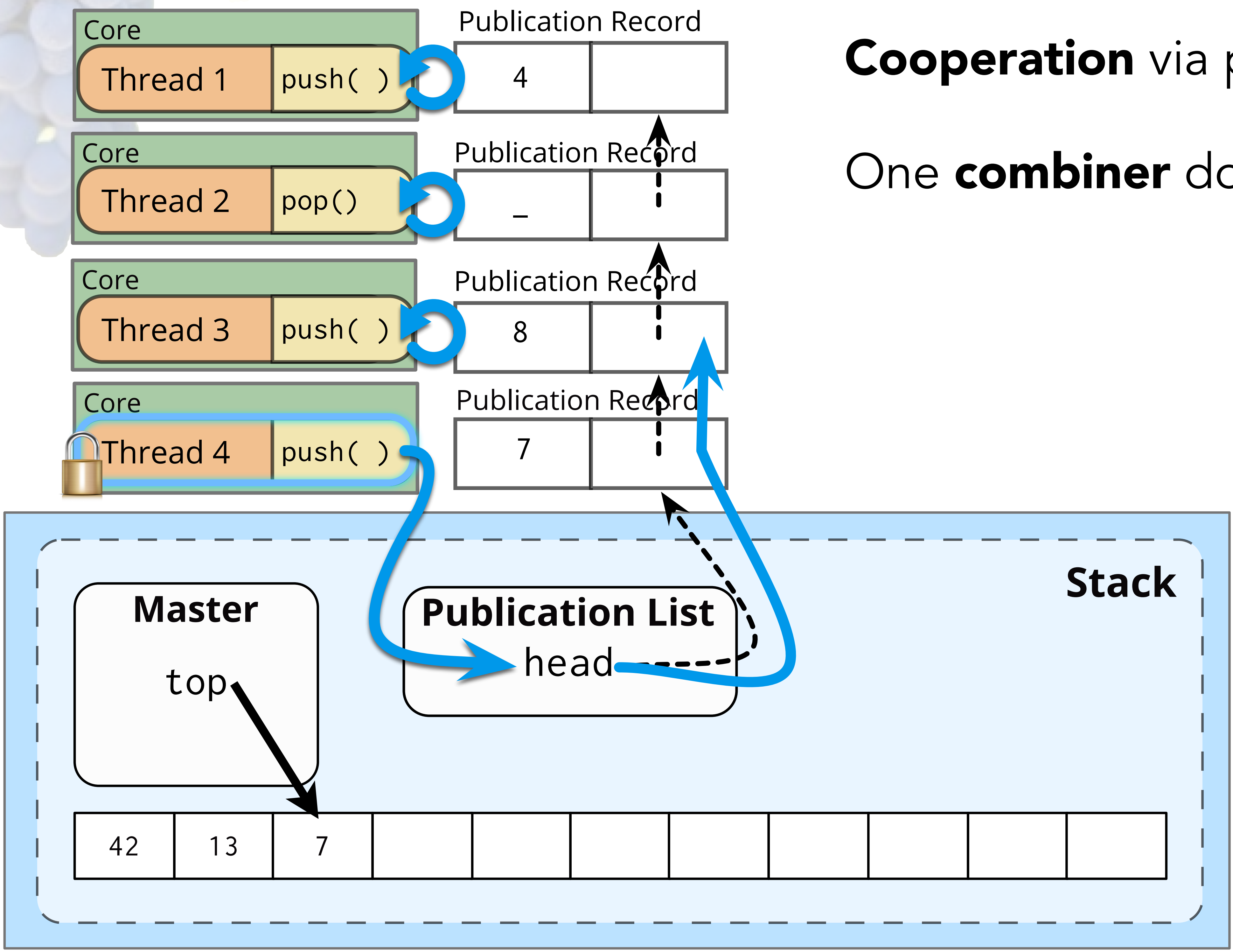


Cooperation via publication list

One **combiner** does all the work

[1] *"Flat Combining and the Synchronization-Parallelism Tradeoff"* Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA '10)

cooperation: flat combining [1]

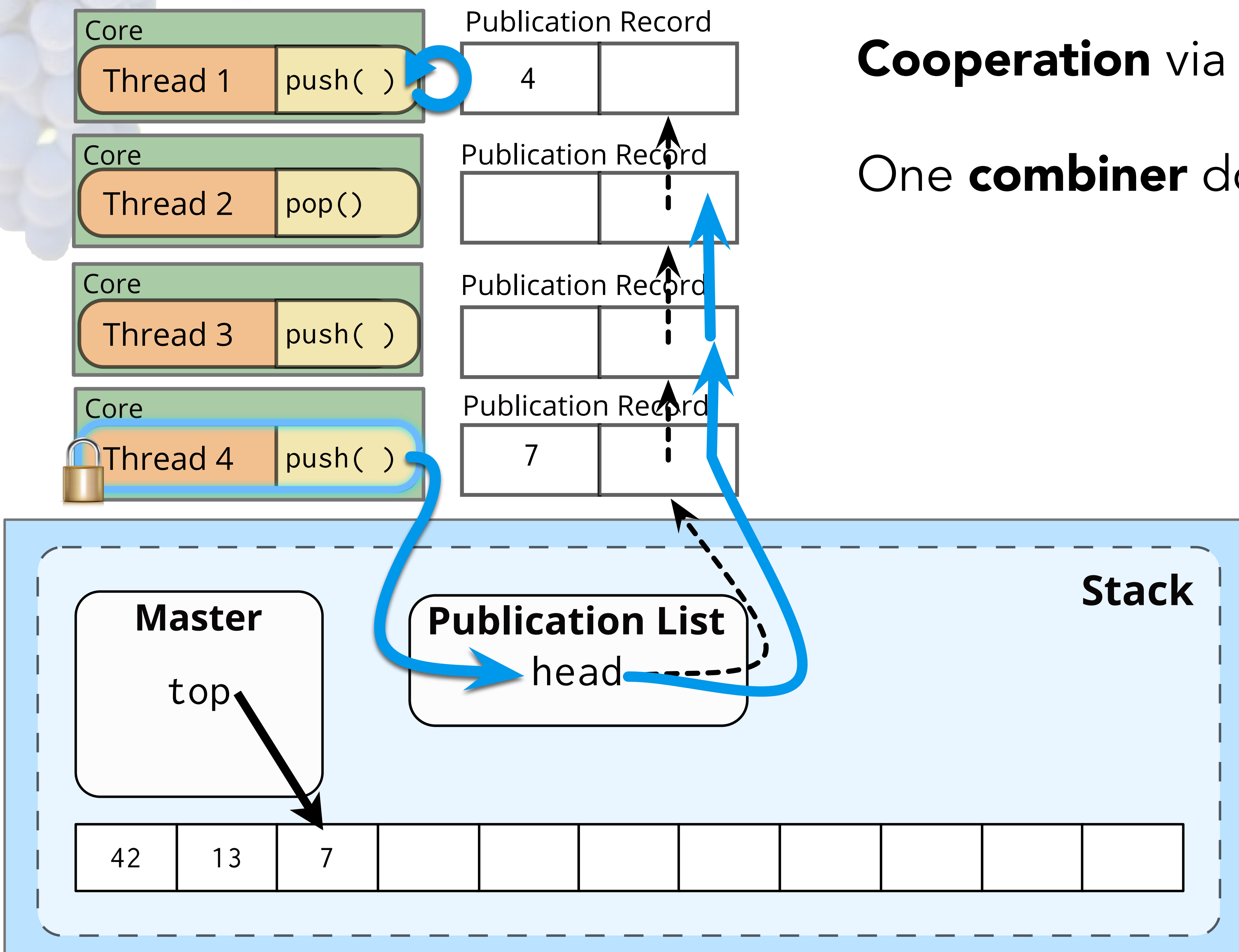


Cooperation via publication list

One **combiner** does all the work

[1] *"Flat Combining and the Synchronization-Parallelism Tradeoff"*
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA '10)

cooperation: flat combining

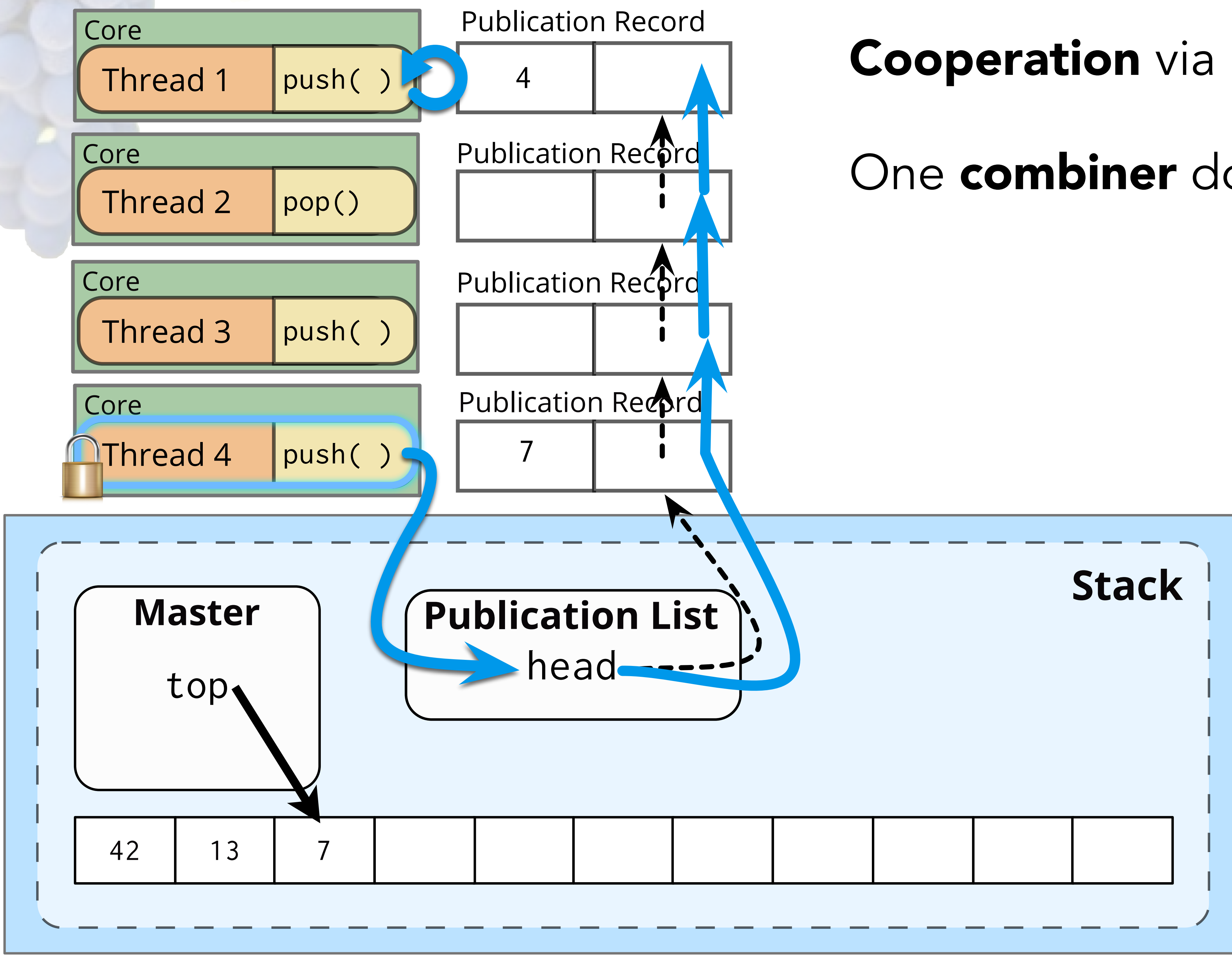


Cooperation via publication list

One **combiner** does all the work

[1] *"Flat Combining and the Synchronization-Parallelism Tradeoff"*
Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA '10)

cooperation: flat combining [1]

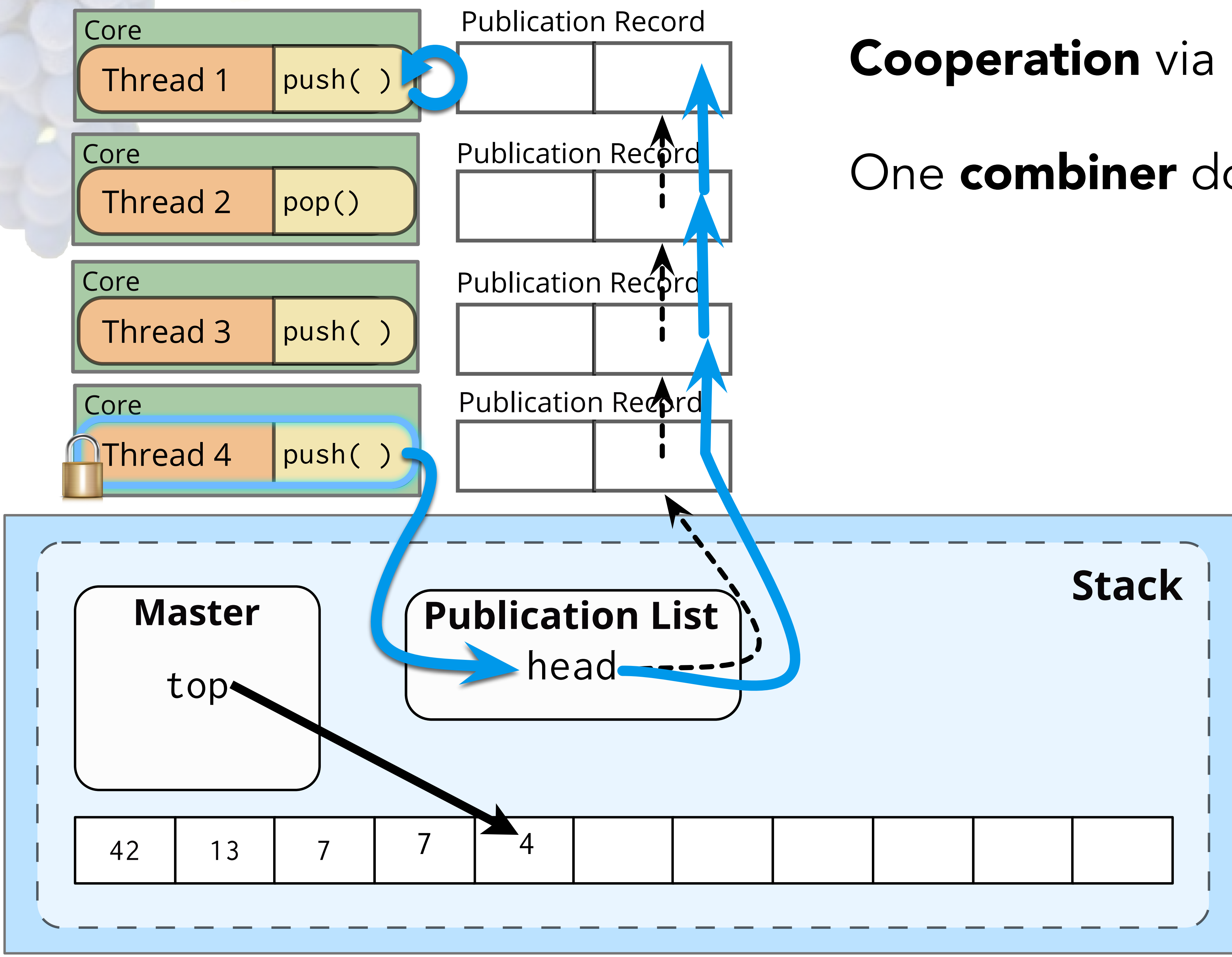


Cooperation via publication list

One **combiner** does all the work

[1] *"Flat Combining and the Synchronization-Parallelism Tradeoff"* Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA '10)

cooperation: flat combining [1]

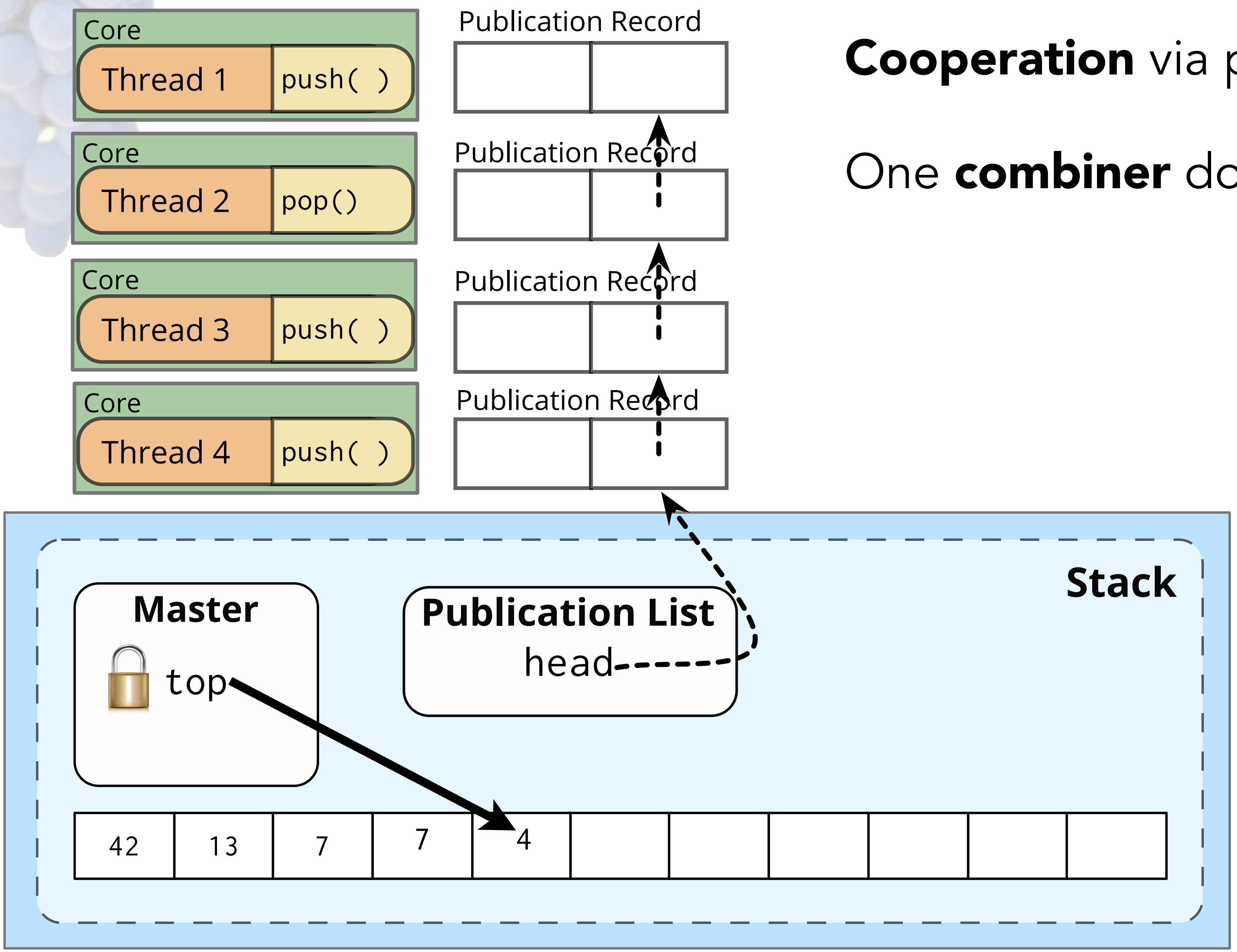


Cooperation via publication list

One **combiner** does all the work

[1] *"Flat Combining and the Synchronization-Parallelism Tradeoff"* Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA '10)

cooperation: flat combining [1]



Cooperation via publication list

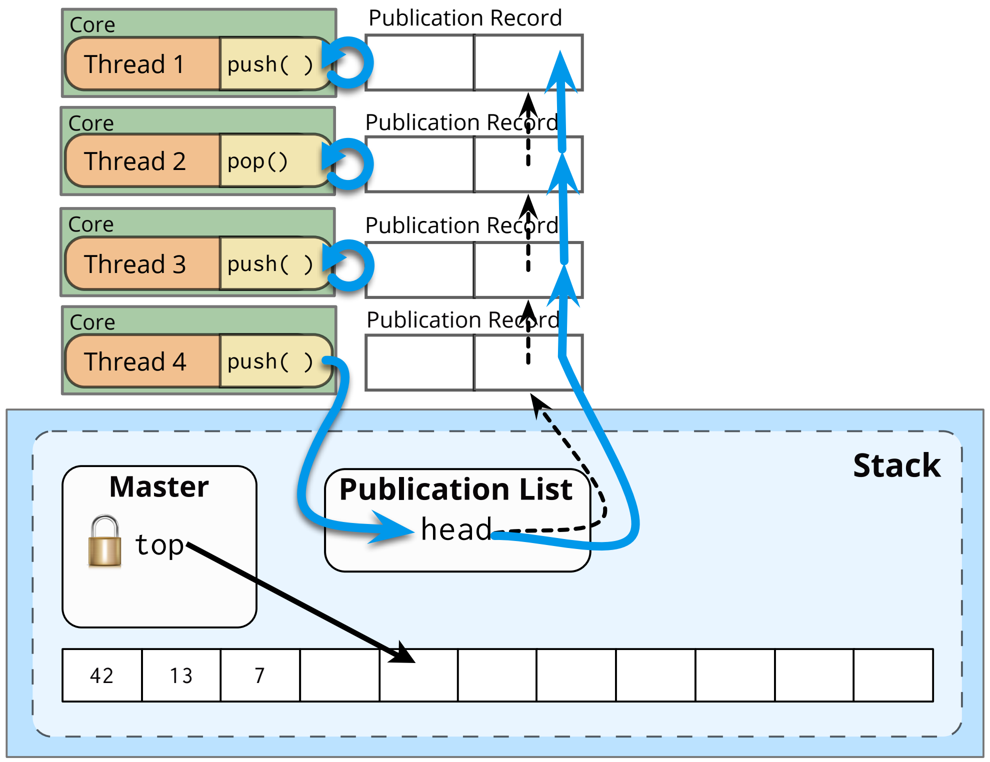
One **combiner** does all the work

[1] *"Flat Combining and the Synchronization-Parallelism Tradeoff"* Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir (SPAA '10)

Flat combining[1,2] in multicore

Simple locking scheme, but maximum of 1 failed CAS per thread

- beats combining **trees**[5] and **funnels**[3,4]
- beats fine-grained synchronization



Applicable if combined ops are faster than individually, due to:

- cache locality
- shared traversal (e.g. some linked list)
- better sequential algorithm (priority queue: pairing heap vs. skiplist)

[1] D. Handler, I. Incze, N. Shavit, M. Tzafrir. "*Flat Combining and the Synchronization-Parallelism Tradeoff*" (SPAA 2010)

[2] D. Hendler, I. Incze, N. Shavit, M. Tzafrir. "*Scalable Flat-Combining Based Synchronous Queues*" (DISC 2010)

[3] S. Kahan and P. Konecny. "*MAMA!*" (2006)

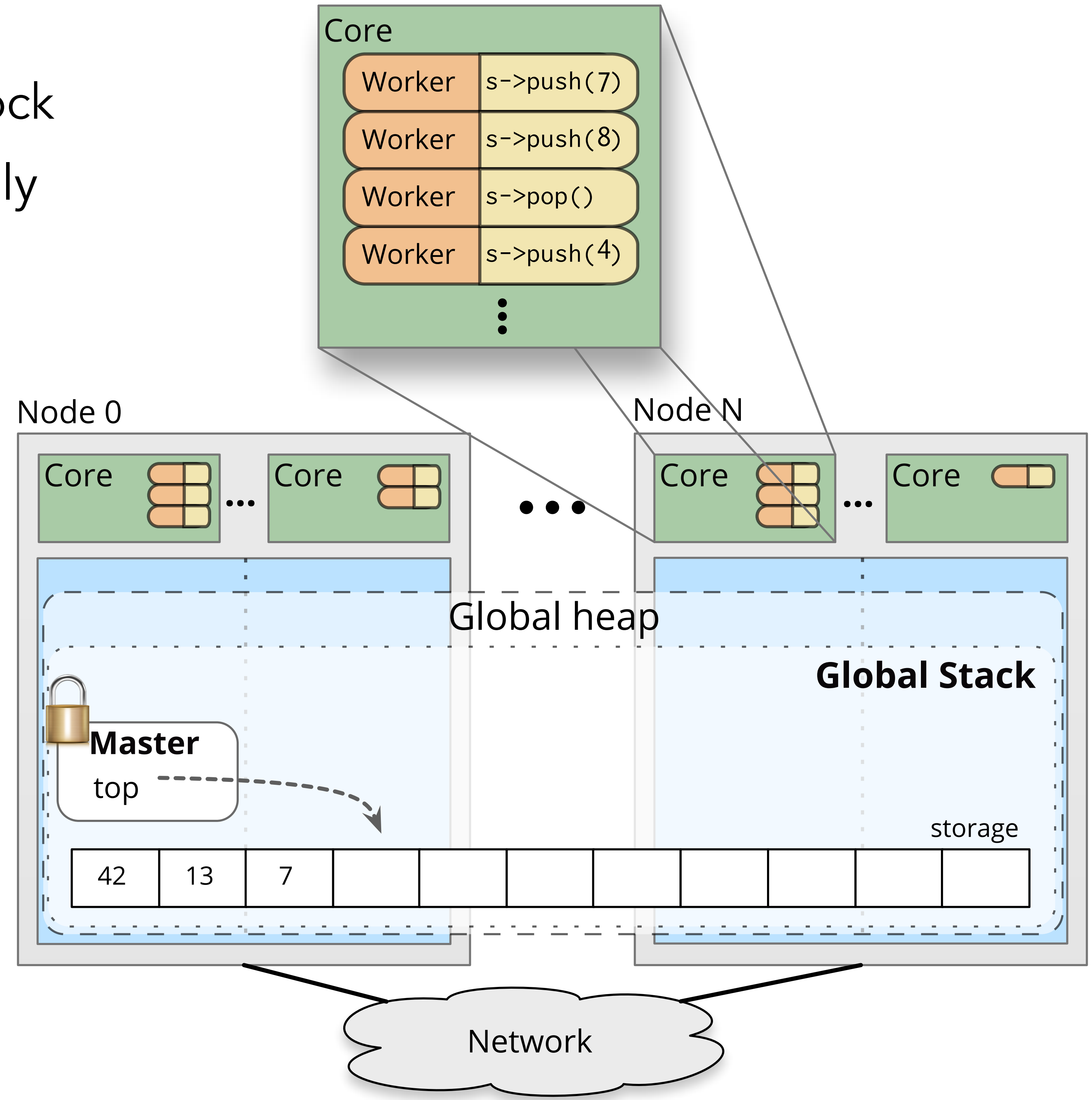
[4] N. Shavit and A. Zemach. "*Combining funnels*" (2000)

[5] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. "*Combining trees*" (1987)

Flat combining in PGAS

Distributed synchronization

- reduce serialization on global lock
- avoid making operations globally visible if possible



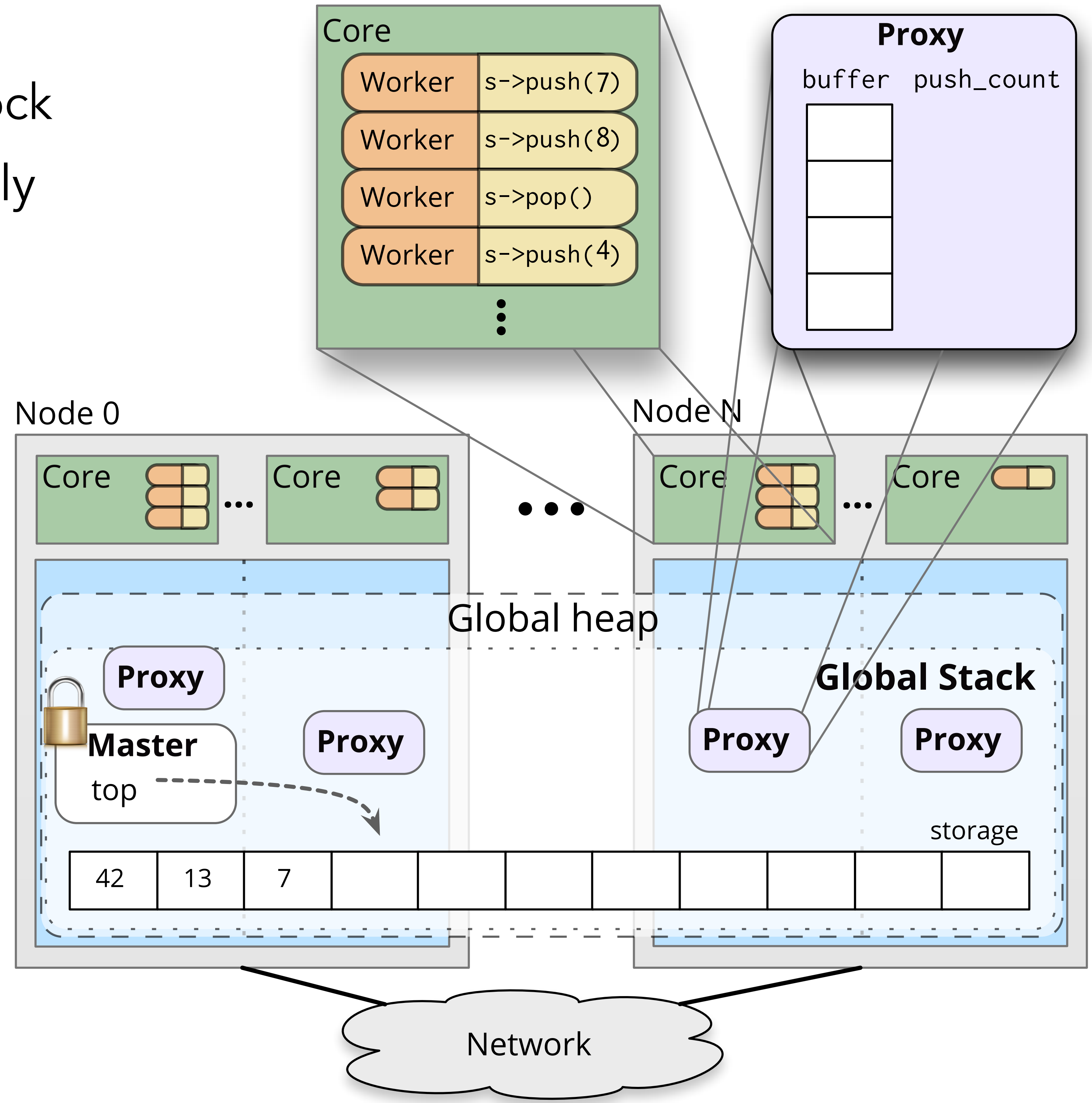
Flat combining in PGAS

Distributed synchronization

- reduce serialization on global lock
- avoid making operations globally visible if possible

Combining structure: local **proxy**

- calls operate on this instead
- resolve locally if possible



Flat combining in PGAS

Distributed synchronization

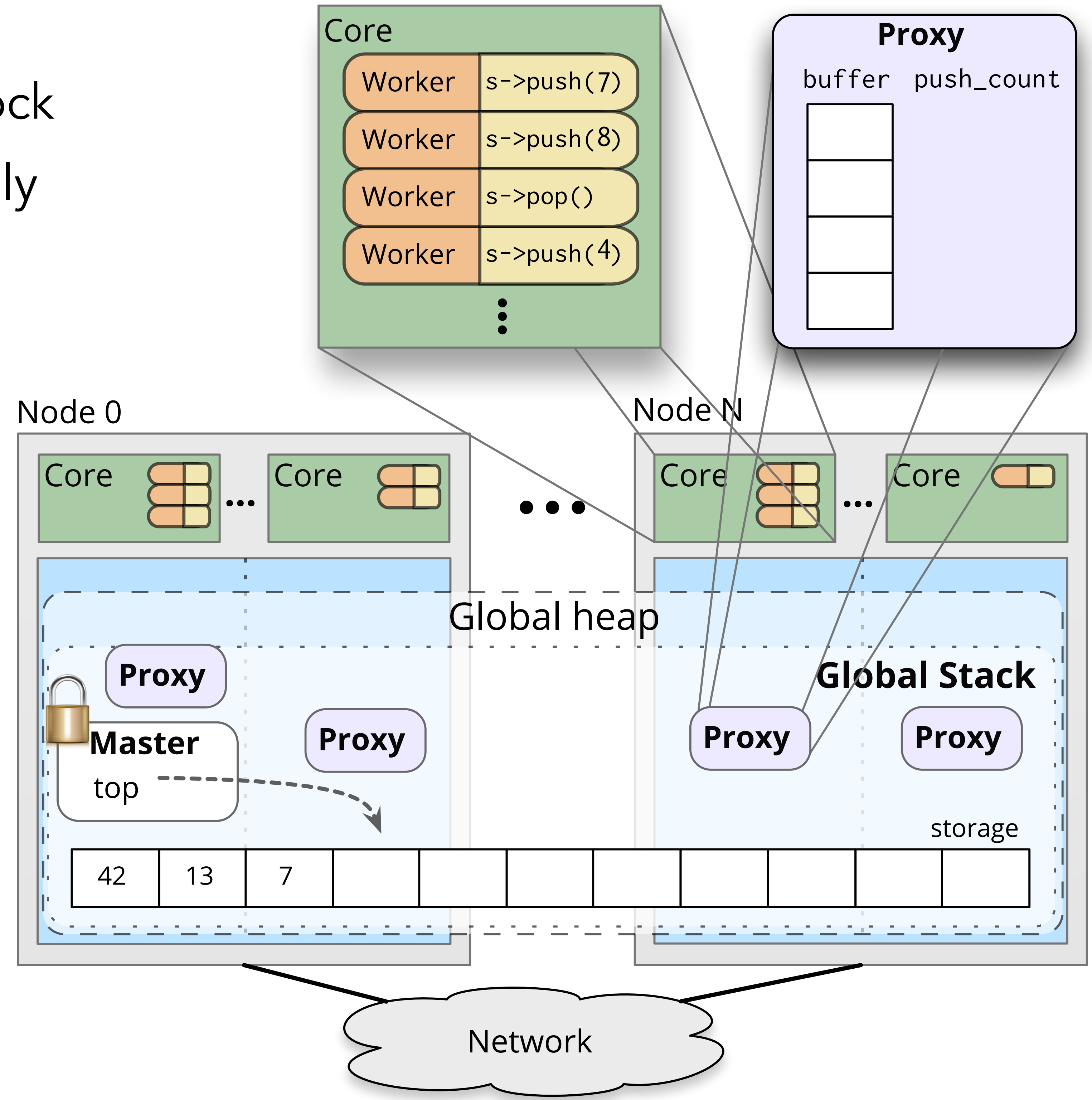
- reduce serialization on global lock
- avoid making operations globally visible if possible

Combining structure: local **proxy**

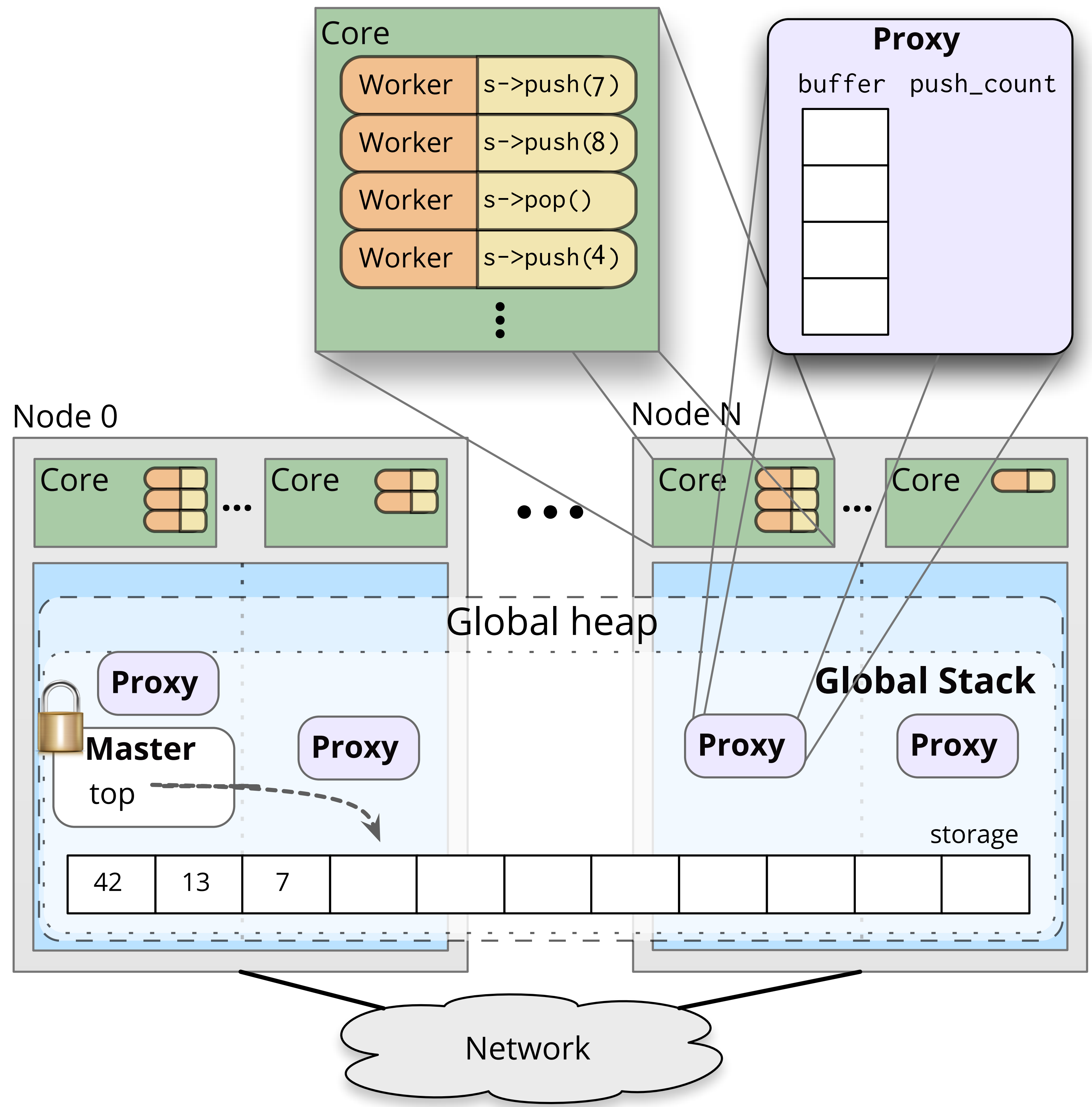
- calls operate on this instead
- resolve locally if possible

One worker commits combined op

- progress guarantee: always one in flight per core



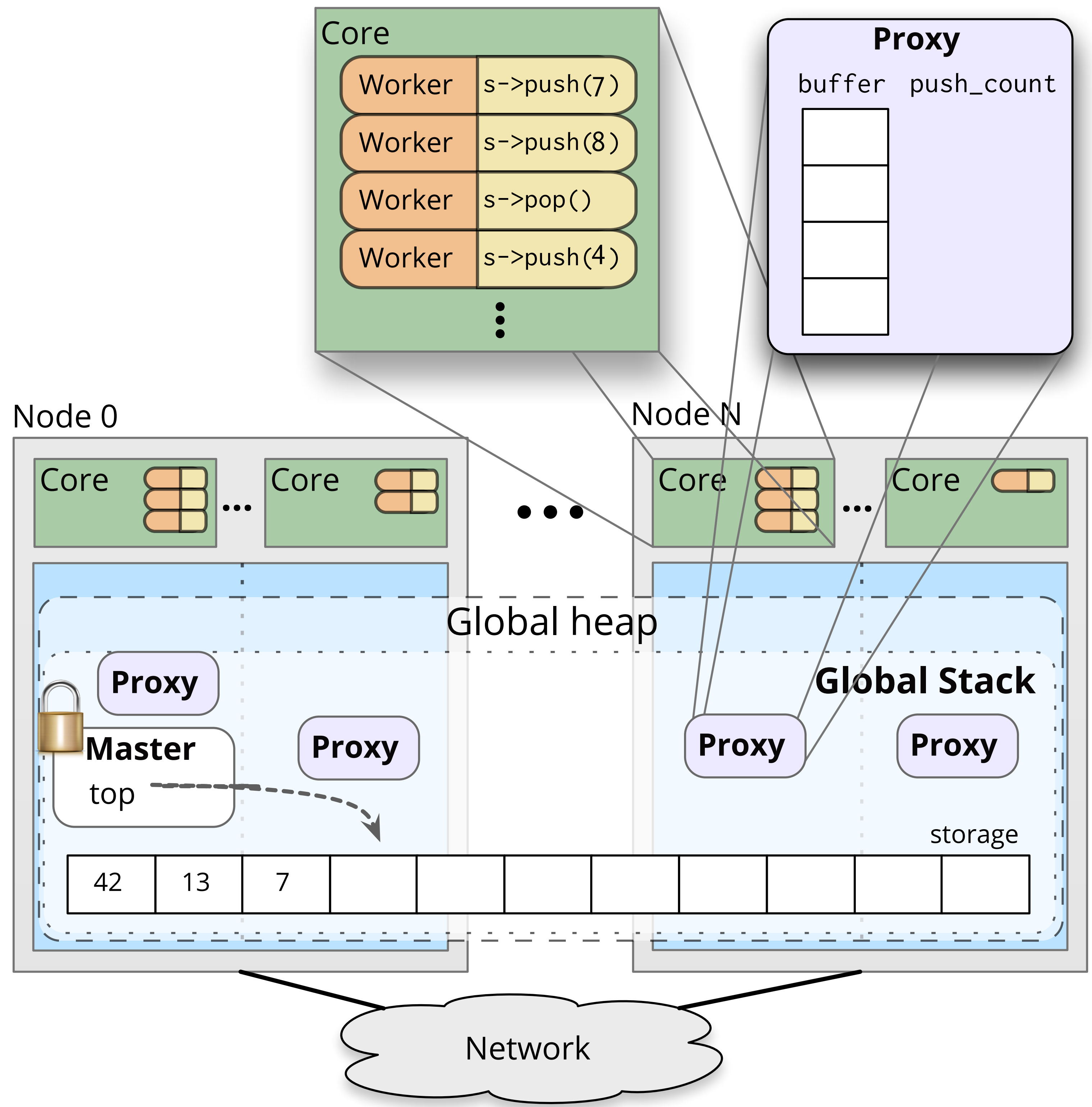
Flat combining in PGAS



Flat combining in PGAS

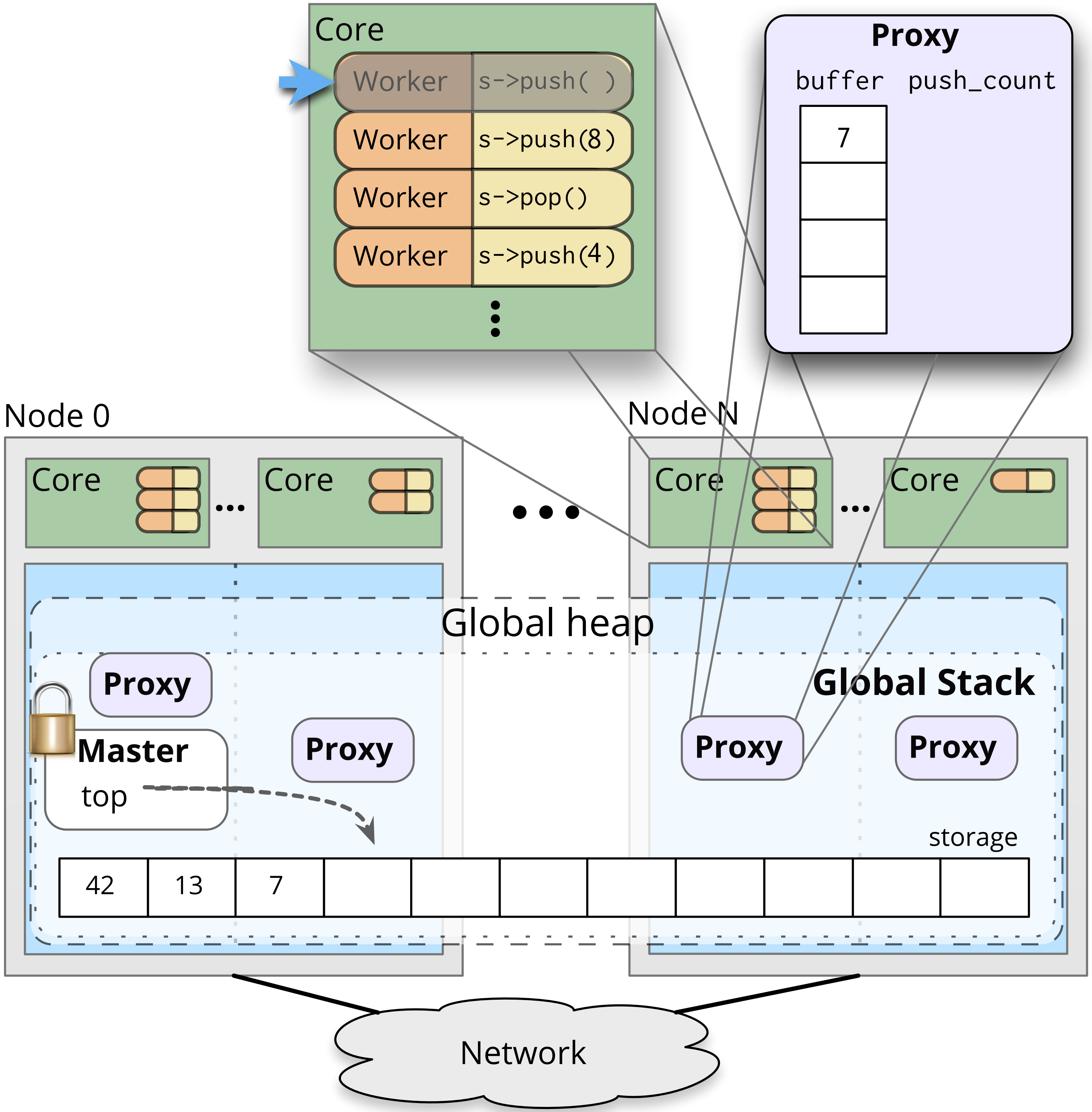
Workers operate on local proxy

- resolve locally where possible



Flat combining in PGAS

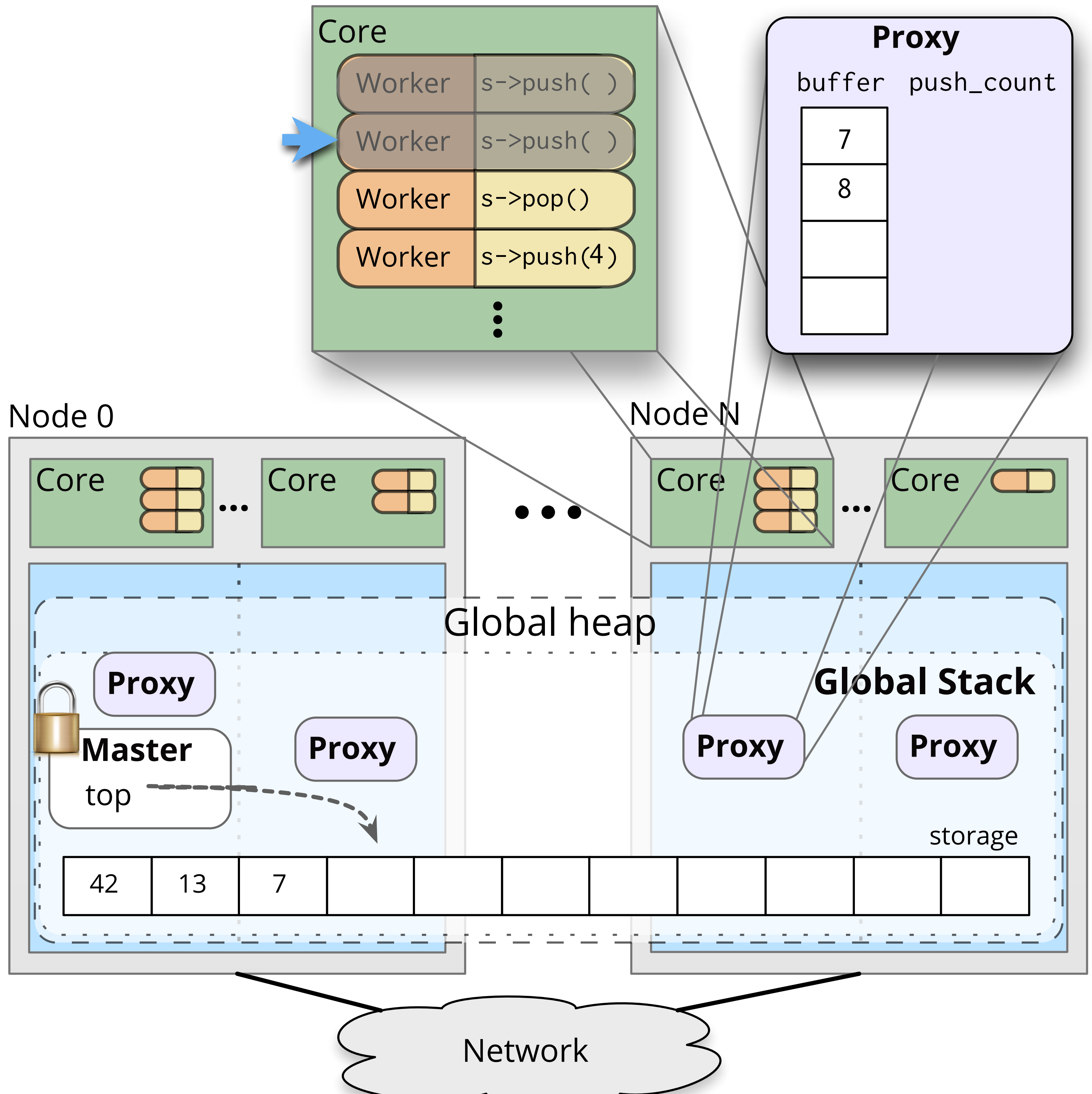
Workers operate on local proxy
– resolve locally where possible



Flat combining in PGAS

Workers operate on local proxy

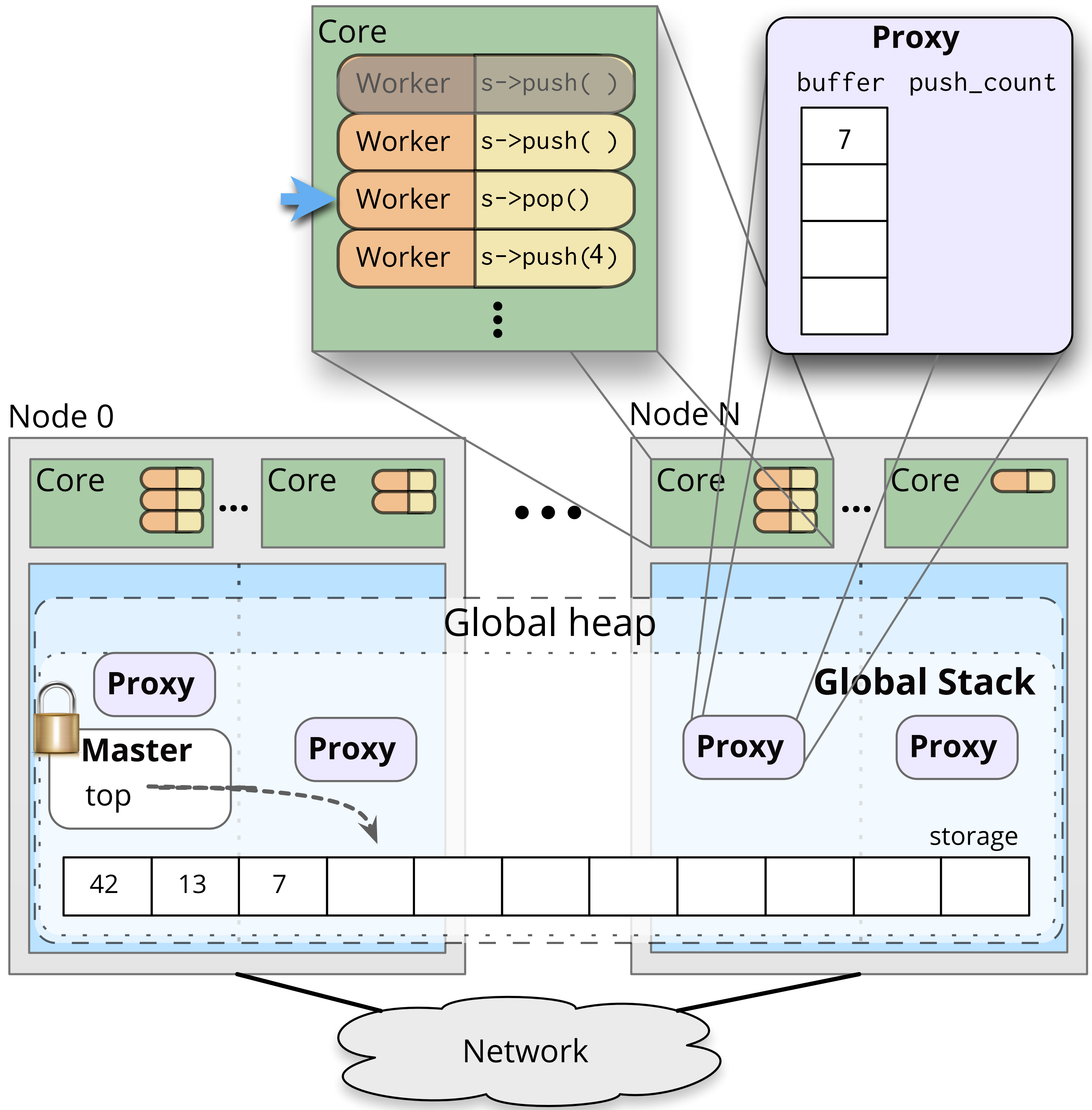
– resolve locally where possible



Flat combining in PGAS

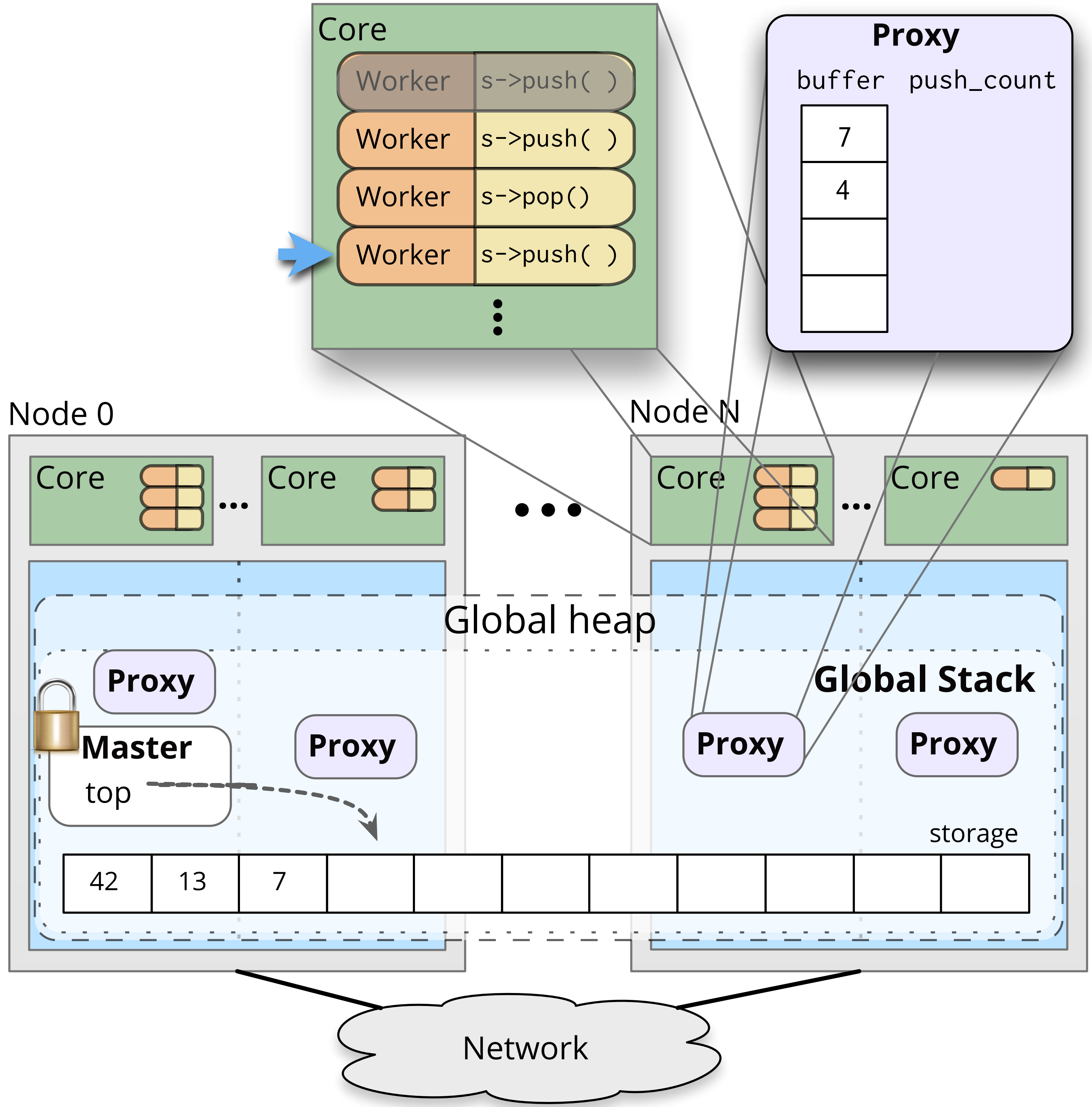
Workers operate on local proxy

– resolve locally where possible



Flat combining in PGAS

Workers operate on local proxy
- resolve locally where possible



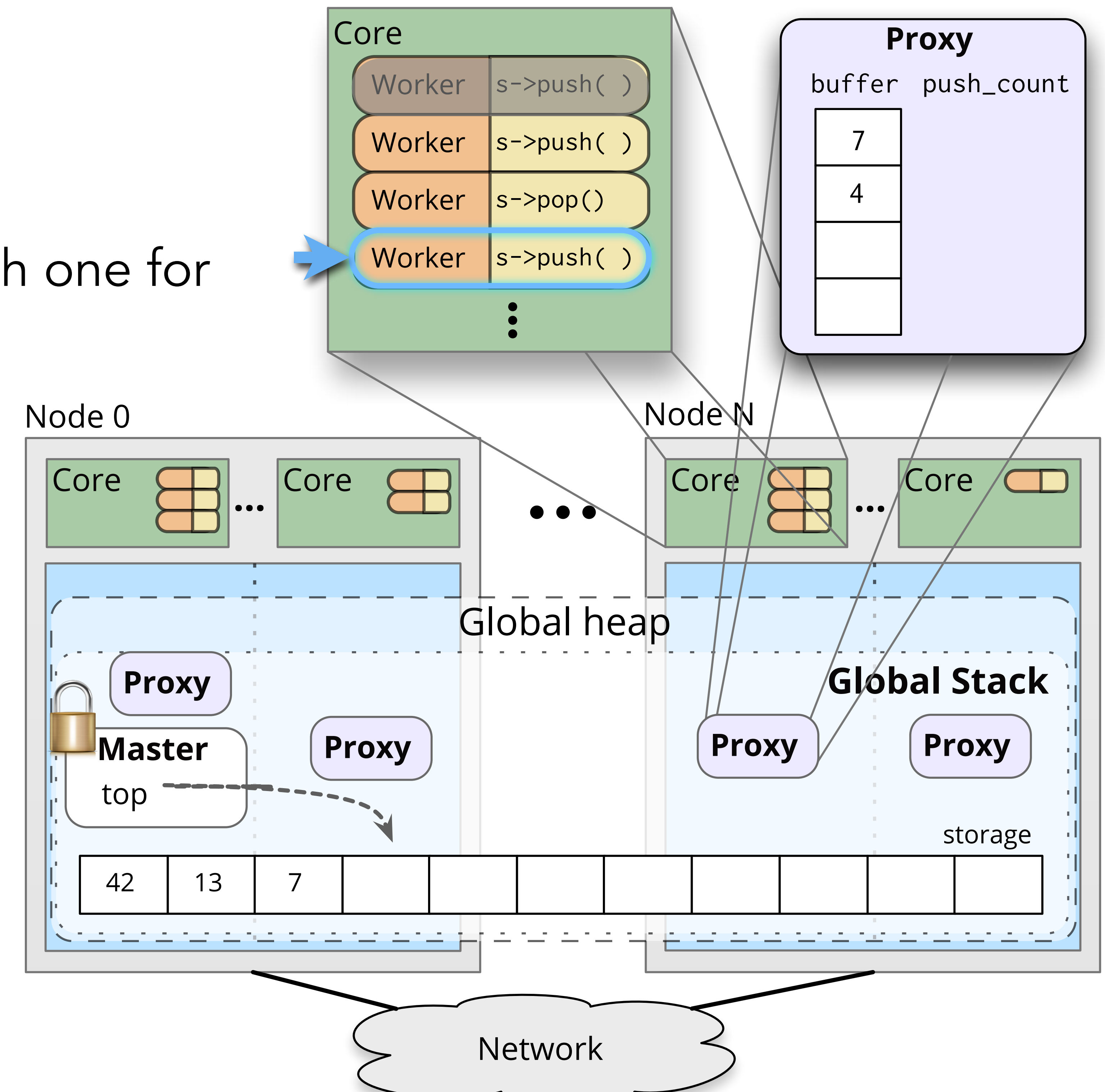
Flat combining in PGAS

Workers operate on local proxy

- resolve locally where possible

One worker becomes **combiner**:

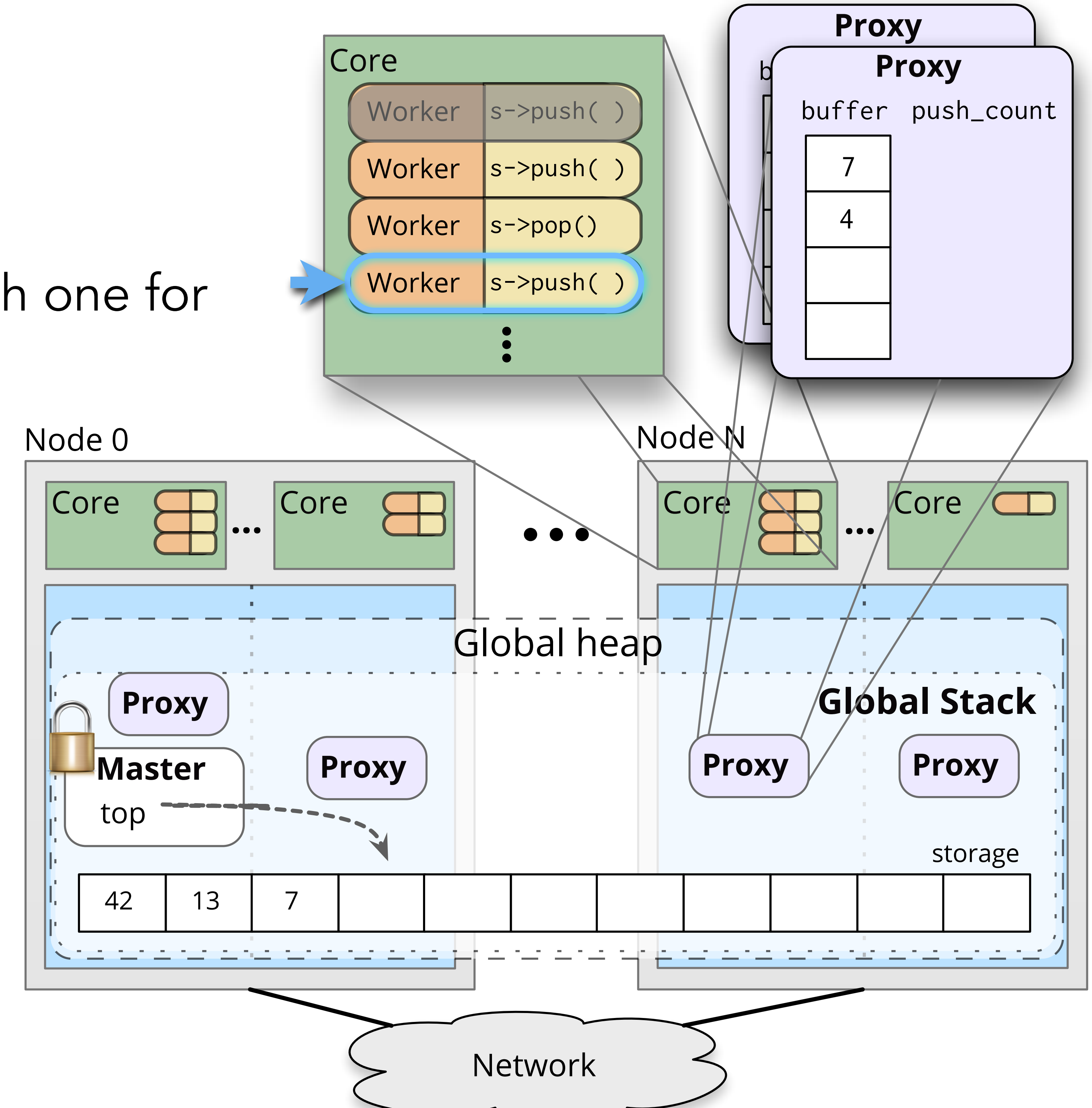
- freeze current Proxy, create fresh one for next round
- globally commit
- wake blocked workers when finished
- trigger next Proxy to go



Flat combining in PGAS

Workers operate on local proxy
- resolve locally where possible

One worker becomes **combiner**:
- freeze current Proxy, create fresh one for next round
- globally commit
- wake blocked workers when finished
- trigger next Proxy to go



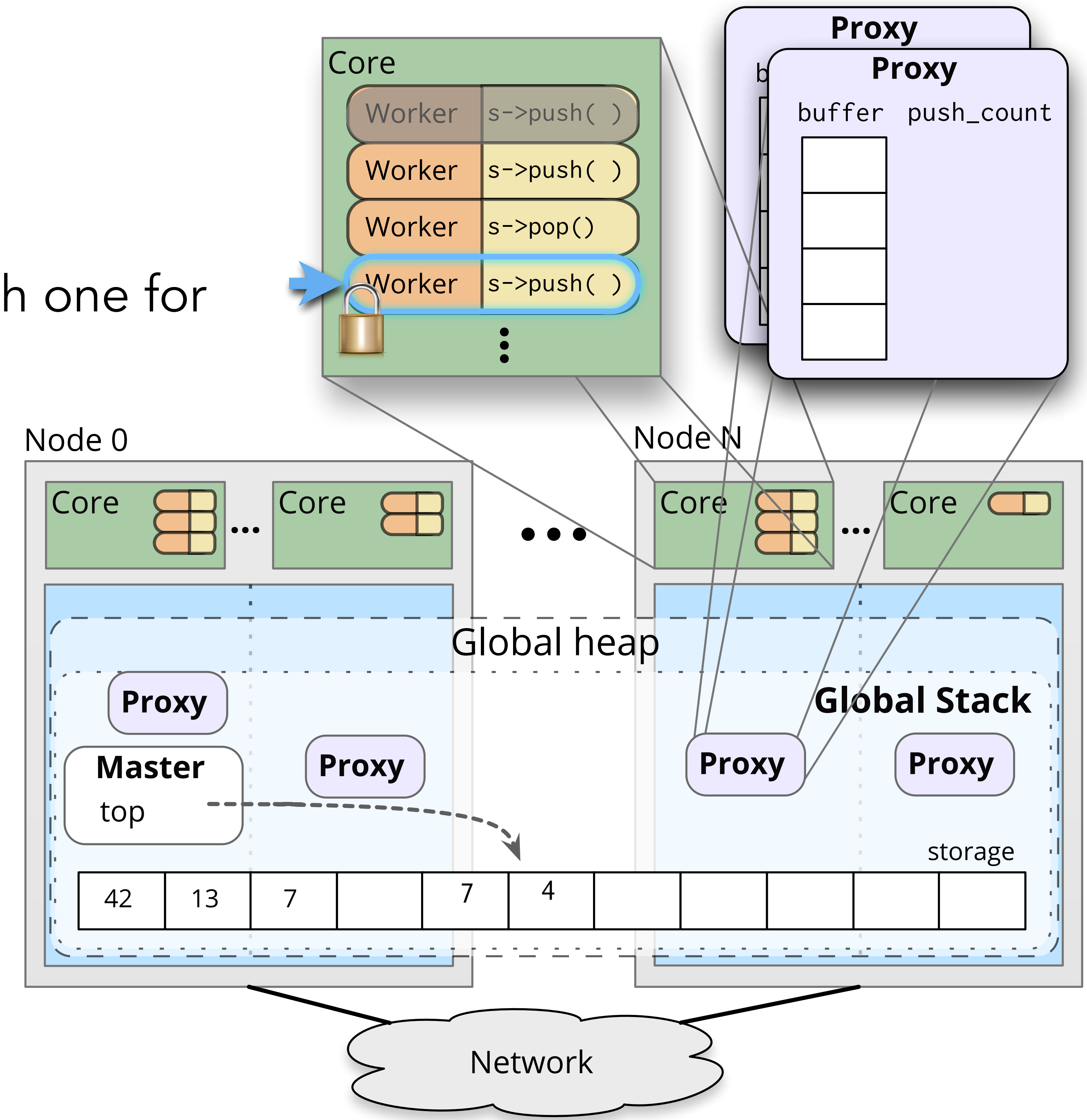
Flat combining in PGAS

Workers operate on local proxy

- resolve locally where possible

One worker becomes **combiner**:

- freeze current Proxy, create fresh one for next round
- globally commit
- wake blocked workers when finished
- trigger next Proxy to go



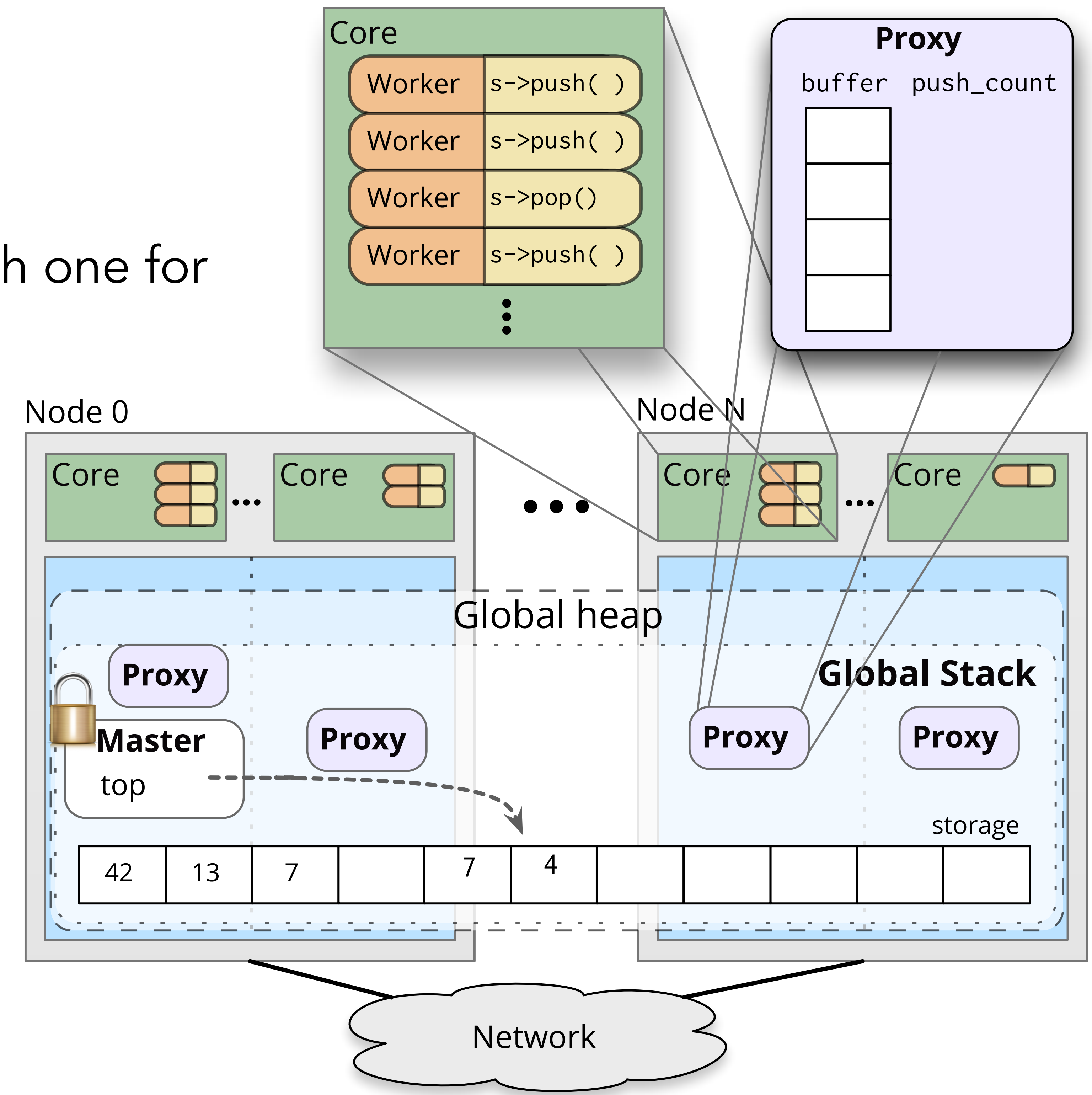
Flat combining in PGAS

Workers operate on local proxy

- resolve locally where possible

One worker becomes **combiner**:

- freeze current Proxy, create fresh one for next round
- globally commit
- wake blocked workers when finished
- trigger next Proxy to go

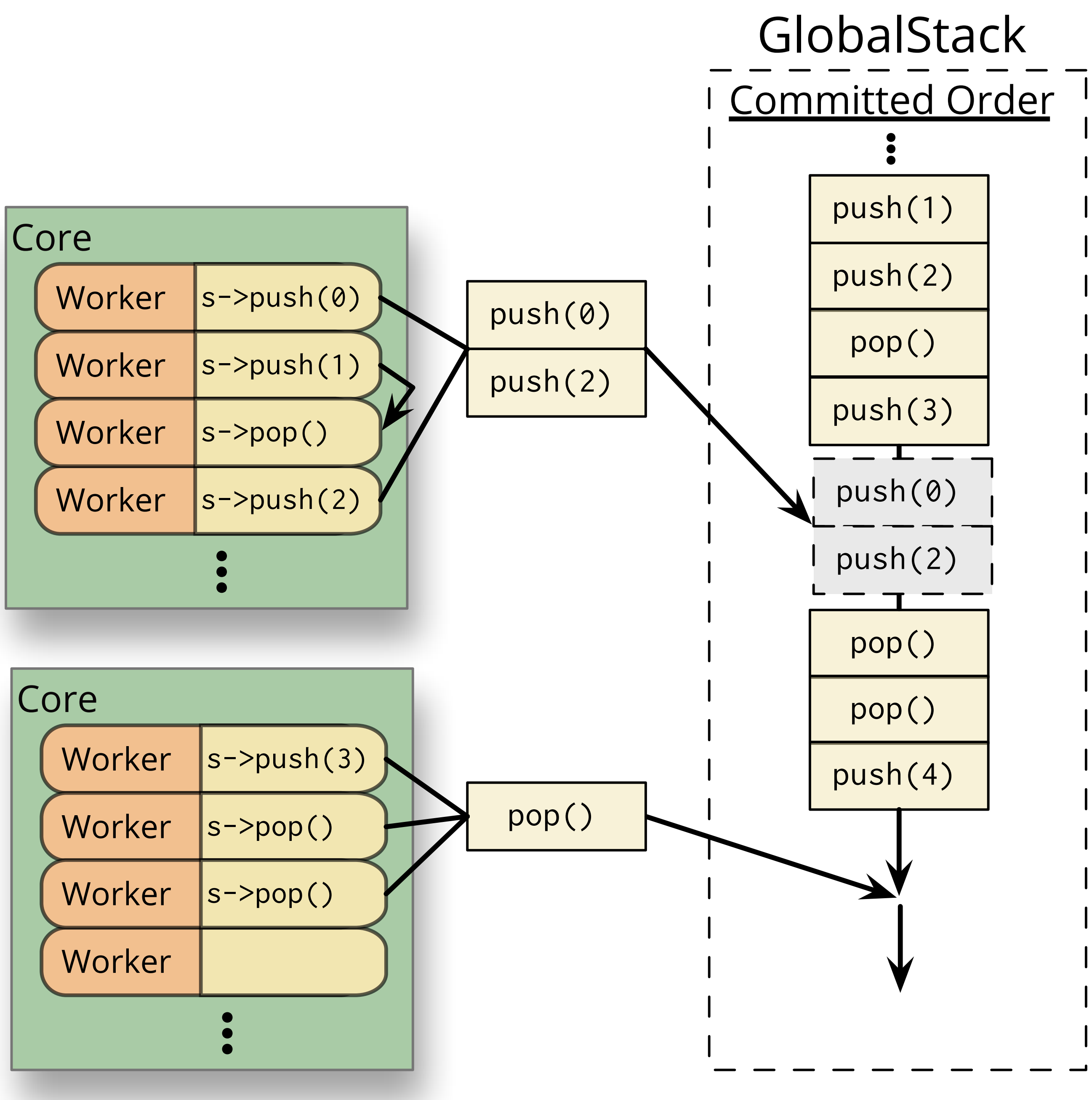


Flat combining in PGAS
Sequential Consistency

C++ model: SC for Data-Race-Free

Enforcing **linearizability:**

- ensure program order by blocking thread until globally committed
- globally- and locally-observable order must coincide

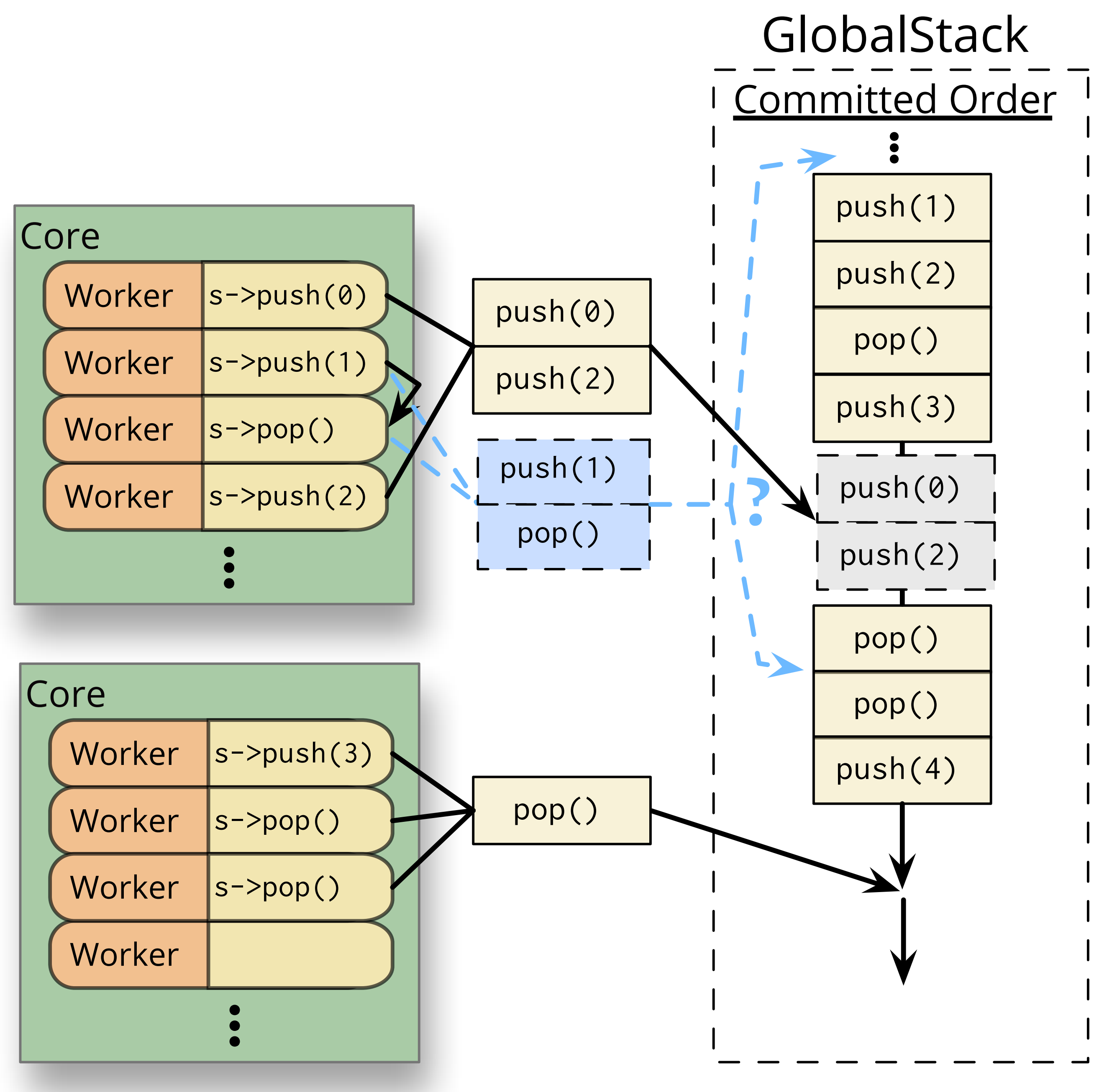


Flat combining in PGAS
Sequential Consistency

C++ model: SC for Data-Race-Free

Enforcing **linearizability:**

- ensure program order by blocking thread until globally committed
- globally- and locally-observable order must coincide



GlobalStack

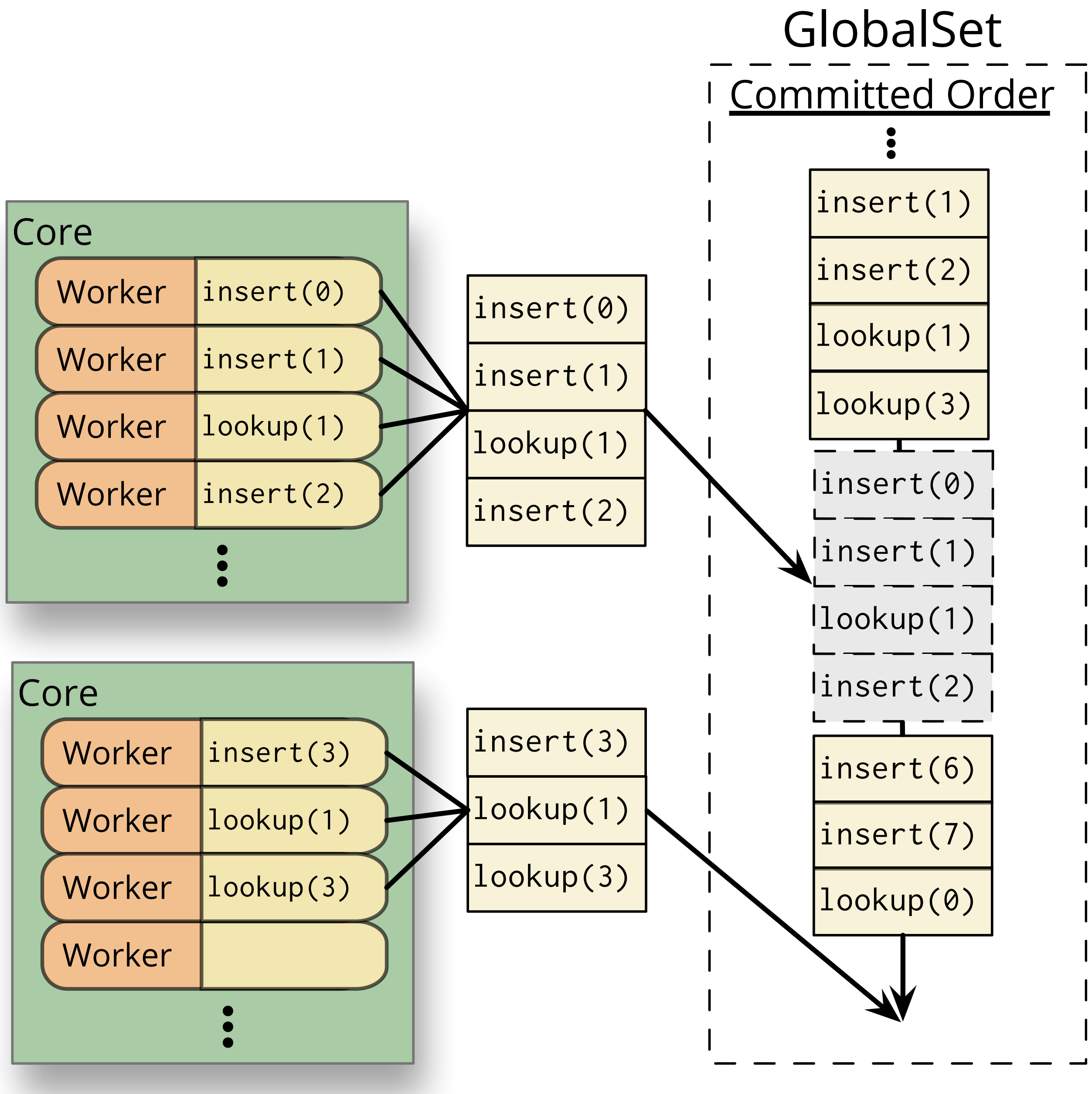
push/pop *annihilate* each other, can be anywhere in global order

Flat combining in PGAS
Sequential Consistency

C++ model: SC for Data-Race-Free

Enforcing **linearizability:**

- ensure program order by blocking thread until globally committed
- globally- and locally-observable order must coincide



Flat combining in PGAS

Sequential Consistency

C++ model: SC for Data-Race-Free

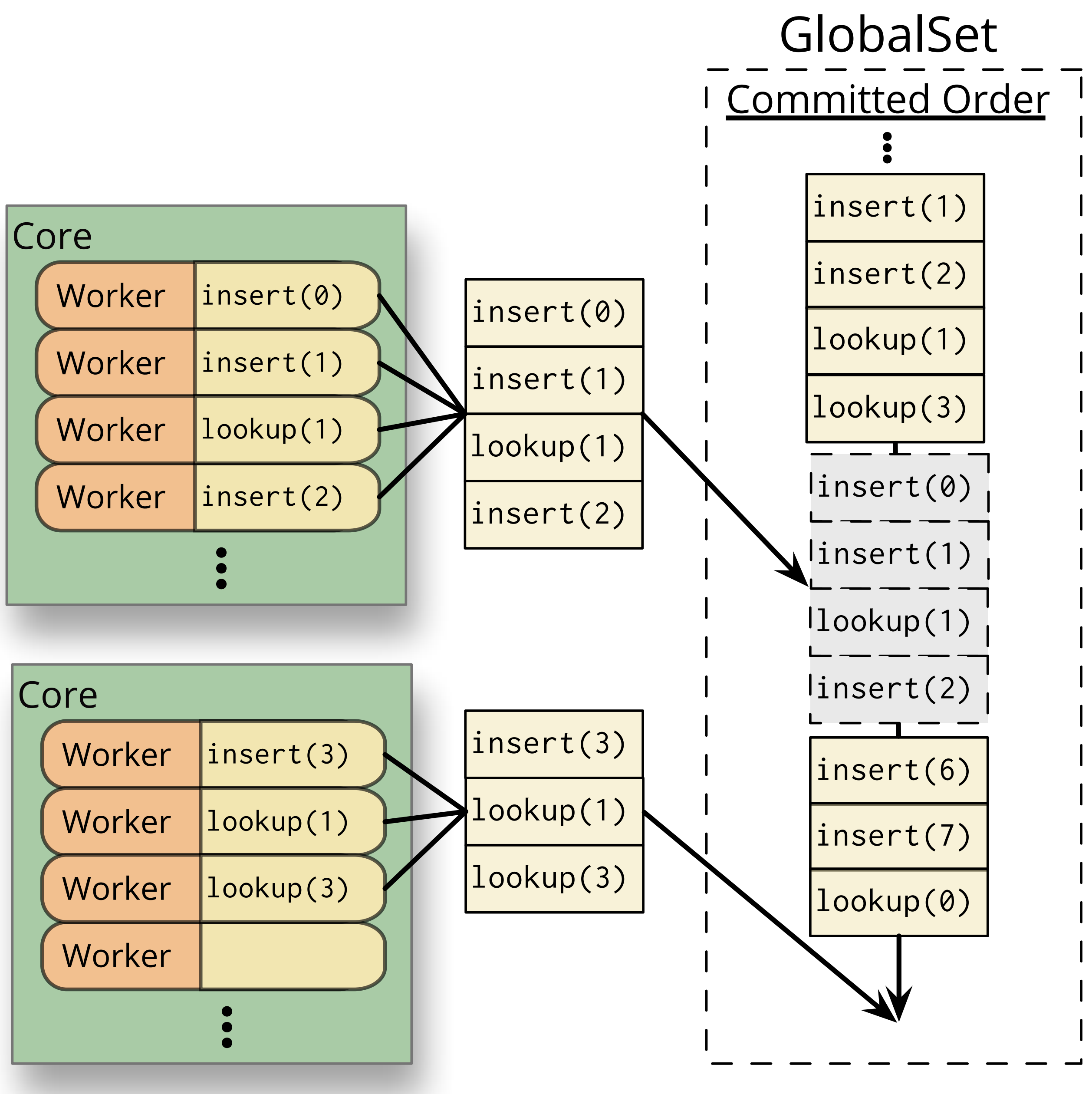
Enforcing **linearizability:**

- ensure program order by blocking thread until globally committed
- globally- and locally-observable order must coincide

GlobalSet/GlobalMap

– insert/lookup must preserve order

– cheaper to disallow local lookups



Flat combining in PGAS

Sequential Consistency

C++ model: SC for Data-Race-Free

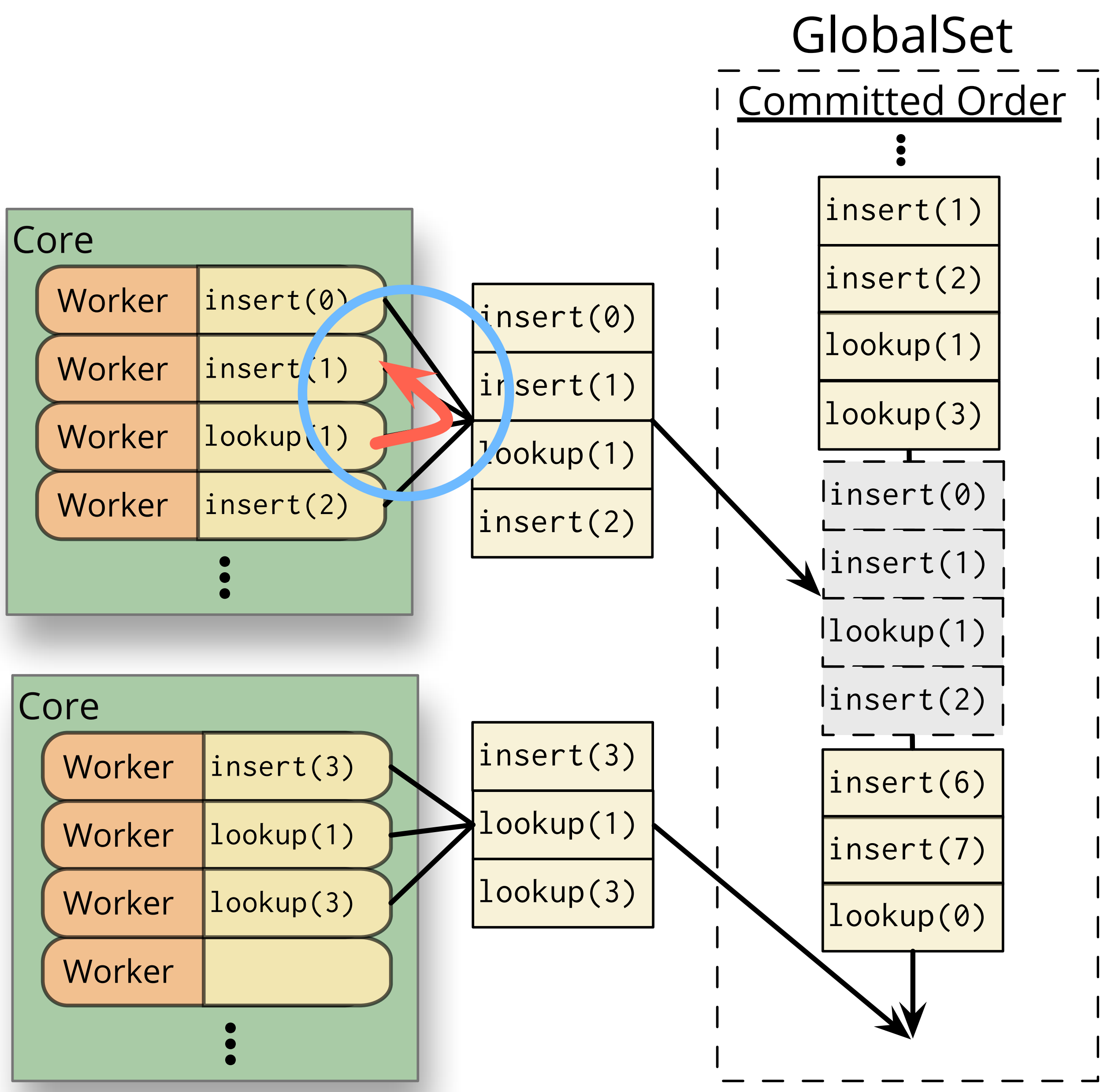
Enforcing **linearizability:**

- ensure program order by blocking thread until globally committed
- globally- and locally-observable order must coincide

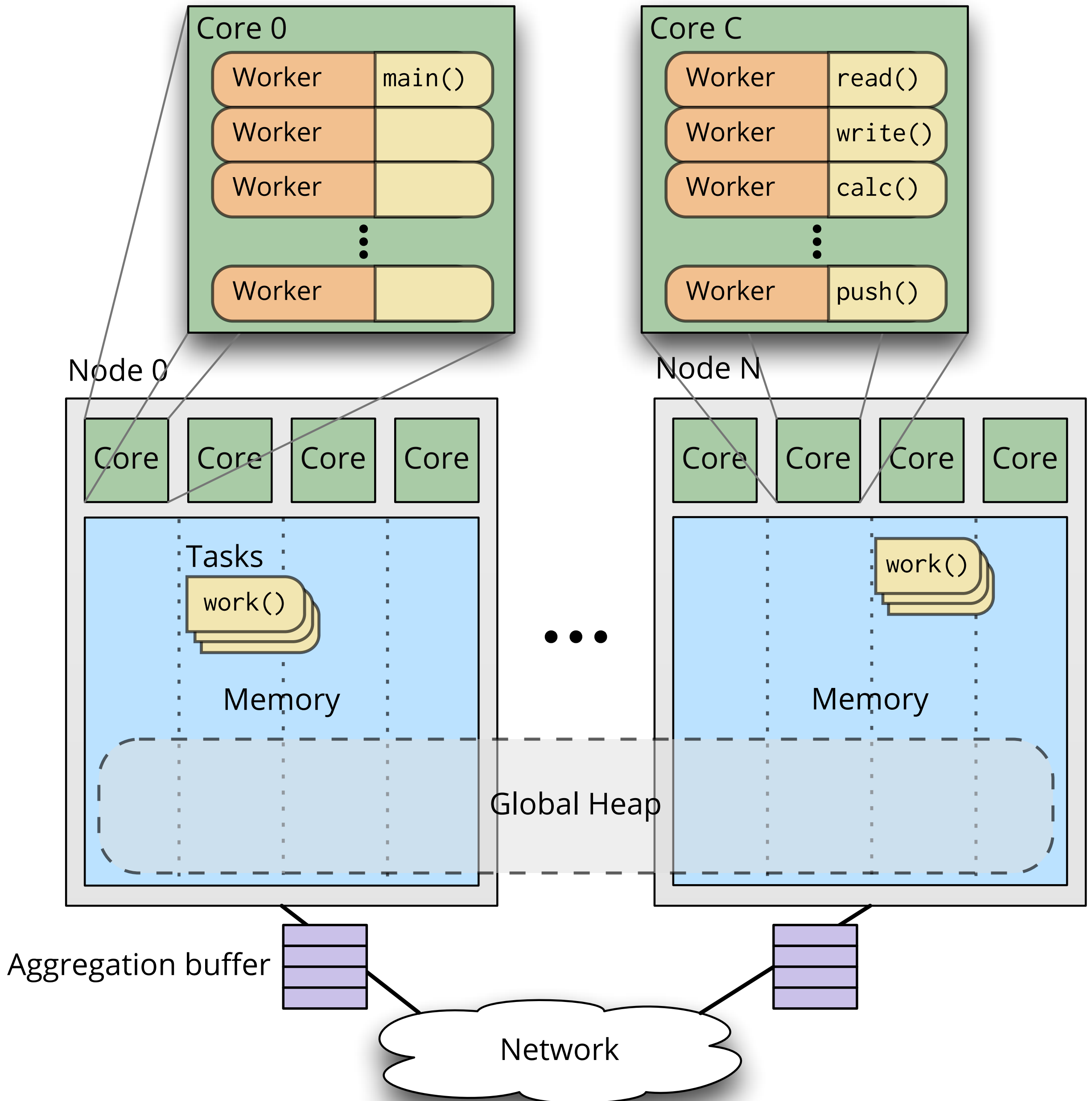
GlobalSet/GlobalMap

– insert/lookup must preserve order

– cheaper to disallow local lookups



Flat combining in Grappa



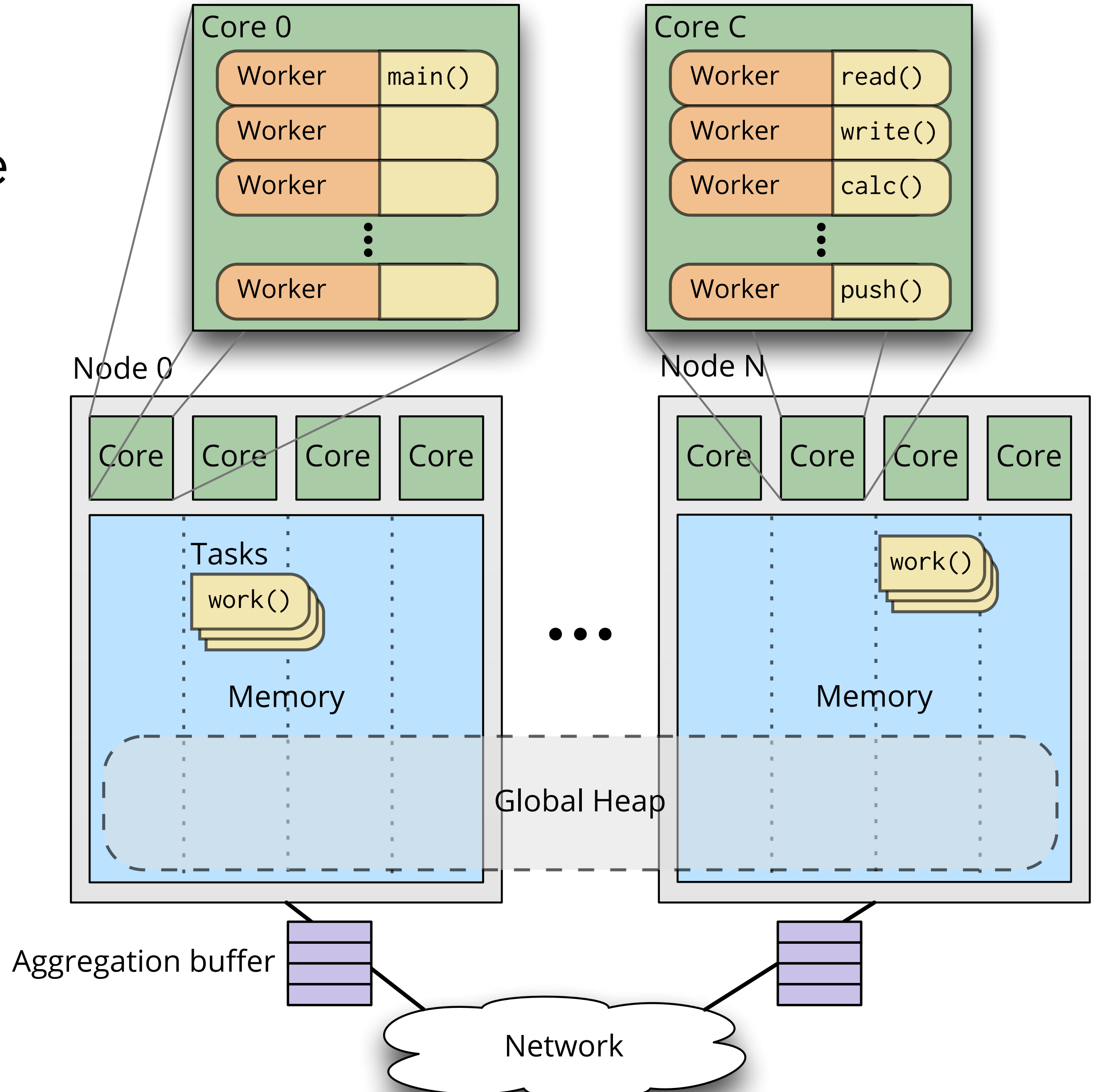
Flat combining in Grappa

Massive multithreading

- many workers, lots of combining
- lightweight suspend/wake

Synchronizing with Proxy is free

- **cooperative multithreading** within core
- only access other cores' memory via **delegate ops**



Flat combining
performance evaluation

Experimental setup

– Run on the PIC cluster at Pacific Northwest National Lab (PNNL)
– AMD Interlagos 2.1 GHz,
  40 Gb Infiniband (Mellanox Connect-X 2, with QLogic switch)
– 16 cores per node,
  2048 workers per core

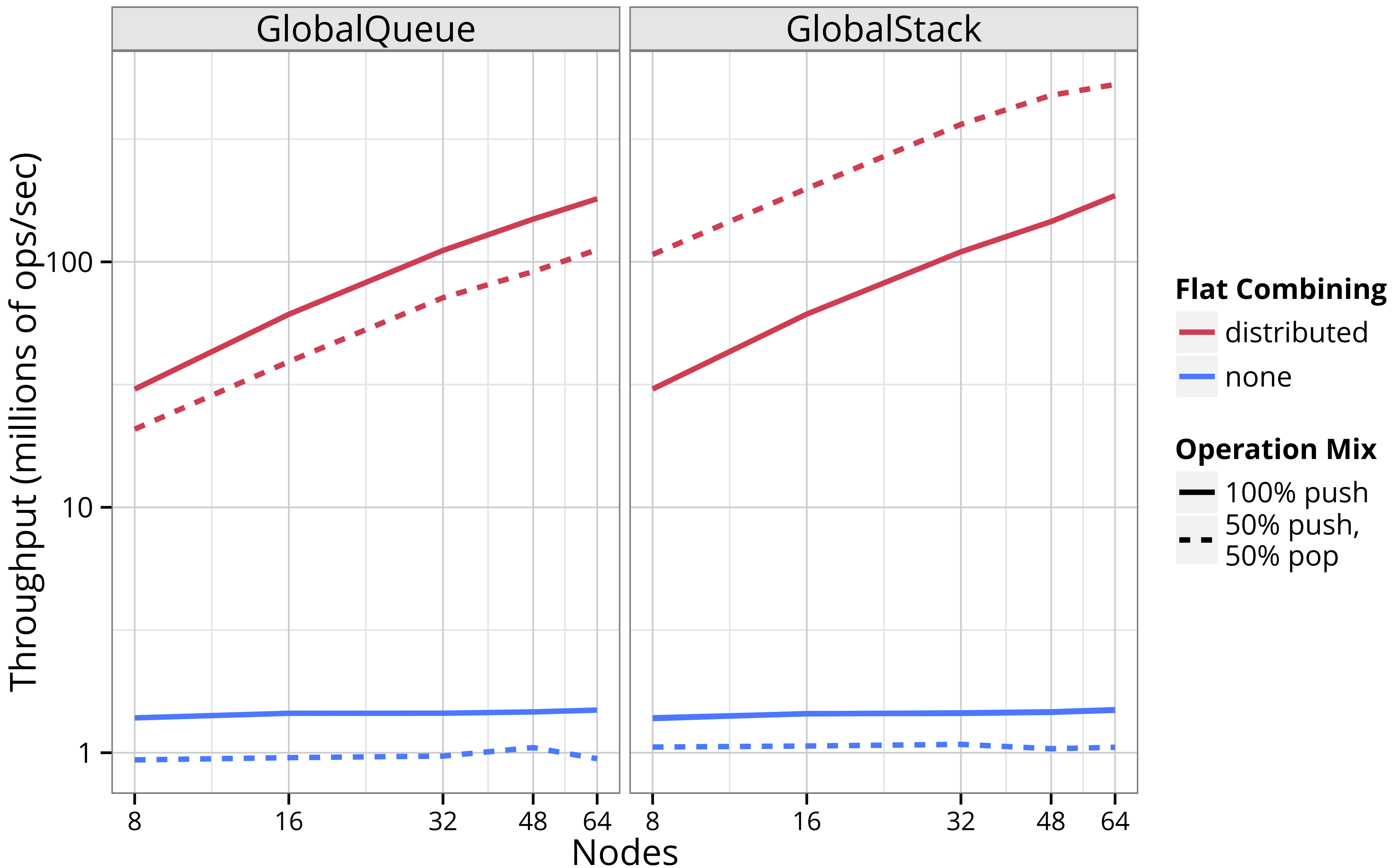
```
void test(GlobalAddress<GlobalStack<long>> stack)
{
  forall_global(0, 1<<28, [=](long i){
    if (choose_random(push_mix)) {
      stack->push(next_random<long>());
    } else {
      stack->pop();
    }
  });
}
```

Methodology

Random throughput workload
– With/without flat combining
– Varied operation mix
  (push/pop, lookup/insert)

Flat combining
performance evaluation

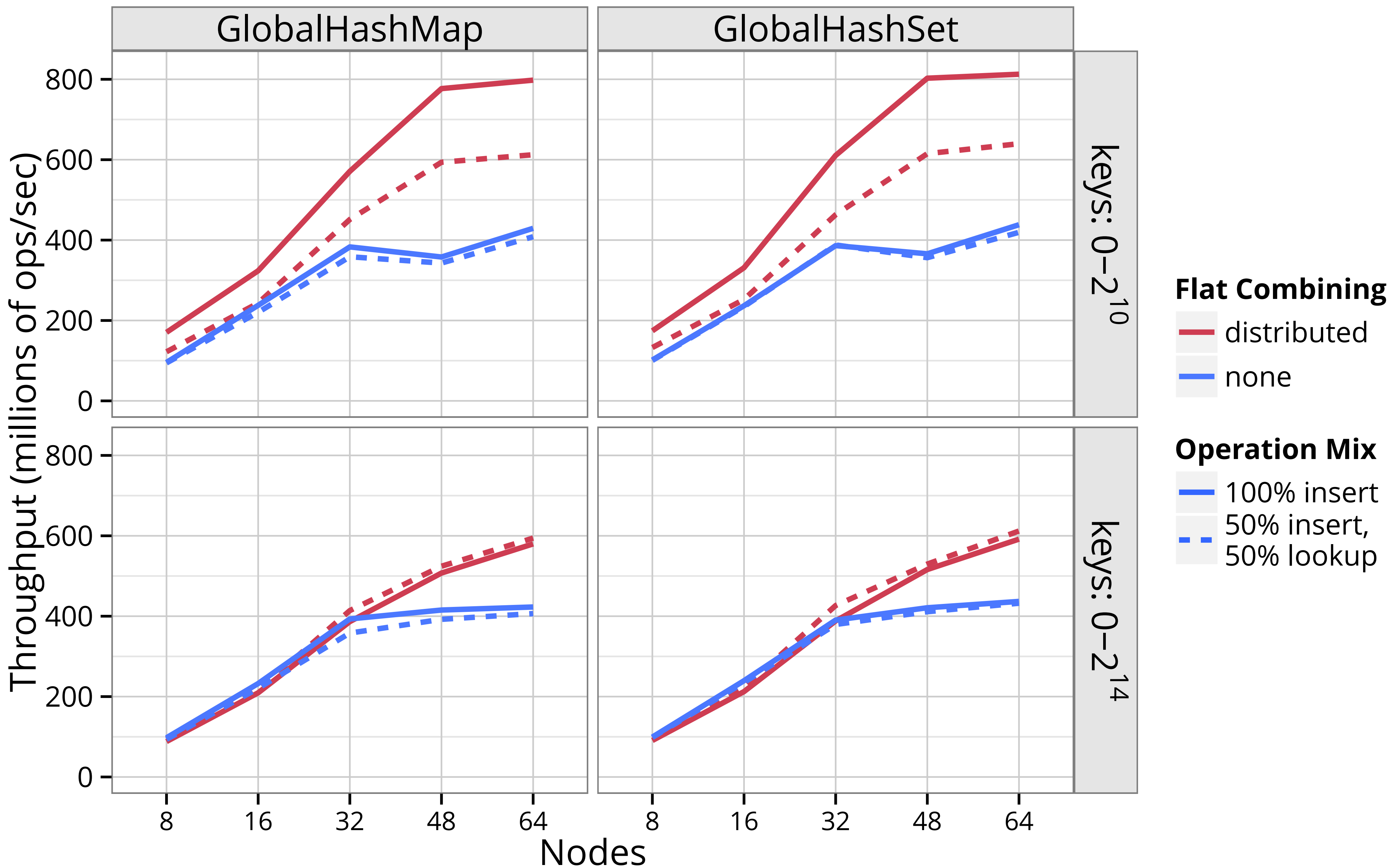
Flat combining
performance evaluation



Flat combining

performance evaluation

Flat combining
performance evaluation



Flat combining

performance evaluation

Experimental setup

– Run on the PIC cluster at Pacific Northwest National Lab (PNNL)

– AMD Interlagos 2.1 GHz,
40 Gb Infiniband (Mellanox Connect-X 2, with QLogic switch)

– 16 cores per node,
2048 workers per core

Application Kernels

– Scale 26 Graph500-spec graph
(64 M vertices, 1 B edges)

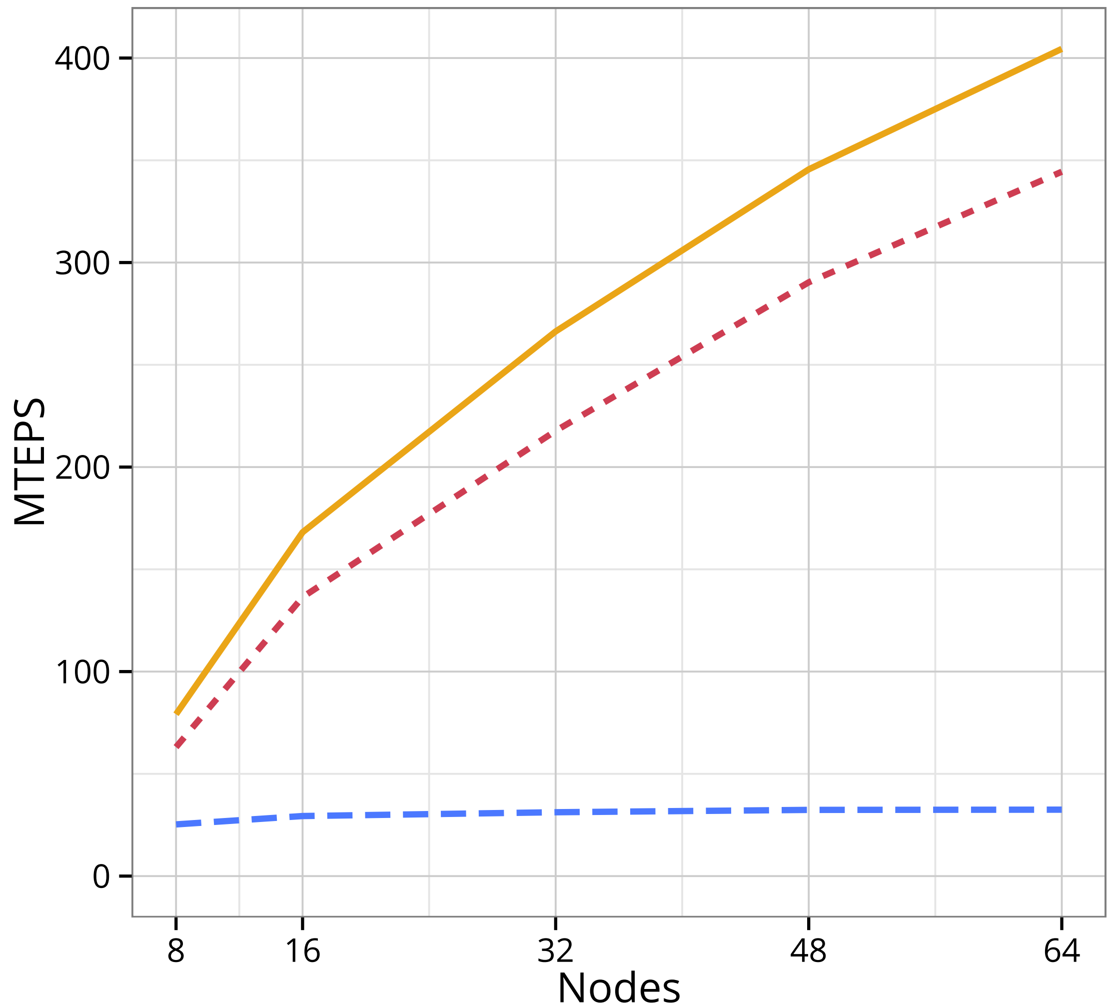
– **Breadth First Search** benchmark
(find parent tree from random root)

– **Connected Components**
(using 3-phase algorithm)

Flat combining

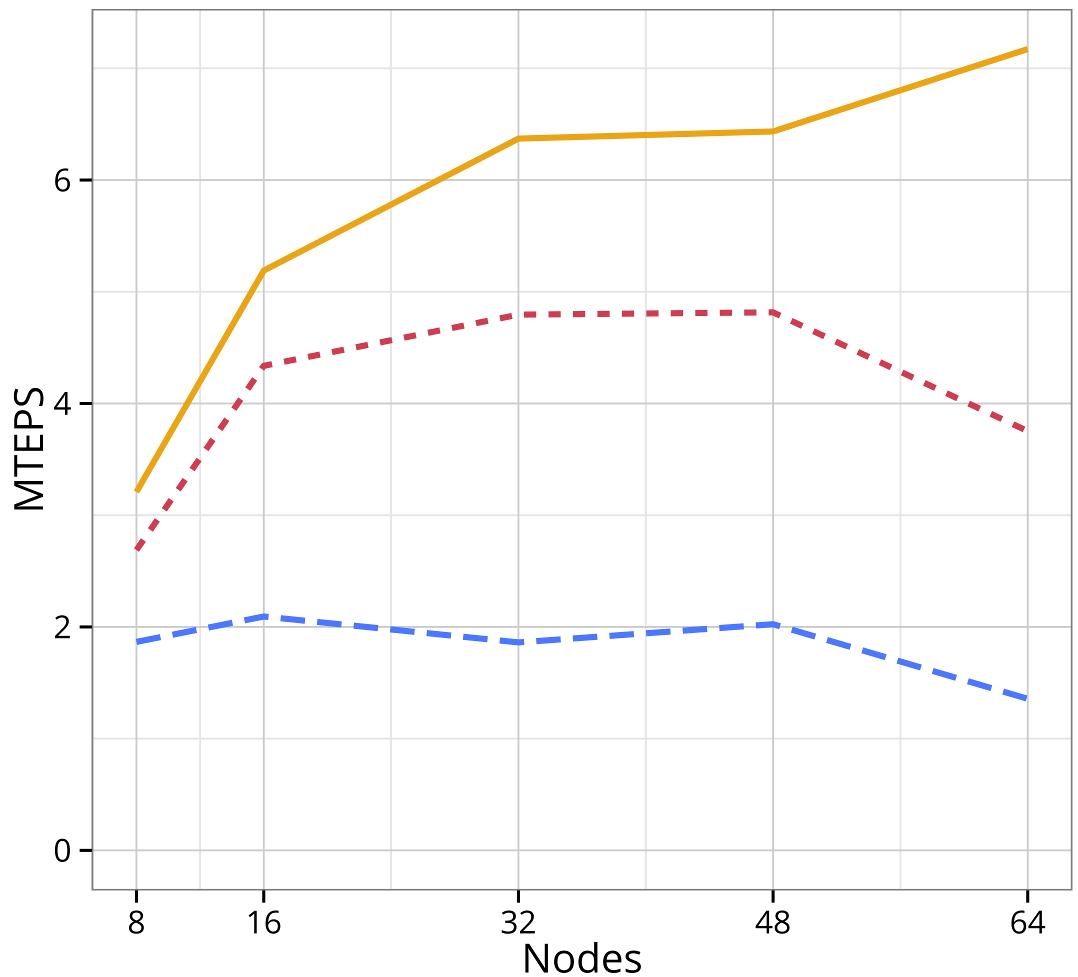
performance evaluation

Breadth First Search

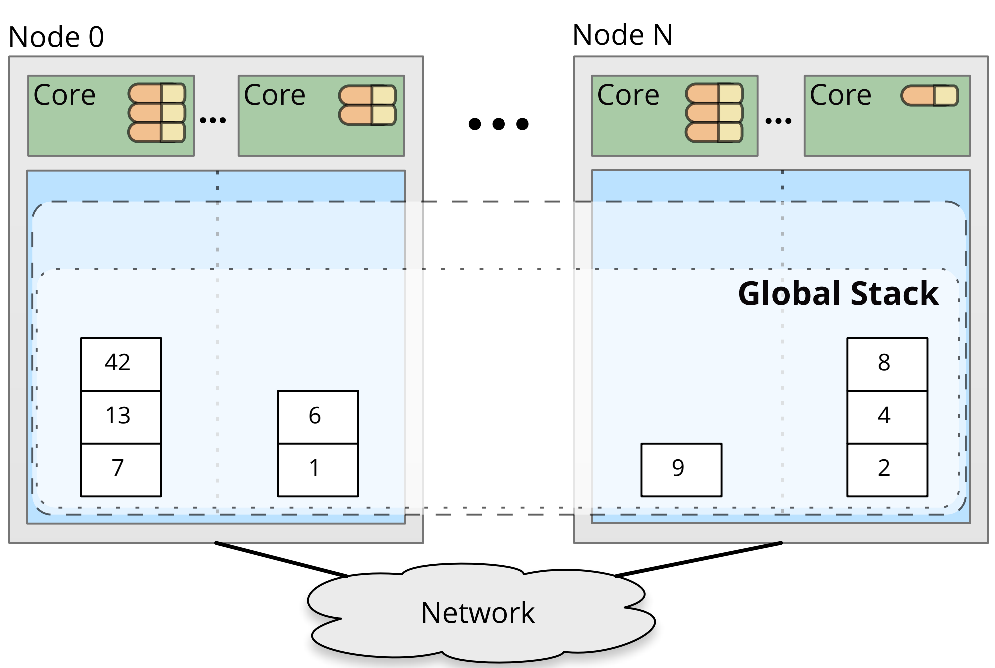


Flat Combining
- custom
- distributed
- none

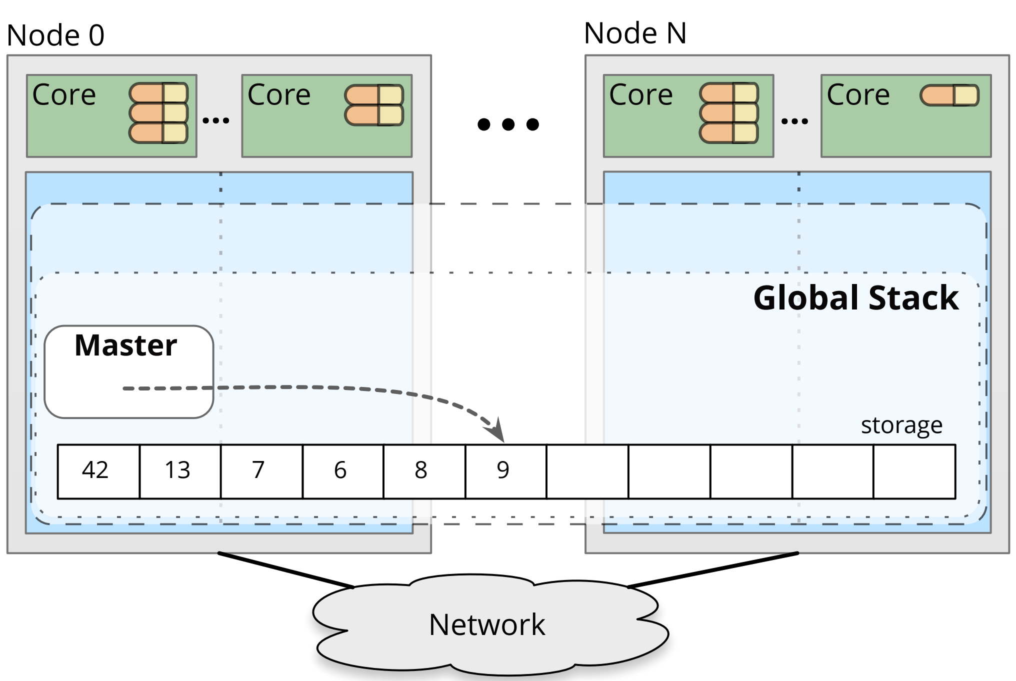
Connected Components



Future directions:
"Schrödinger" consistency



?



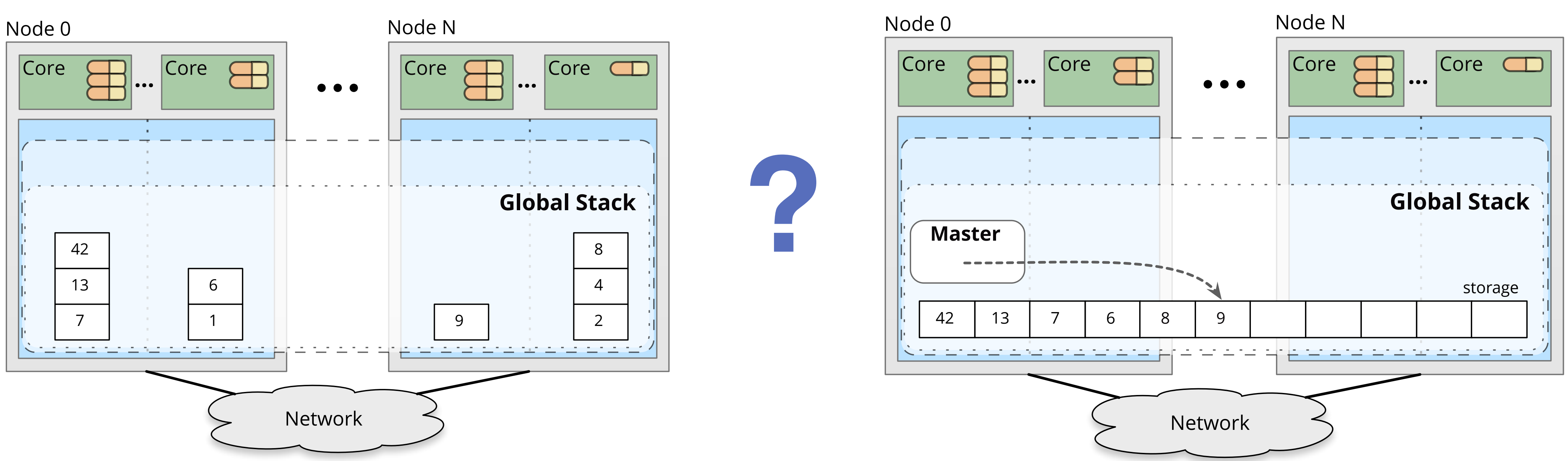
Future directions:

"Schrödinger" consistency

Hiding even more behind high-level data structure abstraction

Delay synchronization as long as possible

- commit when operation would be able to **observe** order
- example: *pushes* kept local, *pops* search for an available *push*



Future directions:
abstract data structure semantics

Future directions:
abstract data structure semantics

"Transactional Boosting"

- abstract semantics to determine conflicts
- express how operations affect and observe abstract state
- **abstract locks** determine what can happen concurrently
- **inverse operations** for rolling back aborted transaction

Applying to Grappa and distributed memory

- commutative ops proceed locally in parallel
- inverse ops annihilate without external synchronization
- tasks with conflicting operations delayed; when out of tasks with commutative ops, then commit and allow others to proceed

Synthesize abstract lock conditions from annotations

Maurice Herlihy & Eric Koskinen. PPoPP 2008.
Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects.

Jacob Nelson

© Disney, Inc. *Fantasia (The Pastoral Symphony)*

Simon Kahan

Brandon Myers

Luis Ceze

Mark Oskin

Preston Briggs

Thank you!

© Disney, Inc. *Fantasia (The Pastoral Symphony)*

Jacob Nelson

Simon Kahan

Brandon Myers

Luis Ceze

Mark Oskin

Preston Briggs