Concurrent programming

Shared Counters and Parallelism

Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear

Modified by Piotr Witkowski

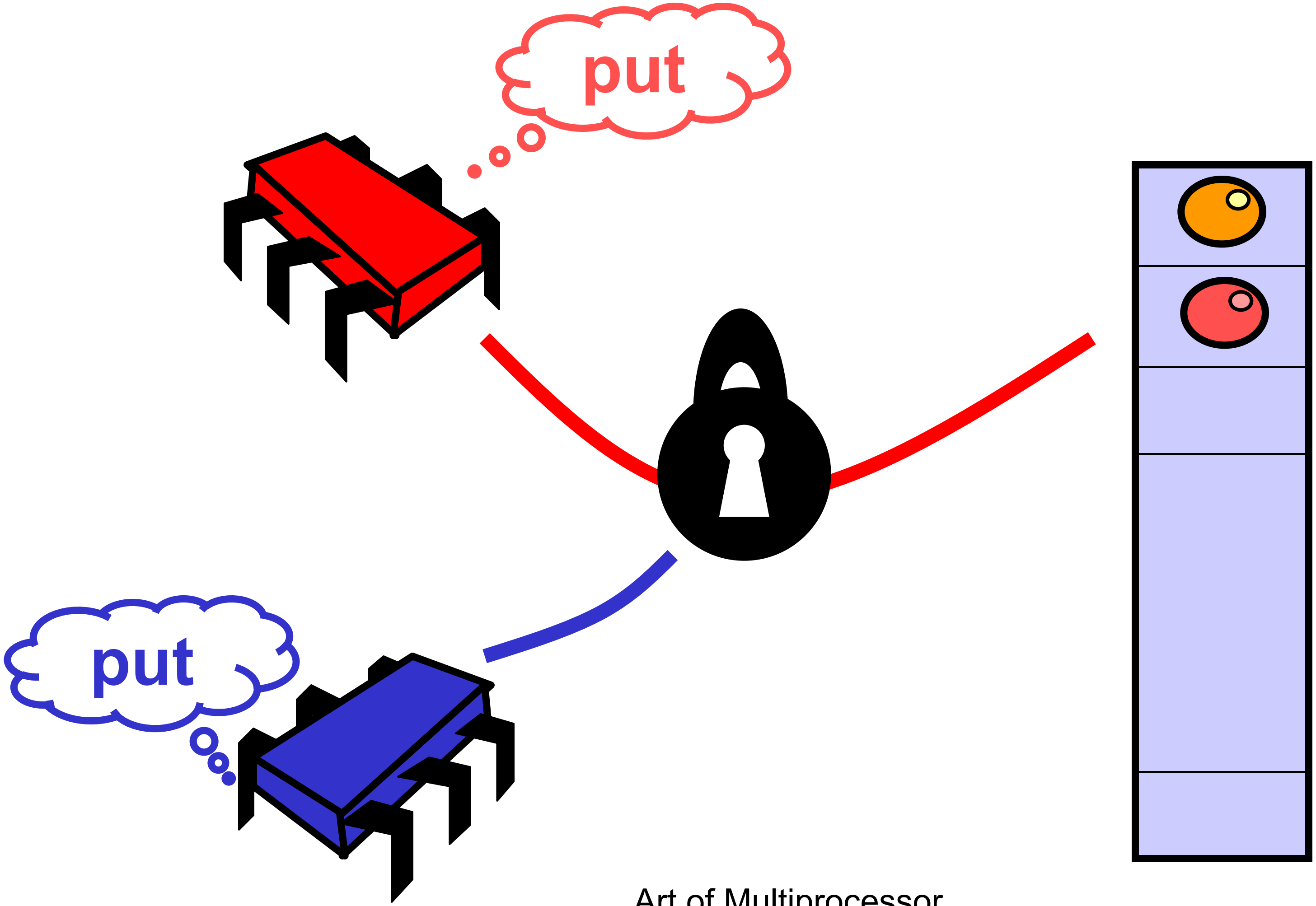
A Shared Pool

- Put
  - Insert item
  - block if full

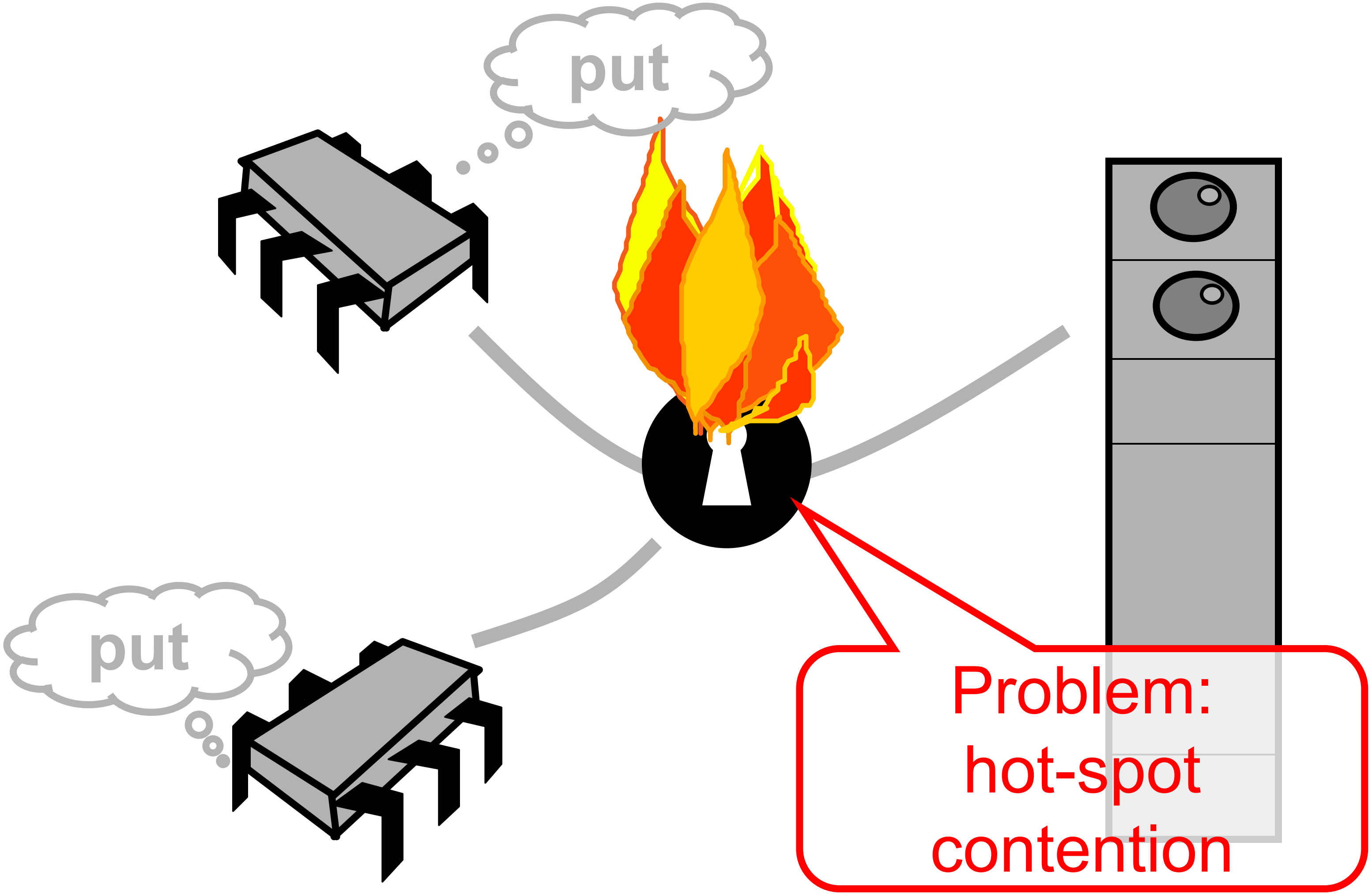
- Remove
  - Remove & return item
  - block if empty

```
public interface Pool<T> {
  public void put(T x);
  public T remove();
}
```

Simple Locking Implementation

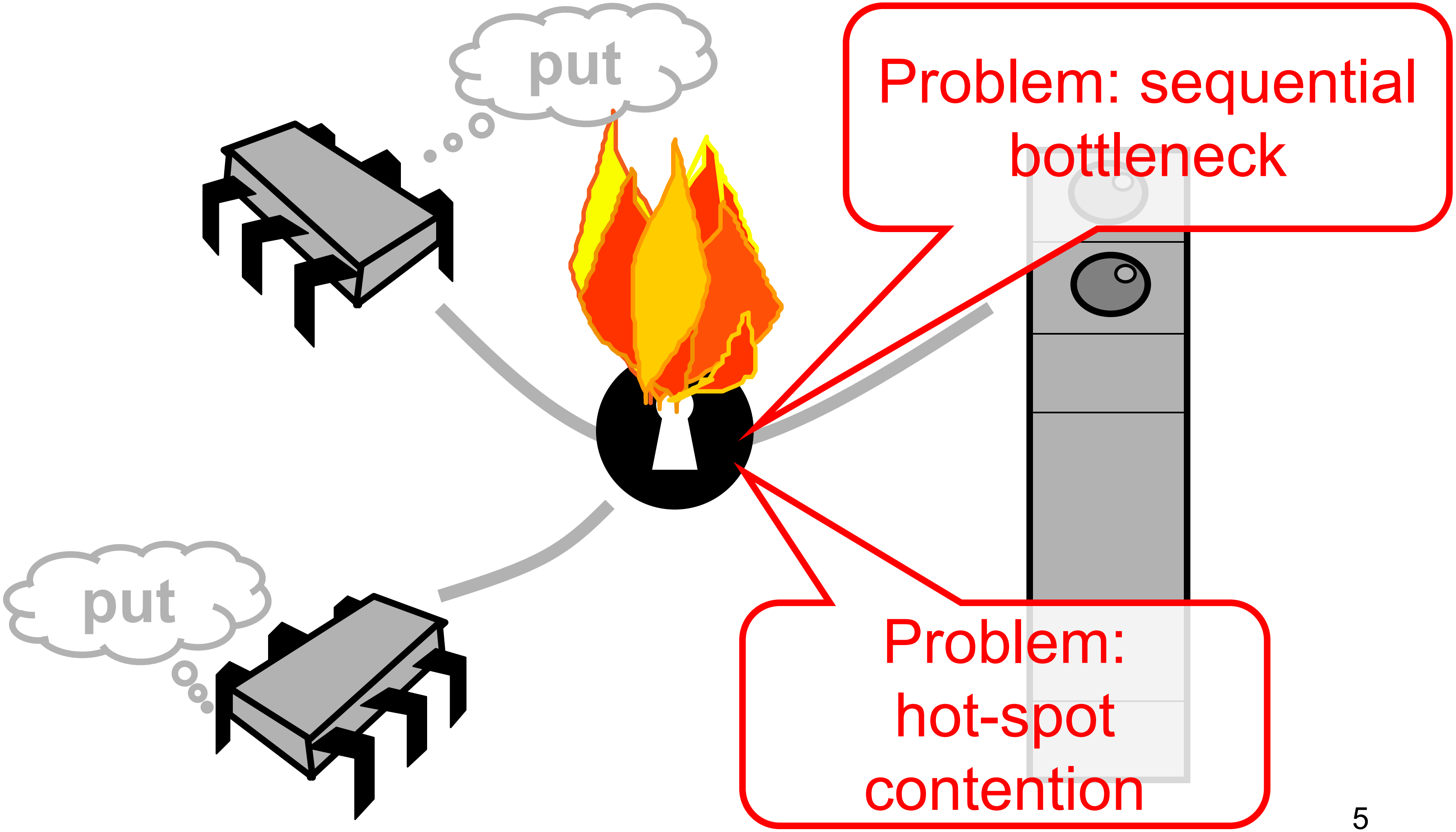


Simple Locking Implementation



Problem:
hot-spot
contention

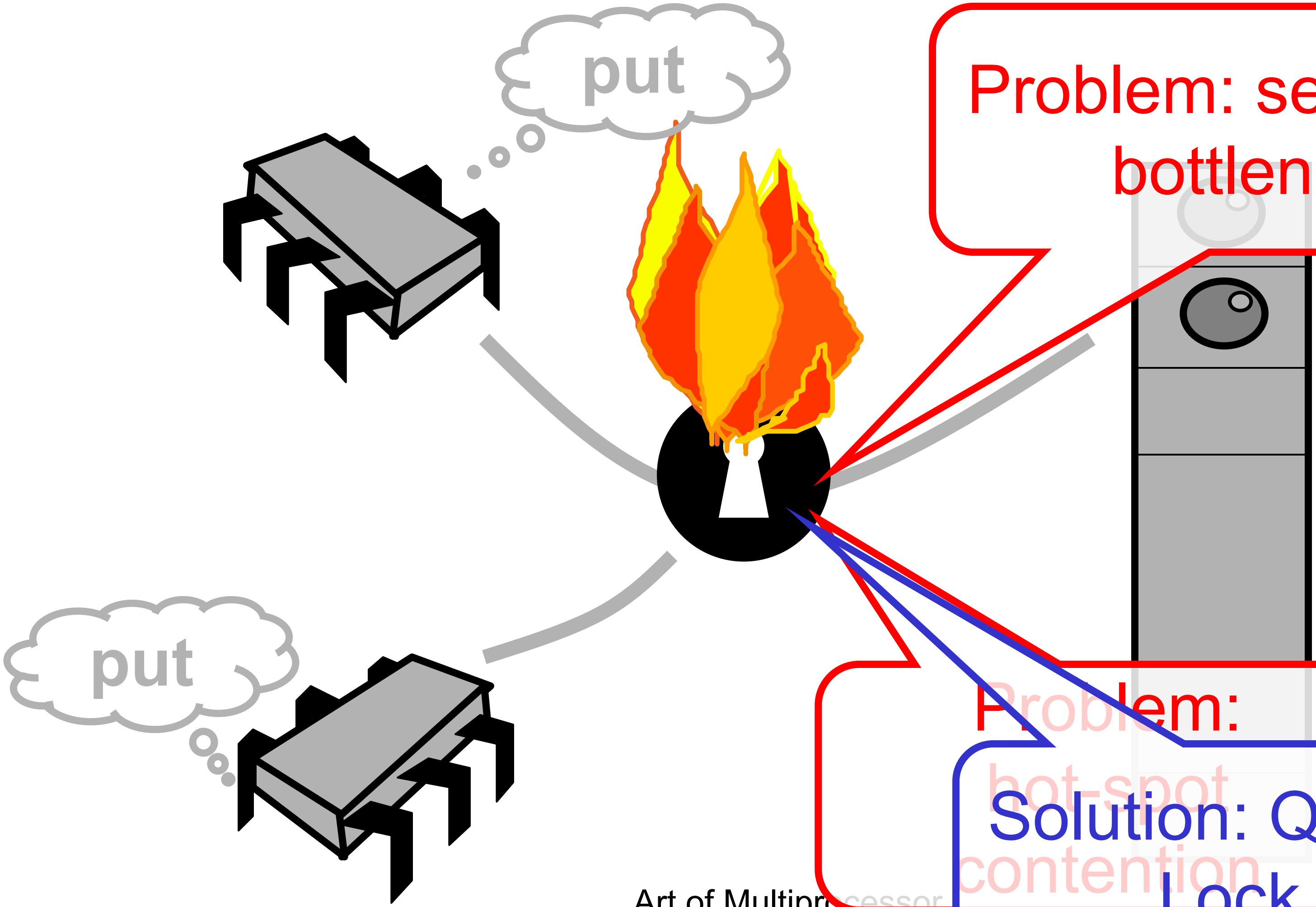
Simple Locking Implementation



Problem: sequential bottleneck

Problem: hot-spot contention

Simple Locking Implementation

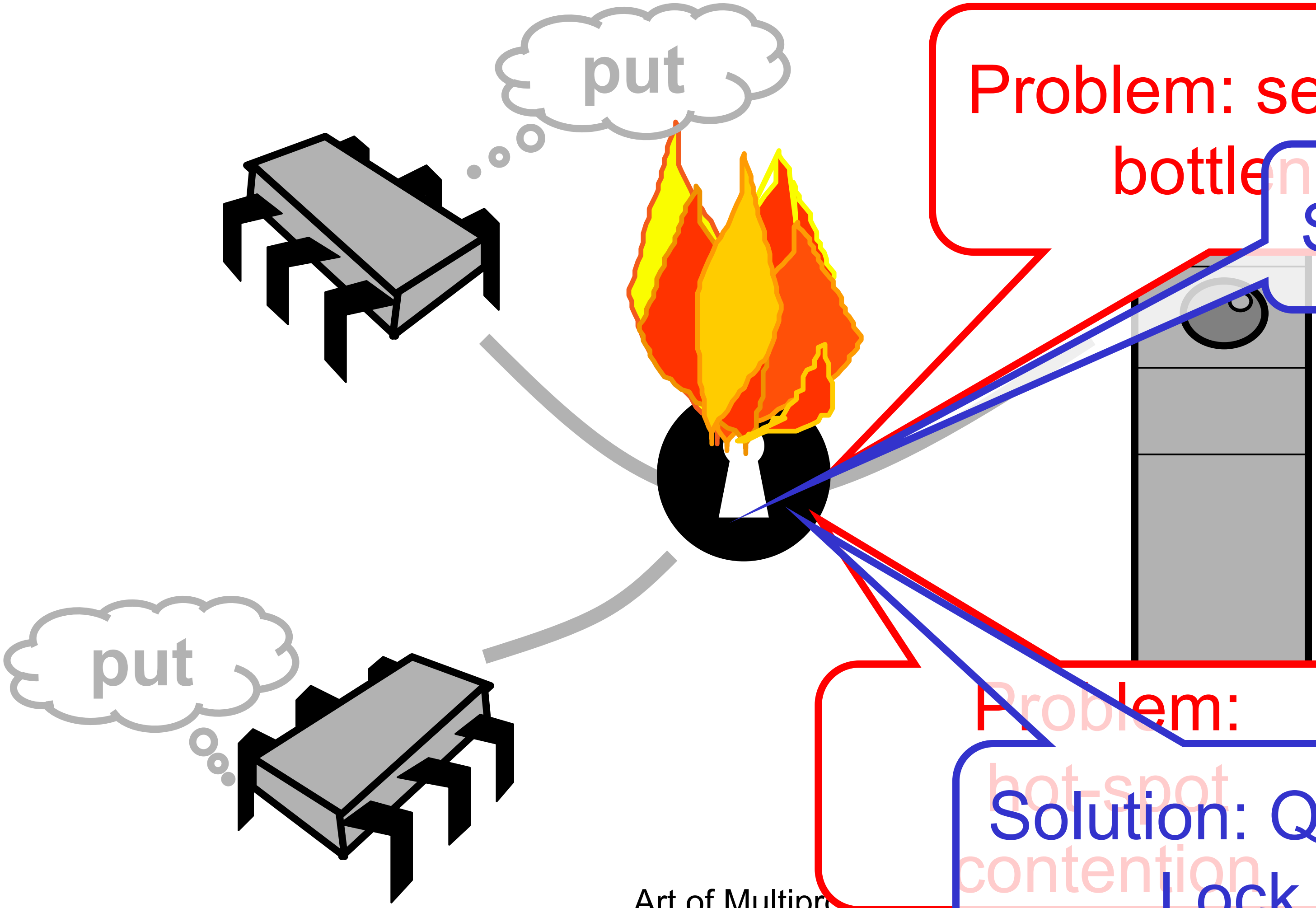


Problem: sequential bottleneck

Problem: hot-spot contention

Solution: Queue Lock

Simple Locking Implementation



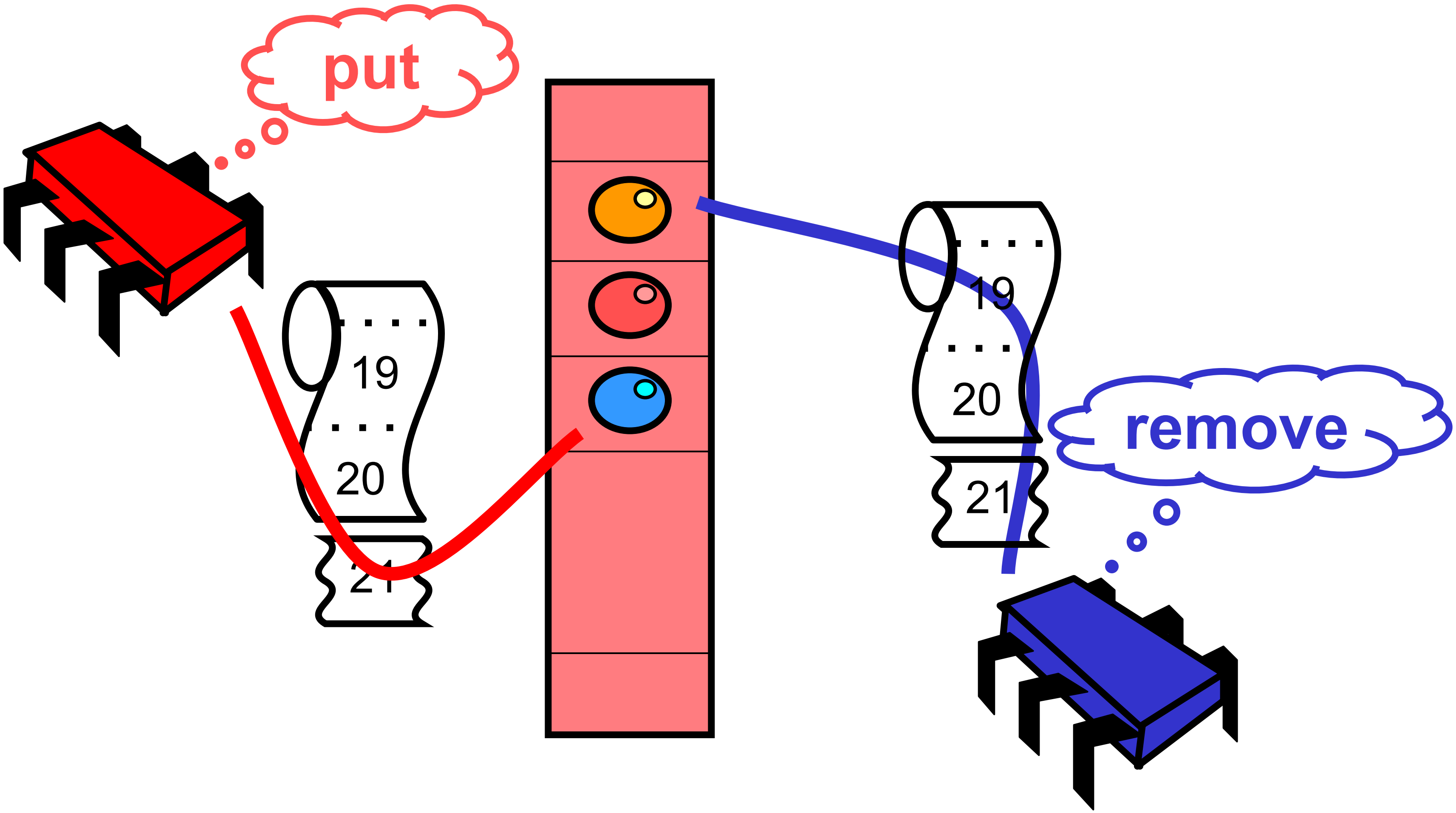
Problem: sequential bottleneck

Solution:?

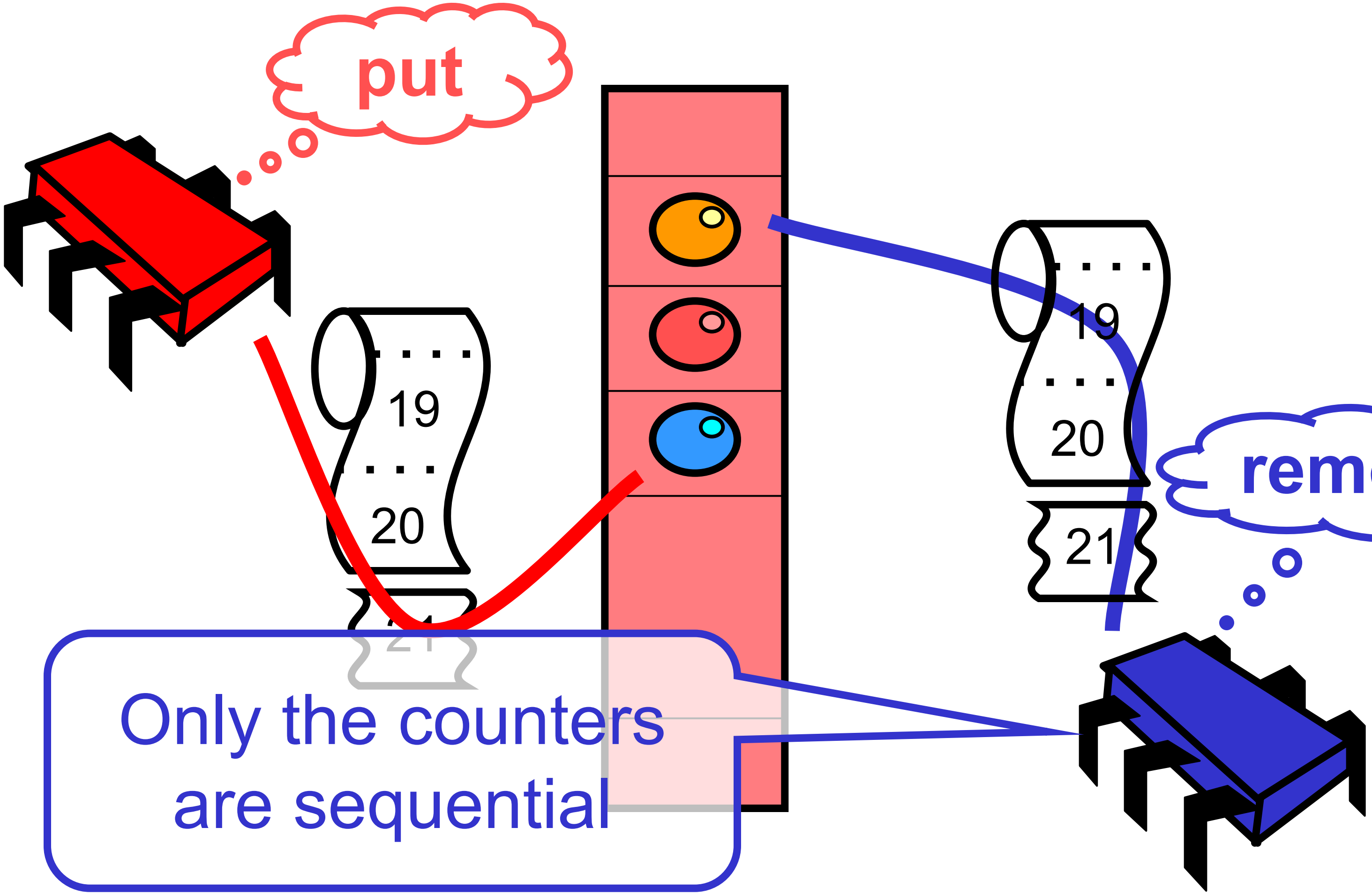
Problem: hot-spot contention

Solution: Queue Lock

Counting Implementation

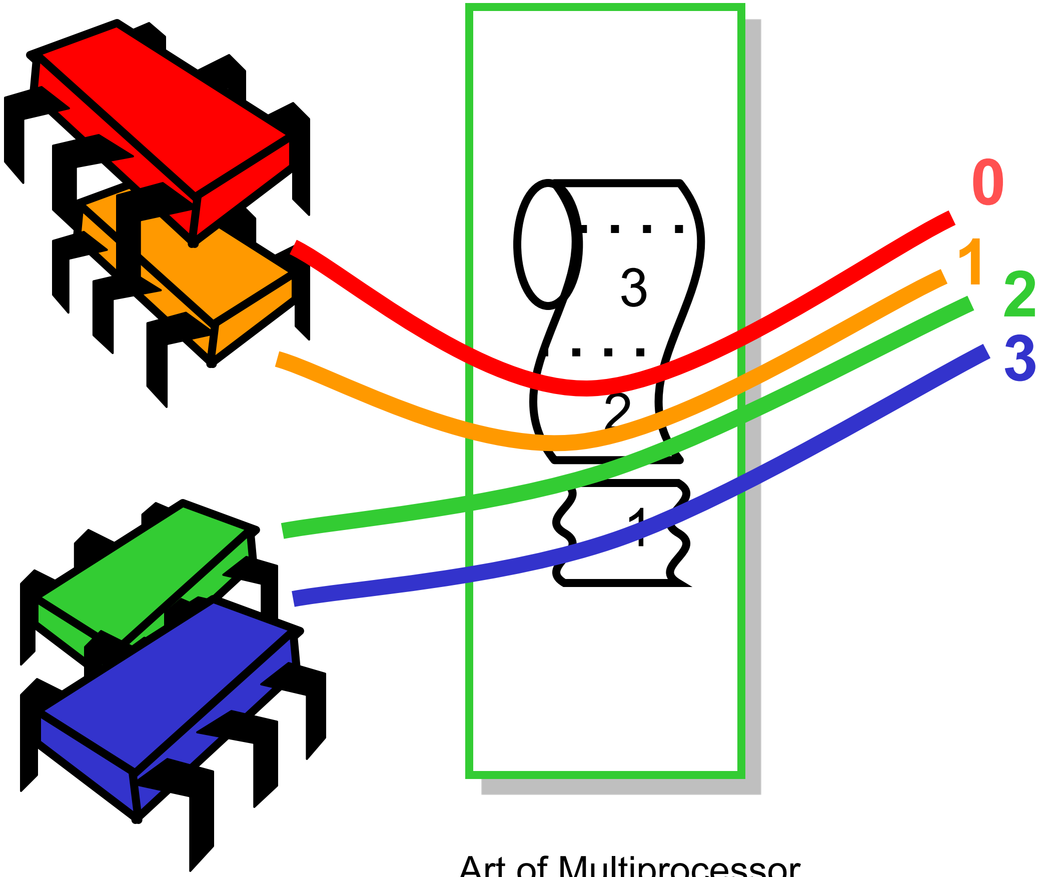


Counting Implementation

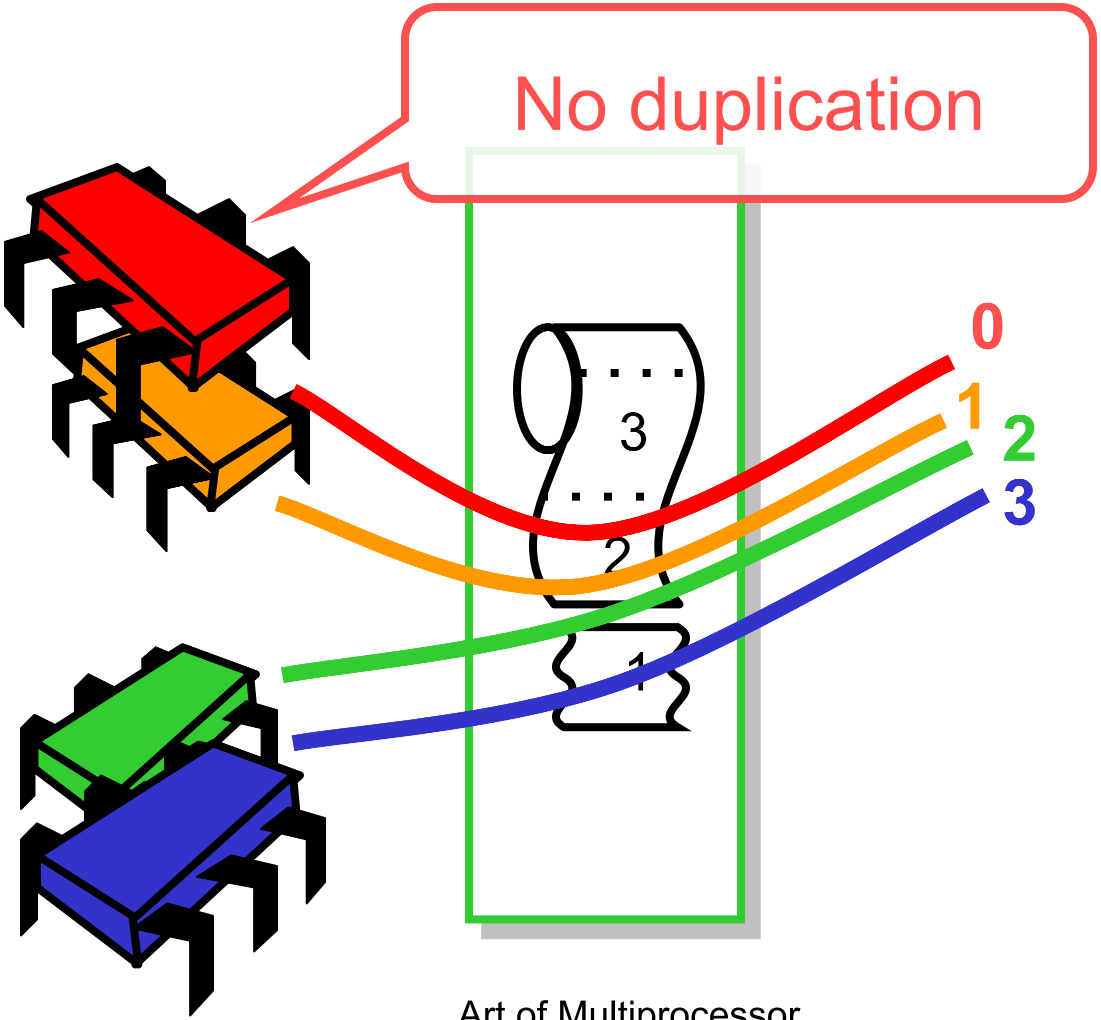


Only the counters are sequential

Shared Counter

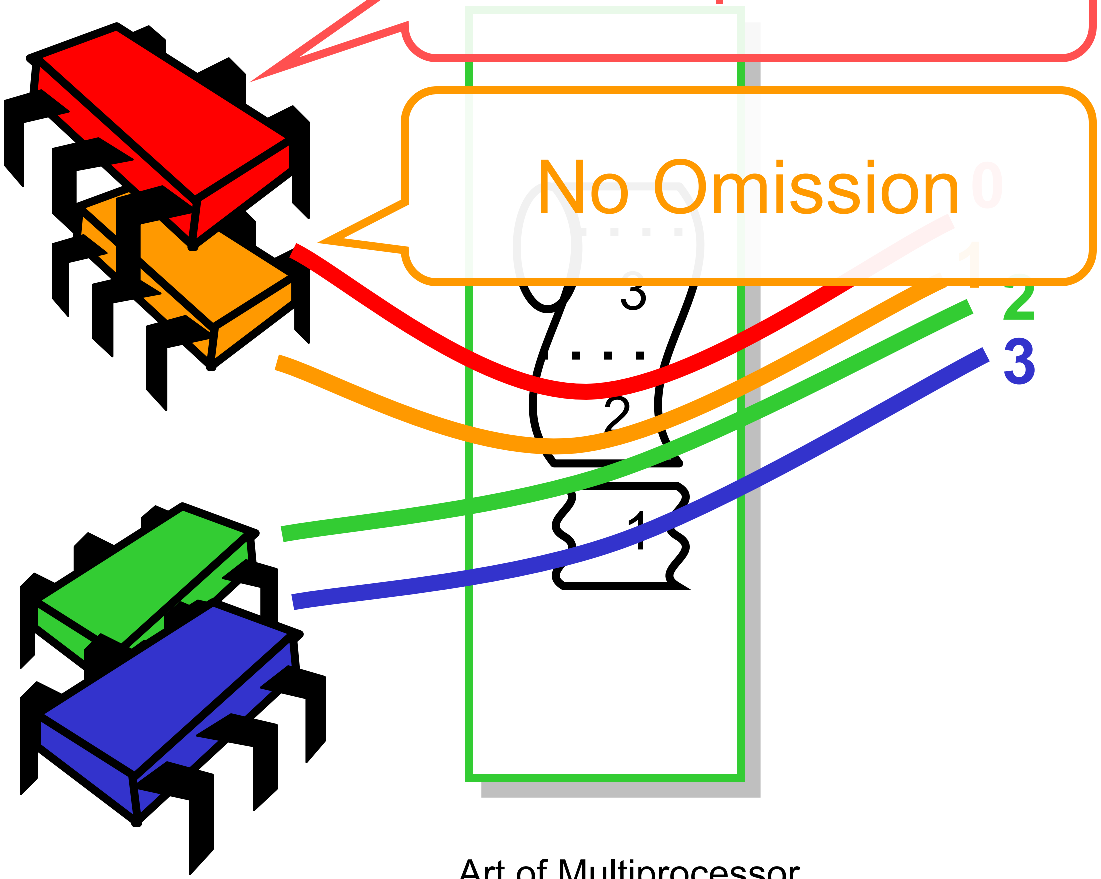


Shared Counter



No duplication

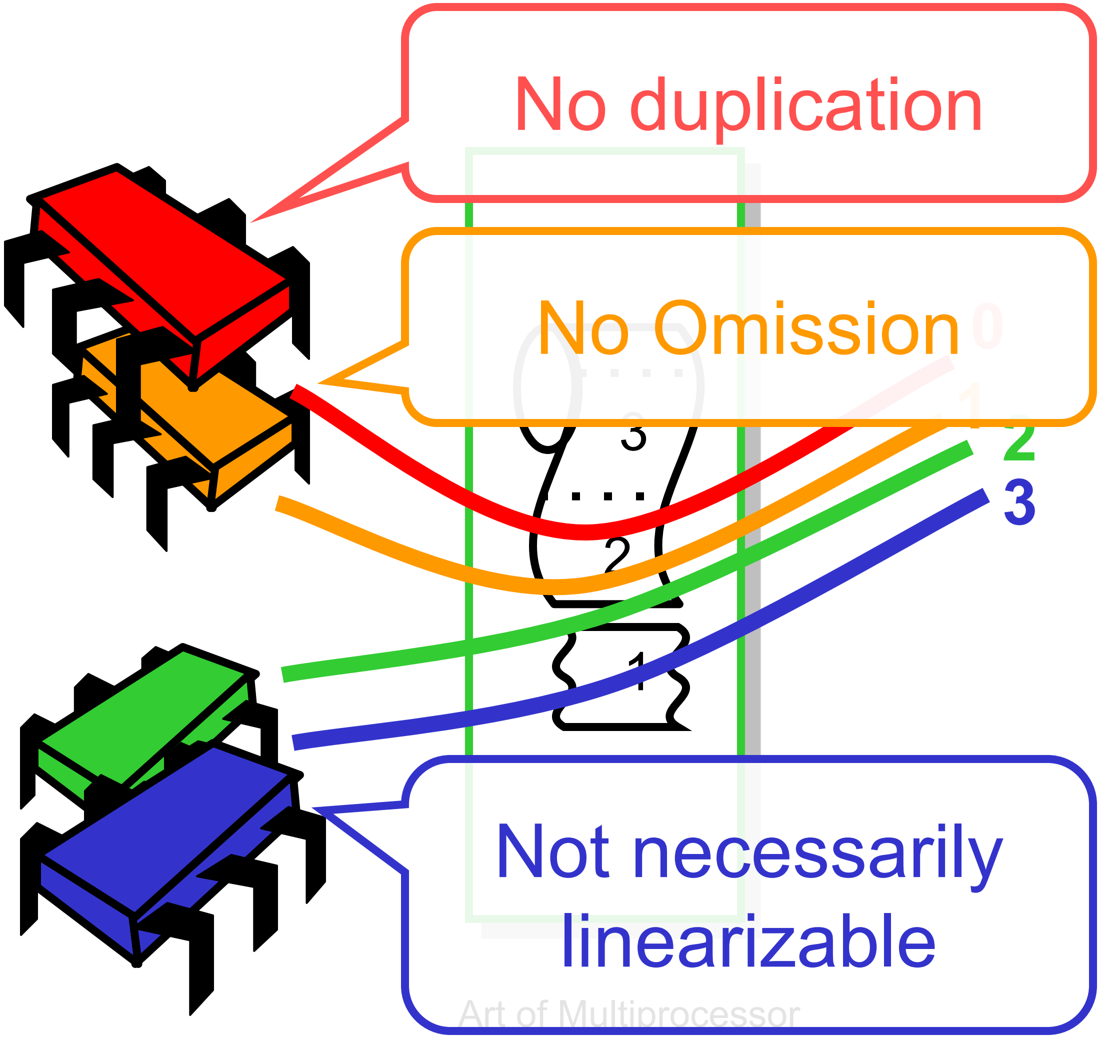
Shared Counter



No duplication

No Omission

Shared Counter



No duplication

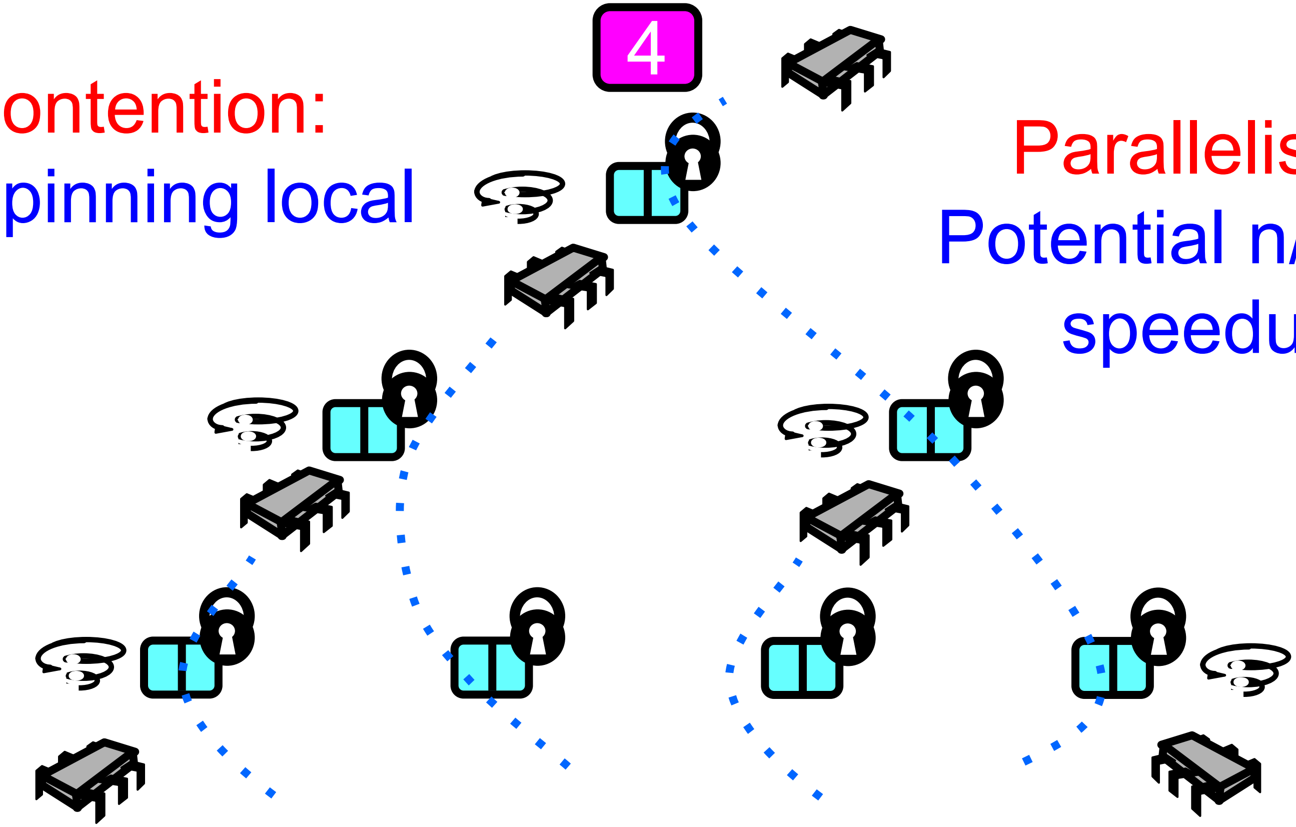
No Omission

Not necessarily linearizable

Shared Counters

- Can we build a shared counter with
  - Low memory contention, and
  - Real parallelism?
- Locking
  - Can use queue locks to reduce contention
  - No help with parallelism issue …

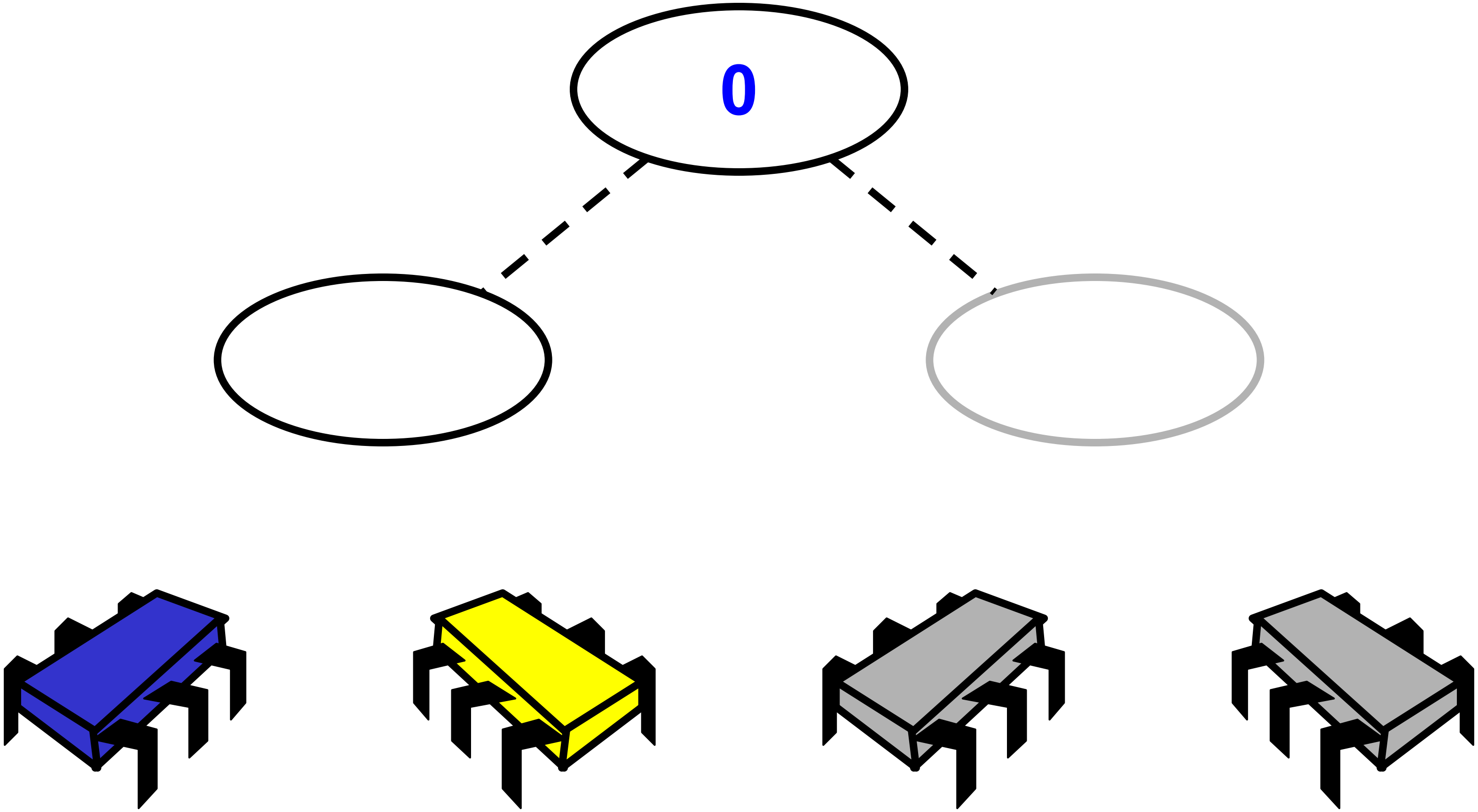
Software Combining Tree



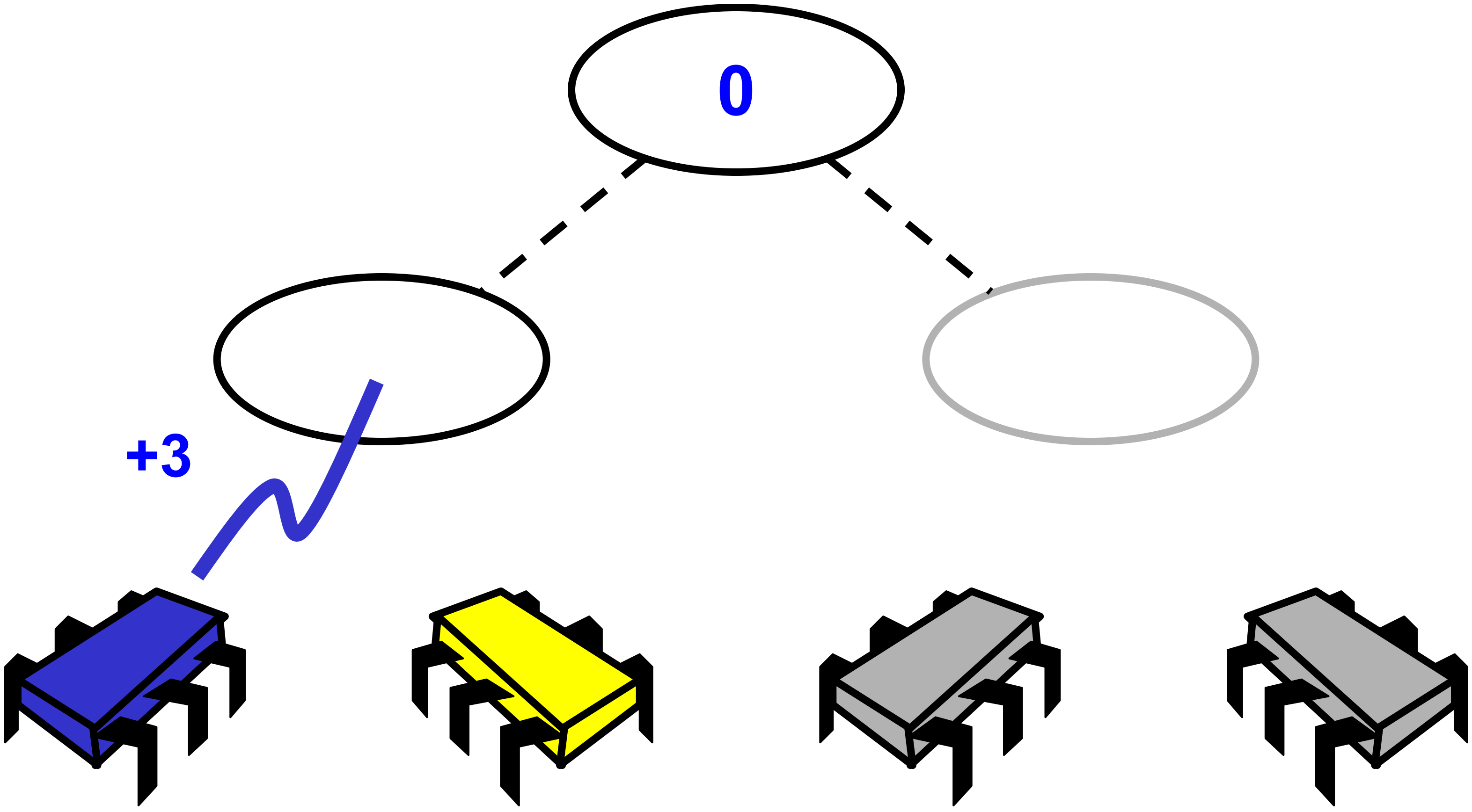
Contention:
All spinning local

Parallelism:
Potential n/log n speedup

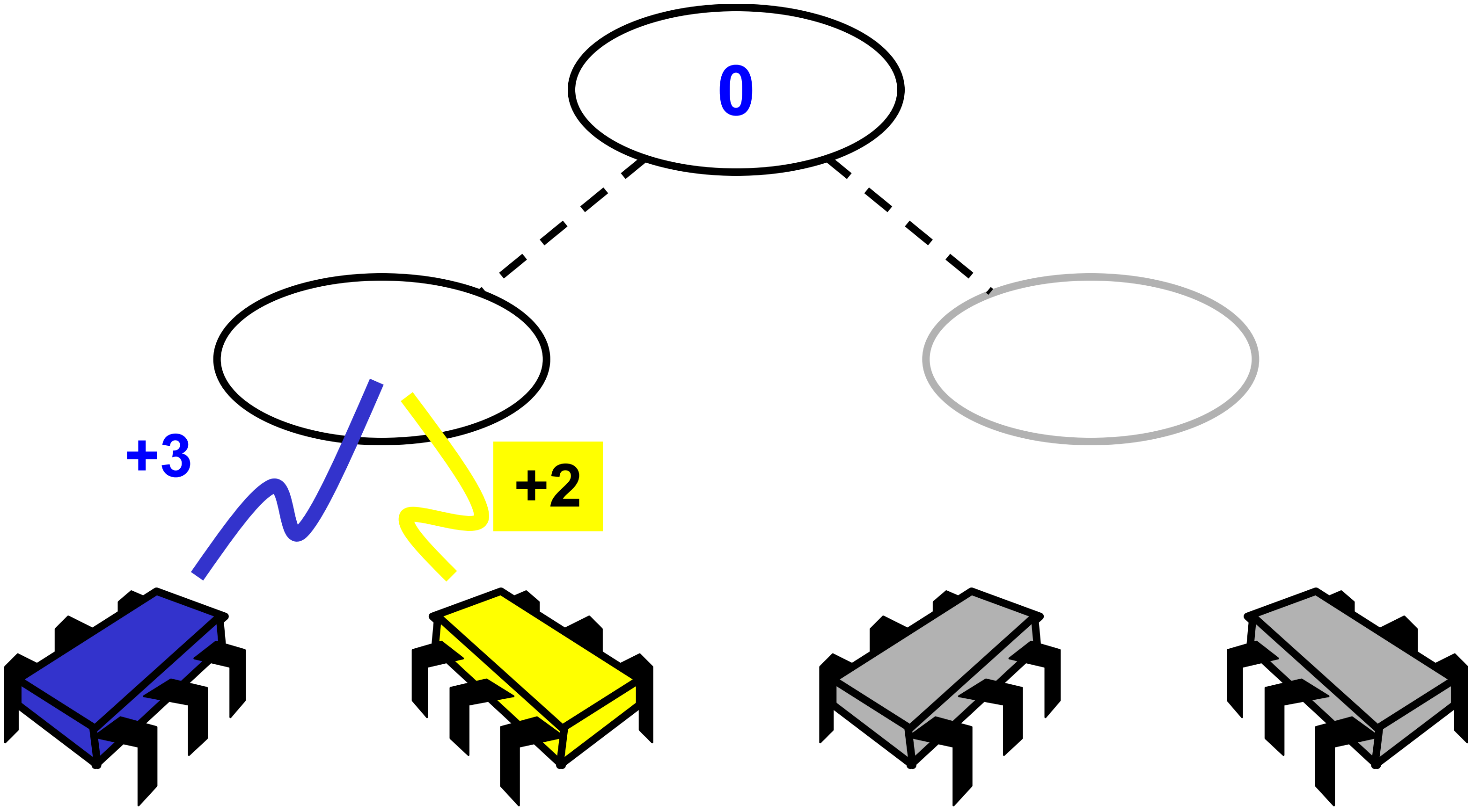
Combining Trees



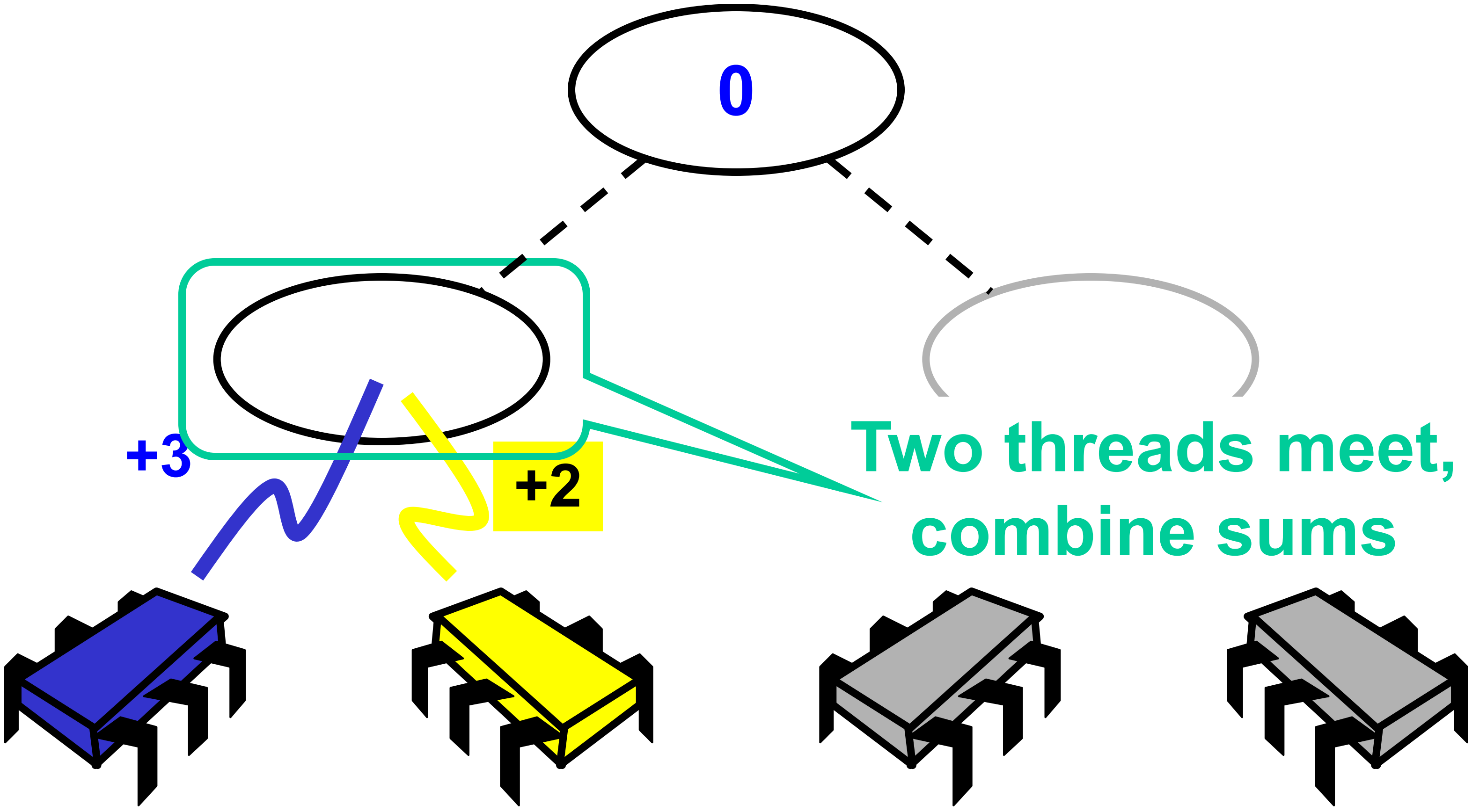
Combining Trees



Combining Trees

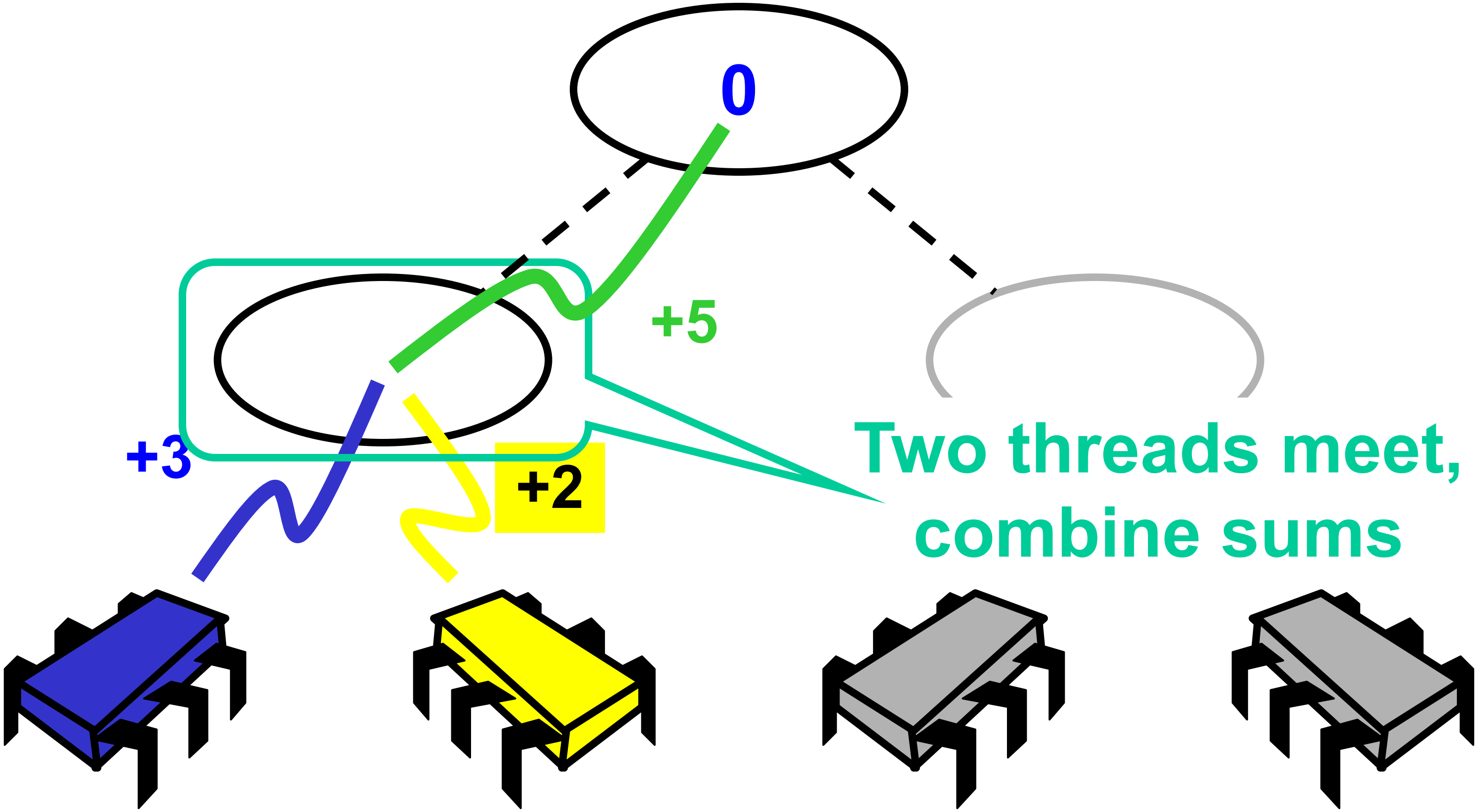


Combining Trees



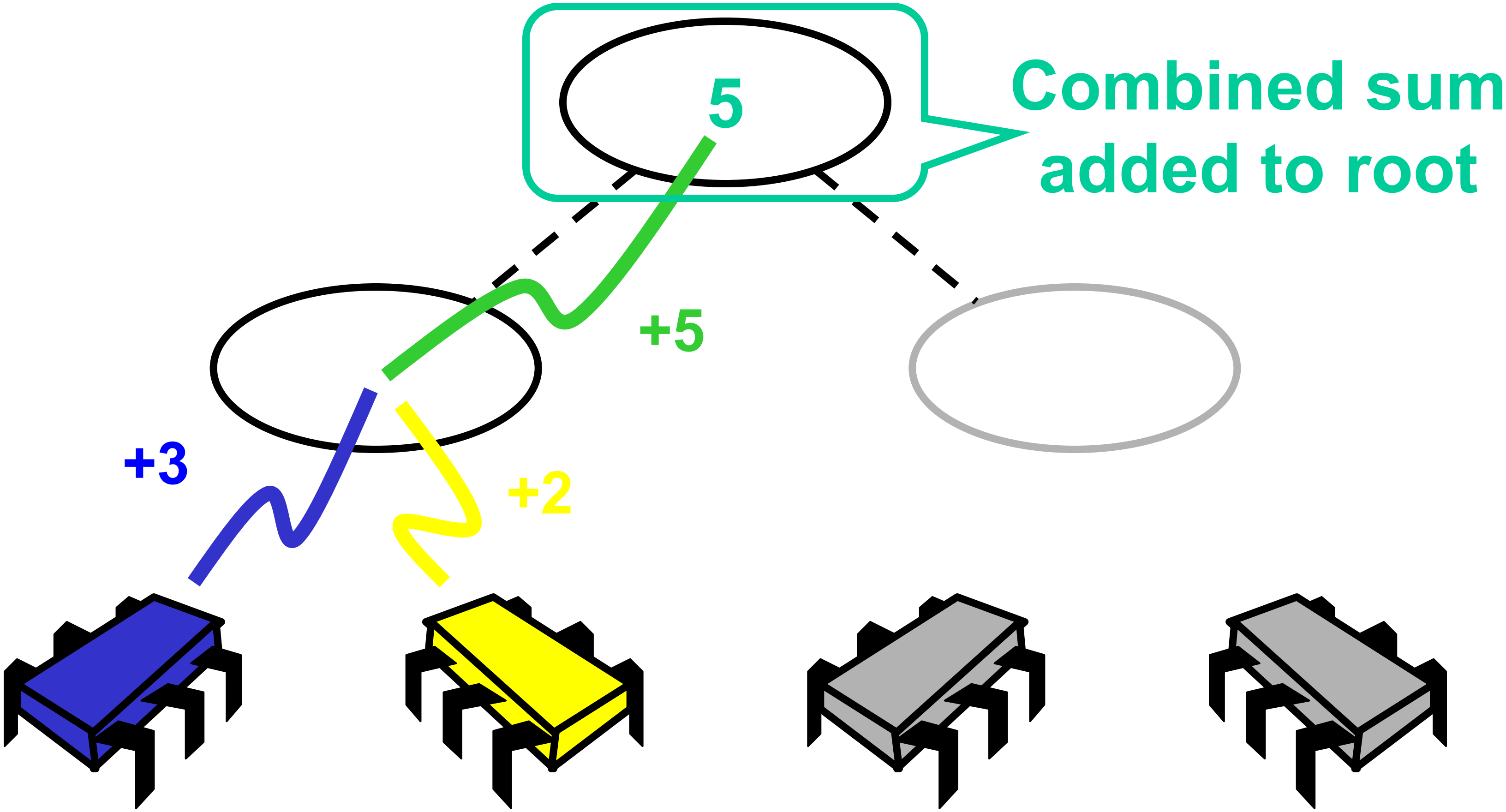
Two threads meet, combine sums

Combining Trees



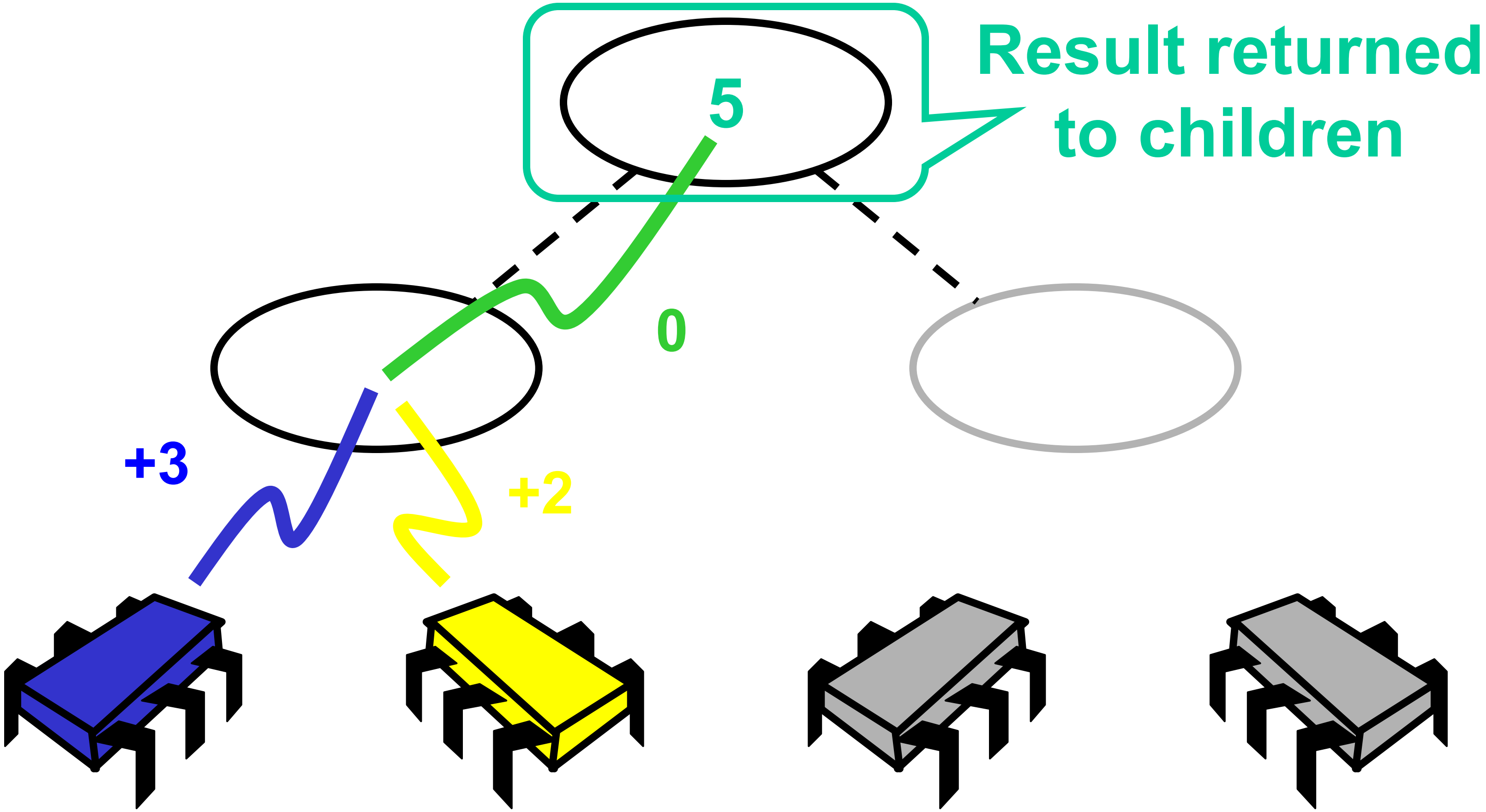
Two threads meet, combine sums

Combining Trees

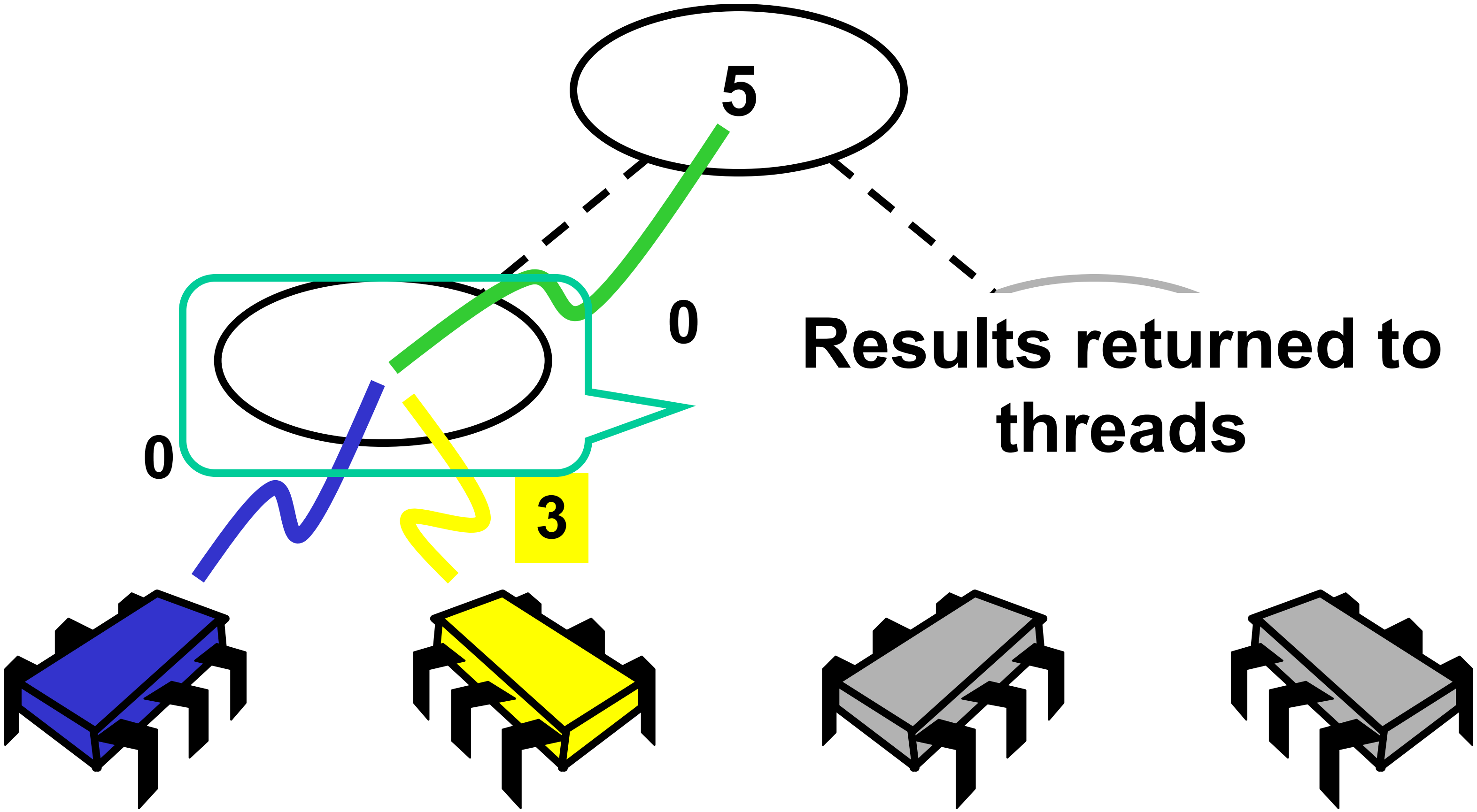


Combined sum added to root

Combining Trees



Combining Trees



Results returned to threads

What if?

- Threads don’t arrive together?
  - Should I stay or should I go?
- How long to wait?
  - Waiting times add up …
- Idea:
  - Use multi-phase algorithm
  - Where threads wait in parallel …

Combining Status

```
enum CStatus{
  IDLE, FIRST, SECOND, RESULT, ROOT
  };
```

Combining Status

```
enum CStatus{
 IDLE, FIRST, SECOND, RESULT, ROOT
 };
```

Nothing going on

Combining Status

```
enum CStatus{
  IDLE, FIRST, SECOND, RESULT, ROOT
 };
```

1st thread is a partner for combining, will return to check for 2nd thread

Combining Status

```
enum CStatus{
  IDLE, FIRST, SECOND, RESULT, ROOT
  };
```

2nd thread has arrived with value for combining

Combining Status

```
enum CStatus{
  IDLE, FIRST, SECOND, RESULT, ROOT
  };
```

1st thread has deposited result for 2nd thread

Combining Status

```
enum CStatus{
  IDLE, FIRST, SECOND, RESULT, ROOT
  };
```

Special case: root node

Node Synchronization

Use “Meta Locking:”

- Short-term
  - Synchronized methods
  - Consistency during method call
- Long-term
  - Boolean locked field
  - Consistency across calls

Phases

- Precombining
  - Set up combining rendez-vous

Phases

- Precombining
  - Set up combining rendez-vous
- Combining
  - Collect and combine operations

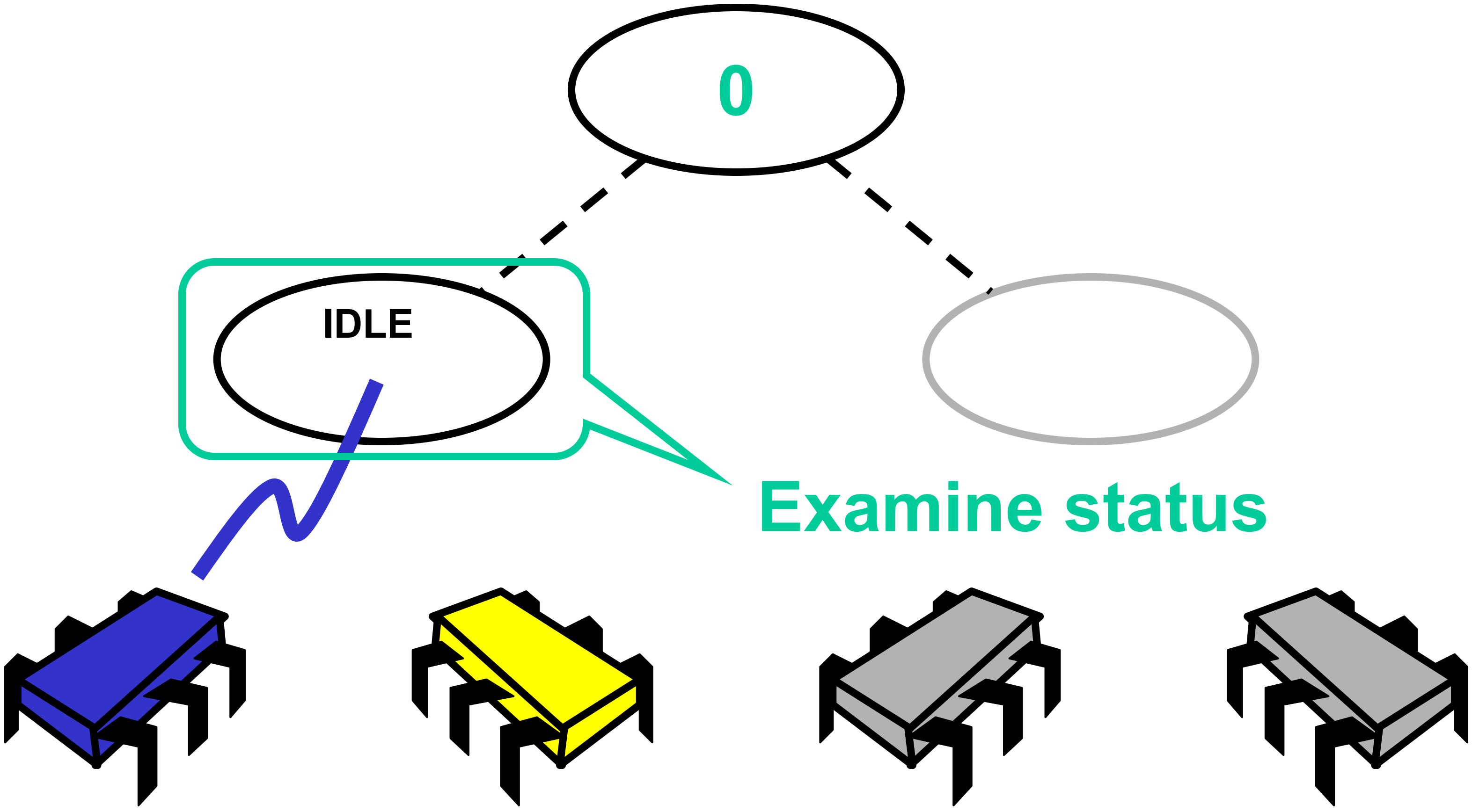
Phases

- Precombining
  - Set up combining rendez-vous
- Combining
  - Collect and combine operations
- Operation
  - Hand off to higher thread

Phases

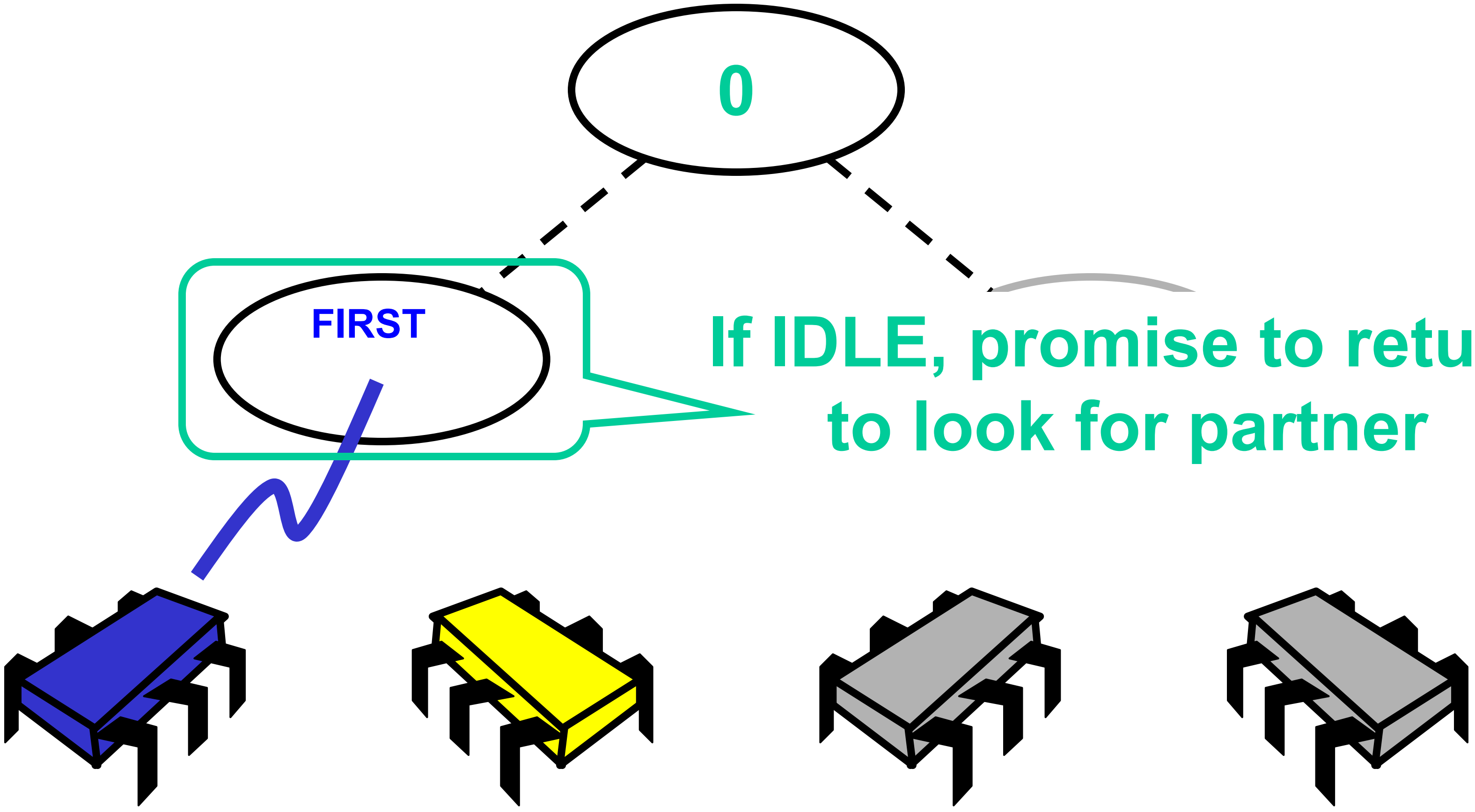
- Precombining
  - Set up combining rendez-vous
- Combining
  - Collect and combine operations
- Operation
  - Hand off to higher thread
- Distribution
  - Distribute results to waiting threads

Precombining Phase



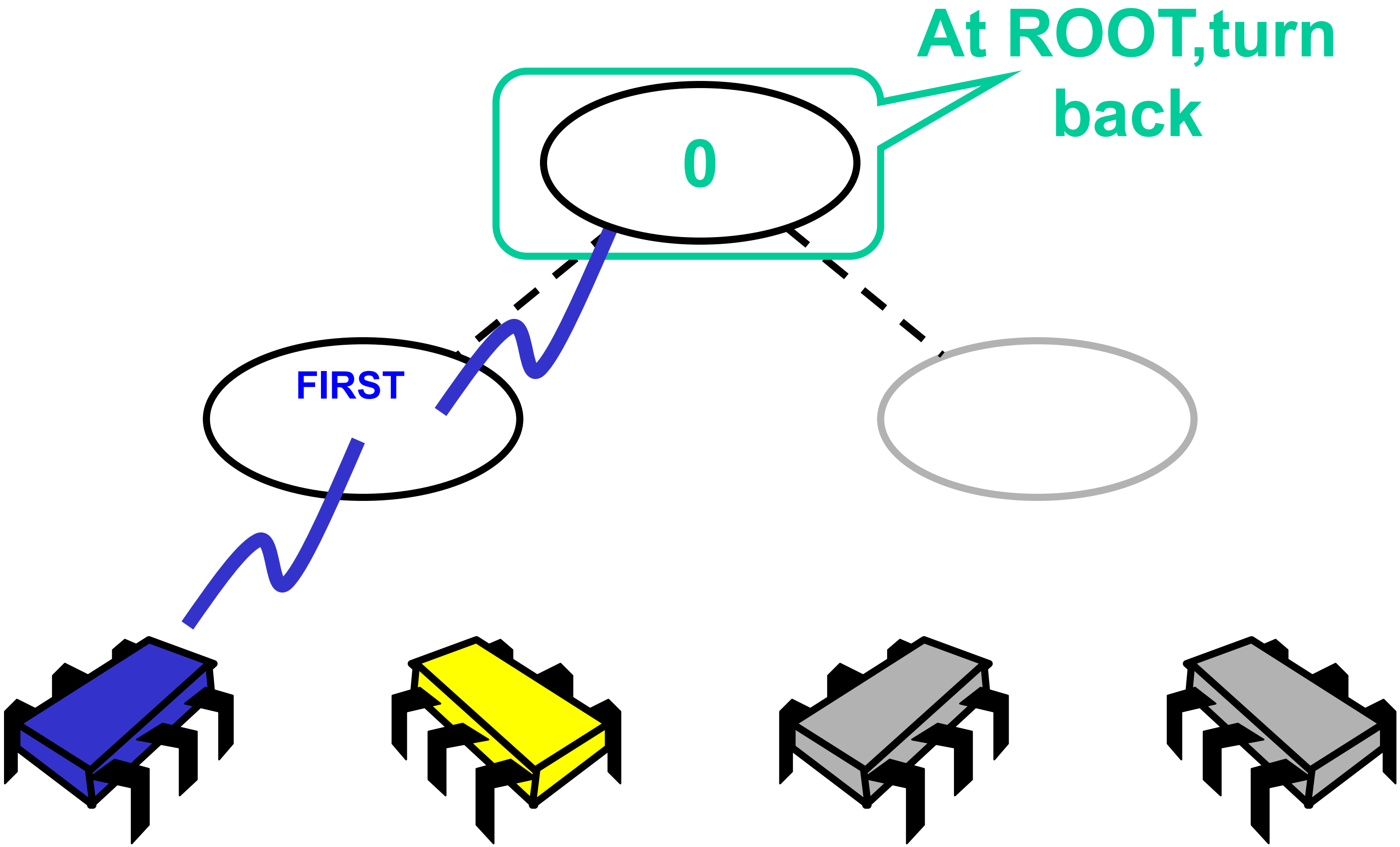
Examine status

Precombining Phase



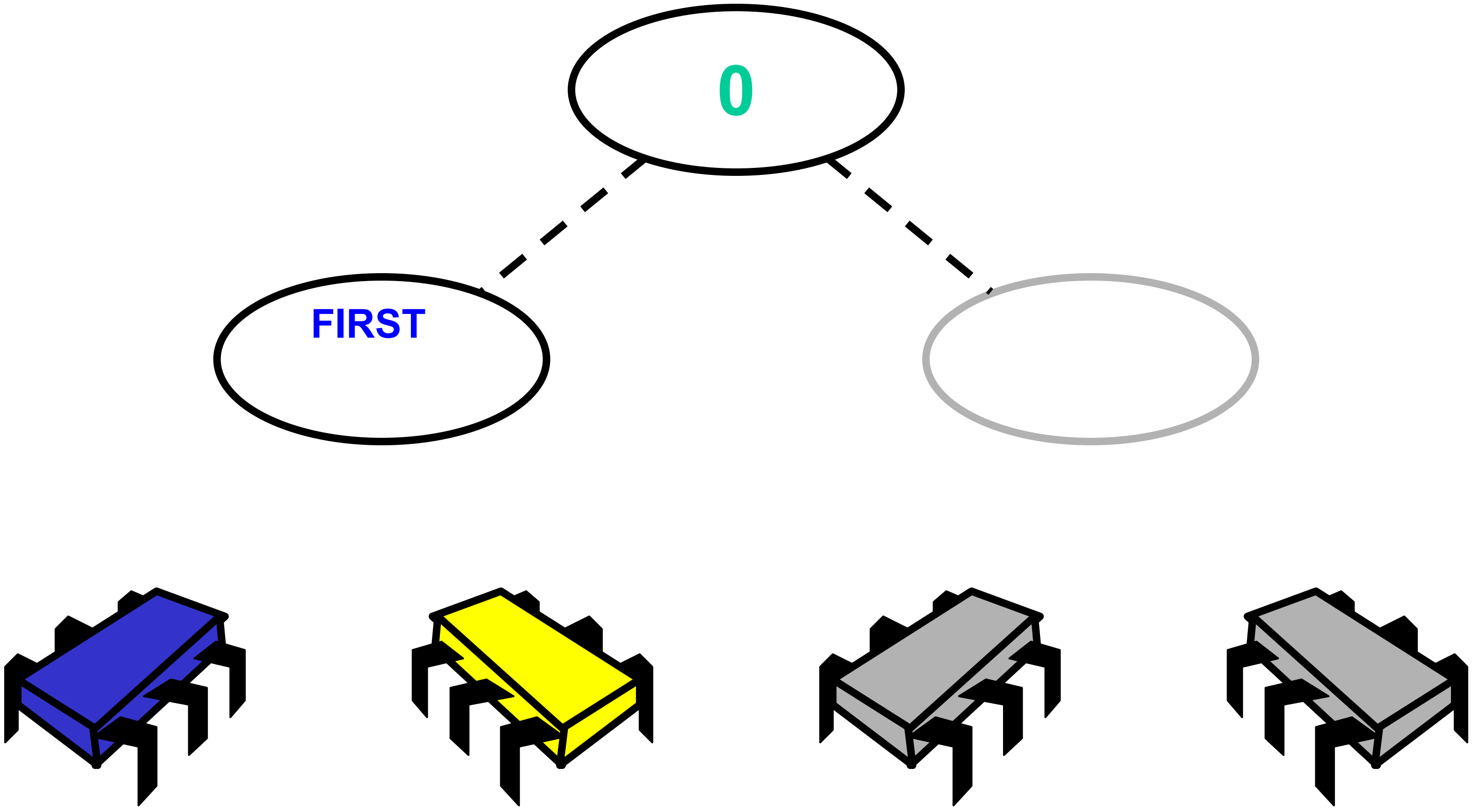
If IDLE, promise to return to look for partner

Precombining Phase

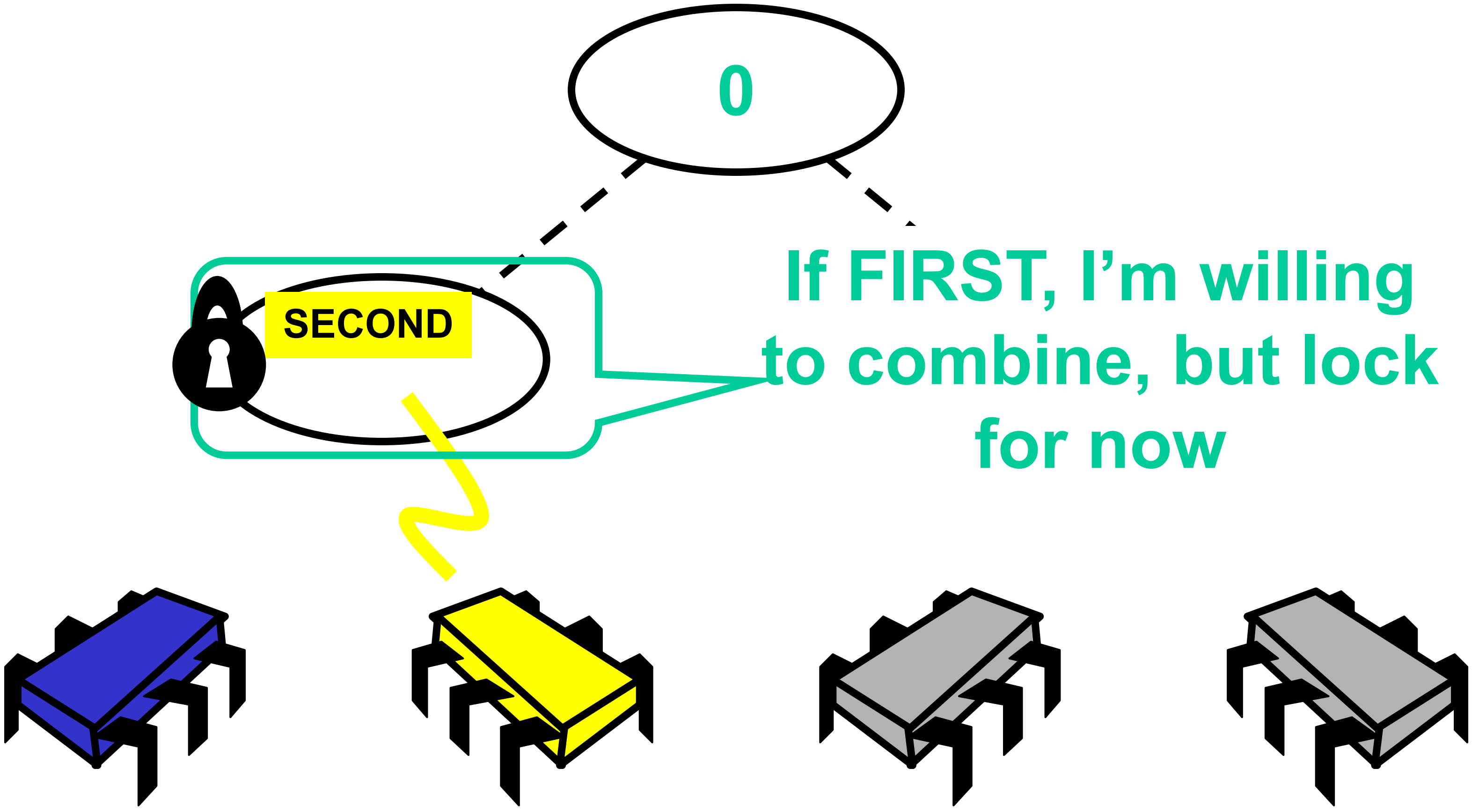


At ROOT,turn back

Precombining Phase



Precombining Phase



If FIRST, I'm willing to combine, but lock for now

Code

- Tree class
  - In charge of navigation
- Node class
  - Combining state
  - Synchronization state
  - Bookkeeping

Precombining Navigation

```
Node node = myLeaf;
while (node.precombine()) {
  node = node.parent;
  }
Node stop = node;
```

Precombining Navigation

```
Node node = myLeaf;
while (node.precombine()) {
  node = node.parent;
  }
Node stop = node;
```

Start at leaf

Precombining Navigation

```
Node node = myLeaf;
while (node.precombine()) {
  node = node.parent;
  }
Node stop = node;
```

Move up while instructed to do so

Precombining Navigation

```
Node node = myLeaf;
while (node.precombine()) {
  node = node.parent;
  }
Node stop = node;
```

Remember where we stopped

Precombining Node

```
synchronized boolean precombine() {
 while (locked) wait();
 switch (cStatus) {
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
  }
}
```

Precombining Node

```
synchronized boolean precombine() {
 while (locked) wait();
 switch (cStatus) {
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
  }
}
```

Short-term synchronization

Synchronization

```
synchronized boolean precombine() {
 while (locked) wait();
 switch (cStatus) {
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;


  case F
  default: throw new PanicException()
  }
}
```

Wait while node is locked
(in use by earlier combining phase)

Precombining Node

```
synchronized boolean precombine() {
 while (locked) wait();
 switch (cStatus) {
  case IDLE: cStatus = CStatus.FIRST;
           return true;
  case FIRST: locked = true;
            cStatus = CStatus.SECOND;
            return false;
  case ROOT: return false;
  default: throw new PanicException()
  }
}
```

Check combining status

Node was IDLE

```
synchronized boolean precombine() {
 while (locked) {wait();}
 switch (cStatus) {
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
```

I will return to look for 2nd thread's input value

Precombining Node

```
synchronized boolean precombine() {
 while (locked) {wait();}
 switch (cStatus) {
  case IDLE: cStatus = CStatus.FIRST;
        return true;
  case FIRST: locked = true;
            cStatus = CStatus.SECOND;
            return f
  case ROOT: re
  default: throw new PanicException()
  }
}
```

Continue up the tree

I’m the 2nd Thread

```
synchronized boolean precombine() {
 while (locked) {wait();}
 switch (cStatus) {
  case IDLE: cStatus = CStatus.FIRST;
           return true;
  case FIRST: locked = true;
            cStatus = CStatus.SECOND;
            return false;
  case ROOT: return false;
  default: throw new PanicException()
```

If 1st thread has promised to return, lock node so it won’t leave without me

Precombining Node

```
synchronized boolean precombine() {
 while (locked) {wait();}
 switch (cStatus) {
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
  }
}
```

Prepare to deposit 2nd thread’s input value

Precombining Node

```
synchronized boolean phase1() {
                                        ait();}

                                        RST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
  }
}
```

End of precombining phase, don’t continue up tree

Node is the Root

If root, precombining phase ends, don't continue up tree

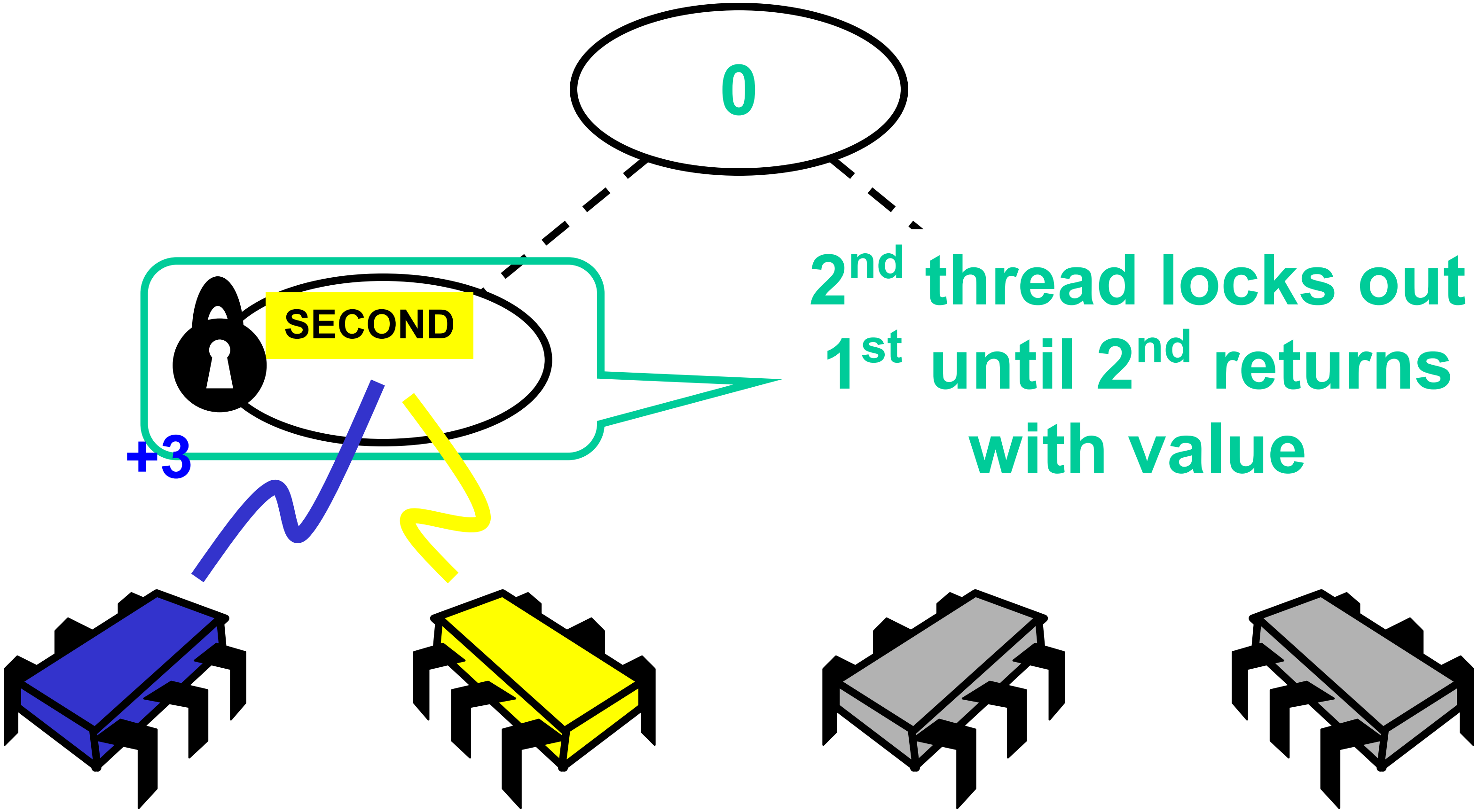
```
                            ) {
                            Y) {wait();}

                            us.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
  }
}
```

Precombining Node

```
synchronized boolean precombine() {
 while (locked) {w
 switch (cStatus)
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
  }
}
```

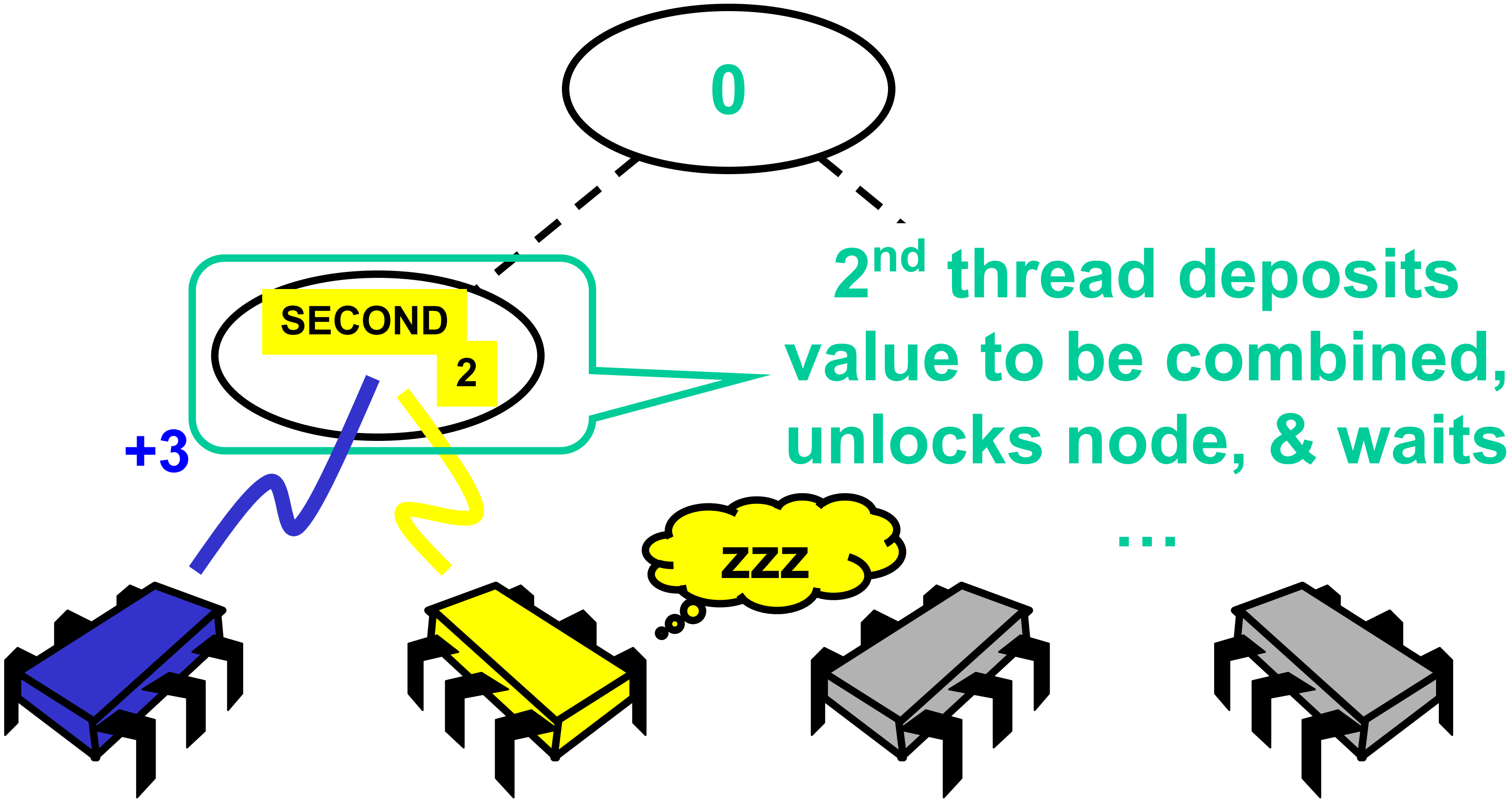
Always check for unexpected values!

Combining Phase



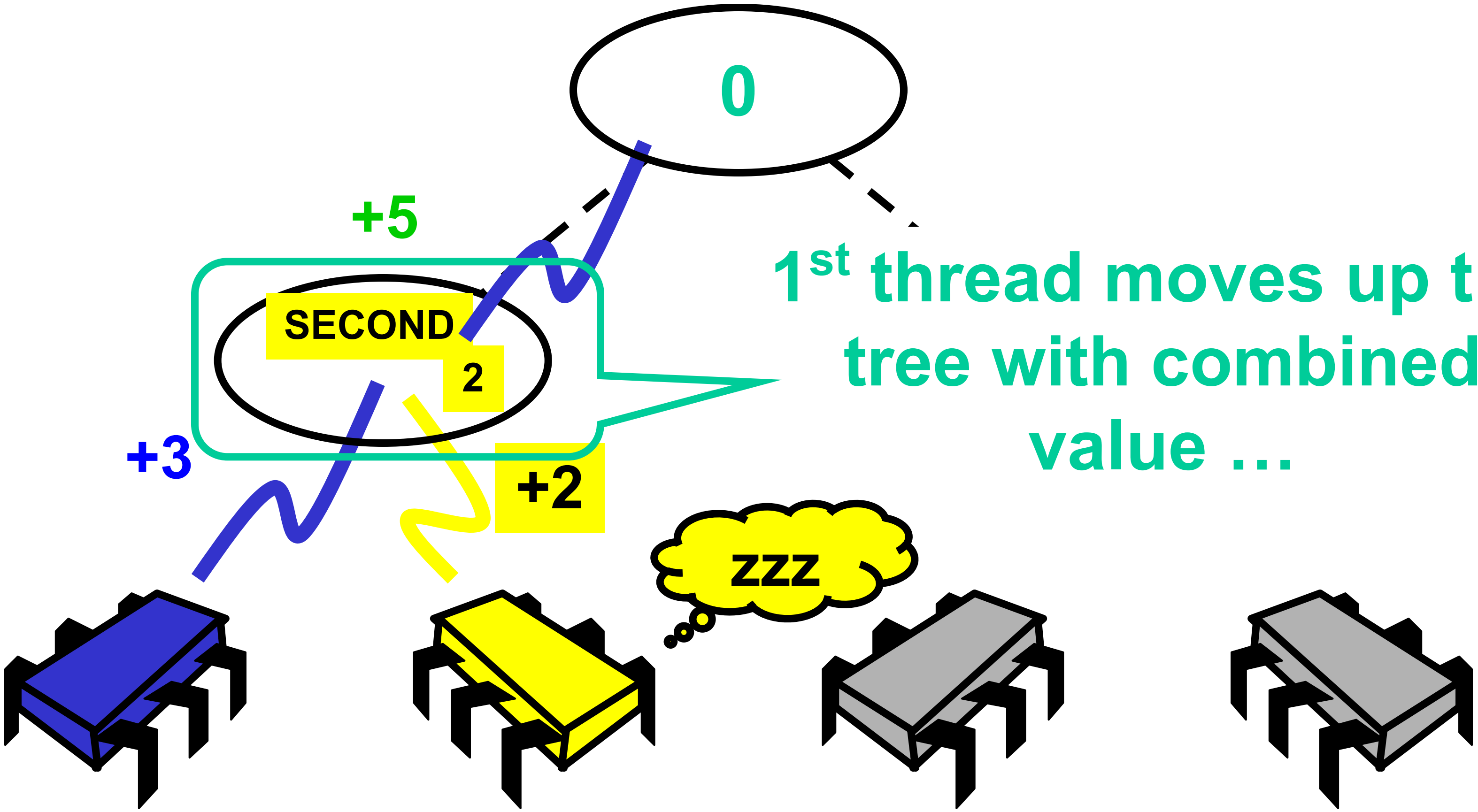
2nd thread locks out 1st until 2nd returns with value

Combining Phase



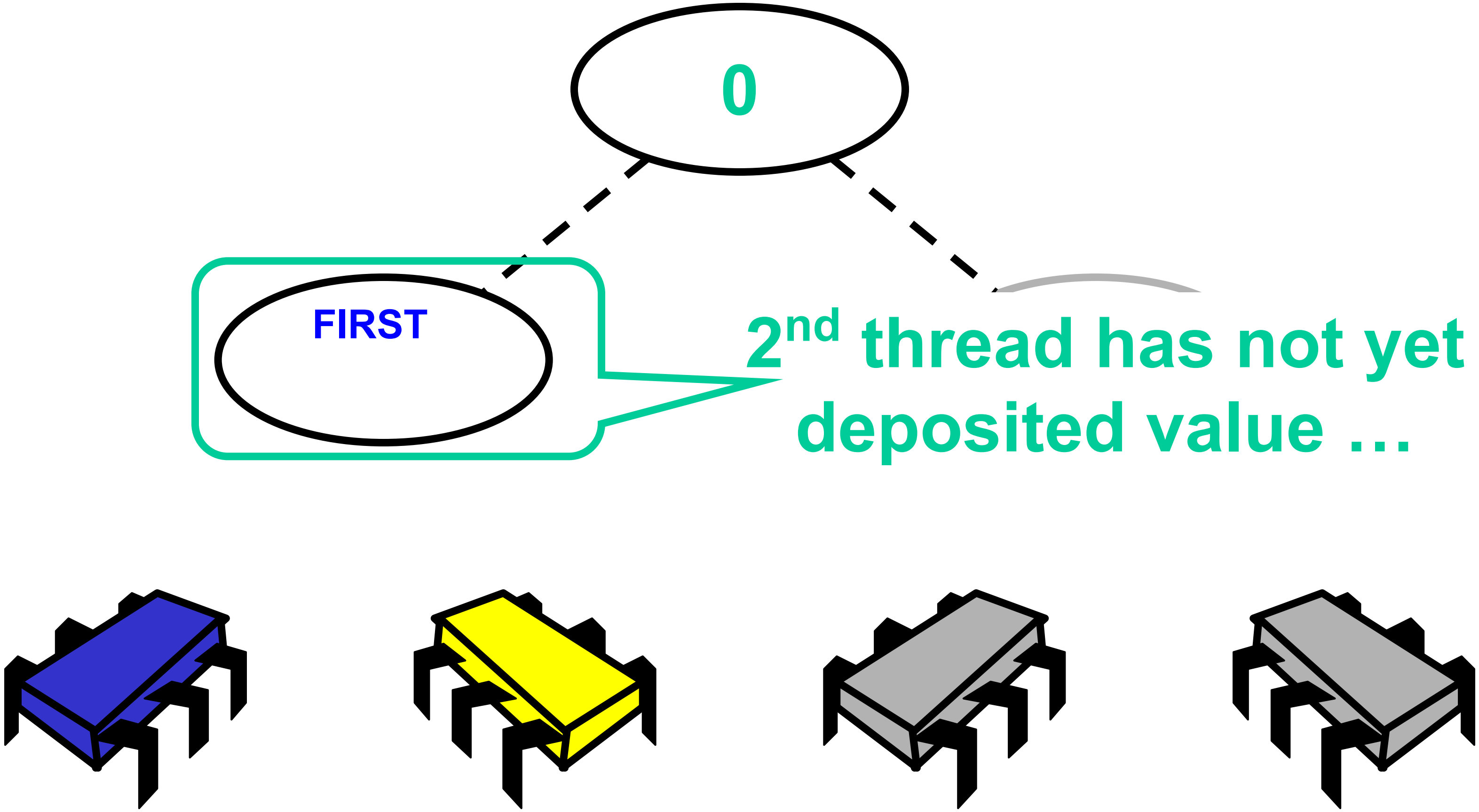
2nd thread deposits value to be combined, unlocks node, & waits

Combining Phase



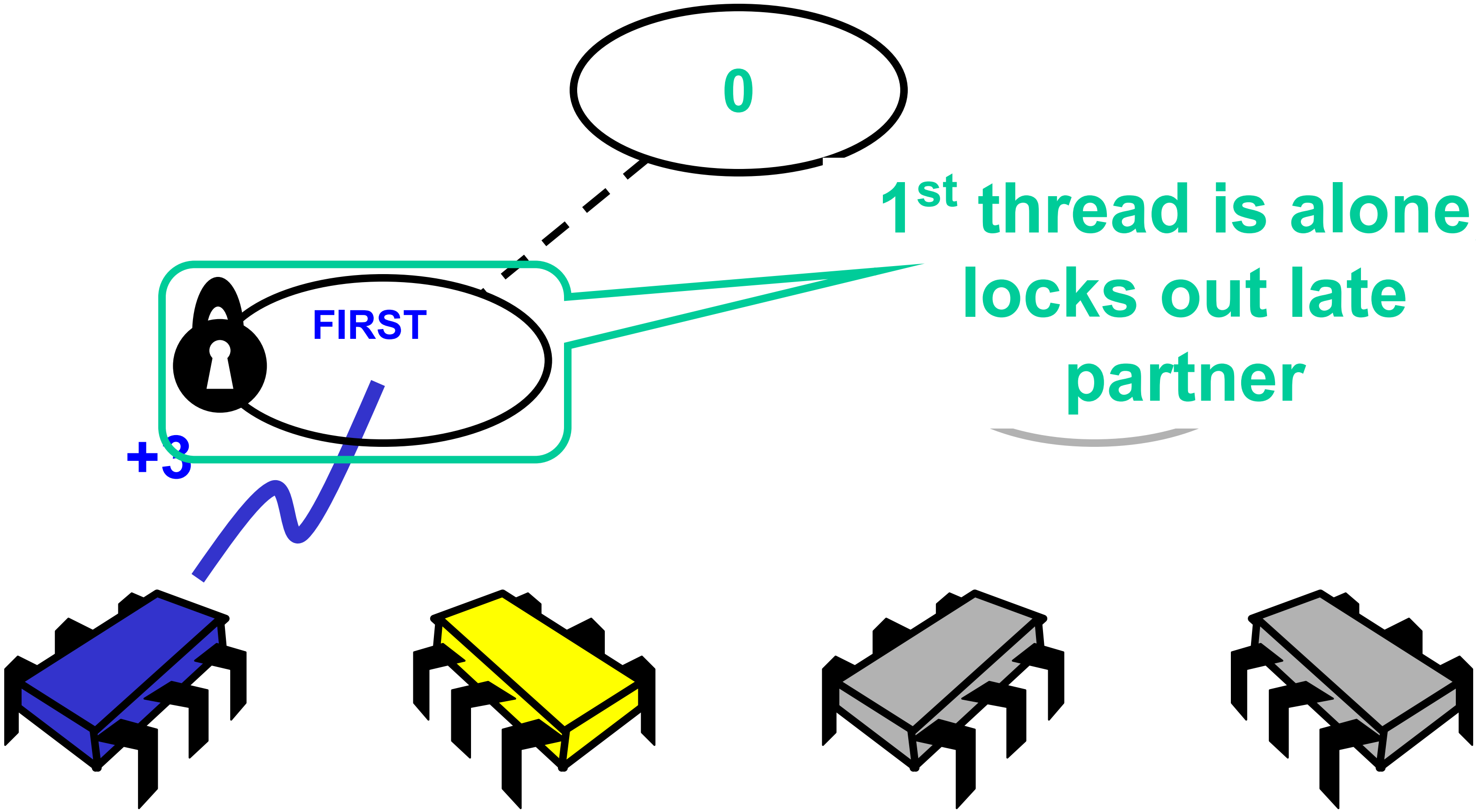
1st thread moves up the tree with combined value …

Combining (reloaded)



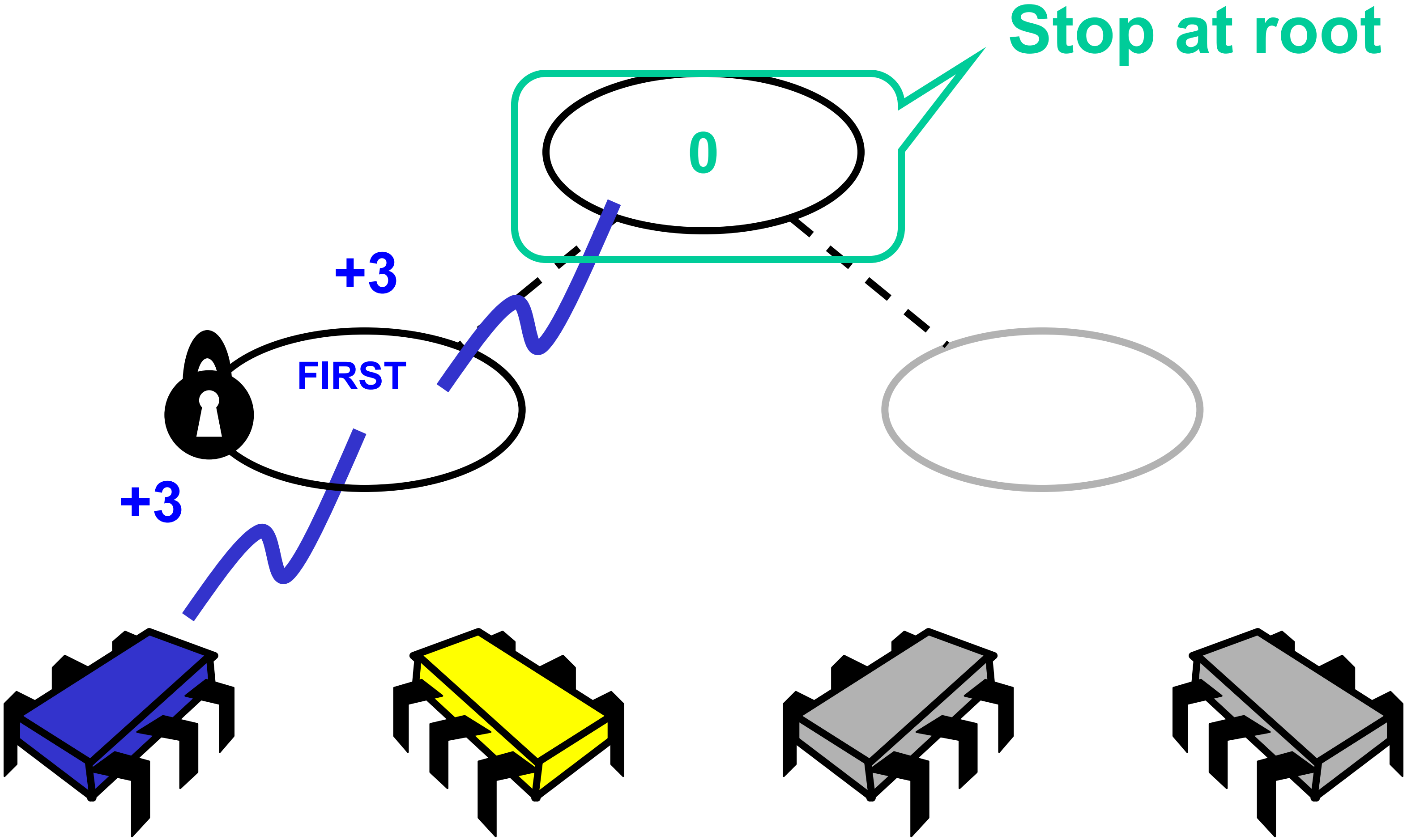
2nd thread has not yet deposited value …

Combining (reloaded)

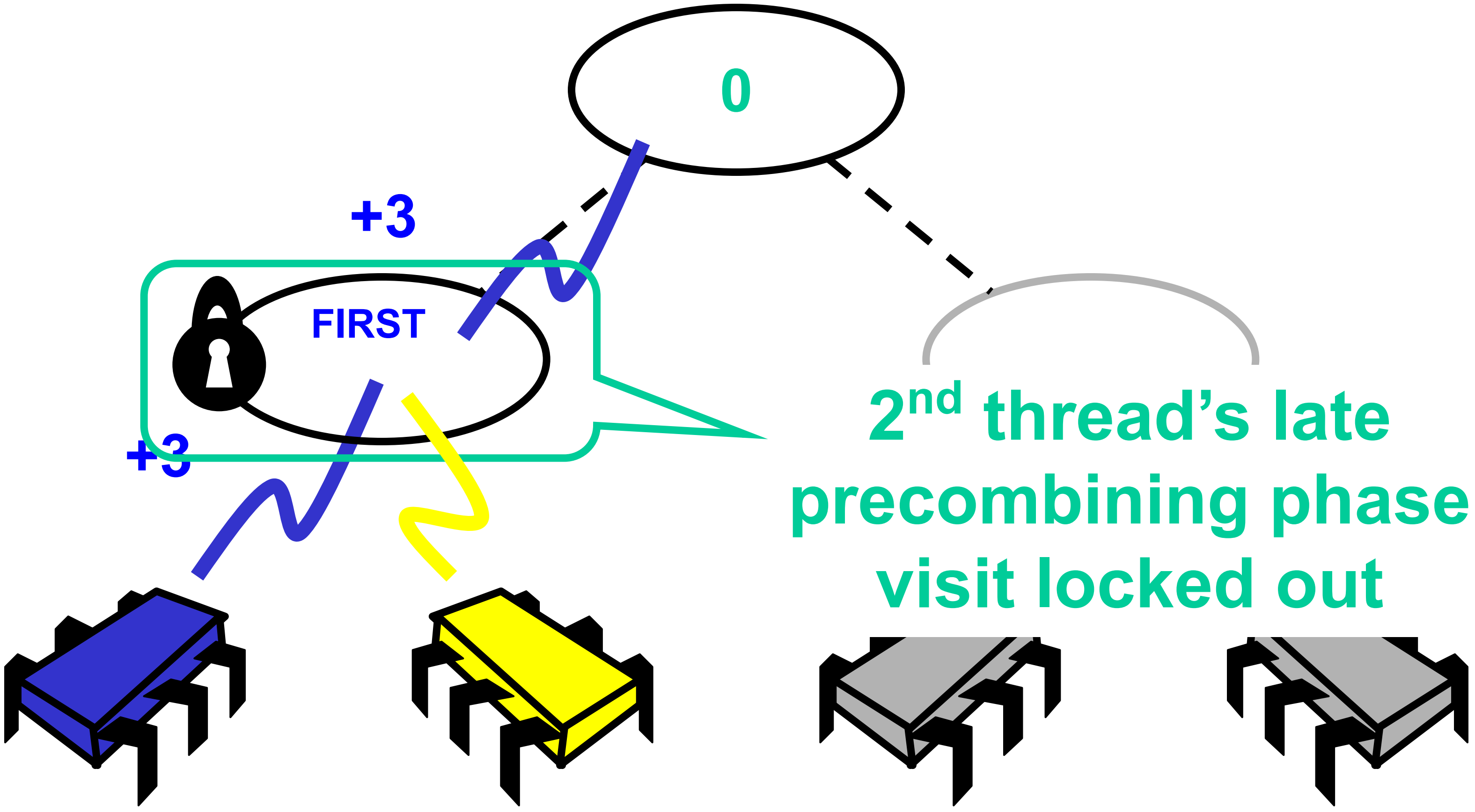


1st thread is alone, locks out late partner

Combining (reloaded)



Combining (reloaded)



2nd thread's late precombining phase visit locked out

Combining Navigation

```
node = myLeaf;
int combined = 1;
while (node != stop) {
  combined = node.combine(combined);
  stack.push(node);
  node = node.parent;
  }
```

Combining Navigation

```
node = myLeaf;
int combined = 1;
while (node != stop) {
  combined = node.combine(combined);
  stack.push(node);
  node = node.parent;
  }
```

Start at leaf

Combining Navigation

```
node = myLeaf;
int combined = 1;
while (node != stop) {
   combined = node.combine(combined);
   stack.push(node);
   node = node.parent;
   }
```

Add 1

Combining Navigation

```
node = myLeaf;
int combined = 1;
while (node != stop) {
   combined = node.combine(combined);
   stack.push(node);
   node = node.parent;
   }
```

Revisit nodes visited in precombining

Combining Navigation

```
node = myLeaf;
int combined = 1;
while (node != stop) {
  combined = node.combine(combined);
  stack.push(node);
  node = node.parent;
  }
```

Accumulate combined values, if any

Combining Navigation

```
node = myLeaf;
int combined = 1;
while (node != stop) {
  combined = node.combine(combined);
  stack.push(node);
  node = node.parent;
  }
```

We will retraverse path in reverse order …

Combining Navigation

```
node = myLeaf;
int combined = 1;
while (node != stop) {
   combined = node.combine(combined);
   stack.push(node);
   node = node.parent;
   }
```

Move up the tree

Combining Phase Node

```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default: …
    }
  }
```

Combining Phase Node

```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default: …
    }
  }
```

If node is locked by the 2nd thread, wait until it deposits its value

Combining Phase Node

```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default: …
    }
  }
```

How do we know that no thread acquires the lock between the two lines?

Because the methods are *synchronized*

Combining Phase Node

```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default: …
    }
  }
```

Lock out late attempts to combine (by threads still in precombining)

Combining Phase Node

```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default: …
    }
  }
```

Remember my (1st thread) contribution

Combining Phase Node

```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default: …
    }
  }
```

Check status

Combining Phase Node

```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default: …
    }
  }
```

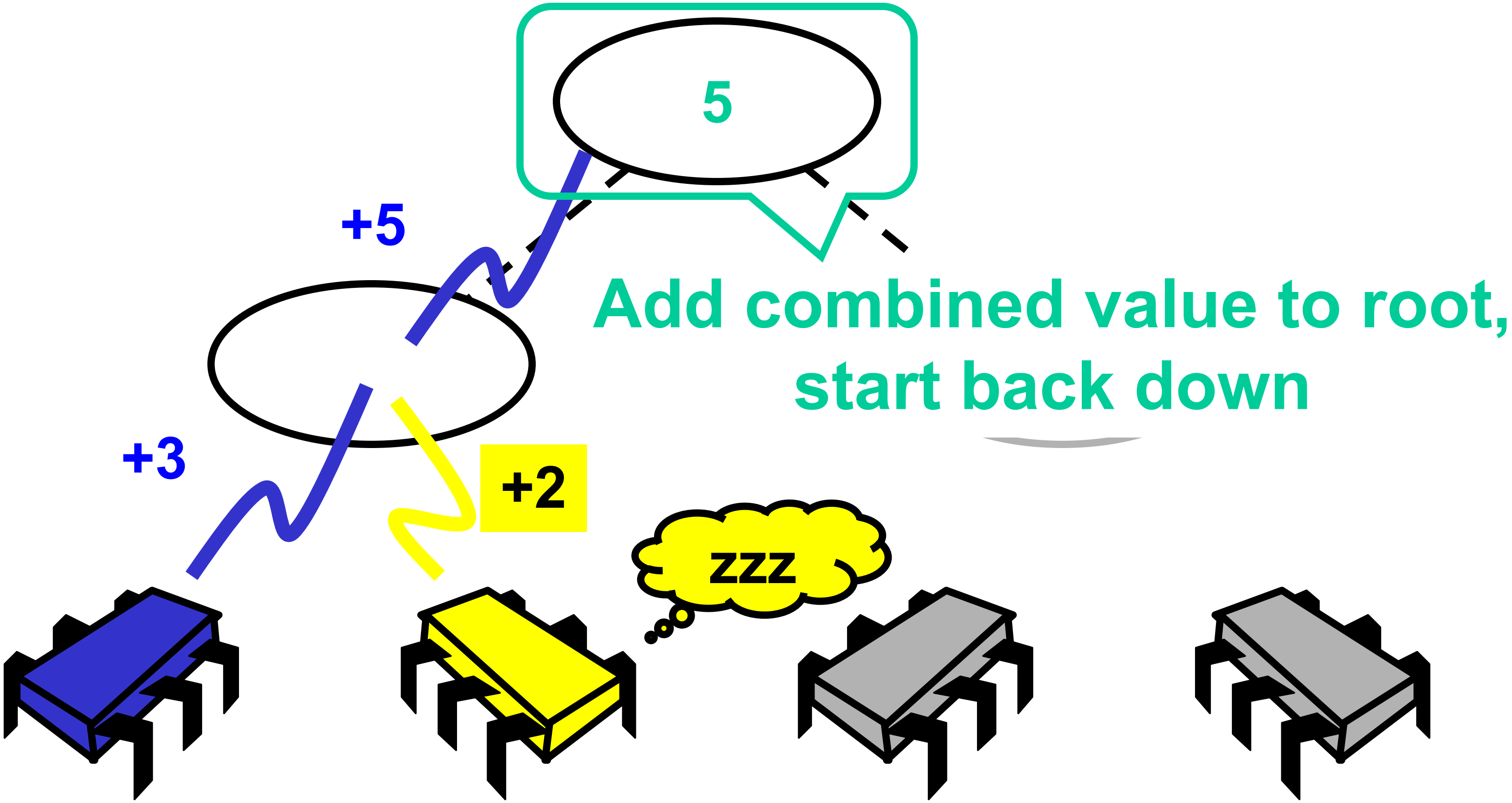
I (1st thread) am alone

Combining Node

```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default: …
    }
  }
```

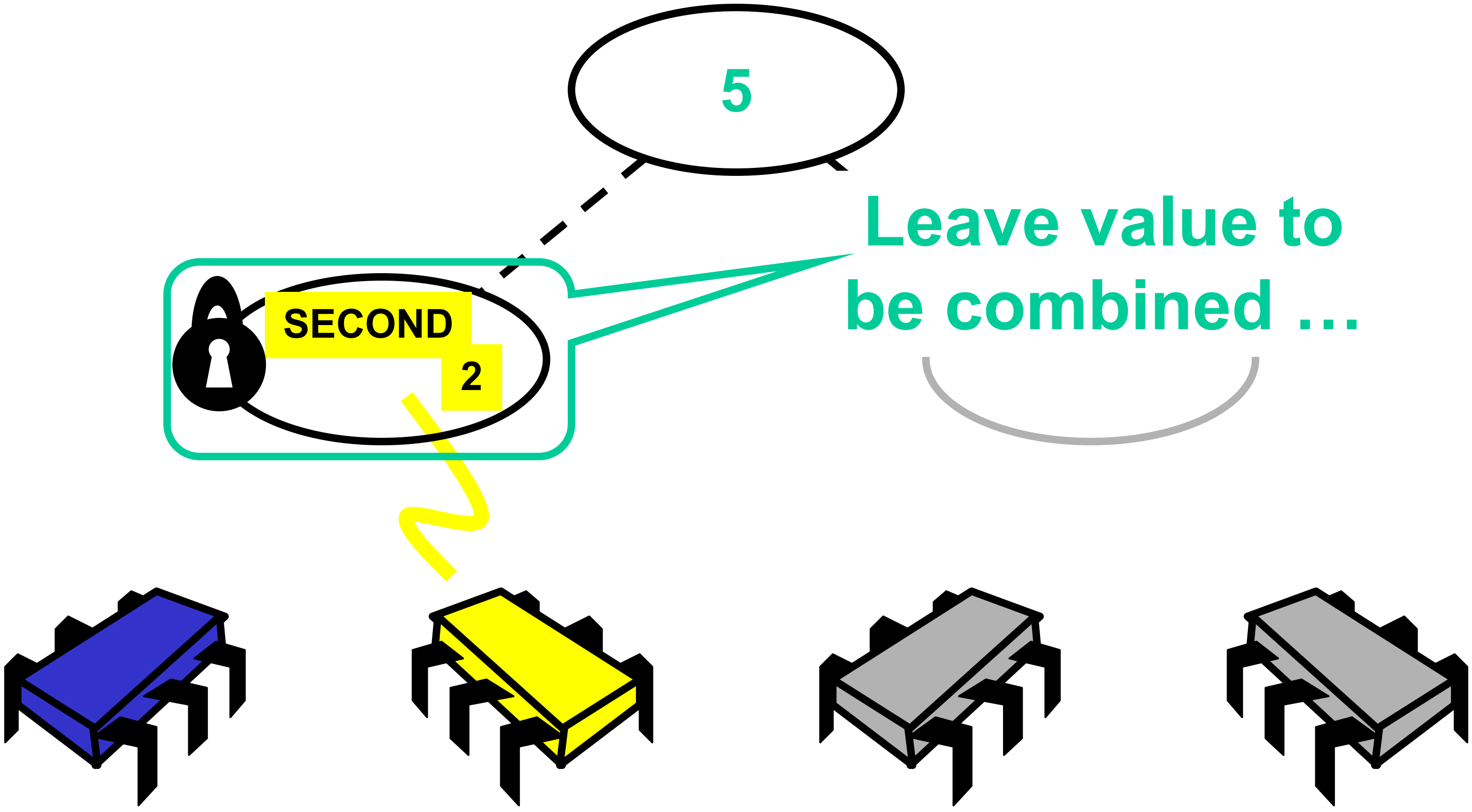
Not alone: combine with 2nd thread

Operation Phase



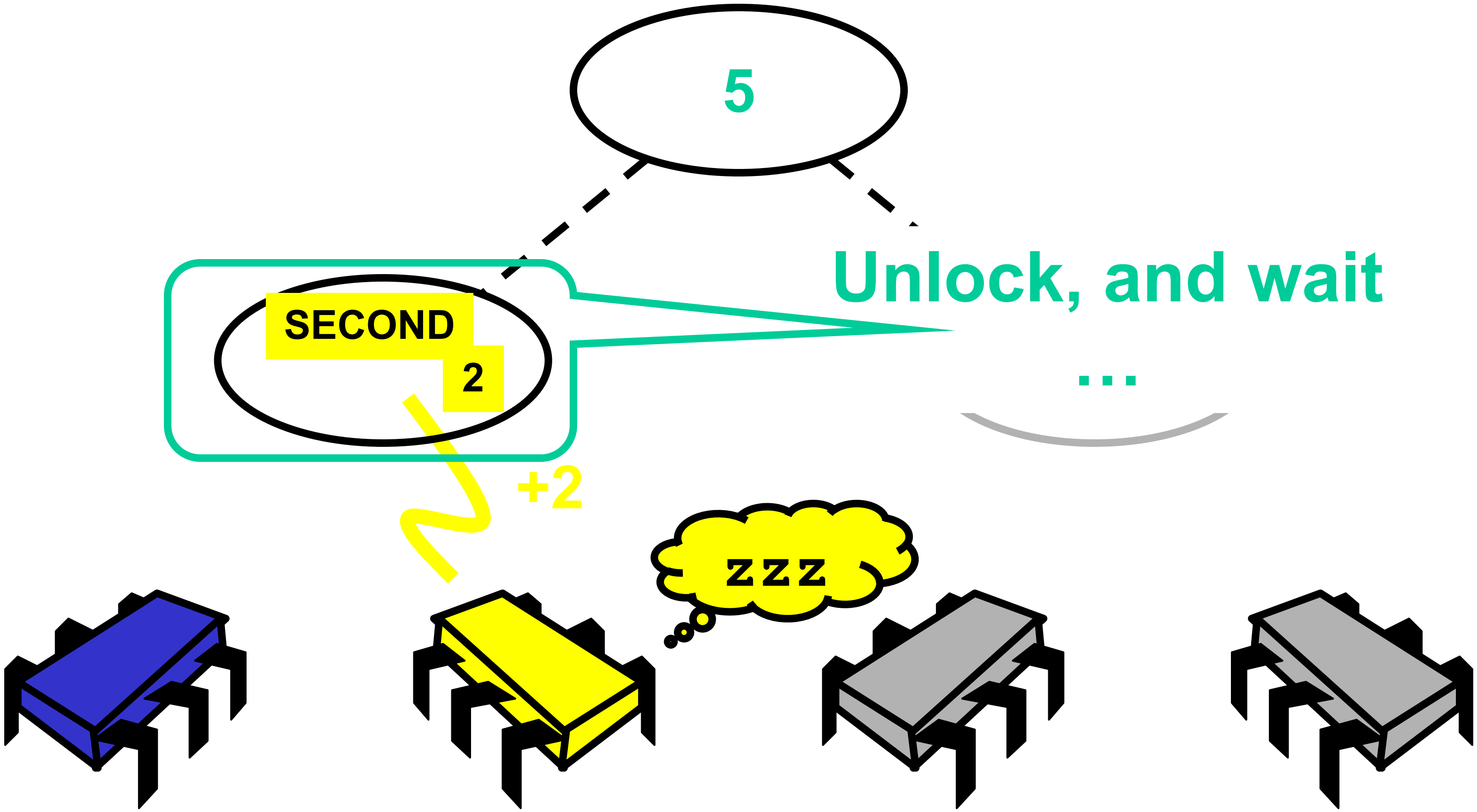
Add combined value to root, start back down

Operation Phase (reloaded)



Leave value to be combined …

Operation Phase (reloaded)



Unlock, and wait

…

Operation Phase Navigation

```
prior = stop.op(combined);
```

Operation Phase Navigation

```
prior = stop.op(combined);
```

The node where we stopped. Provide collected sum and wait for combining result

Operation on Stopped Node

```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int prior = result;
      result += combined;
      return prior;
    case SECOND: secondValue = combined;
      locked = false; notifyAll();
      while (cStatus != CStatus.RESULT) wait();
      locked = false; notifyAll();
      cStatus = CStatus.IDLE;
      return result;
    default: …
```

Op States of Stop Node

```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int prior = result;
      result += combined;
      return prior;
    case SECOND: secondValue = combined;
      locked = false; notifyAll();
      while (cStatus != CStatus.RESULT) wait();
      locked = false; notifyAll();
      cStatus = CStatus.IDLE;
      return result;
    default: …
```

Only ROOT and SECOND possible. Why?

At Root

```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int prior = result;
      result += combined;
      return prior;
    case SECOND: secondValue = combined;
      locked = false; notifyAll();
      while (cStatus != CStatus.RESULT) wait();
      locked = false; notifyAll();
      cStatus = CStatus.IDLE;
      return result;
    default: …
```

Add sum to root, return prior value

Intermediate Node

```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int prior = result;
      result += combined;
      return prior;
    case SECOND: secondValue = combined;
      locked = false; notifyAll();
      while (cStatus != CStatus.RESULT) wait();
      locked = false; notifyAll();
      cStatus = CStatus.IDLE;
      return result;
    default: …
```

Deposit value for later combining …

Intermediate Node

```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int prior = result;
      result += combined;
      return prior;
    case SECOND: secondValue = combined;
      locked = false; notifyAll();
      while (cStatus != CStatus.RESULT) wait();
      locked = false; notifyAll();
      cStatus = CStatus.IDLE;
      return result;
    default: …
```

Unlock node (locked in precombining), then notify 1st thread

Intermediate Node

```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int pric
      result += combine
      return prior;
    case SECOND: secondValue = combined;
      locked = false; notifyAll();
      while (cStatus != CStatus.RESULT) wait();
      locked = false; notifyAll();
      cStatus = CStatus.IDLE;
      return result;
    default: …
```

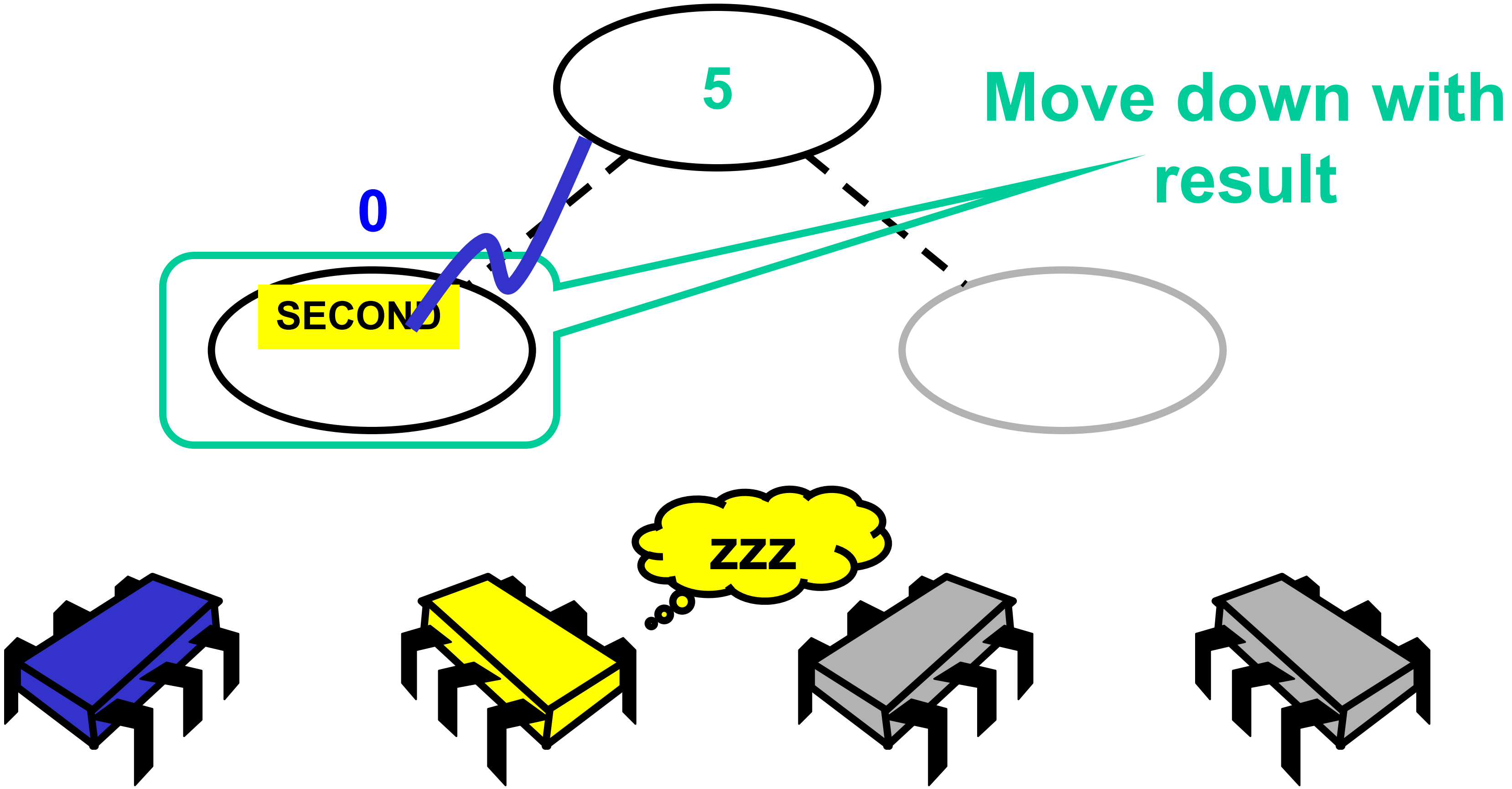
Wait for 1st thread to deliver results

Intermediate Node

```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int prior = result;
      result += combined;
      return prior;
    case SECOND: secondValue = combined;
      locked = false; notifyAll();
      while (cStatus != CStatus.RESULT) wait();
      locked = false; notifyAll();
      cStatus = CStatus.IDLE;
      return result;
    default: …
```

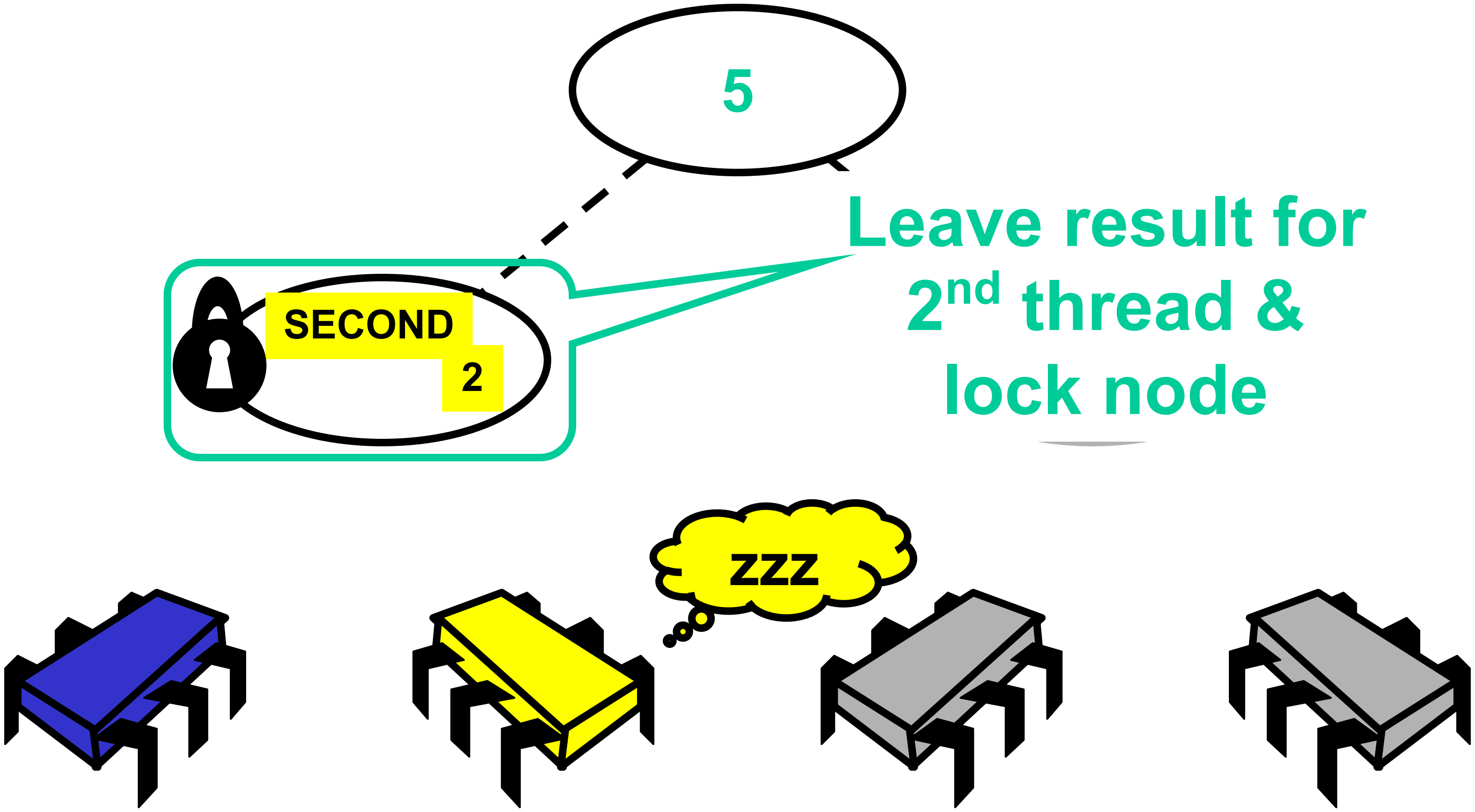
Unlock node (locked by 1st thread in combining phase) & return

Distribution Phase



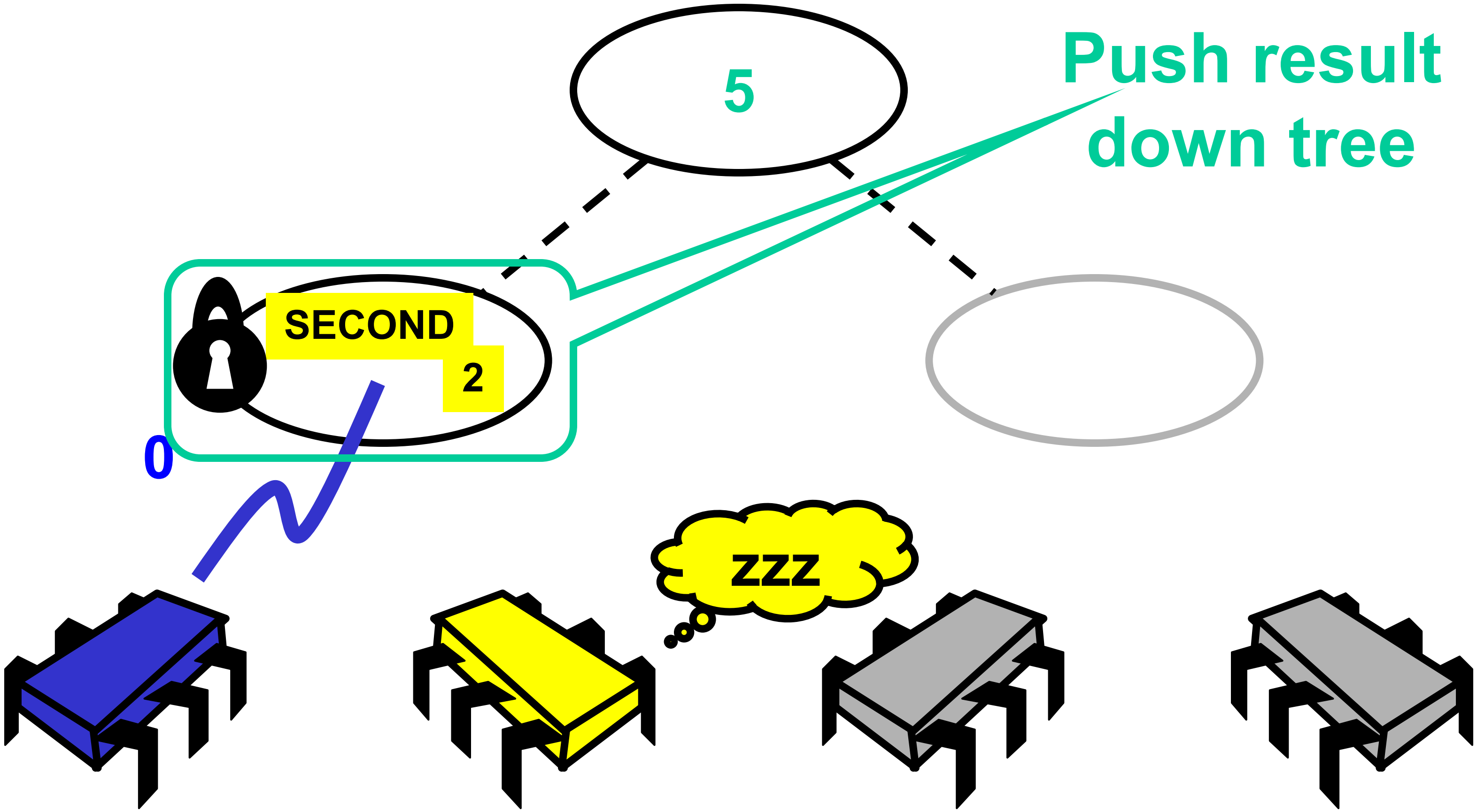
Move down with result

Distribution Phase



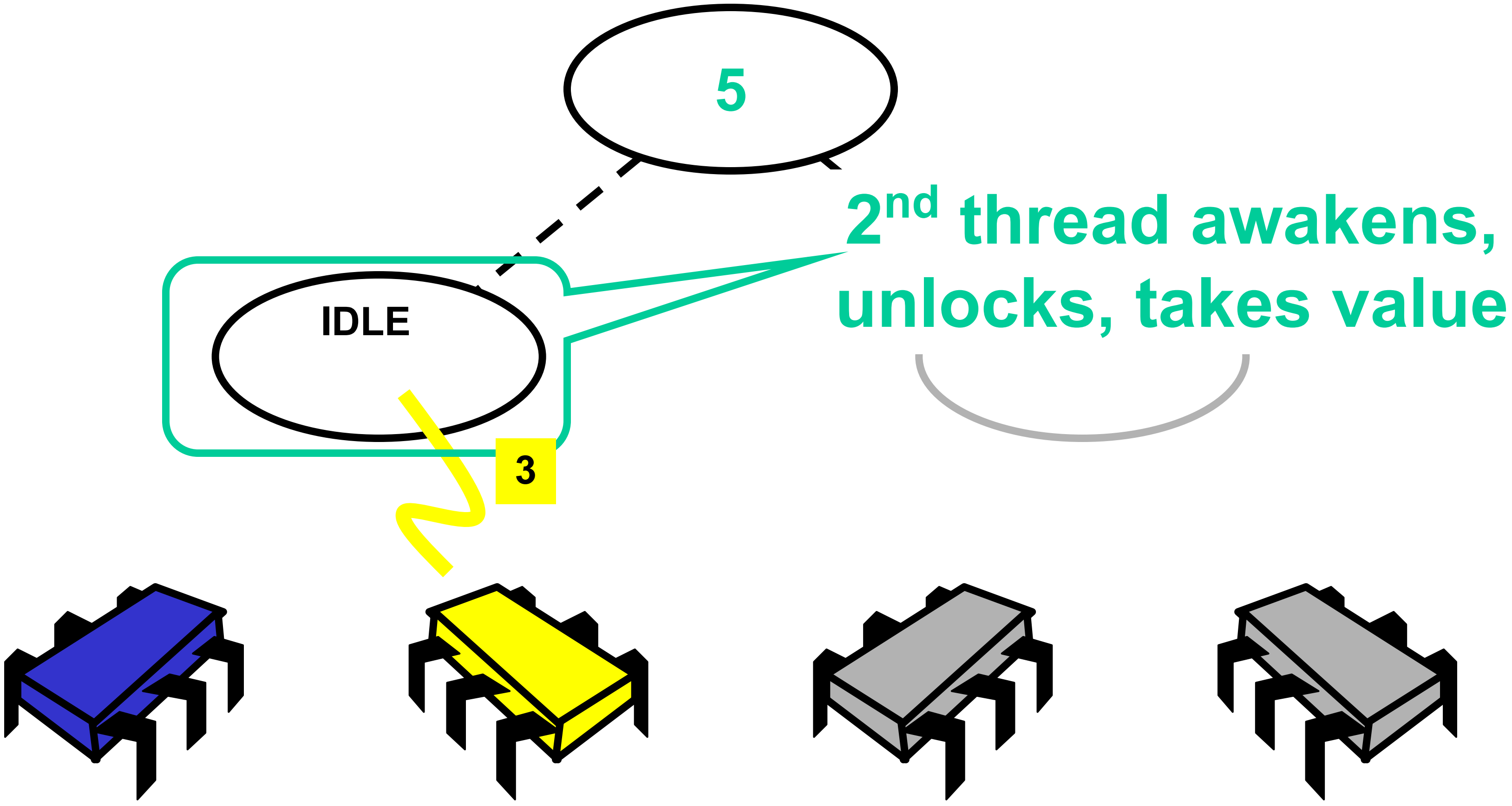
Leave result for 2nd thread & lock node

Distribution Phase



Push result down tree

Distribution Phase



2nd thread awakens, unlocks, takes value

Distribution Phase Navigation

```
while (!stack.empty()) {
  node = stack.pop();
  node.distribute(prior);
  }
return prior;
```

Distribution Phase Navigation

```
while (!stack.empty()) {
  node = stack.pop();
  node.distribute(prior);
  }
return prior;
```

Traverse path in reverse order

Distribution Phase Navigation

```
while (!stack.empty()) {
  node = stack.pop();
  node.distribute(prior);
  }
return prior;
```

Distribute results to waiting 2nd threads

Distribution Phase Navigation

```
while (!stack.empty()) {
  node = stack.pop();
  node.distribute(prior);
  }
return prior;
```

Return result to caller

Distribution Phase

```
synchronized void distribute(int prior) {
   switch (cStatus) {
     case FIRST:
       cStatus = CStatus.IDLE;
       locked = false; notifyAll();
       return;
     case SECOND:
       result = prior + firstValue;
       cStatus = CStatus.RESULT; notifyAll();
       return;
     default: …
```

Distribution Phase

```
synchronized void distribute(int prior) {
   switch (cStatus) {
     case FIRST:
       cStatus = CStatus.IDLE;
       locked = false; notifyAll();
       return;
     case SECOND:
       result = prior + firstValue;
       cStatu
       return
     default: …
```

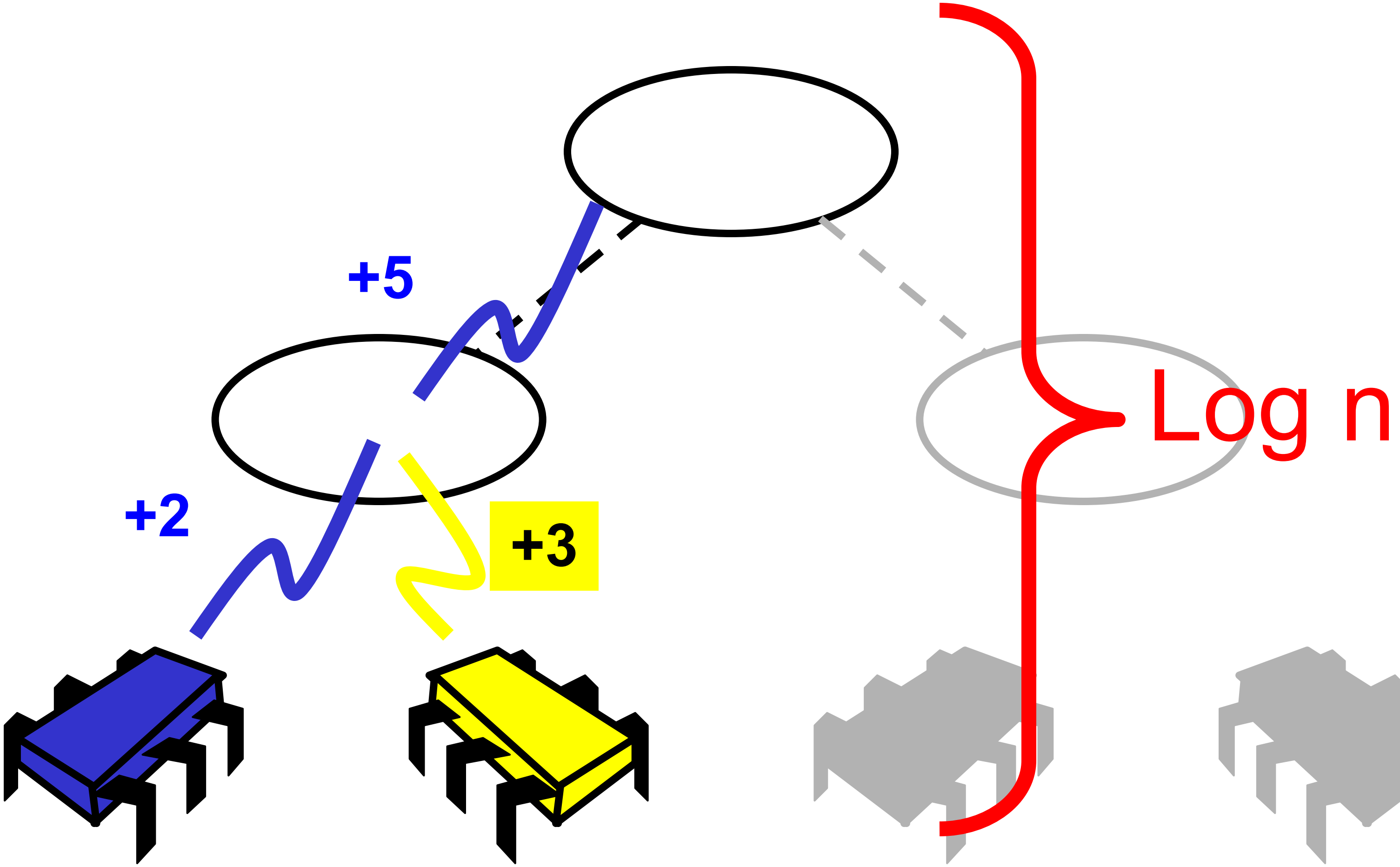
No 2nd thread to combine with me, unlock node & reset

Distribution Phase

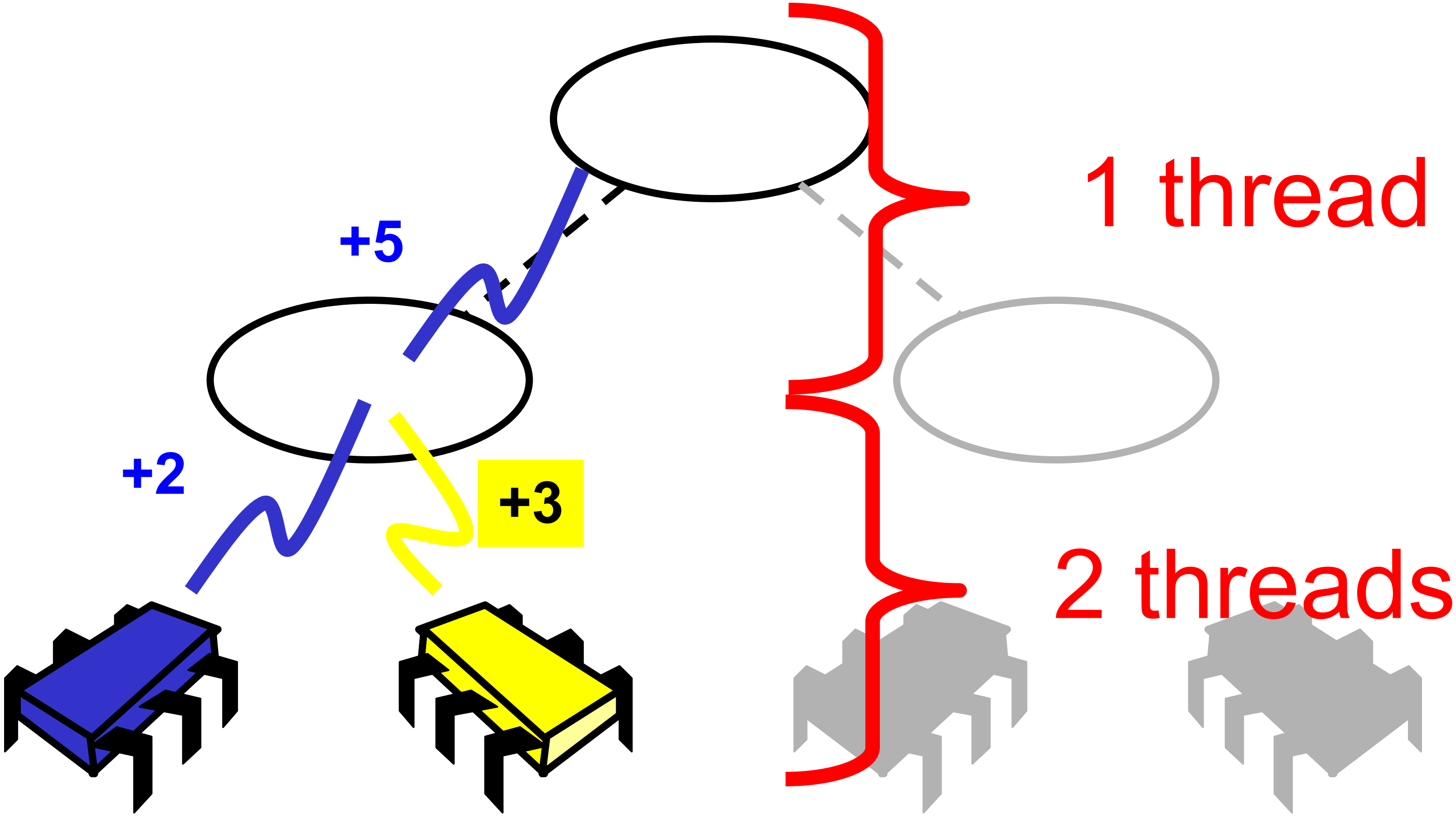
Notify 2nd thread that result is available (2nd thread will release lock)

```
case FIRST:
  cStatus = CStatus.IDLE;
  locked = false; notifyAll();
  return;
case SECOND:
  result = prior + firstValue;
  cStatus = CStatus.RESULT; notifyAll();
  return;
default: …
```

Bad News: High Latency



Good News: Real Parallelism



Throughput Puzzles

- Ideal circumstances
  - All n threads move together, combine
  - n increments in O(log n) time
- Worst circumstances
  - All n threads slightly skewed, locked out
  - n increments in O(n · log n) time

Index Distribution Benchmark

```
void indexBench(int iters, int work) {
 while (int i < iters) {
  i = r.getAndIncrement();
  Thread.sleep(random() % work);
 }}
```

Index Distribution Benchmark

```
void indexBench(int iters, int work) {
 while (int i < iters) {
  i = r.getAndIncrement();
  Thread.sleep(random() % work);
 }}
```

How many iterations

Index Distribution Benchmark

```
void indexBench(int iters, int work) {
 while (int i < iters) {
  i = r.getAndIncrement();
  Thread.sleep(random() % work);
 }}
```

Expected time between incrementing counter

Index Distribution Benchmark

```
void indexBench(int iters, int work) {
 while (int i < iters) {
  i = r.getAndIncrement();
  Thread.sleep(random() % work);
 }}
```

Take a number

Index Distribution Benchmark

```
void indexBench(int iters, int work) {
 while (int i < iters) {
  i = r.getAndIncrement();
  Thread.sleep(random() % work);
 }}
```

Pretend to work
(more work, less concurrency)

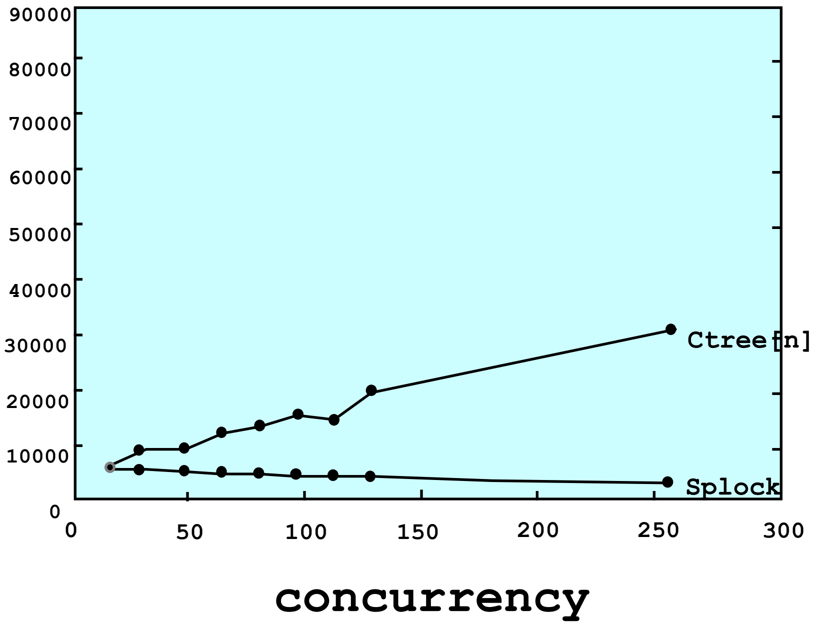
Performance

- Here are some graphs
- Throughput
  - Average increments in 1 million cycles
- Latency
  - Average cycles per inc

Performance (Simulated)

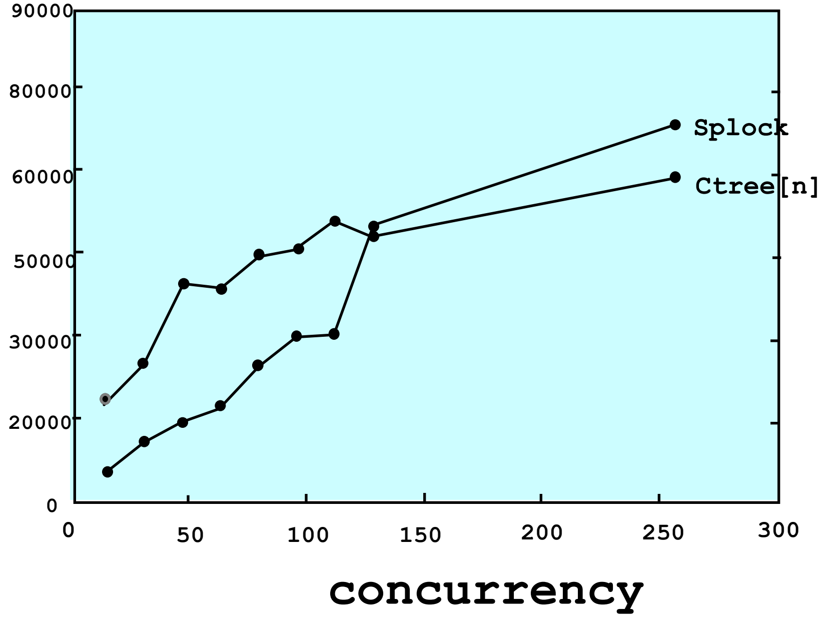
operations
per million
cycles

Throughput:



cycles
per
operation

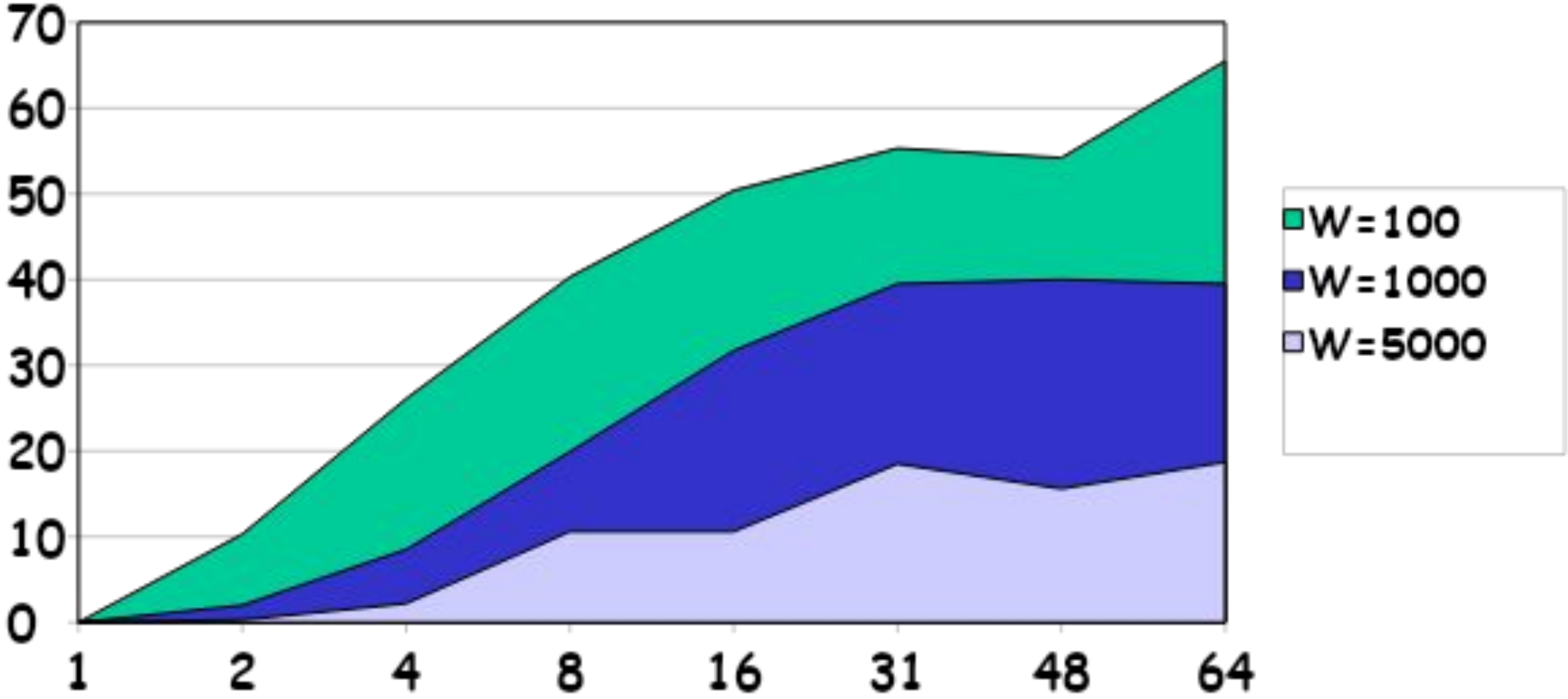
Latency:



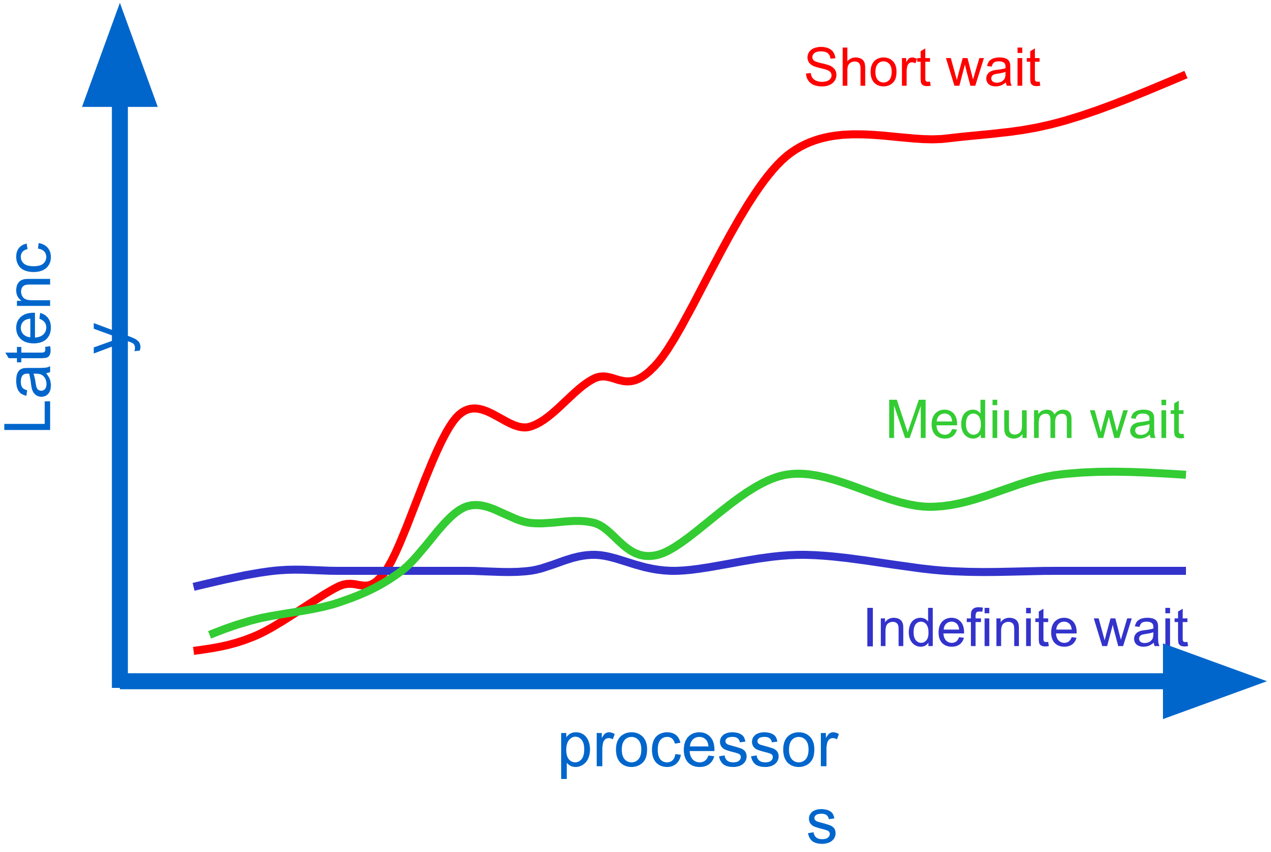
Load Fluctuations

- Combining is sensitive:
  - if arrival rates drop …
  - So do combining rates …
  - & performance deteriorates!
- Test
  - Vary "work"
  - Duration between accessess …

Combining Rate vs Work



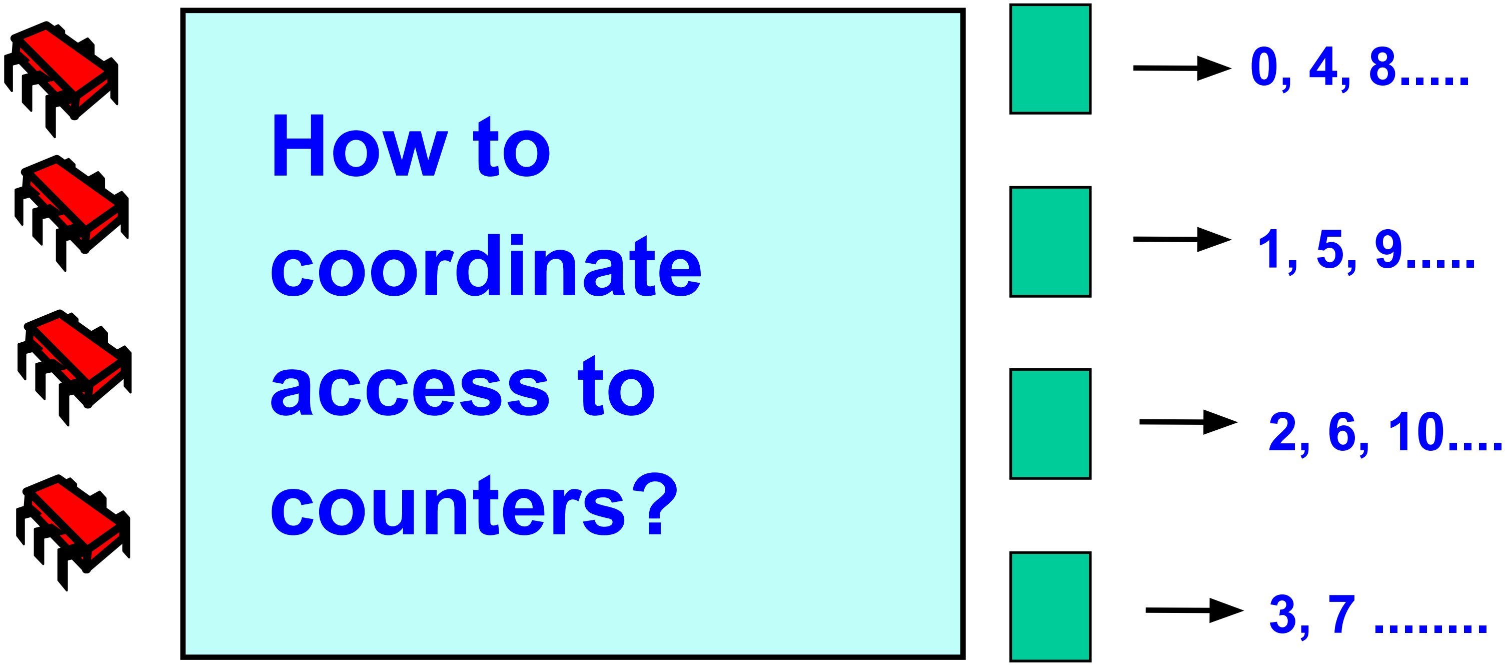
Better to Wait Longer



Conclusions

- Combining Trees
  - Linearizable Counters
  - Work well under high contention
  - Sensitive to load fluctuations
  - Can be used for getAndMumble() ops

Parallel Counter Approach



How to coordinate access to counters?

0, 4, 8.....

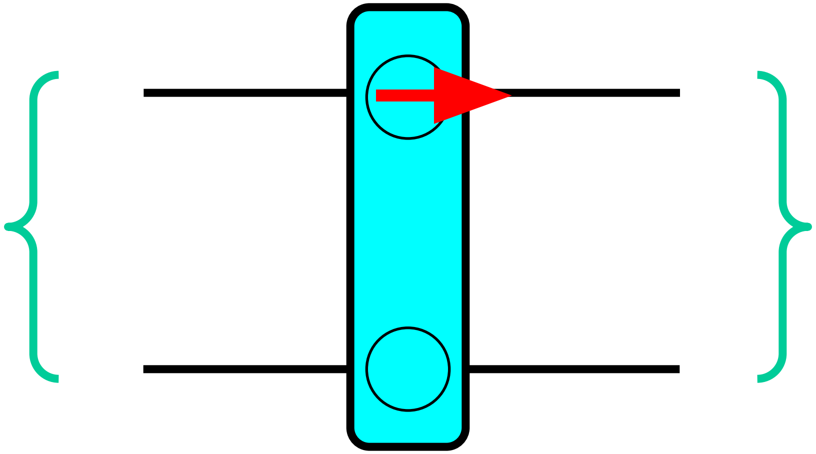
1, 5, 9.....

2, 6, 10....

3, 7 ....

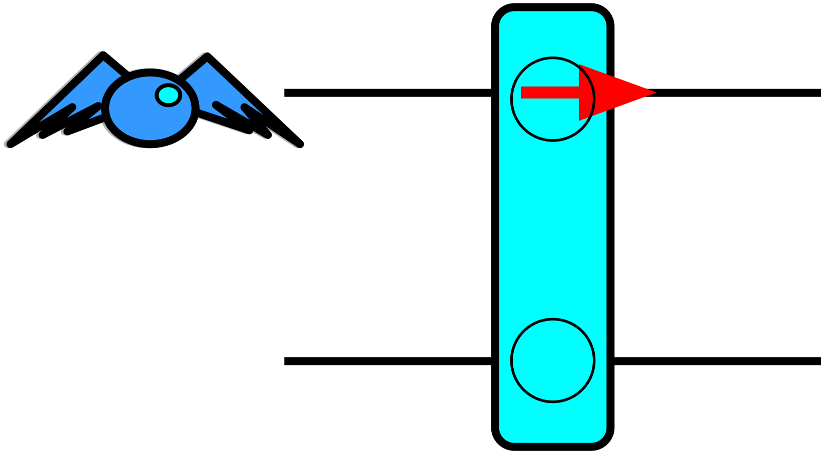
A Balancer

Input wires



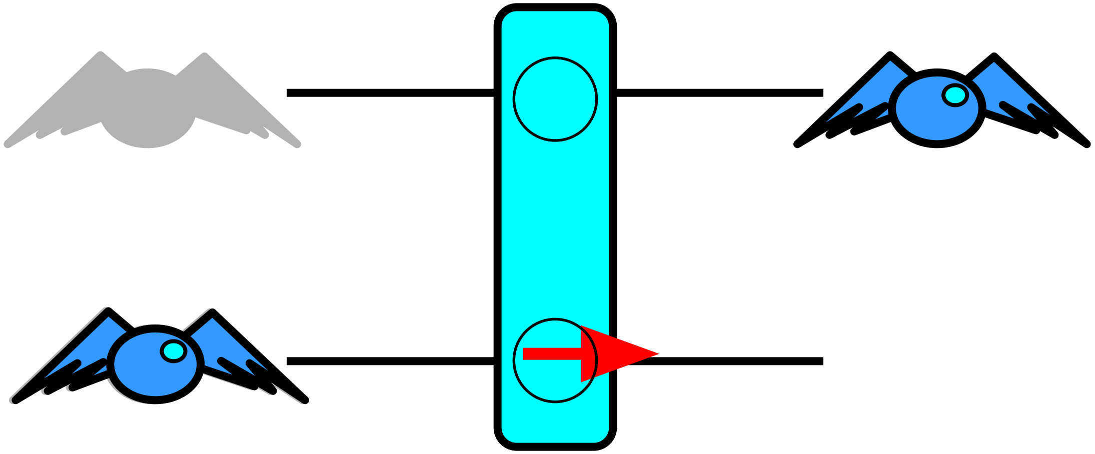
Output wires

Tokens Traverse Balancers

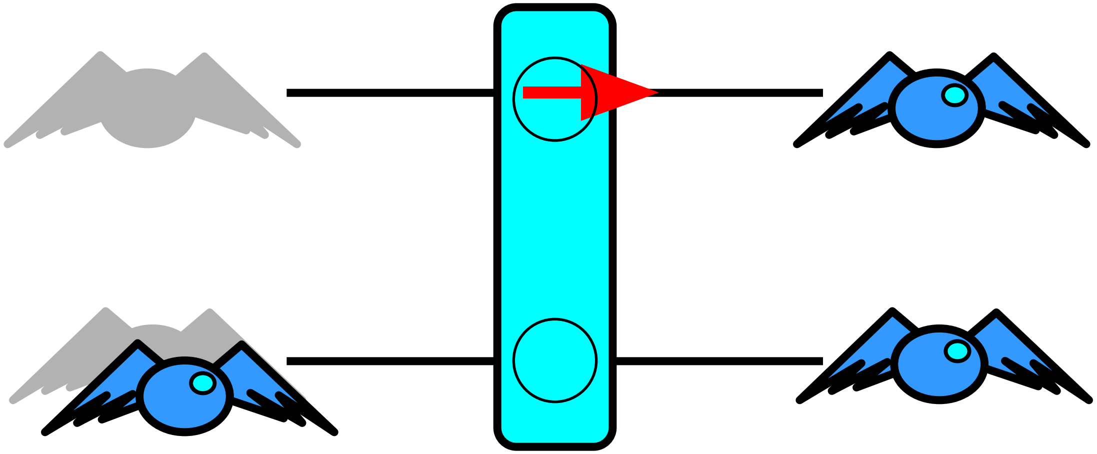


- Token i enters on any wire
- leaves on wire i (mod 2)

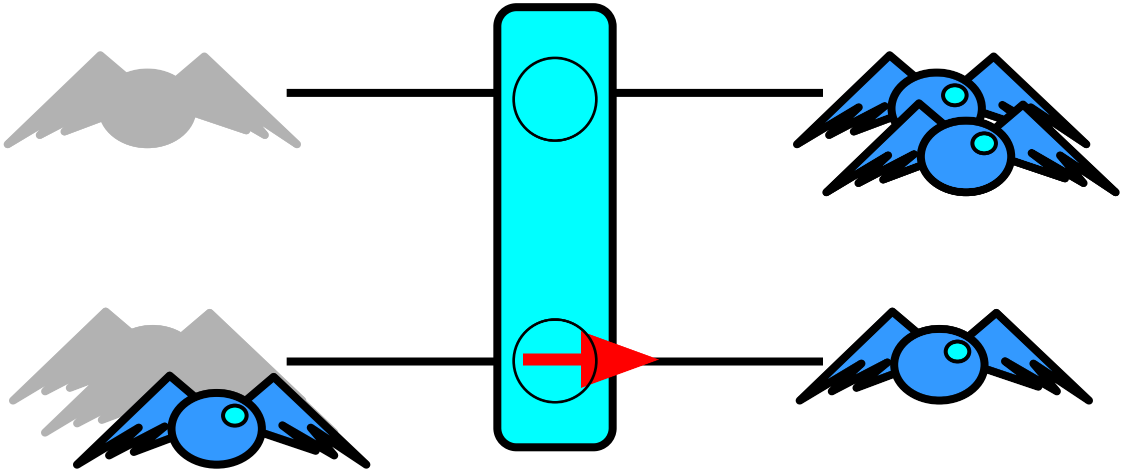
Tokens Traverse Balancers



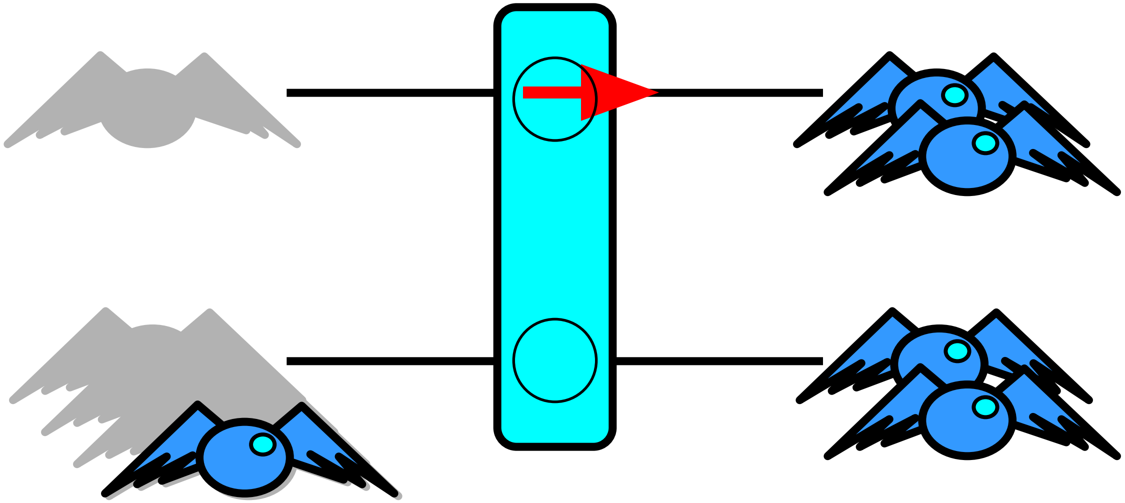
Tokens Traverse Balancers



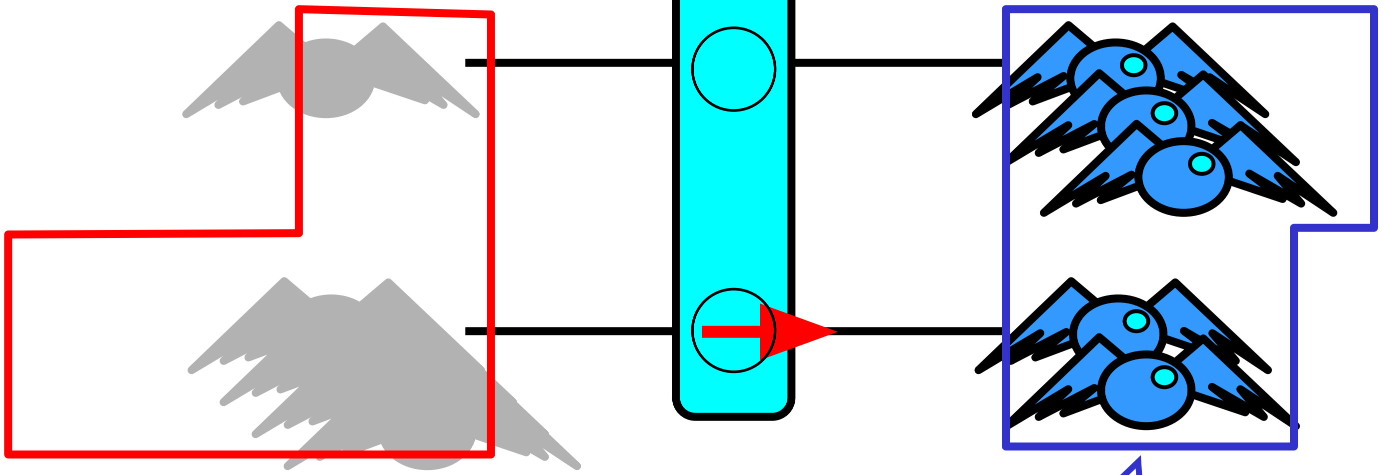
Tokens Traverse Balancers



Tokens Traverse Balancers



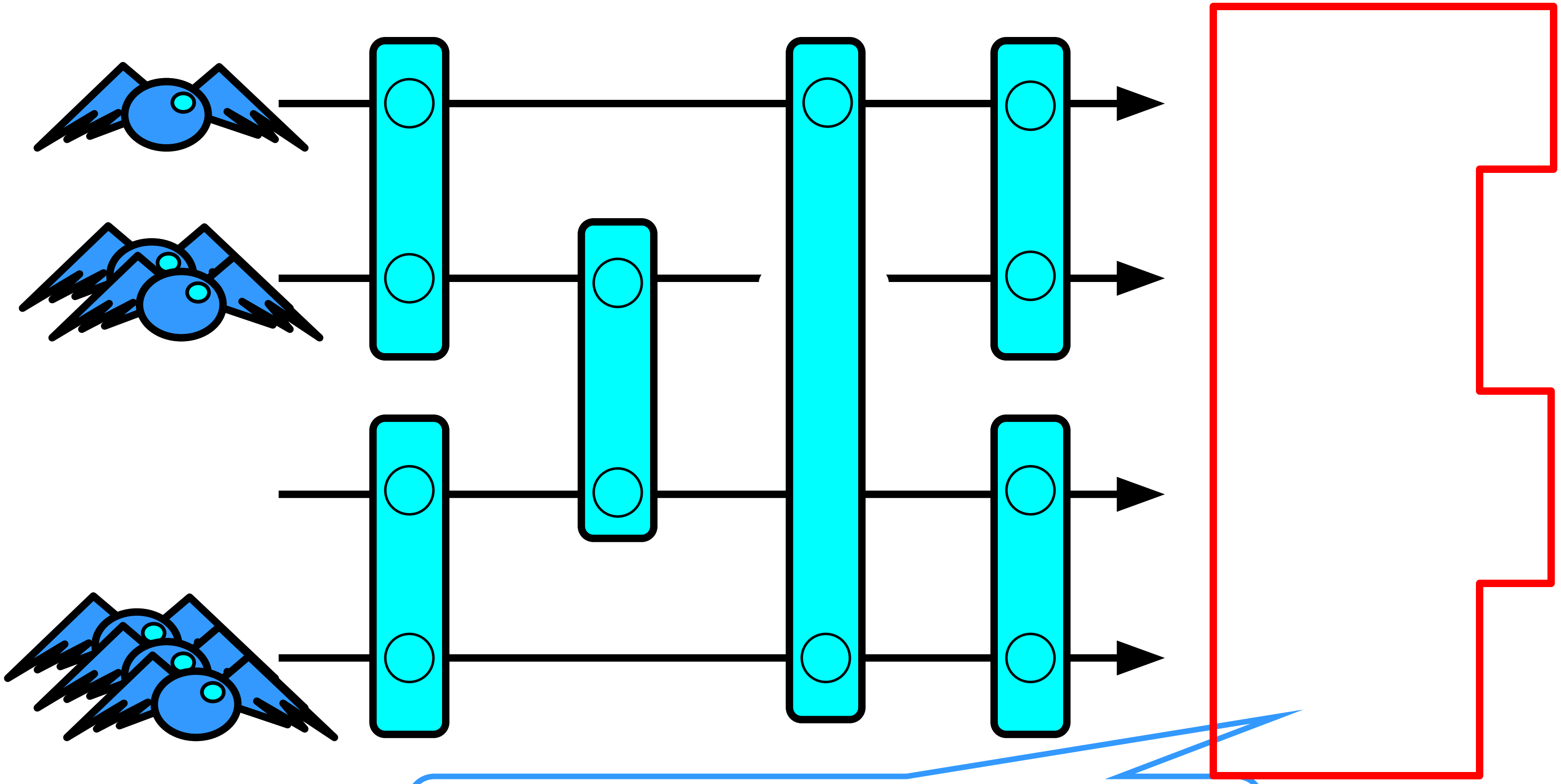
Quiescent State: all tokens have exited



Arbitrary input distribution

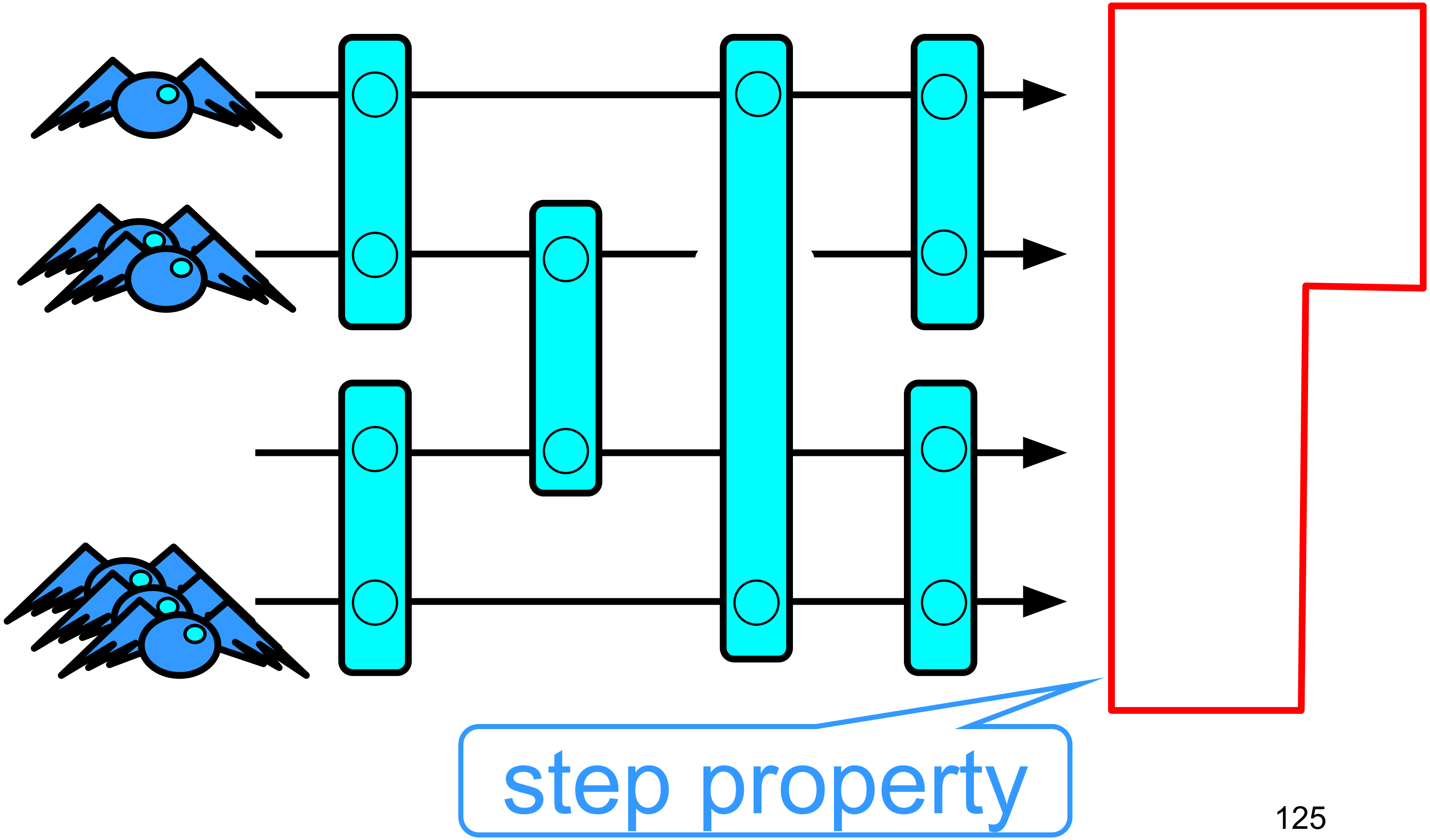
Balanced output distribution

Smoothing Network



1-smooth property

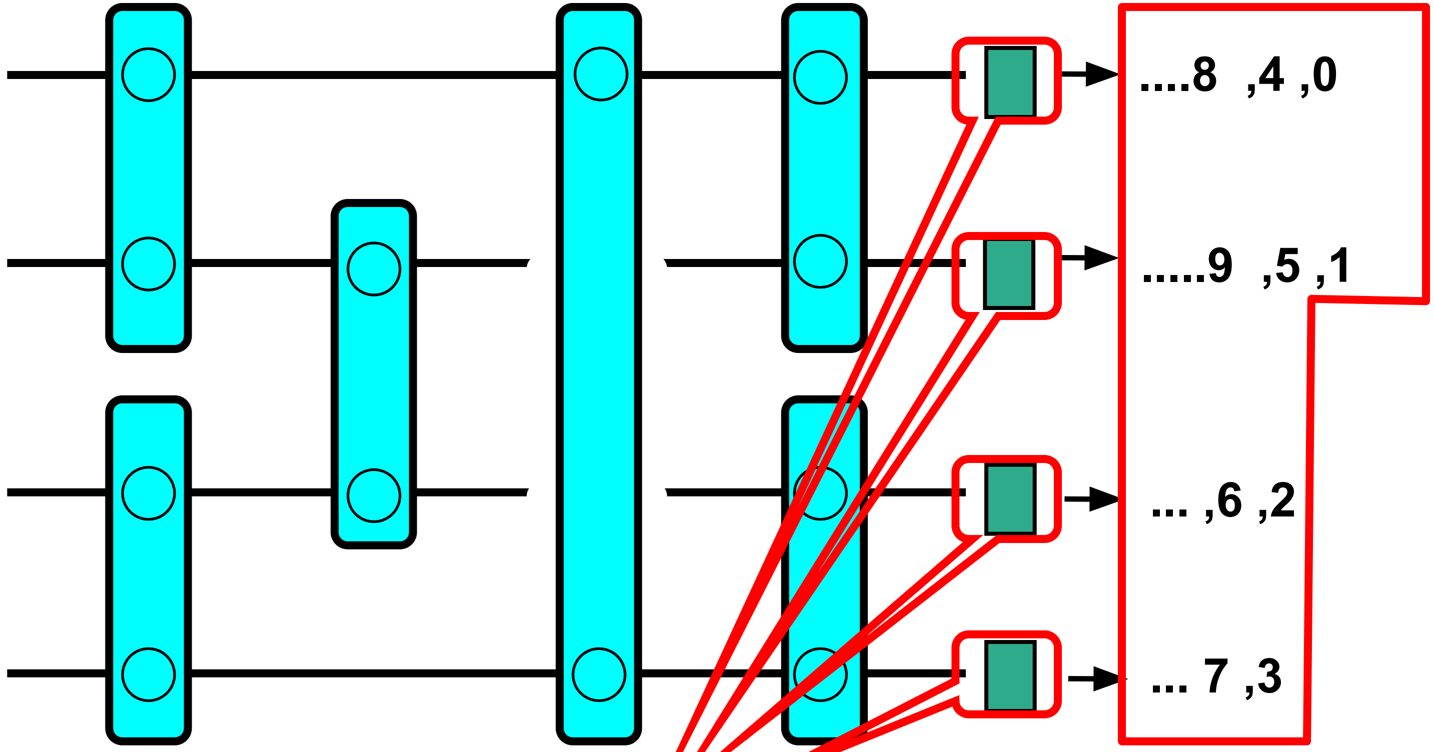
Counting Network



step property

Counting

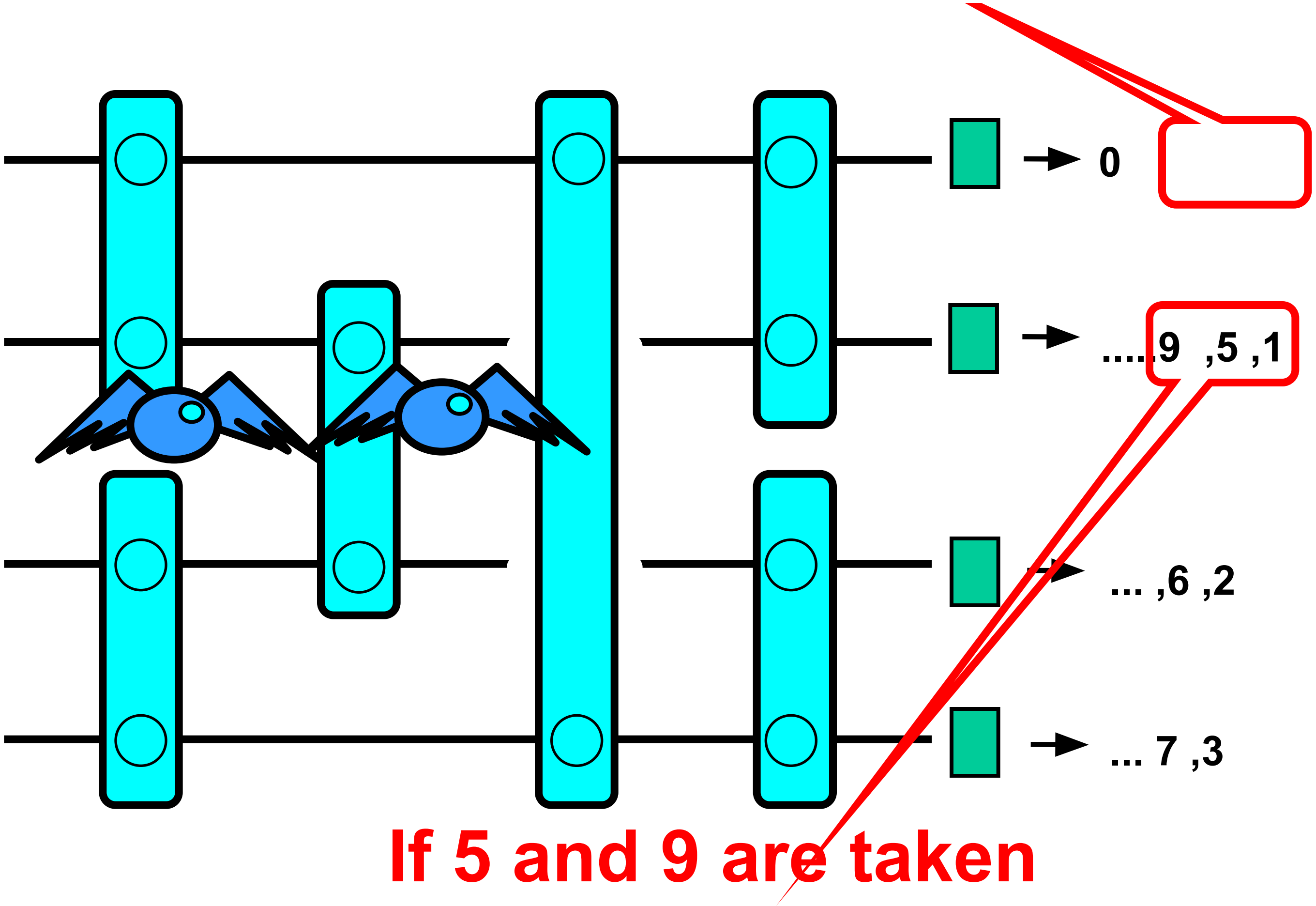
Step property guarantees no duplication or omissions, how?



counters

Multiple counters distribute load

Step property guarantees that in-flight tokens will take missing values

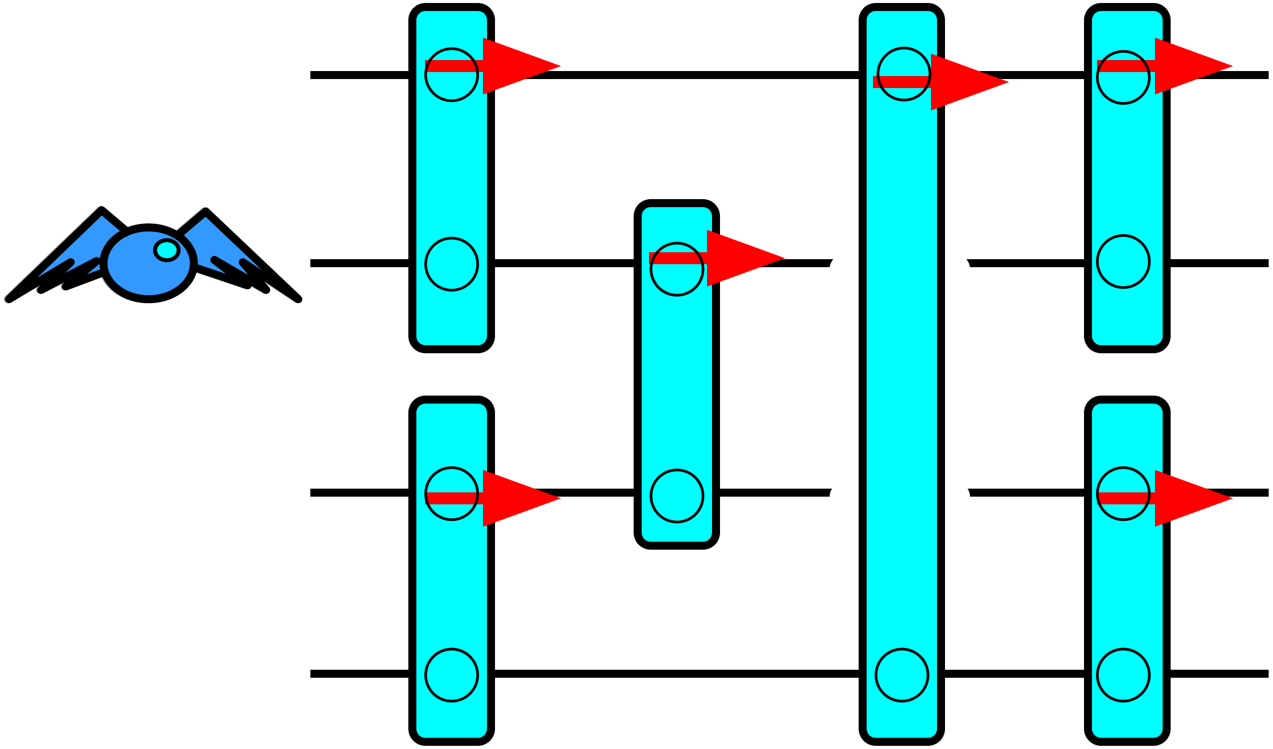


If 5 and 9 are taken before 4 and 8

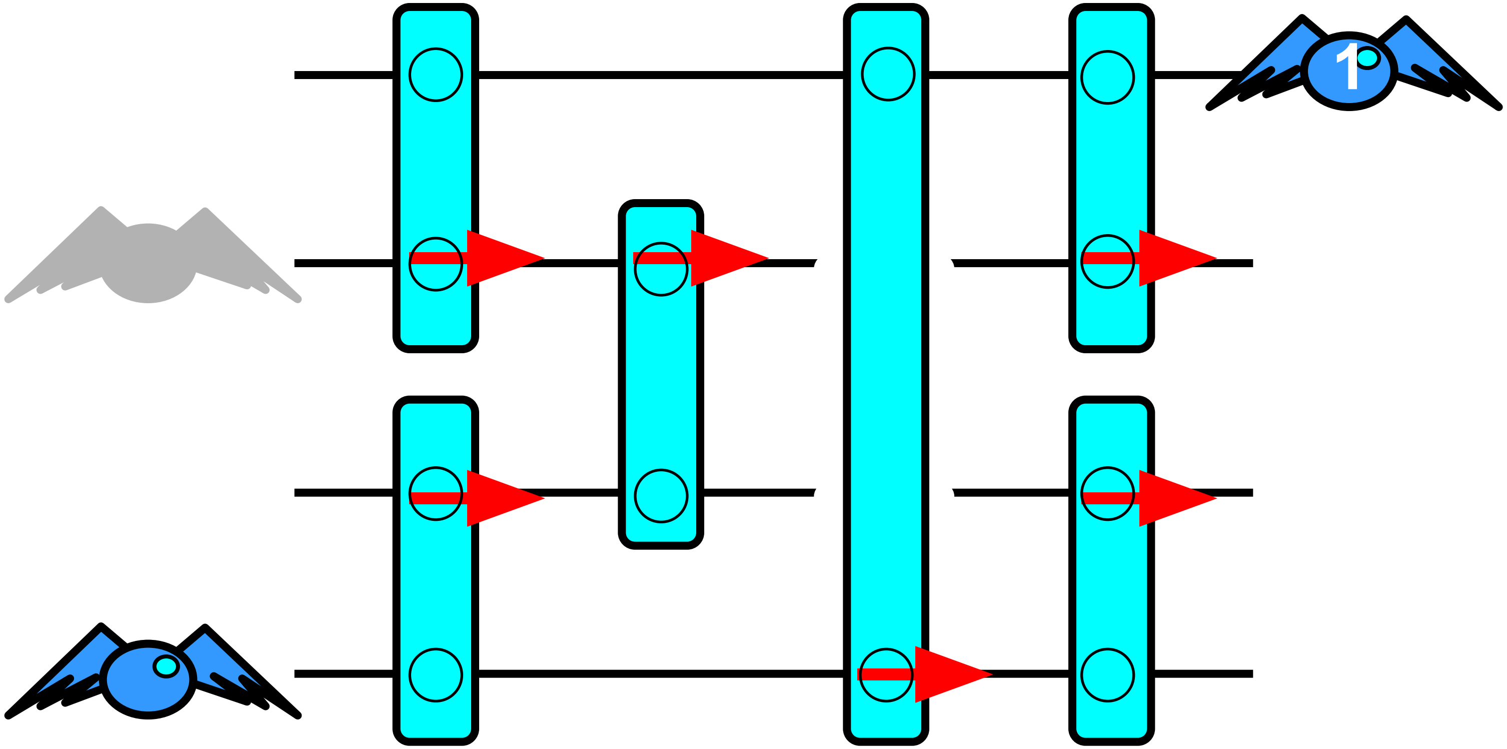
Counting Networks

- Good for counting number of tokens
- low contention
- no sequential bottleneck
- high throughput
- practical networks depth $\log^2 \boldsymbol{n}$

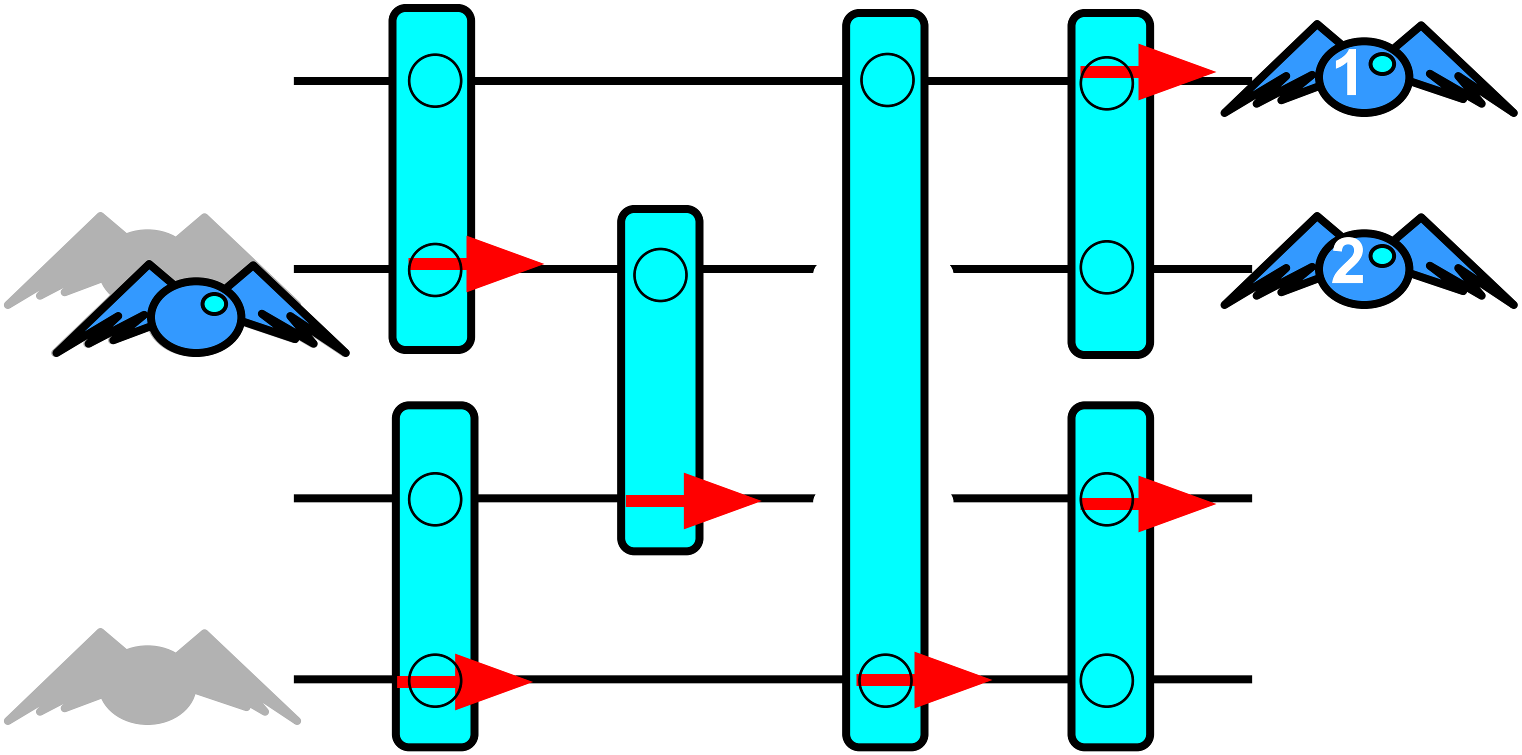
Counting Network



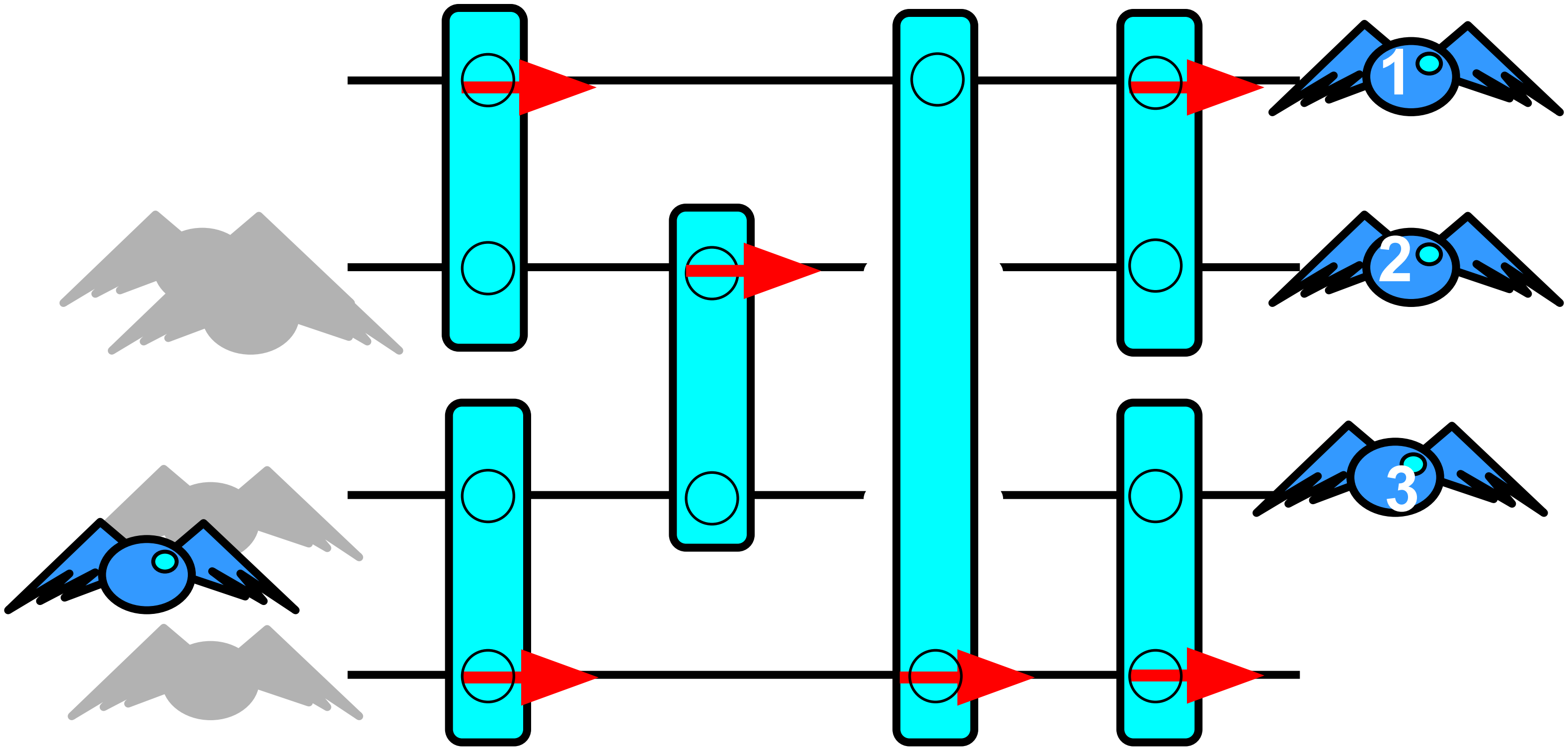
Counting Network



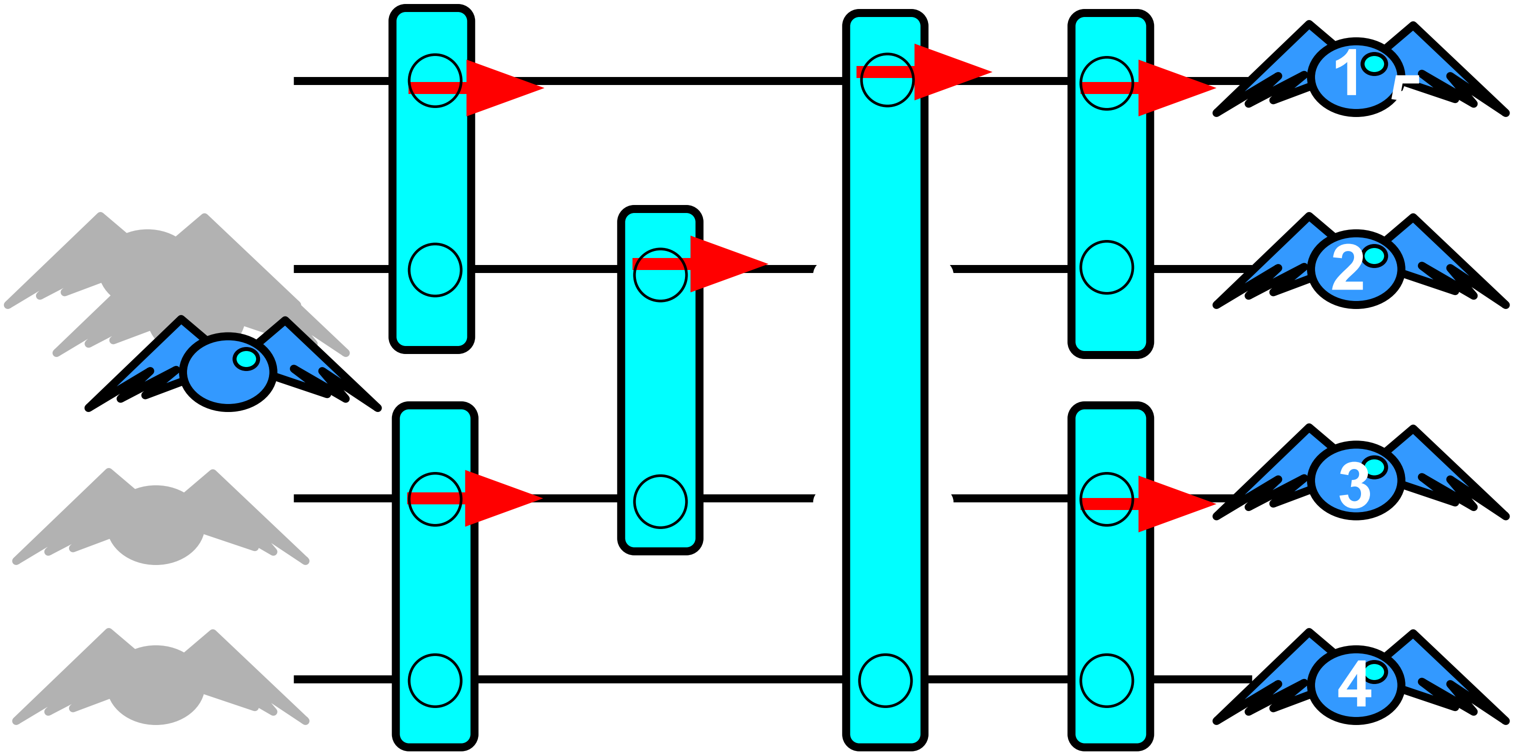
Counting Network



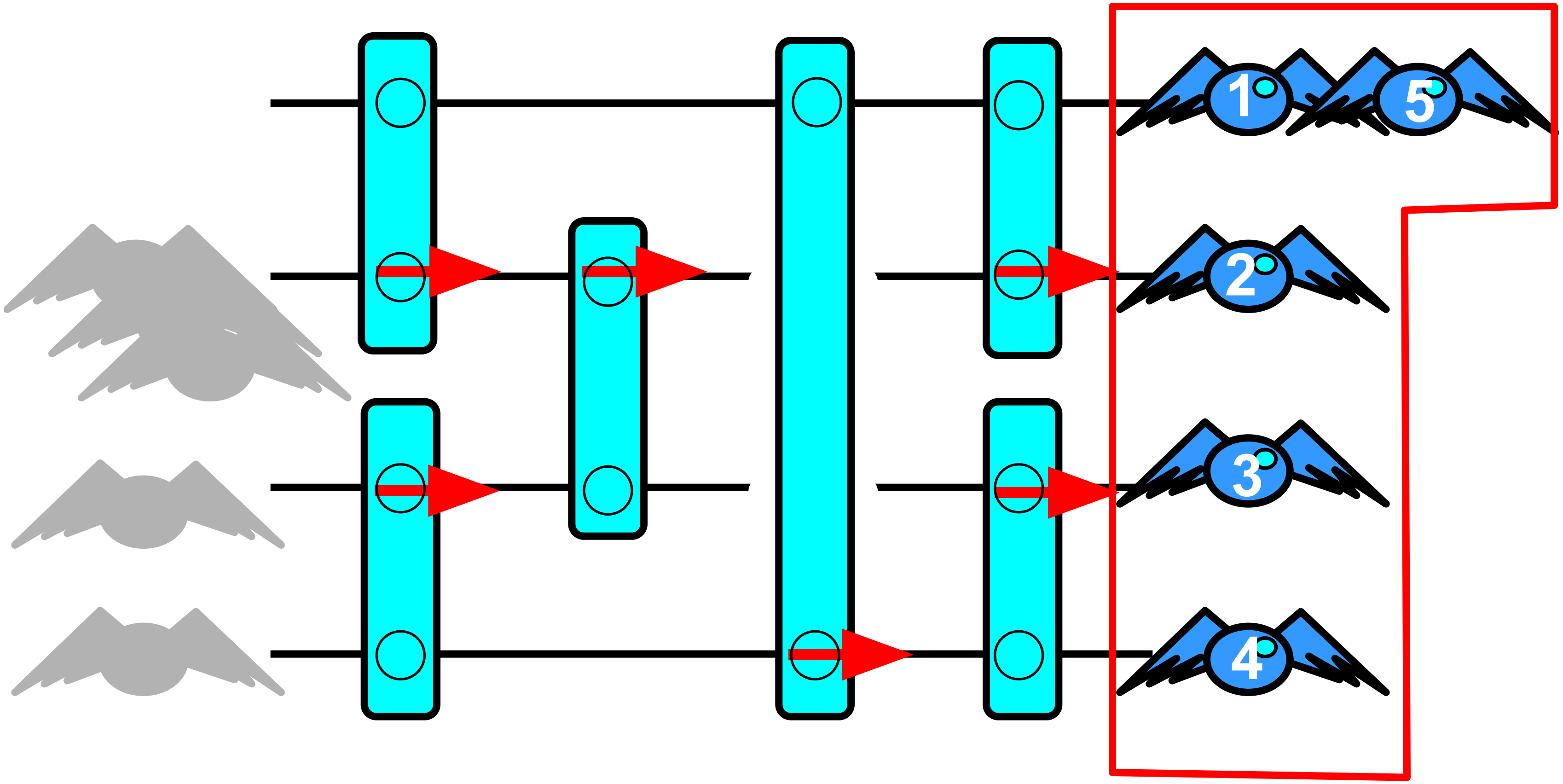
Counting Network



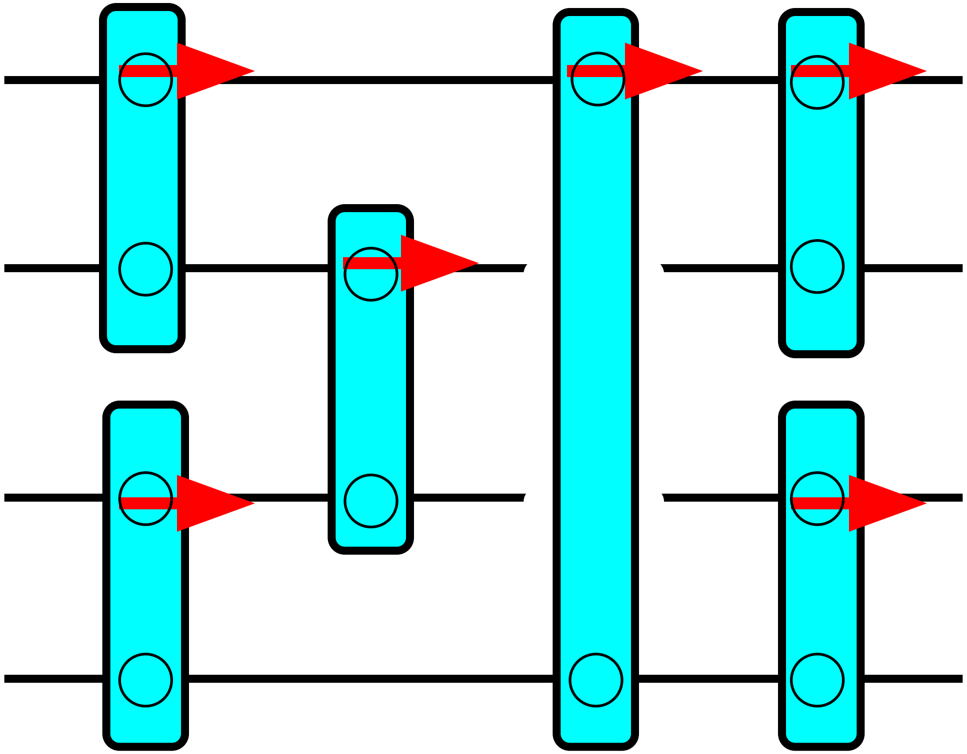
Counting Network



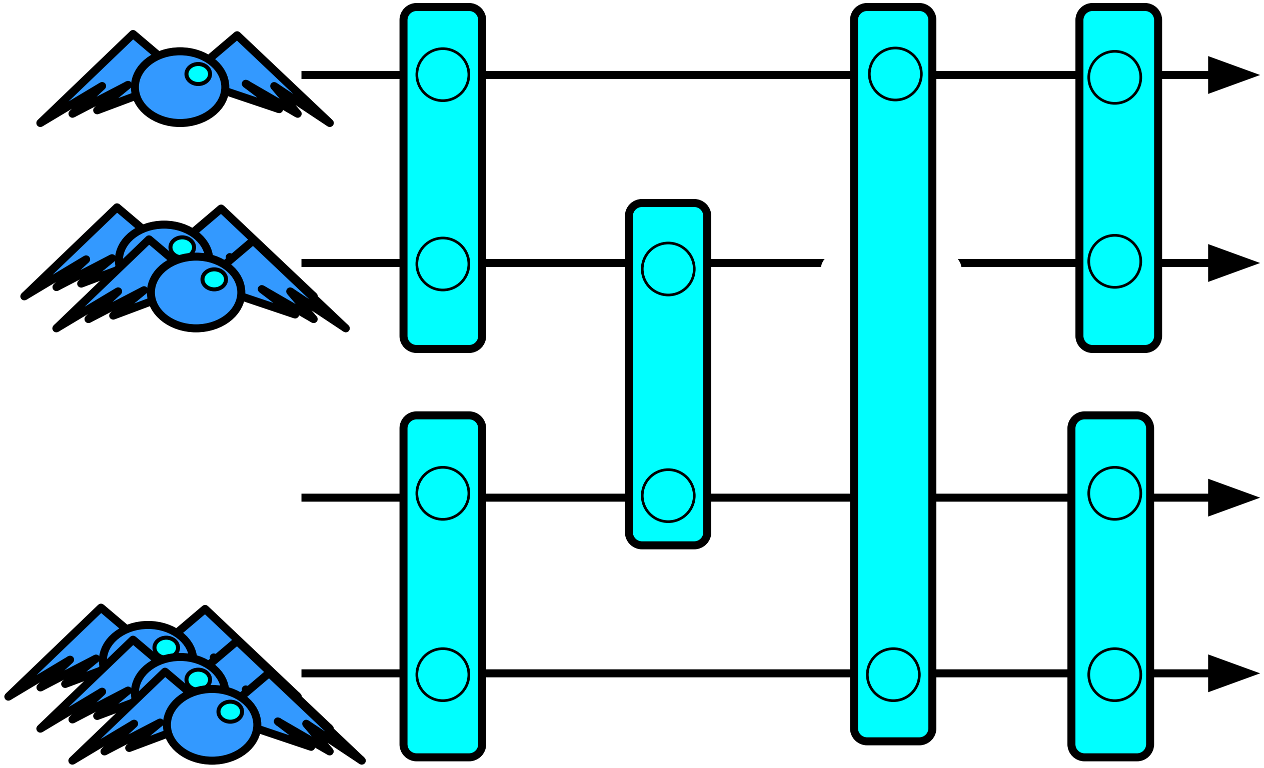
Counting Network



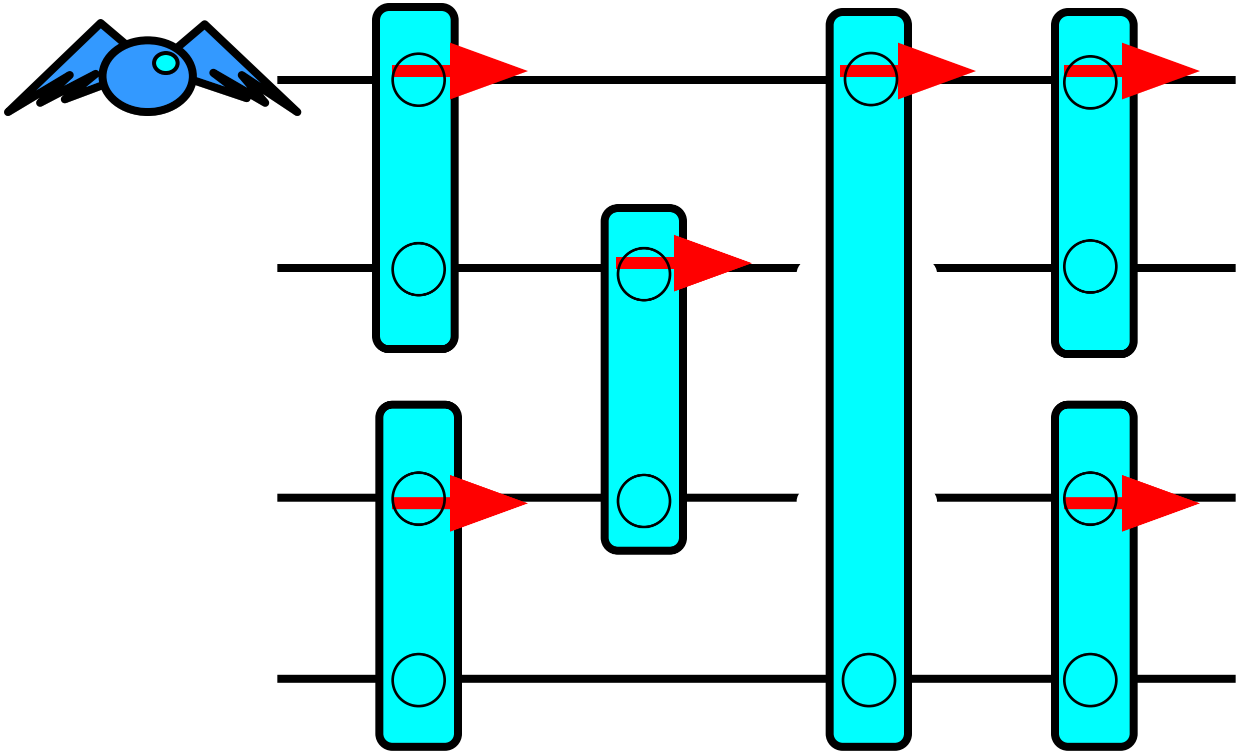
Bitonic[k] Counting Network



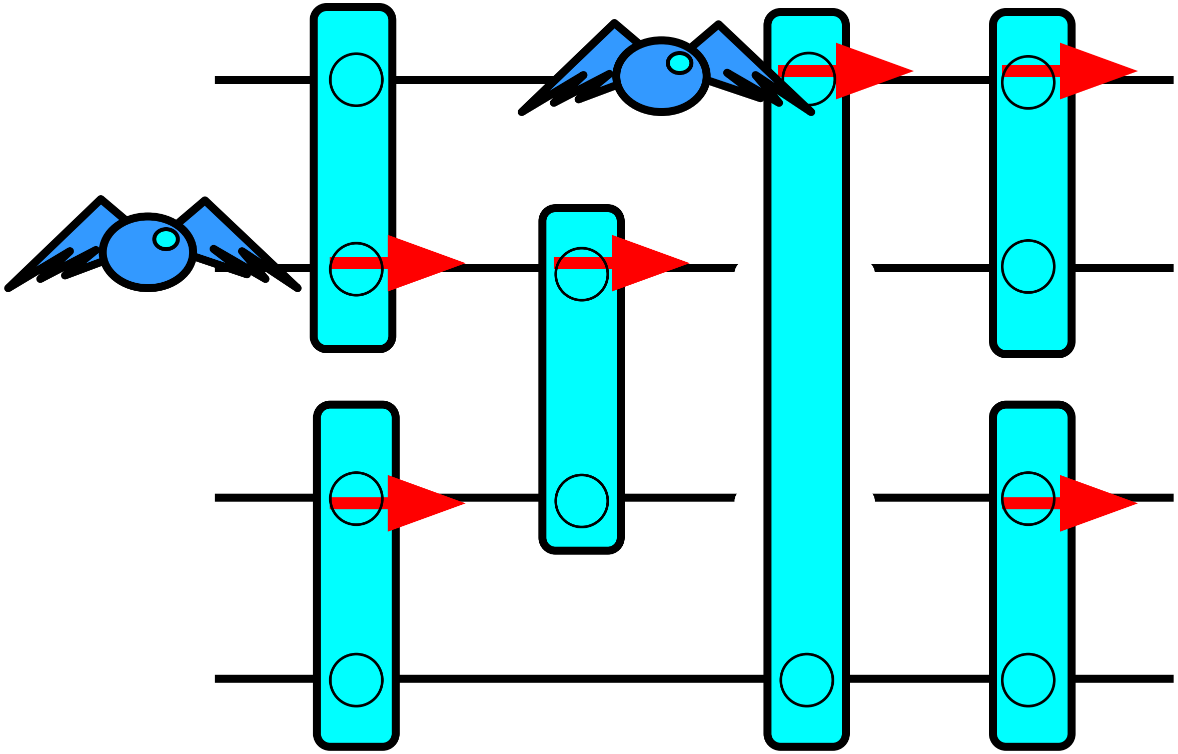
Bitonic[k] Counting Network



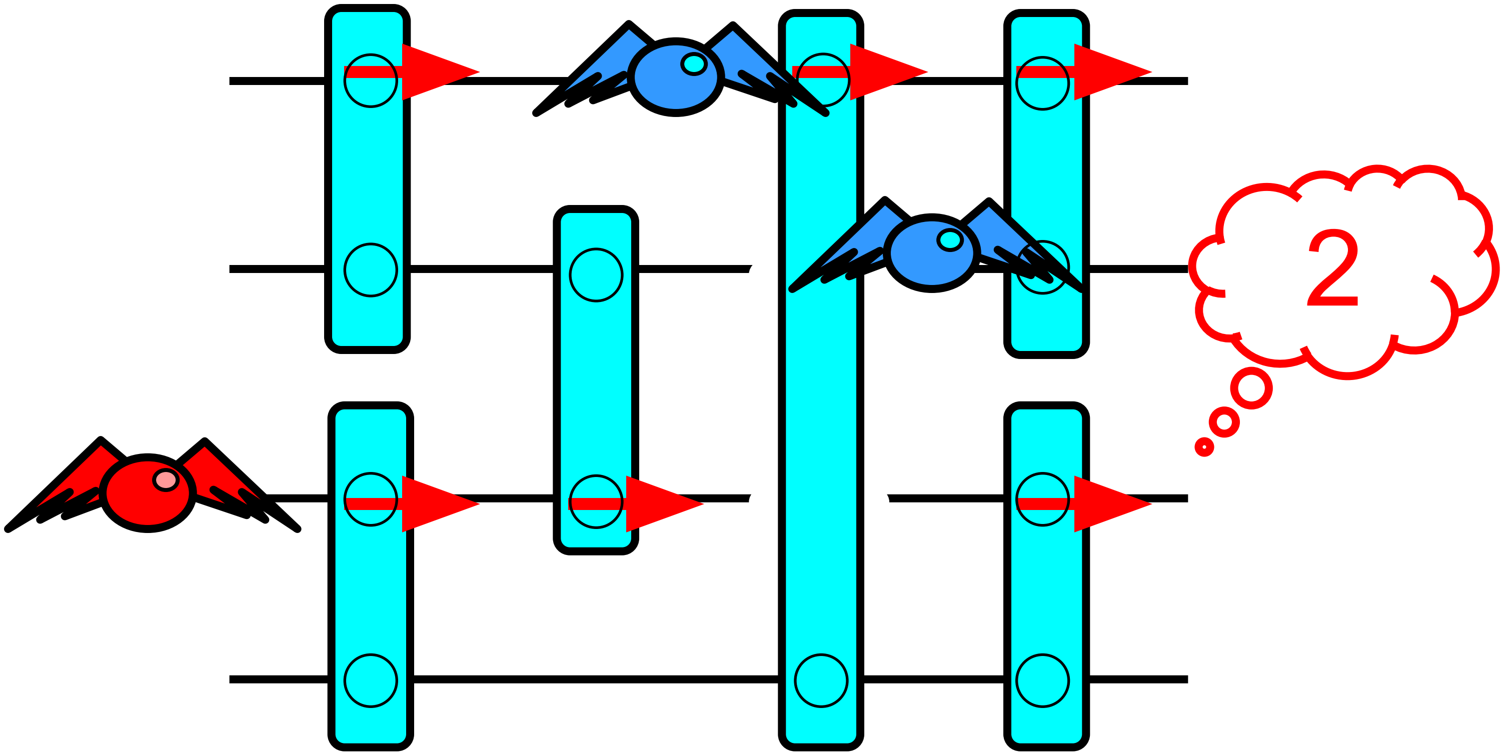
Bitonic[k] not Linearizable



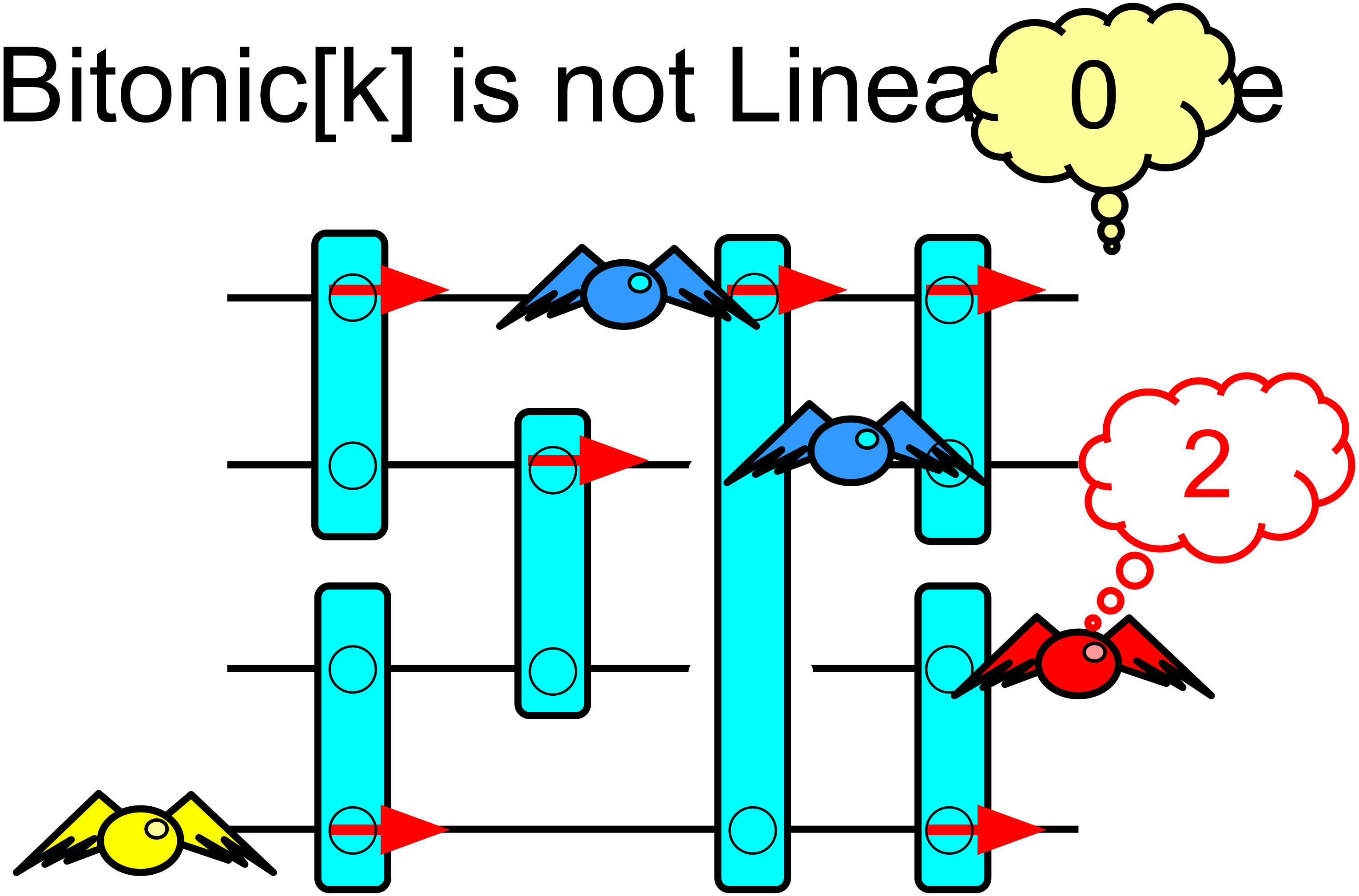
Bitonic[k] is not Linearizable



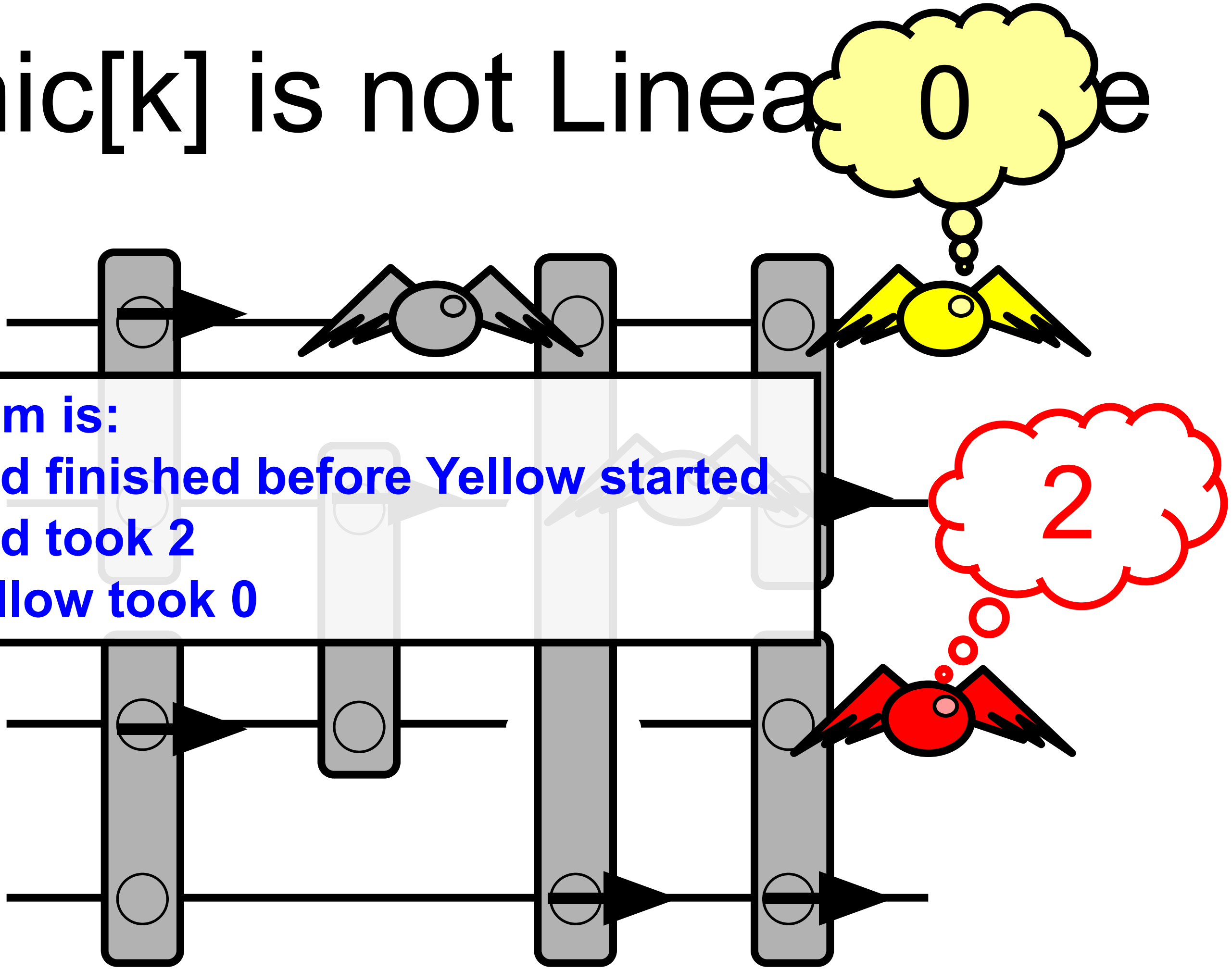
Bitonic[k] is not Linearizable



Bitonic[k] is not Linea e



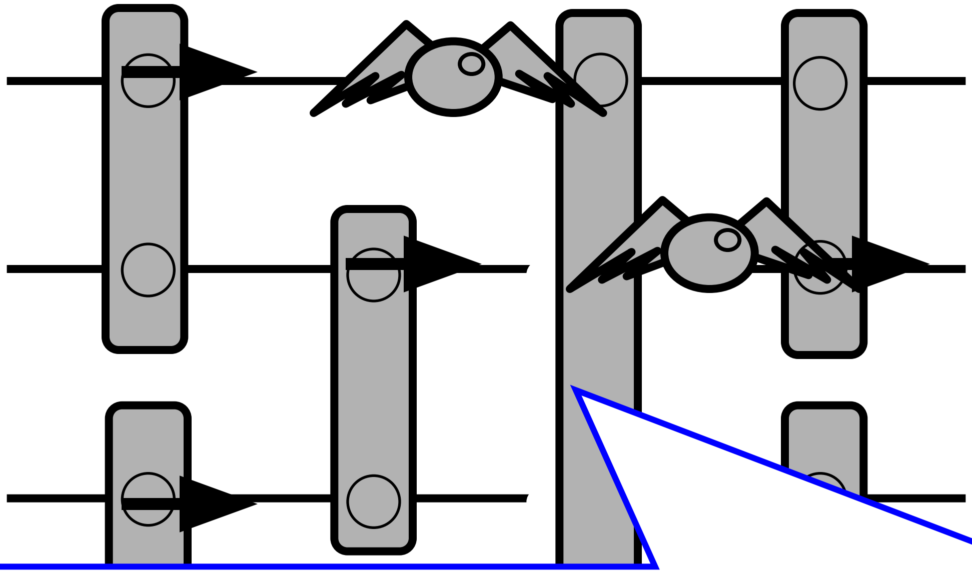
Bitonic[k] is not Linea e



Problem is:

- **Red finished before Yellow started**
- **Red took 2**
- **Yellow took 0**

But it is “Quiescently Consistent”



Has Step Property in any quiescent State (one in which all tokens have exited)

Shared Memory Implementation

```
class balancer {
 boolean toggle;
 balancer[] next;

synchronized boolean flip() {
 boolean oldValue = this.toggle;
 this.toggle = !this.toggle;
 return oldValue;
}
```

Shared Memory Implementation

```
class balancer {
 boolean toggle;
 balancer[] next;

synchronized boolean flip() {
 boolean oldValue = this.toggle;
 this.toggle = !this.toggle;
 return oldValue;
}
```

state

Shared Memory Implementation

```
class balancer {
 boolean toggle;
 balancer[] next;

synchronized boolean flip() {
 boolean oldValue = this.toggle;
 this.toggle = !this.toggle;
 return oldValue;
}
```

Output connections to balancers

Shared Memory Implementation

```
class balancer {
 boolean toggle;
 balancer[] next;

synchronized boolean flip() {
 boolean oldValue = this.toggle;
 this.toggle = !this.toggle;
 return oldValue;
}
```

getAndComplement

Shared Memory Implementation

```
Balancer traverse (Balancer b) {
 while(!b.isLeaf()) {
  boolean toggle = b.flip();
  if (toggle)
    b = b.next[0]
  else
    b = b.next[1]
  return b;
}
```

Shared Memory Implementation

```
Balancer traverse (Balancer b) {
 while(!b.isLeaf()) {
  boolean toggle = b.flip();
  if (toggle)
    b = b.next[0]
  else
    b = b.next[1]
  return b;
}
```

Stop when we exit the network

Shared Memory Implementation

```
Balancer traverse (Balancer b) {
 while(!b.isLeaf()) {
  boolean toggle = b.flip();
  if (toggle)
    b = b.next[0]
  else
    b = b.next[1]
  return b;
}
```

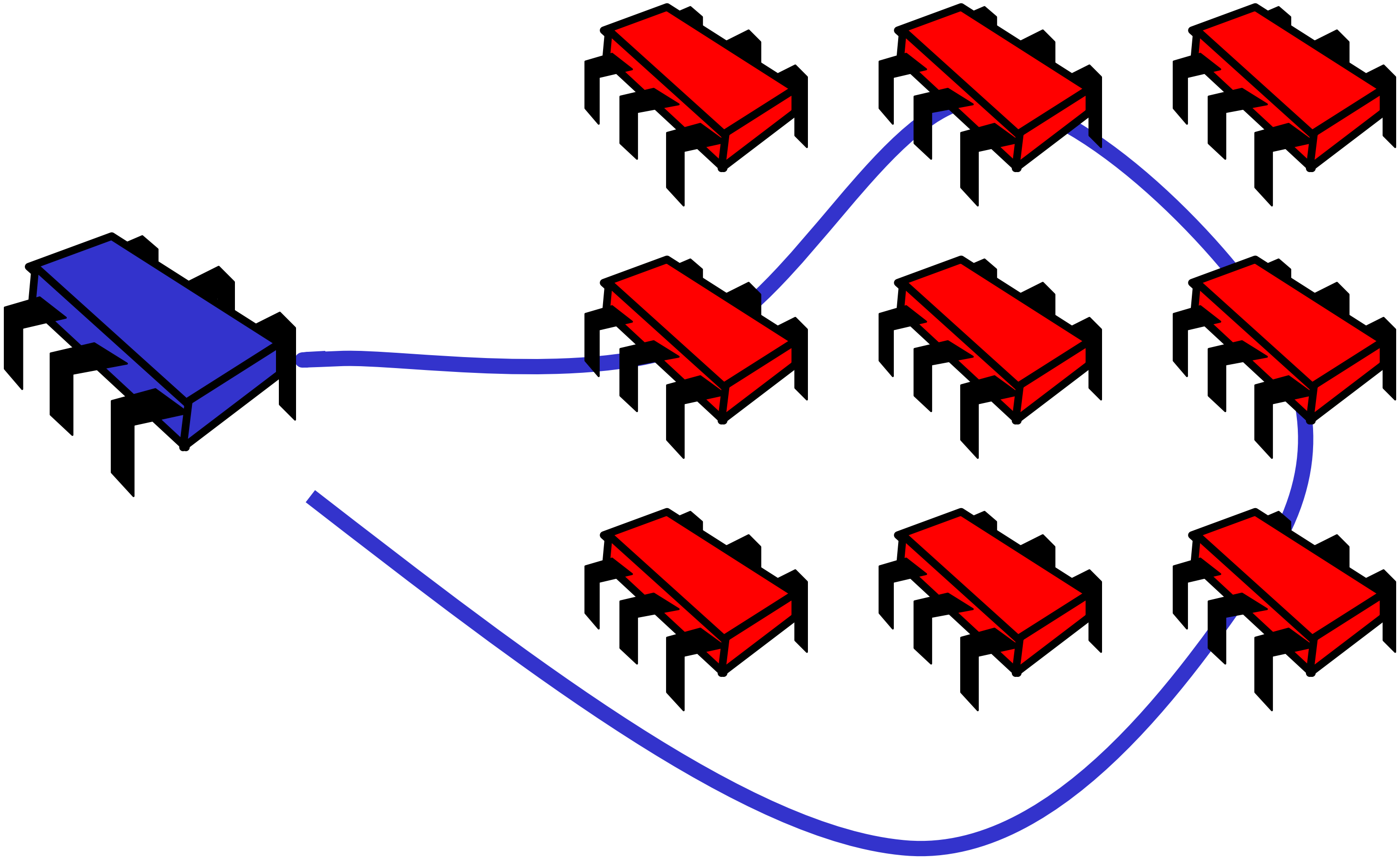
Flip state

Shared Memory Implementation

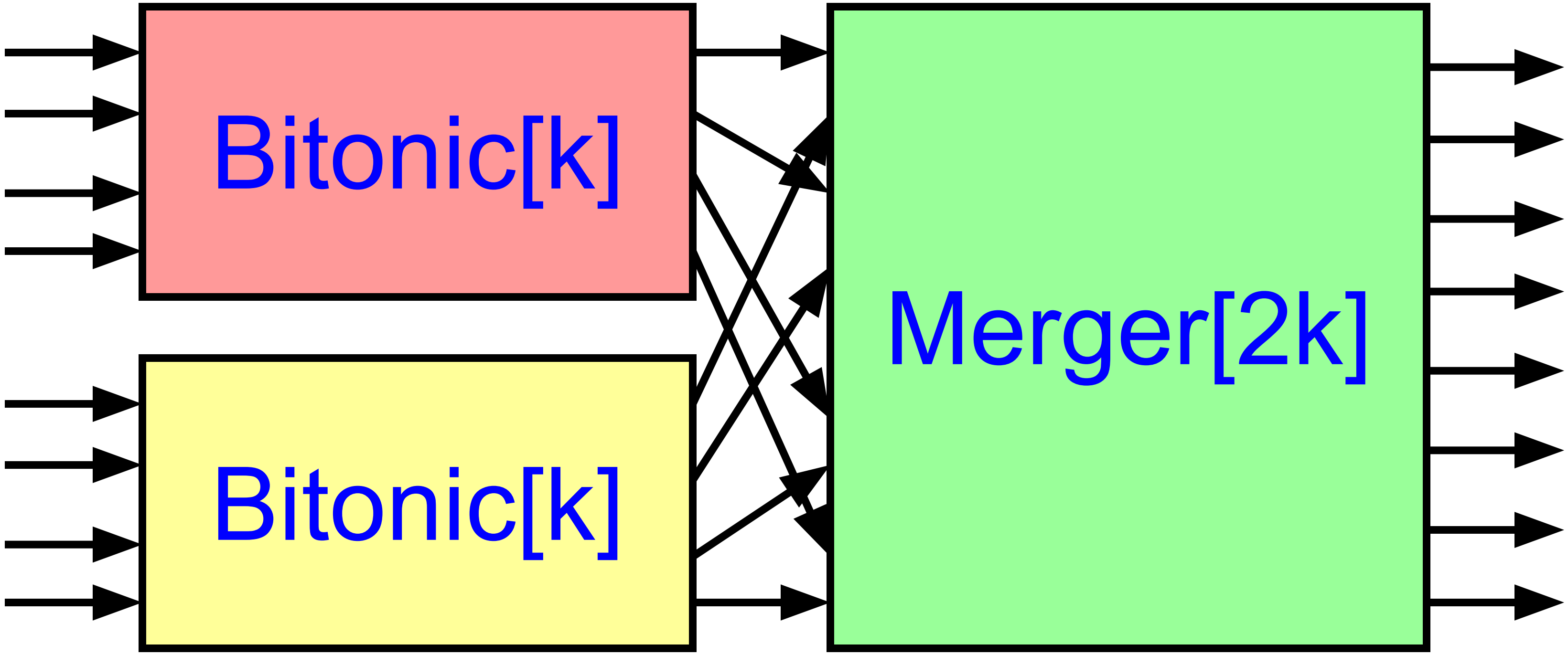
```
Balancer traverse (Balancer b) {
 while(!b.isLeaf()) {
  boolean toggle = b.flip();
  if (toggle)
    b = b.next[0]
  else
    b = b.next[1]
  return b;
}
```

Exit on wire

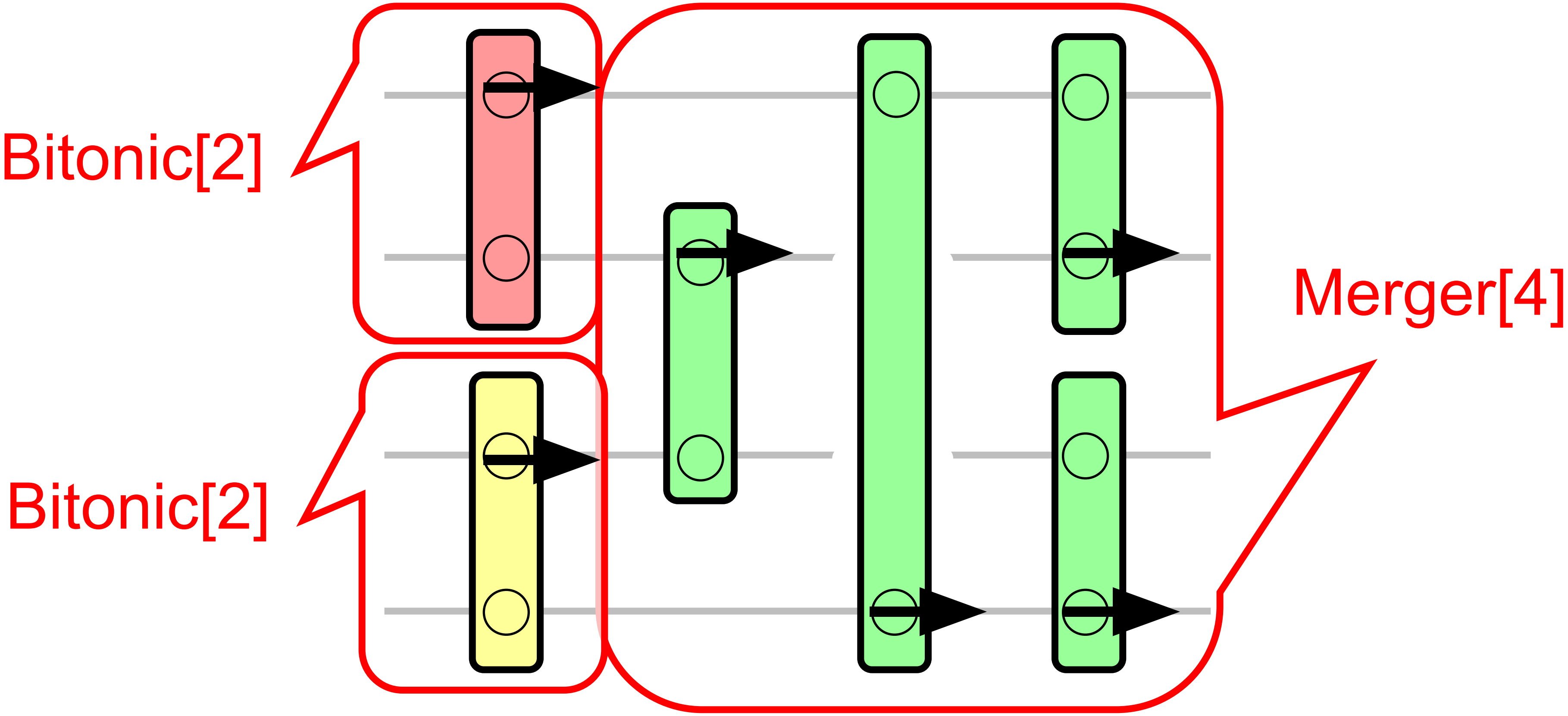
Alternative Implementation: Message-Passing



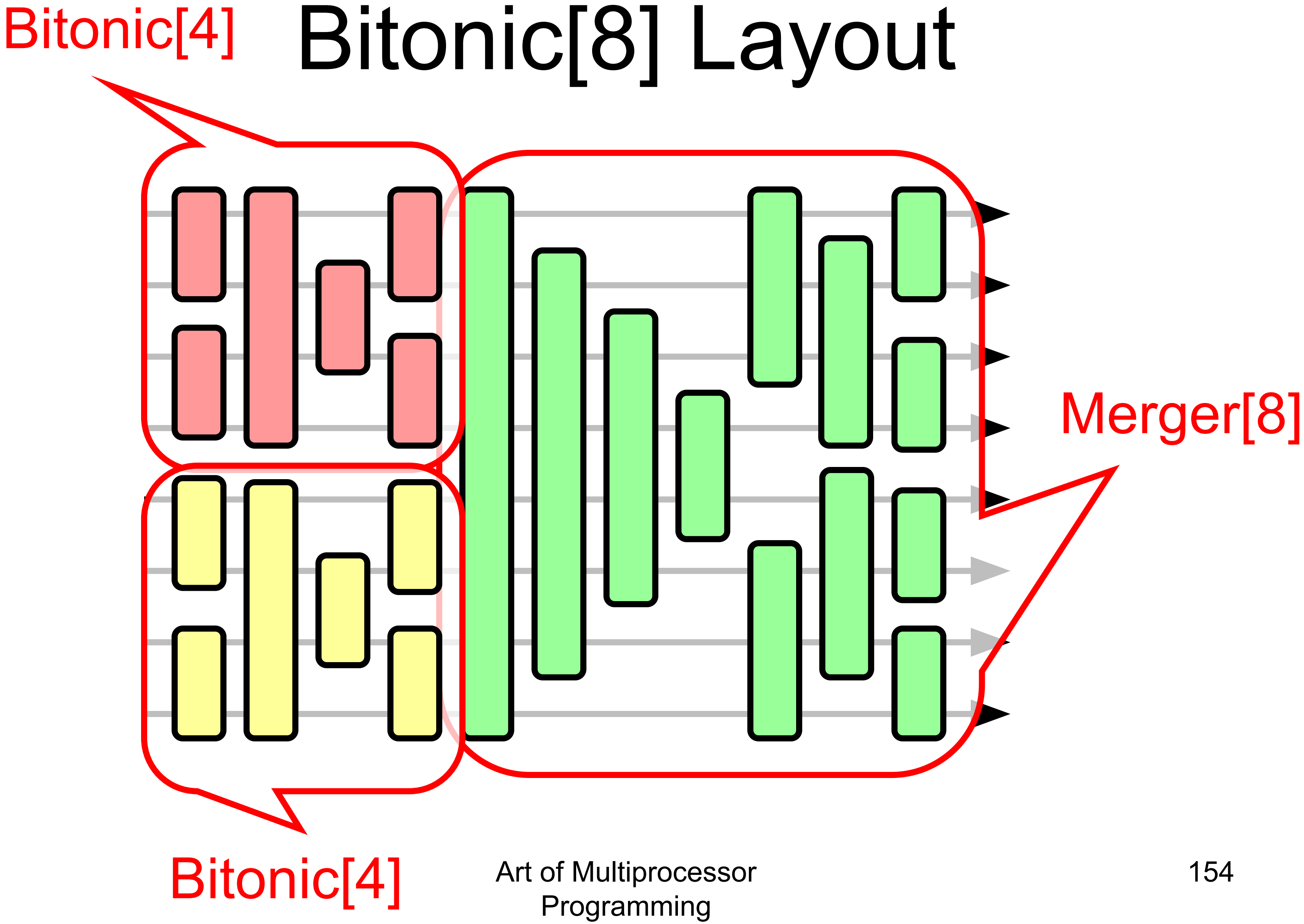
Bitonic[2k] Inductive Structure



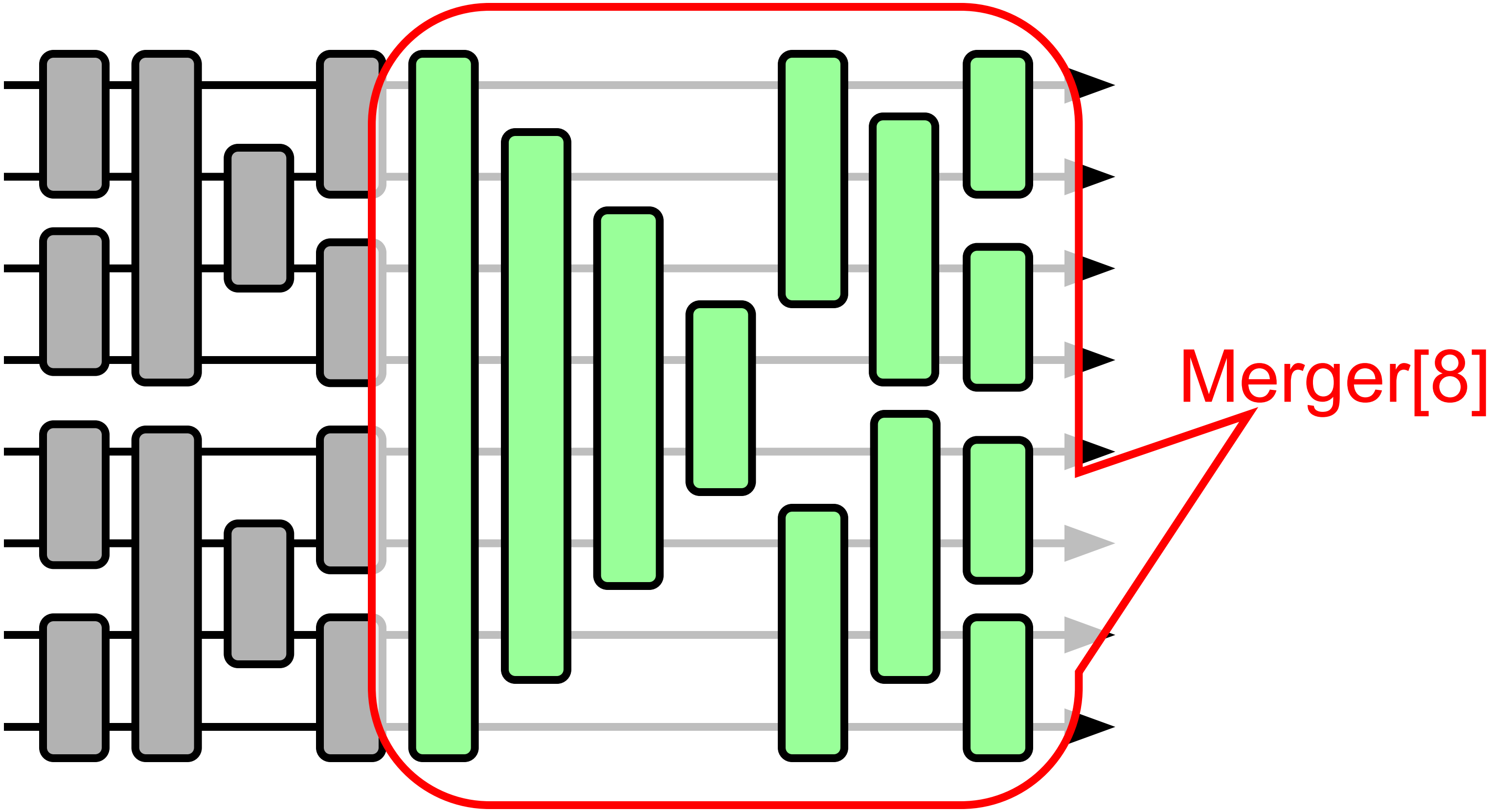
Bitonic[4] Counting Network



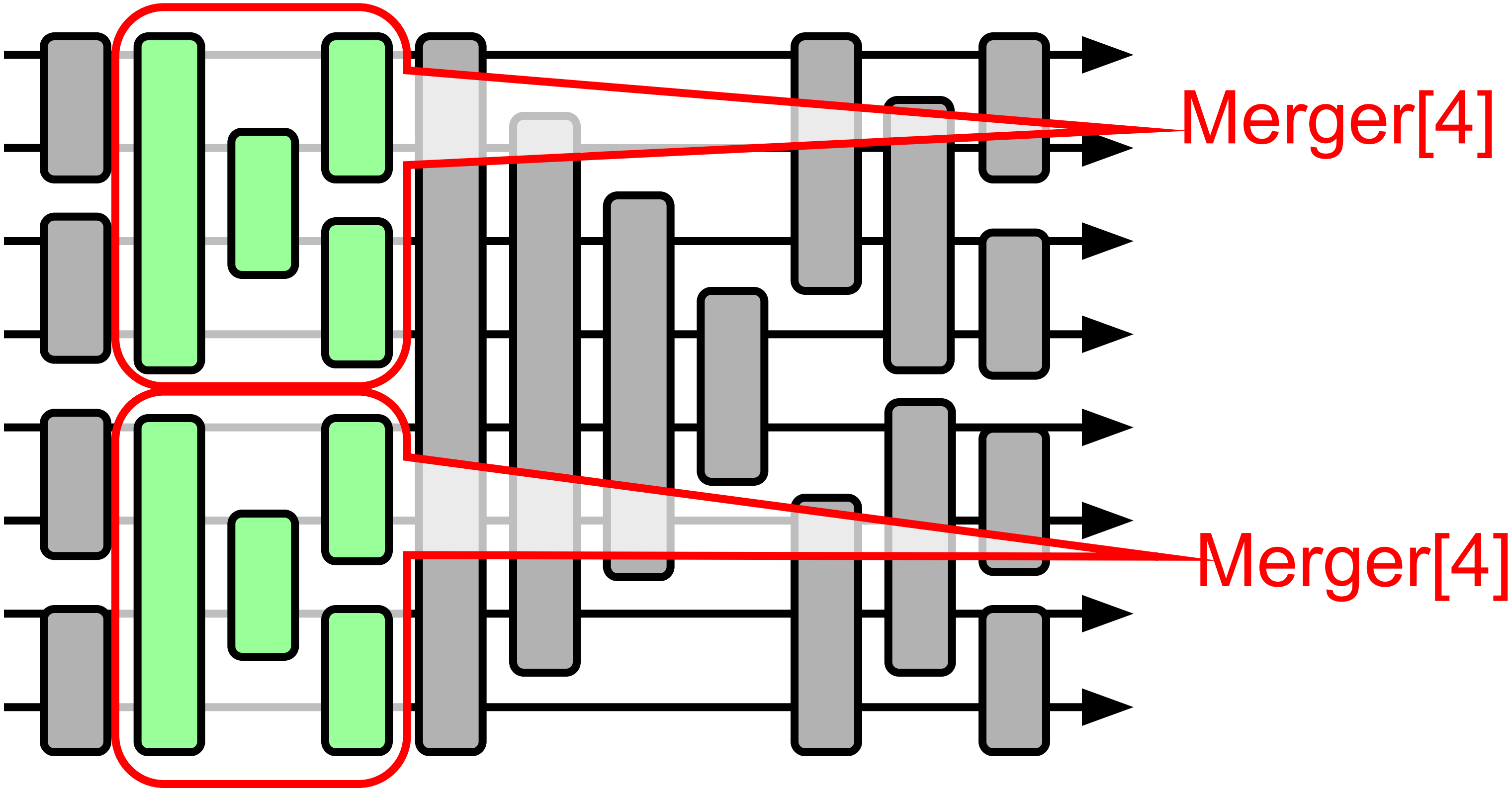
Bitonic[8] Layout



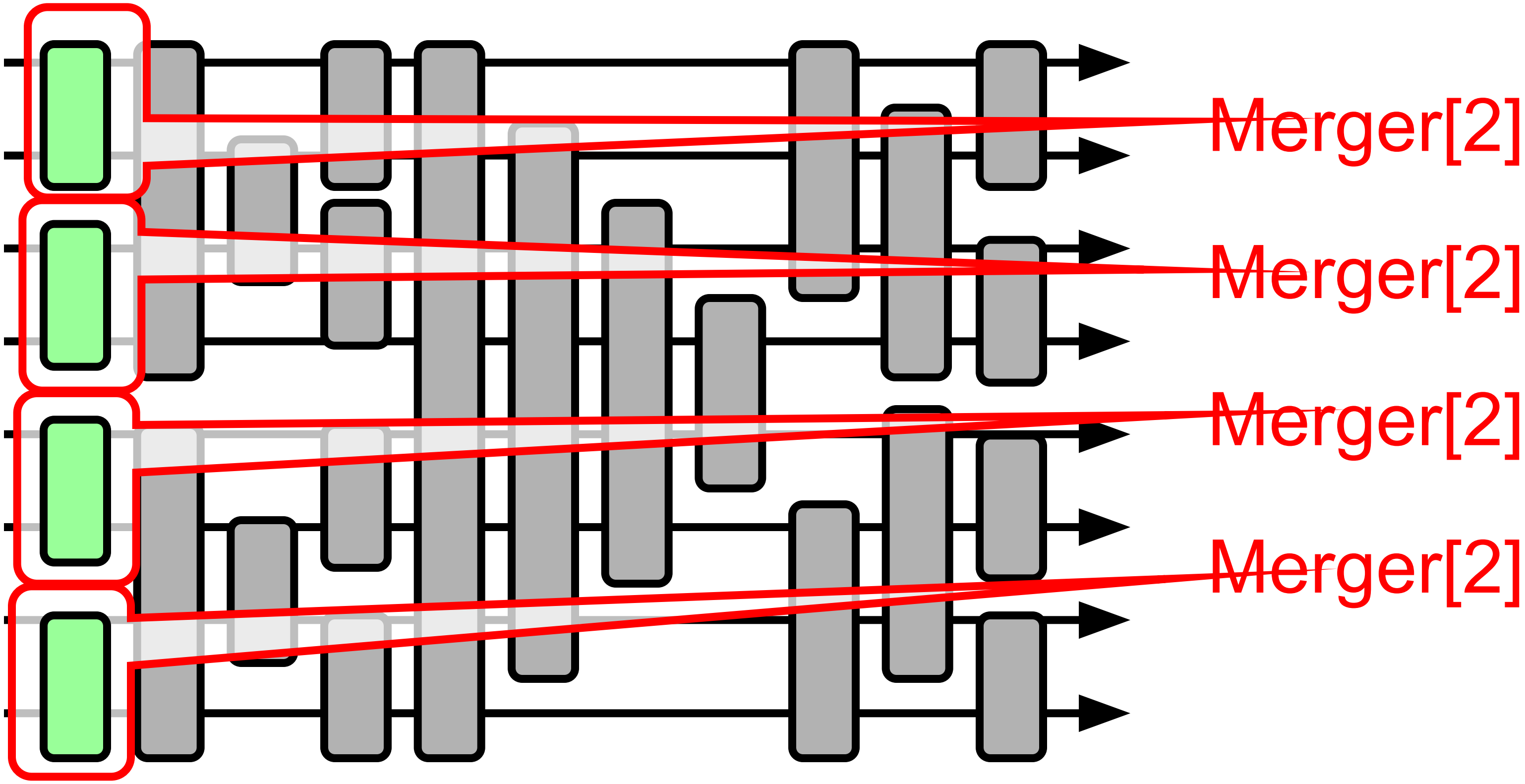
Unfolded Bitonic[8] Network



Unfolded Bitonic[8] Network



Unfolded Bitonic[8] Network



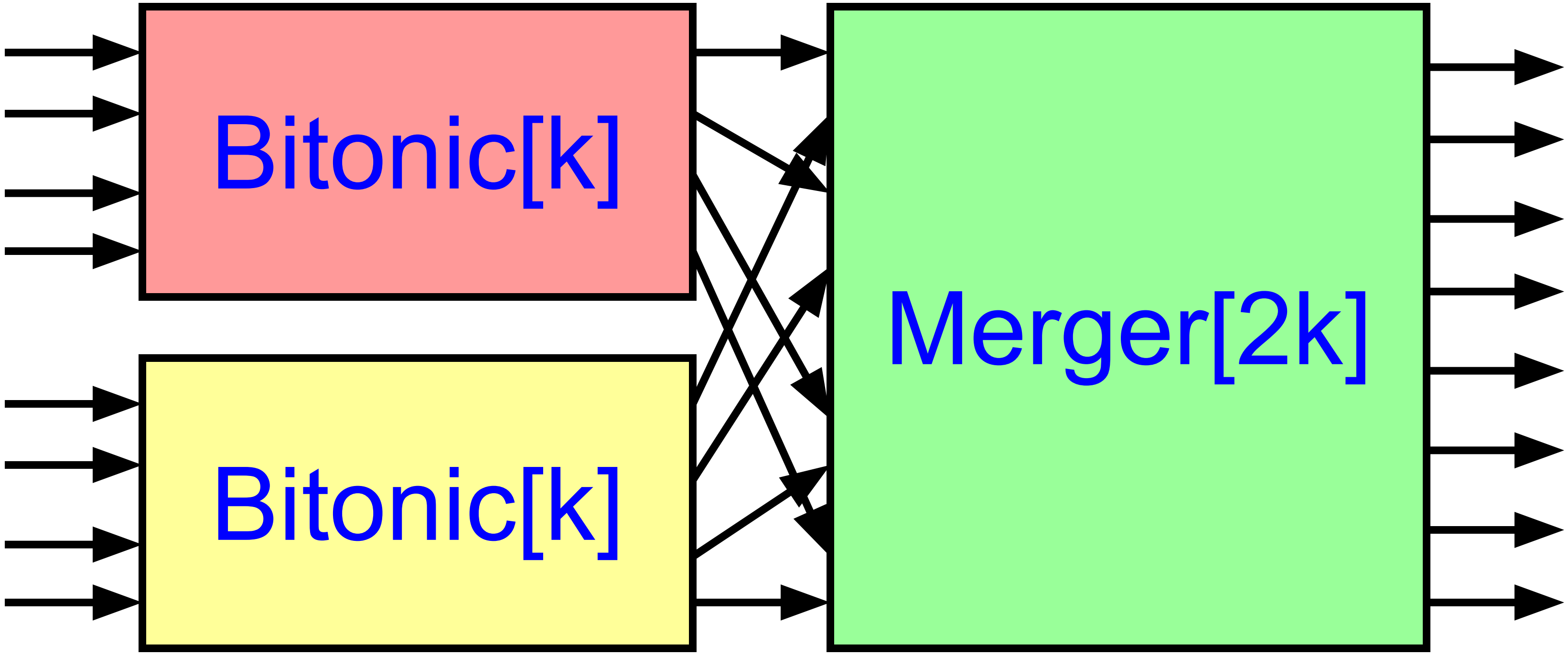
Bitonic[k] Depth

- Width k
- Depth is $(\log_2 k)(\log_2 k + 1)/2$

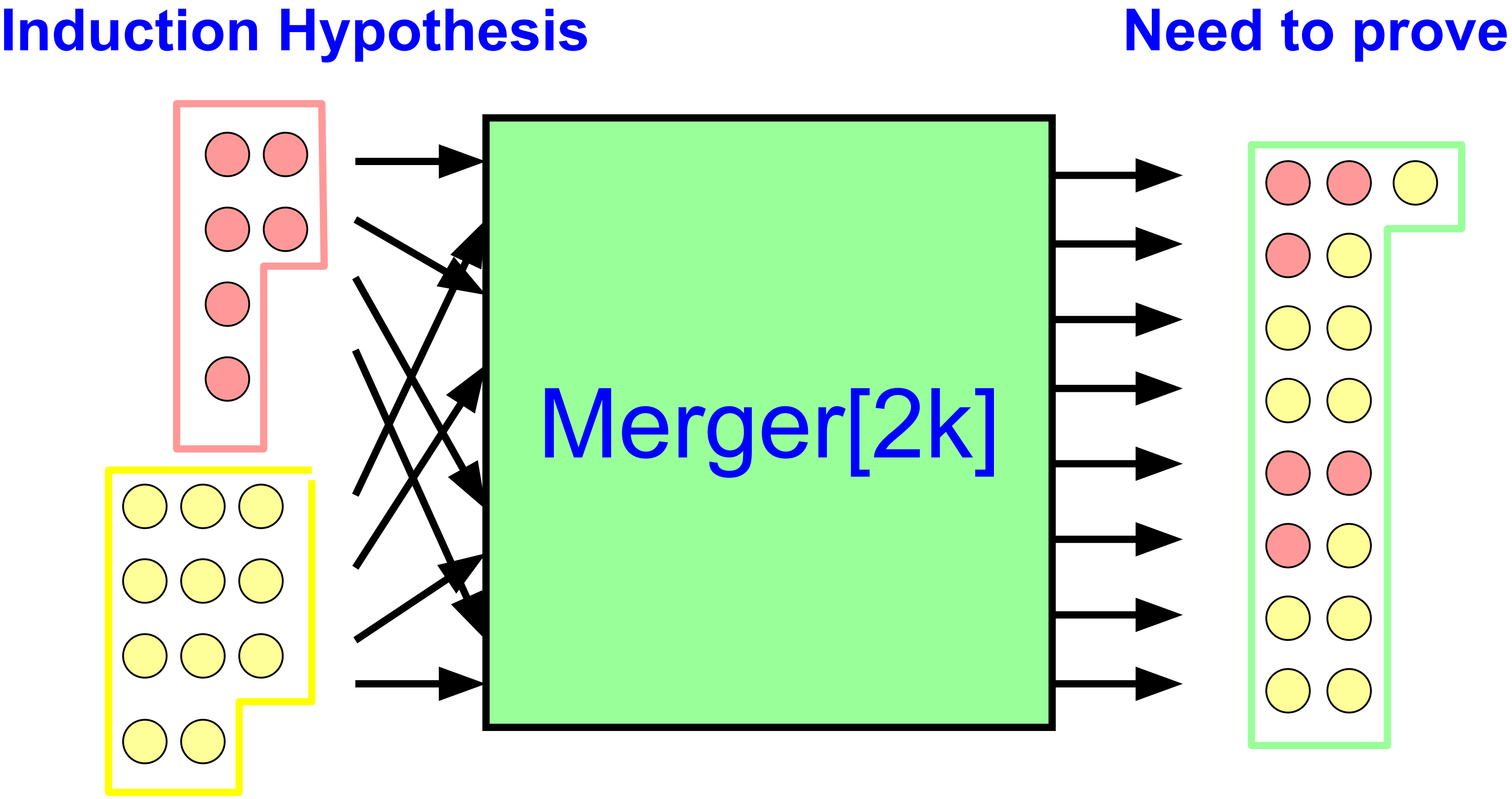
Proof by Induction

- Base:
  - Bitonic[2] is single balancer
  - has step property by definition
- Step:
  - If Bitonic[k] has step property …
  - So does Bitonic[2k]

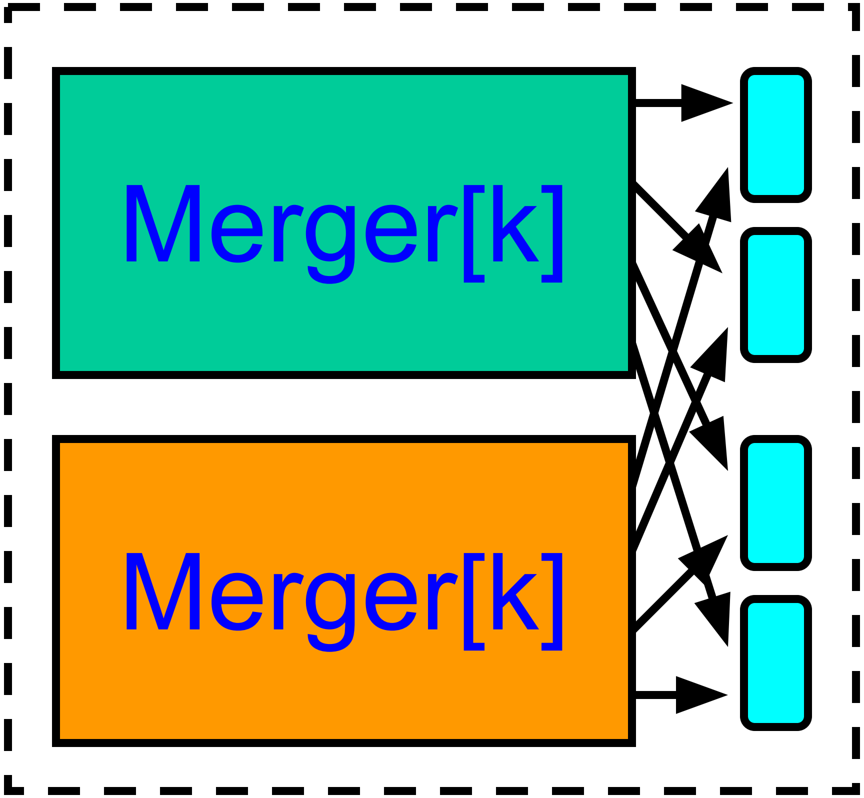
Bitonic[2k] Schematic



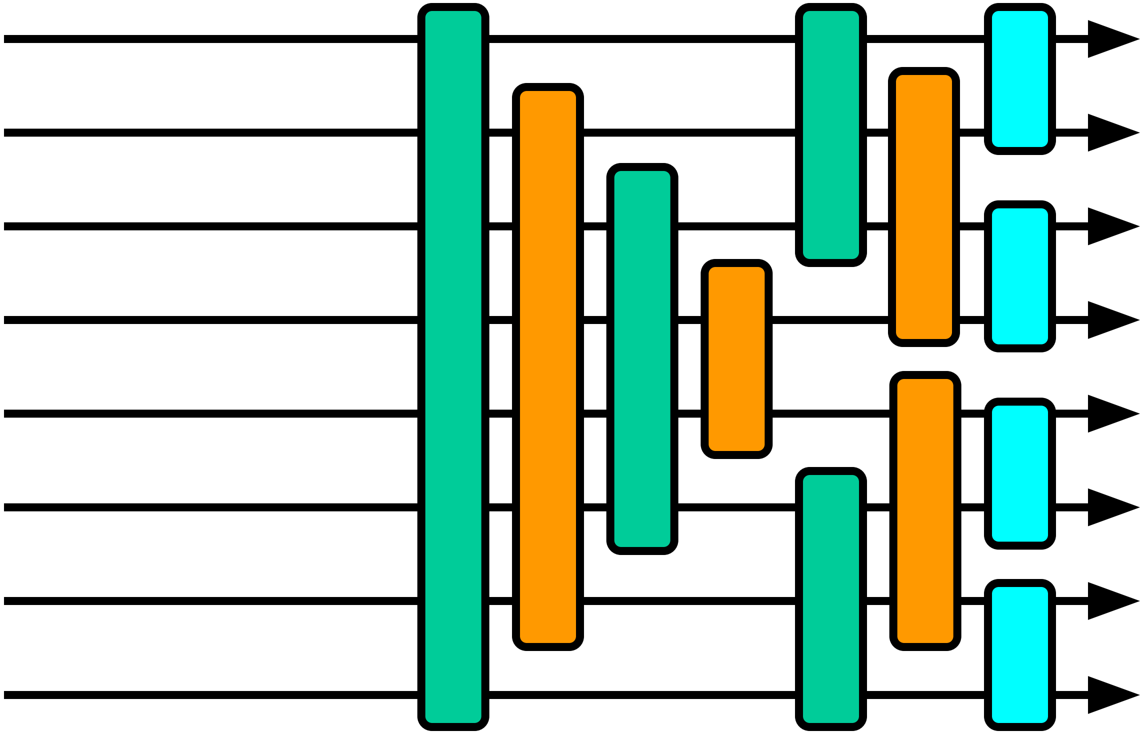
Need to Prove only Merger[2k]



Merger[2k] Schematic



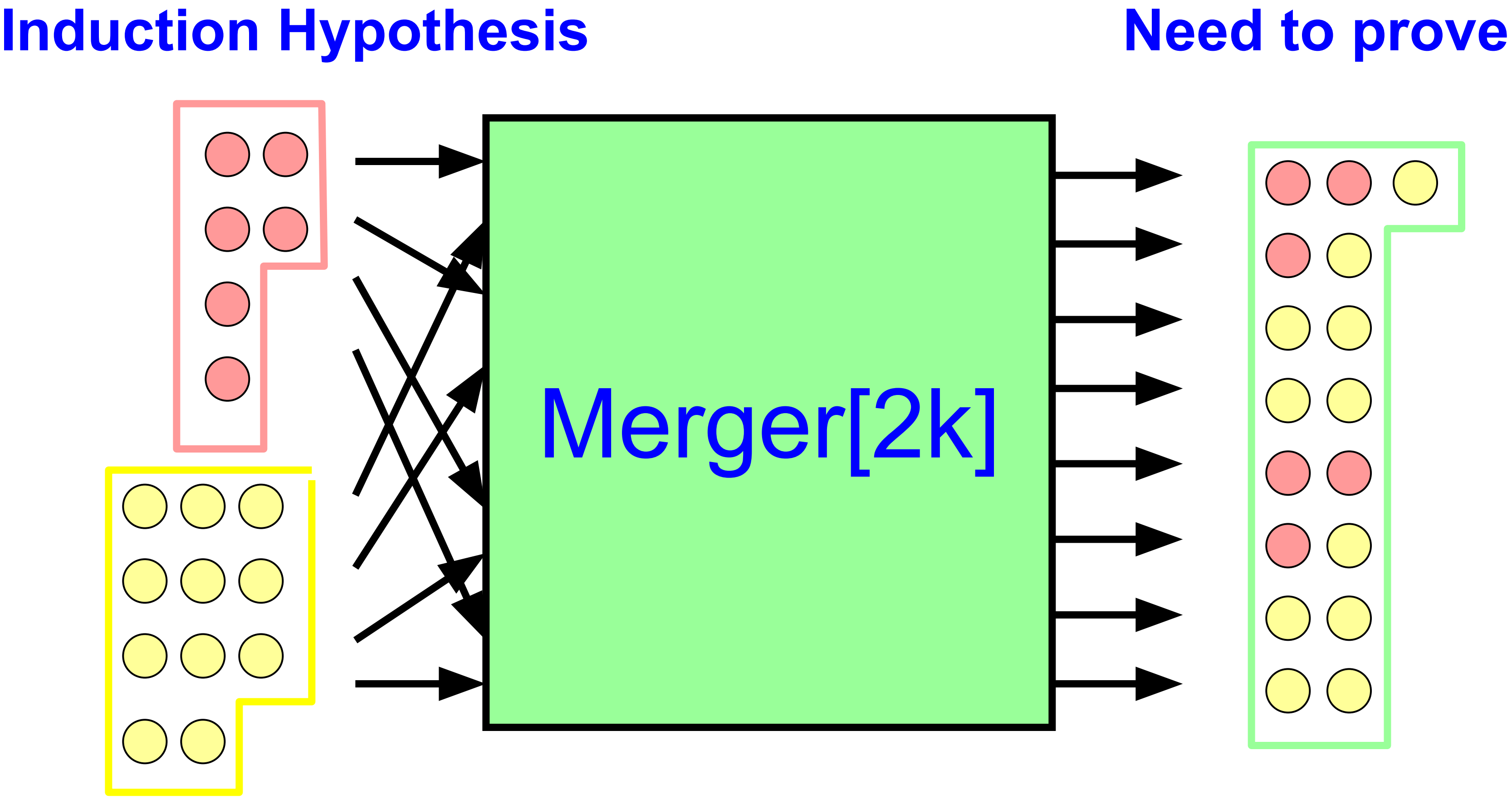
Merger[2k] Layout



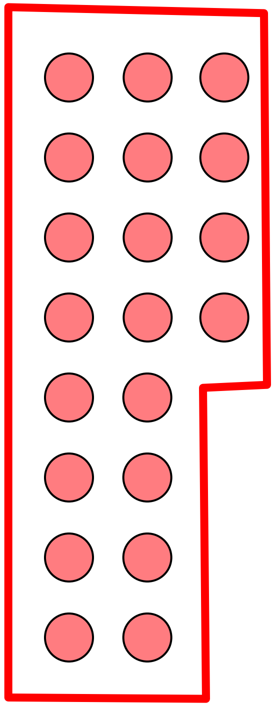
Induction Step

- Bitonic[k] has step property …
- Assume Merger[k] has step property if it gets 2 inputs with step property of size k/2 and
- prove Merger[2k] has step property

Assume Bitonic[k] and Merger[k] and Prove Merger[2k]

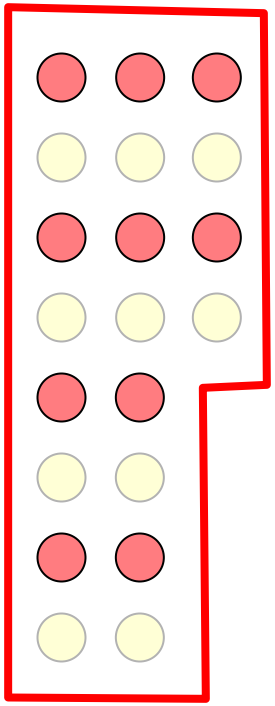


Proof: Lemma 1



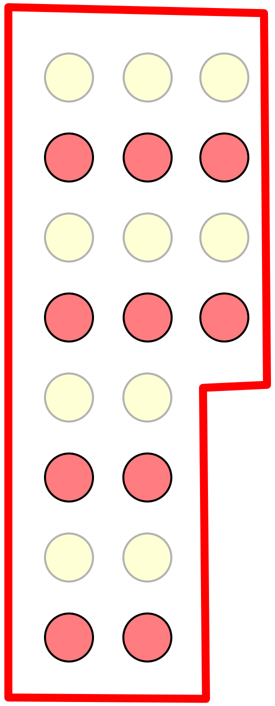
If a sequence has the step property …

Lemma 1



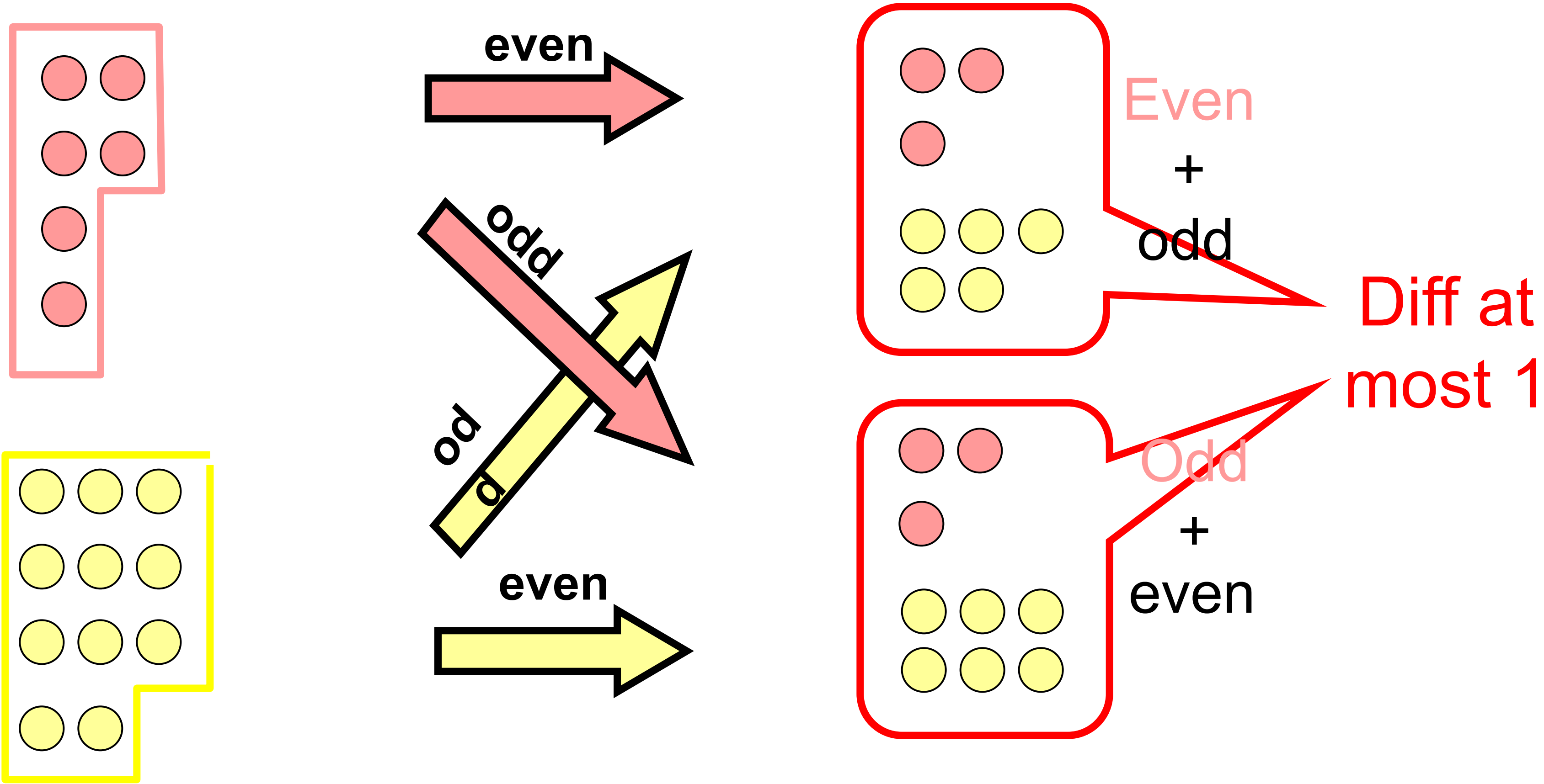
So does its even subsequence

Lemma 1



Also its odd subsequence

Lemma 2



even

odd

od

d

even

Even

+

odd

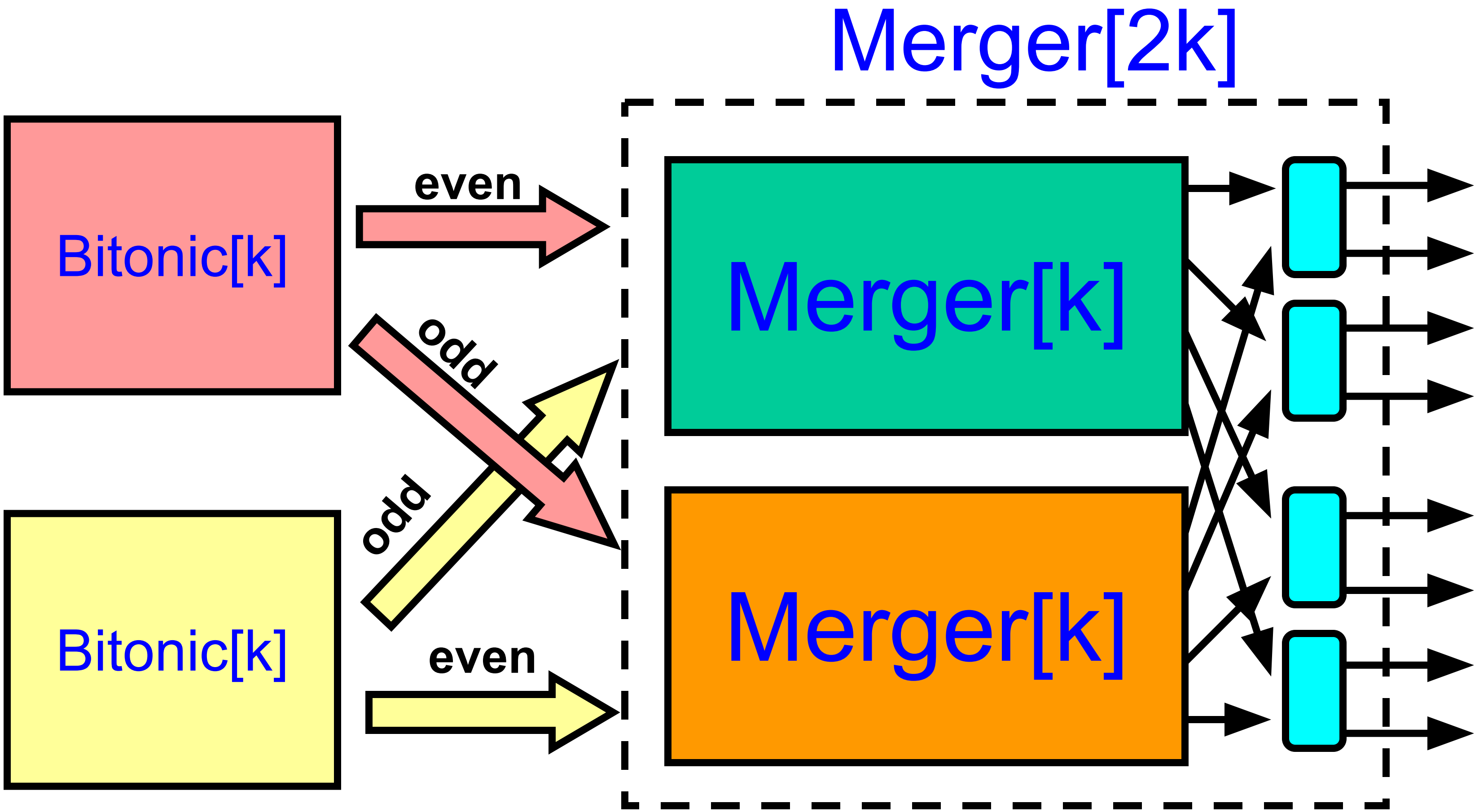
Odd

+

even

Diff at most 1

Bitonic[2k] Layout Details



Merger[2k]

Bitonic[k]

Bitonic[k]

even

odd

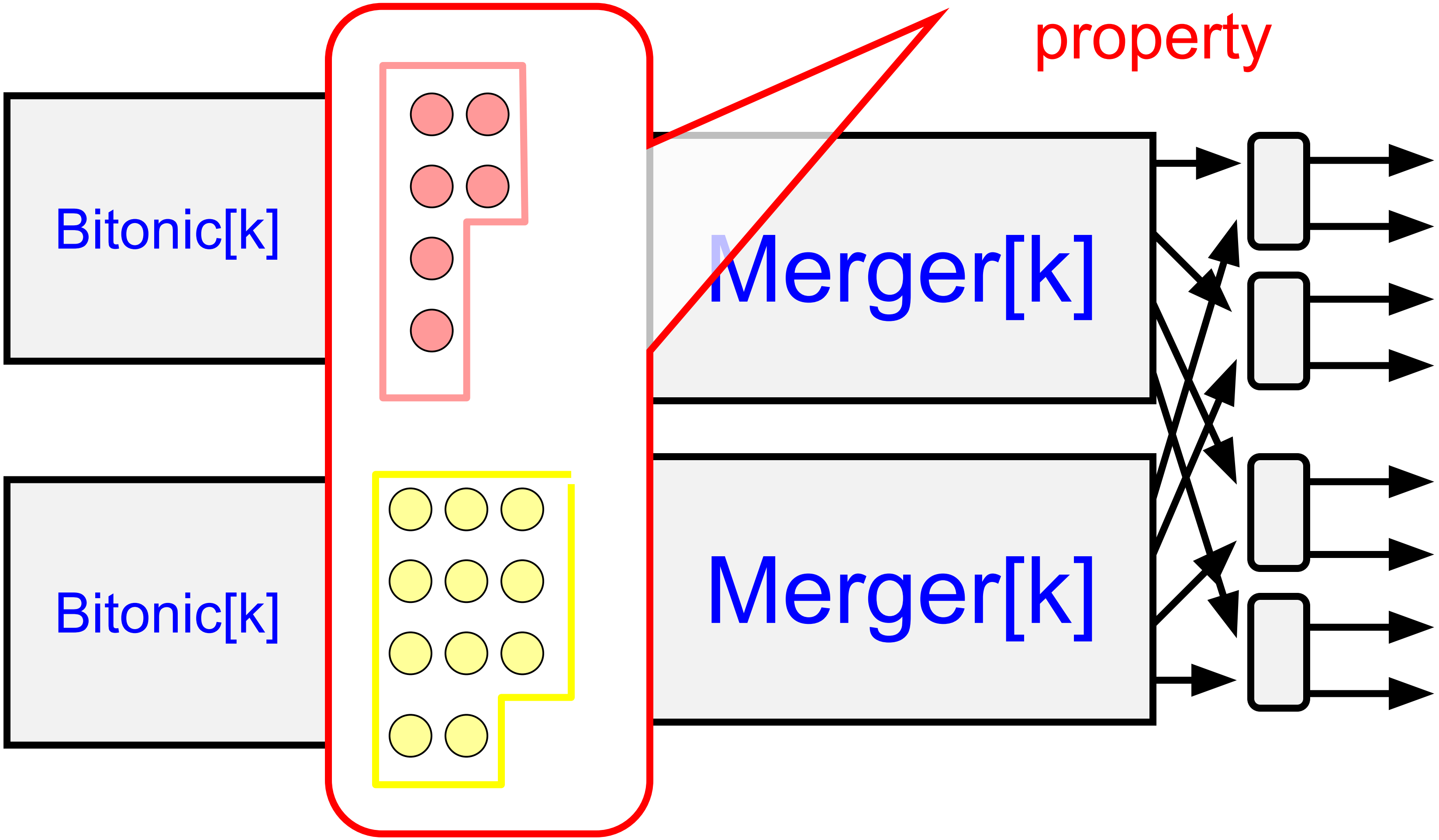
odd

even

Merger[k]

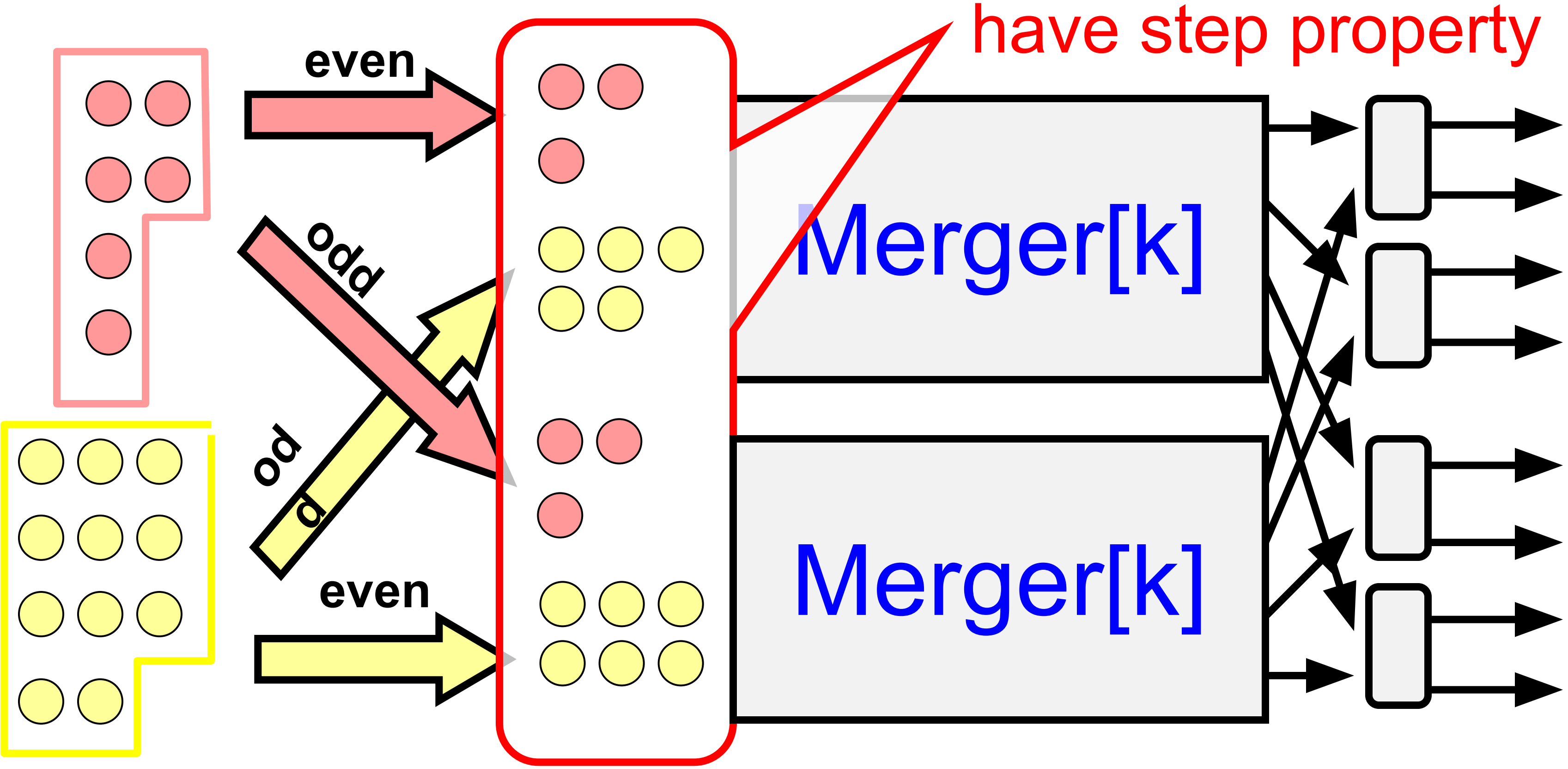
Merger[k]

By induction hypothesis



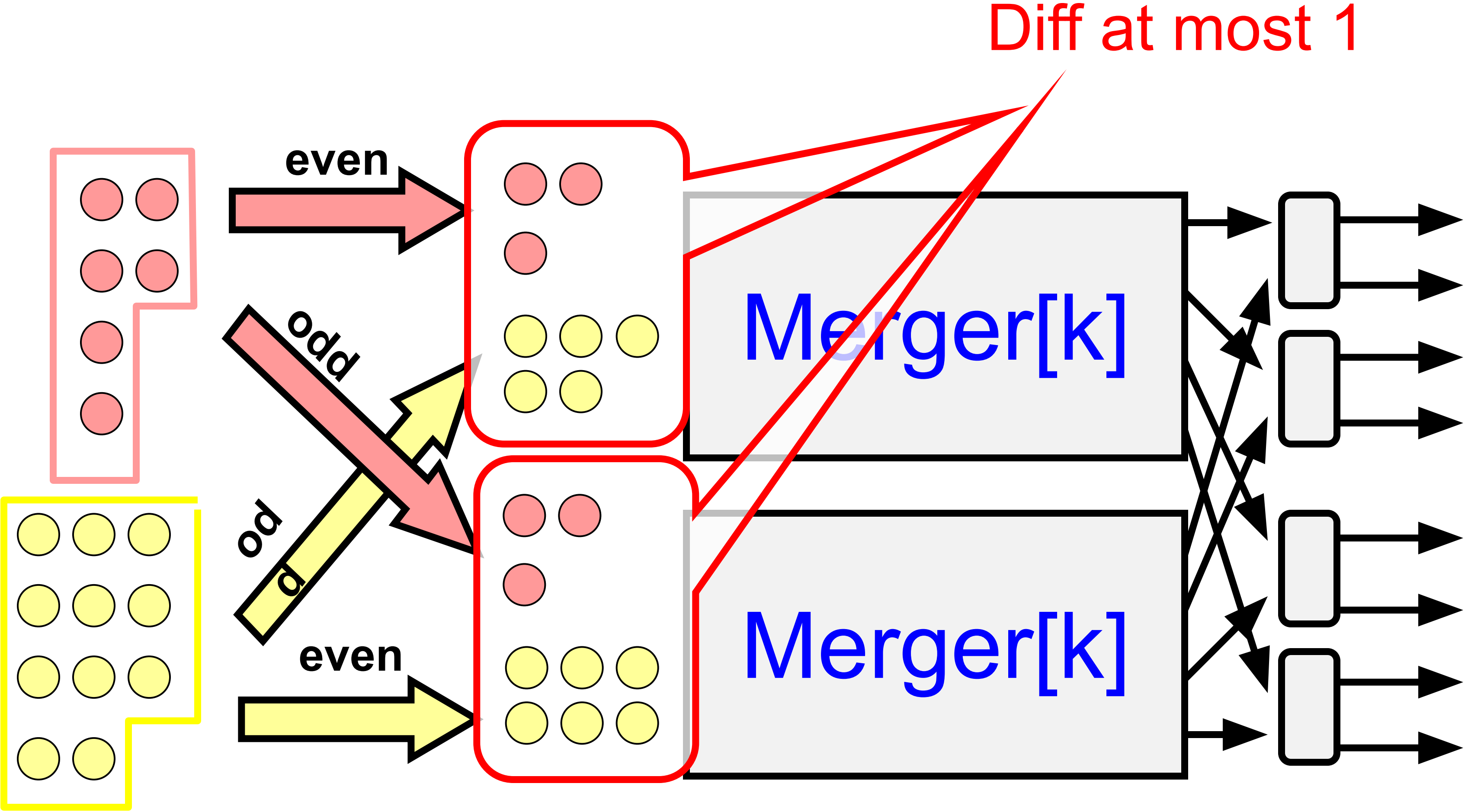
Outputs have step property

By Lemma 1



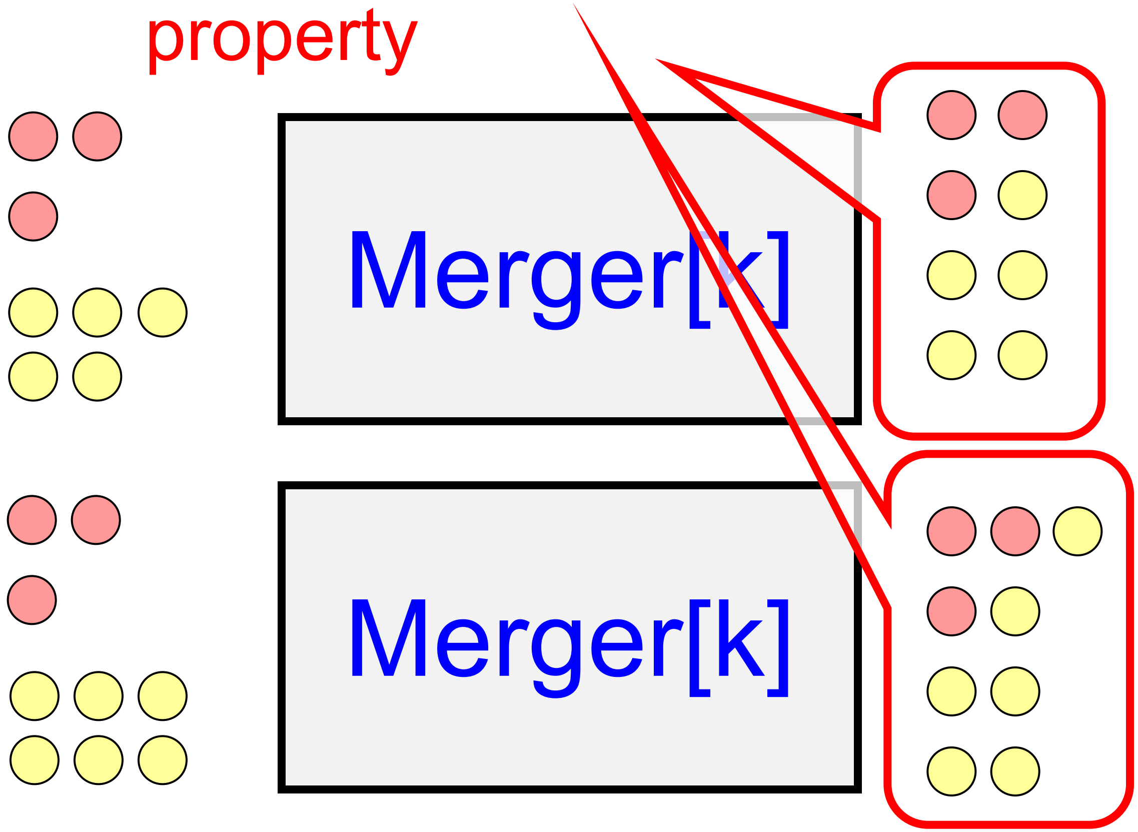
All subsequences have step property

By Lemma 2

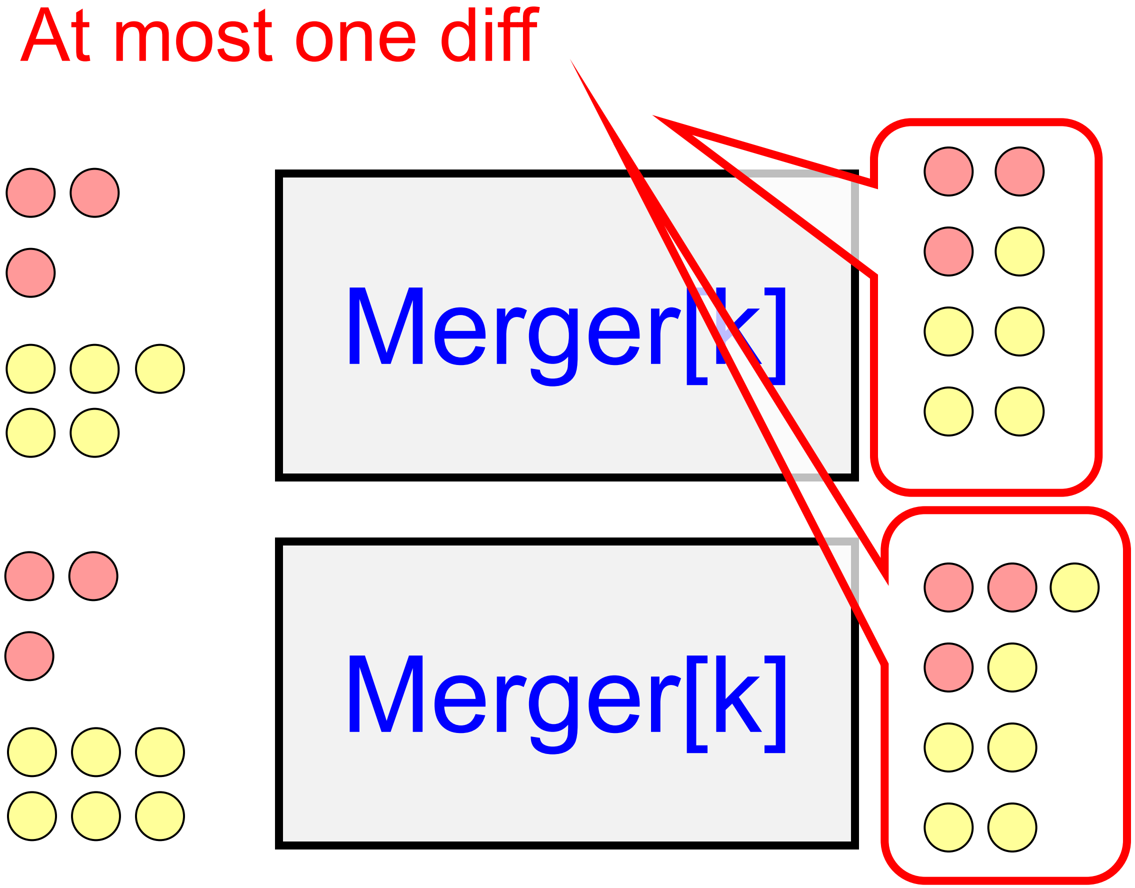


By Induction Hypothesis

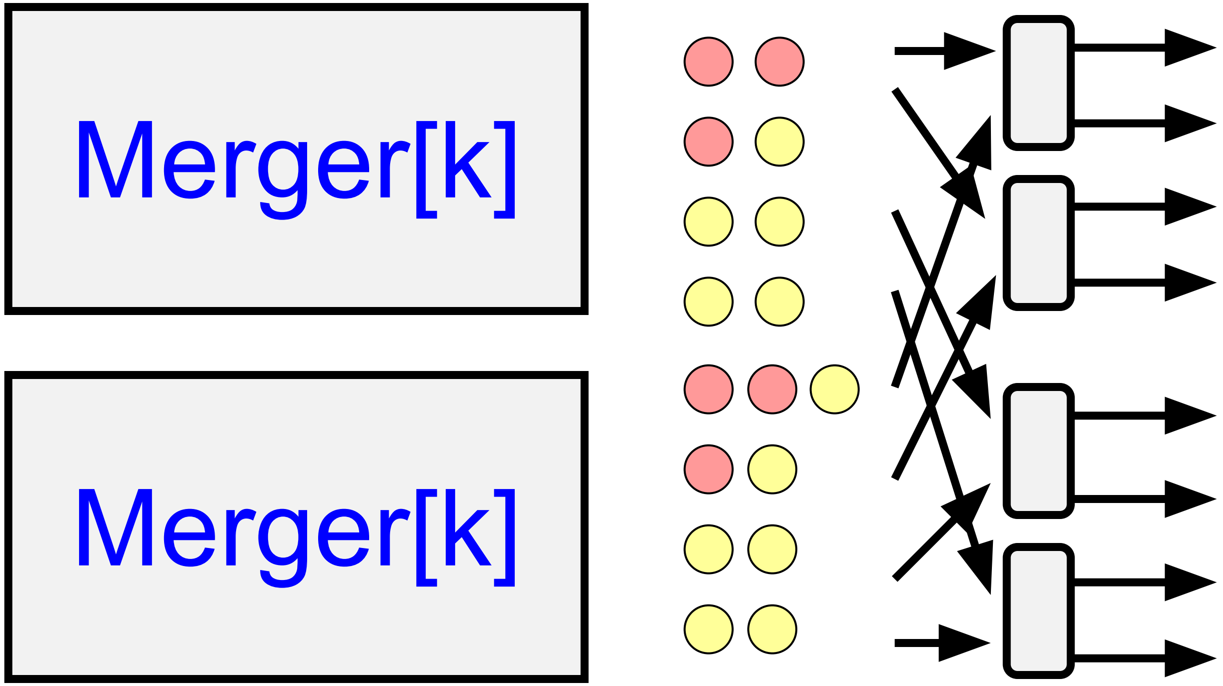
Outputs have step property



By Lemma 2



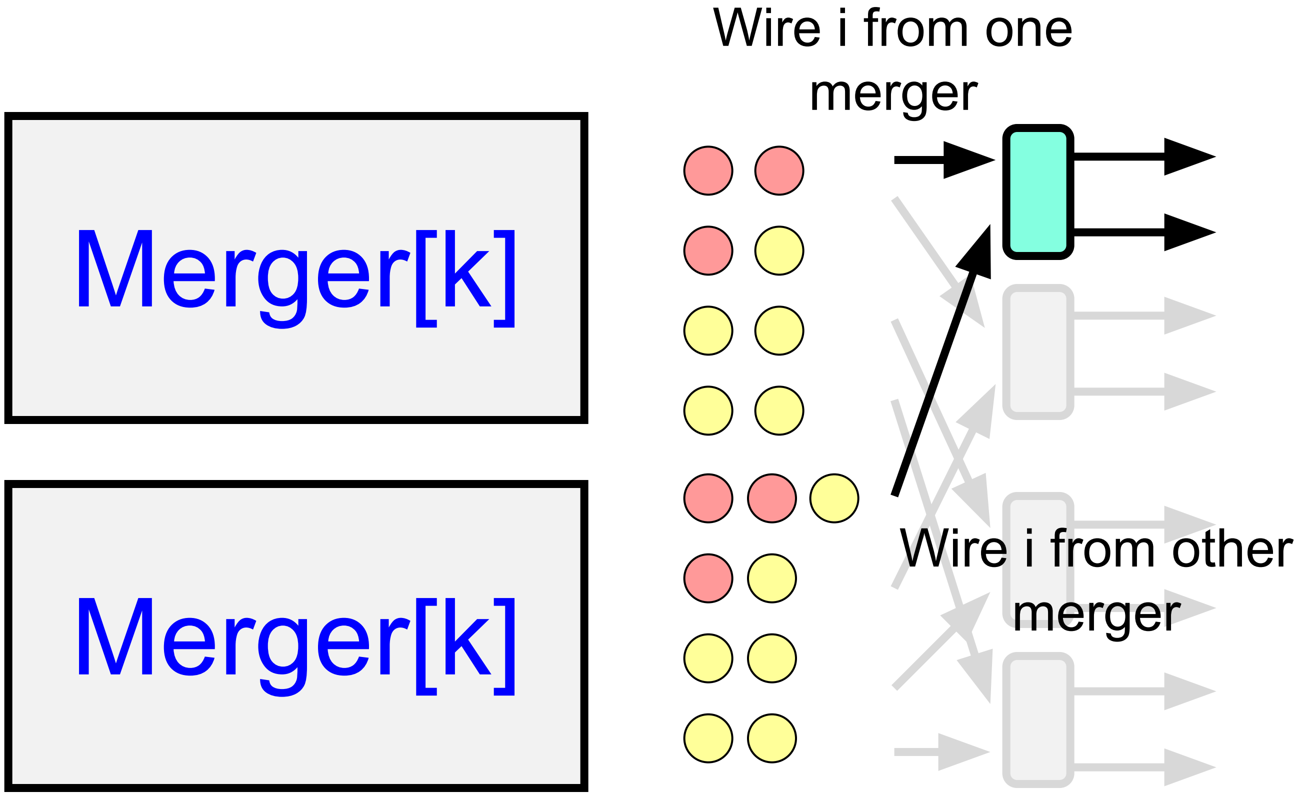
Last Row of Balancers



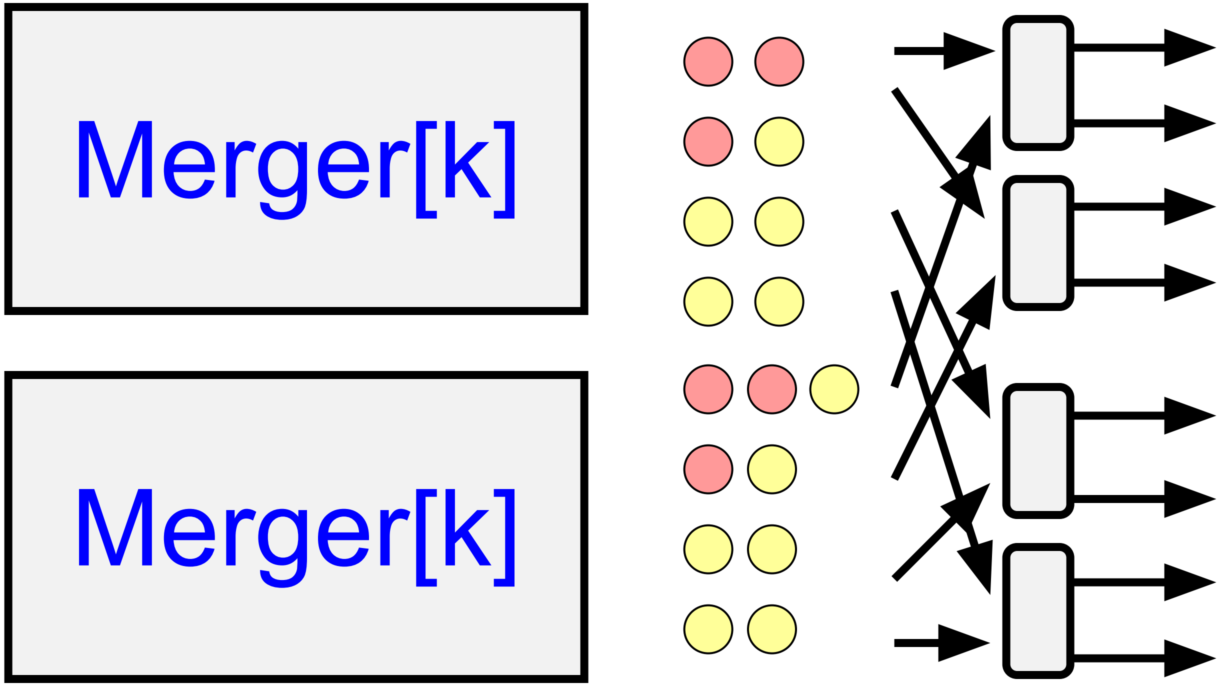
Outputs of Merger[k]

Outputs of last layer

Last Row of Balancers



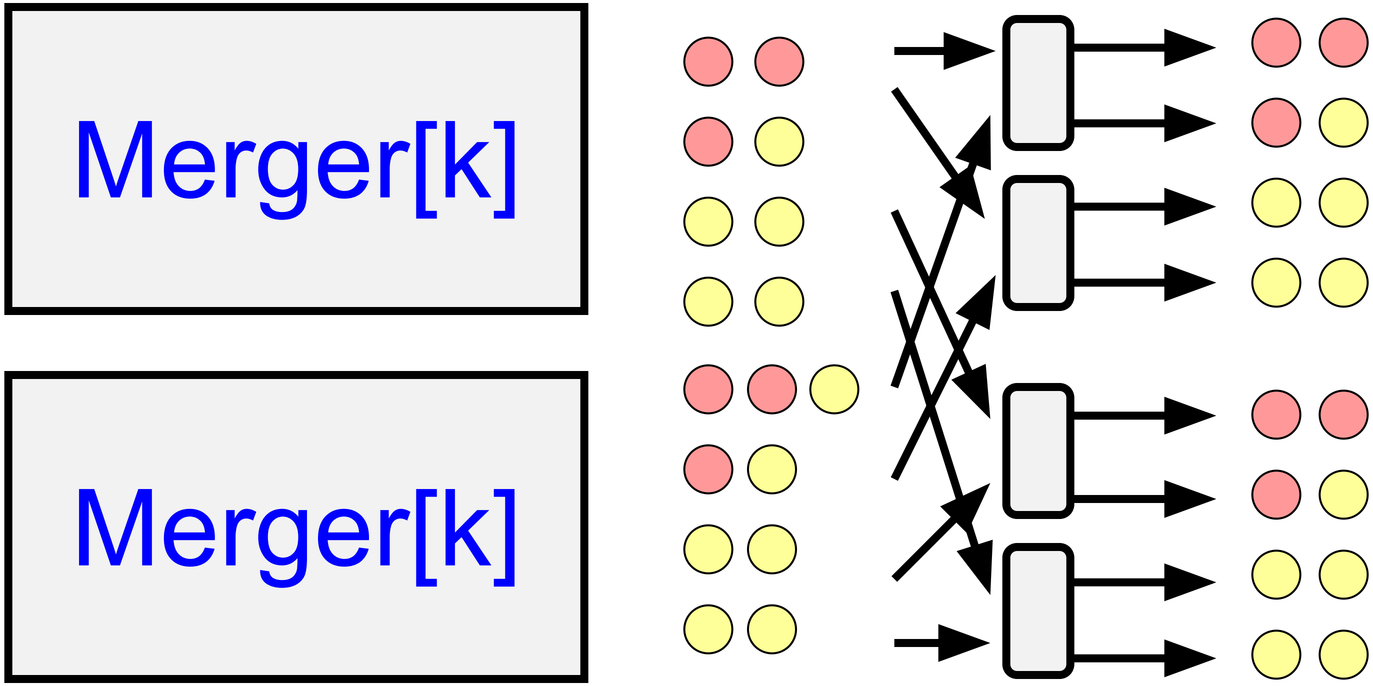
Last Row of Balancers



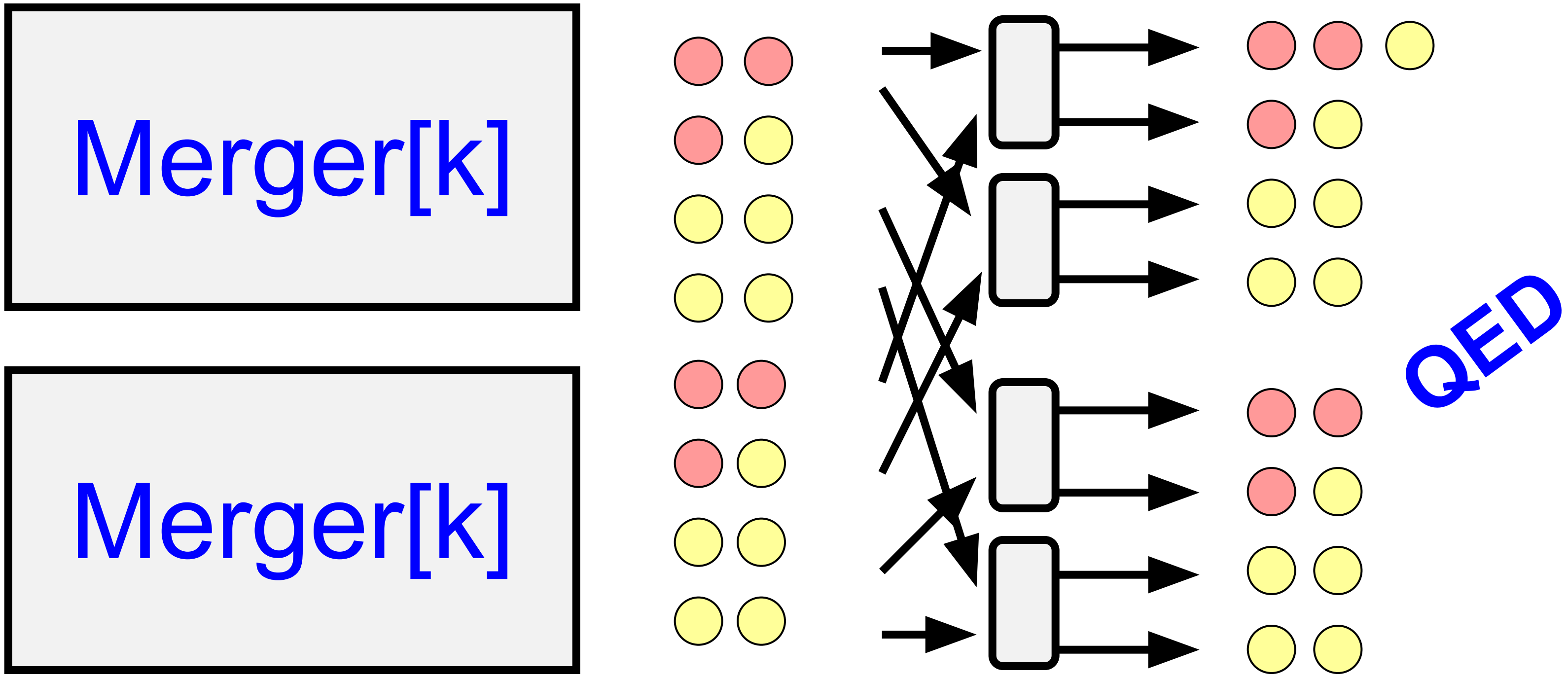
Outputs of Merger[k]

Outputs of last layer

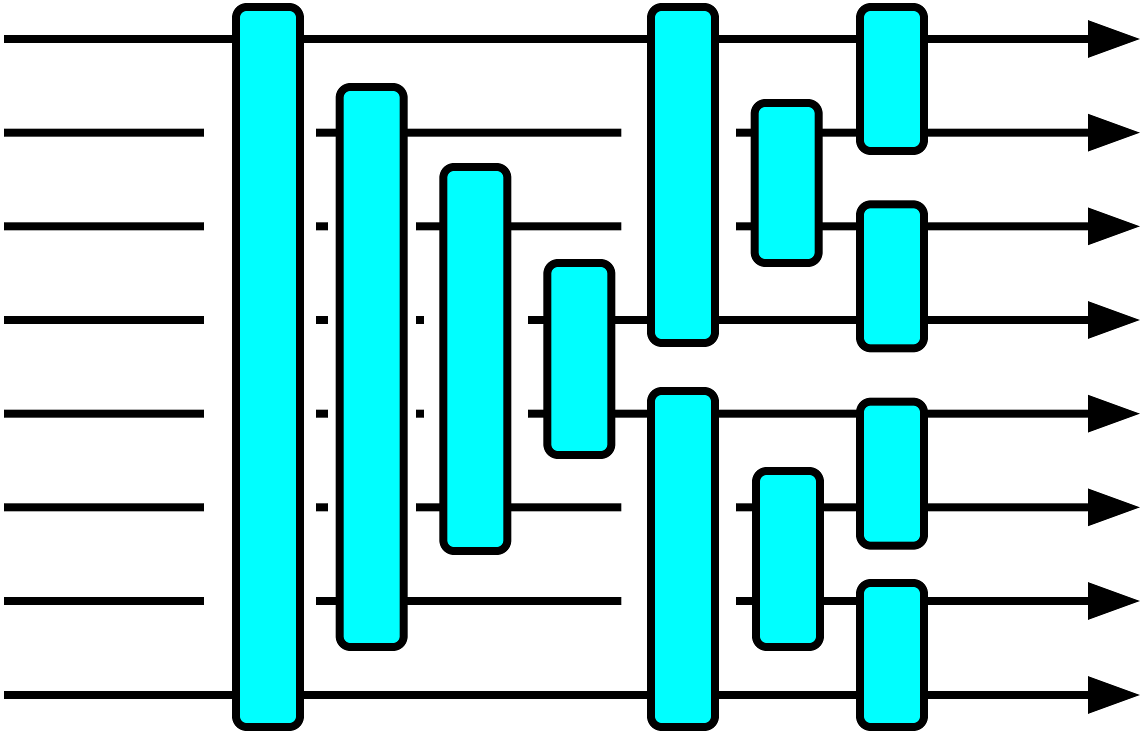
Last Row of Balancers



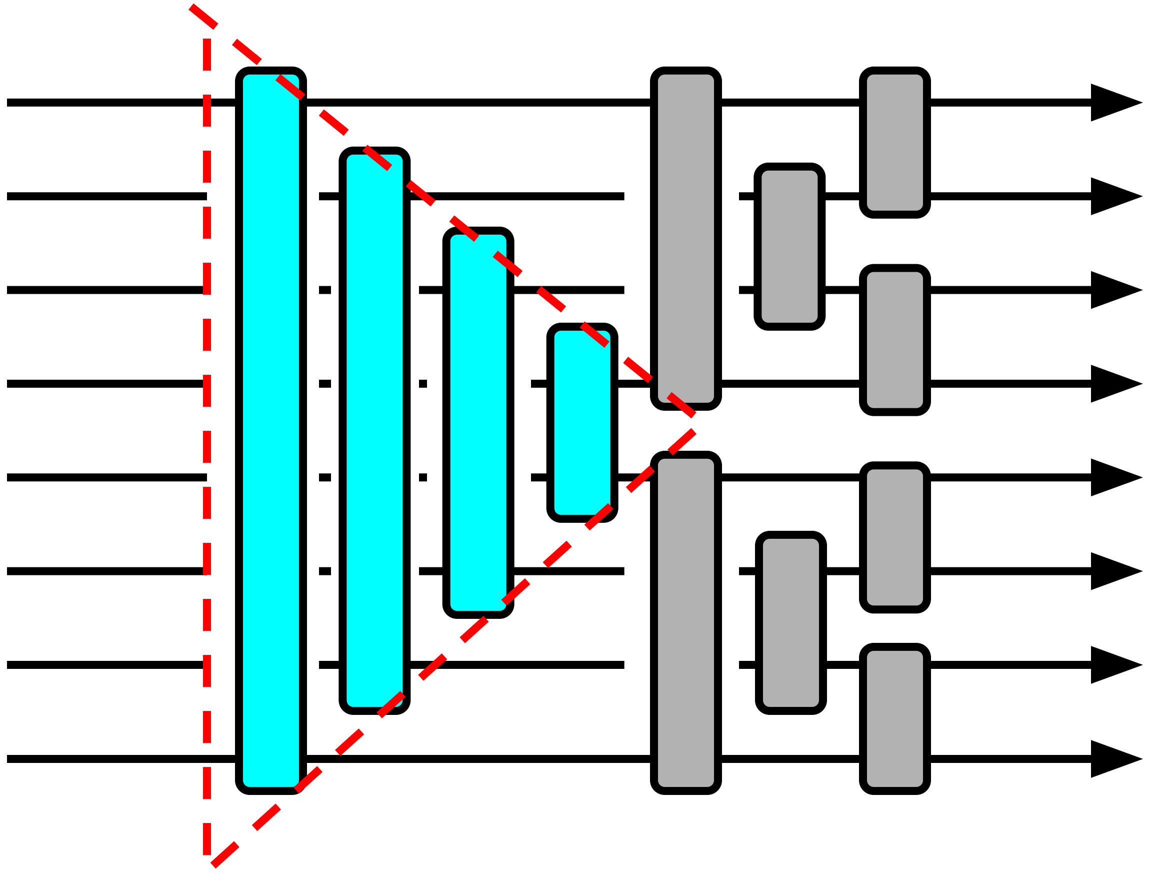
So Counting Networks Count



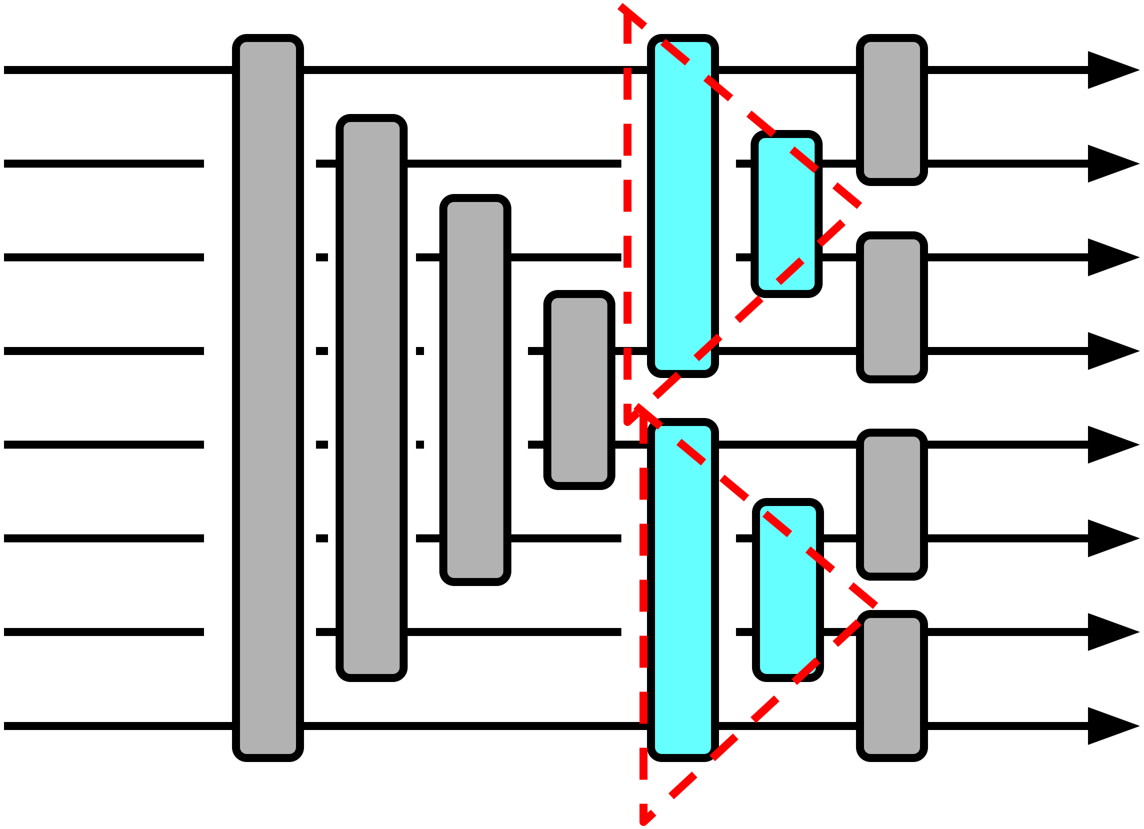
Periodic Network Block



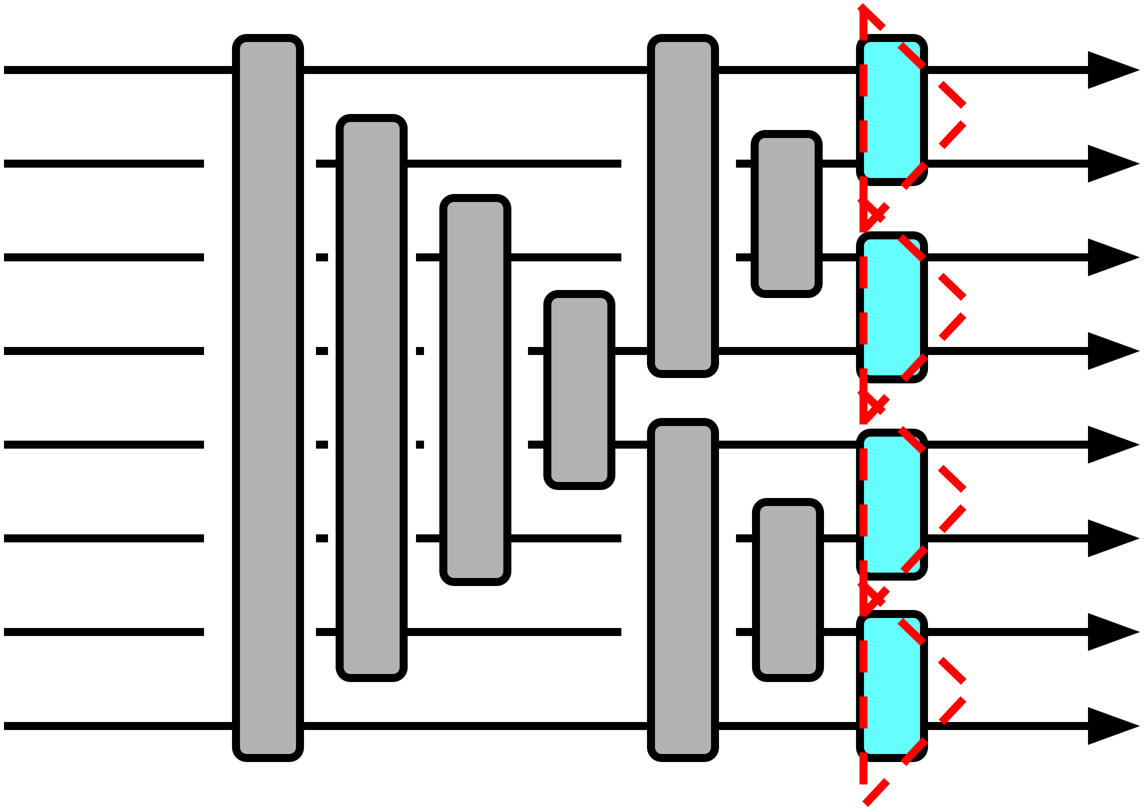
Periodic Network Block



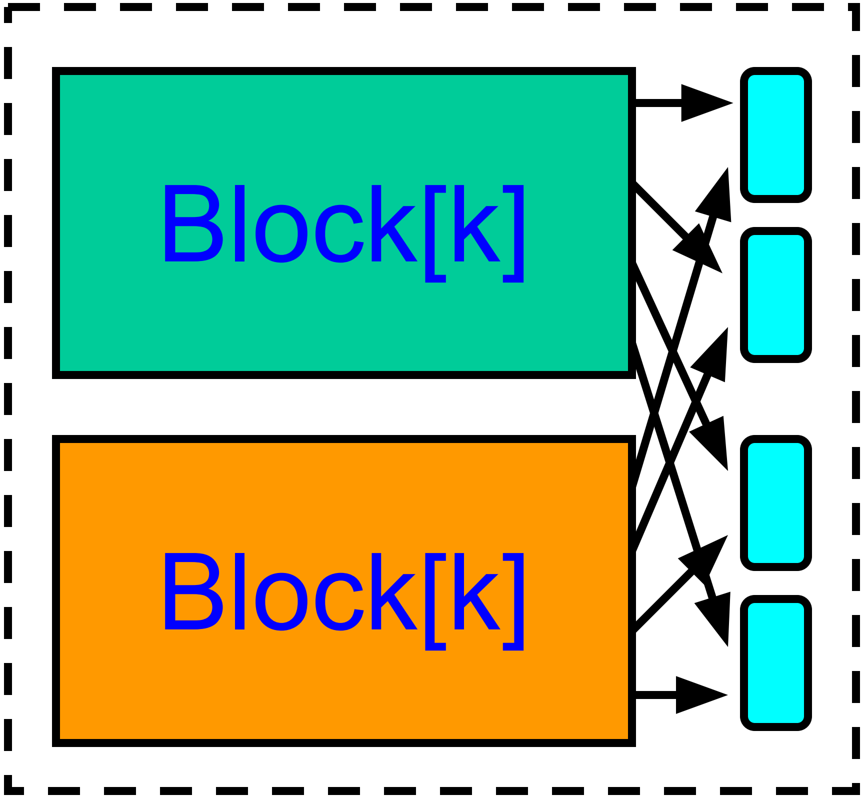
Periodic Network Block



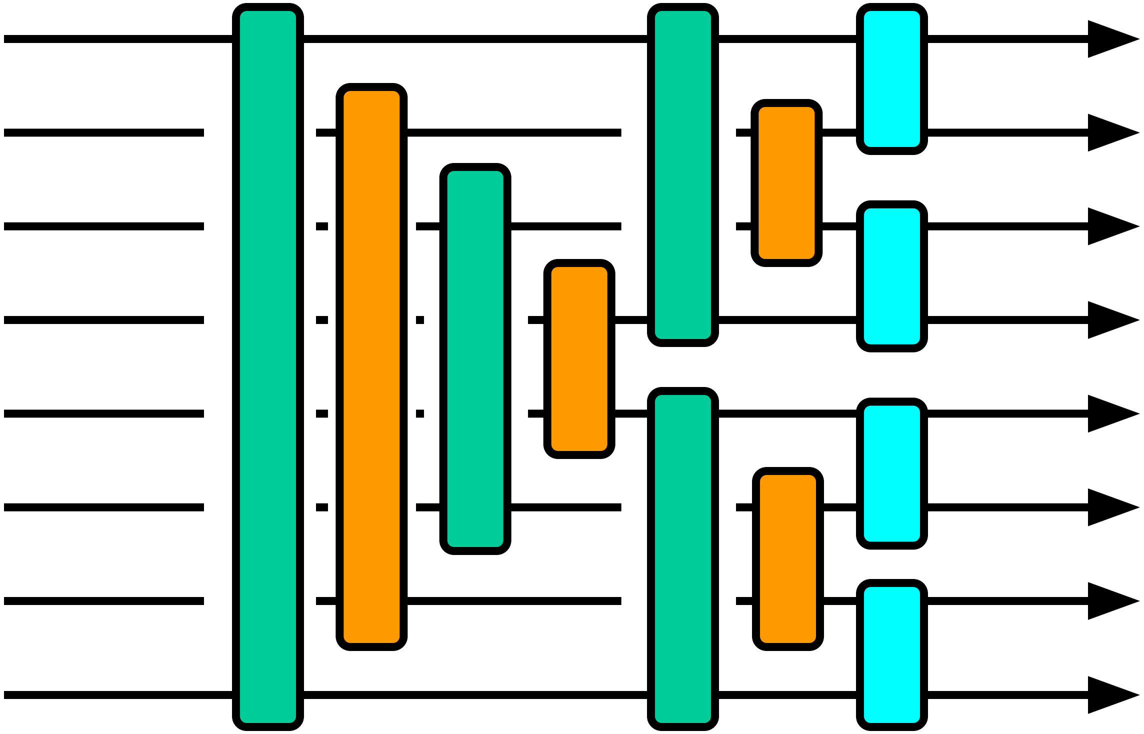
Periodic Network Block



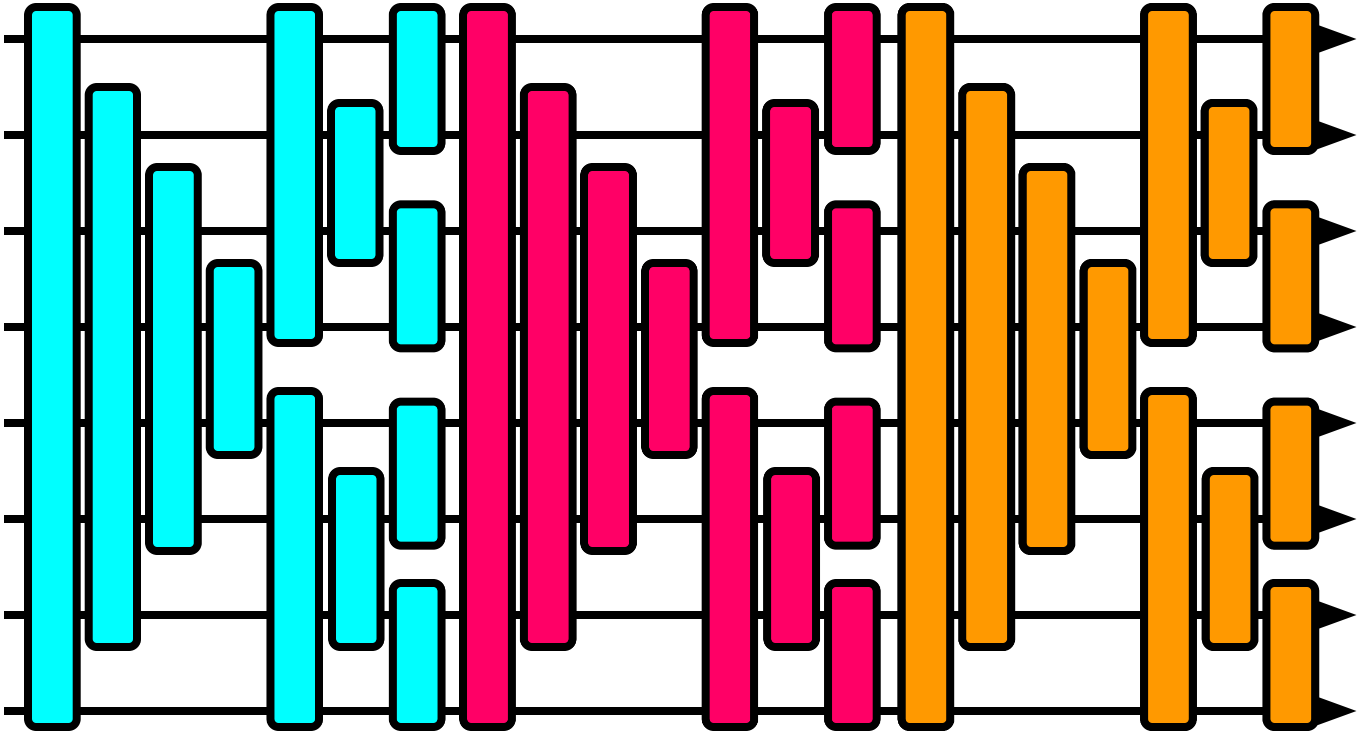
Block[2k] Schematic



Block[2k] Layout



Periodic[8]



Network Depth

- Each block[k] has depth $\log_2 k$
- Need $\log_2 k$ blocks
- Grand total of $(\log_2 k)^2$

Lower Bound on Depth

Theorem: The depth of any width w counting network is at least Ω(log w).

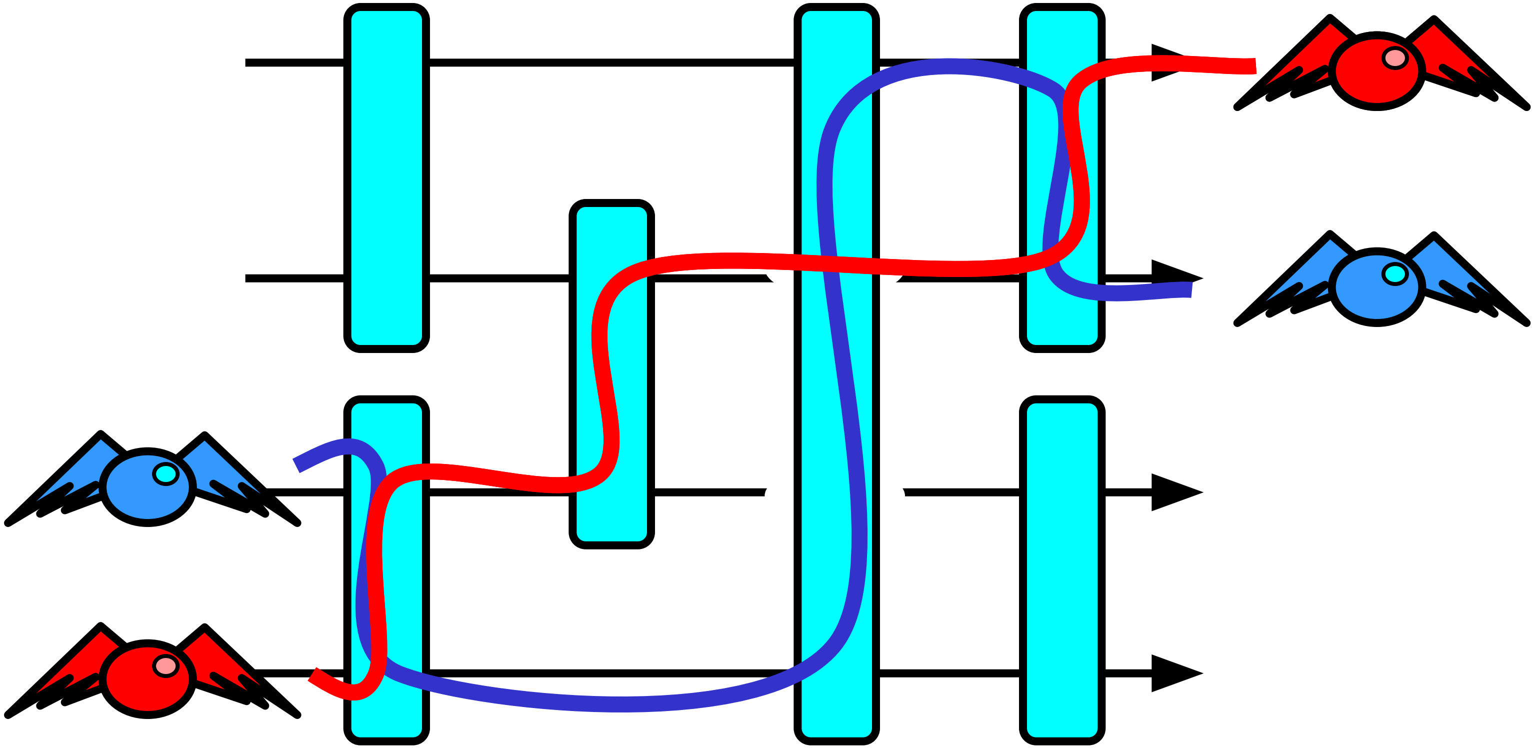
Theorem: there exists a counting network of θ (log w) depth.

Unfortunately, proof is non-constructive and constants in the 1000s.

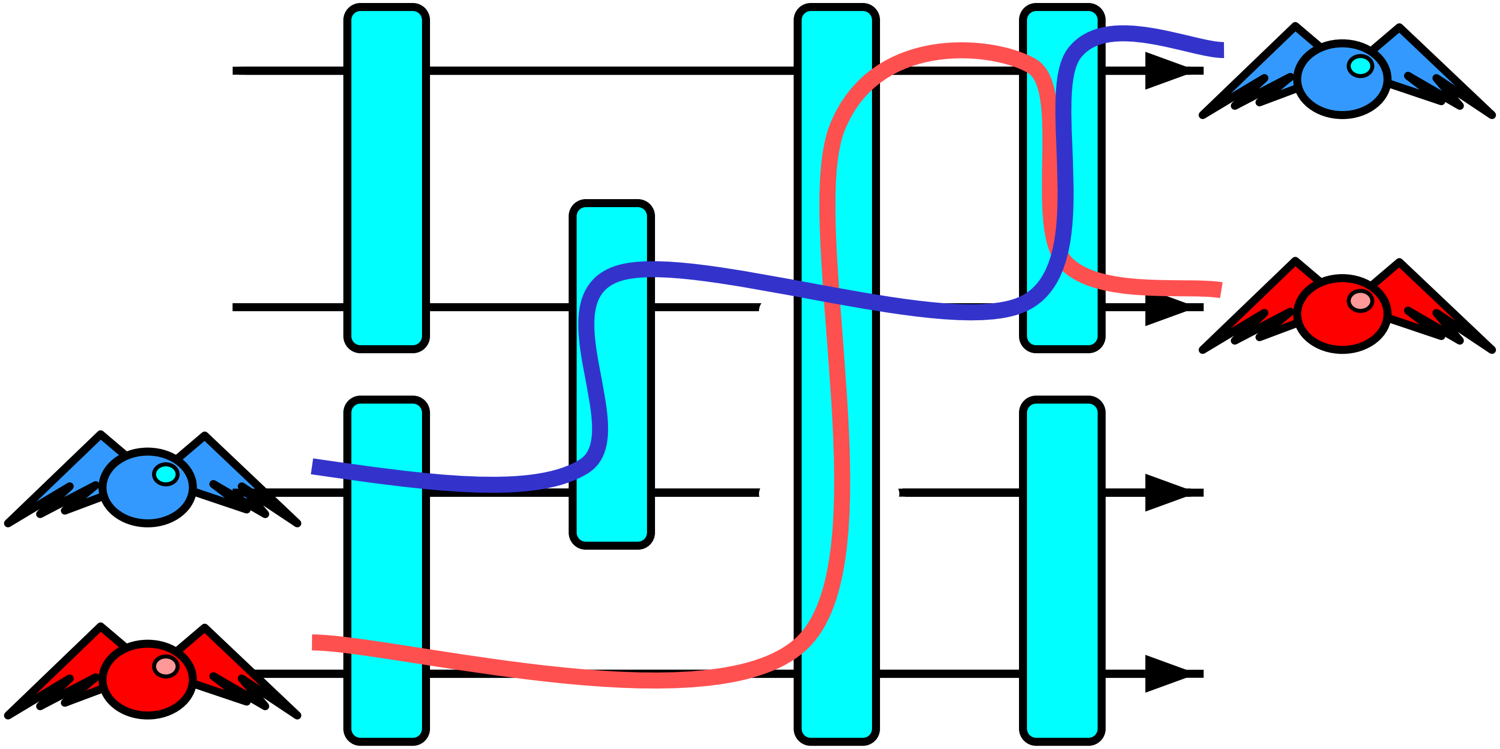
Sequential Theorem

- If a balancing network counts
  - Sequentially, meaning that
  - Tokens traverse one at a time
- Then it counts
  - Even if tokens traverse concurrently

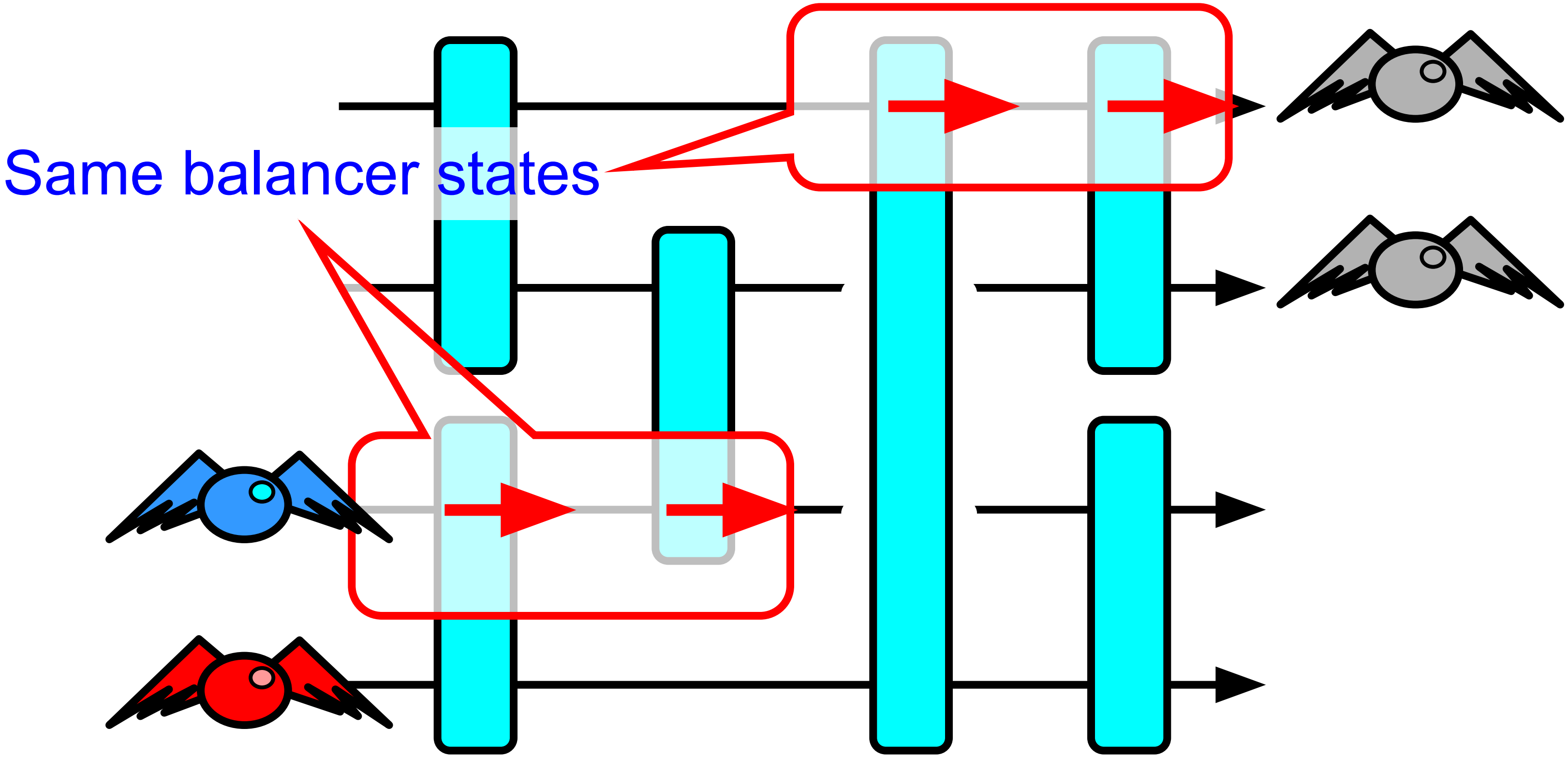
Red First, Blue Second



Blue First, Red Second

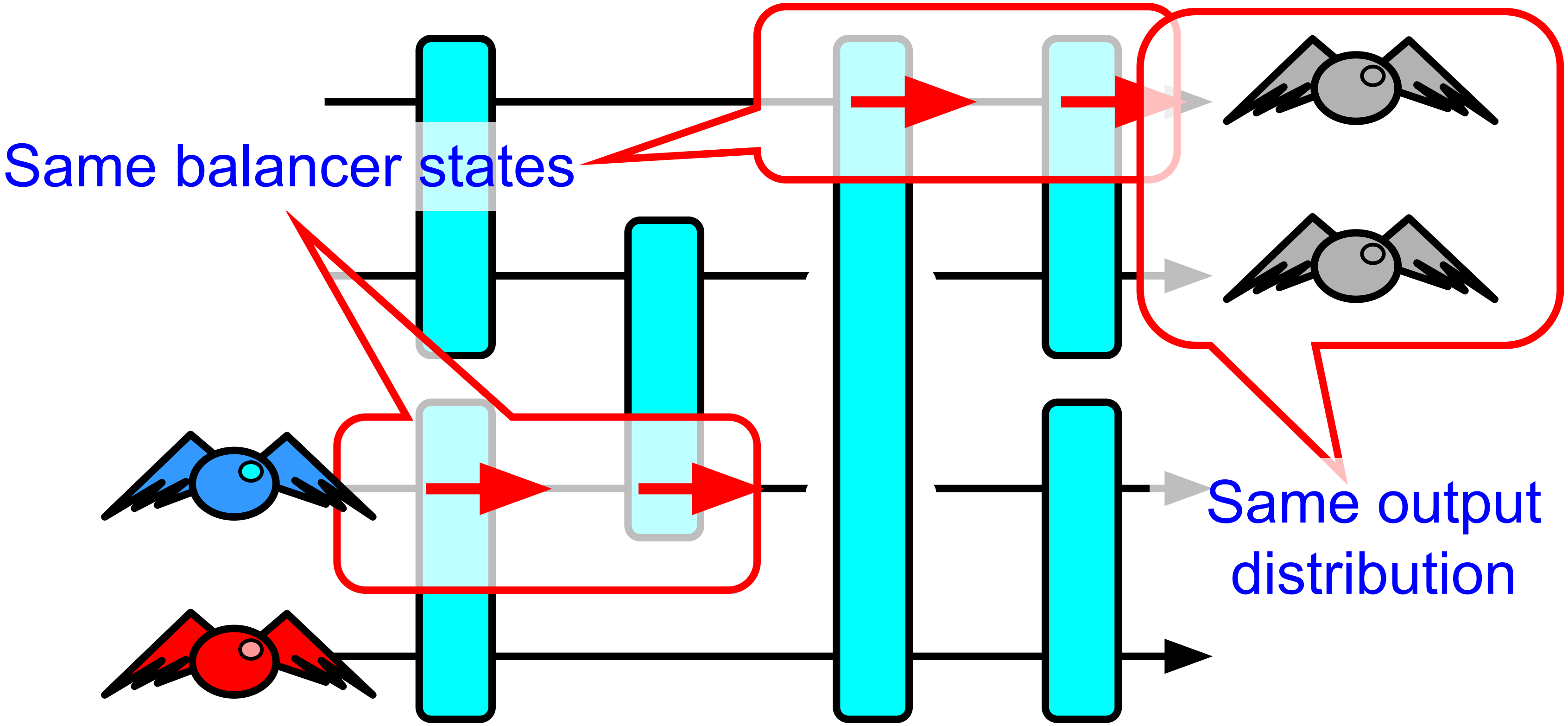


Either Way



Same balancer states

Order Doesn't Matter



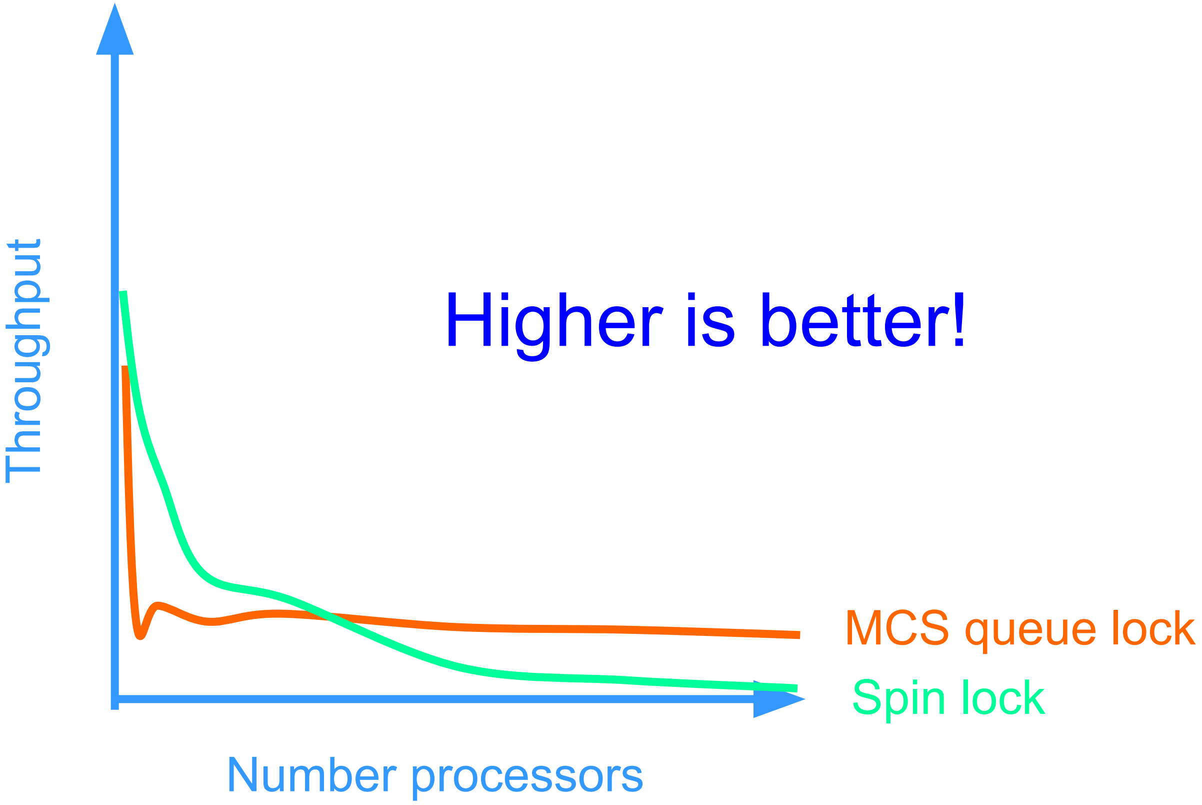
Same balancer states

Same output distribution

Index Distribution Benchmark

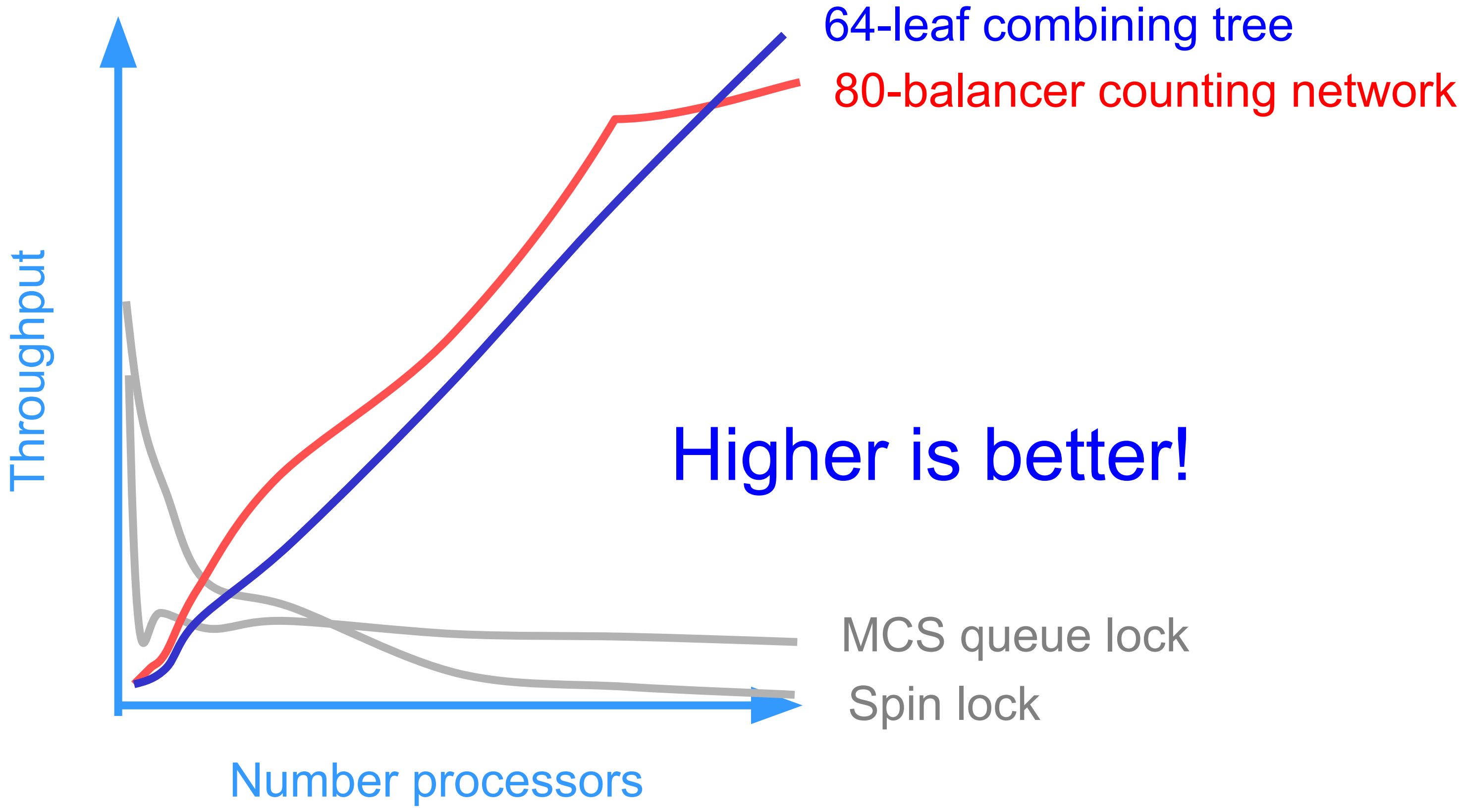
```
void indexBench(int iters, int work) {
 while (int i = 0 < iters) {
  i = fetch&inc();
  Thread.sleep(random() % work);
 }
}
```

Performance (Simulated)

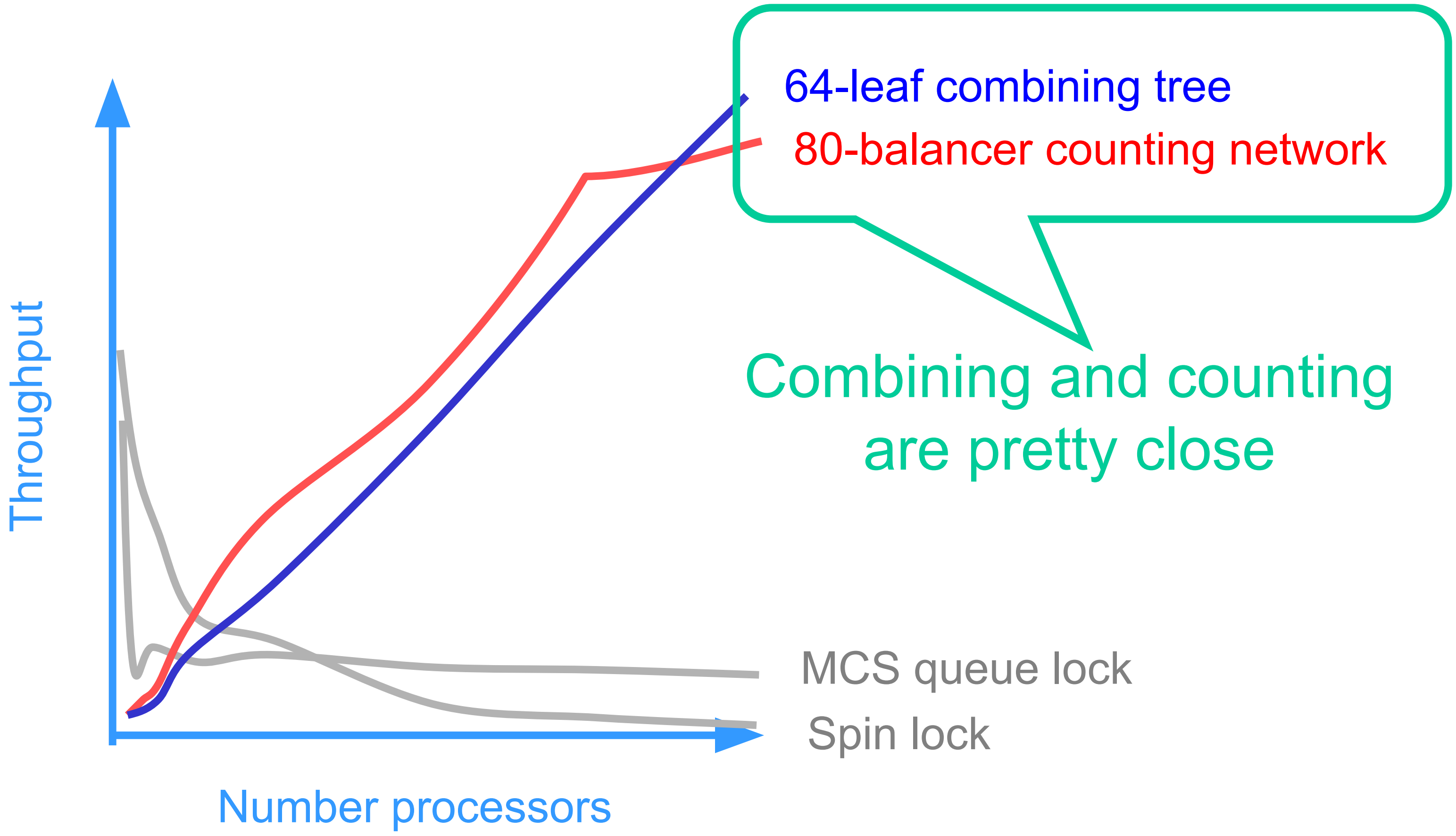


* All graphs taken from Herlihy,Lim,Shavit, copyright ACM.

Performance (Simulated)

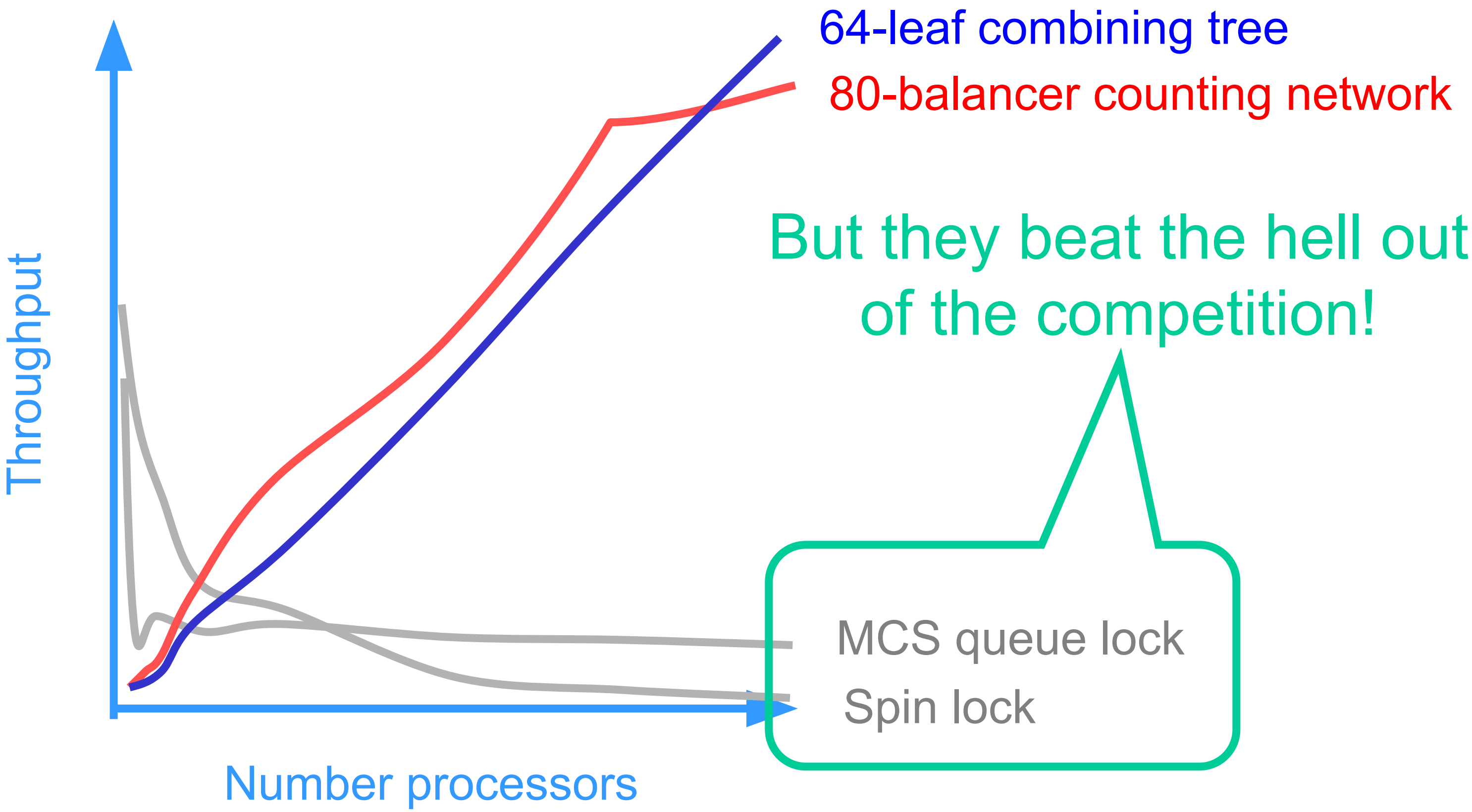


Performance (Simulated)



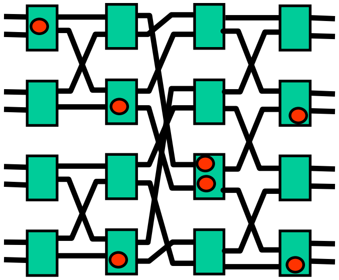
Combining and counting are pretty close

Performance (Simulated)

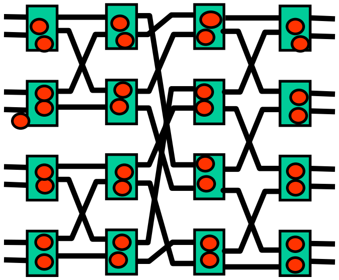


But they beat the hell out of the competition!

Saturation and Performance

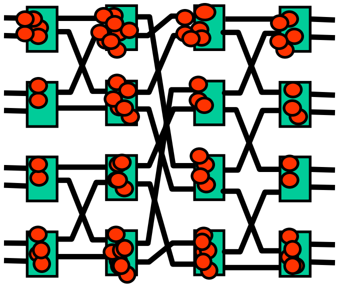


Undersaturated $P < w \log w$



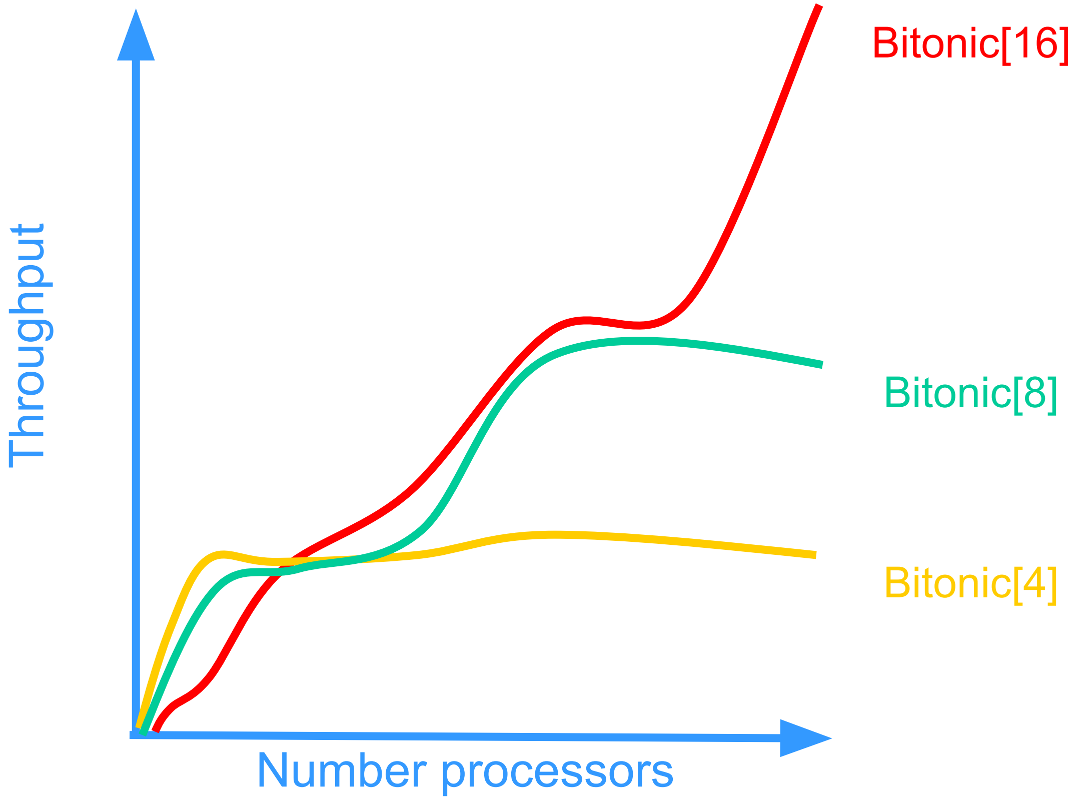
Optimal performance

Saturated $P = w \log w$

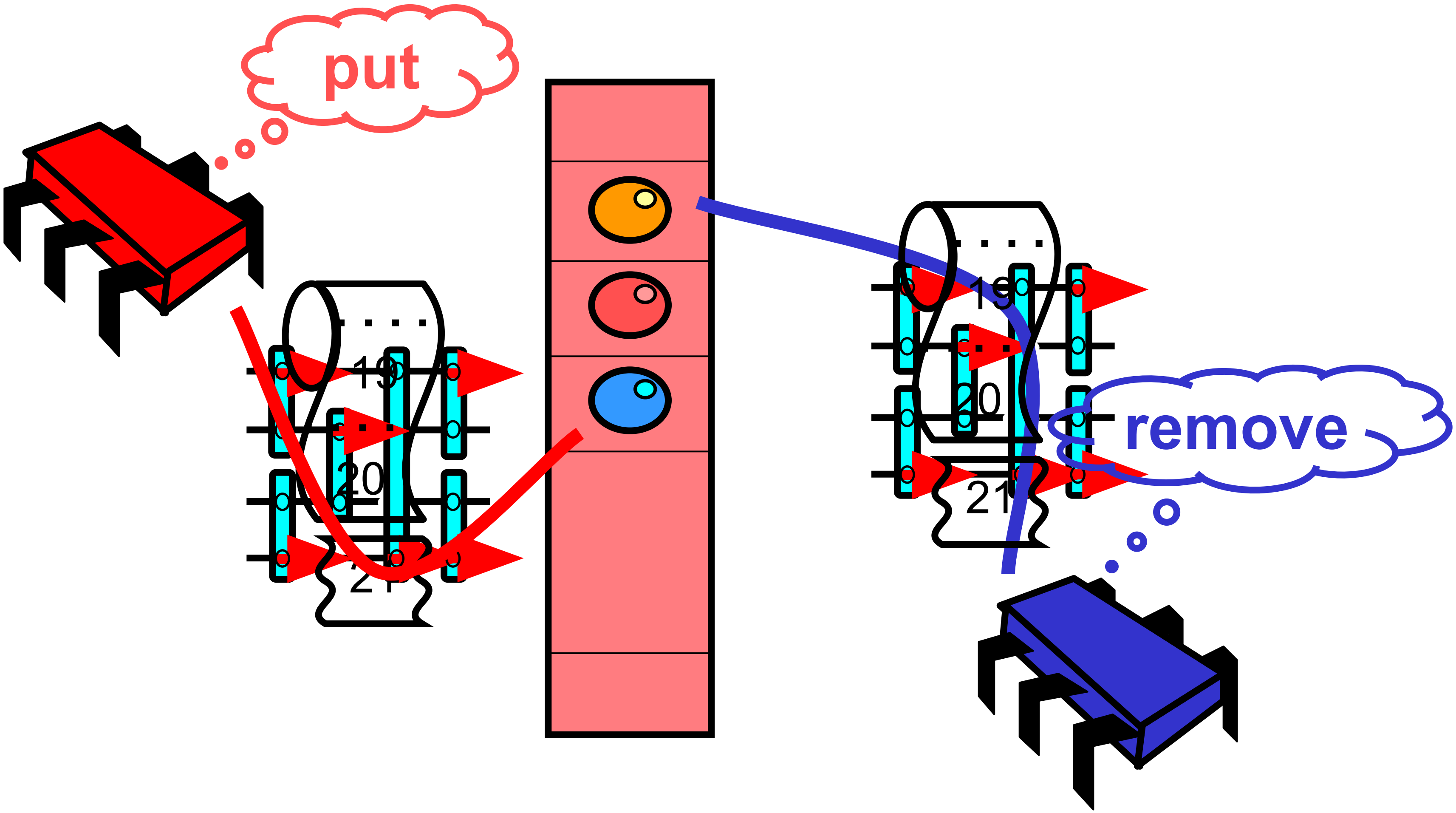


Oversaturated $P > w \log w$

Throughput vs. Size



Shared Pool



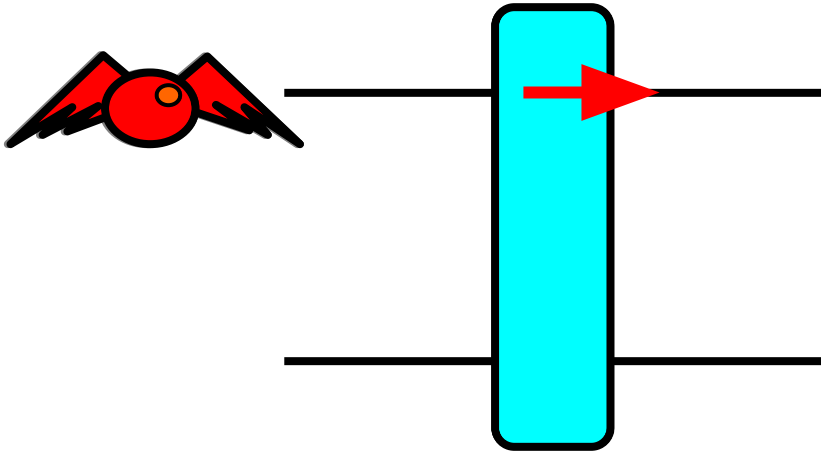
What About

- Decrements
- Adding arbitrary values
- Other operations
  - Multiplication
  - Vector addition
  - Horoscope casting …

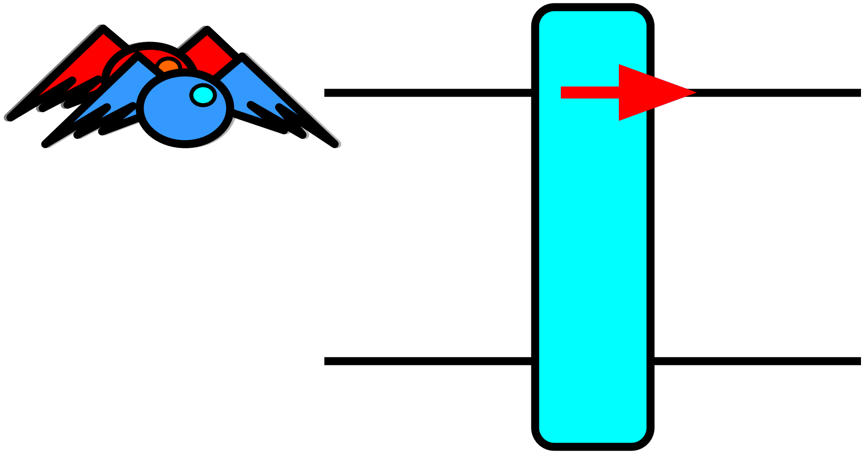
First Step

- Can we decrement as well as increment?
- What goes up, must come down …

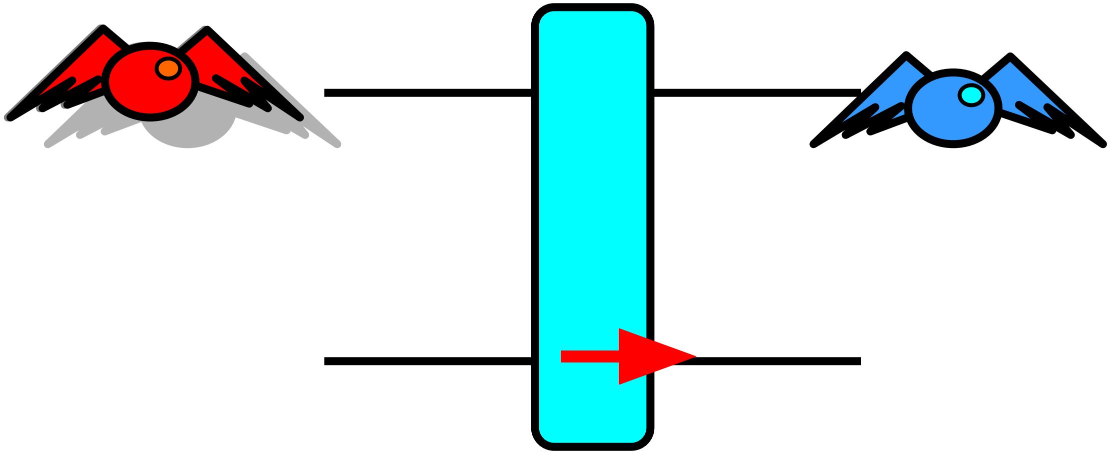
Anti-Tokens



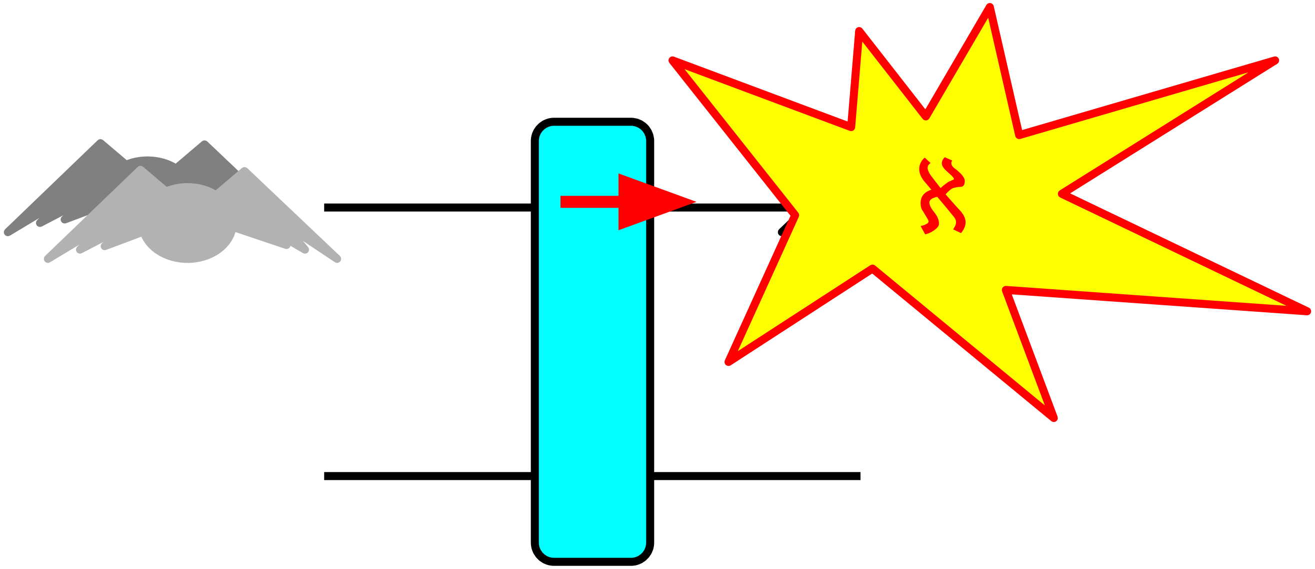
Tokens & Anti-Tokens Cancel



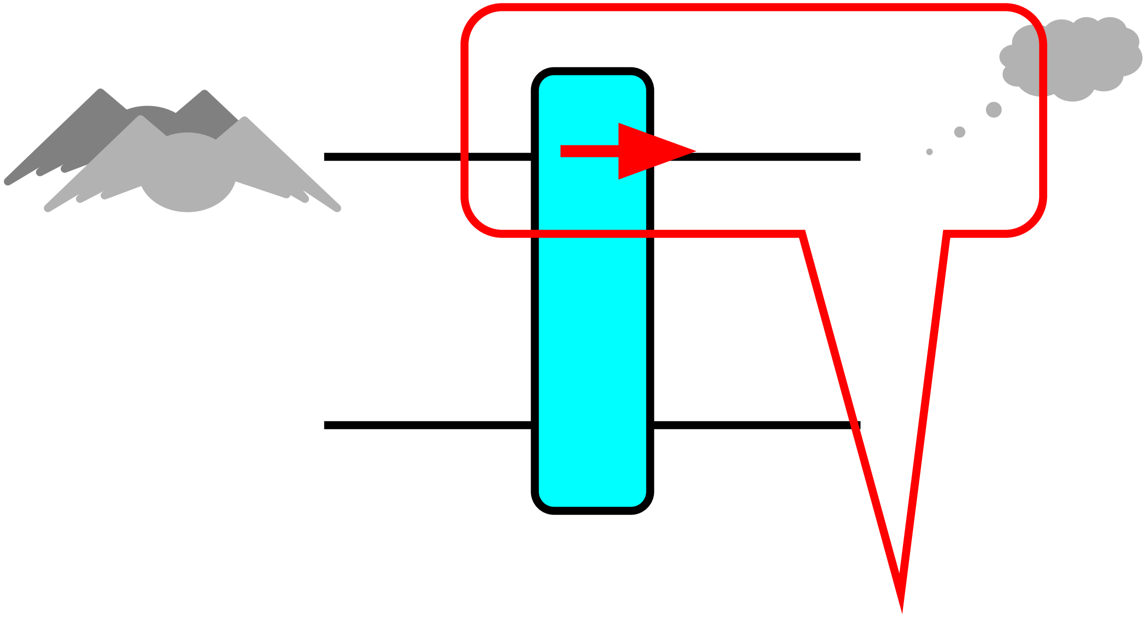
Tokens & Anti-Tokens Cancel



Tokens & Anti-Tokens Cancel



Tokens & Anti-Tokens Cancel



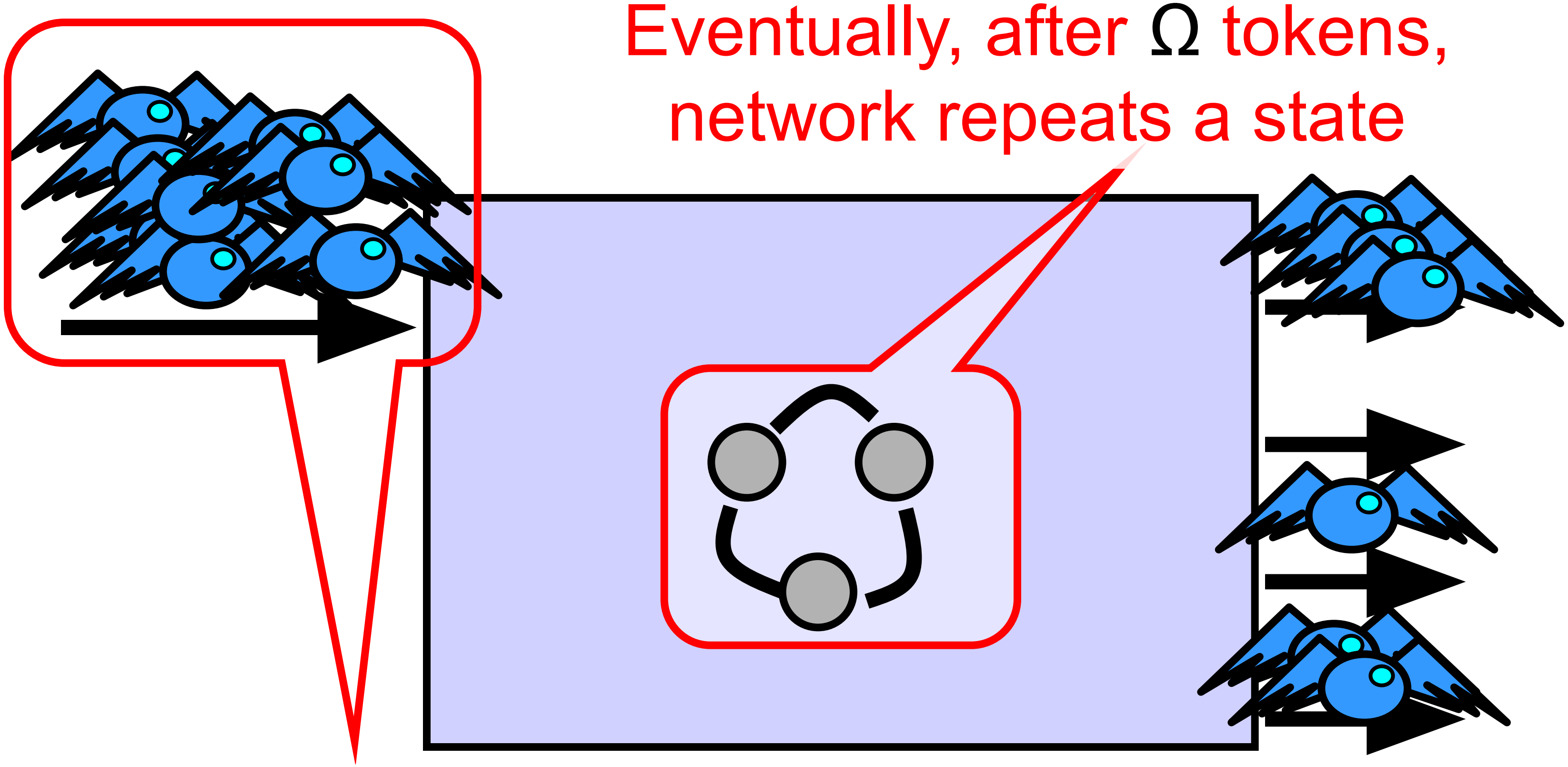
As if nothing happened

Tokens vs Antitokens

- Tokens
  - read balancer
  - flip
  - proceed

- Antitokens
  - flip balancer
  - read
  - proceed

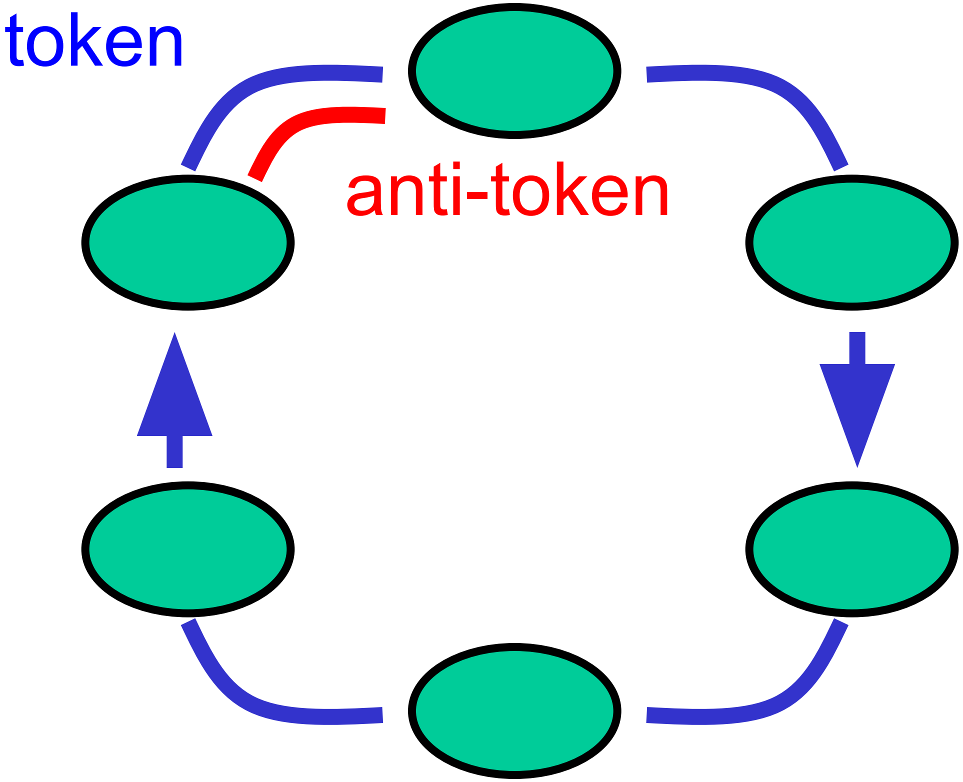
Pumping Lemma



Eventually, after Ω tokens, network repeats a state

Keep pumping tokens through one wire

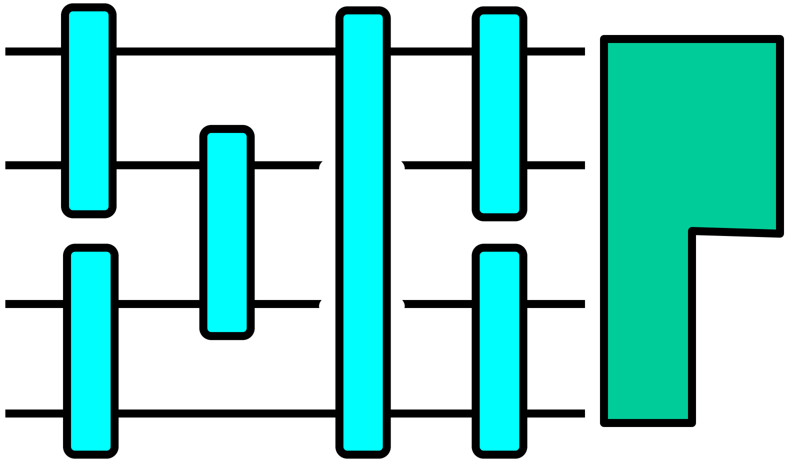
Anti-Token Effect



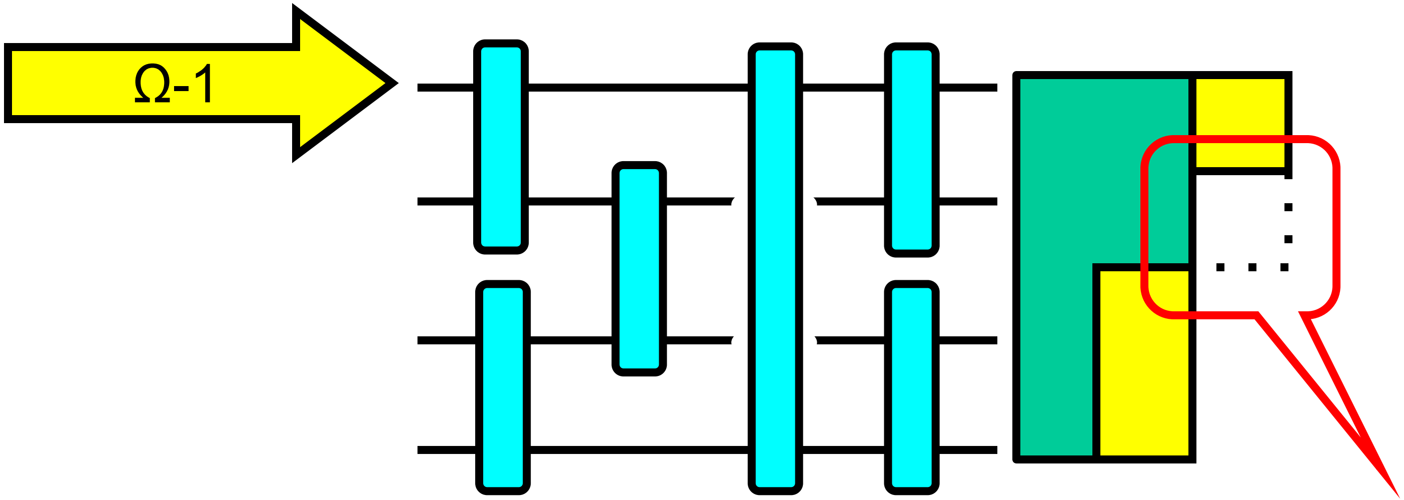
Observation

- Each anti-token on wire i
  - Has same effect as Ω-1 tokens on wire i
  - So network still in legal state
- Moreover, network width w divides Ω
  - So Ω-1 tokens

Before Antitoken

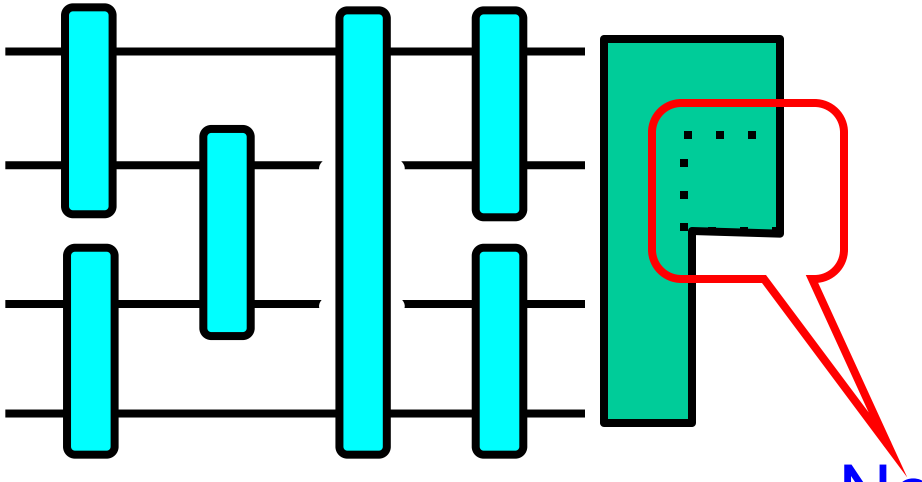


Balancer states as if …



Ω-1 is one brick shy of a load

Post Antitoken



Next token shows up here

Implication

- Counting networks with
  - Tokens (+1)
  - Anti-tokens (-1)
- Give
  - Highly concurrent
  - Low contention
- getAndIncrement + getAndDecrement methods

QED

Adding Networks

- Combining trees implement
  - Fetch&add
  - Add any number, not just 1
- What about counting networks?

Fetch-and-add

- Beyond getAndIncrement + getAndDecrement
- What about getAndAdd(x)?
  - Atomically returns prior value
  - And adds x to value?
- Not to mention
  - getAndMultiply
  - getAndFourierTransform?

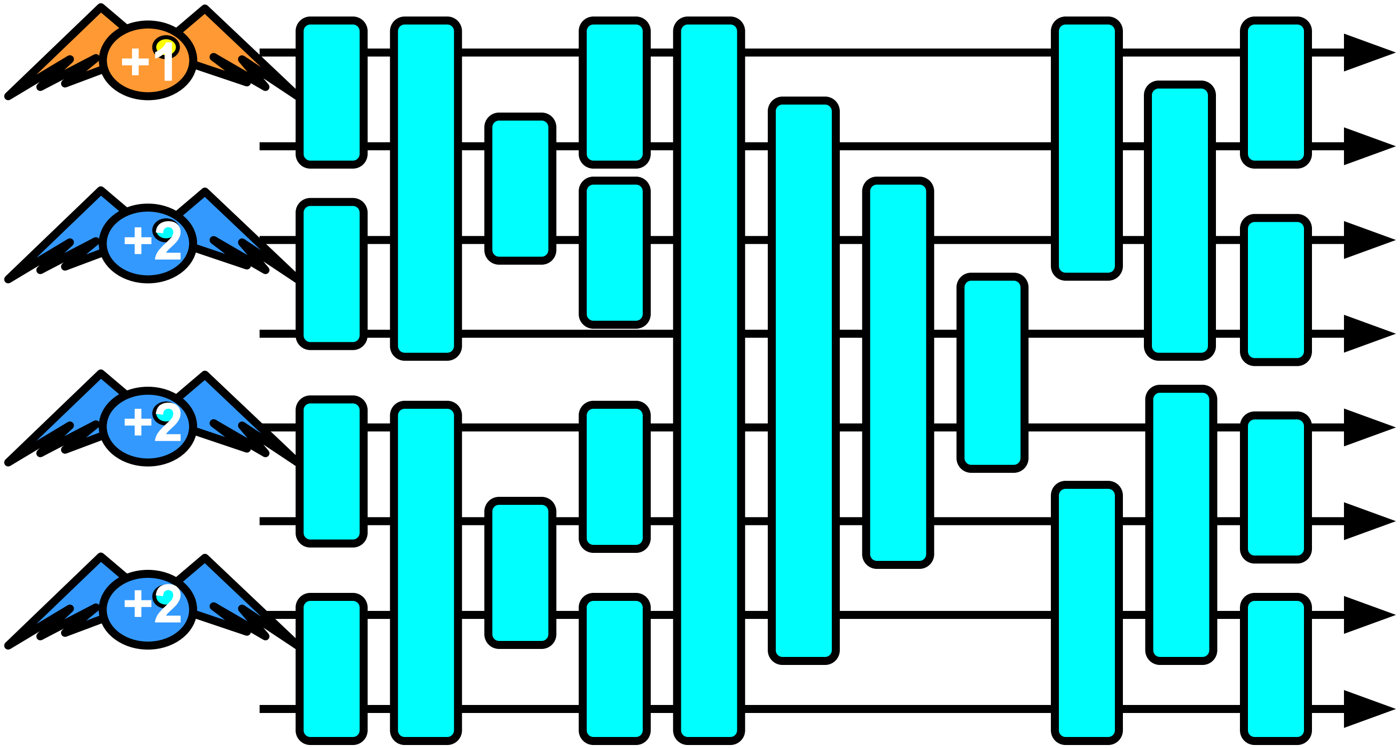
Bad News

- If an adding network
  - Supports n concurrent tokens
- Then every token must traverse
  - At least n-1 balancers
  - In sequential executions

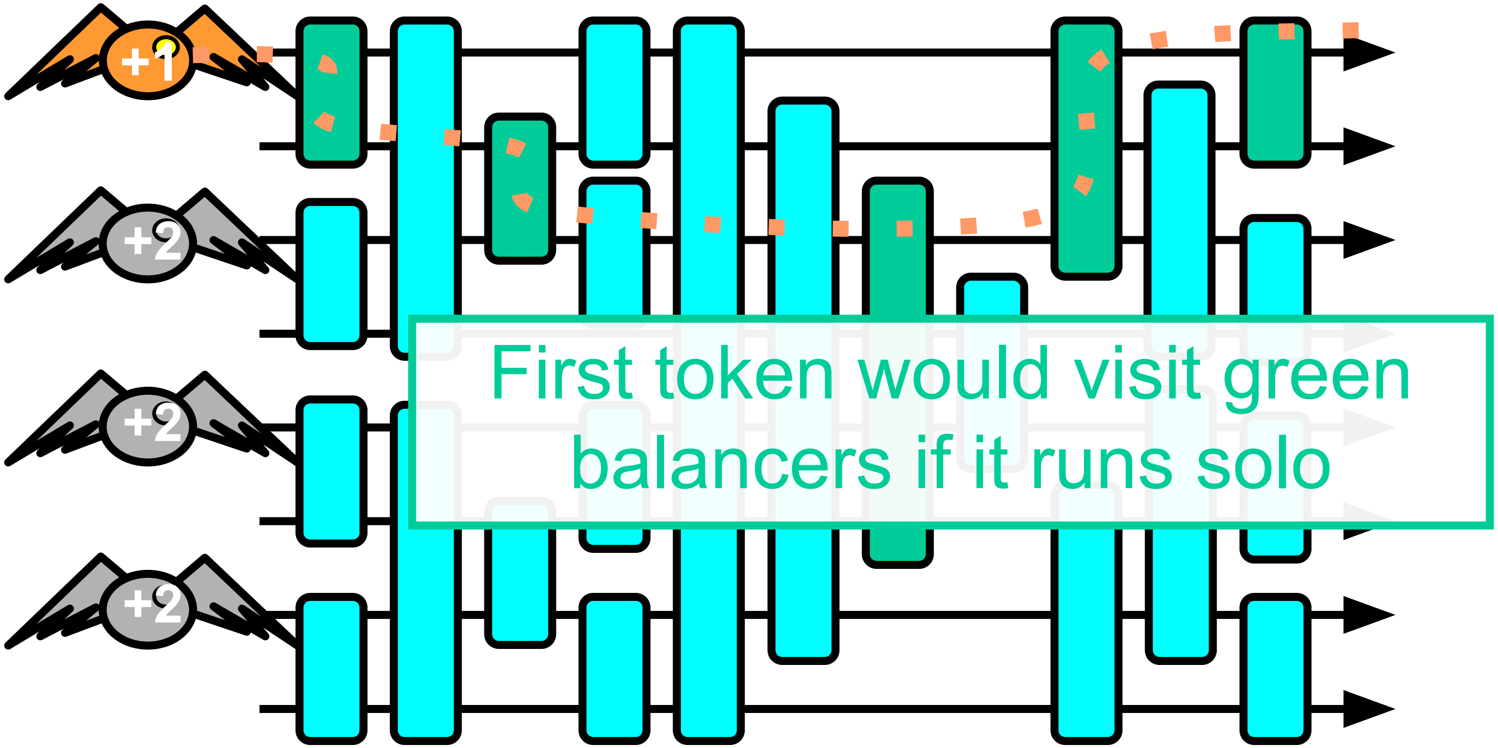
Uh-Oh

- Adding network size depends on n
  - Like combining trees
  - Unlike counting networks
- High latency
  - Depth linear in n
  - Not logarithmic in w

Generic Counting Network



First Token

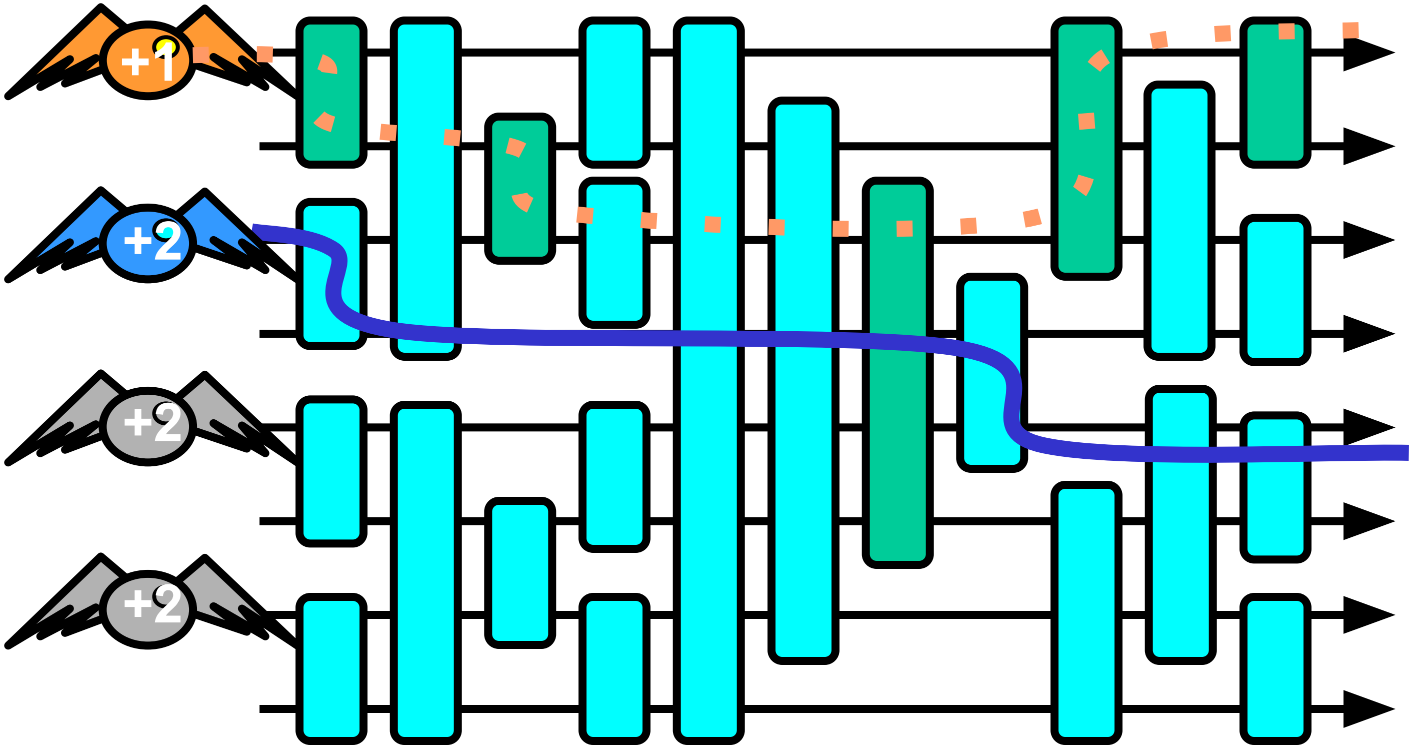


First token would visit green balancers if it runs solo

Claim

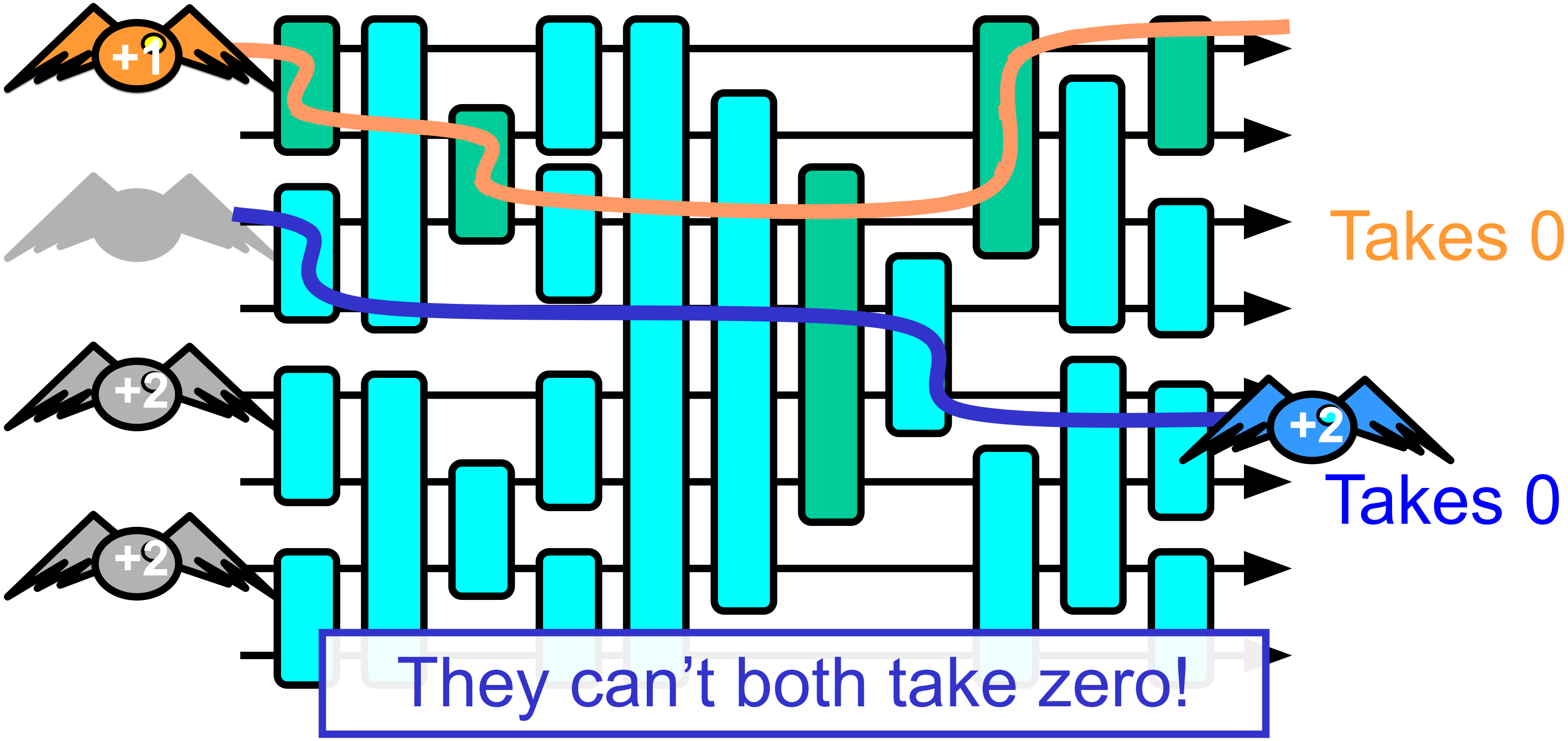
- Look at path of +1 token
- All other +2 tokens must visit some balancer on +1 token's path

Second Token



Takes 0

Second Token



They can't both take zero!

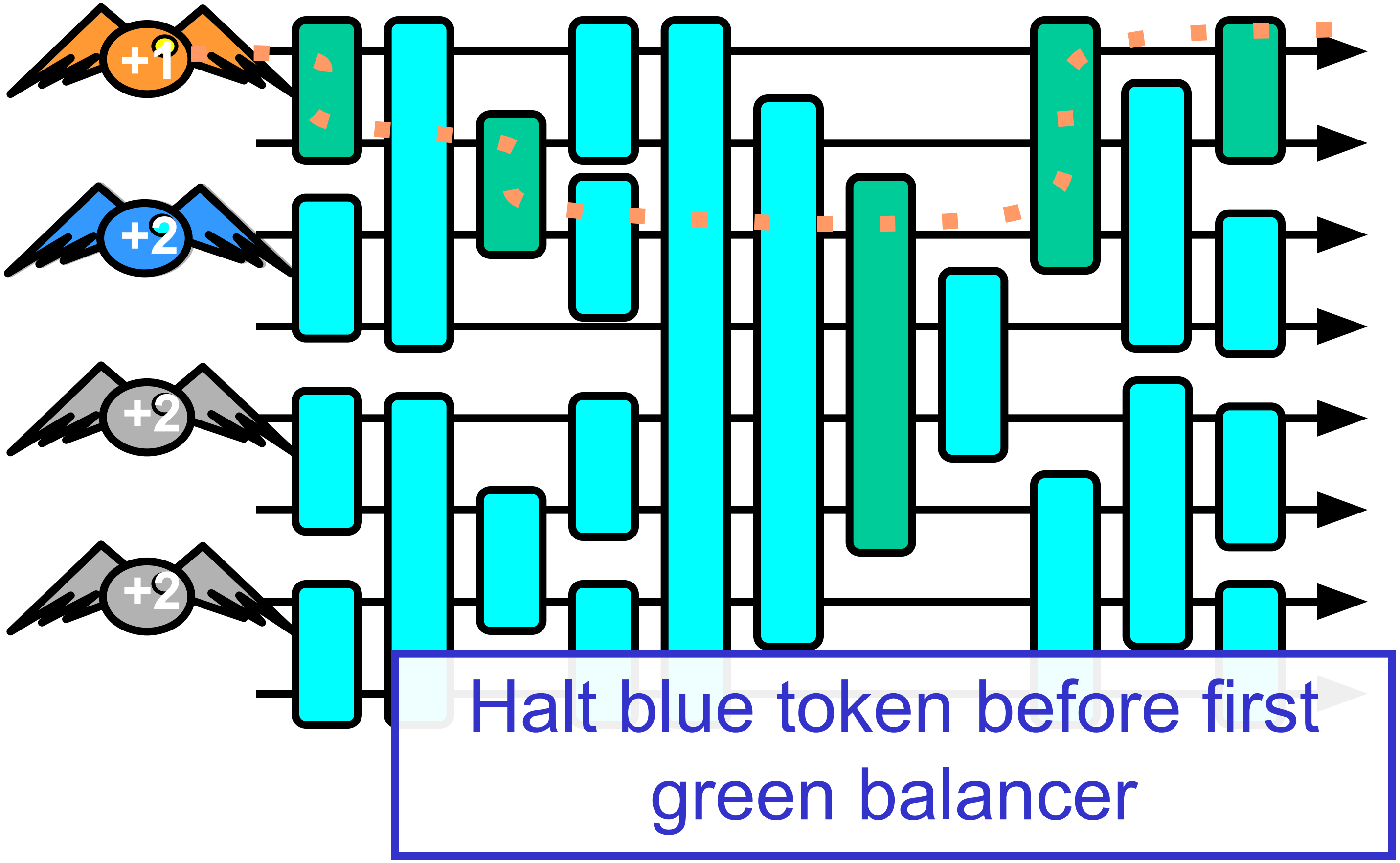
If Second avoids First’s Path

- Second token
  - Doesn’t observe first
  - First hasn’t run
  - Chooses 0
- First token
  - Doesn’t observe second
  - Disjoint paths
  - Chooses 0

If Second avoids First’s Path

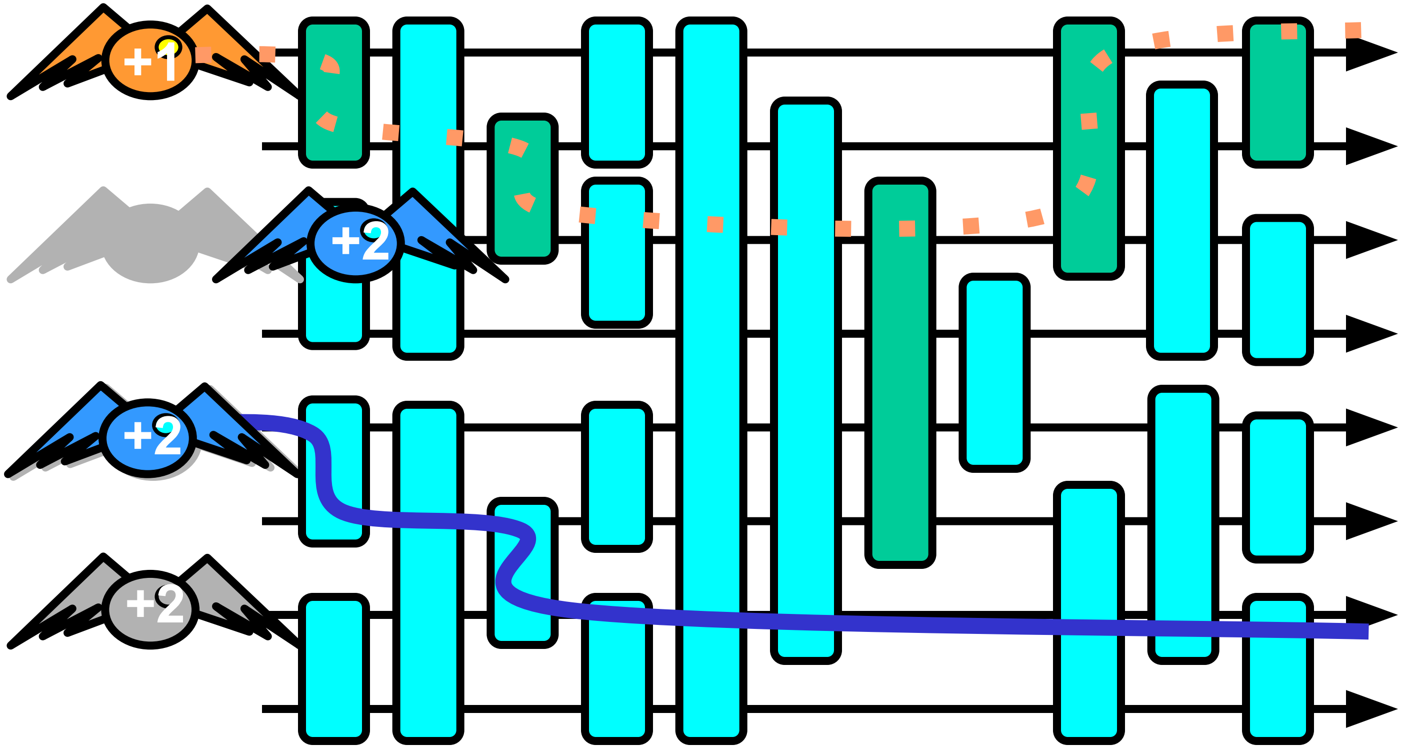
- Because +1 token chooses 0
  - It must be ordered first
  - So +2 token ordered second
  - So +2 token should return 1
- Something’s wrong!

Second Token



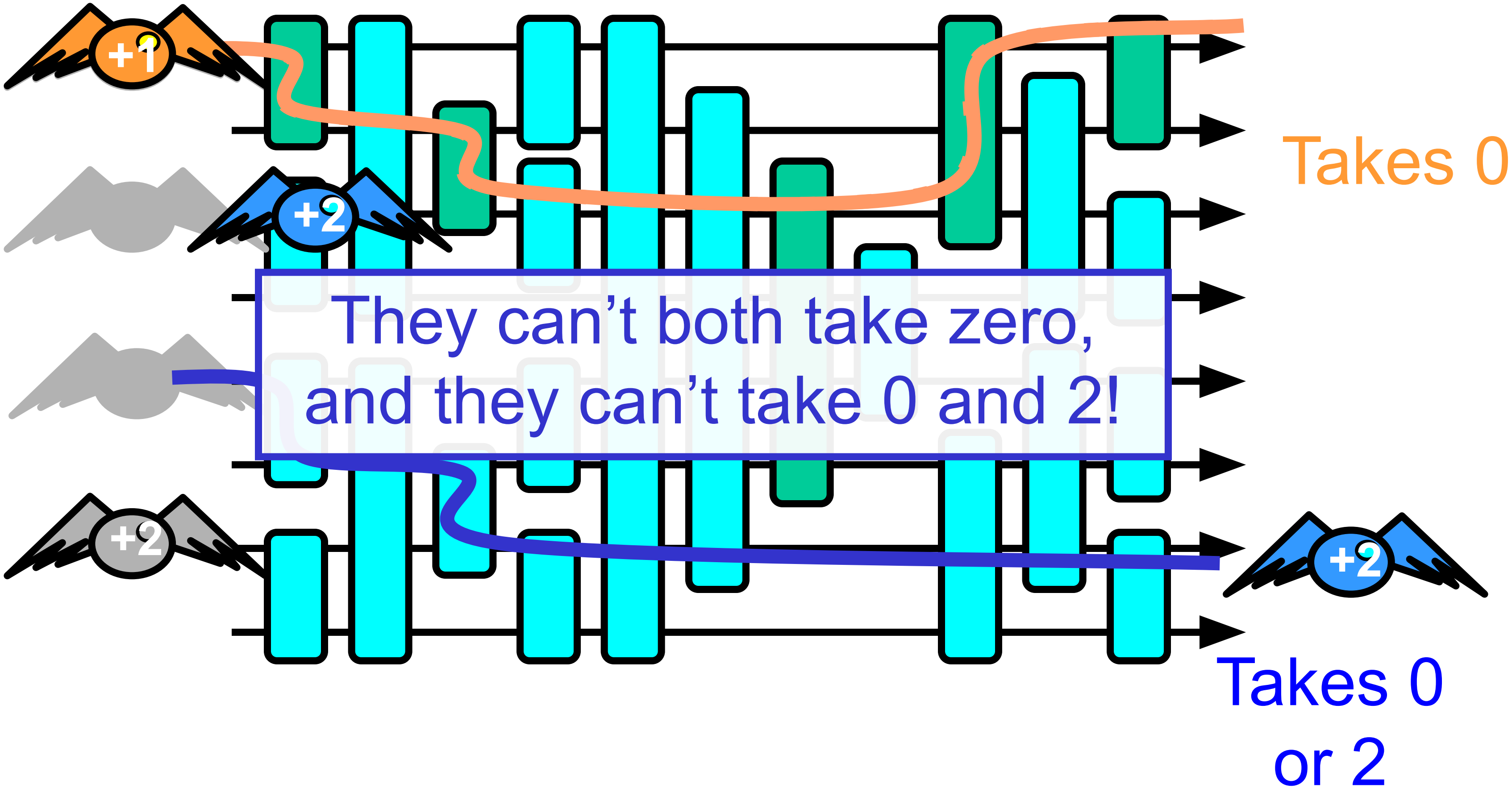
Halt blue token before first green balancer

Third Token



Takes 0 or 2

Third Token



They can’t both take zero, and they can’t take 0 and 2!

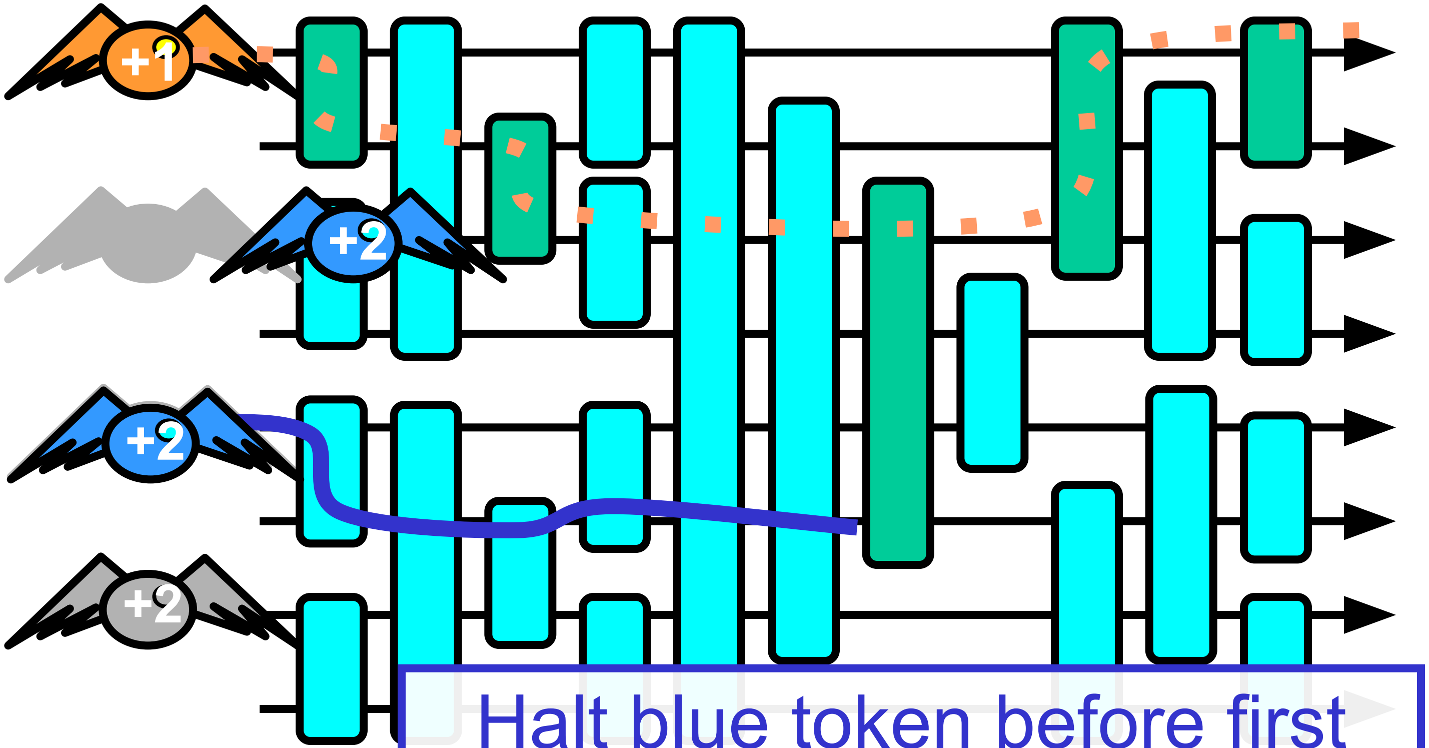
First,Second, & Third Tokens must be Ordered

- Third (+2) token
  - Did not observe +1 token
  - May have observed earlier +2 token
  - Takes an even number

First,Second, & Third Tokens must be Ordered

- Because +1 token's path is disjoint
  - It chooses 0
  - Ordered first
  - Rest take odd numbers
- But last token takes an even number
- Something's wrong!

Third Token



Halt blue token before first green balancer

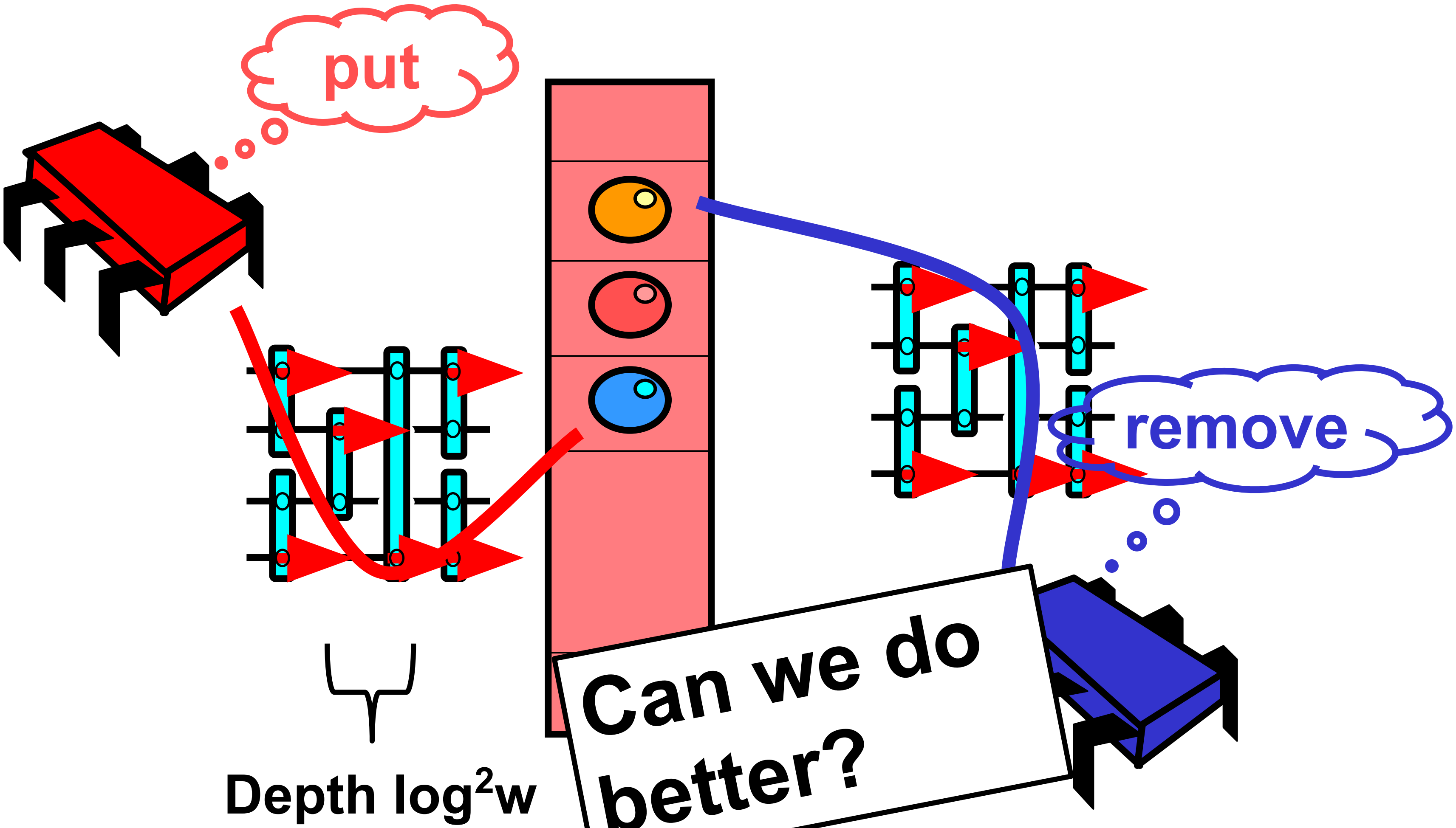
Continuing in this way

- We can “park” a token
  - In front of a balancer
  - That token #1 will visit
- There are n-1 other tokens
  - Two wires per balancer
  - Path includes n-1 balancers!

Theorem

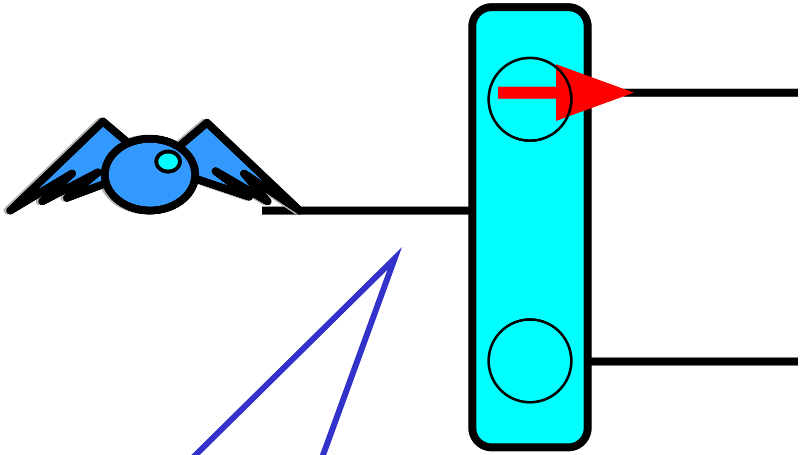
- In any adding network
  - In sequential executions
  - Tokens traverse at least n-1 balancers
- Same arguments apply to
  - Linearizable counting networks
  - Multiplying networks
  - And others

Shared Pool



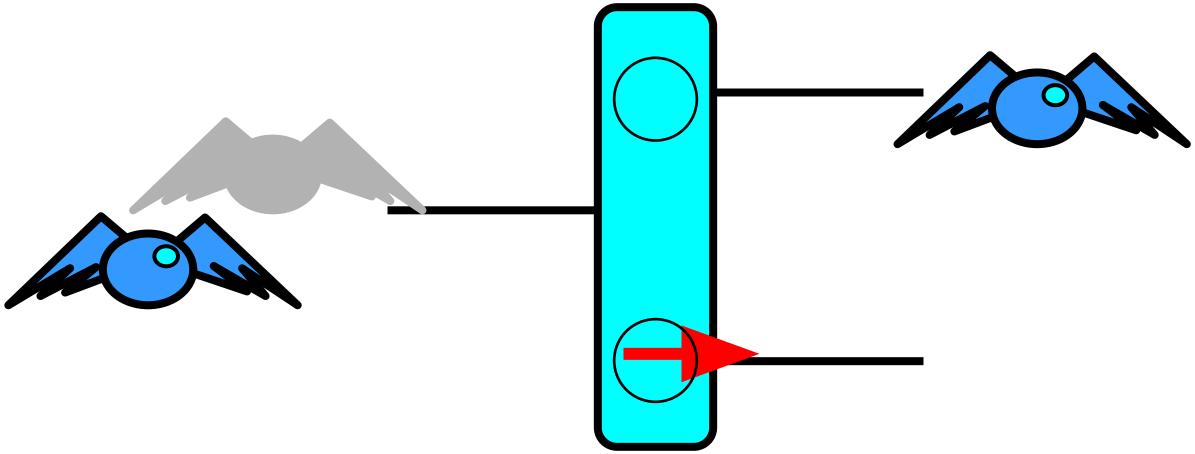
Can we do better?

Counting Trees

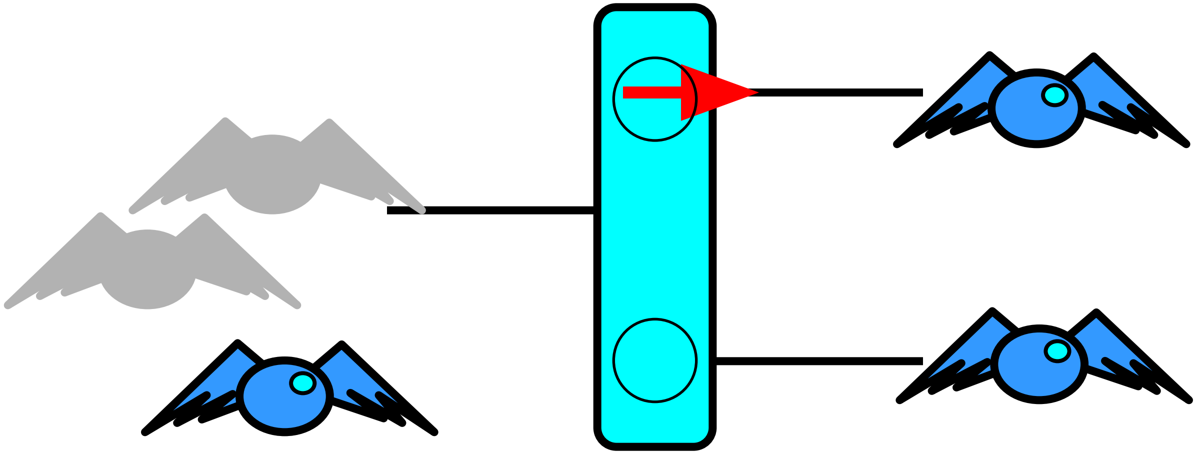


Single input wire

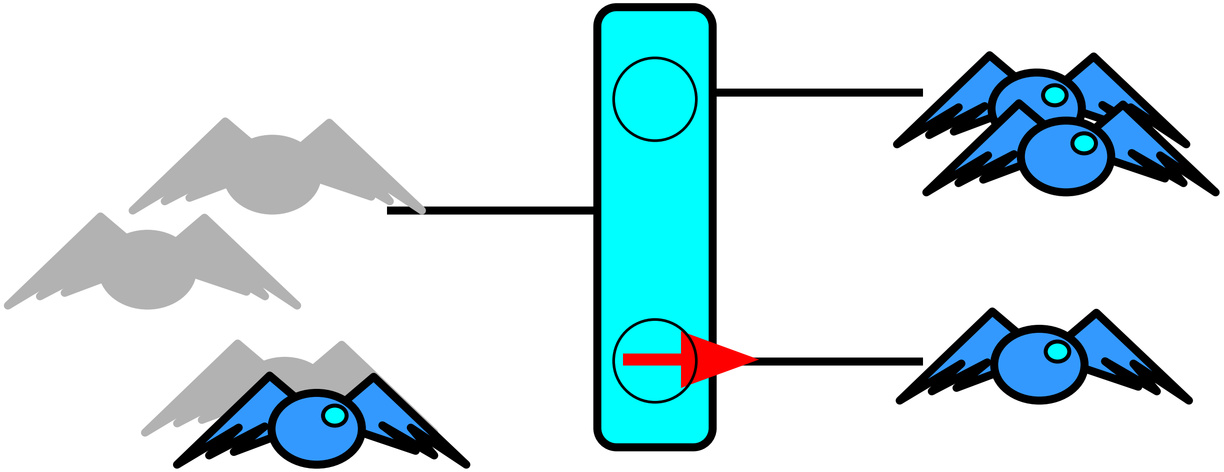
Counting Trees



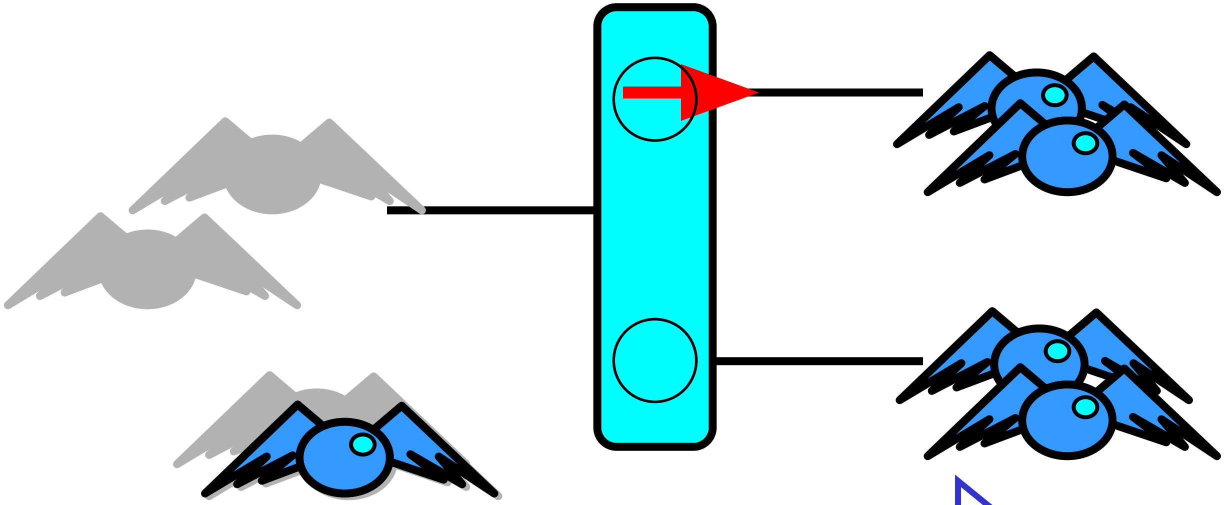
Counting Trees



Counting Trees

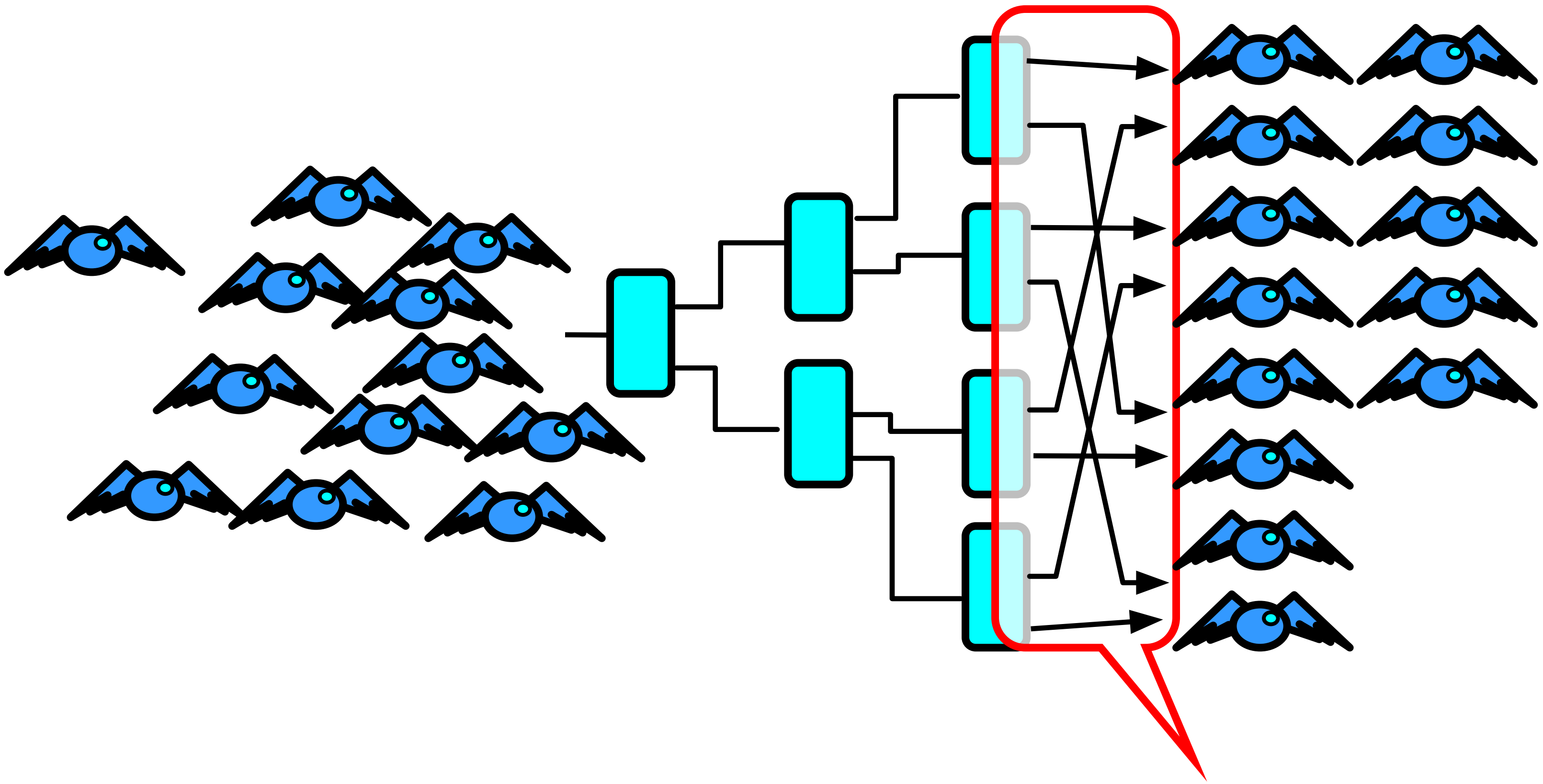


Counting Trees



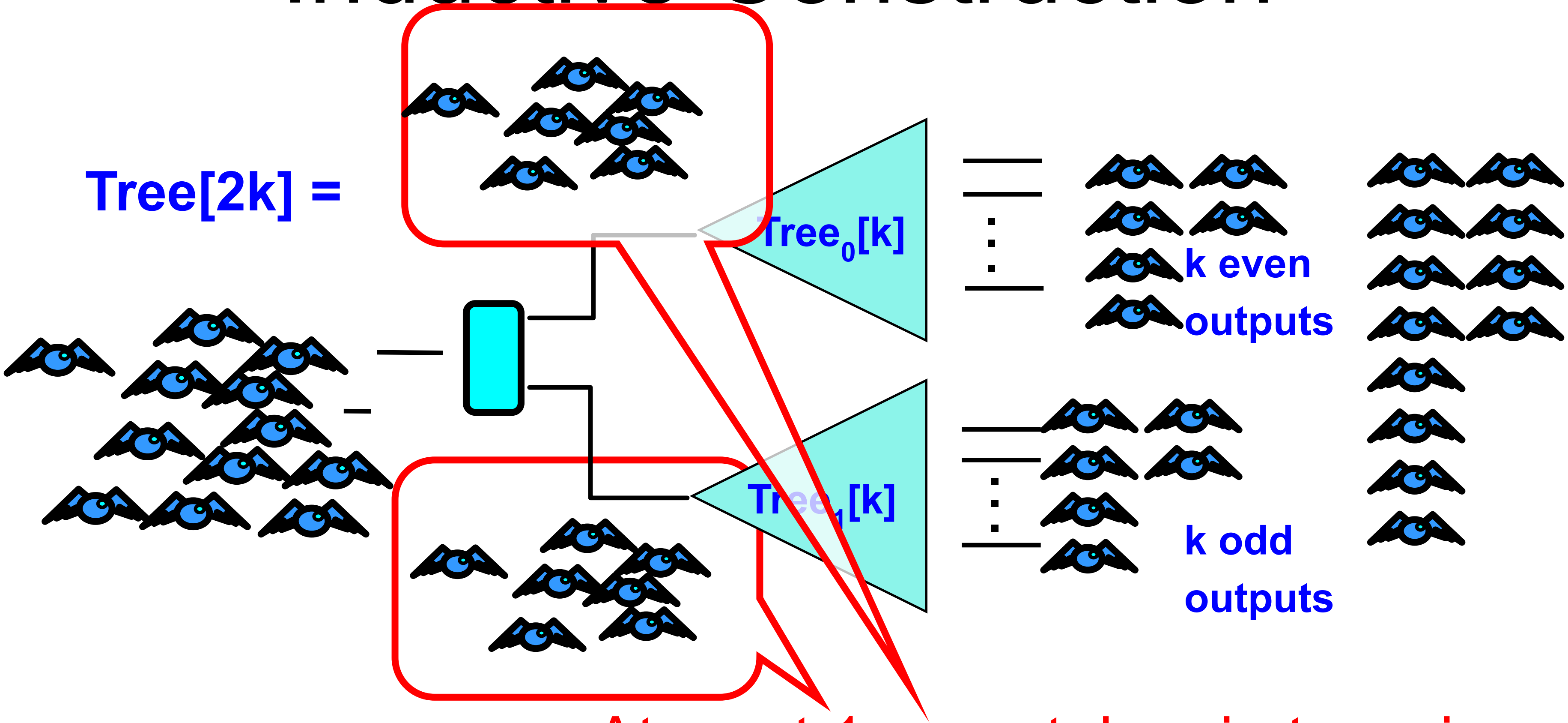
Step property in quiescent state

Counting Trees



Interleaved output wires

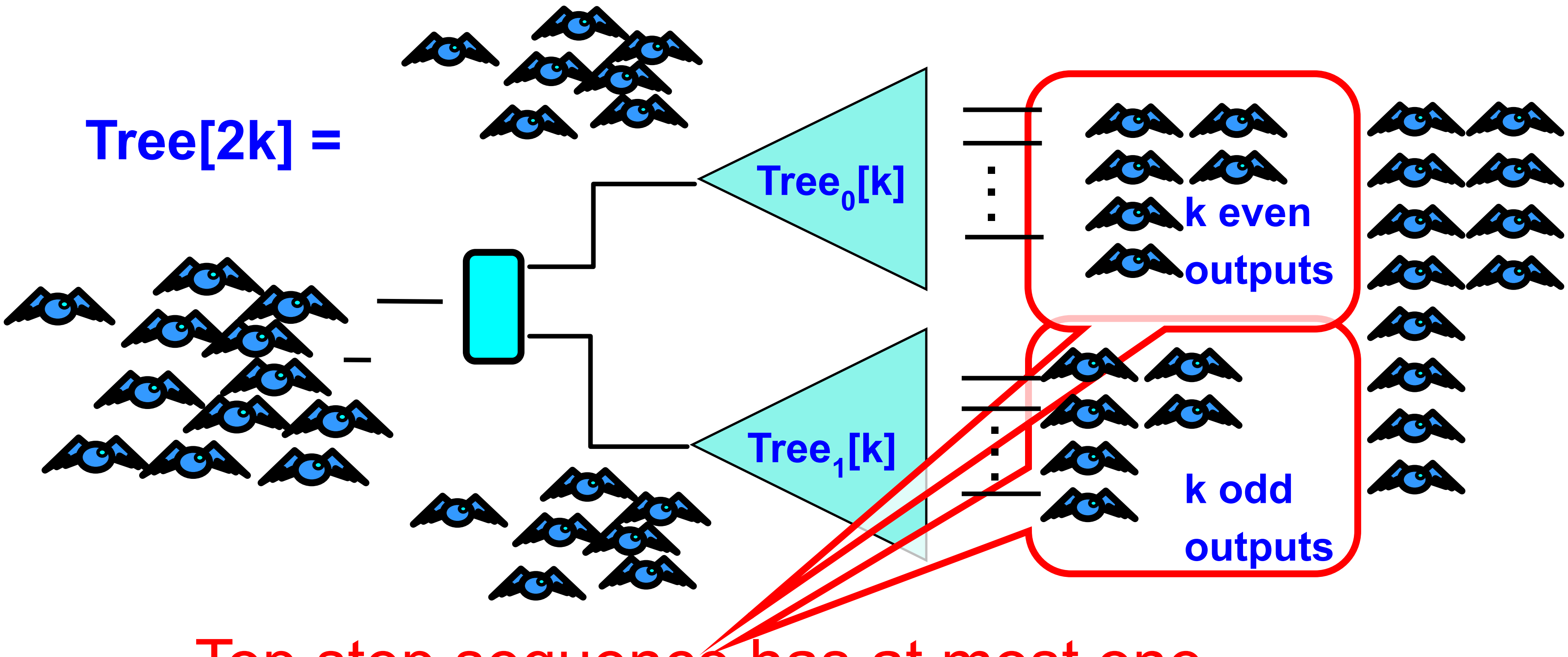
Inductive Construction



At most 1 more token in top wire

Tree[2k] has step property in quiescent state.

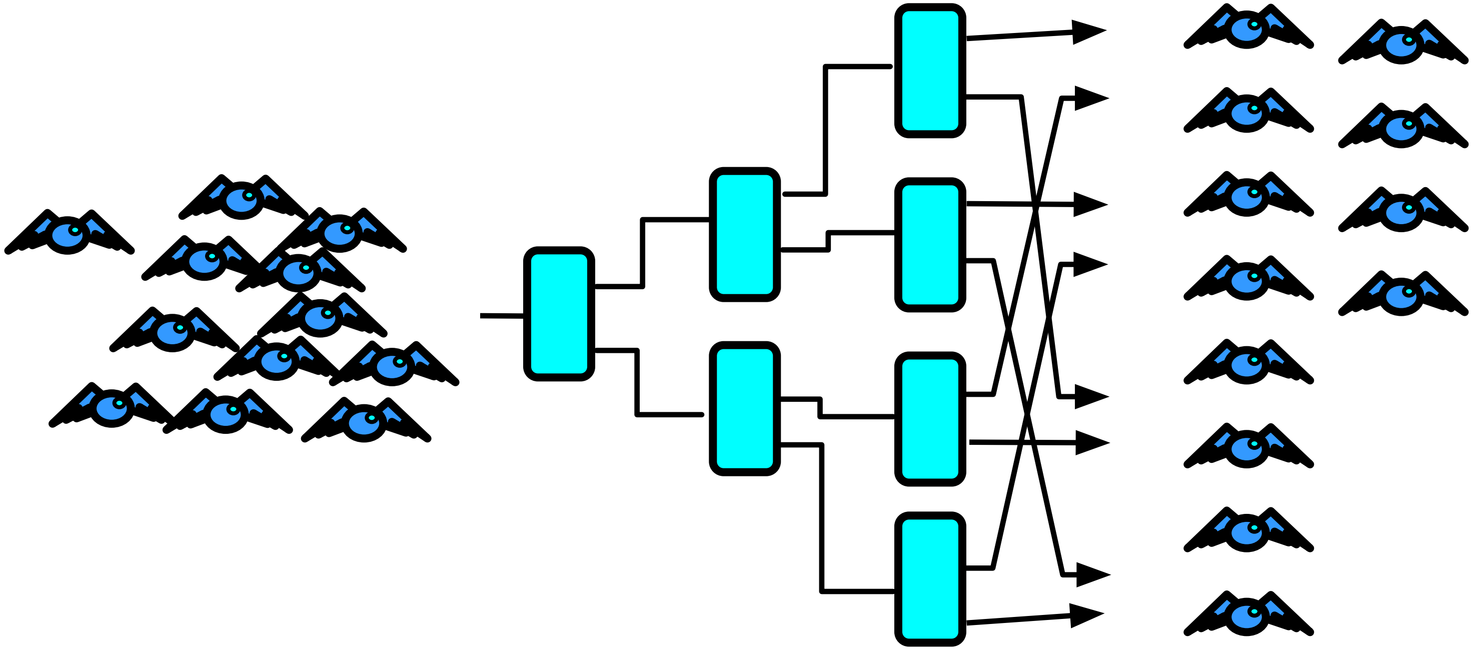
Inductive Construction



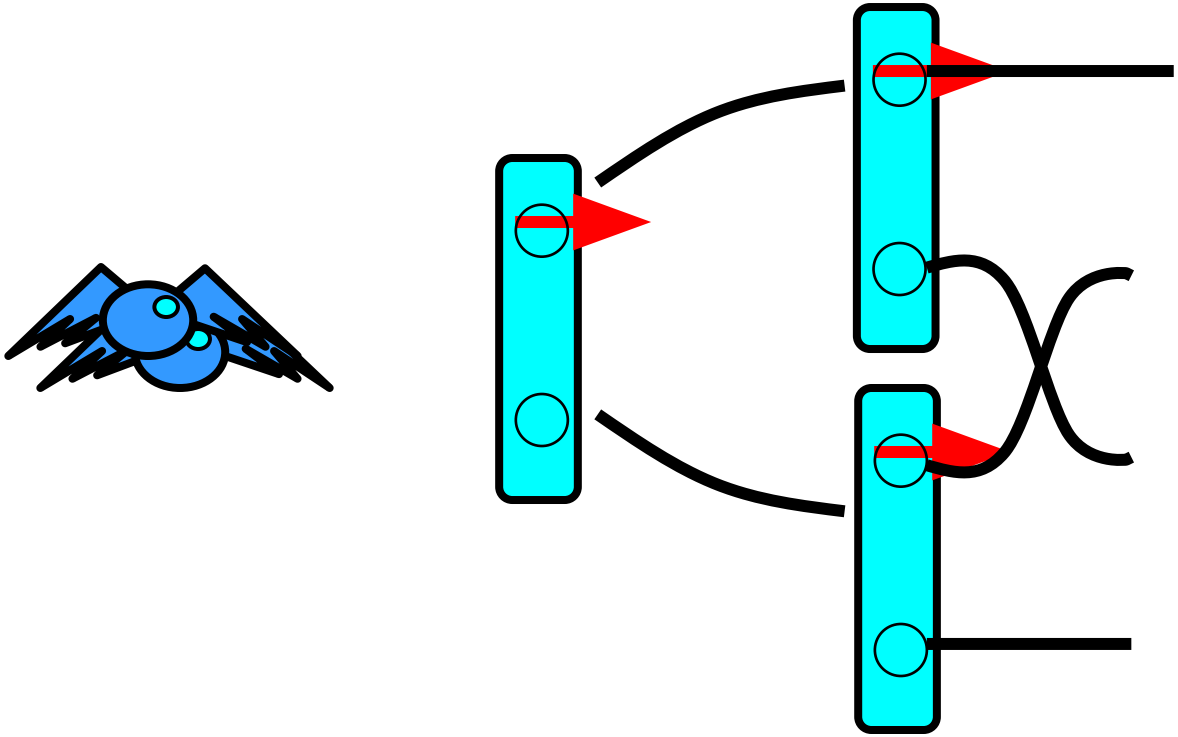
Top step sequence has at most one extra on last wire of step

Tree[2k] has step property in quiescent state.

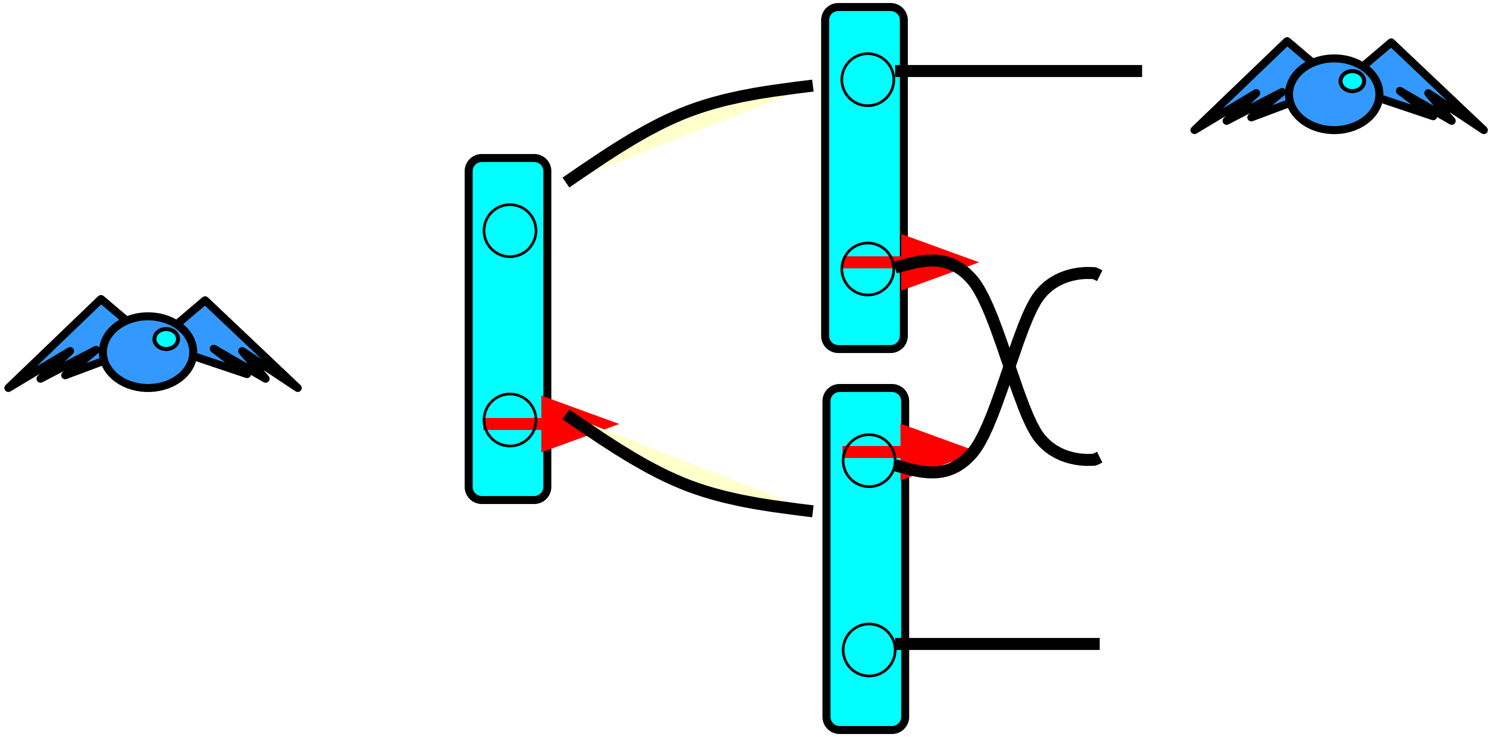
Implementing Counting Trees



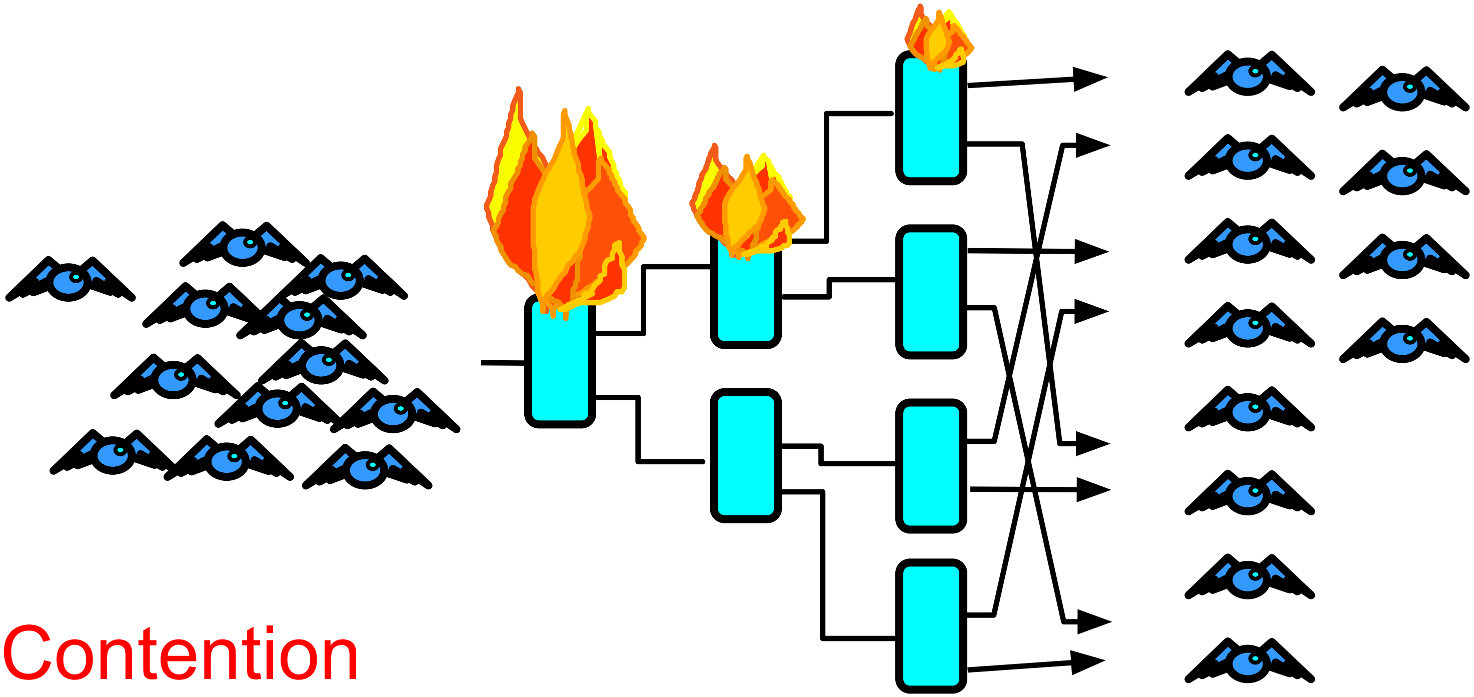
Example



Example



Implementing Counting Trees

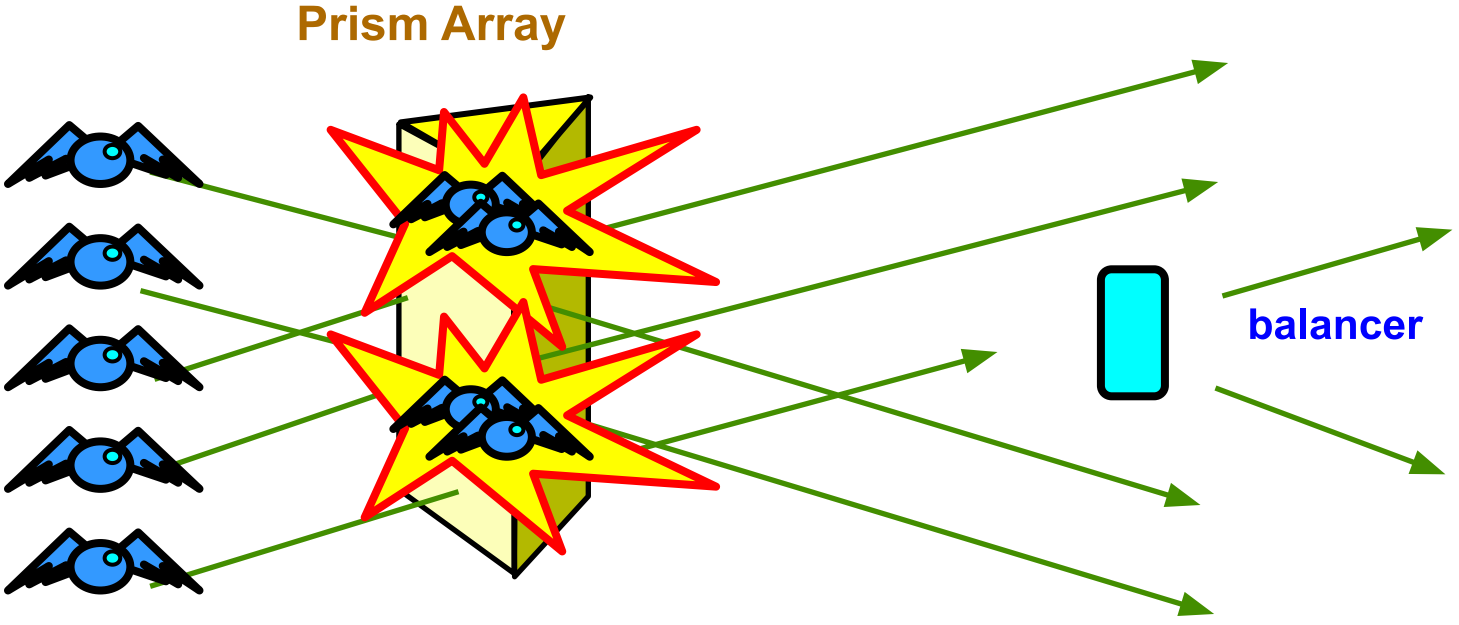


Contention

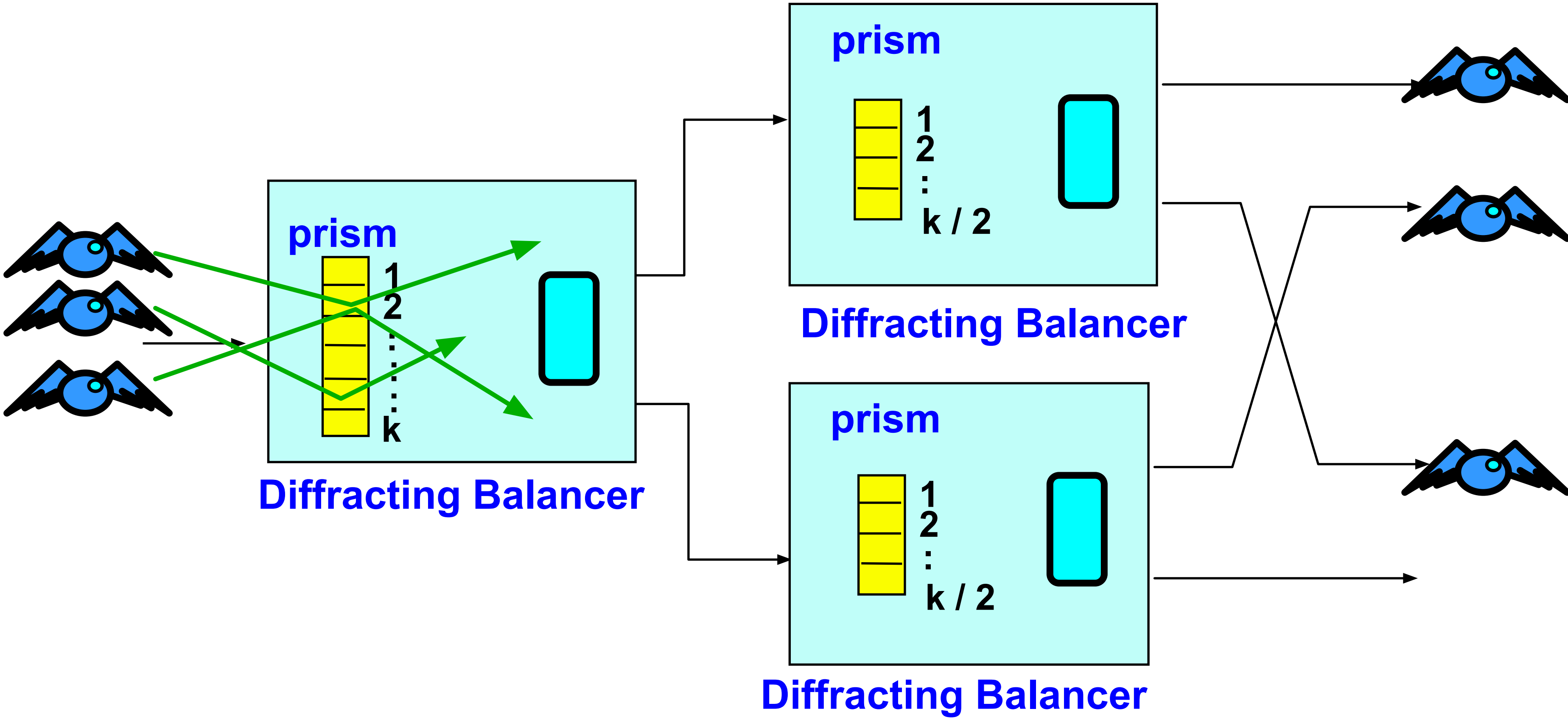
Sequential bottleneck

Diffraction Balancing

If an even number of tokens visit a balancer, the toggle bit remains unchanged!

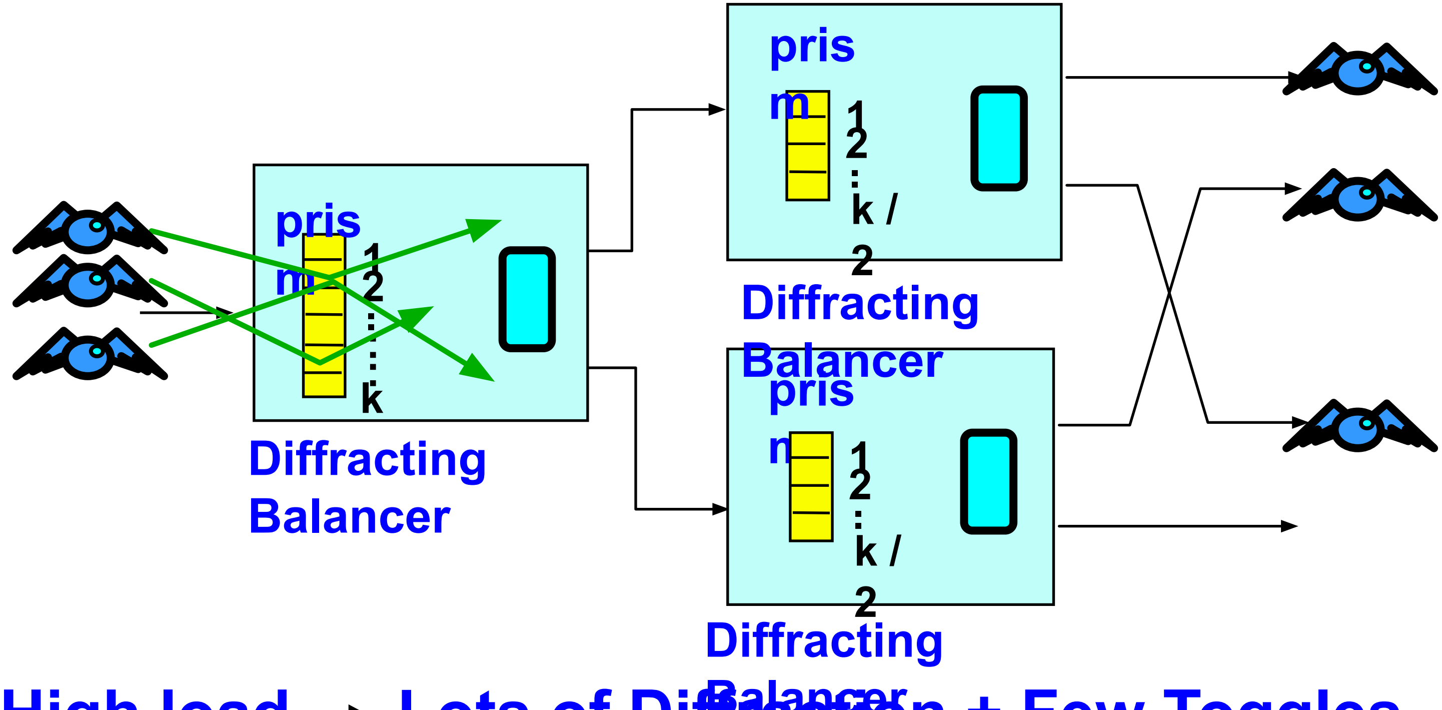


Diffracting Tree



Diffracting balancer same as balancer.

Diffracting Tree



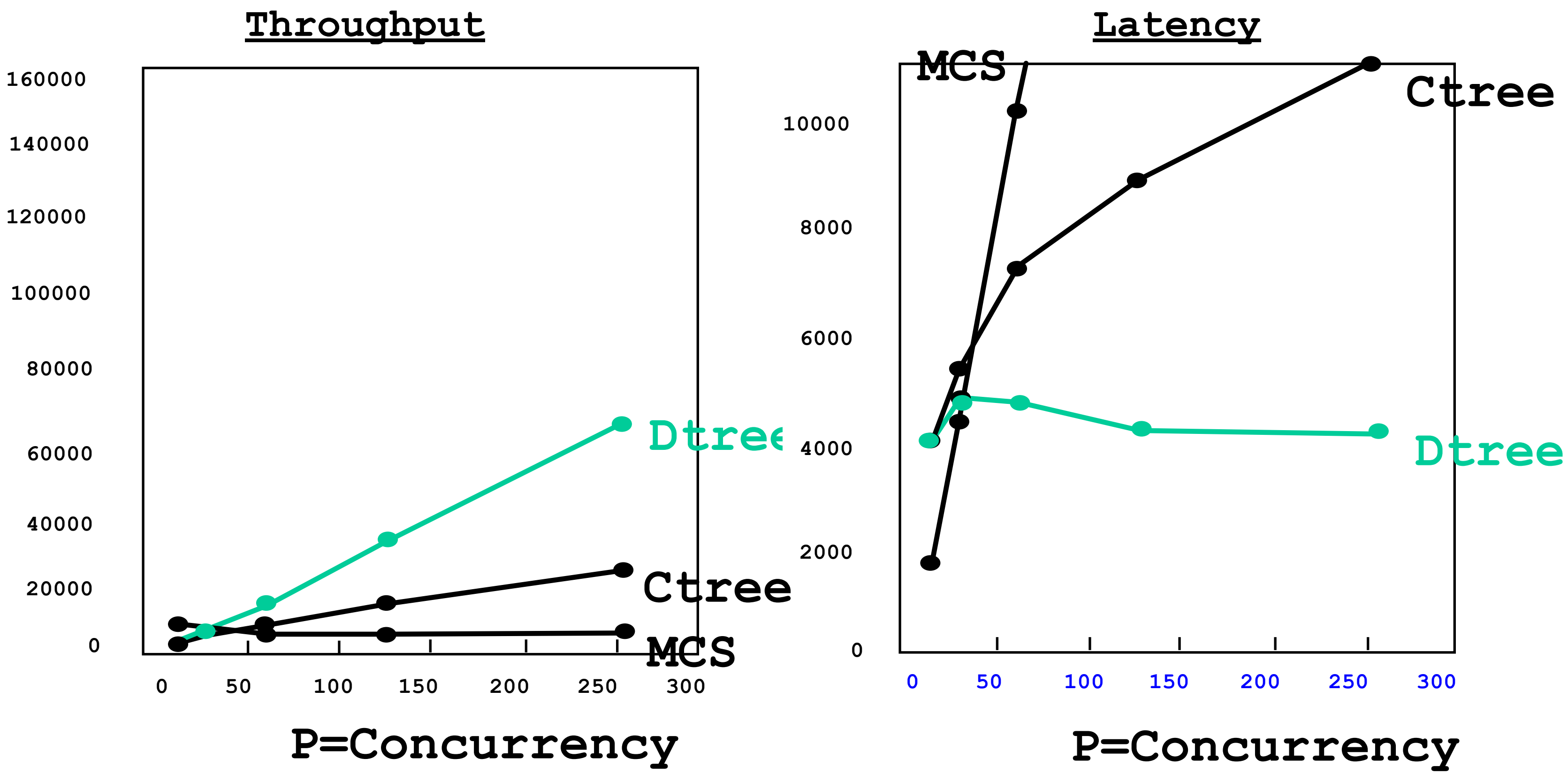
High load ➡ Lots of Diffraction + Few Toggles

Low load ➡ Low Diffraction + Few Toggles

⬇

High Throuhput with Low Contention

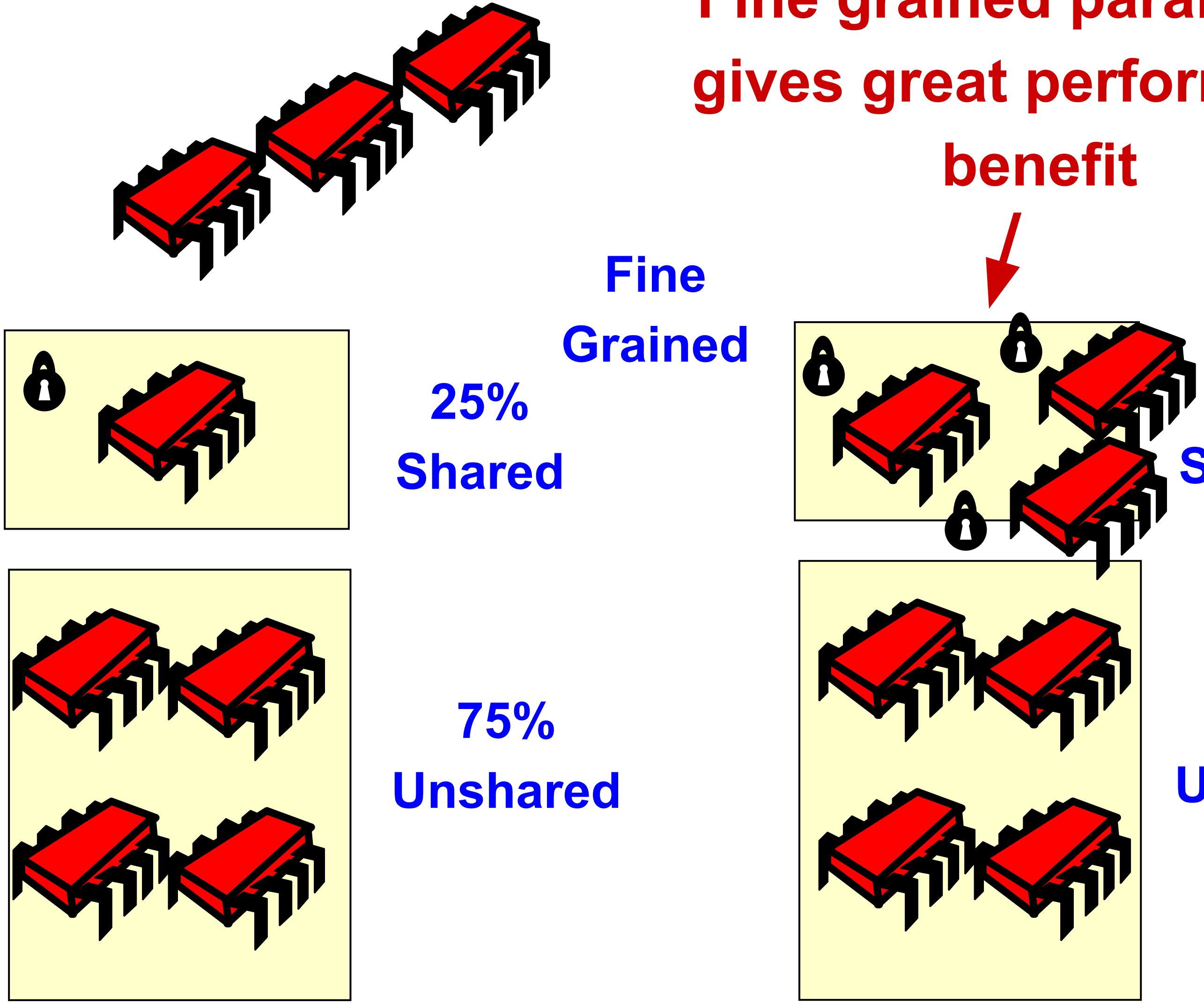
Performance



Throughput

Latency

Amdahl’s Law Works



Coarse Grained

25% Shared

75% Unshared

Fine Grained

25% Shared

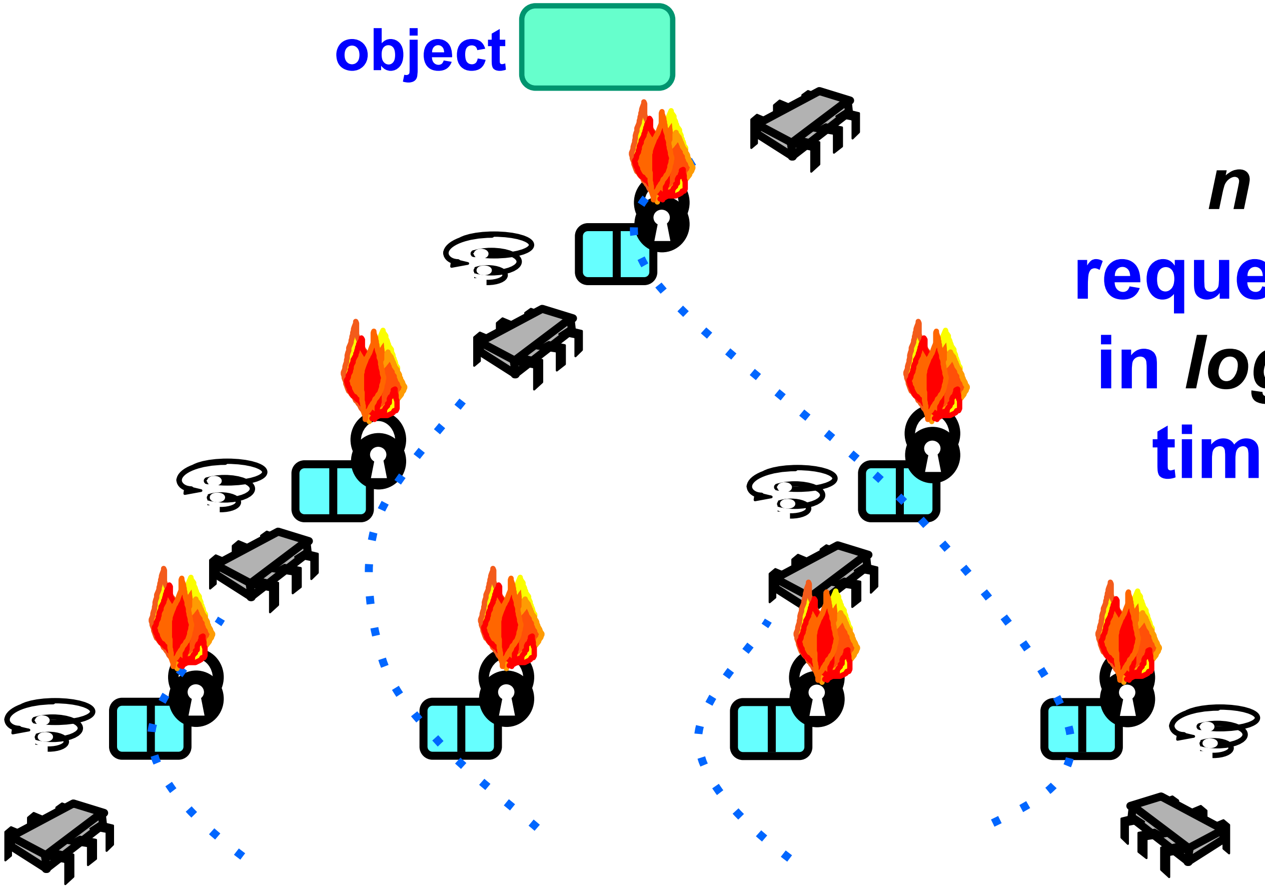
75% Unshared

Fine grained parallelism gives great performance benefit

But…

- Can we always draw the right conclusions from Amdahl's law?
- Claim: sometimes the overhead of fine-grained synchronization is so high…that it is better to have a single thread do all the work sequentially in order to avoid it

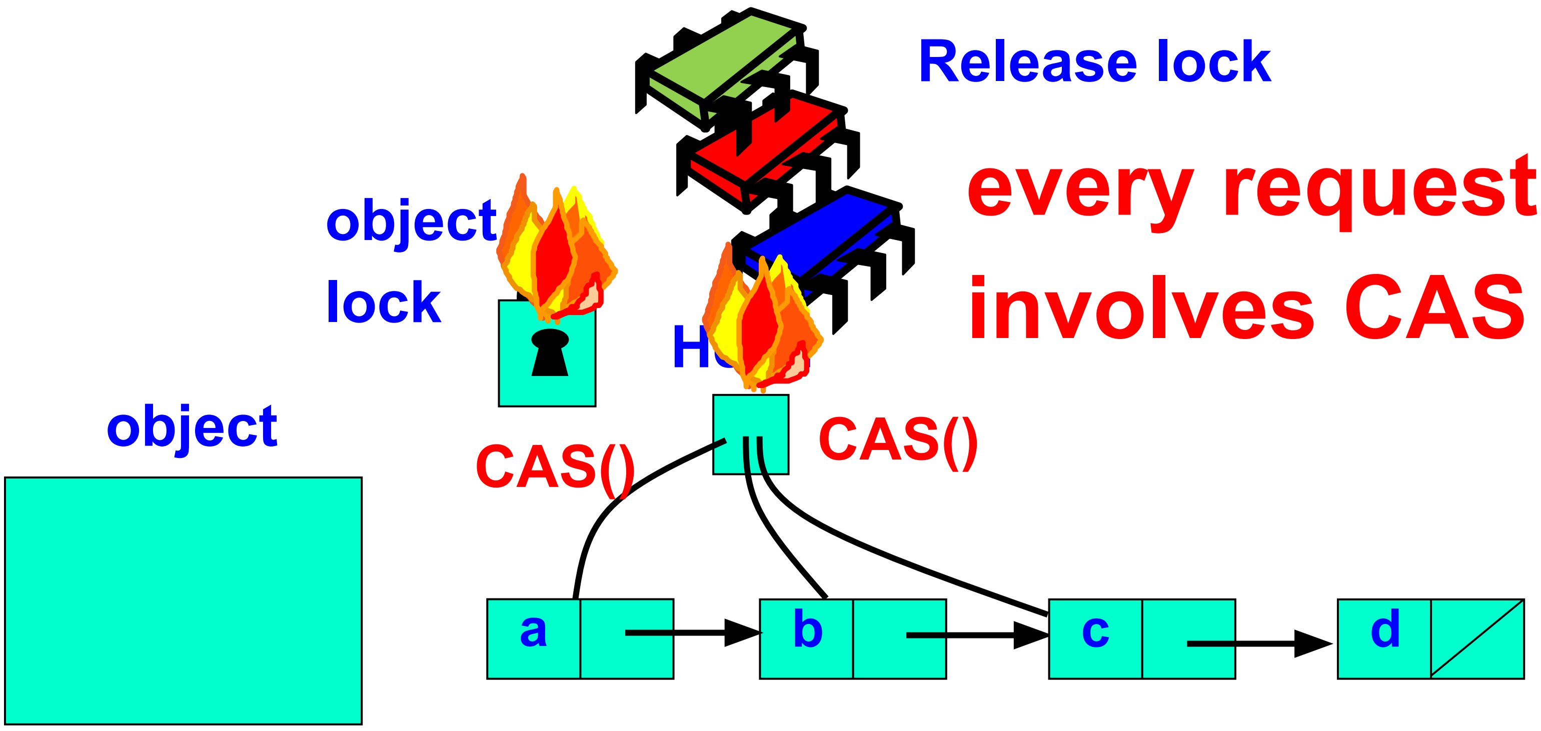
Software Combining Tree



n requests in *log n* time

Tree requires a major coordination effort: multiple CAS operations, cache-misses, etc

Oyama et. al Mutex



Release lock

every request involves CAS

Apply a,b,c, and d to object

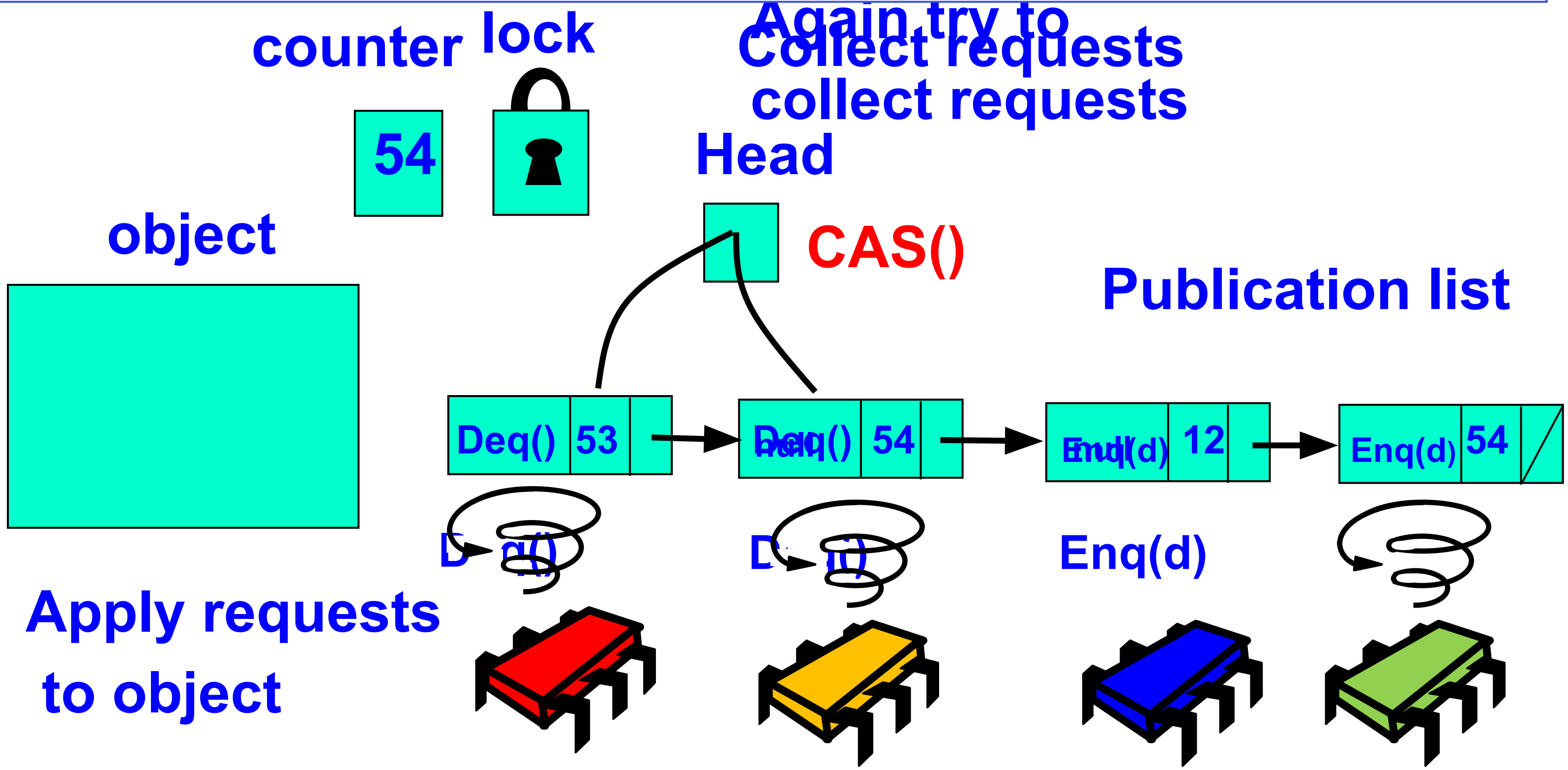
return responses

Flat Combining

- Have single lock holder collect and perform requests of all others
  - Without using CAS operations to coordinate requests
  - With combining of requests (if cost of k batched operations is less than that of k operations in sequence □ we win)

Flat-Combining

Most requests do not involve a CAS, in fact, not even a memory barrier



counter

lock

54

Again try to
Collect requests

collect requests

Head

CAS()

object

Publication list

Deq() 53

Deq() 54

Enq(d) 12

Enq(d) 54

Deq()

Deq()

Enq(d)

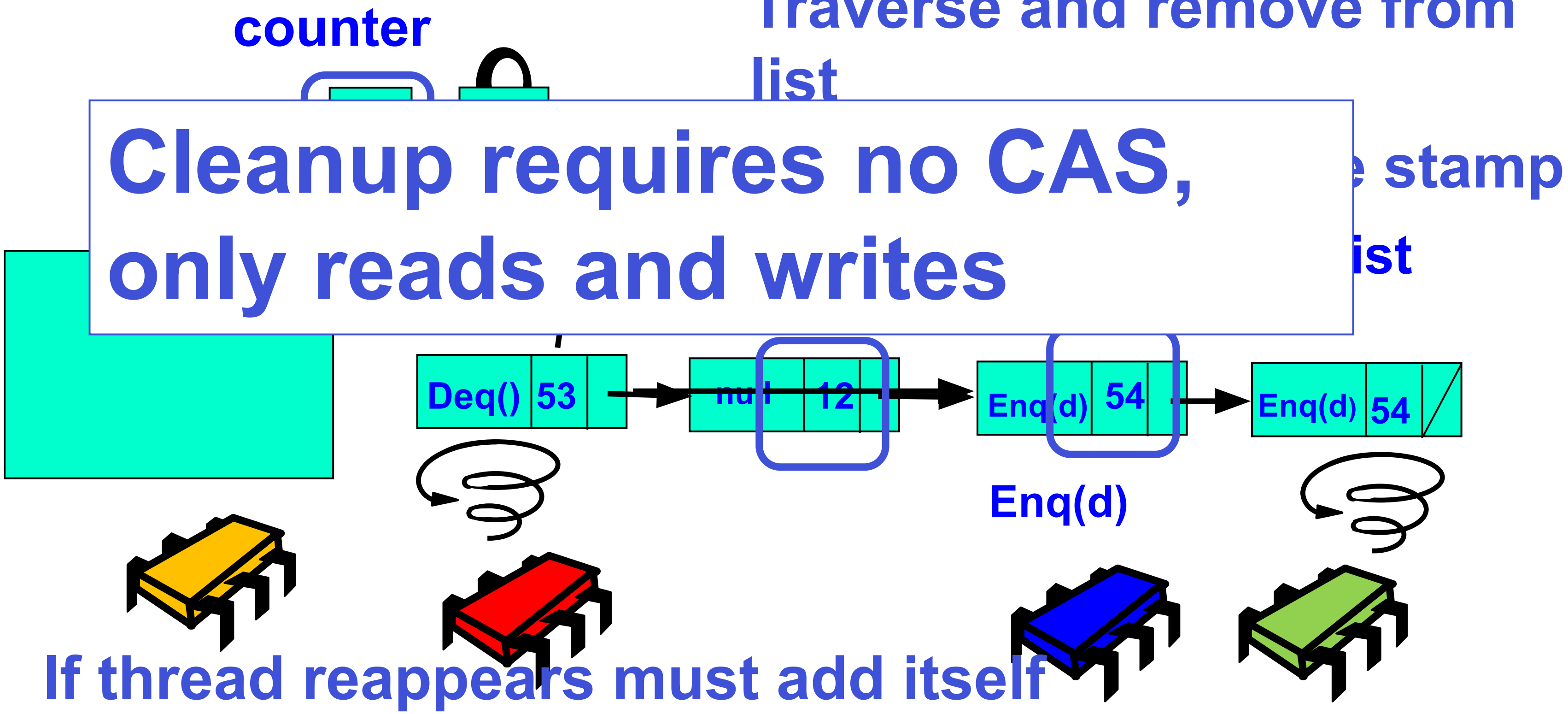
Apply requests
to object

Flat-Combining Pub-List Cleanup

Every combiner increments counter and updates record's time stamp when returning response

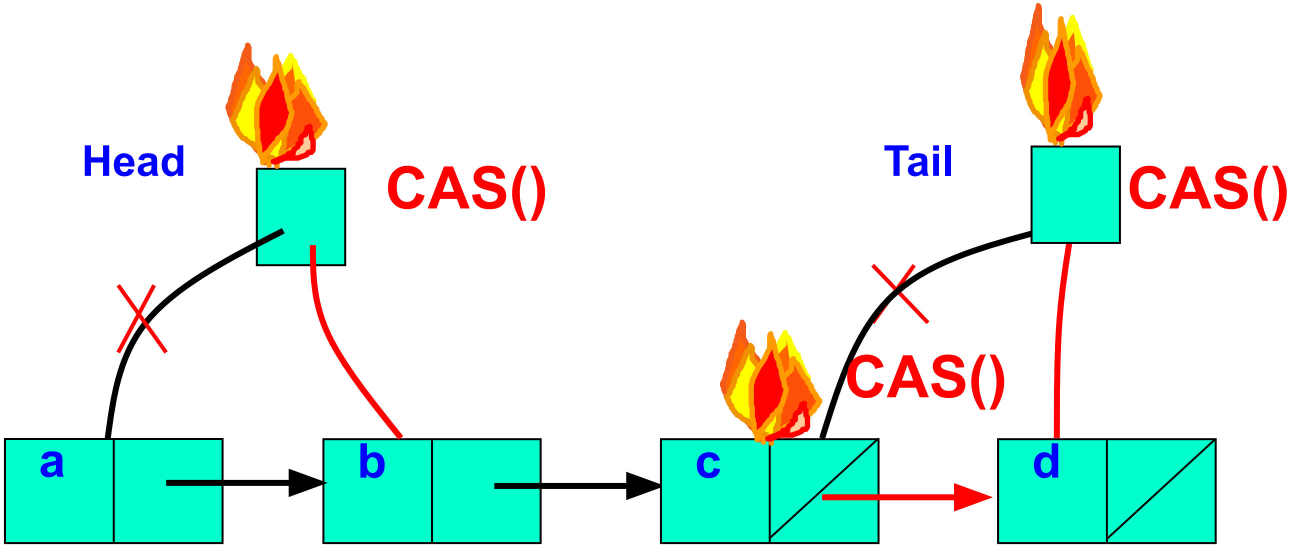
Traverse and remove from list

Cleanup requires no CAS, only reads and writes



If thread reappears must add itself to pub list

Fine-Grained Lock-free FIFO Queue

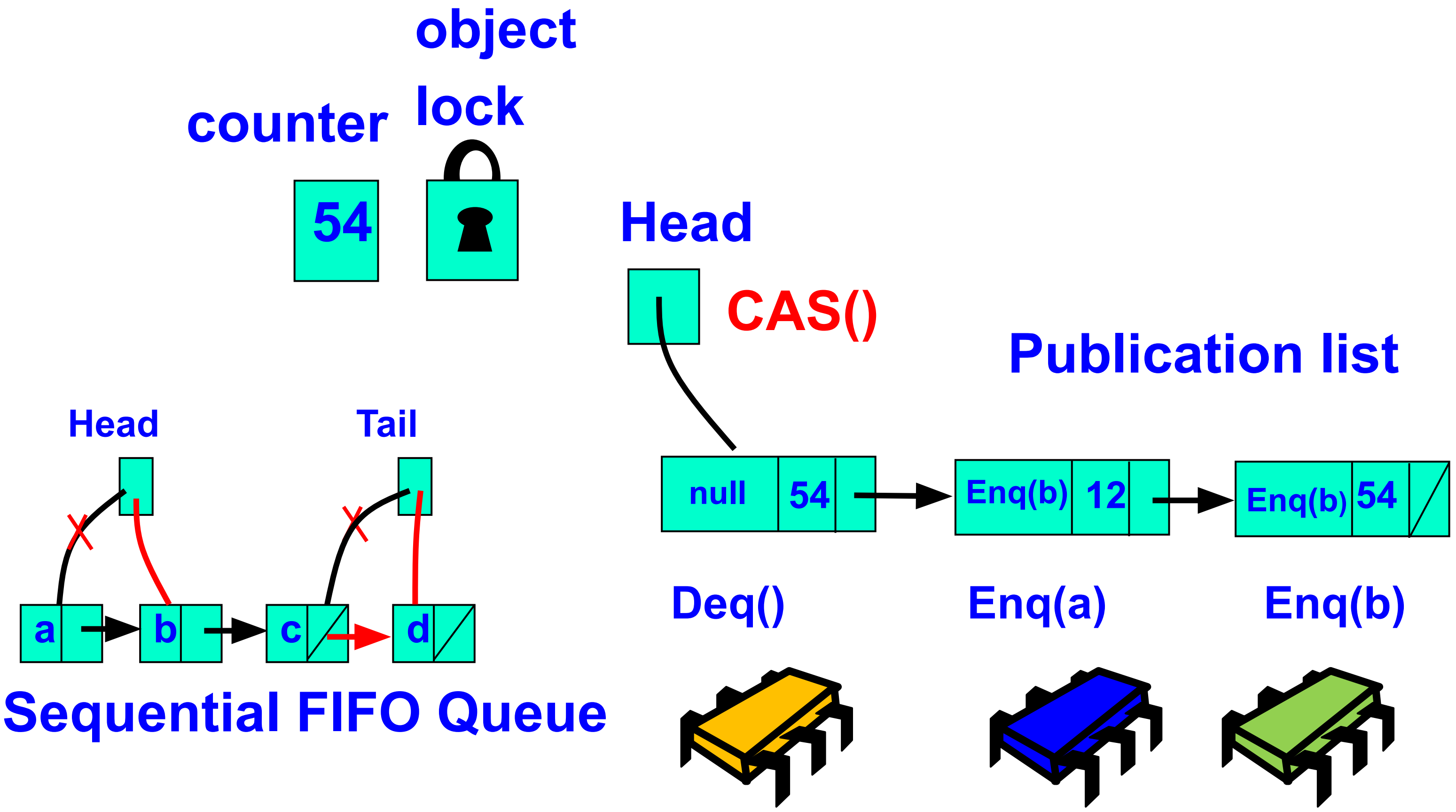


P: Dequeue() => a

Q: Enqueue(d)

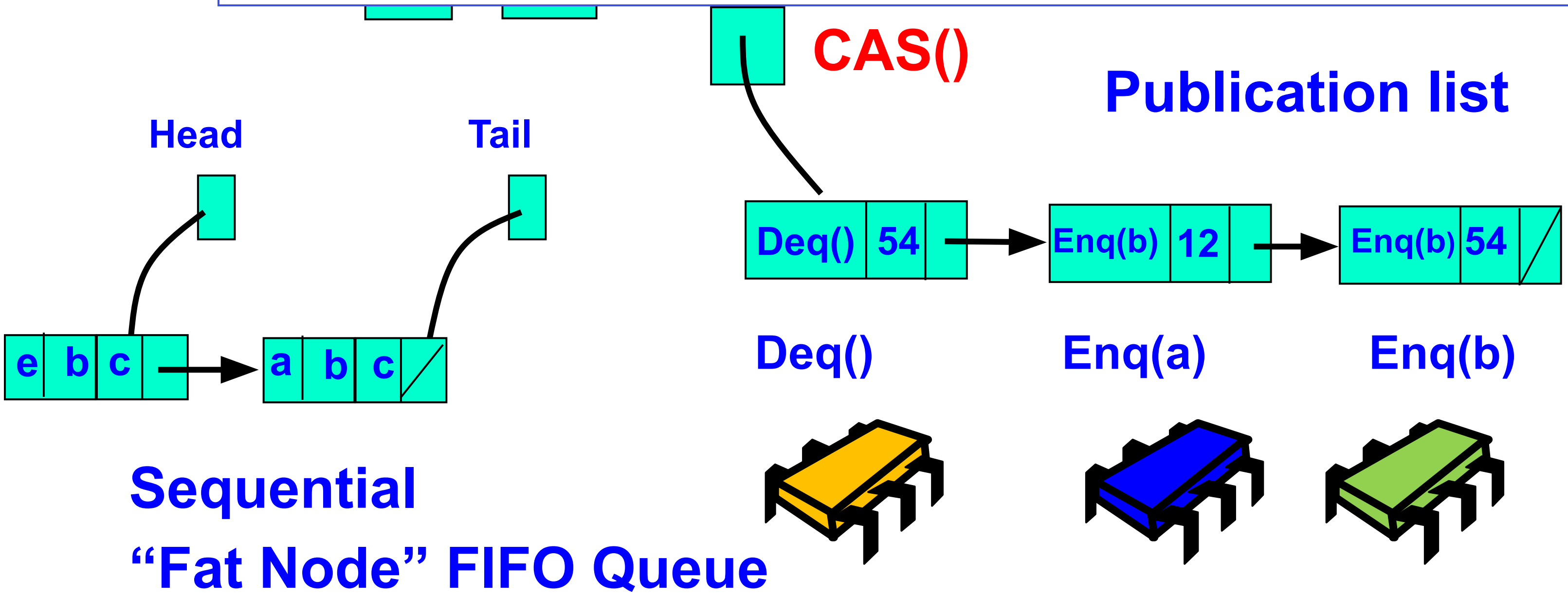
Flat Combining FIFO Queue

OK, but can do better…combining: collect all items into a “fat node”, enqueue in one step

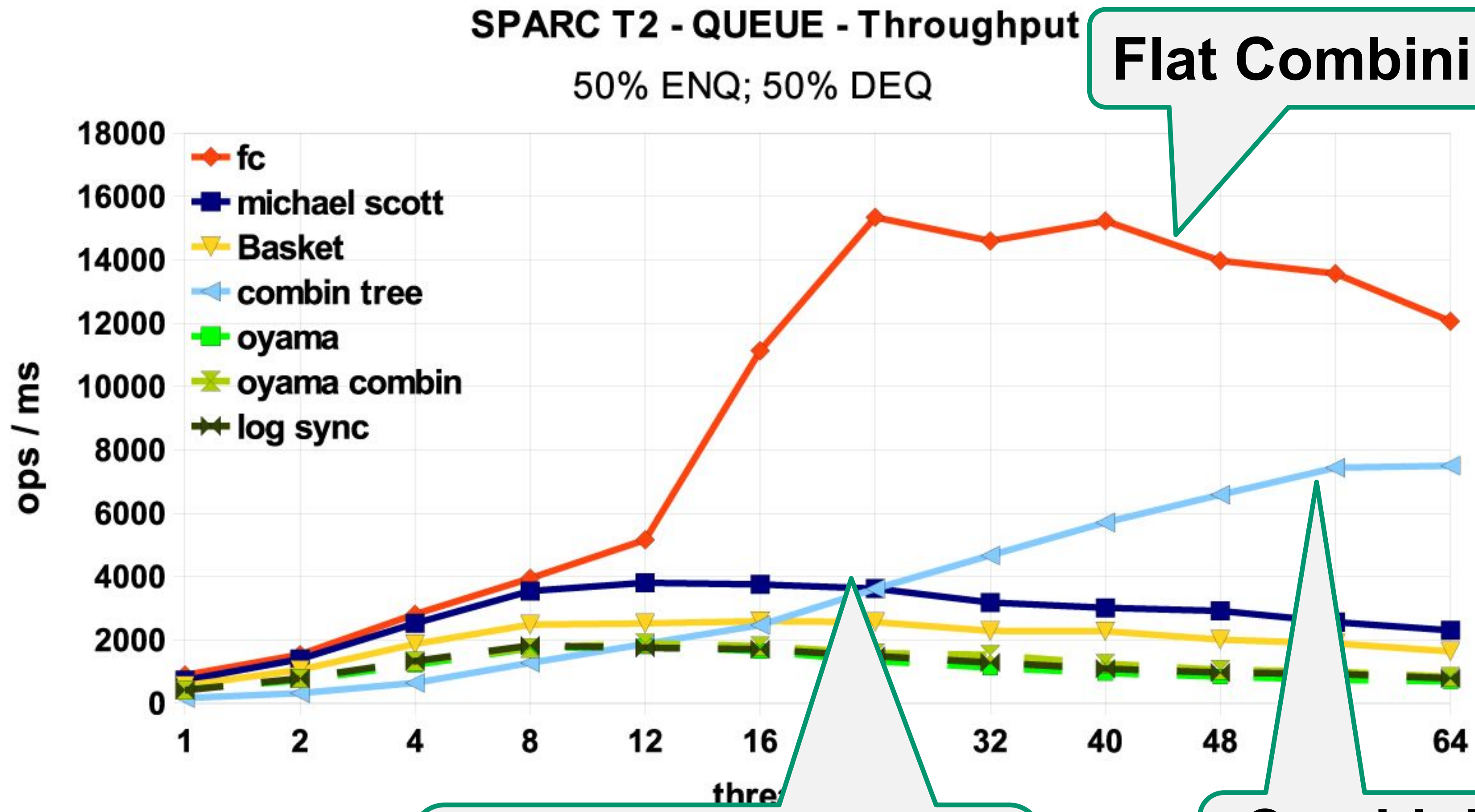


OK, but can do better…combining: collect all items into a “fat node”, enque

“Fat Node” easy sequentially but cannot be done in concurrent alg without CAS



Linearizable FIFO Queue

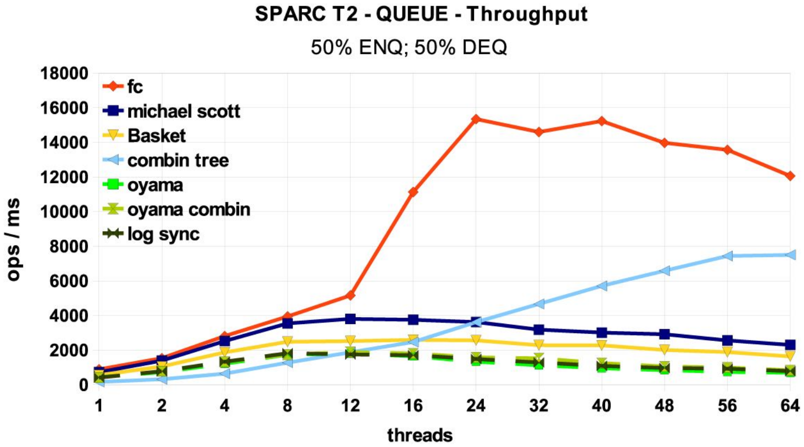


Flat Combining

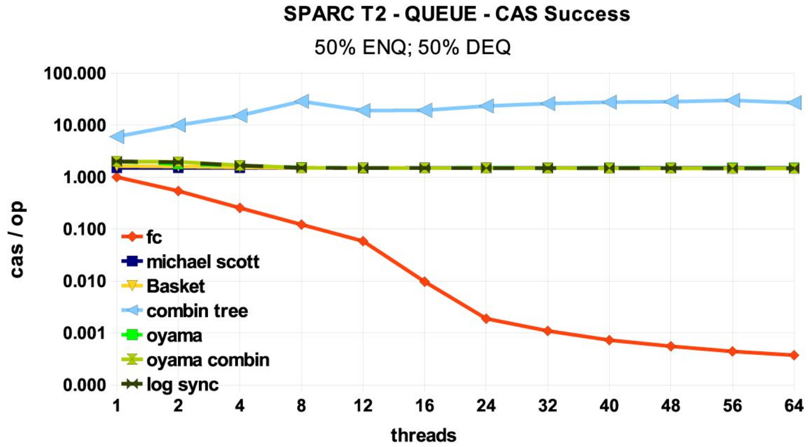
MS queue, Oyama, and Log-Synch

Combining tree

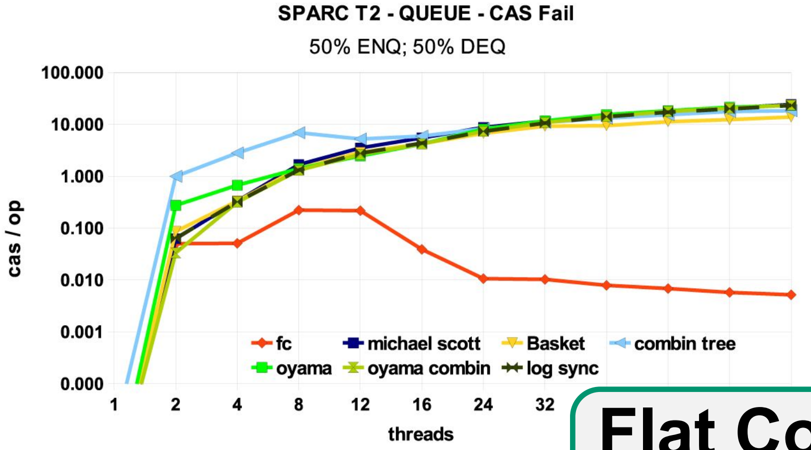
Benefits of Flat Combining


SPARC T2 - QUEUE - Throughput

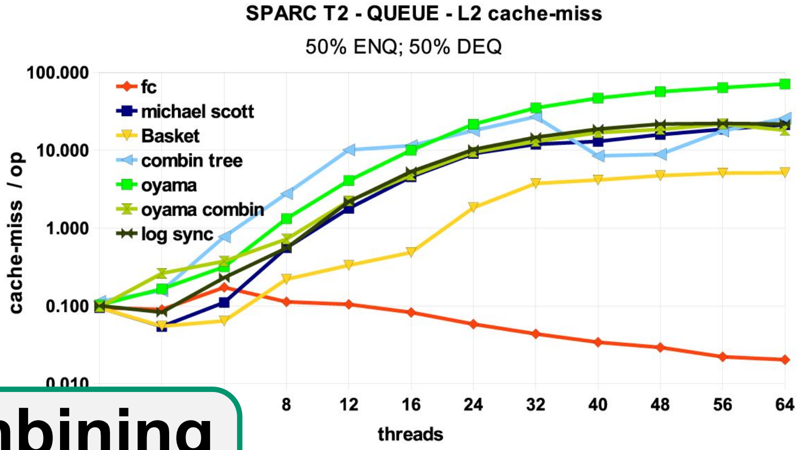
50% ENQ; 50% DEQ


SPARC T2 - QUEUE - CAS Success

50% ENQ; 50% DEQ


SPARC T2 - QUEUE - CAS Fail

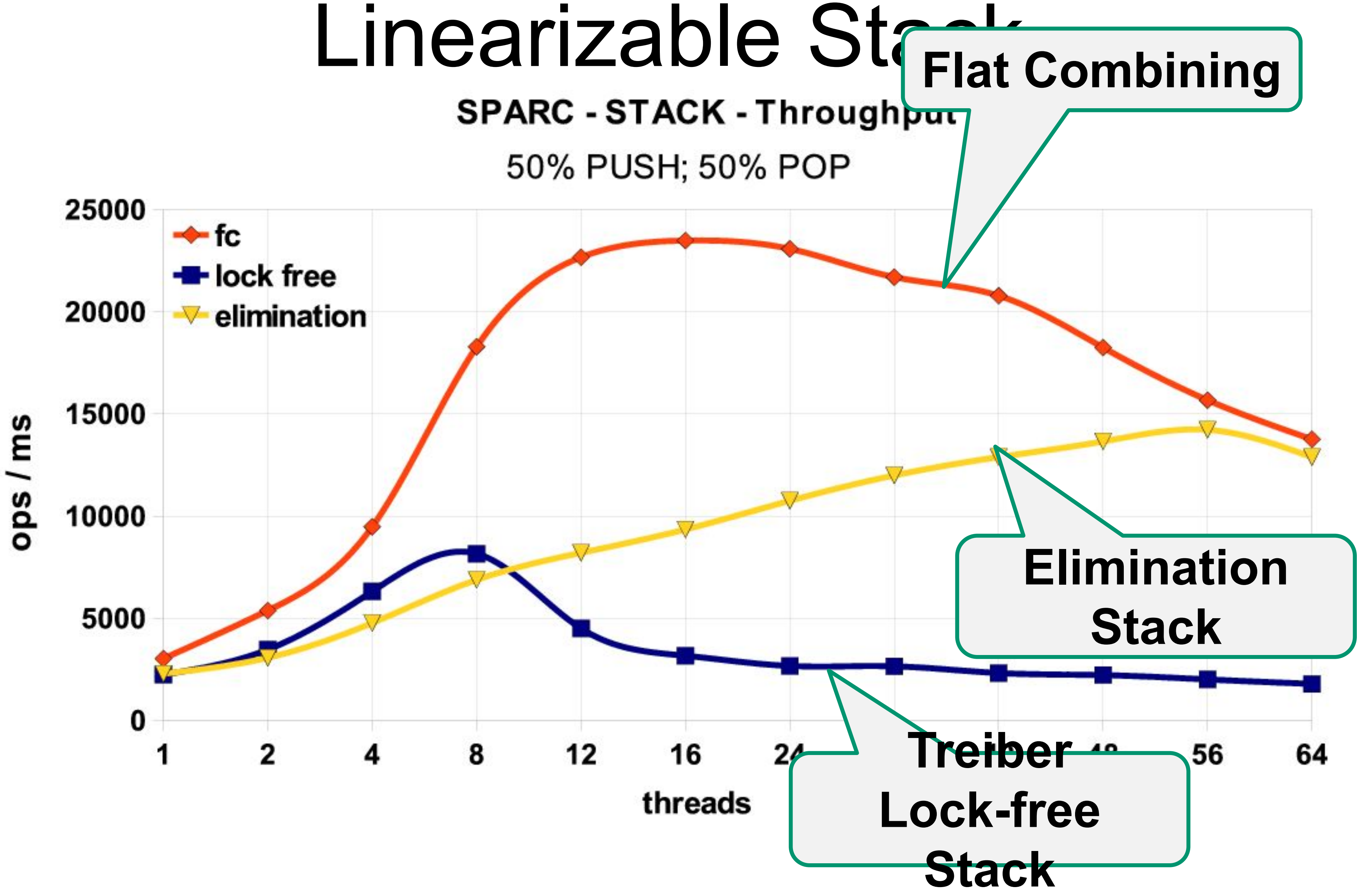
50% ENQ; 50% DEQ


SPARC T2 - QUEUE - L2 cache-miss

50% ENQ; 50% DEQ

Flat Combining in Red

Linearizable Stack



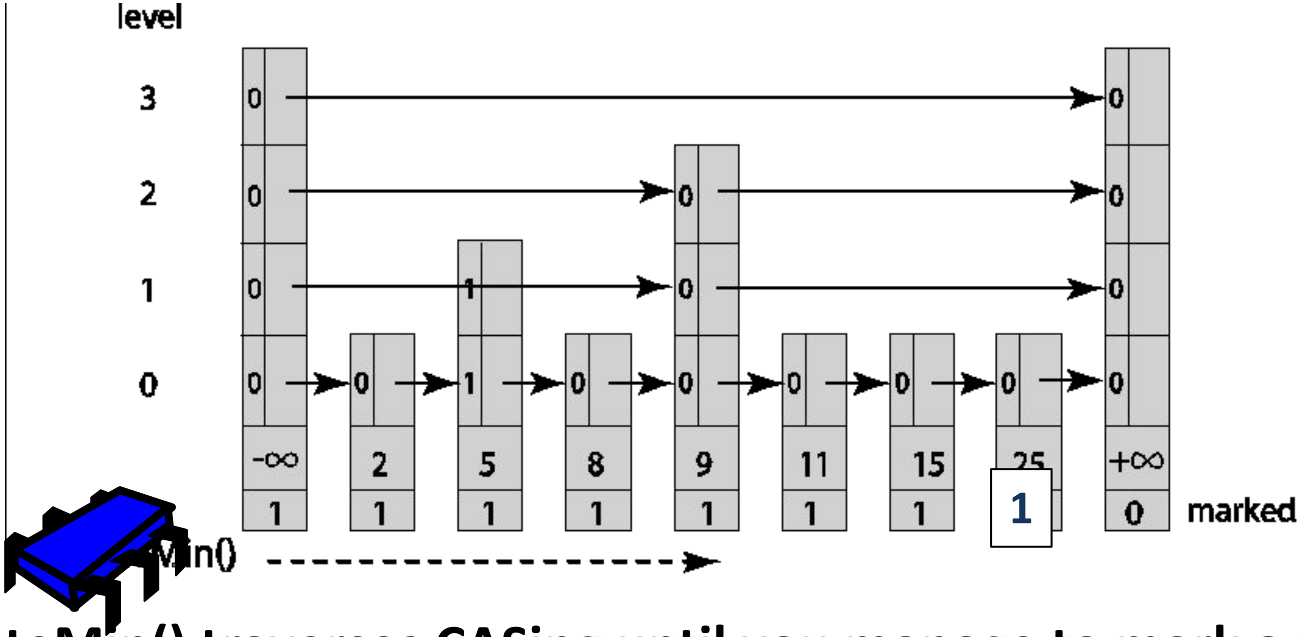
Flat Combining

Elimination Stack

Treiber Lock-free Stack

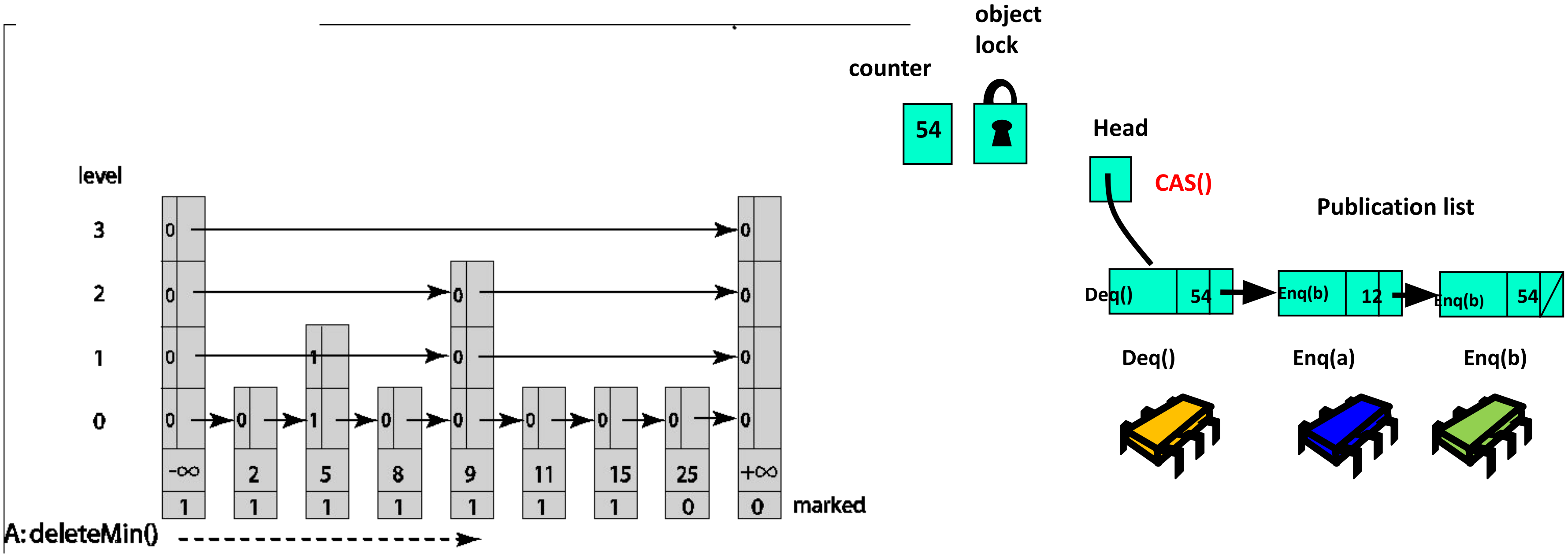
Concurrent Priority Queue (Chapter 15)

k deleteMin operations take O(k*log n)



deleteMin() traverses CASing until you manage to mark a node, then use skiplist remove your marked node

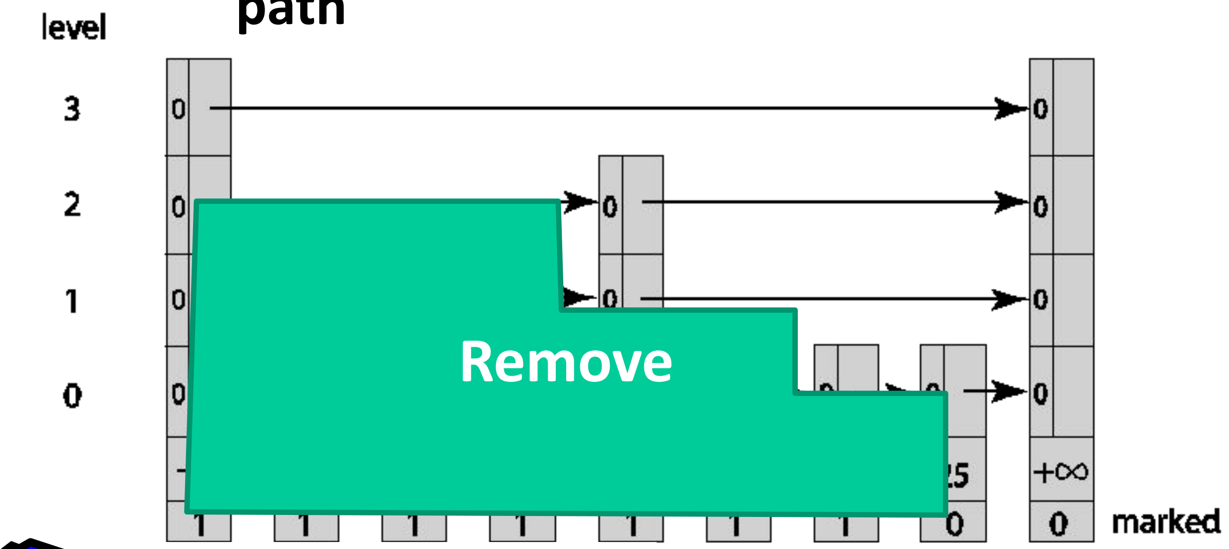
Flat-Combining Priority Queue



Flat Combining Priority Queue

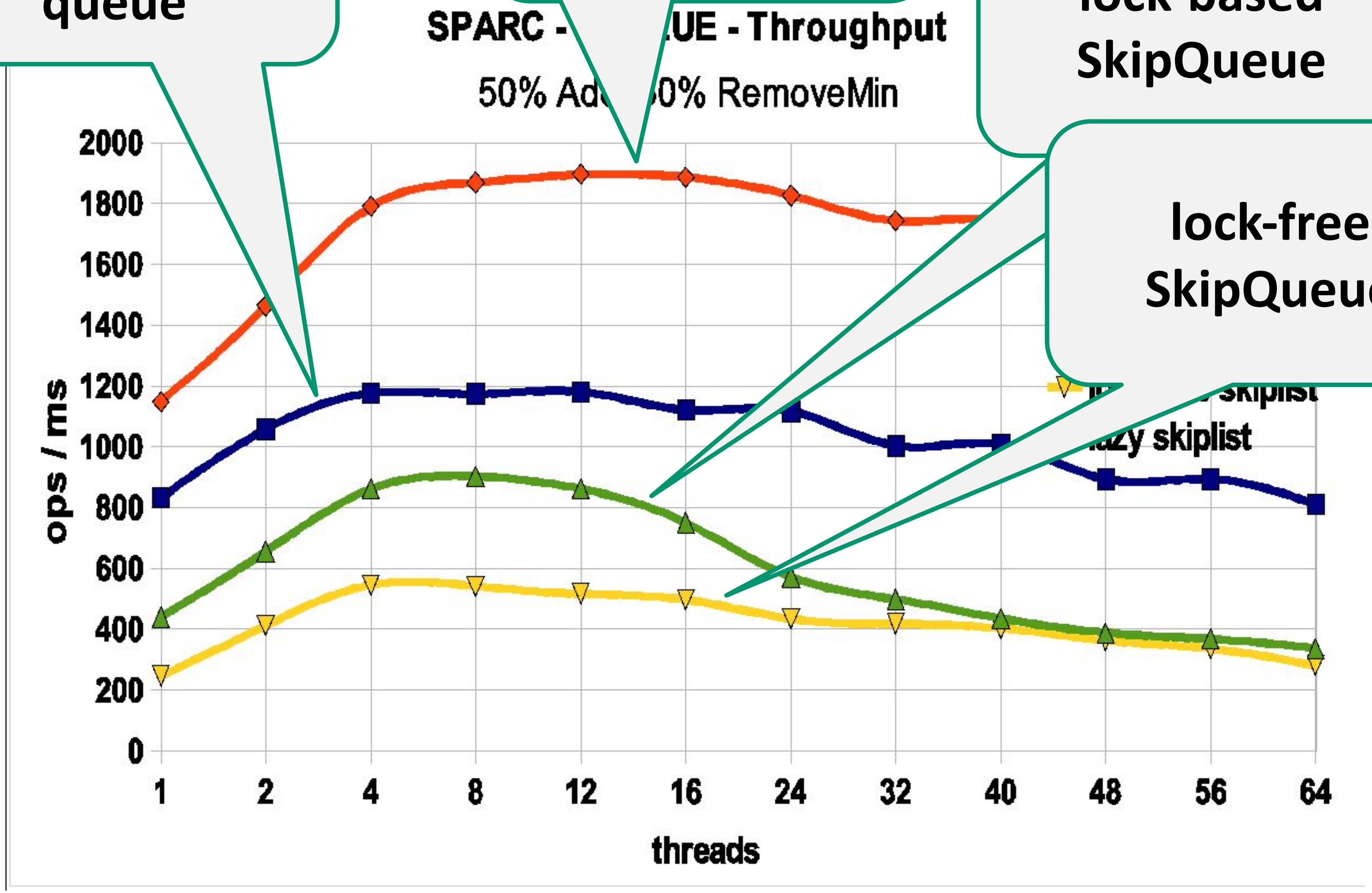
k deleteMin operations take O(k+log n)

removing all nodes below your path



traverse to find kth key, collect values to be returned

Pri ue



Flat Combining Skiplist based queue

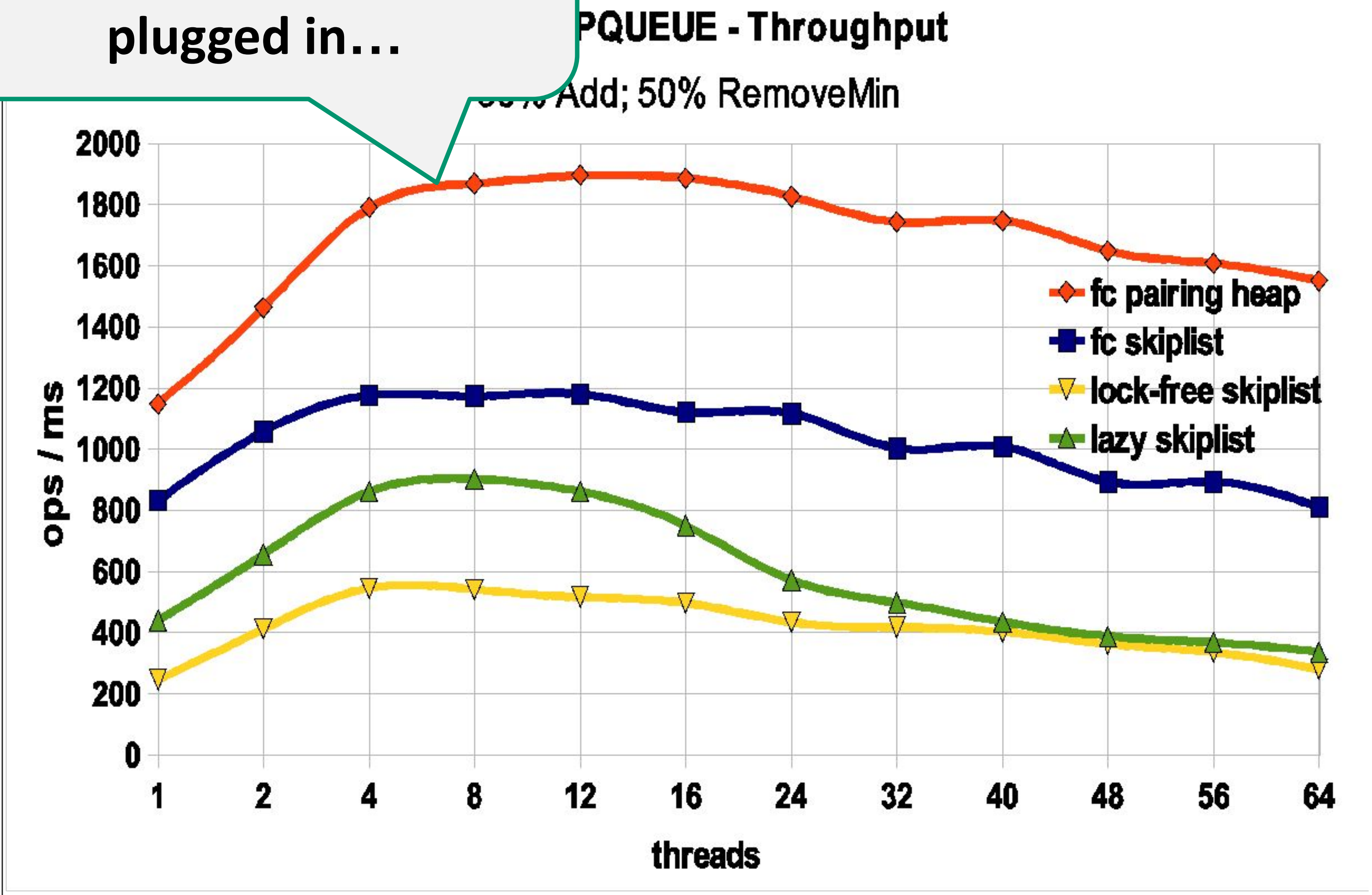
Whats this?

lock-based SkipQueue

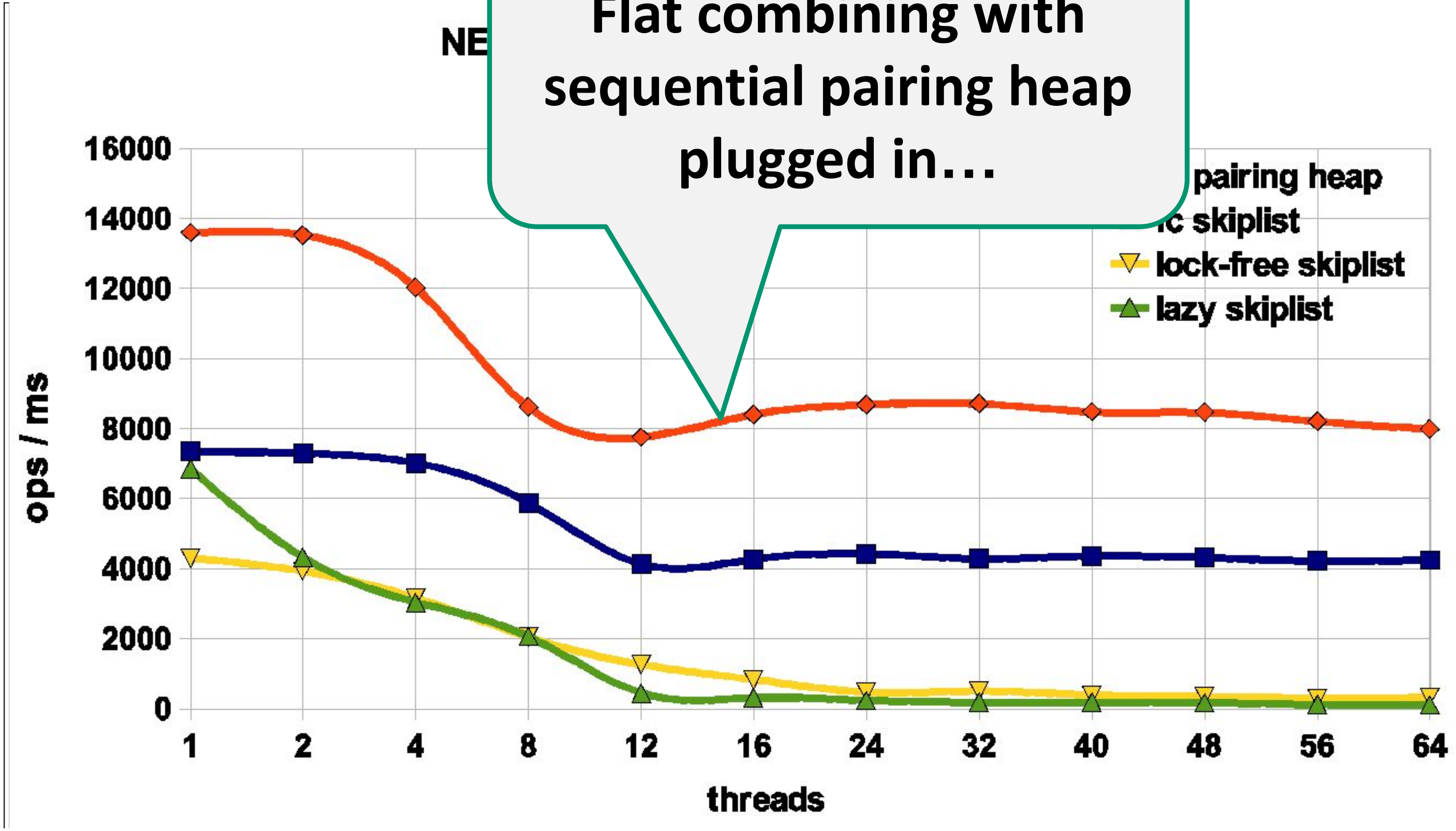
lock-free SkipQueue

rity Queue

Flat combining with sequential pairing heap plugged in…



Priority Queue on Intel



Flat combining with sequential pairing heap plugged in…

Don’t be Afraid of the Big Bad Lock

- Fine grained parallelism comes with an overhead…not always worth the effort.
- Sometimes using a single global lock is a win.