Scaling Ordered Stream Processing on Shared-Memory Multicores

Guna Prasaad

UNIVERSITY of WASHINGTON

G. Ramalingam, Kaushik Rajan

Microsoft Research

26 Aug, 2019

Realtime Stream Processing

- In today's world, data is of utmost value as it "*arrives*"
- Ability to process data in realtime is key to enabling several applications
- Stream processing has a very long history both inside and outside the database community
- New use-cases: surveillance, fraud detection, ad-serving, shopping cart analysis, online multiplayer games, live video streaming and distribution
- Processing large volumes of high-speed data in realtime is a challenge

Stream Processing Engines

- Allow users to define a computation pipeline that operates on a continuous stream of incoming data
- Architectures vary from a single core to shared-memory multicores to distributed shared-nothing
- Predominantly adopt the micro-batch architecture

APACHE kafka®
A distributed streaming platform

Trill

Apache Flink

Spark Streaming

APACHE STORM™

Shared-Memory Parallelism

Single shared-memory machine is "often" sufficient

- Streaming pipelines generally have a bounded memory footprint
- Tremendous growth in memory sizes and core counts

Building block for distributed stream processing engines

- Treat each core in a multicore machine as an individual node
- Fail to exploit low-overhead shared-memory parallelism

Ordered Stream Processing

Semantically equivalent to executing the stream computation on input stream serially one after another

Ordered Stream Processing

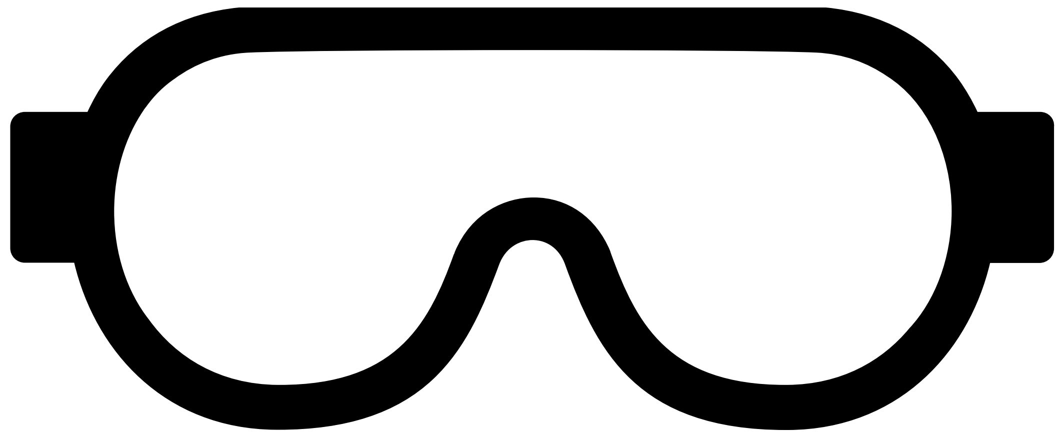
Semantically equivalent to executing the stream computation on input stream serially one after another

- Streams of events/tuples already have a notion of *temporal ordering*
- In many scenarios application logic depends on the event order
- For instance, timeout based sessions in a clickstream

Ordered Stream Processing

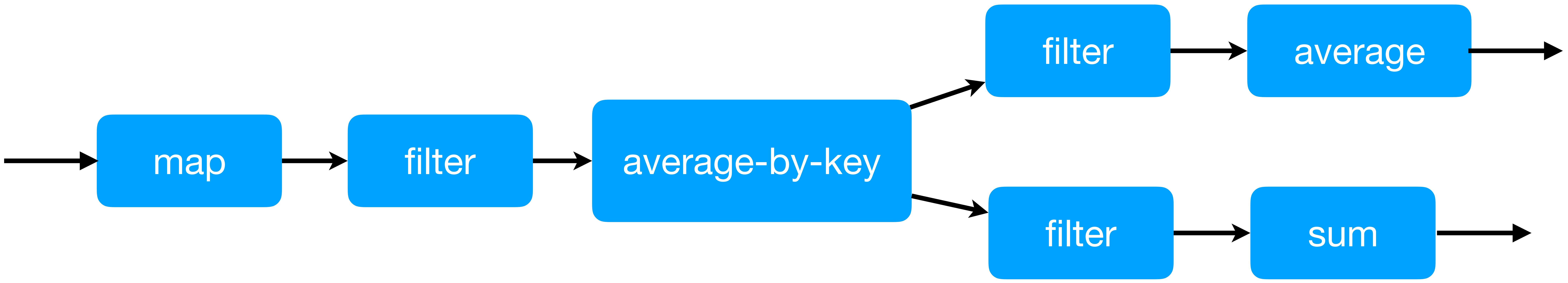
Semantically equivalent to executing the stream computation on input stream serially one after another

- Streams of events/tuples already have a notion of *temporal ordering*
- In many scenarios application logic depends on the event order
- For instance, timeout based sessions in a clickstream
- Easy deployment with fault-tolerance in the distributed setting
- Active replication requires deterministic processing guarantee

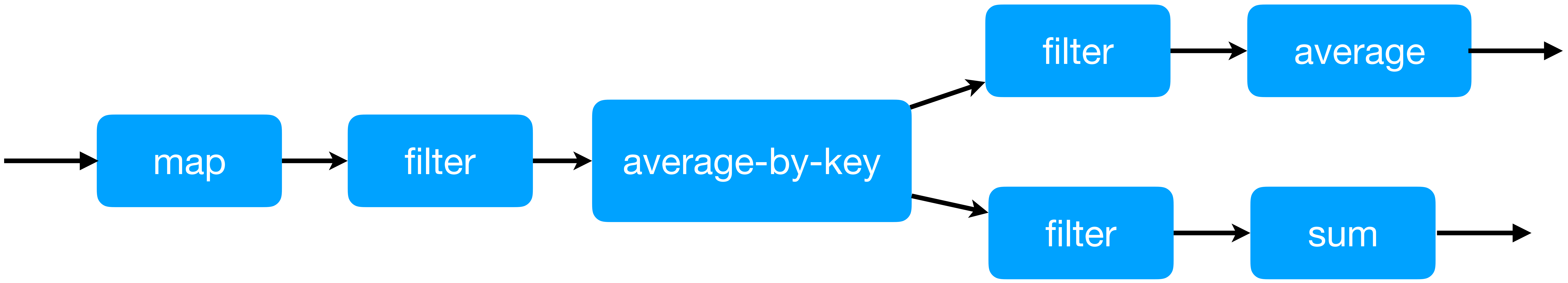


Overview

Parallelism in a Stream Computation

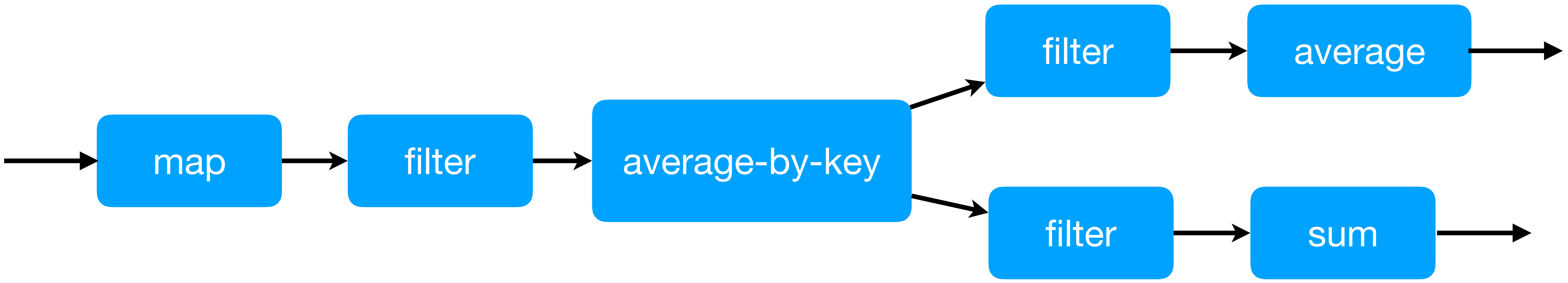


Parallelism in a Stream Computation



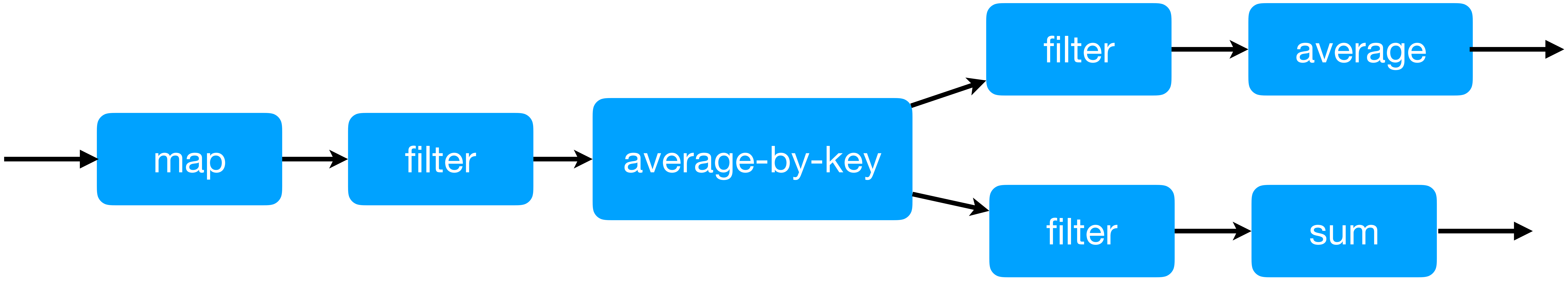
Data Parallelism

Parallelism in a Stream Computation



Data Parallelism

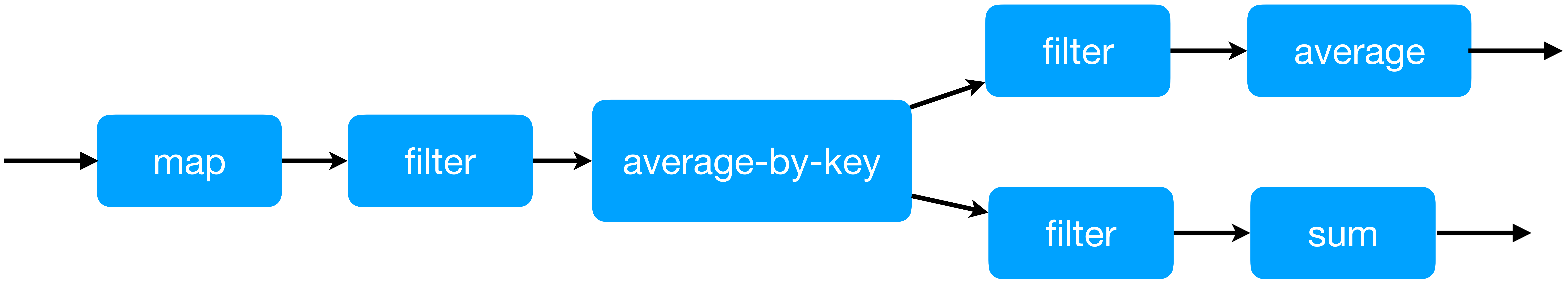
Parallelism in a Stream Computation



Pipeline Parallelism

Data Parallelism

Parallelism in a Stream Computation



Data Parallelism

Pipeline Parallelism

Task Parallelism

Order & Data Parallelism

$$\ldots, i_3, i_2, i_1 \rightarrow \blacklozenge \rightarrow \ldots o_3, o_2, o_1$$

Order & Data Parallelism

$$\ldots, i_3, i_2, i_1 \rightarrow \blacklozenge \rightarrow \ldots o_3, o_2, o_1$$

Value of Output

$$i_n \rightarrow \blacklozenge \rightarrow o_n$$

Order & Data Parallelism

$$\ldots, i_3, i_2, i_1 \rightarrow \blacklozenge \rightarrow \ldots o_3, o_2, o_1$$

Value of Output

$$i_n \rightarrow \blacklozenge \rightarrow o_n$$

Order of Outputs

$$\ldots, o_3, o_2, o_1$$

Ordering Semantics

If $i \prec i'$, when can i and i' be processed out-of-order?

Ordering Semantics

If $i \prec i'$, when can i and i' be processed out-of-order?

$$(o_n, S_n) = \textbf{operate}(S_{n-1}, i_n)$$

Ordering Semantics

If $i \prec i'$, when can i and i' be processed out-of-order?

$$(o_n, S_n) = \textbf{operate}(S_{n-1}, i_n)$$

Stateless

Ordering Semantics

If $i \prec i'$, when can i and i' be processed out-of-order?

$$(o_n, S_n) = \textbf{operate}(S_{n-1}, i_n)$$

Stateless

Always

Ordering Semantics

If $i \prec i'$, when can i and i' be processed out-of-order?

$$(o_n, S_n) = \textbf{operate}(S_{n-1}, i_n)$$

Stateless

Stateful

Always

Ordering Semantics

If $i \prec i'$, when can i and i' be processed out-of-order?

$$(o_n, S_n) = \textbf{operate}(S_{n-1}, i_n)$$

Stateless

Always

Stateful

Never

Ordering Semantics

If $i \prec i'$, when can i and i' be processed out-of-order?

$$(o_n, S_n) = \textbf{operate}(S_{n-1}, i_n)$$

Stateless

Stateful

Partitionable Stateful

Always

Never

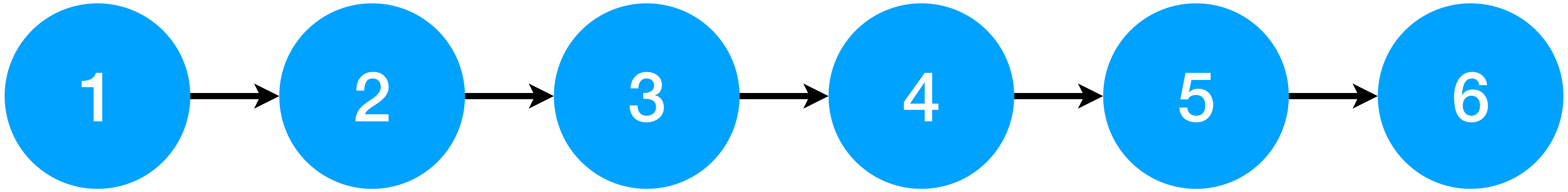
Ordering Semantics

If $i \prec i'$, when can i and i' be processed out-of-order?

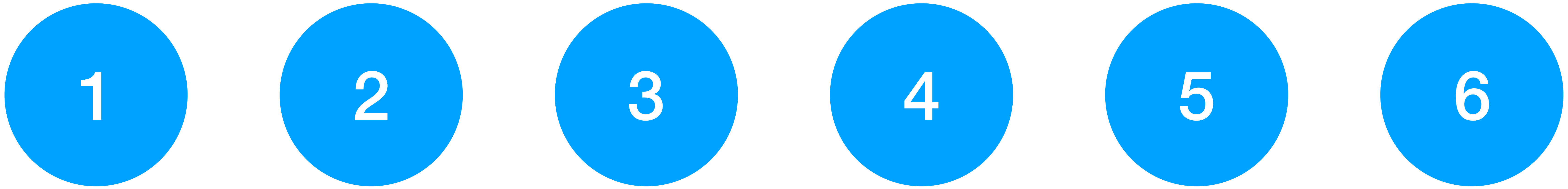
$$(o_n, S_n) = \textbf{operate}(S_{n-1}, i_n)$$

Stateless	Stateful	Partitionable Stateful
Always	**Never**	$\mathbb{P}(i) \neq \mathbb{P}(i')$

Stream Dataflow Graph

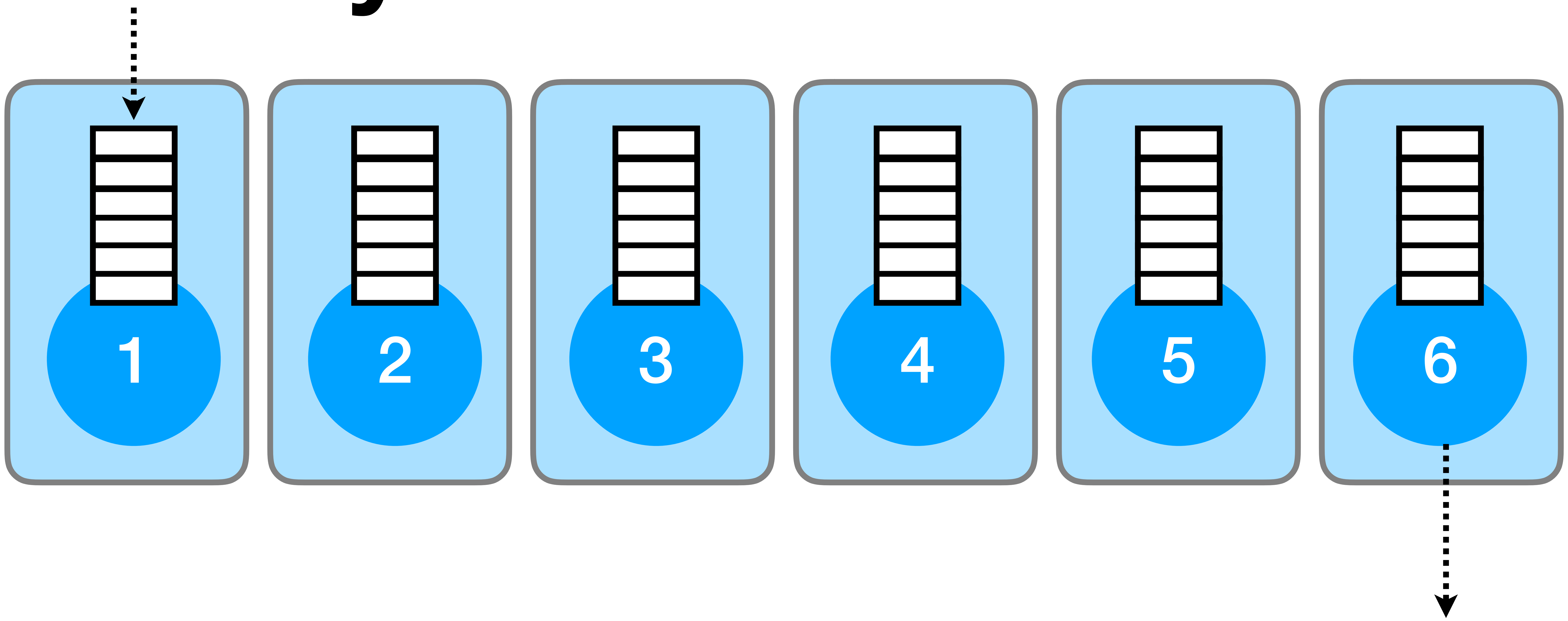


Async Executable



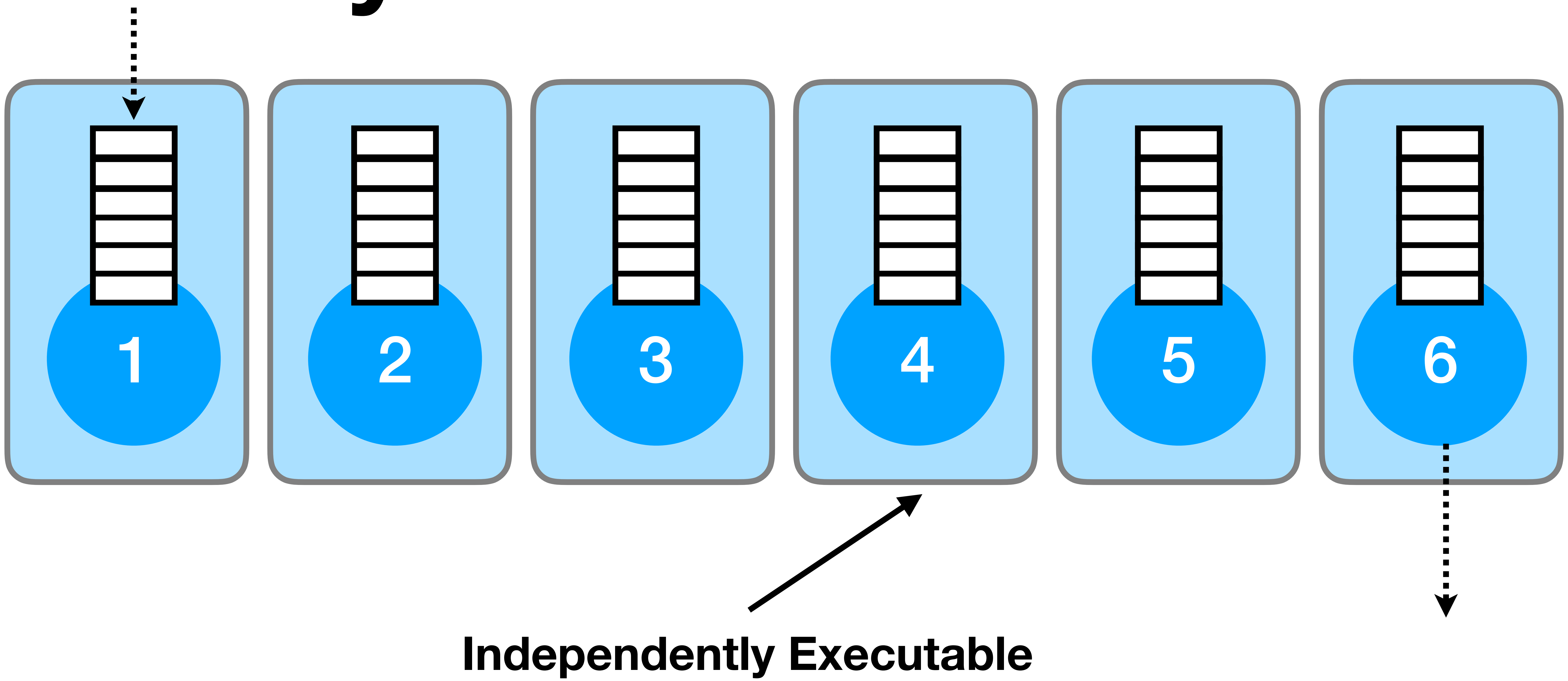
Decouple operators by allowing inputs to be processed “asynchronously”

Async Executable



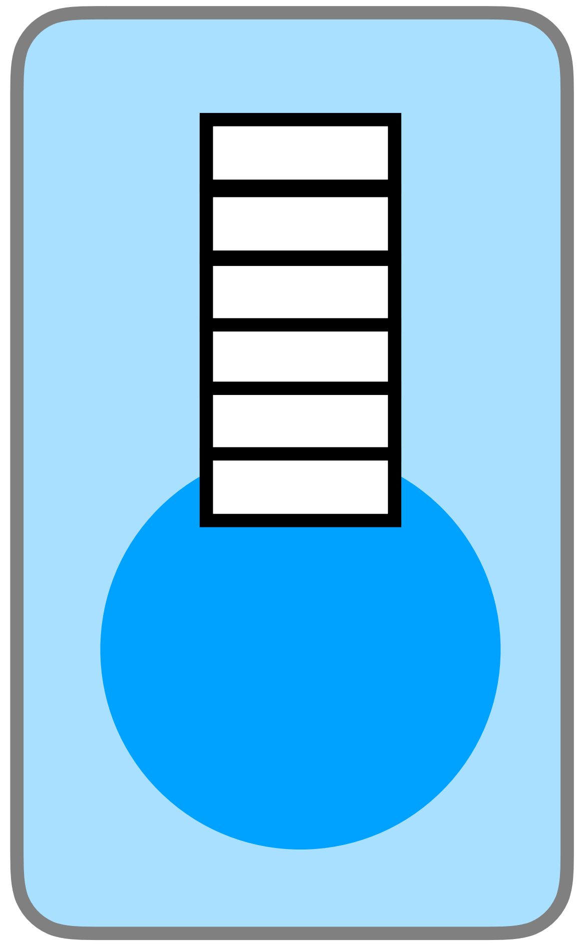
Decouple operators by allowing inputs to be processed “asynchronously”

Async Executable



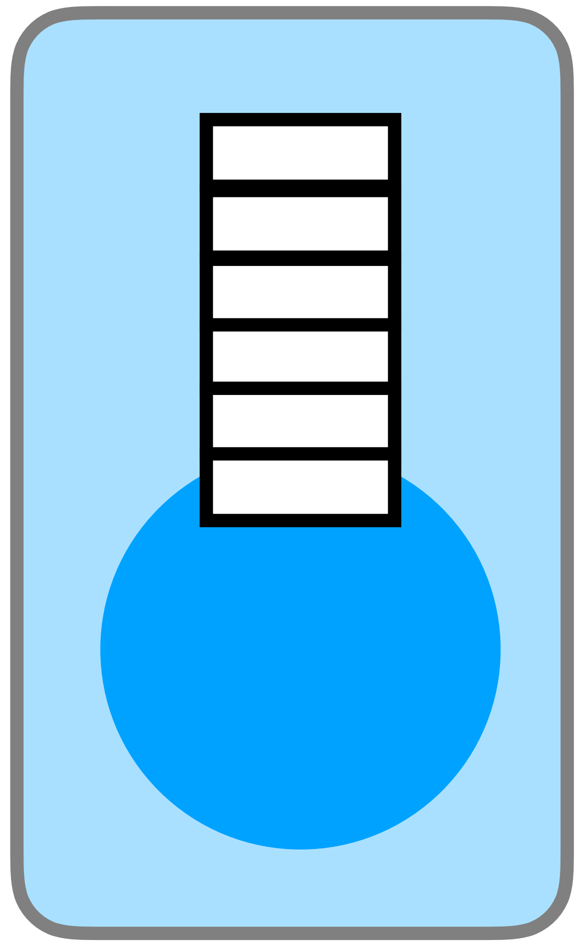
Decouple operators by allowing inputs to be processed “asynchronously”

Key Requirement



Each operator executable must individually provide the ordering guarantee when multiple workers are allotted

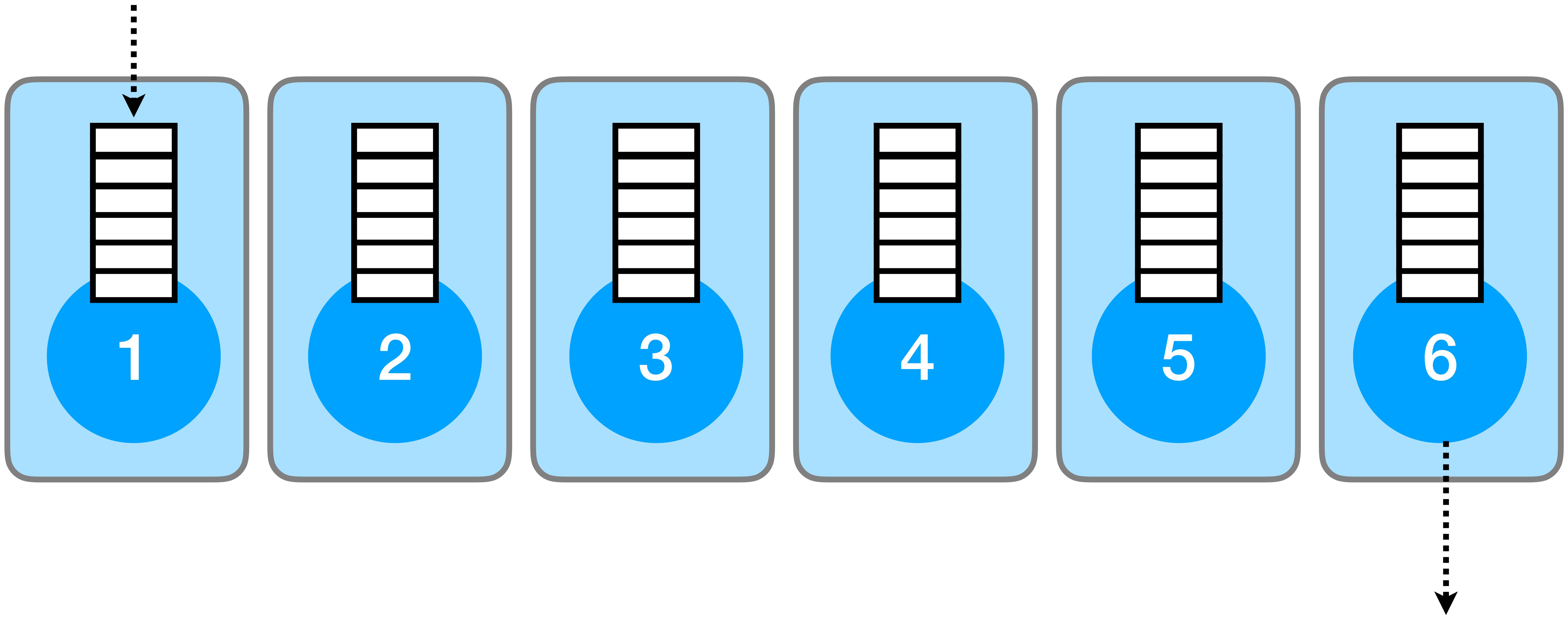
Key Requirement



Each operator executable must individually provide the ordering guarantee when multiple workers are allotted

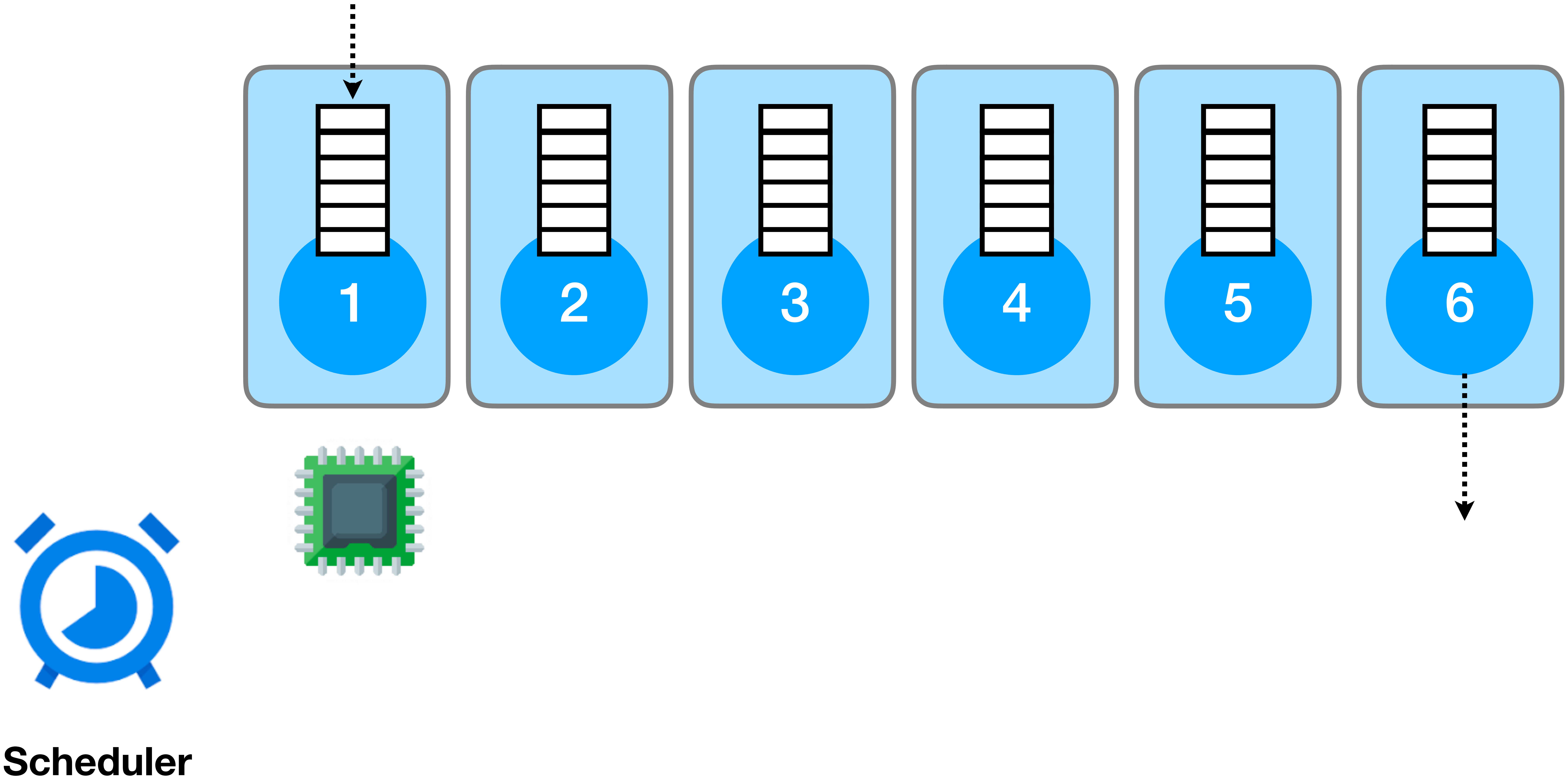
Clear separation of concerns between correctness and optimisation

Scheduler

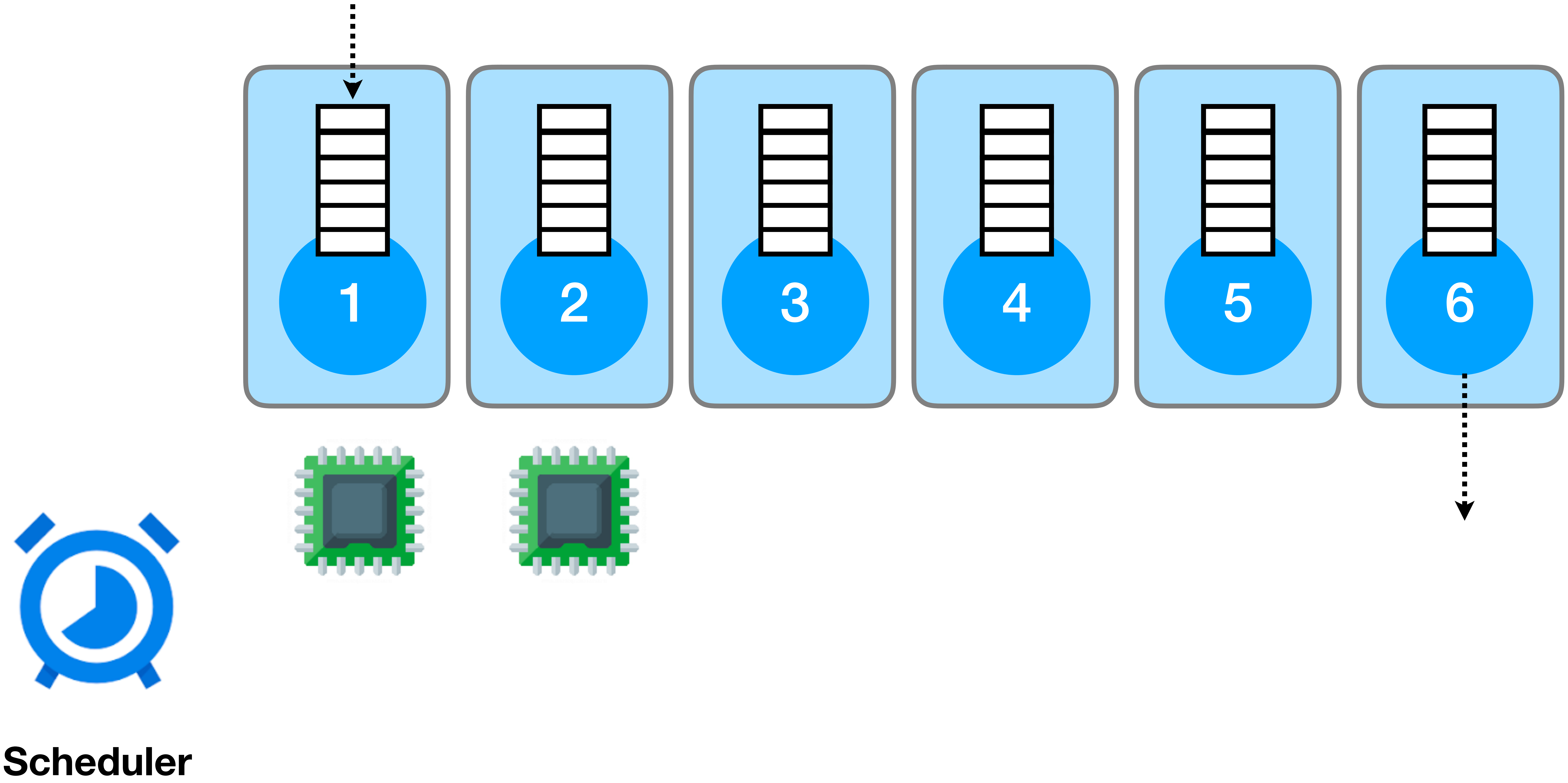


Scheduler

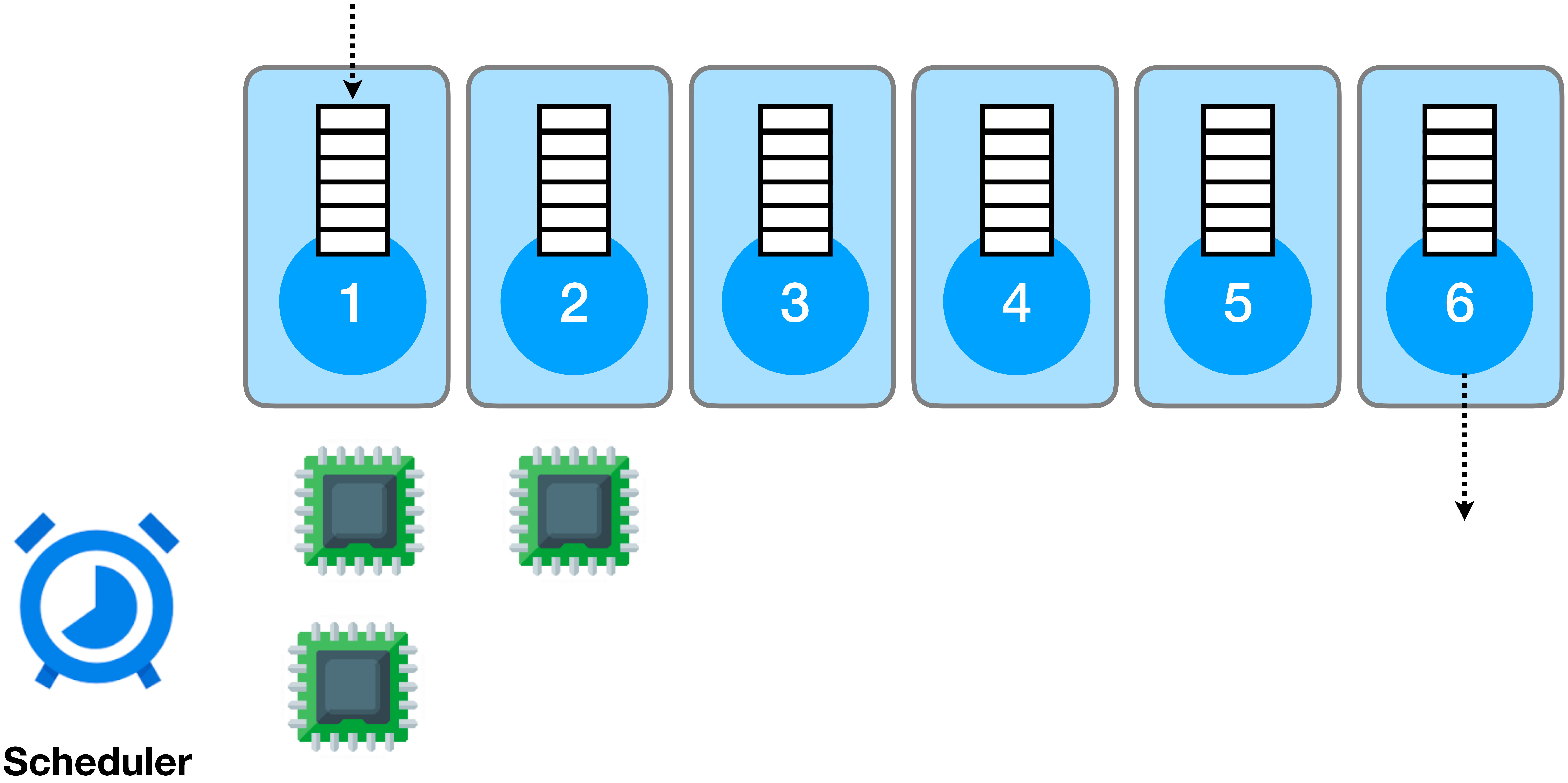
Scheduler



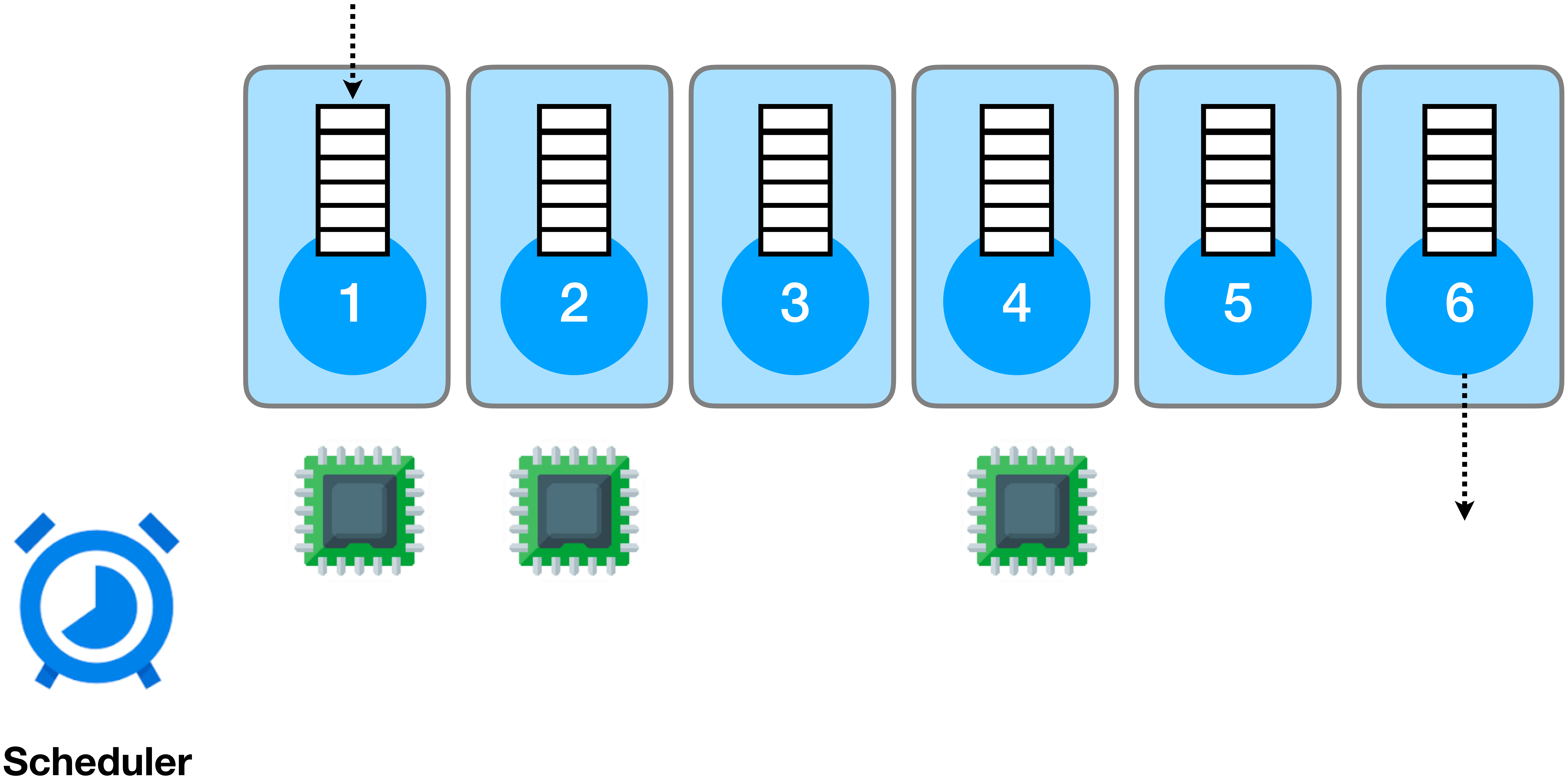
Scheduler



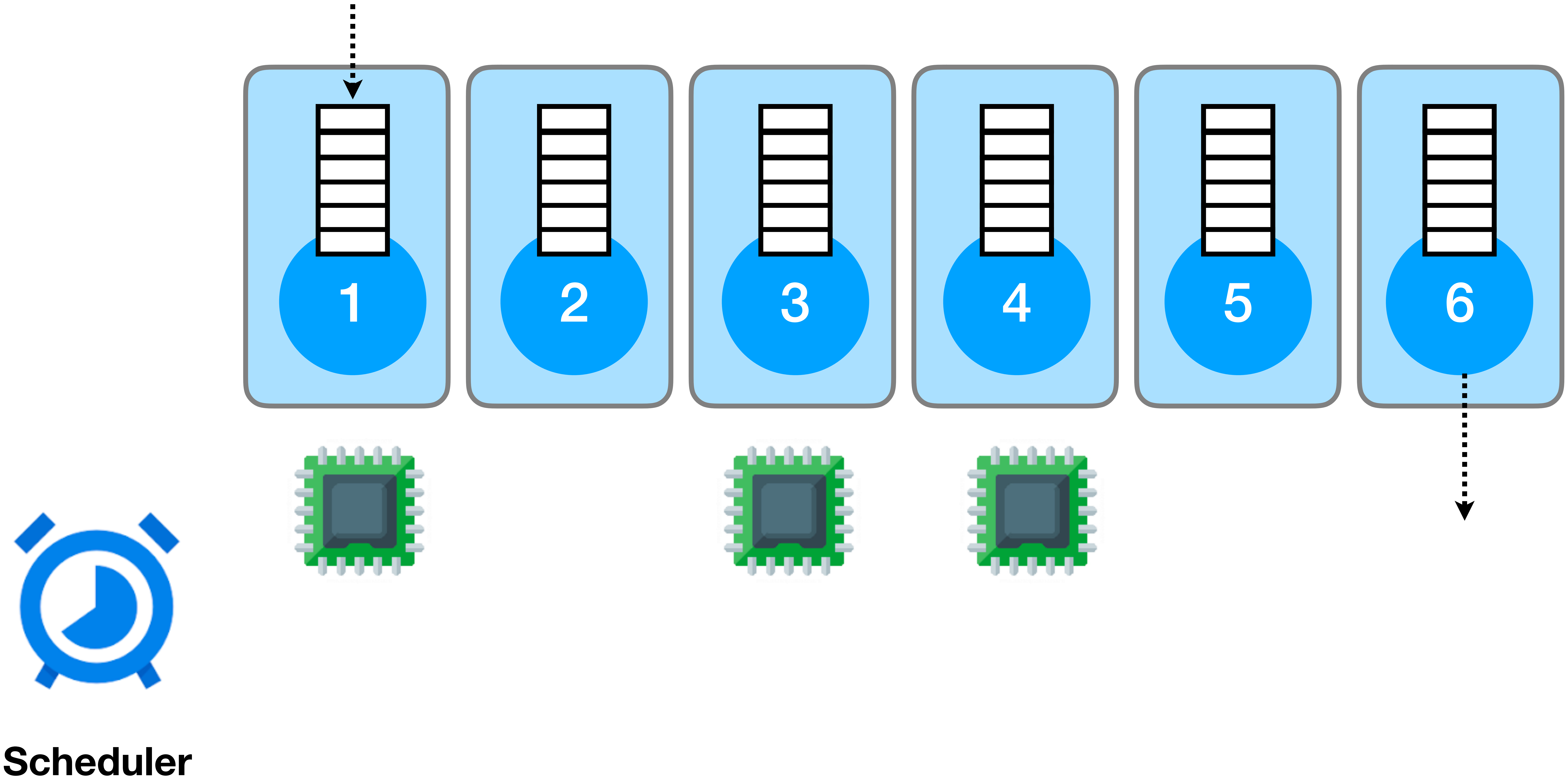
Scheduler



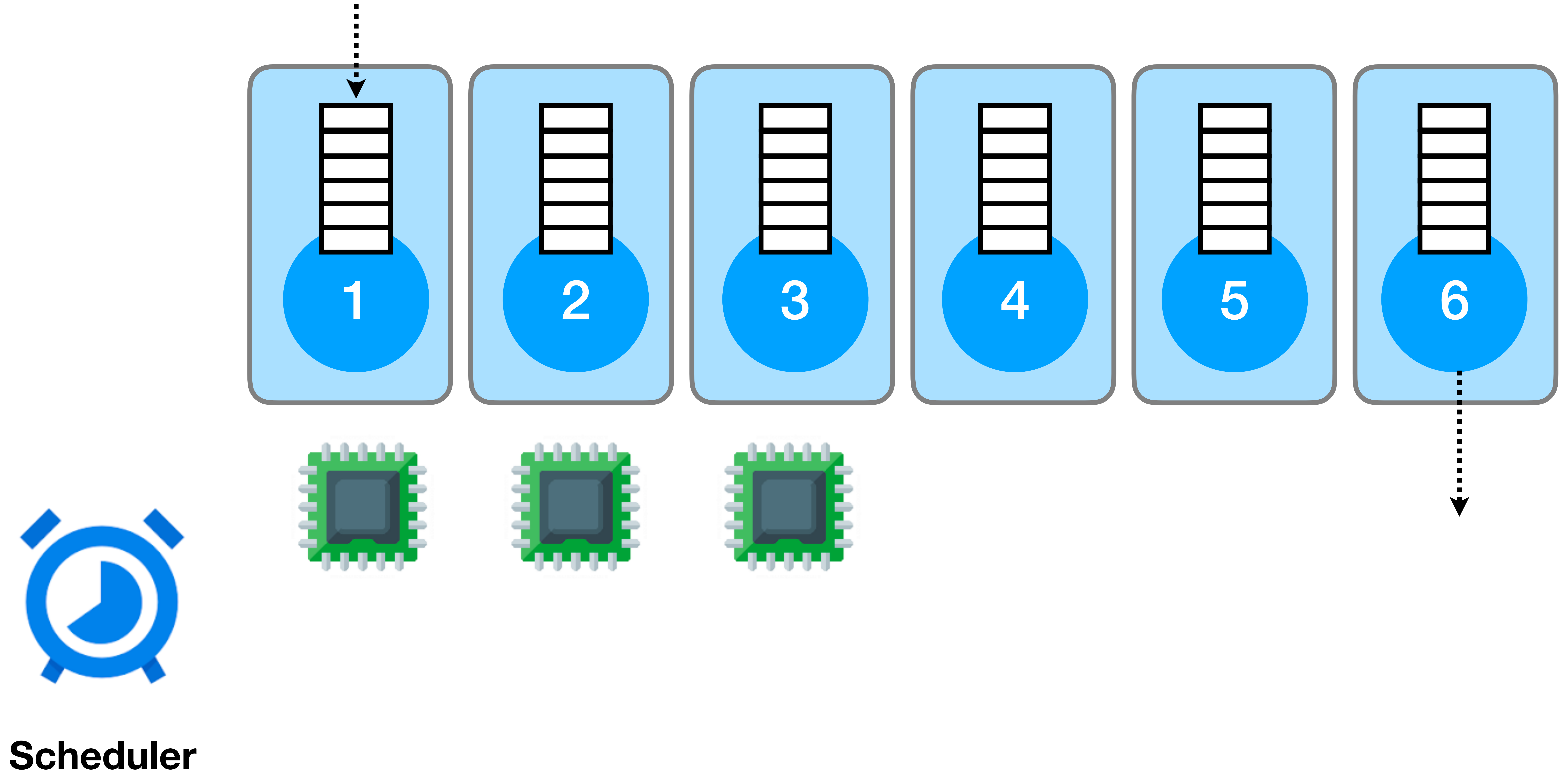
Scheduler



Scheduler



Scheduler



Outline

- Reordering Outputs
- Partitionable Stateful Operators
- Scheduling Runtime
- Evaluation
- Conclusion

Reordering Outputs

Reordering Outputs

- Each input is assigned a "*sequence number*" based on their arrival
- Concurrent workers are operating on inputs to produce outputs
- We want to reorder and send them downstream in the input arrival order
- Output o_{i+1} , even if produced earlier, can only be sent downstream after all of $o_1, o_2, \ldots, o_i$ have been sent

Lock-Based Reordering

```
void send(o_t) {
  lock();
  if (t ≠ next) {
    add o_t to buffer
  } else {
    send_downstream(o_t);
    next++;
    while(buffer has o_next) {
      send_downstream(o_next);
      next++;
    }
  }
  unlock();
}
```

Lock-Based Reordering

```
void send(o_t) {
  lock();
  if (t ≠ next) {
    add o_t to buffer
  } else {
    send_downstream(o_t);
    next++;
    while(buffer has o_next) {
      send_downstream(o_next);
      next++;
    }
  }
  unlock();
}
```

Lock-Based Reordering

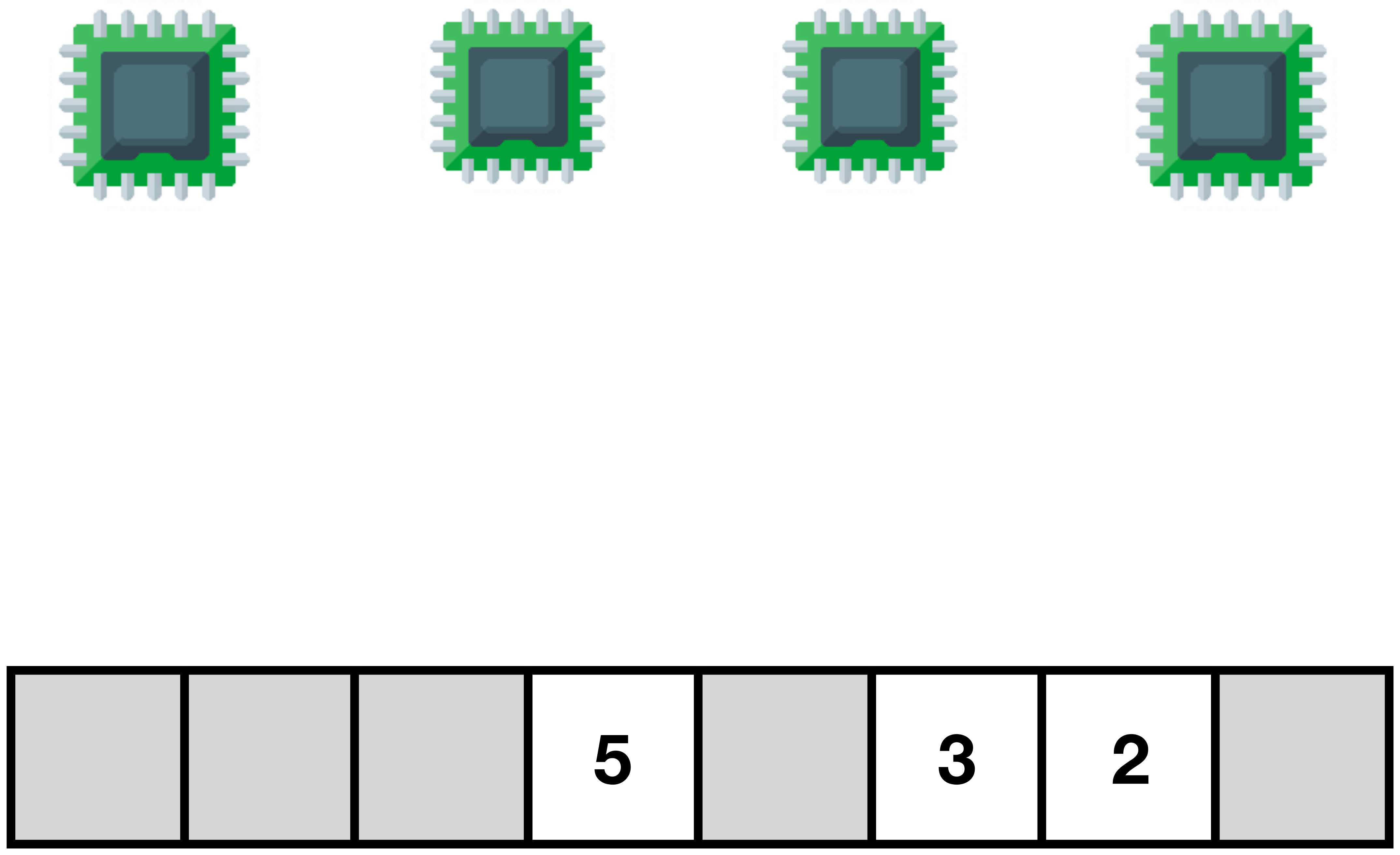
```
void send(o_t) {
  lock();
  if (t ≠ next) {
    add o_t to buffer
  } else {
    send_downstream(o_t);
    next++;
    while(buffer has o_next) {
      send_downstream(o_next);
      next++;
    }
  }
  unlock();
}
```

Lock-Based Reordering

```
void send(o_t) {
  lock();
  if (t ≠ next) {
    add o_t to buffer
  } else {
    send_downstream(o_t);
    next++;
    while(buffer has o_next) {
      send_downstream(o_next);
      next++;
    }
  }
  unlock();
}
```

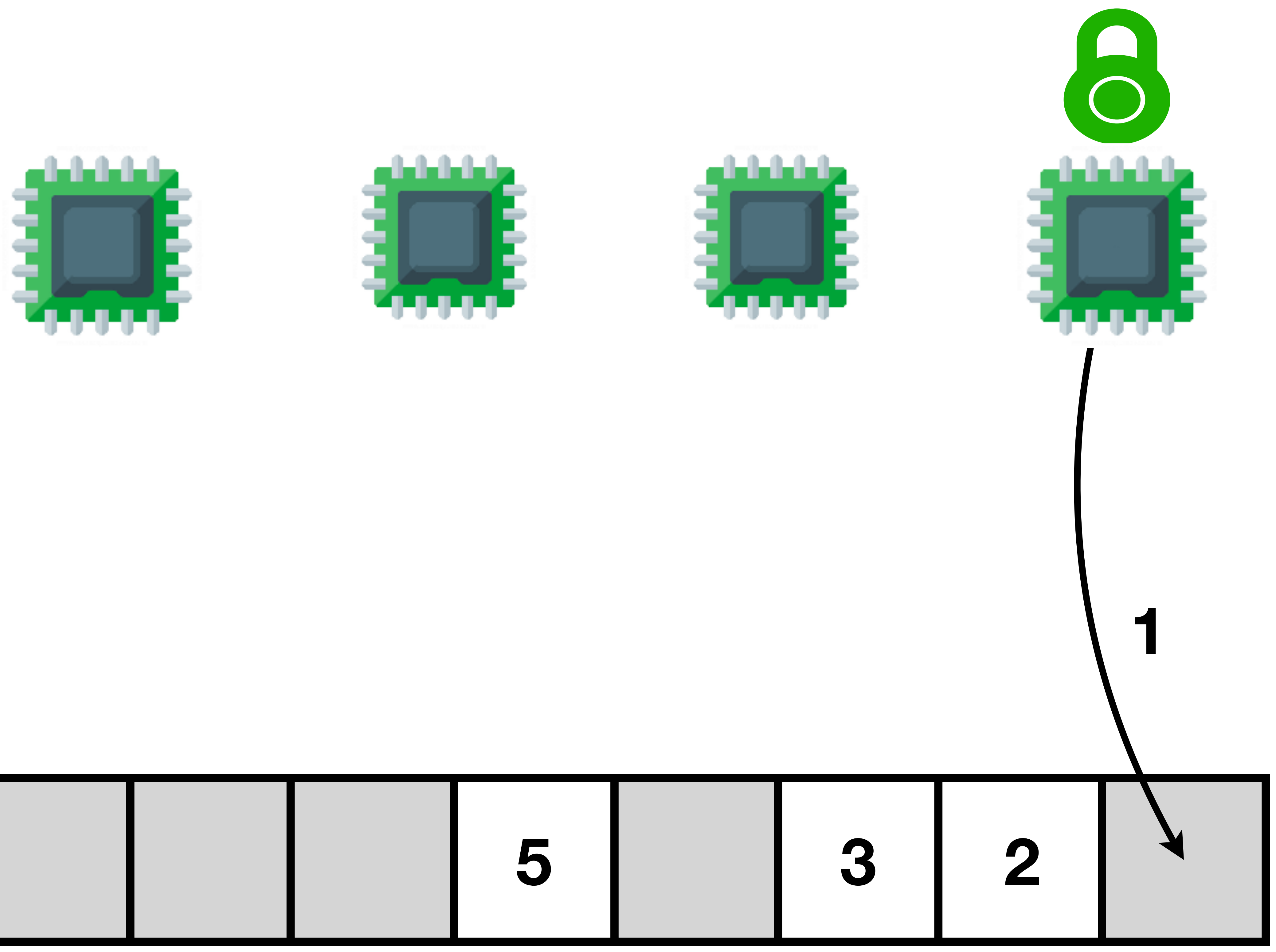
Lock-Based Reordering

```
void send(o_t) {
  lock();
  if (t ≠ next) {
    add o_t to buffer
  } else {
    send_downstream(o_t);
    next++;
    while(buffer has o_next) {
      send_downstream(o_next);
      next++;
    }
  }
  unlock();
}
```



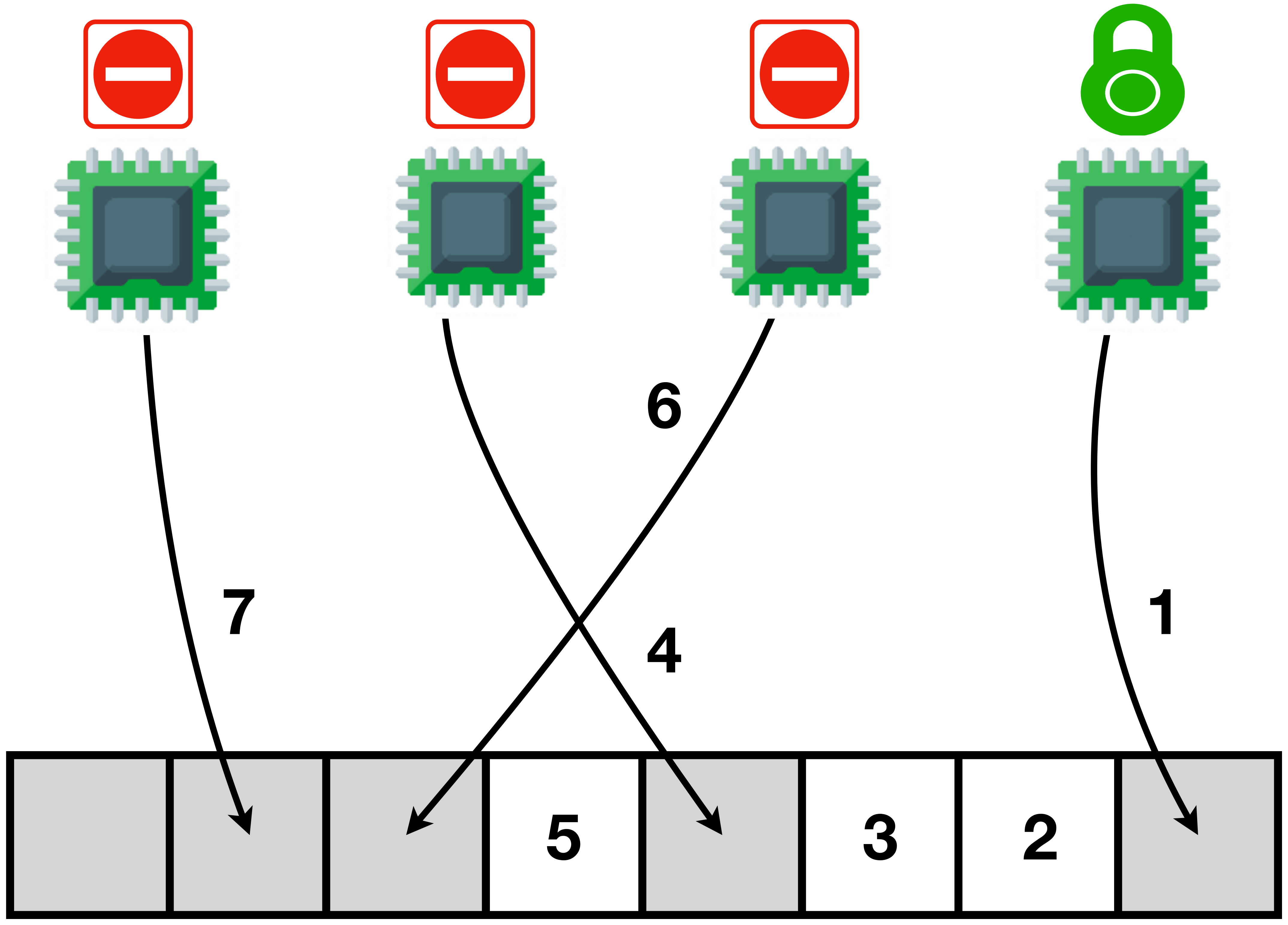
Lock-Based Reordering

```
void send(o_t) {
  lock();
  if (t ≠ next) {
    add o_t to buffer
  } else {
    send_downstream(o_t);
    next++;
    while(buffer has o_next) {
      send_downstream(o_next);
      next++;
    }
  }
  unlock();
}
```



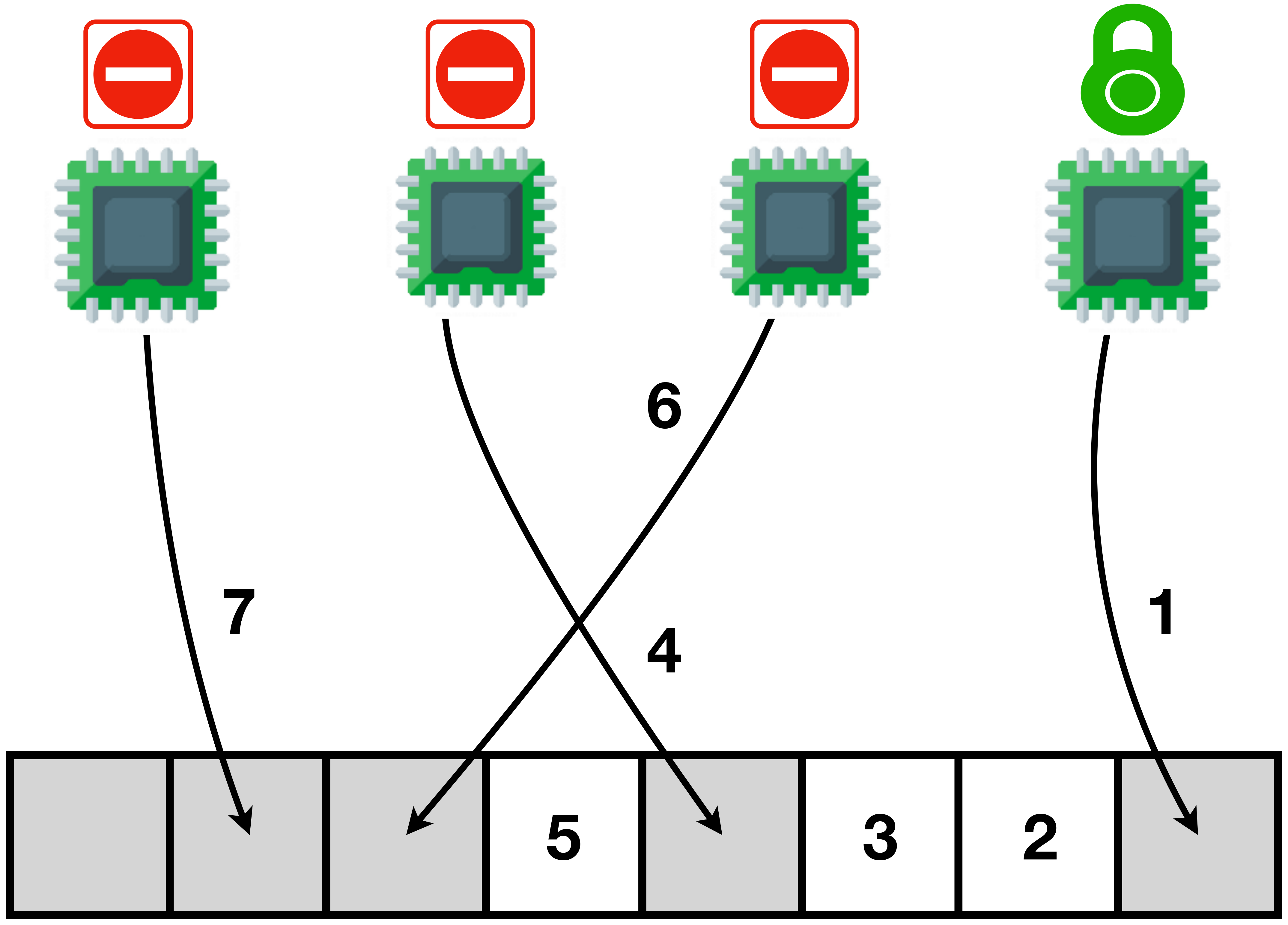
Lock-Based Reordering

```
void send(o_t) {
  lock();
  if (t ≠ next) {
    add o_t to buffer
  } else {
    send_downstream(o_t);
    next++;
    while(buffer has o_next) {
      send_downstream(o_next);
      next++;
    }
  }
  unlock();
}
```



Lock-Based Reordering

- Each output already has a designated location on the buffer
- Decouple adding to buffer from sending downstream
- If a worker is already sending outputs downstream, delegate the work to it and return to do more useful work



Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```

Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```

Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```

Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```

Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```

Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```

Our Solution: Non-Blocking Reordering

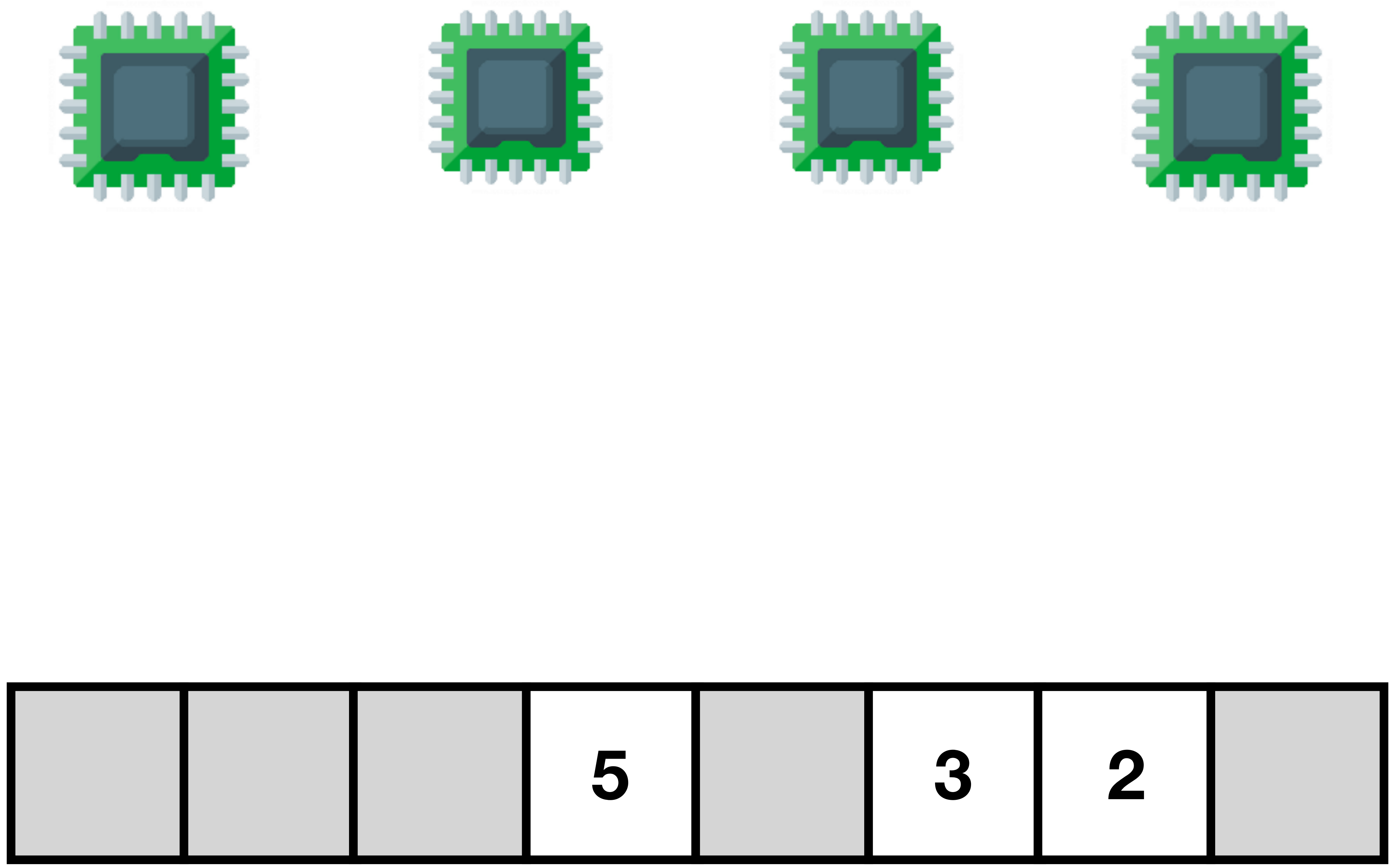
```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```

Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```

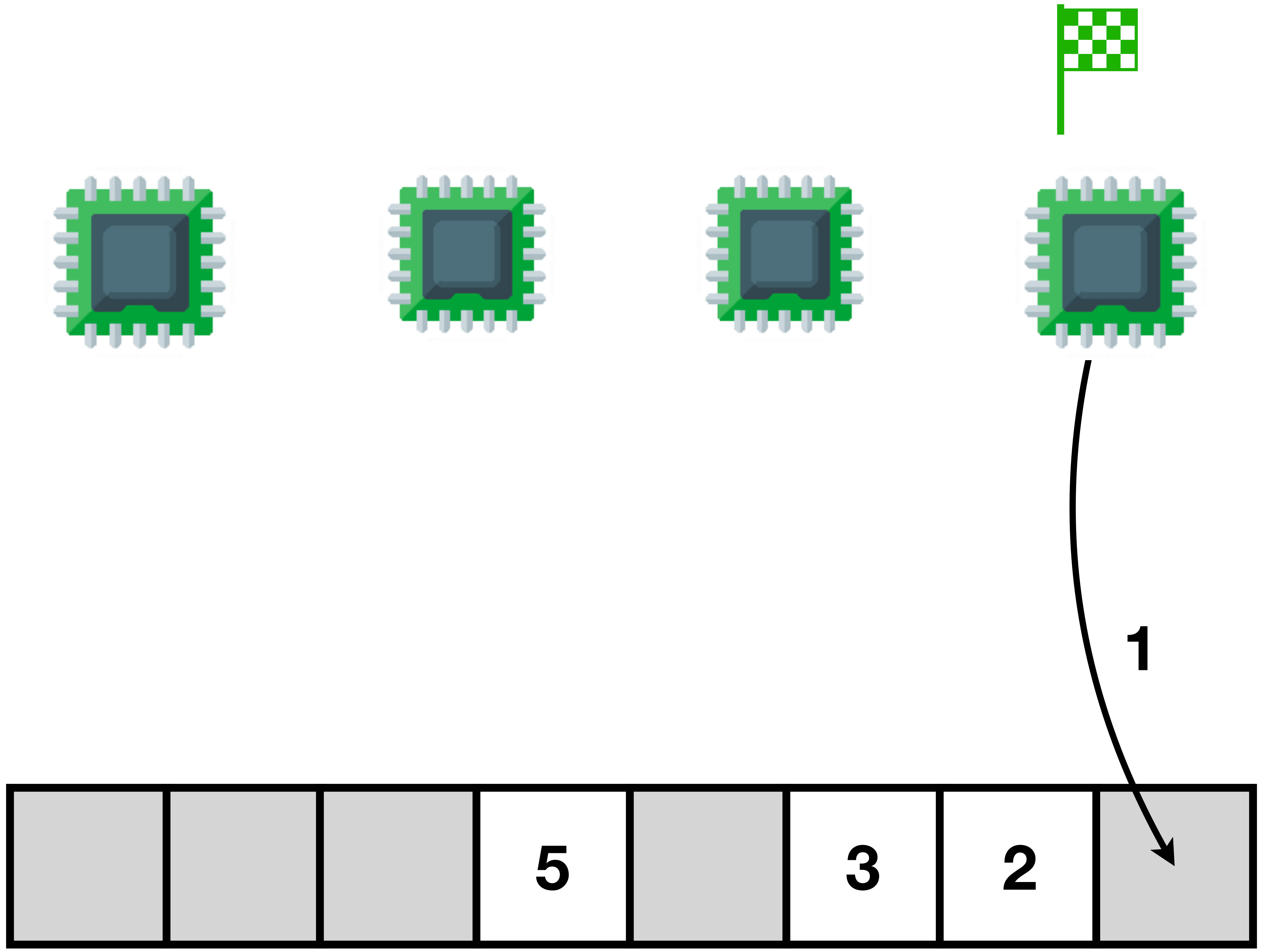
Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```



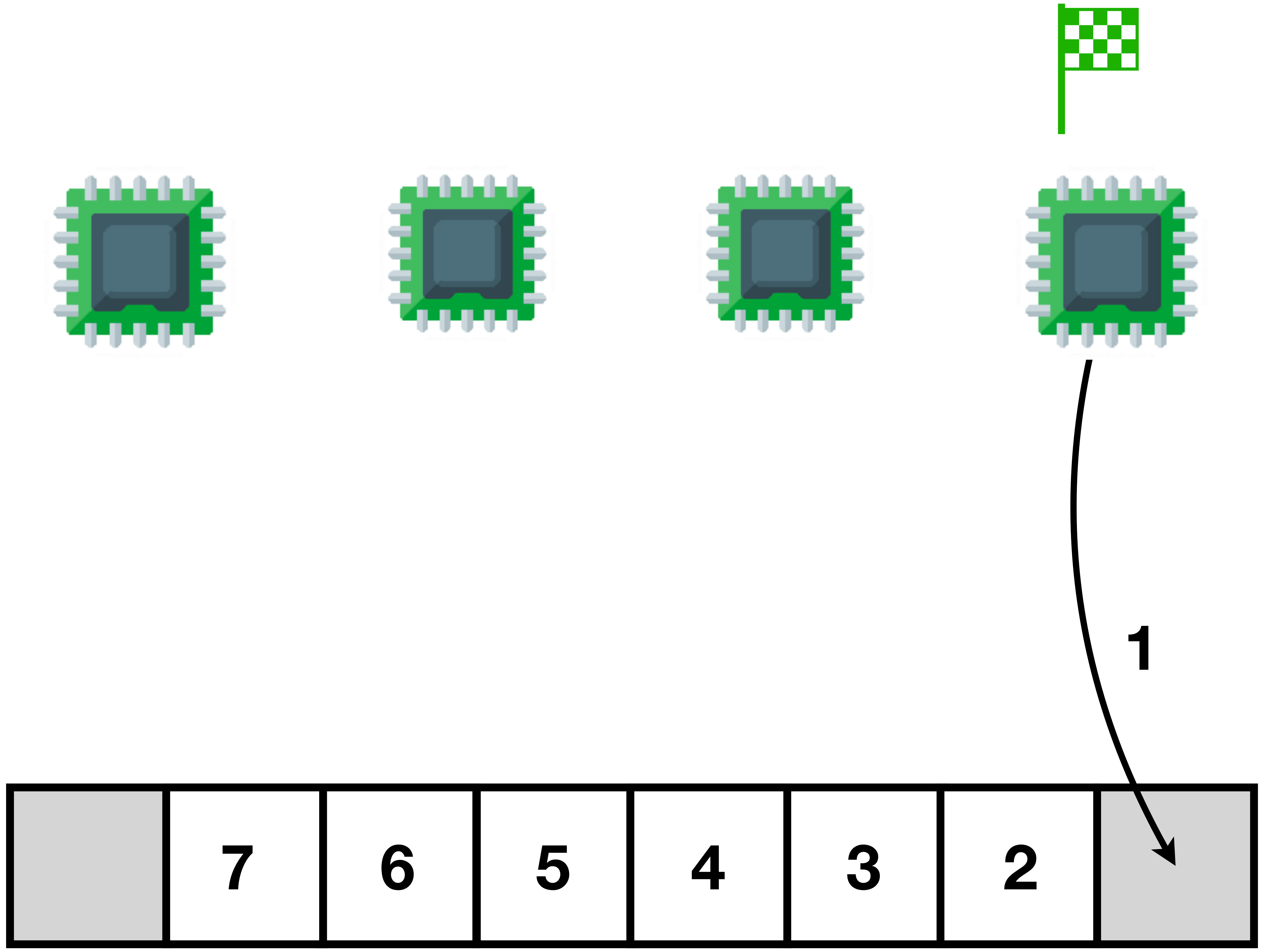
Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```



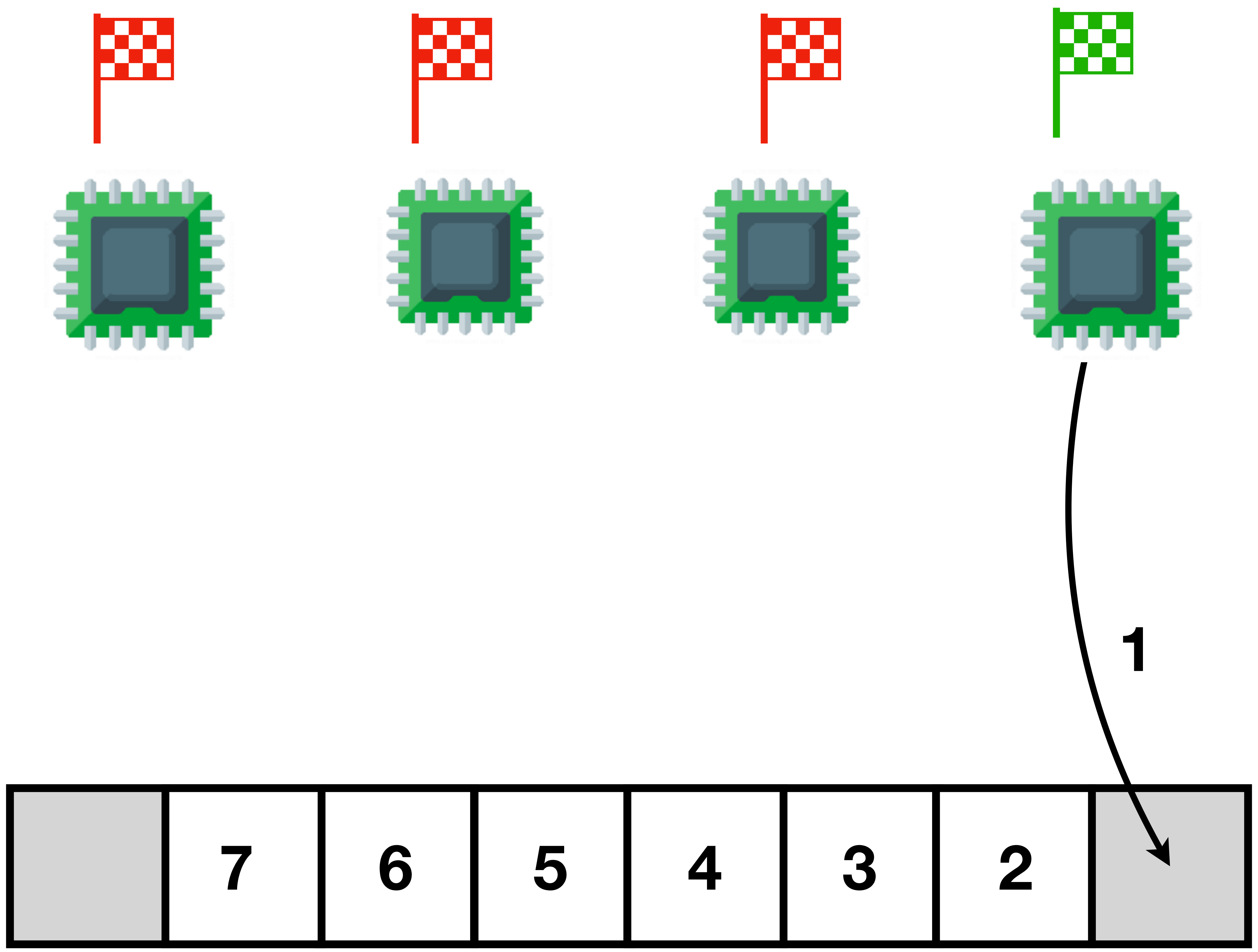
Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```



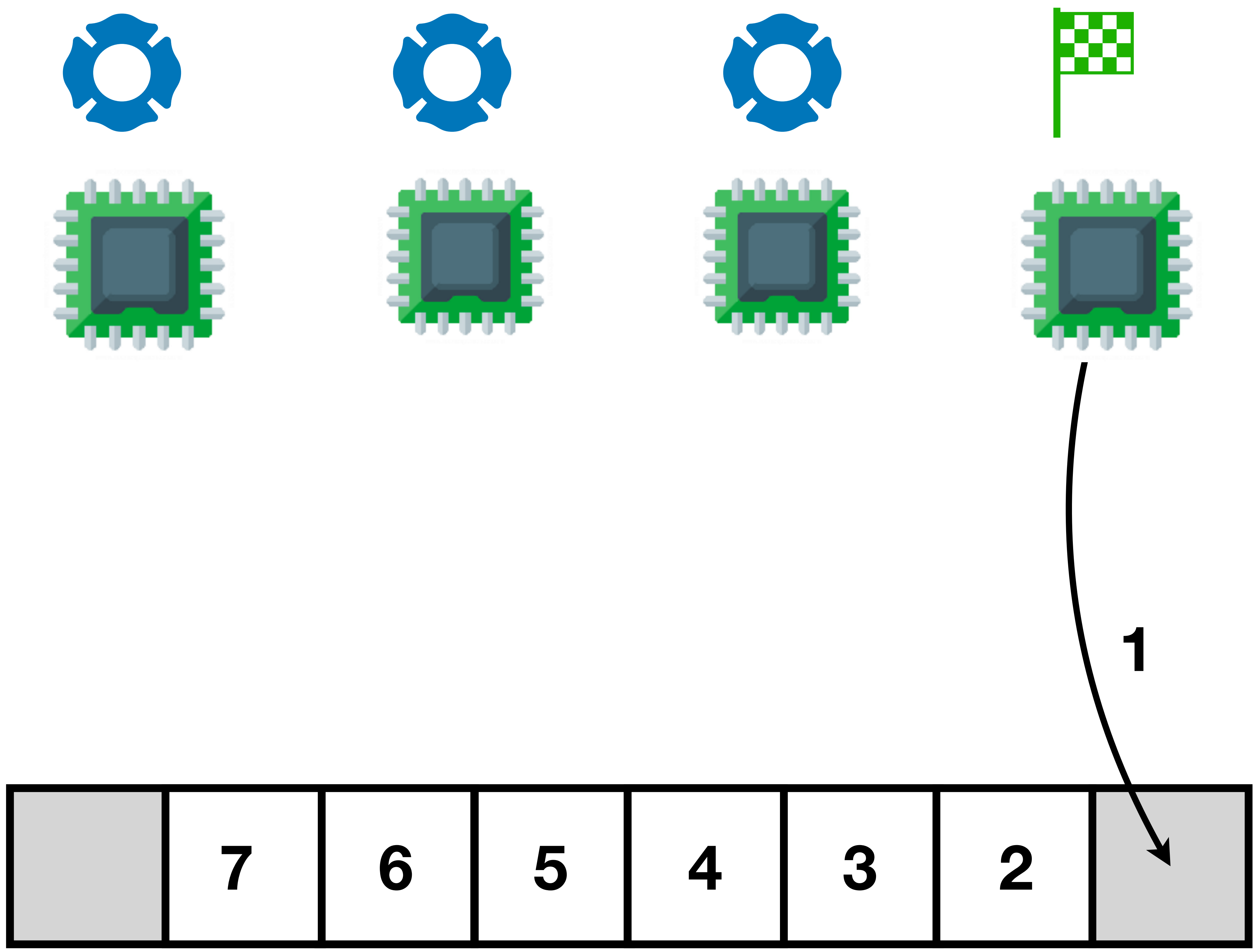
Our Solution: Non-Blocking Reordering

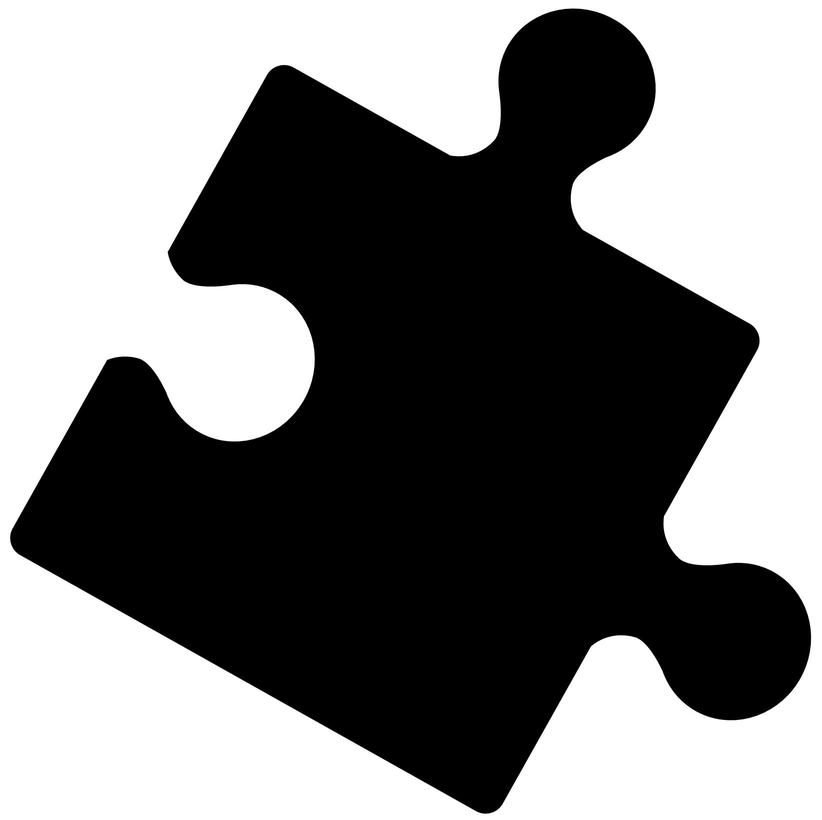
```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```



Our Solution: Non-Blocking Reordering

```
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
    break;
    }
  }
  return success;
}
```





Partitionable Stateful Operators

Partitionable Stateful Operators

We have been working on group-by-aggregates for several decades, what’s new?

Partitionable Stateful Operators

We have been working on group-by-aggregates for several decades, what's new?

Latency-Critical

Ordering Requirement

Partitionable Stateful Operators

We have been working on group-by-aggregates for several decades, what's new?

Latency-Critical

Ordering Requirement

If $i \prec i'$, they can be processed out-of-order (or concurrently) only when

$$\mathbb{P}(i) \neq \mathbb{P}(i')$$

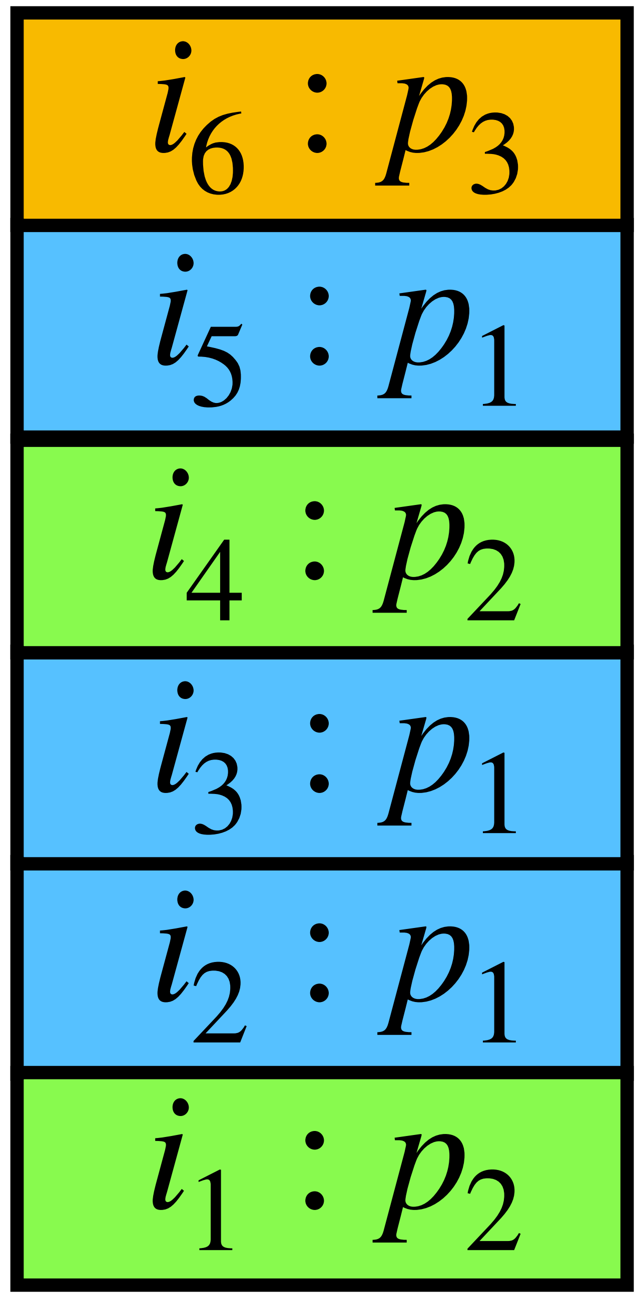
Shared Queue

- Each tuple has an associated partition

- All inputs are added into a single shared linearizable concurrent queue

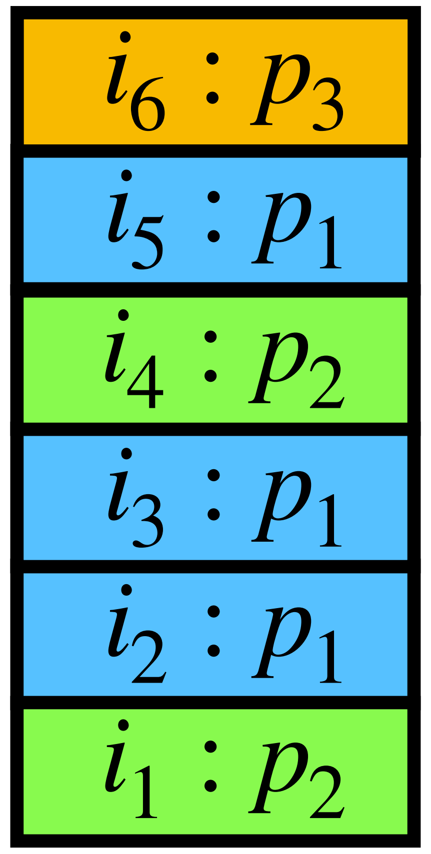
Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue



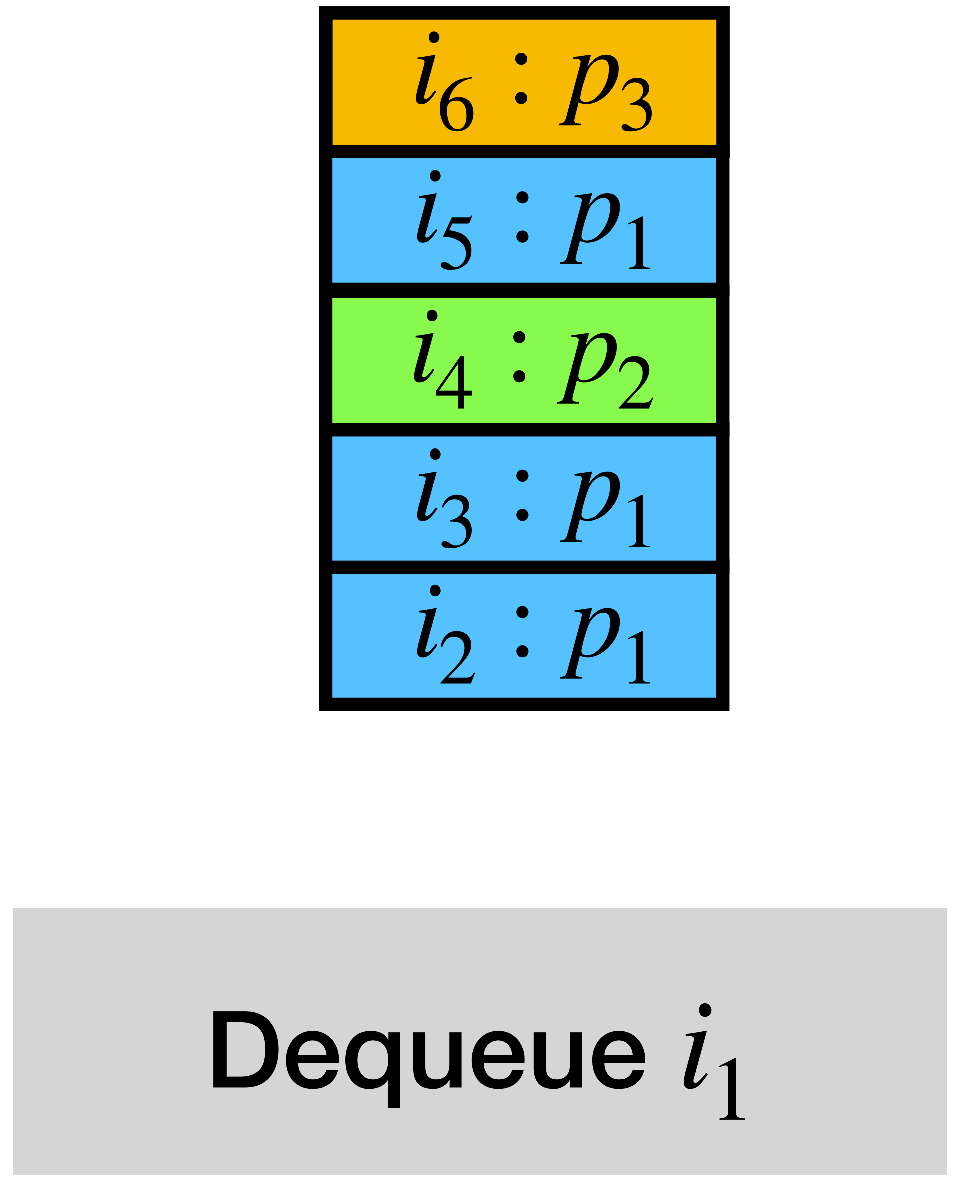
Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition



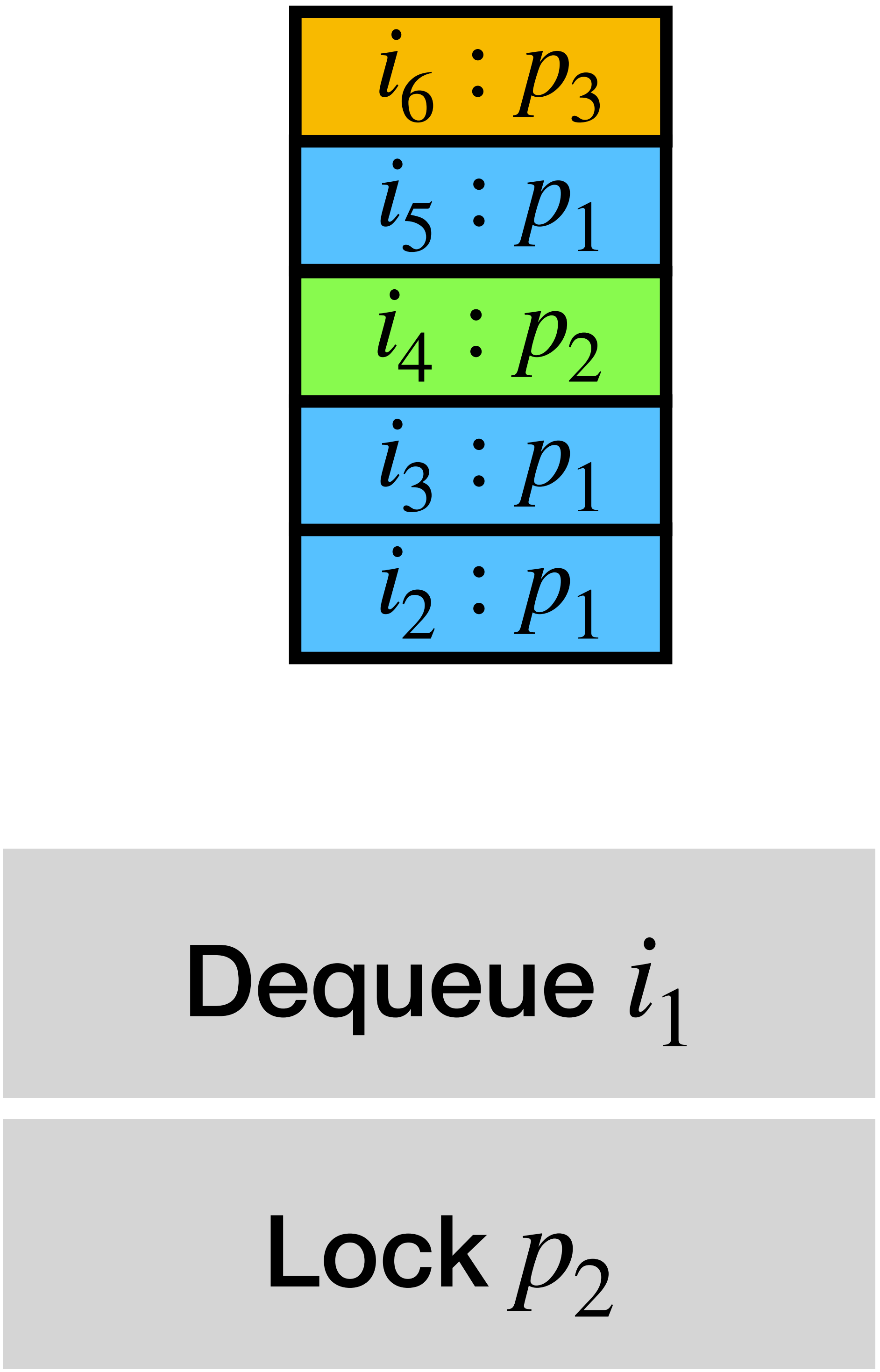
Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition



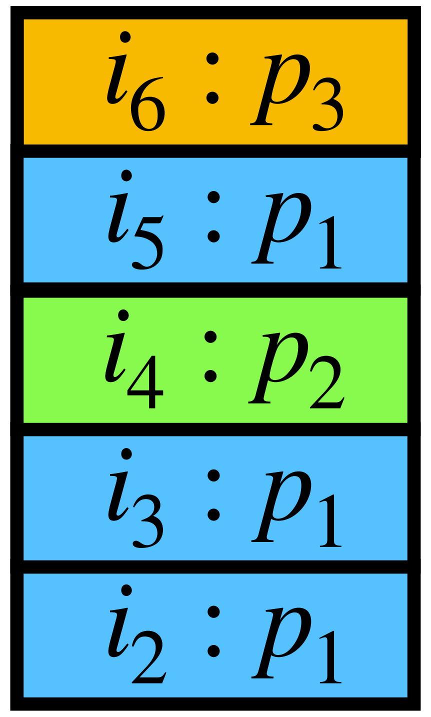
Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
    1. Dequeue input
    2. Lock partition
    3. Operate on input
    4. Unlock partition



Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition



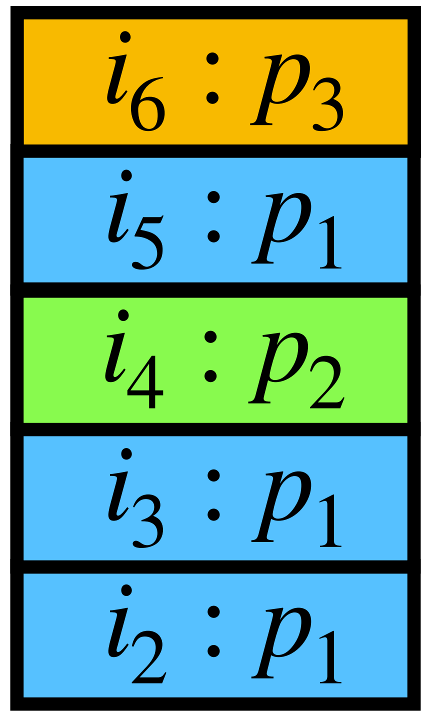
Dequeue i_1

Lock p_2

Operate on i_1

Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition



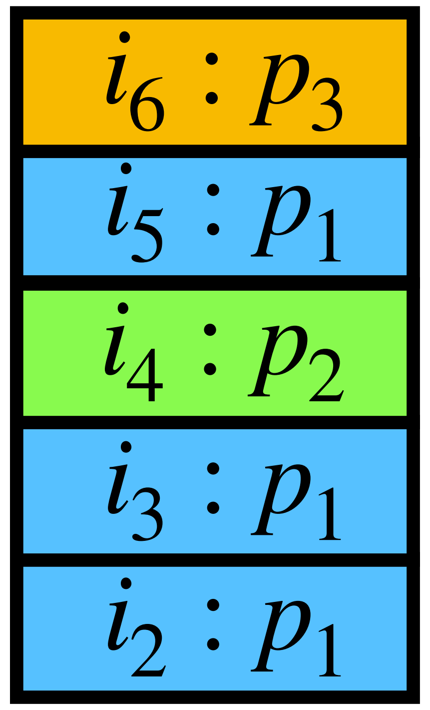
Dequeue i_1

Lock p_2

Operate on i_1

Unlock p_2

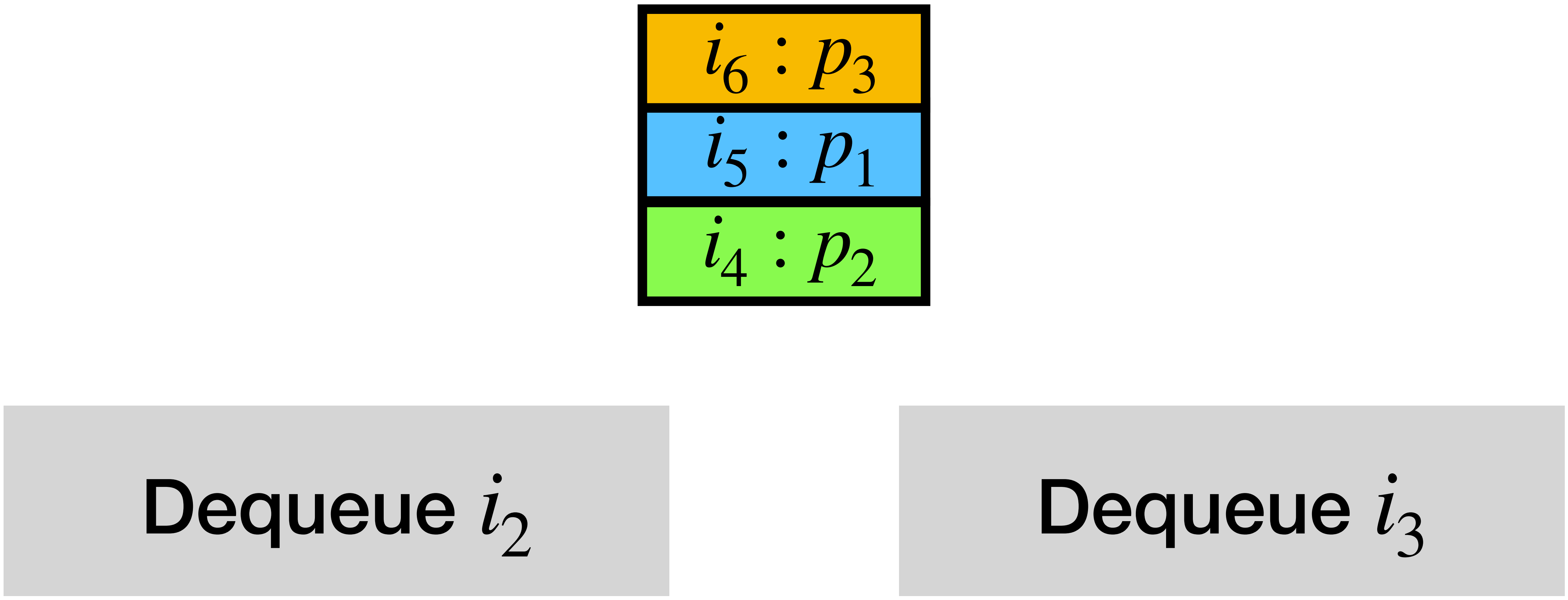
Shared Queue



- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition

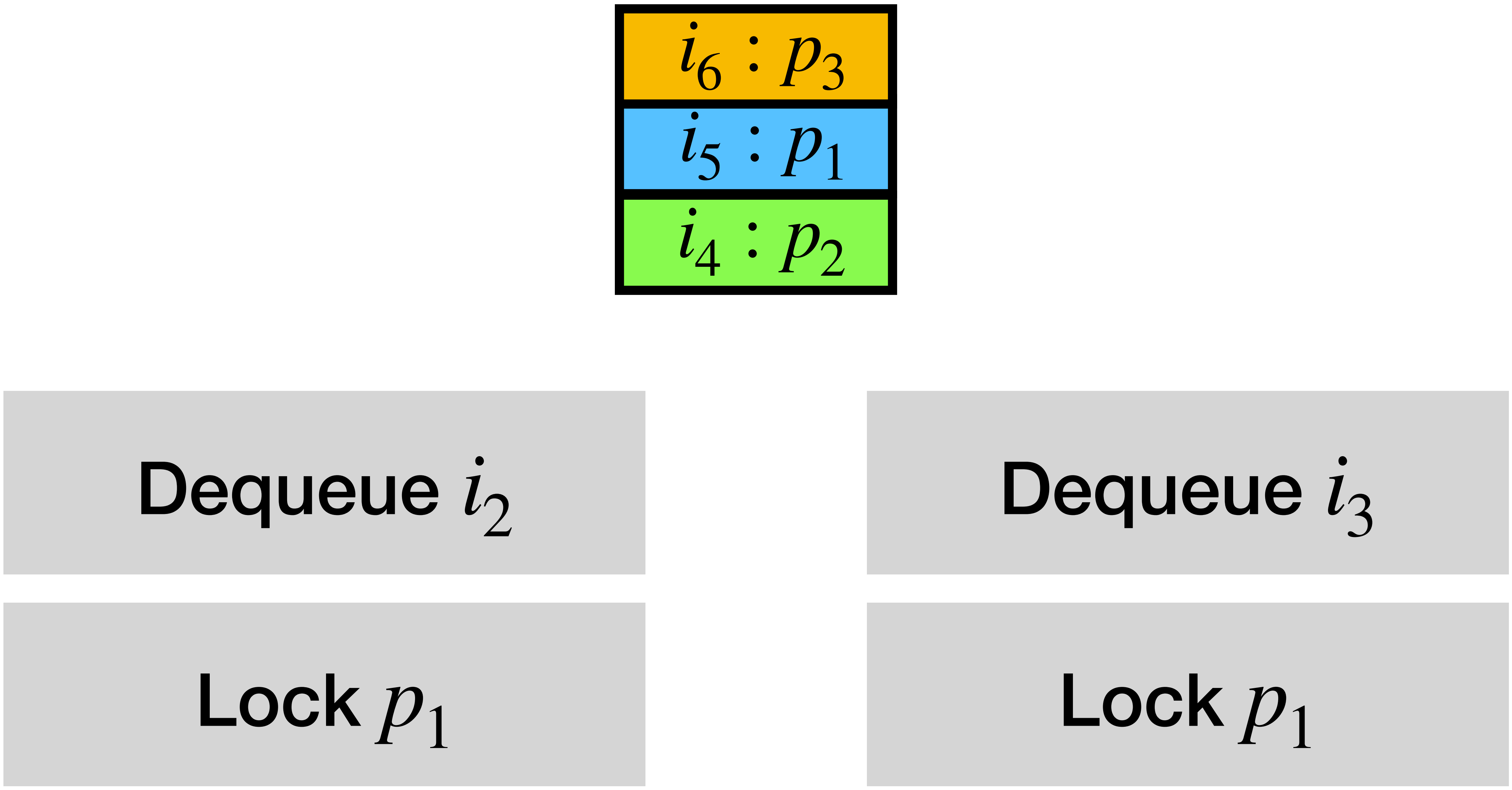
Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
    1. Dequeue input
    2. Lock partition
    3. Operate on input
    4. Unlock partition



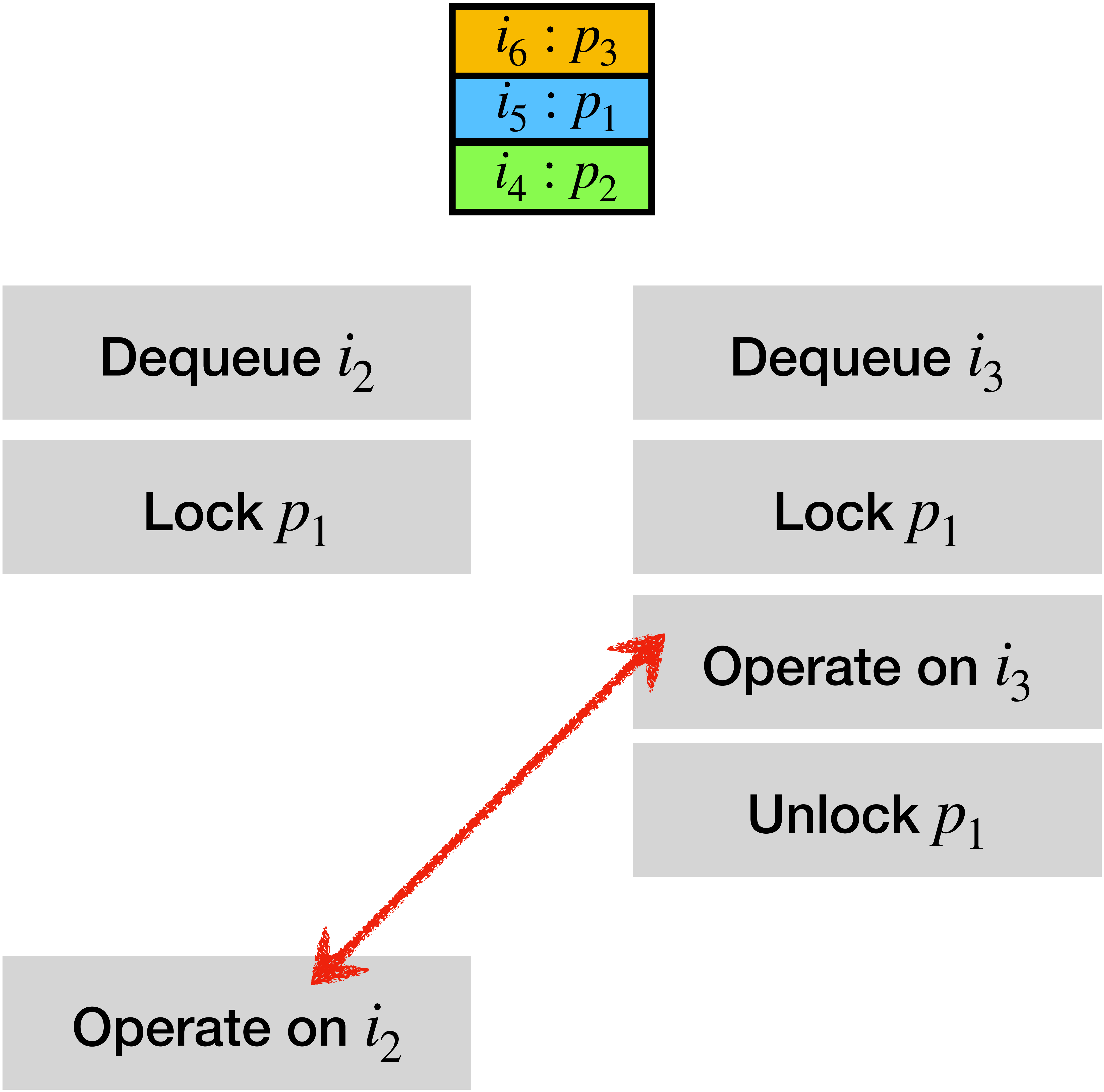
Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition



Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition

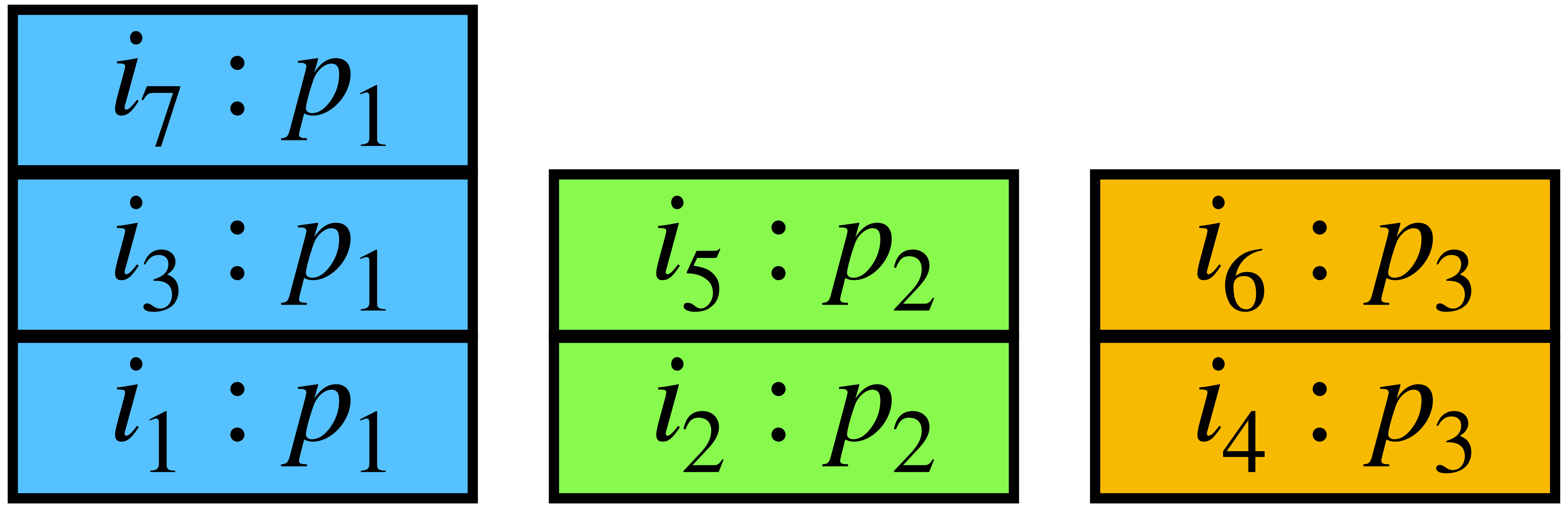


Partitioned Queues

- Consider each partition as a stateful operator with its own queue
- At most one worker can process a partition
- Most commonly used strategy in all stream processing engines
- Unnecessary blocking of outputs in the reordering buffer

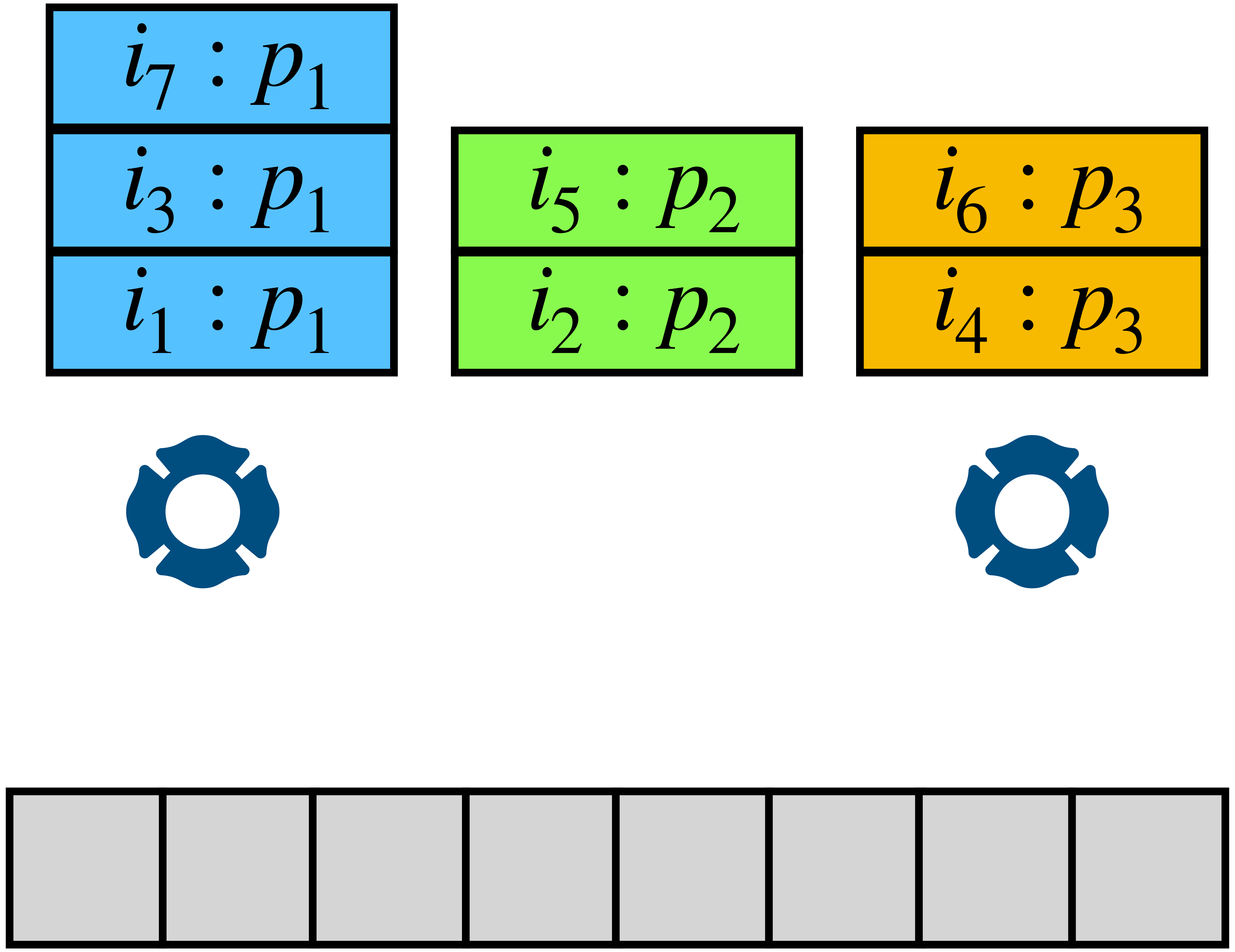
Partitioned Queues

- Consider each partition as a stateful operator with its own queue
- At most one worker can process a partition
- Most commonly used strategy in all stream processing engines
- Unnecessary blocking of outputs in the reordering buffer



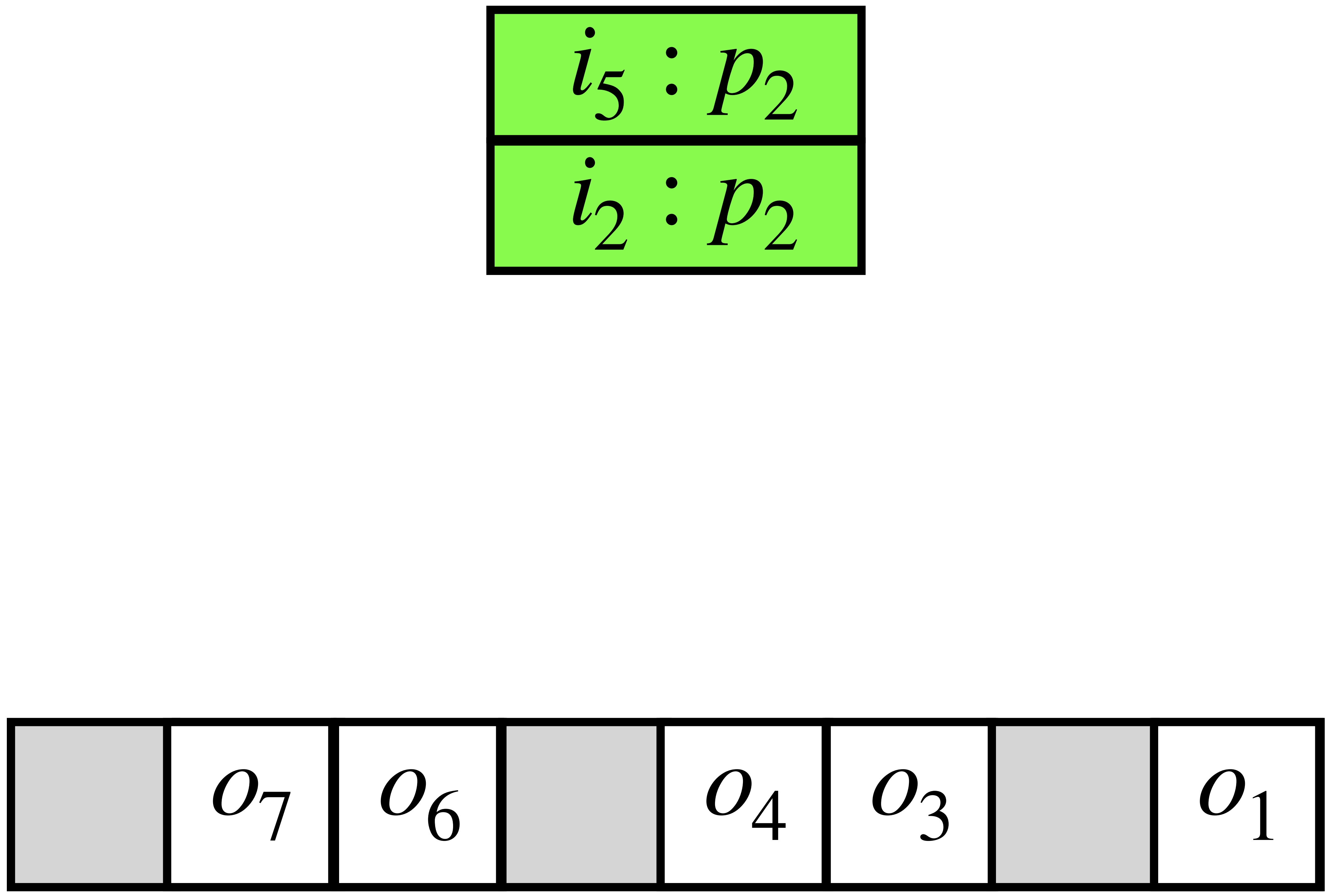
Partitioned Queues

- Consider each partition as a stateful operator with its own queue
- At most one worker can process a partition
- Most commonly used strategy in all stream processing engines
- Unnecessary blocking of outputs in the reordering buffer



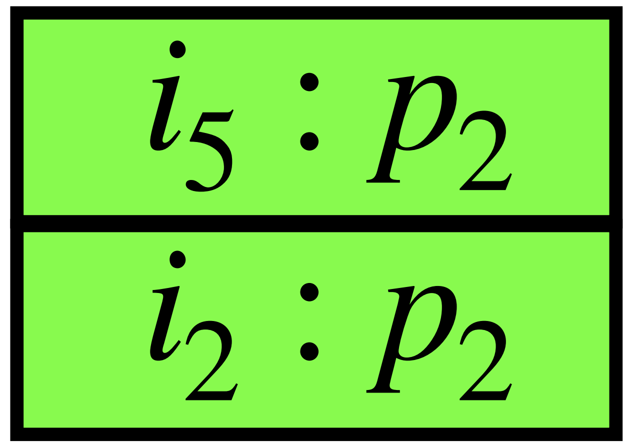
Partitioned Queues

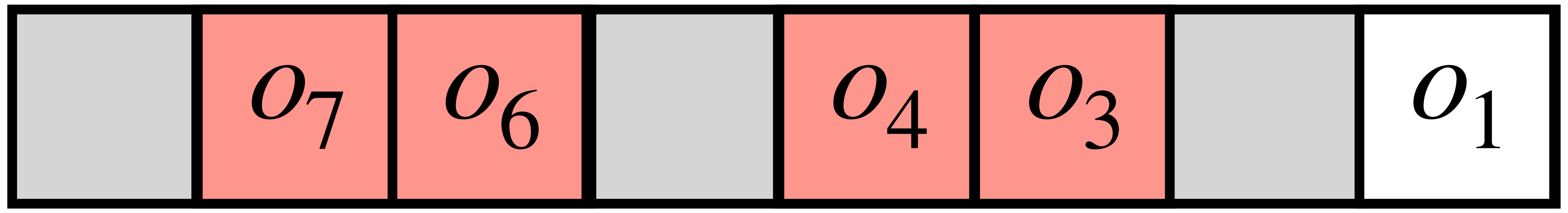
- Consider each partition as a stateful operator with its own queue
- At most one worker can process a partition
- Most commonly used strategy in all stream processing engines
- Unnecessary blocking of outputs in the reordering buffer



Partitioned Queues

- Consider each partition as a stateful operator with its own queue
- At most one worker can process a partition
- Most commonly used strategy in all stream processing engines
- Unnecessary blocking of outputs in the reordering buffer





Solutions

Shared Queue

- Almost ordered processing
- Partition guarantee violation

Partitioned Queues

- Partition guarantee
- Output blocking due to out-of-order processing

Our Solution: Hybrid Strategy

Hybrid Strategy

- Master Queue: Contains only partition ids in the order of arrival
- Partition Queues:
  - One for each partition
  - Each queue contains inputs belonging to a single partition
- Non-blocking strategy in the ordered setting!

Our Solution: Hybrid Queue

```
//invoked by producers
void addInput(tuple) {
 p = getPartition(tuple);
 partitionQueues[p].enqueue(msg);
 masterQueue.enqueue(p);
}
```

Master Queue

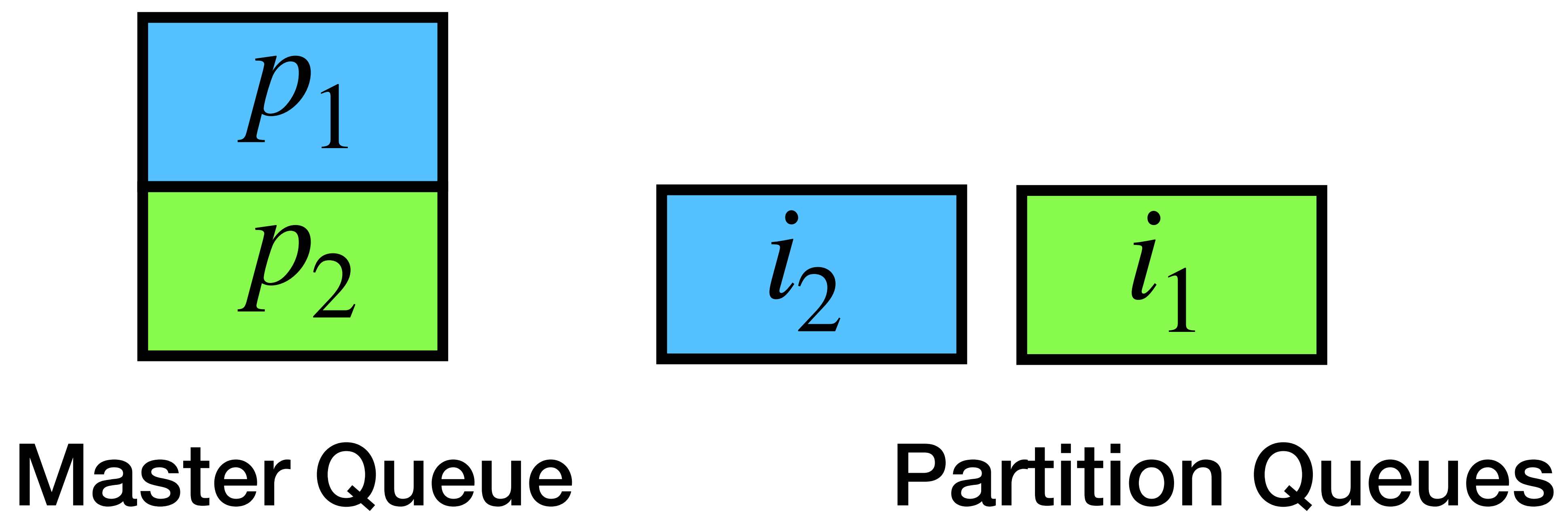
Partition Queues

Our Solution: Hybrid Queue

```
//invoked by producers
void addInput(tuple) {
 p = getPartition(tuple);
 partitionQueues[p].enqueue(msg);
 masterQueue.enqueue(p);
}
```

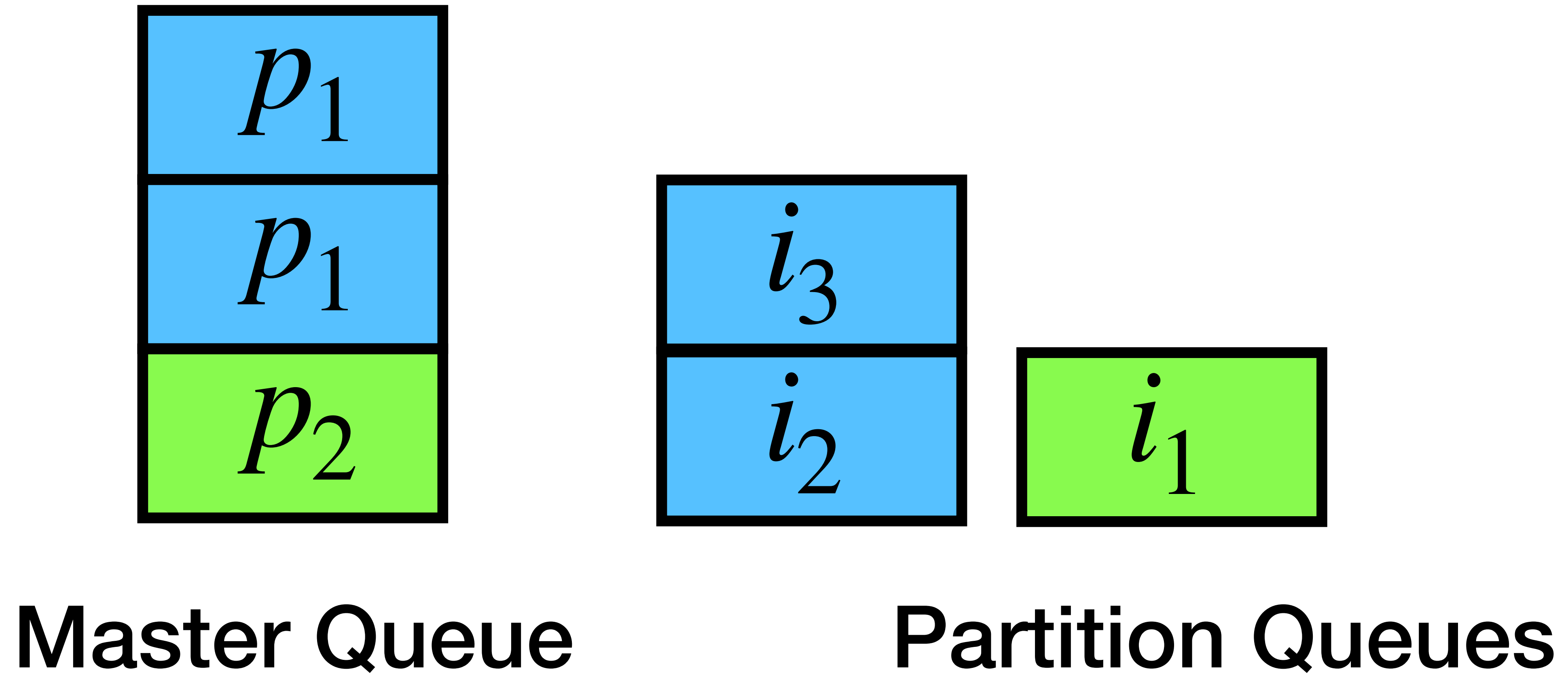
Our Solution: Hybrid Queue

```
//invoked by producers
void addInput(tuple) {
 p = getPartition(tuple);
 partitionQueues[p].enqueue(msg);
 masterQueue.enqueue(p);
}
```



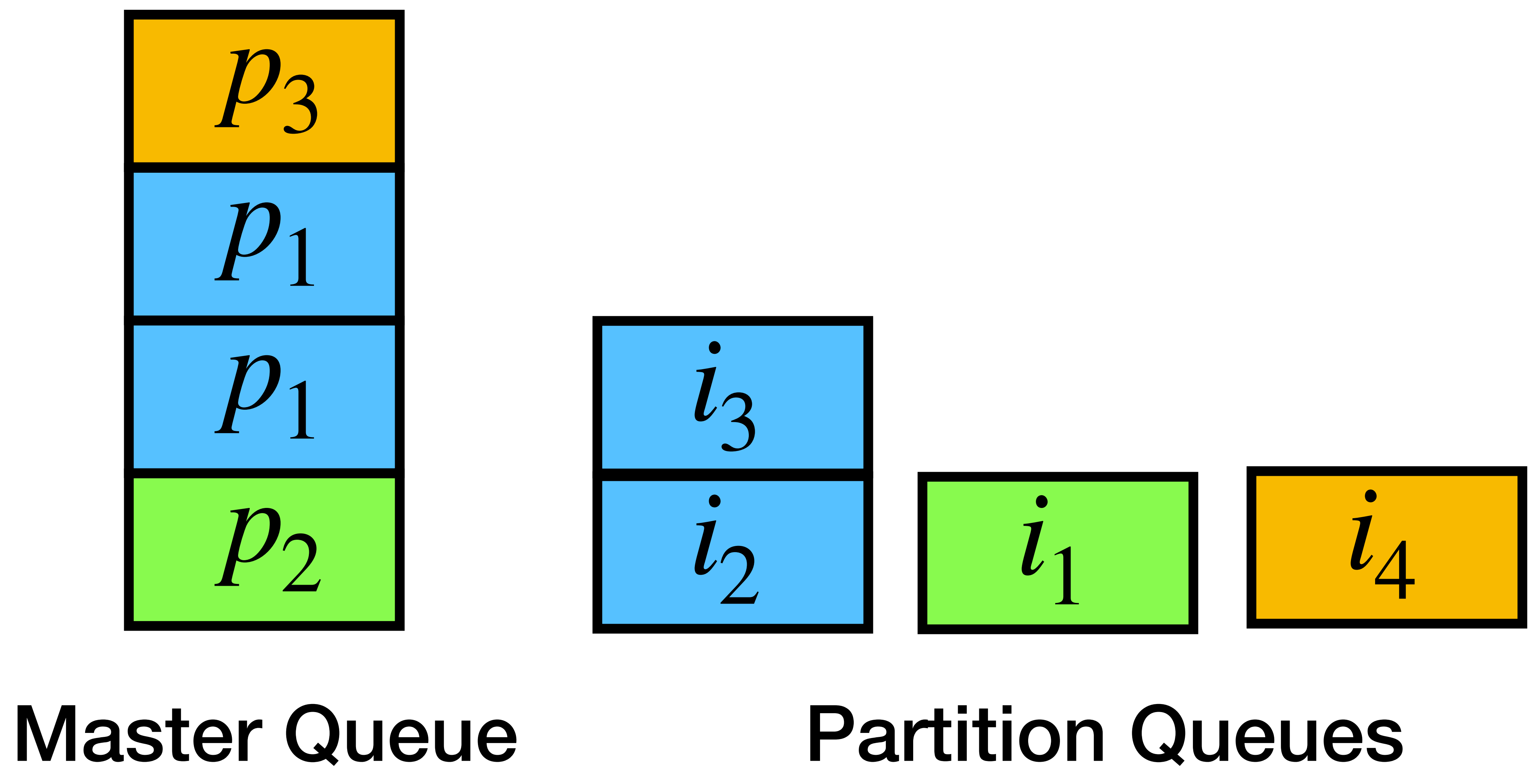
Our Solution: Hybrid Queue

```
//invoked by producers
void addInput(tuple) {
 p = getPartition(tuple);
 partitionQueues[p].enqueue(msg);
 masterQueue.enqueue(p);
}
```



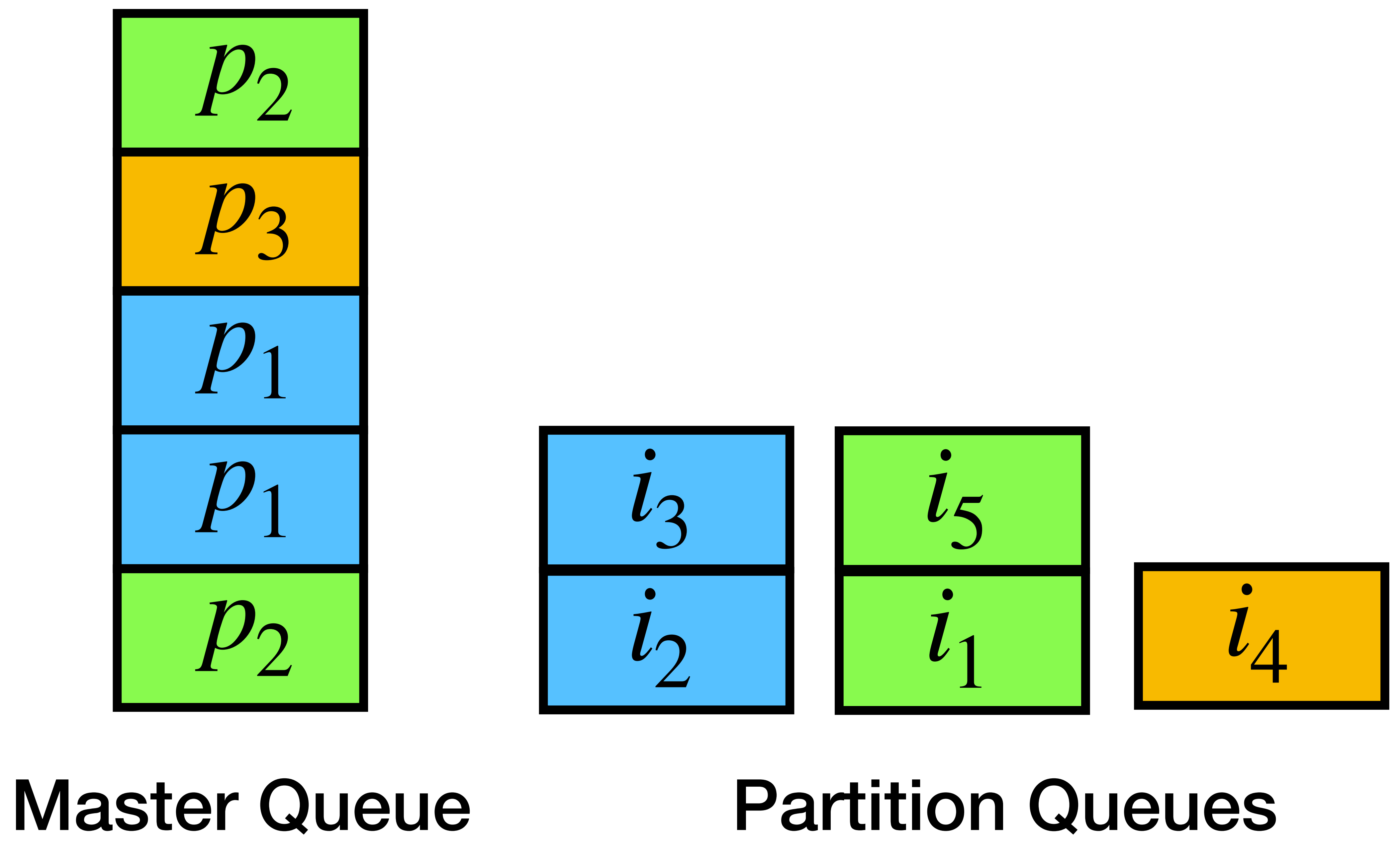
Our Solution: Hybrid Queue

```
//invoked by producers
void addInput(tuple) {
 p = getPartition(tuple);
 partitionQueues[p].enqueue(msg);
 masterQueue.enqueue(p);
}
```



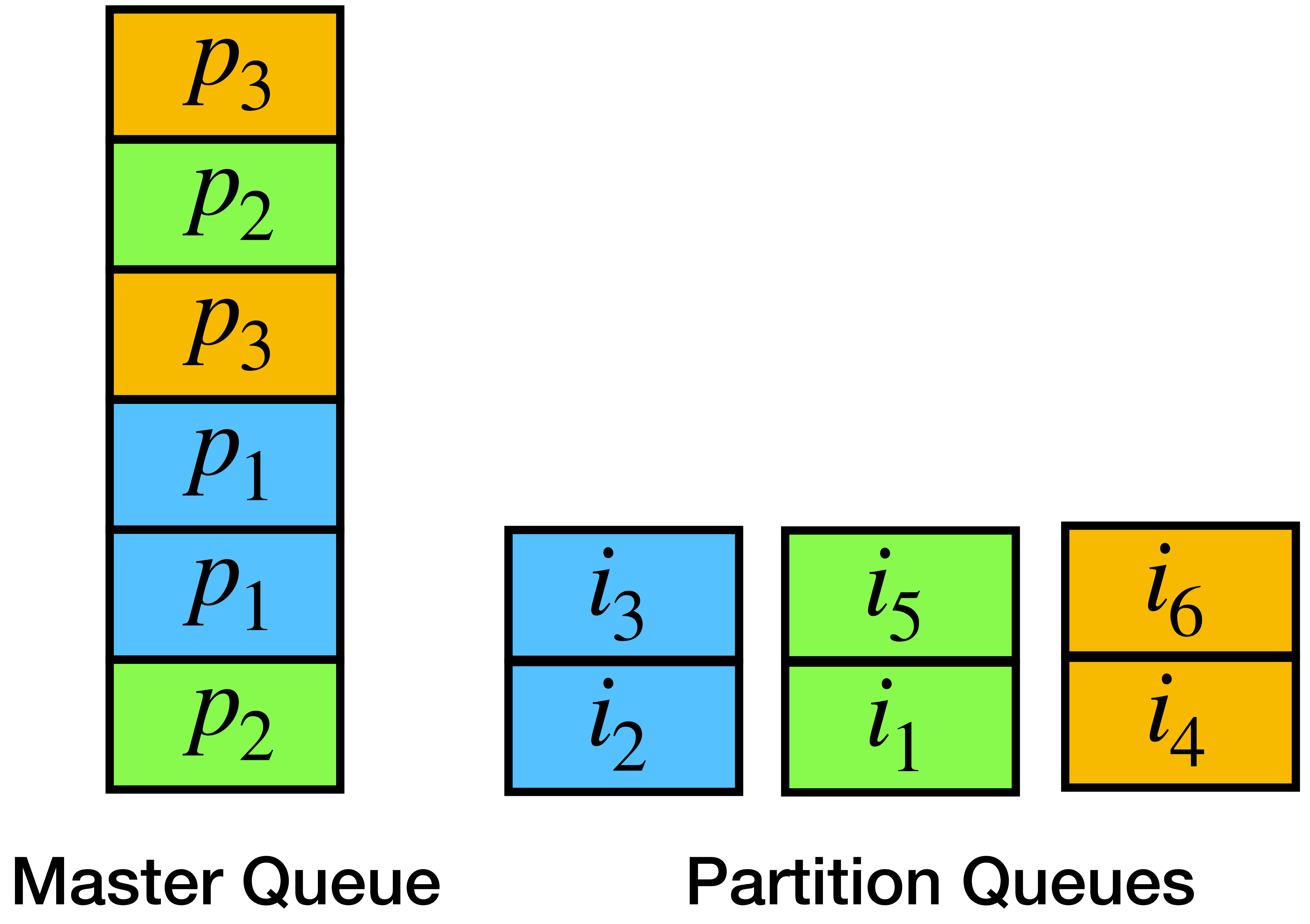
Our Solution: Hybrid Queue

```
//invoked by producers
void addInput(tuple) {
 p = getPartition(tuple);
 partitionQueues[p].enqueue(msg);
 masterQueue.enqueue(p);
}
```



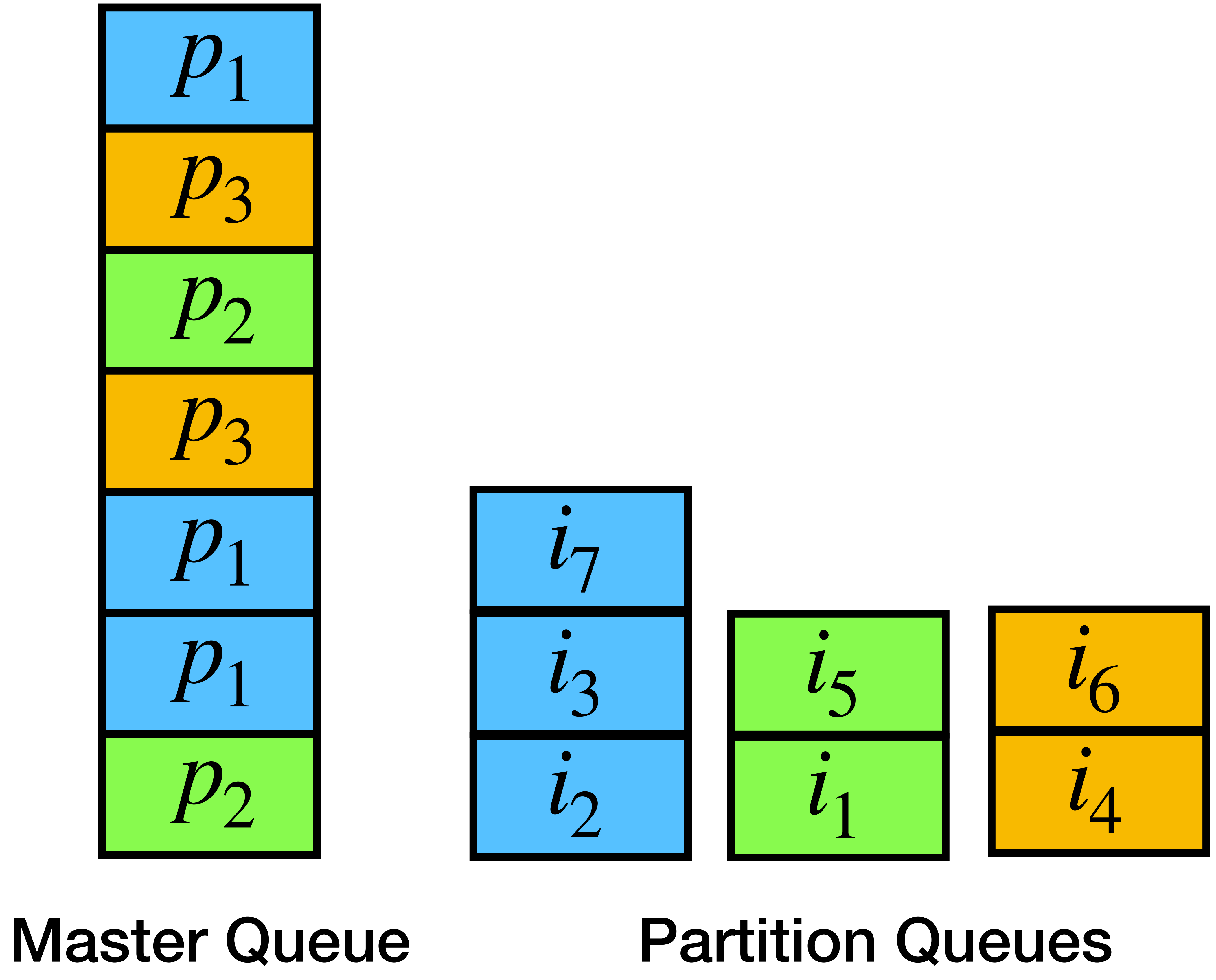
Our Solution: Hybrid Queue

```
//invoked by producers
void addInput(tuple) {
 p = getPartition(tuple);
 partitionQueues[p].enqueue(msg);
 masterQueue.enqueue(p);
}
```



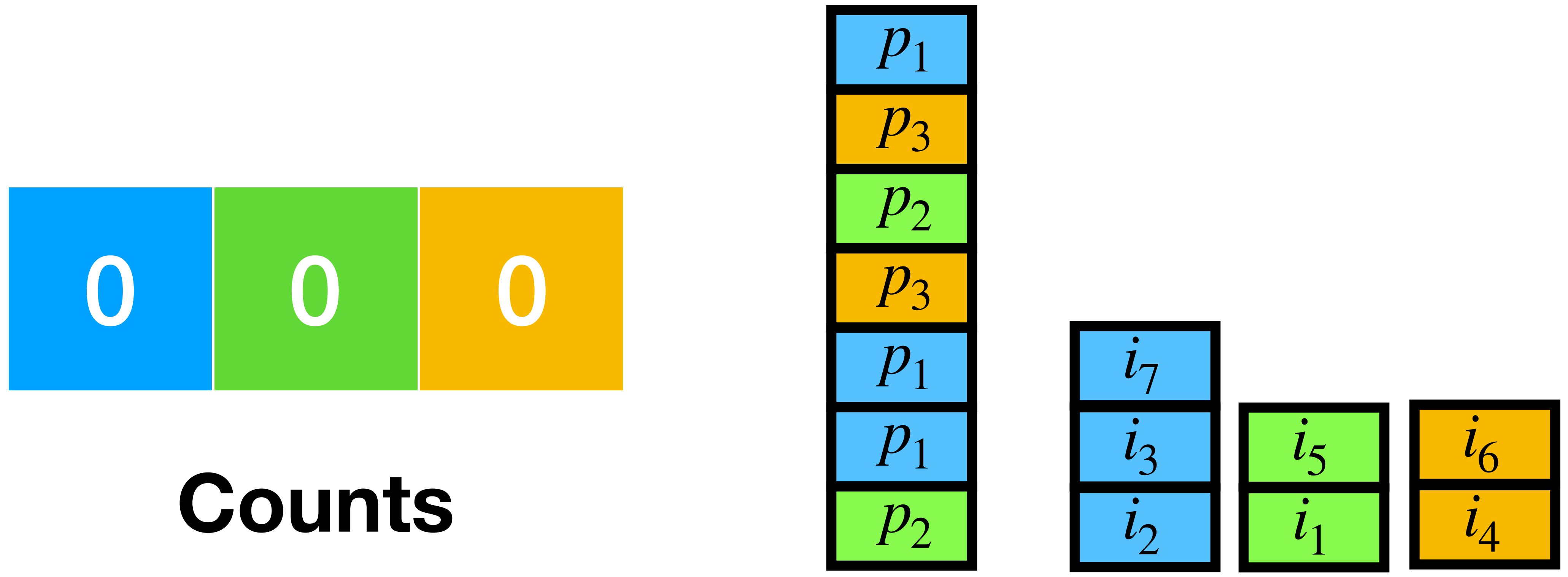
Our Solution: Hybrid Queue

```
//invoked by producers
void addInput(tuple) {
 p = getPartition(tuple);
 partitionQueues[p].enqueue(msg);
 masterQueue.enqueue(p);
}
```



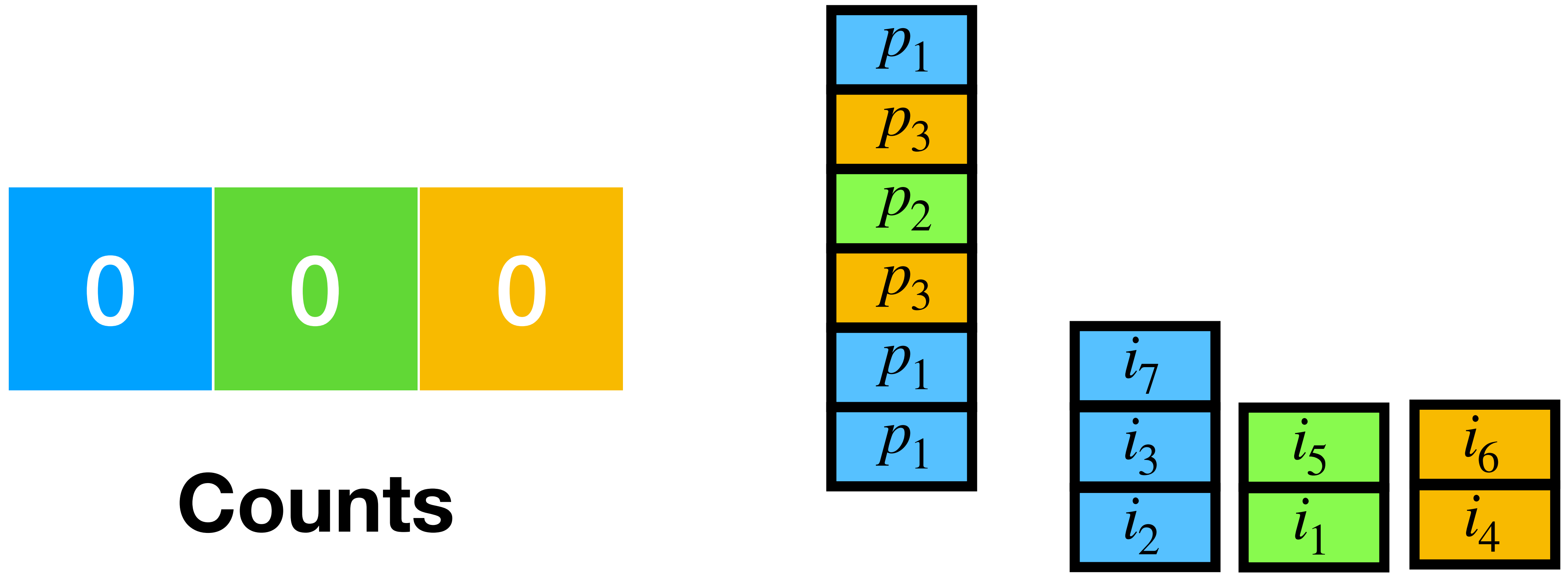
Our Solution: Hybrid Queue

```
//invoked by workers
void consumeInputs() {
 while(masterQueue.tryDequeue(p)) {
  if(count[p].fetch_add(1) == 0) {
   do {
    partitionQueues[p].tryDequeue(tuple);
    operate(tuple);
   } while(count[p].fetch_sub(1) > 1) ;
} } }
```



Our Solution: Hybrid Queue

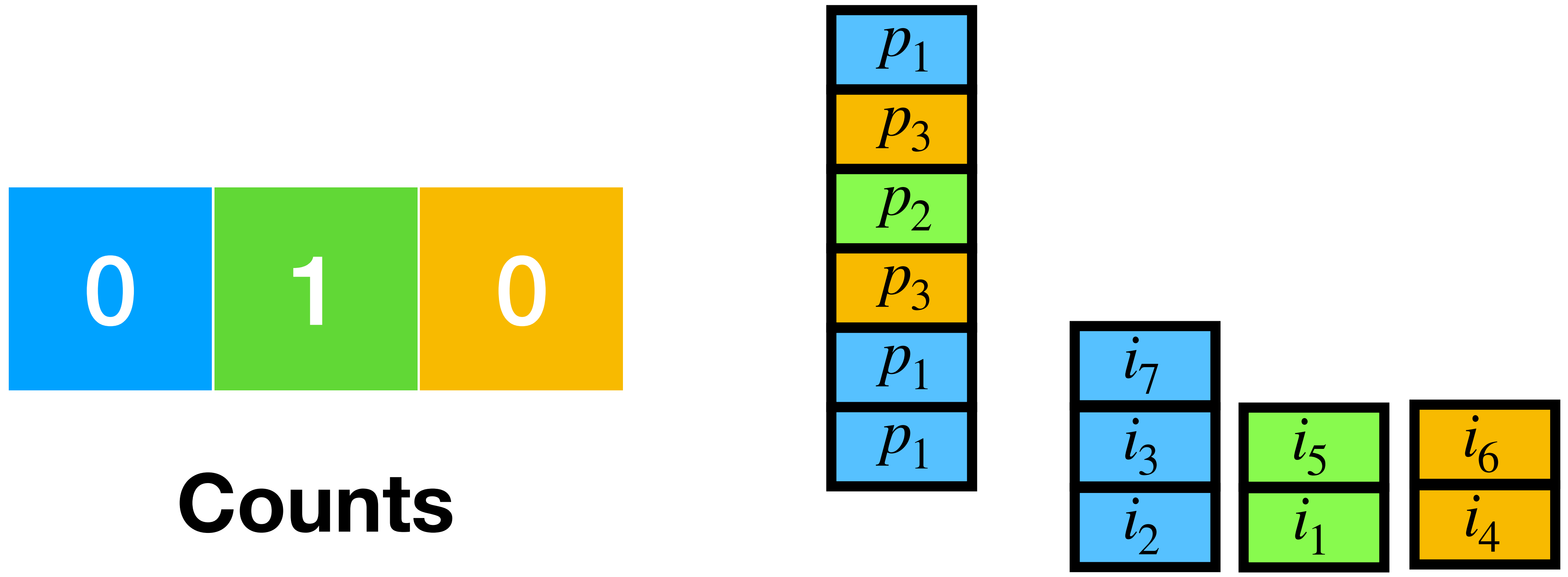
```
//invoked by workers
void consumeInputs() {
 while(masterQueue.tryDequeue(p)) {
  if(count[p].fetch_add(1) == 0) {
   do {
    partitionQueues[p].tryDequeue(tuple);
    operate(tuple);
   } while(count[p].fetch_sub(1) > 1) ;
} } }
```



Dequeue p_2

Our Solution: Hybrid Queue

```
//invoked by workers
void consumeInputs() {
 while(masterQueue.tryDequeue(p)) {
  if(count[p].fetch_add(1) == 0) {
   do {
    partitionQueues[p].tryDequeue(tuple);
    operate(tuple);
   } while(count[p].fetch_sub(1) > 1) ;
} } }
```

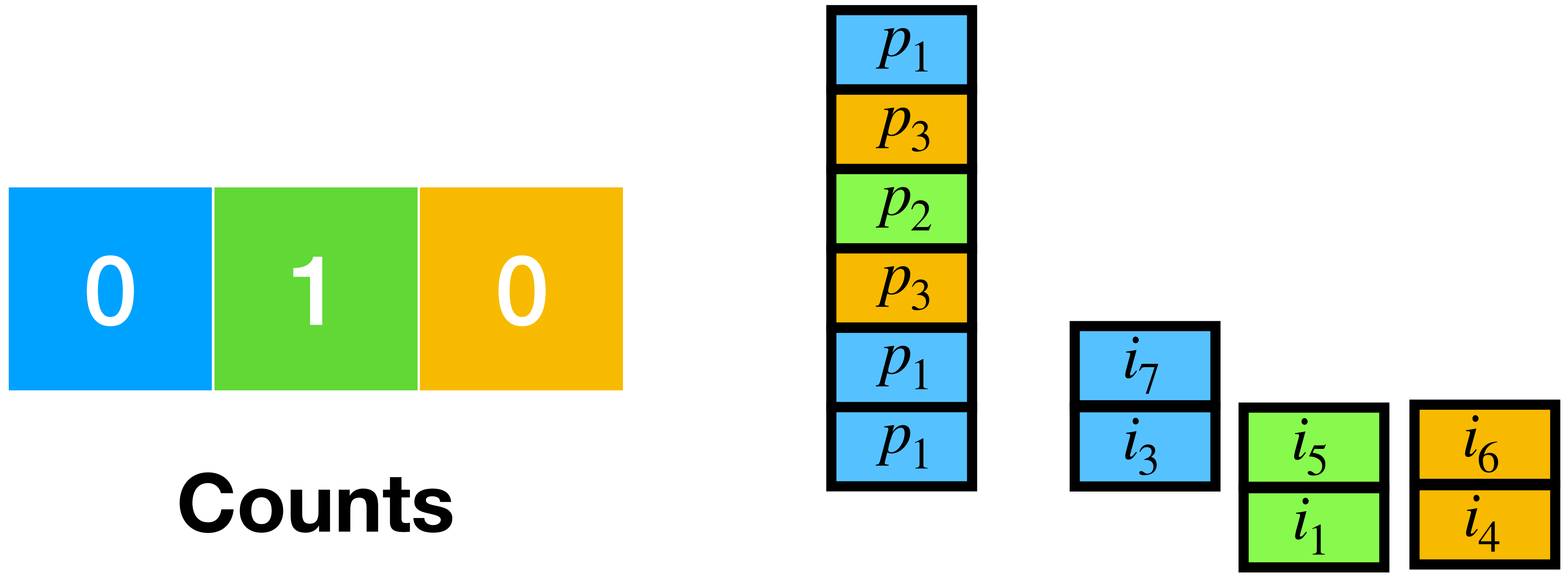


Dequeue p_2

counts$[p_2] : 0 \rightarrow 1$

Our Solution: Hybrid Queue

```
//invoked by workers
void consumeInputs() {
 while(masterQueue.tryDequeue(p)) {
  if(count[p].fetch_add(1) == 0) {
   do {
    partitionQueues[p].tryDequeue(tuple);
    operate(tuple);
   } while(count[p].fetch_sub(1) > 1) ;
} } }
```



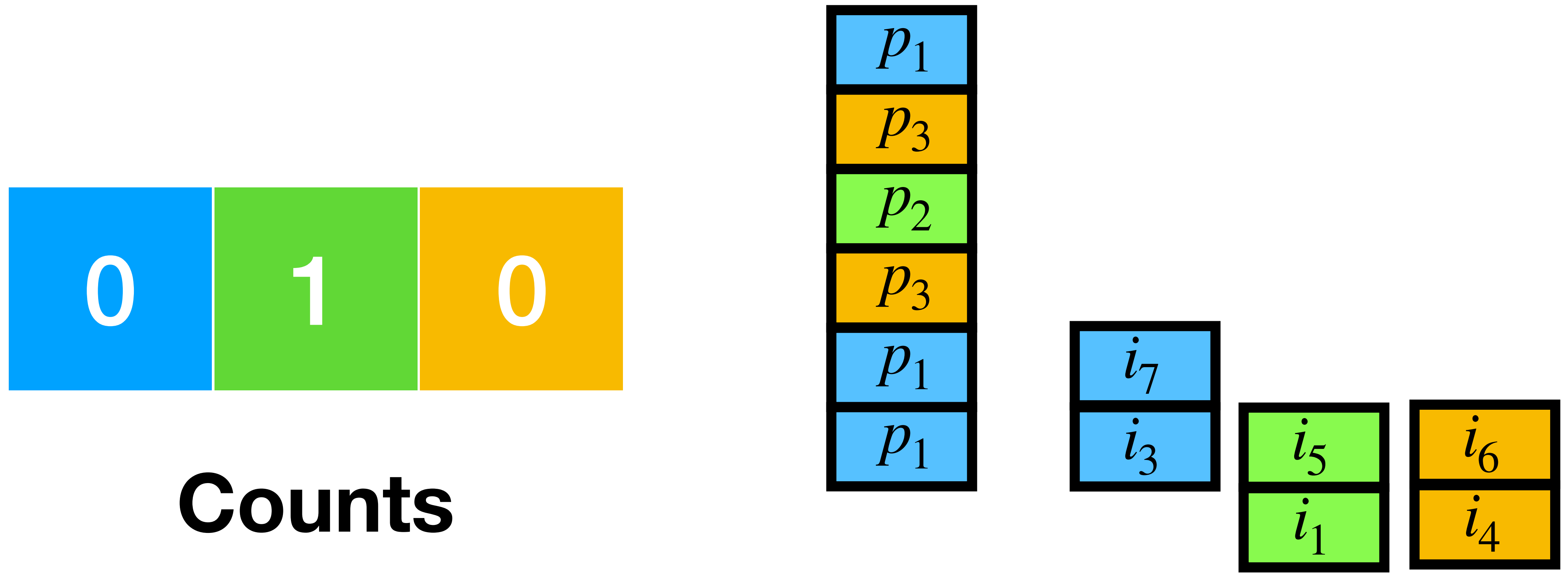
Dequeue p_2

counts$[p_2] : 0 \rightarrow 1$

Dequeue i_1

Our Solution: Hybrid Queue

```
//invoked by workers
void consumeInputs() {
 while(masterQueue.tryDequeue(p)) {
  if(count[p].fetch_add(1) == 0) {
   do {
    partitionQueues[p].tryDequeue(tuple);
    operate(tuple);
   } while(count[p].fetch_sub(1) > 1) ;
} } }
```



Dequeue p_2

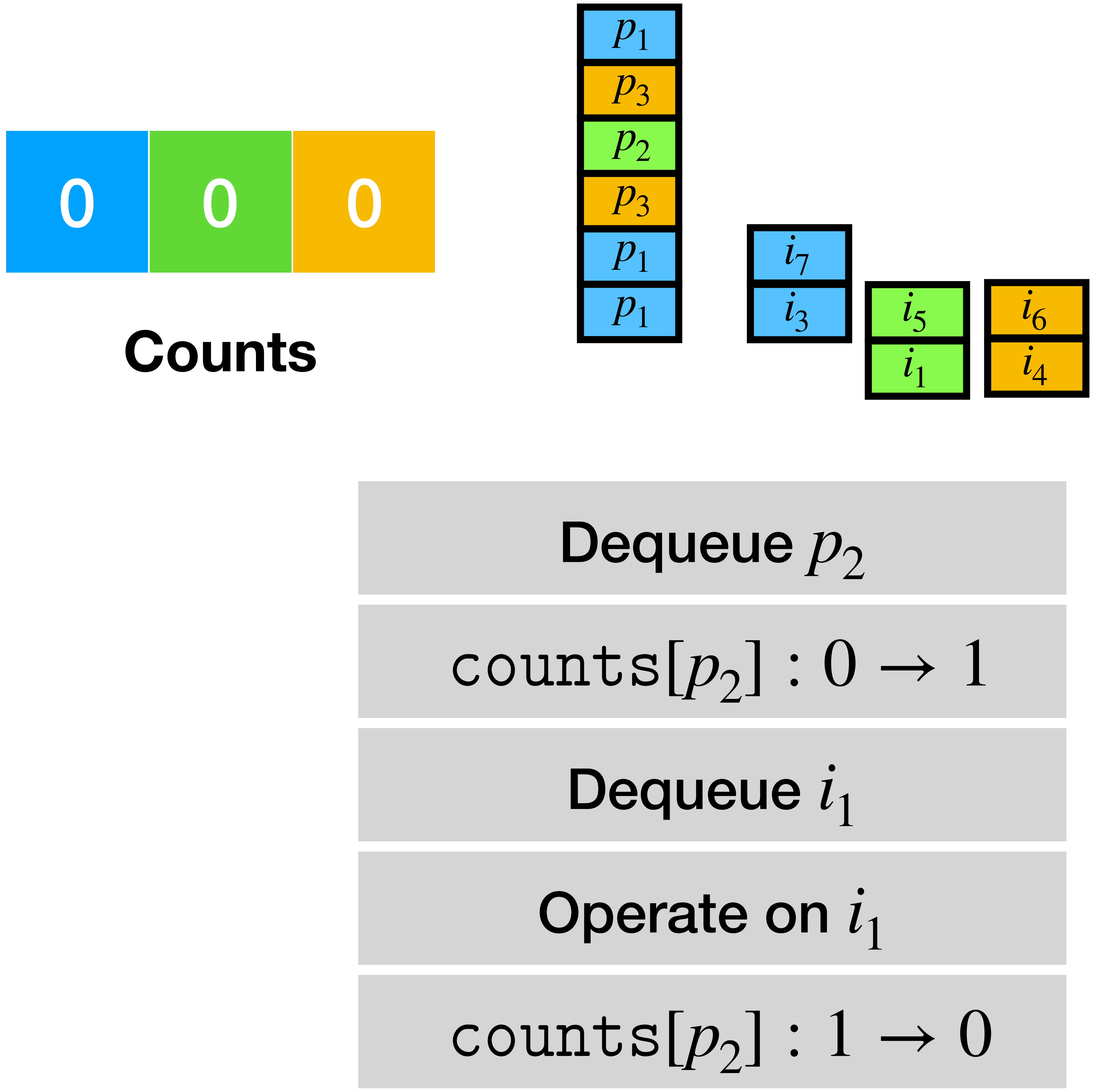
counts$[p_2] : 0 \to 1$

Dequeue i_1

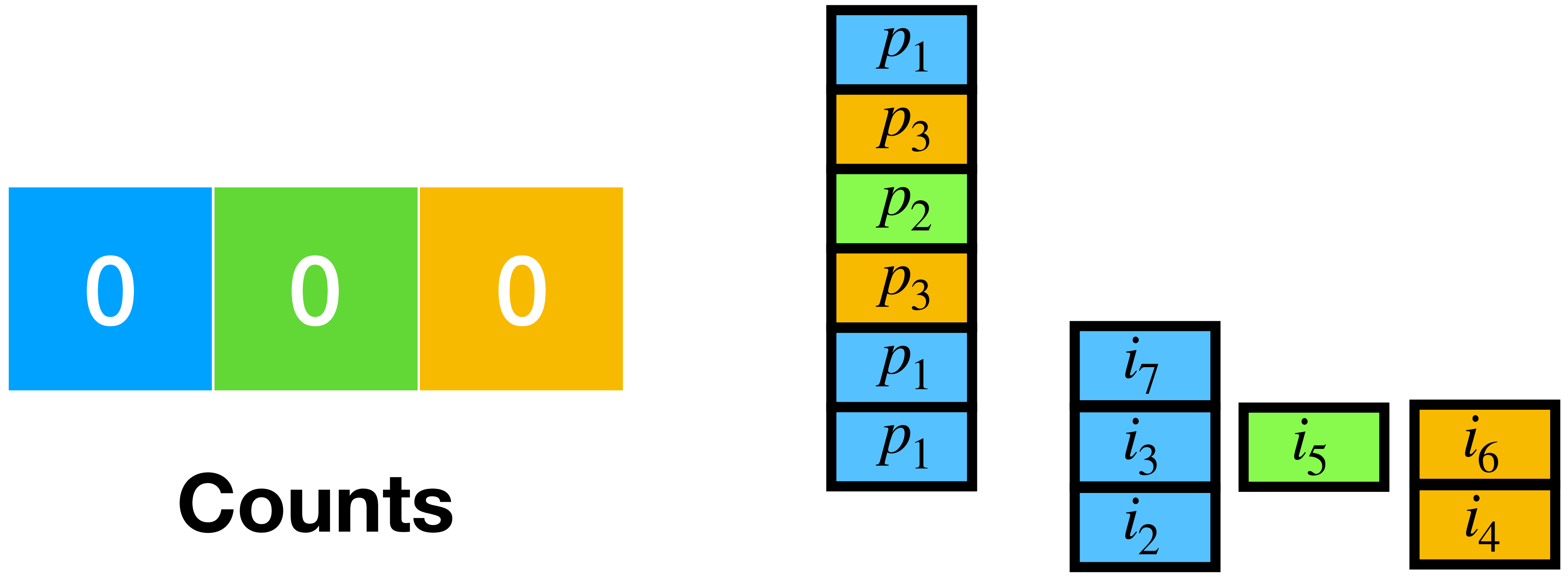
Operate on i_1

Our Solution: Hybrid Queue

```
//invoked by workers
void consumeInputs() {
 while(masterQueue.tryDequeue(p)) {
  if(count[p].fetch_add(1) == 0) {
   do {
    partitionQueues[p].tryDequeue(tuple);
    operate(tuple);
   } while(count[p].fetch_sub(1) > 1) ;
} } }
```



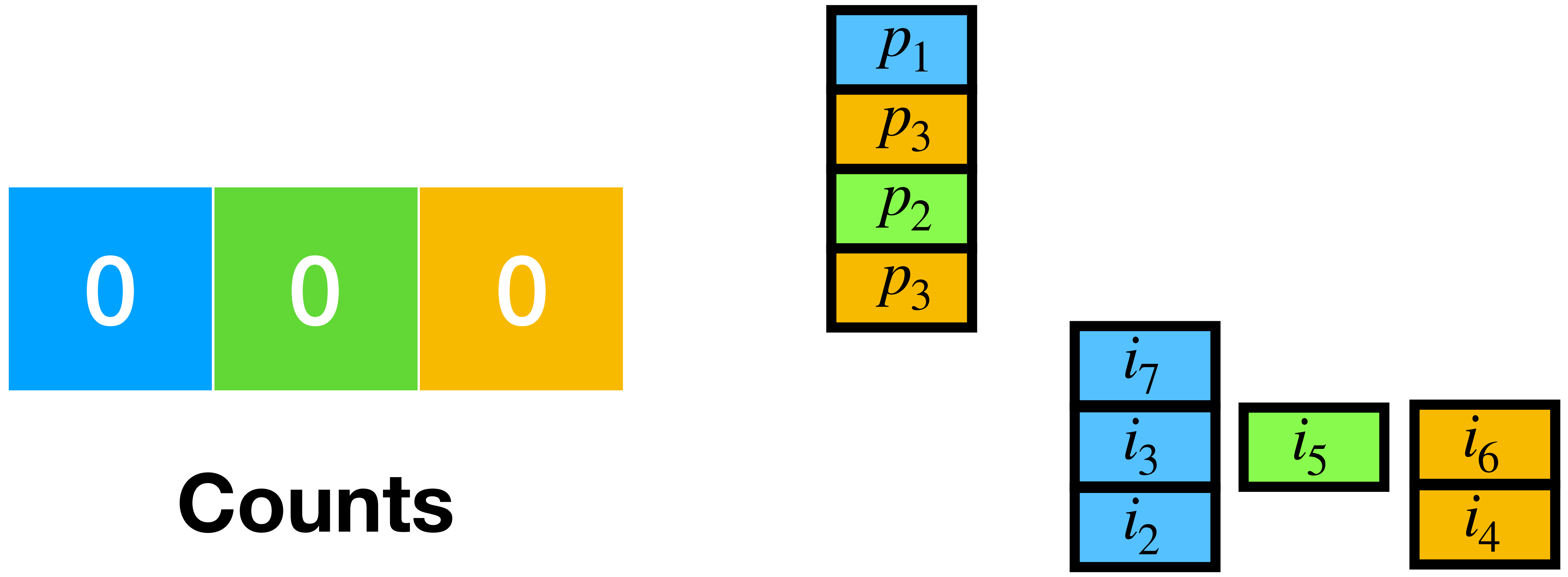
Our Solution: Hybrid Queue



Our Solution: Hybrid Queue

Dequeue p_1

Dequeue p_1



Our Solution: Hybrid Queue

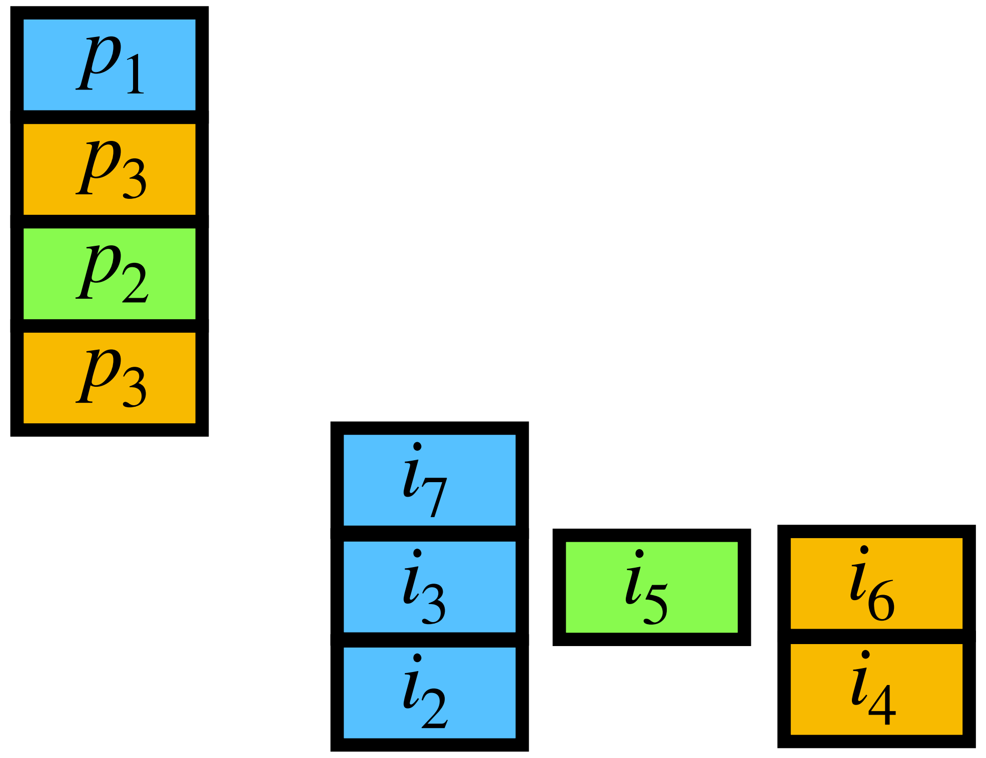
Dequeue p_1

$\texttt{counts}[p_1] : 0 \rightarrow 1$

Dequeue p_1

$\texttt{counts}[p_1] : 1 \rightarrow 2$

Counts



Our Solution: Hybrid Queue

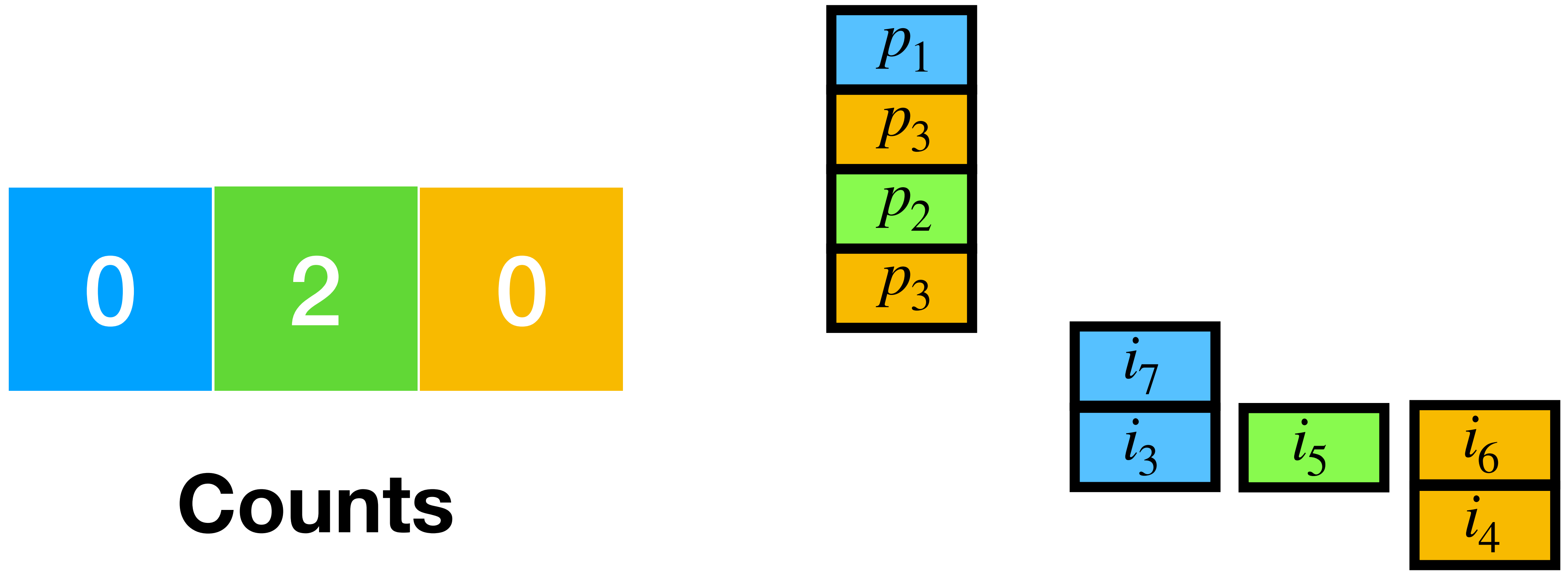
Dequeue p_1

$\texttt{counts}[p_1] : 0 \rightarrow 1$

Dequeue i_2

Dequeue p_1

$\texttt{counts}[p_1] : 1 \rightarrow 2$



Our Solution: Hybrid Queue

Dequeue p_1

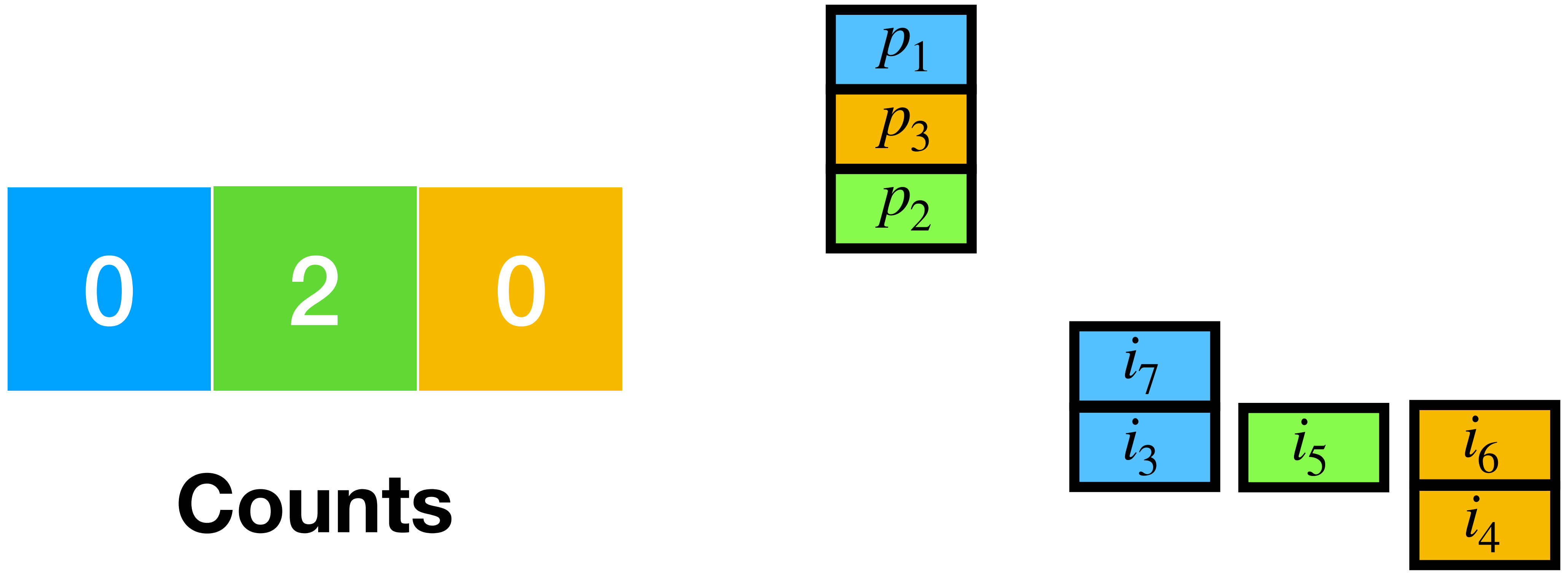
$\texttt{counts}[p_1] : 0 \rightarrow 1$

Dequeue i_2

Dequeue p_1

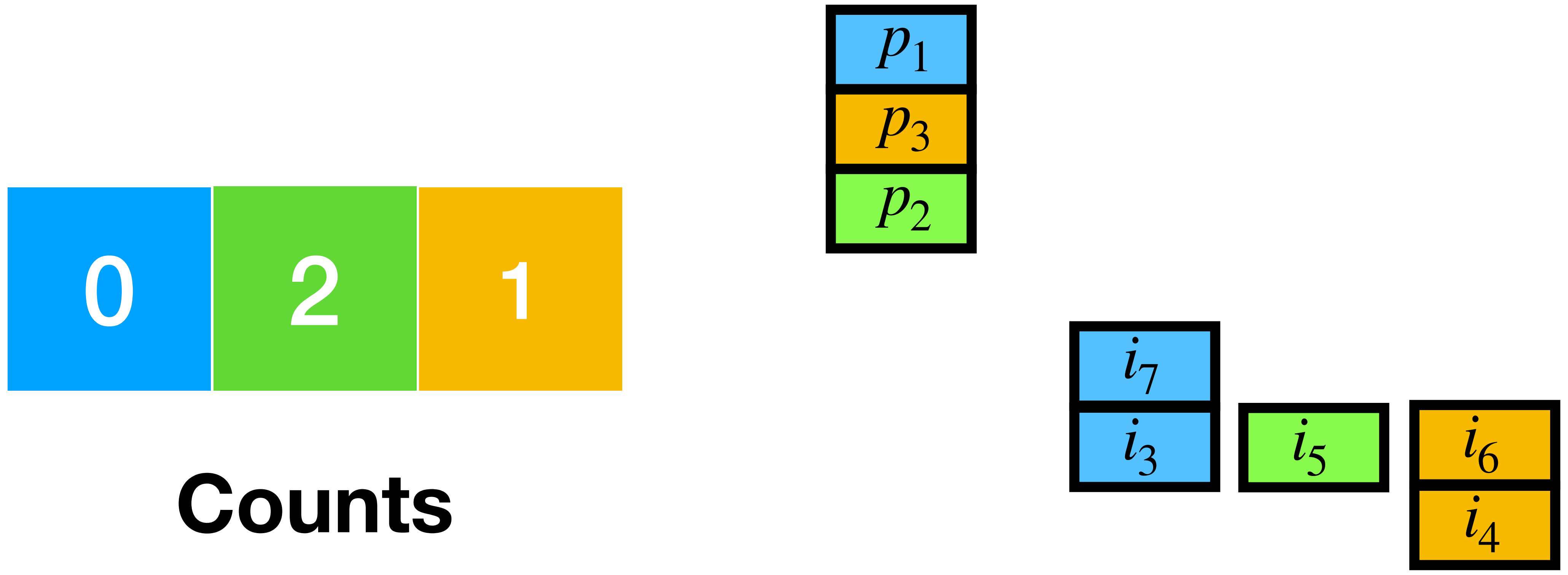
$\texttt{counts}[p_1] : 1 \rightarrow 2$

Dequeue p_3



Our Solution: Hybrid Queue

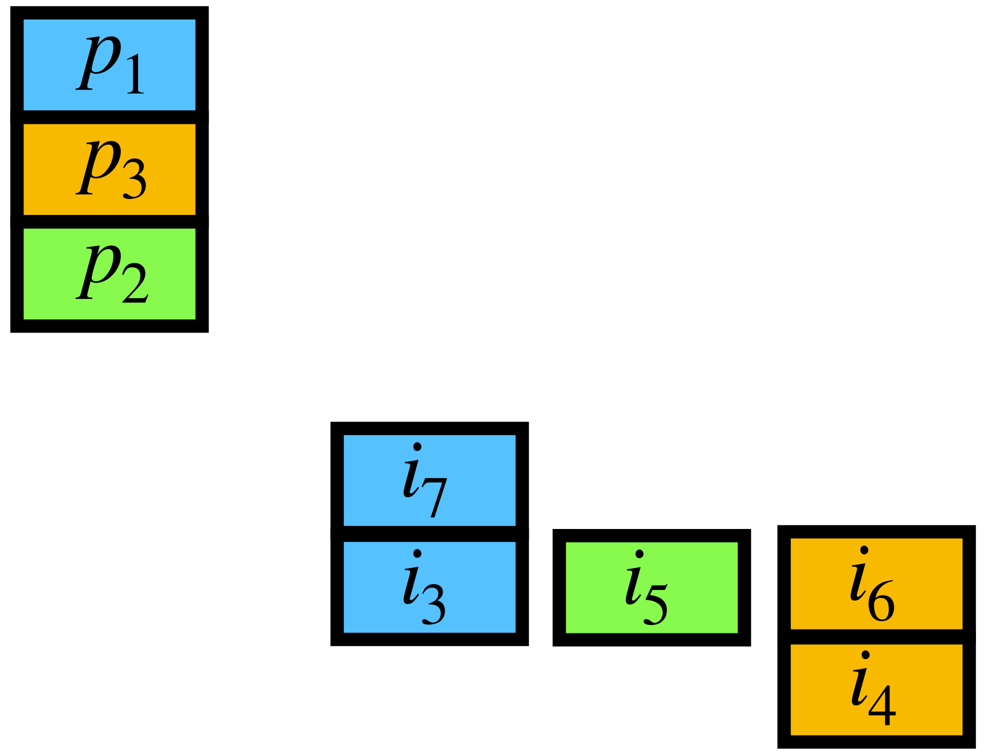
Dequeue p_1	Dequeue p_1
$\texttt{counts}[p_1] : 0 \to 1$	$\texttt{counts}[p_1] : 1 \to 2$
Dequeue i_2	Dequeue p_3
Operate on i_2	$\texttt{counts}[p_3] : 0 \to 1$



Our Solution: Hybrid Queue

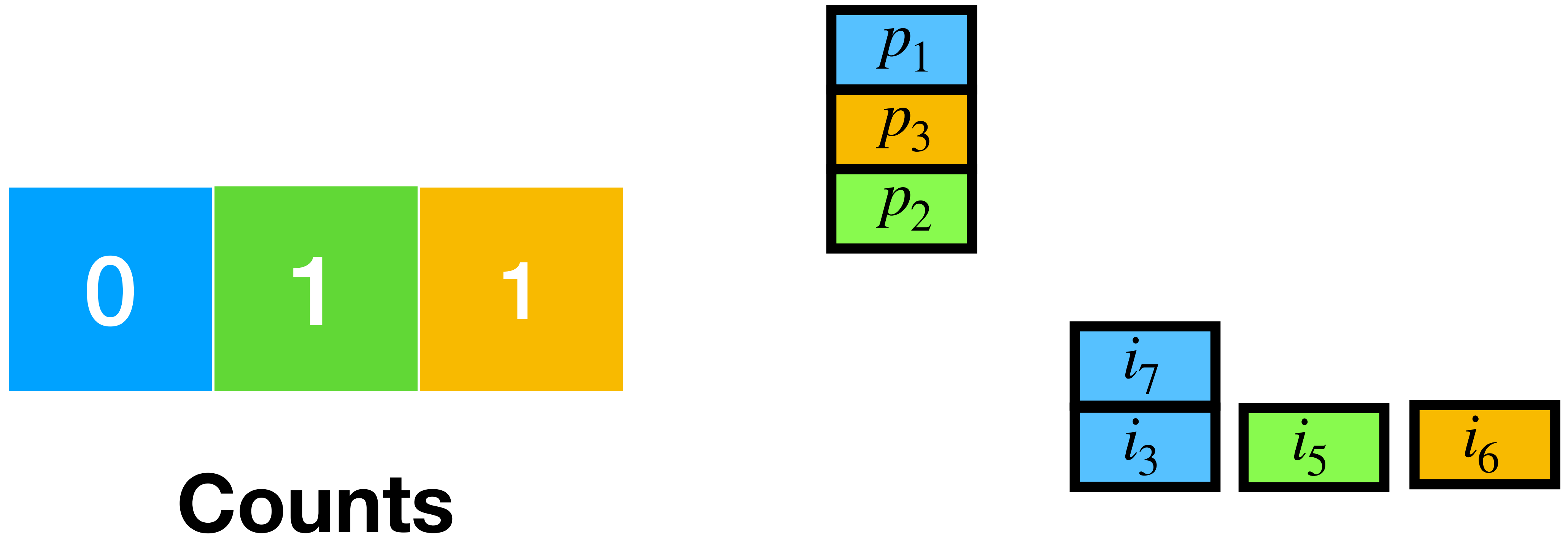
Dequeue p_1	**Dequeue** p_1
$\texttt{counts}[p_1] : 0 \rightarrow 1$	$\texttt{counts}[p_1] : 1 \rightarrow 2$
Dequeue i_2	**Dequeue** p_3
Operate on i_2	$\texttt{counts}[p_3] : 0 \rightarrow 1$
$\texttt{counts}[p_1] : 2 \rightarrow 1$	

Counts



Our Solution: Hybrid Queue

Dequeue p_1	Dequeue p_1
$\texttt{counts}[p_1] : 0 \rightarrow 1$	$\texttt{counts}[p_1] : 1 \rightarrow 2$
Dequeue i_2	Dequeue p_3
Operate on i_2	$\texttt{counts}[p_3] : 0 \rightarrow 1$
$\texttt{counts}[p_1] : 2 \rightarrow 1$	Dequeue i_4



Our Solution: Hybrid Queue

Dequeue p_1

$\texttt{counts}[p_1] : 0 \to 1$

Dequeue i_2

Operate on i_2

$\texttt{counts}[p_1] : 2 \to 1$

Dequeue i_3

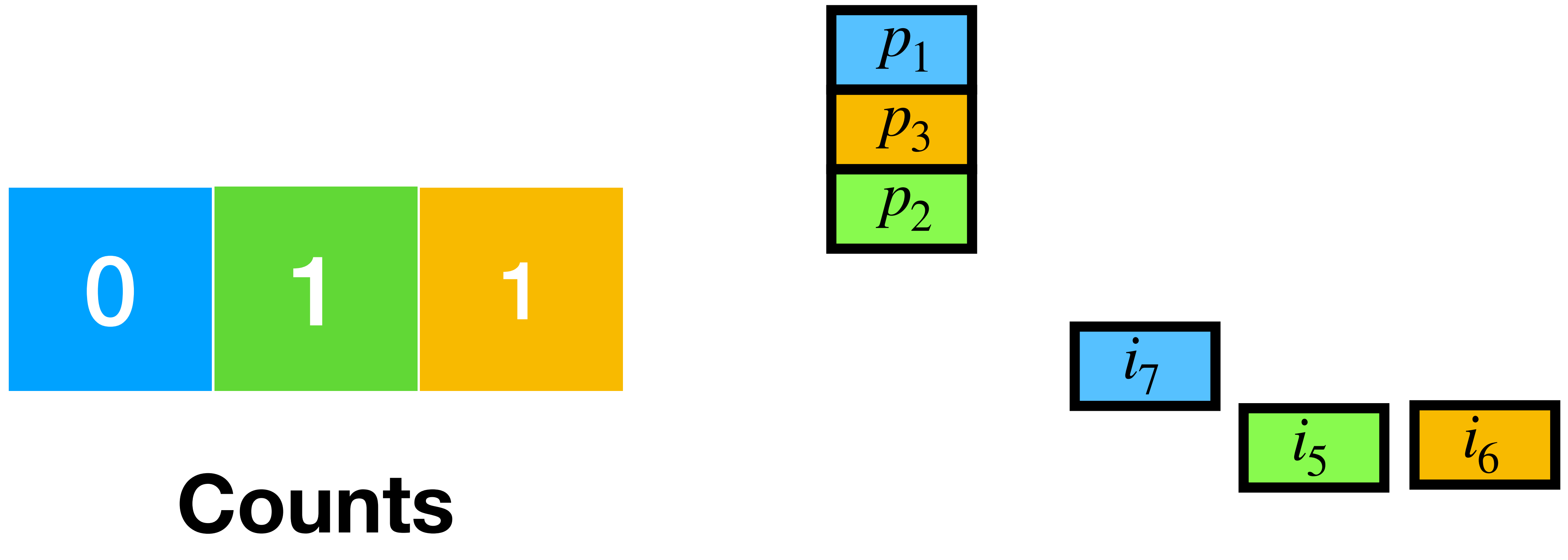
Dequeue p_1

$\texttt{counts}[p_1] : 1 \to 2$

Dequeue p_3

$\texttt{counts}[p_3] : 0 \to 1$

Dequeue i_4



Our Solution: Hybrid Queue

Dequeue p_1

$\texttt{counts}[p_1] : 0 \rightarrow 1$

Dequeue i_2

Operate on i_2

$\texttt{counts}[p_1] : 2 \rightarrow 1$

Dequeue i_3

Dequeue p_1

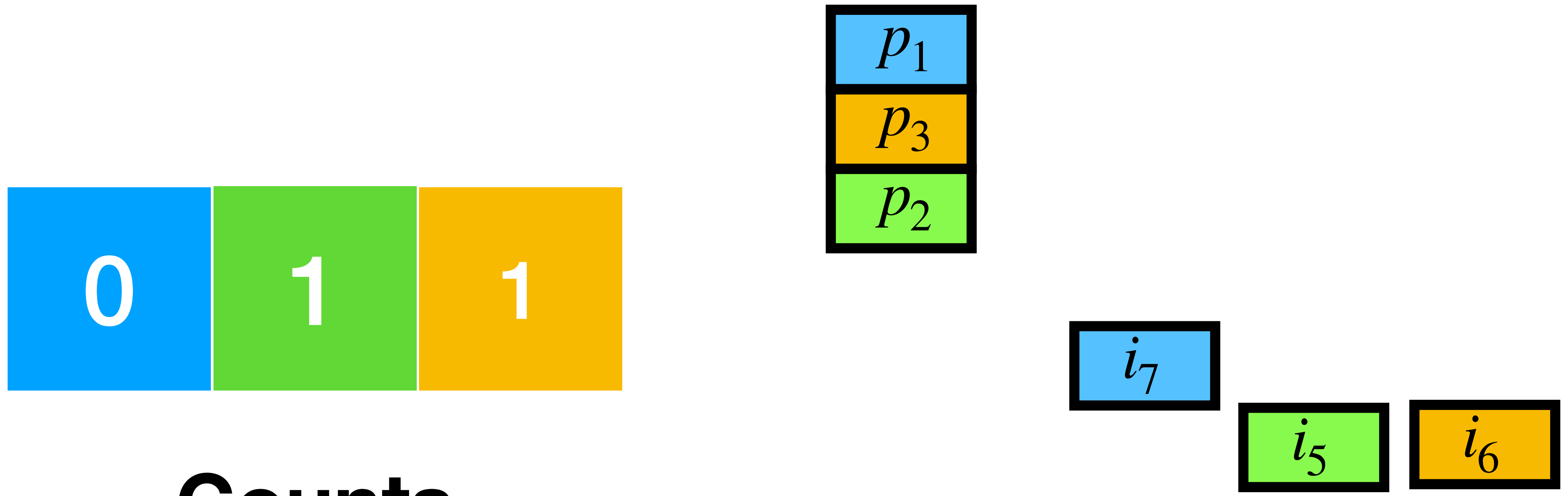
$\texttt{counts}[p_1] : 1 \rightarrow 2$

Dequeue p_3

$\texttt{counts}[p_3] : 0 \rightarrow 1$

Dequeue i_4

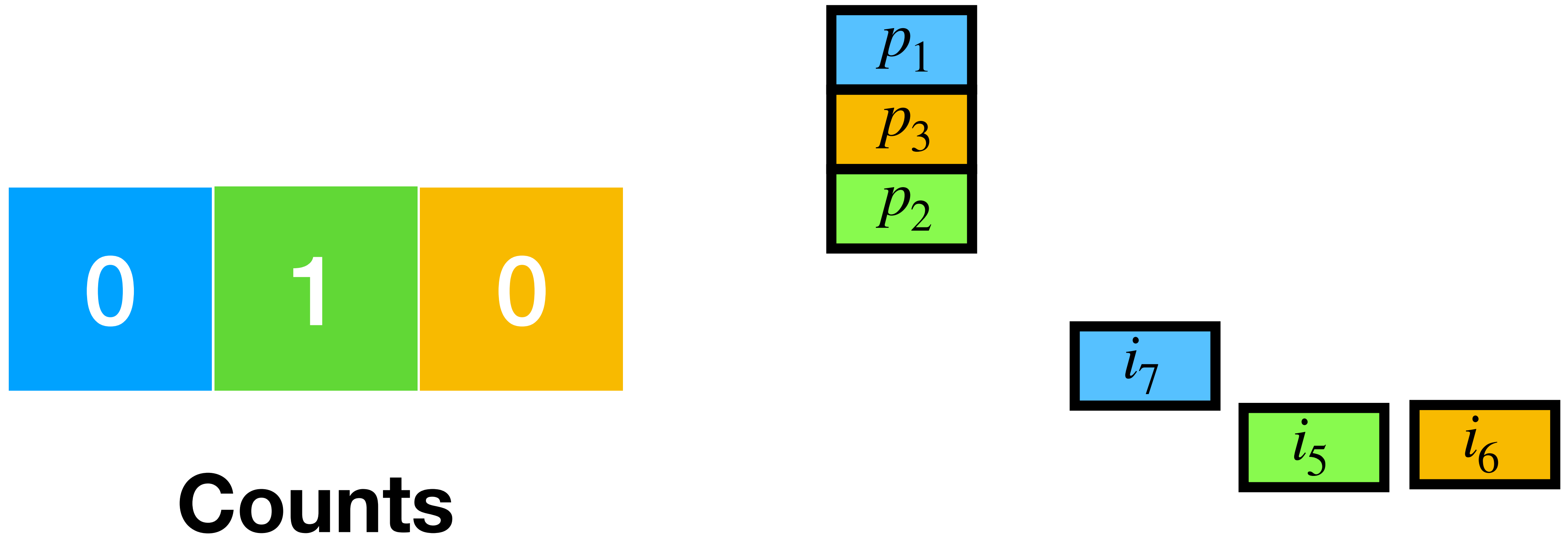
Operate on i_4



Counts

Our Solution: Hybrid Queue

Dequeue p_1	Dequeue p_1
$\texttt{counts}[p_1] : 0 \to 1$	$\texttt{counts}[p_1] : 1 \to 2$
Dequeue i_2	Dequeue p_3
Operate on i_2	$\texttt{counts}[p_3] : 0 \to 1$
$\texttt{counts}[p_1] : 2 \to 1$	Dequeue i_4
Dequeue i_3	Operate on i_4
Operate on i_3	$\texttt{counts}[p_3] : 1 \to 0$



Our Solution: Hybrid Queue

Dequeue p_1

$\texttt{counts}[p_1] : 0 \to 1$

Dequeue i_2

Operate on i_2

$\texttt{counts}[p_1] : 2 \to 1$

Dequeue i_3

Operate on i_3

$\texttt{counts}[p_1] : 1 \to 0$

Dequeue p_1

$\texttt{counts}[p_1] : 1 \to 2$

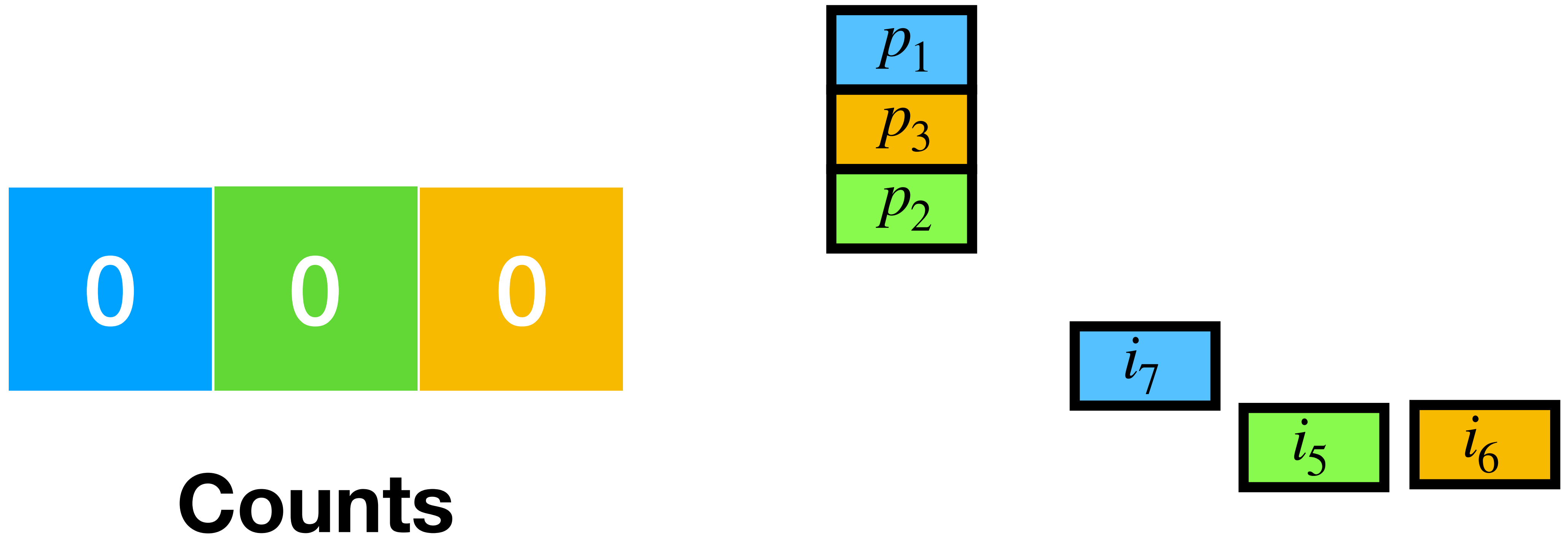
Dequeue p_3

$\texttt{counts}[p_3] : 0 \to 1$

Dequeue i_4

Operate on i_4

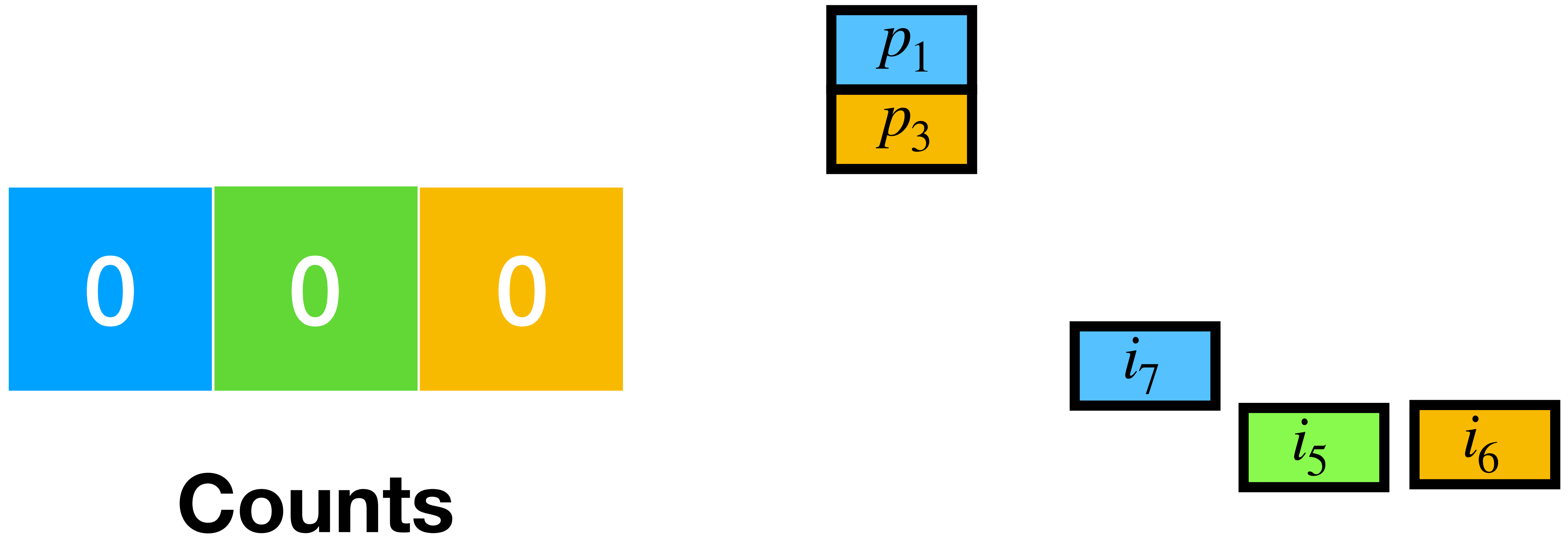
$\texttt{counts}[p_3] : 1 \to 0$



Our Solution: Hybrid Queue

- **Dequeue** p_1
- $\texttt{counts}[p_1] : 0 \rightarrow 1$
- **Dequeue** i_2
- **Operate on** i_2
- $\texttt{counts}[p_1] : 2 \rightarrow 1$
- **Dequeue** i_3
- **Operate on** i_3
- $\texttt{counts}[p_1] : 1 \rightarrow 0$

- **Dequeue** p_1
- $\texttt{counts}[p_1] : 1 \rightarrow 2$
- **Dequeue** p_3
- $\texttt{counts}[p_3] : 0 \rightarrow 1$
- **Dequeue** i_4
- **Operate on** i_4
- $\texttt{counts}[p_3] : 1 \rightarrow 0$
- **Dequeue** p_2



Our Solution: Hybrid Queue

Dequeue p_1

$\texttt{counts}[p_1] : 0 \rightarrow 1$

Dequeue i_2

Operate on i_2

$\texttt{counts}[p_1] : 2 \rightarrow 1$

Dequeue i_3

Operate on i_3

$\texttt{counts}[p_1] : 1 \rightarrow 0$

Dequeue p_3

Dequeue p_1

$\texttt{counts}[p_1] : 1 \rightarrow 2$

Dequeue p_3

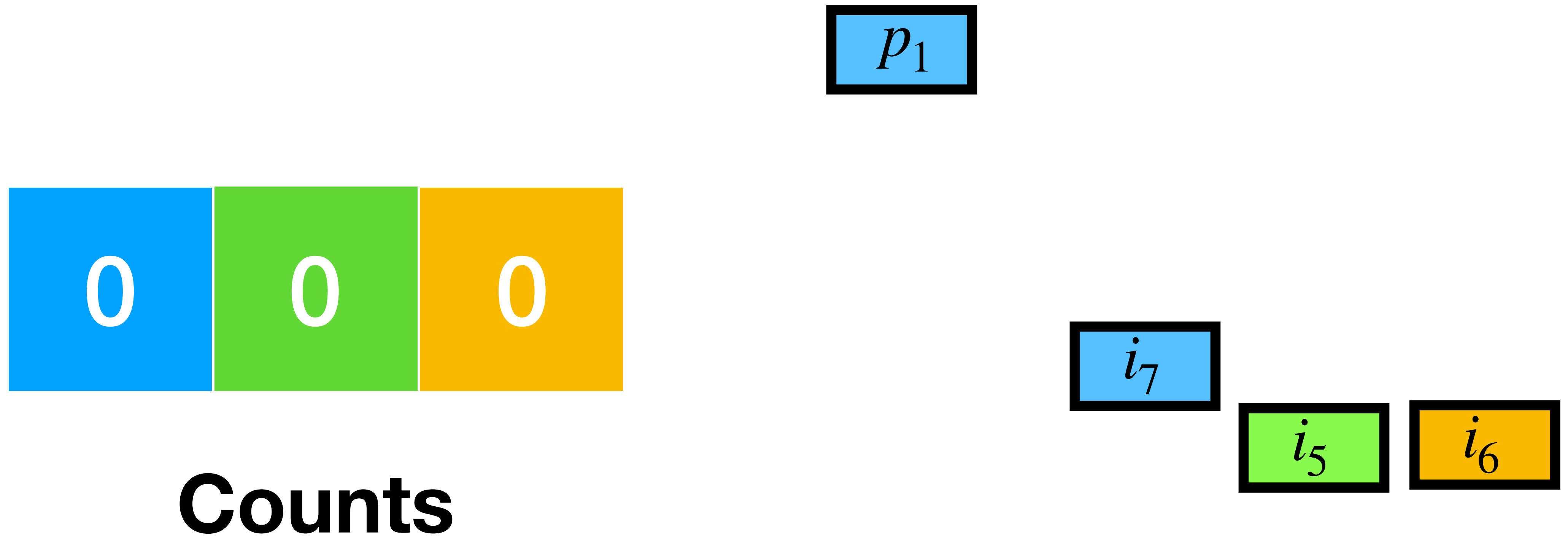
$\texttt{counts}[p_3] : 0 \rightarrow 1$

Dequeue i_4

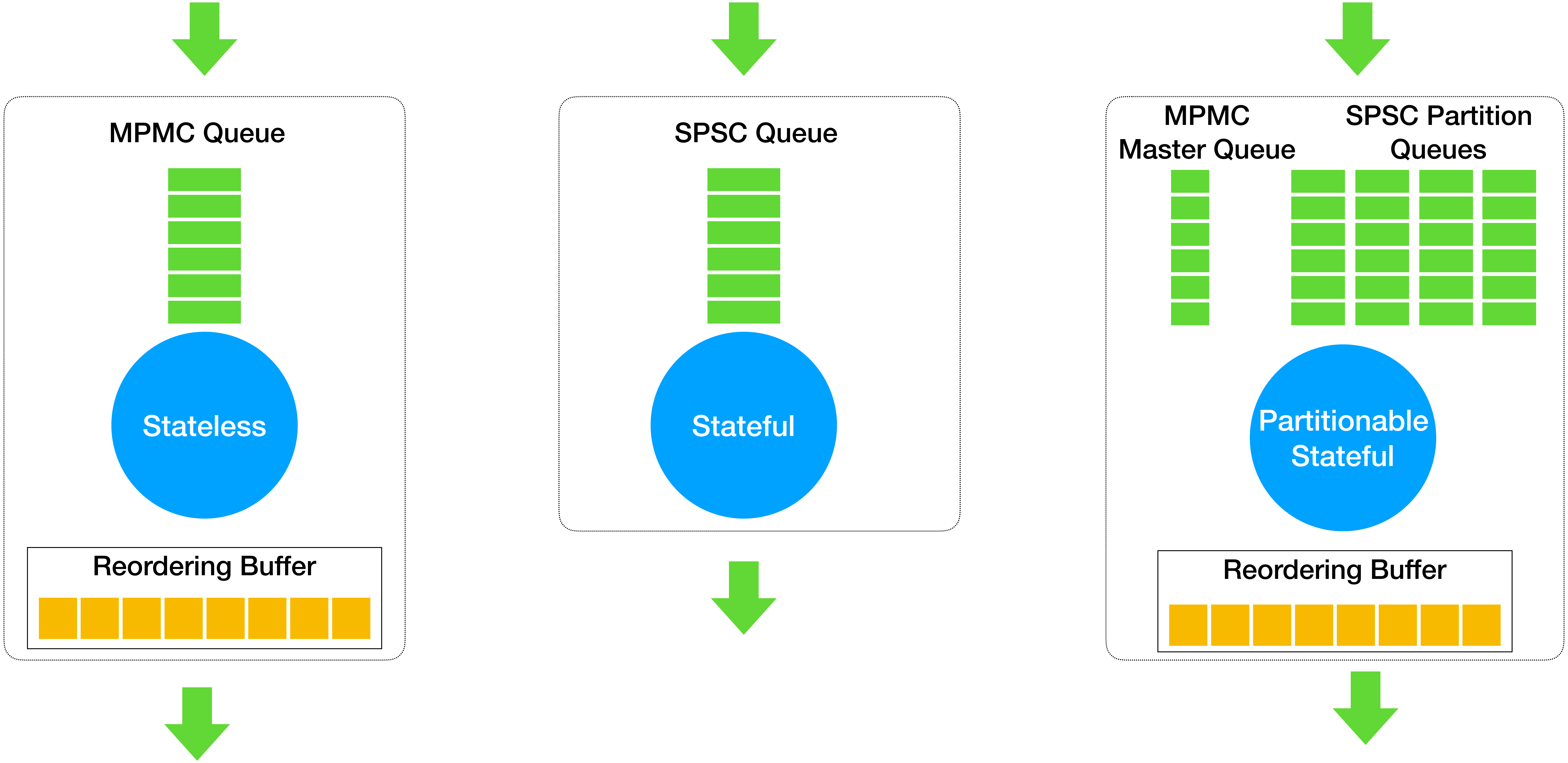
Operate on i_4

$\texttt{counts}[p_3] : 1 \rightarrow 0$

Dequeue p_2



Operator Implementations



Scheduling Runtime

Dynamic Scheduling

Monitor the state of the pipeline and
operator characteristics to answer

Which operator should a worker next work on?

Parameters of Interest

I_i	Input queue size
O_i	Output queue size
s_i	Average selectivity i.e. Number of outputs per input
c_i	Operator cost i.e. Time taken to process each input
w_i	Number of workers allotted currently
M_i	Maximum allowed number of workers

4 Heuristics

Queue-Size Throttling (QST)	Estimated Time (ET)
Last-In-Pipeline (LIP)	Current Throughput (CT)

Queue-Size Throttling (QST)

- Apply pressure from ingress towards egress
- Focus on one operator at a time
- Each operator has an upper bound on output queue size
- Normalize for selectivity
- Pick earliest operator in the pipeline with output queue size less than its threshold

$$cs_i = \prod_{k=1}^{i} s_i$$

Queue-Size Throttling (QST)

- Apply pressure from ingress towards egress
- Focus on one operator at a time
- Each operator has an upper bound on output queue size
- Normalize for selectivity
- Pick earliest operator in the pipeline with output queue size less than its threshold

$$cs_i = \prod_{k=1}^{i} s_i$$

$$T_i = \frac{C * cs_i}{\sum_{i=1}^{n} cs_i}$$

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Queue-Size Throttling (QST)

Last In Pipeline (LIP)

- Complementary to QST
- Provide suction to pull tuples from ingress towards egress
- Prioritizes operators later in the pipeline
- Operator is “schedulable” if it has
  - Less than maximum allowed workers assigned
  - Minimum input queue size

Last In Pipeline (LIP)

Last In Pipeline (LIP)

Last In Pipeline (LIP)

Last In Pipeline (LIP)

Last In Pipeline (LIP)

Last In Pipeline (LIP)

Last In Pipeline (LIP)

Last In Pipeline (LIP)

Estimated Time (ET)

- Priority-based: Compute a priority score for each operator and assign worker to the one with highest score

- Priority score is estimated time to process the current input queue to completion if an additional worker is assigned

- Intuition: *Operator that will take more time to complete needs additional worker time*

$$p_i = \frac{I_i * c_i}{w_i + 1}$$

Estimated Time (ET)

$$p_i = \frac{I_i * c_i}{w_i + 1}$$

Estimated Time (ET)

$$p_i = \frac{I_i * c_i}{w_i + 1}$$

$p_1 = 40\mu s$ $p_2 = 10\mu s$ $p_3 = 15\mu s$ $p_4 = 70\mu s$ $p_5 = 100\mu s$

Estimated Time (ET)

$$p_i = \frac{I_i * c_i}{w_i + 1}$$

Estimated Time (ET)

$$p_i = \frac{I_i * c_i}{w_i + 1}$$

Estimated Time (ET)

$$p_i = \frac{I_i * c_i}{w_i + 1}$$

Current Throughput (CT)

- Schedule the operator with lowest throughput as it is likely to be bottleneck in the pipeline
- Normalize for selectivity
- Divide time into windows of size w, compute "effective" number of tuples processed by operator in w
- Choose operator with smallest n_i^w

$$n_i^w = \frac{T_i^w + (w_i \times s)}{c_i \times cs_{i-1}}$$

Current Throughput (CT)

$$n_i^w = \frac{T_i^w + (w_i \times s)}{c_i \times cs_{i-1}}$$

Current Throughput (CT)

$$n_i^w = \frac{T_i^w + (w_i \times s)}{c_i \times cs_{i-1}}$$

Current Throughput (CT)

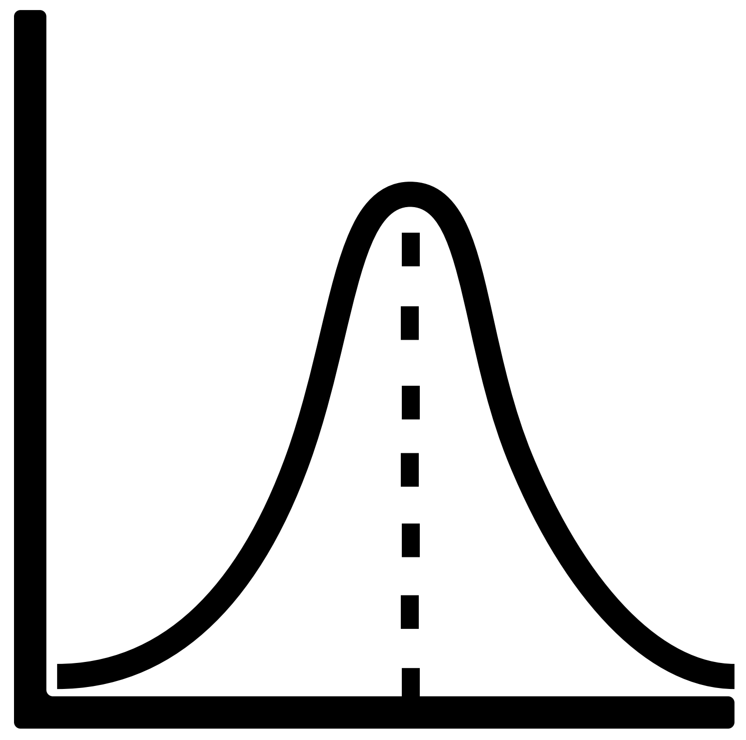
$$n_i^w = \frac{T_i^w + (w_i \times s)}{c_i \times cs_{i-1}}$$

$n_1^w = 2M$ $n_2^w = 0.33M$ $n_3^w = 5M$ $n_4^w = 1M$ $n_5^w = 10M$

Current Throughput (CT)

$$n_i^w = \frac{T_i^w + (w_i \times s)}{c_i \times cs_{i-1}}$$

$n_1^w = 2M$ $n_2^w = 0.75M$ $n_3^w = 5M$ $n_4^w = 1M$ $n_5^w = 10M$



Evaluation

Evaluation

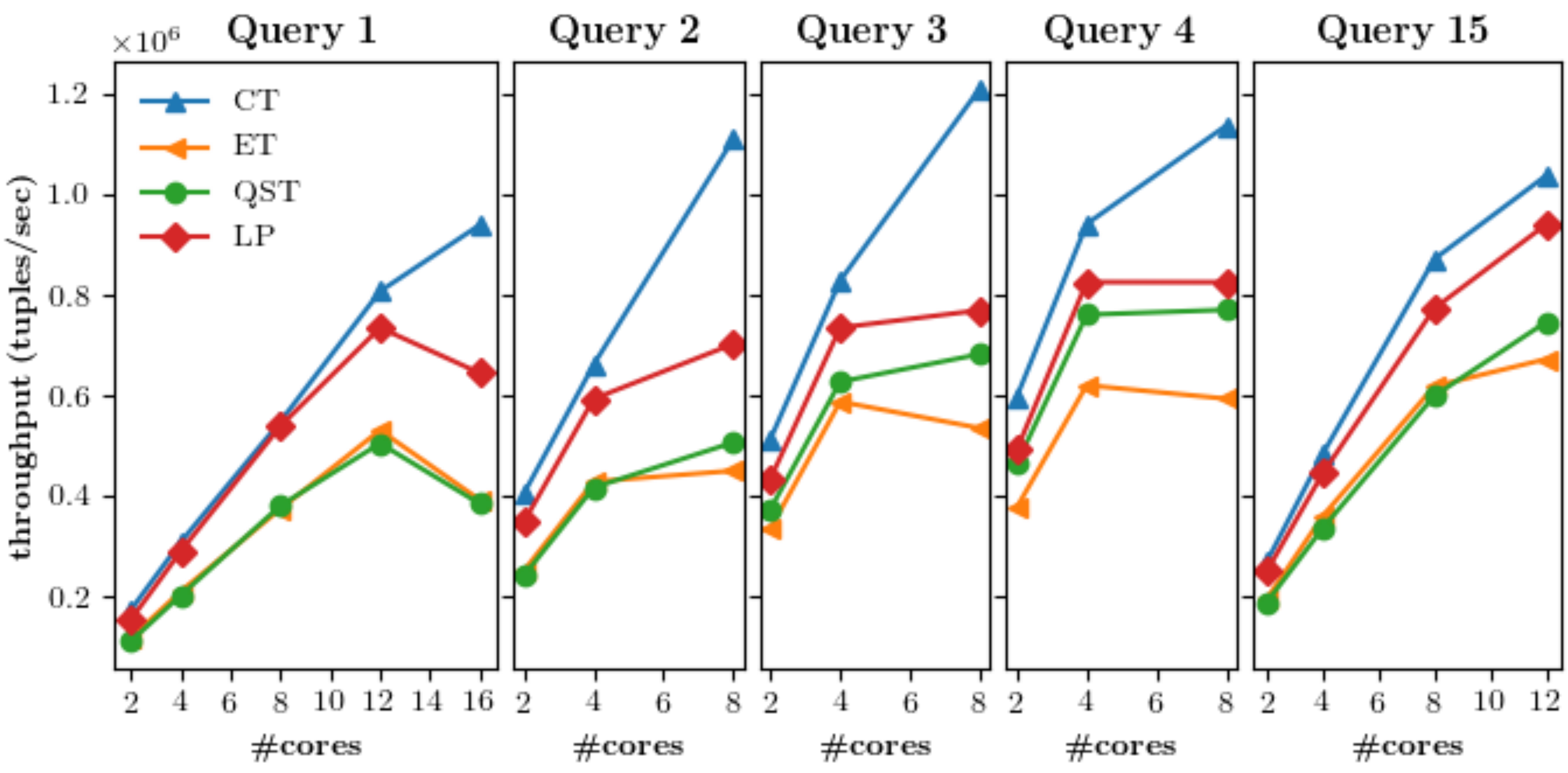
Experimental Setup

- Intel Xeon E5 Family 2698B v3 series
- Windows Server 2012 R2 Datacenter
- 16 Physical Cores
- Cache sizes: 32KB, 256KB & 40MB
- Steady-state throughput, latency

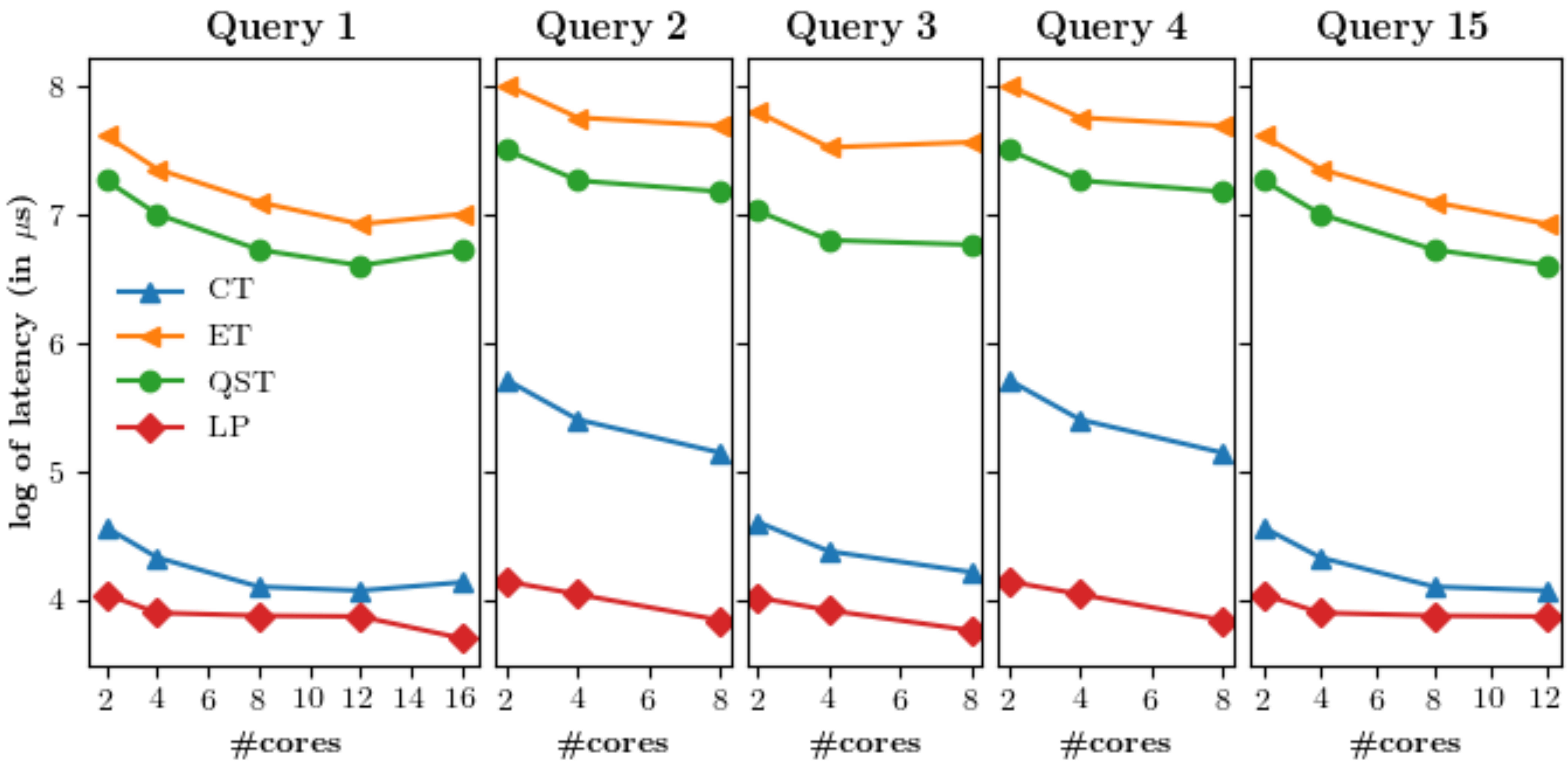
Workloads

- TPCx-BB (Big Bench) Benchmark
  - Modern Big Data Benchmark
  - Q1-4, Q15 are streaming queries
  - Eg. "*Find top 30 products that are viewed together online*"
- Micro-benchmarks

Scheduling - Throughput



Scheduling - Latency

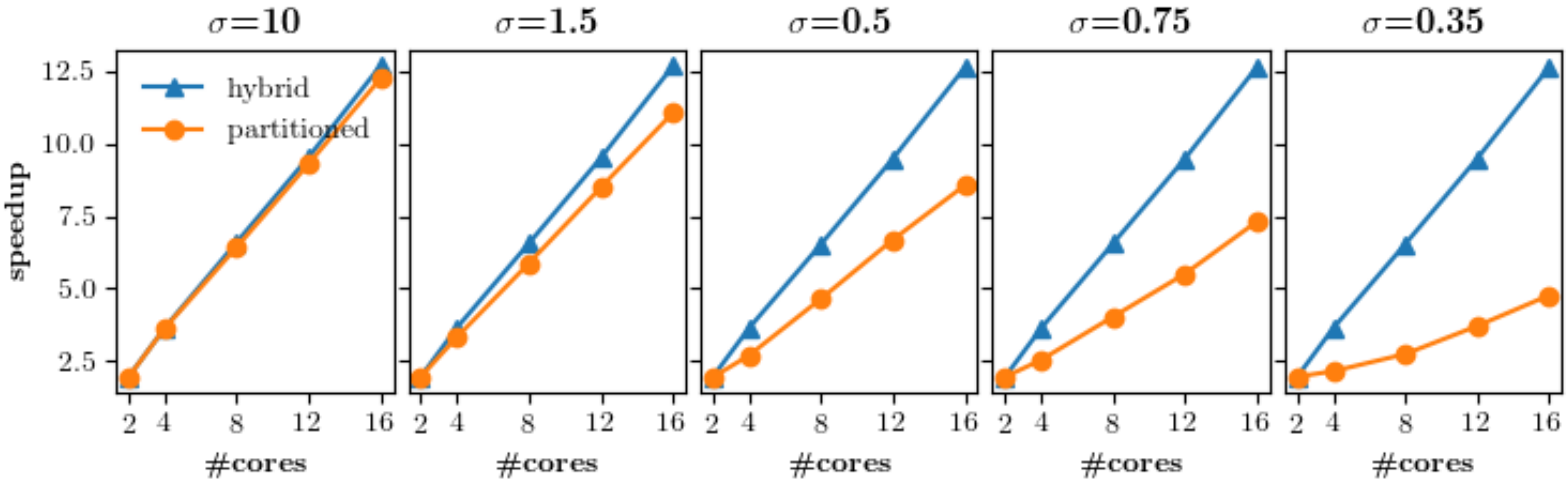


Scheduling - Analysis

- Even when total worker time distribution is same, the “throughput” is different for different heuristics!
- Heuristics that distribute workers across the operators
  - Establish a continuous pipelined flow
  - Yields better throughput and latency
- Heuristics that focus on a single operator at a time
  - Prioritizes data parallelism over pipeline parallelism
  - Suffer from overheads of exploiting data parallelism in ordered setting

Partitioned Stateful Schemes

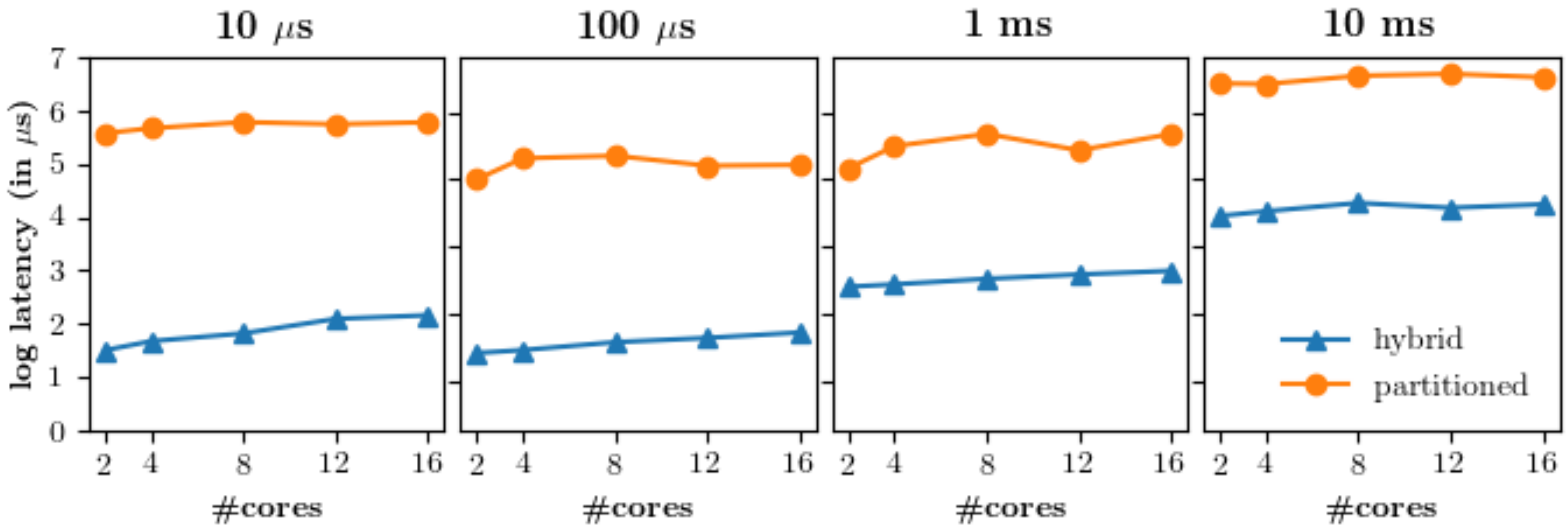
Load Imbalance Across Partitions



Hybrid strategy can afford finer partitions and hence better load balance!

Partitioned Stateful Schemes

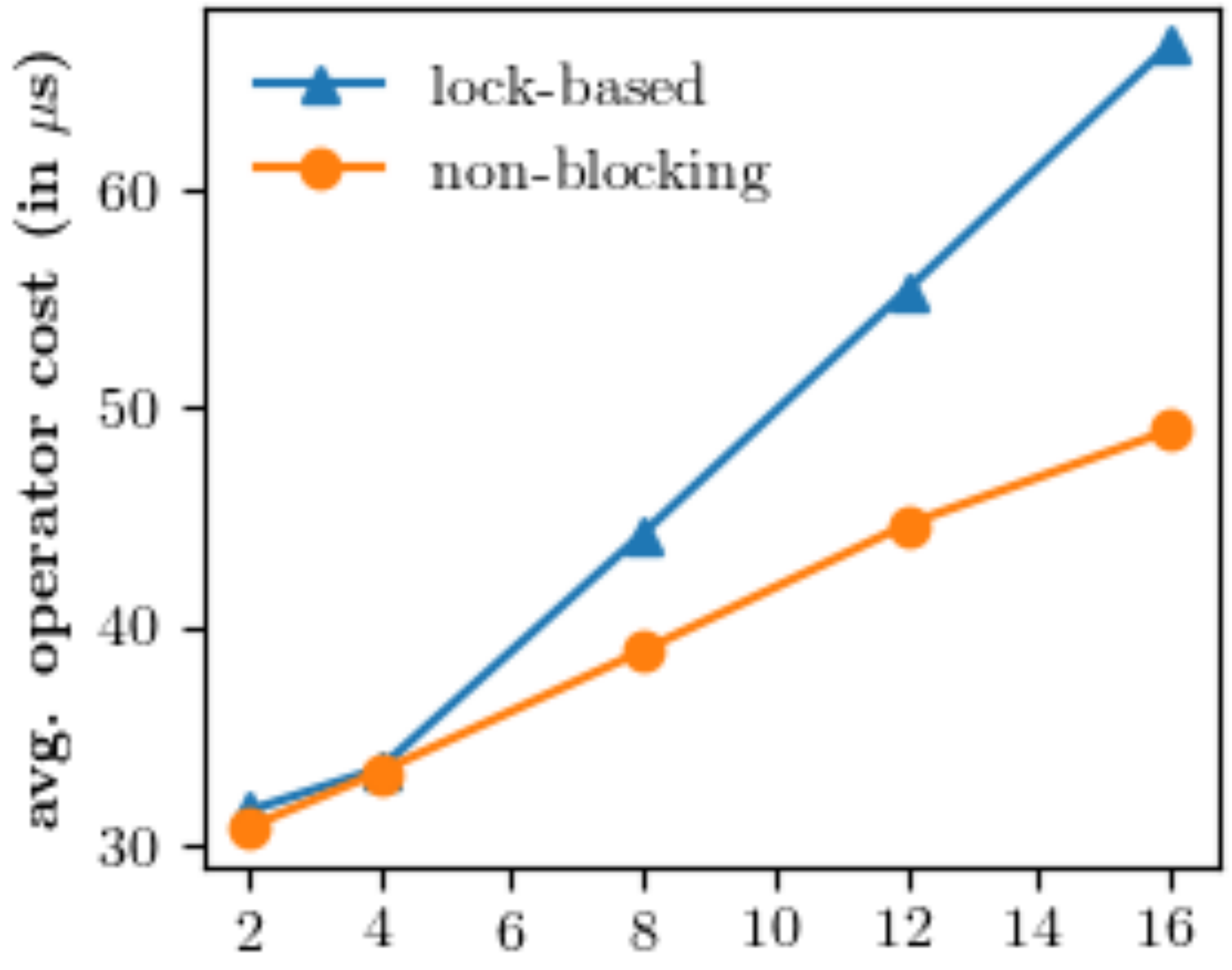
Latency for different operator costs



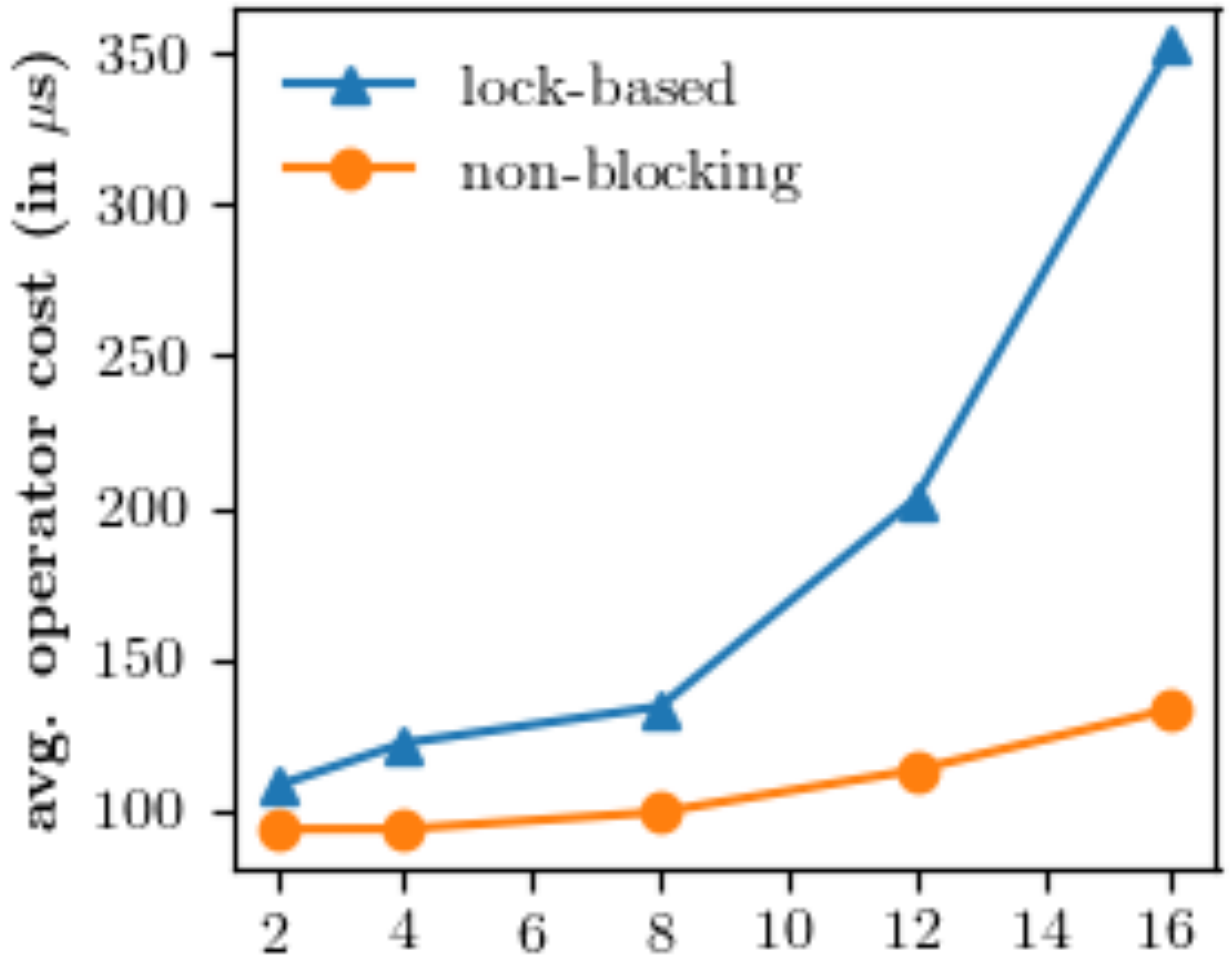
Partitioned scheme blocks outputs, increasing latency!

Output Reordering Schemes

Lightweight Operators



High Selectivity Operators



Non-blocking reordering scheme prevents unnecessary worker blocking

Conclusion

Conclusion

- Framework for parallelizing ordered stream computations on shared-memory multicores
- Implementation of data-parallel operators in the ordered setting
  - Reordering outputs without worker blocking
  - Processing partitioned stateful operators in almost arrival order
- Proposed heuristics for dynamically scheduling stream operators and compared them empirically

Questions?