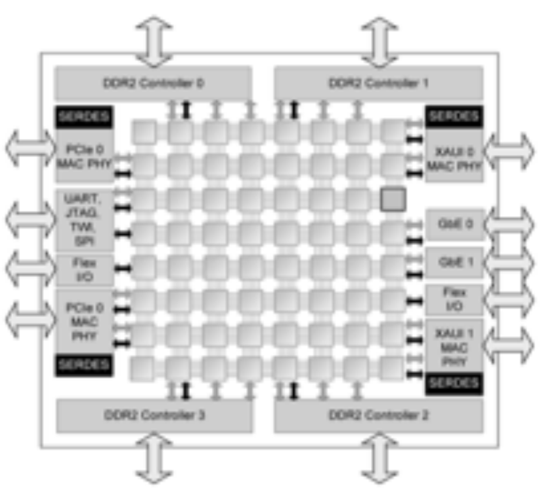
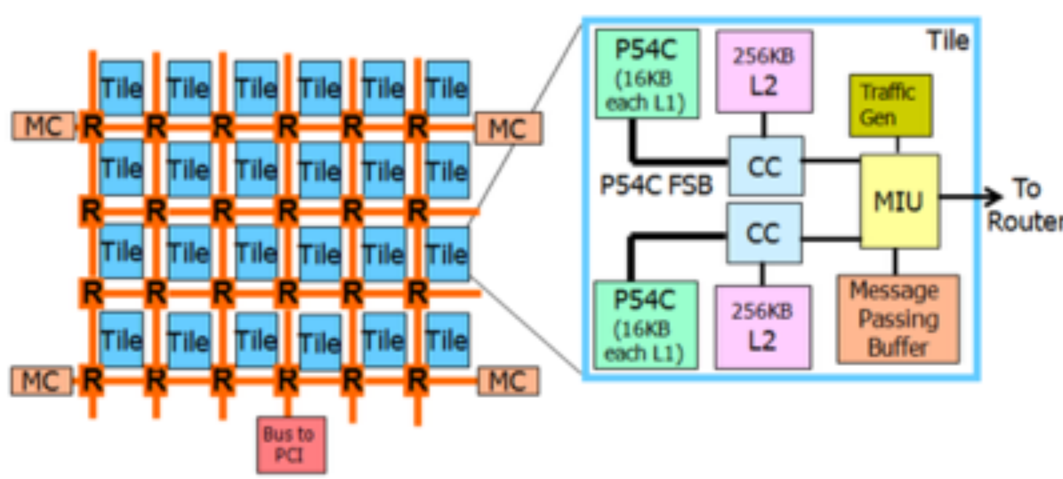
Embracing Explicit Communication in Work-Stealing Runtime Systems

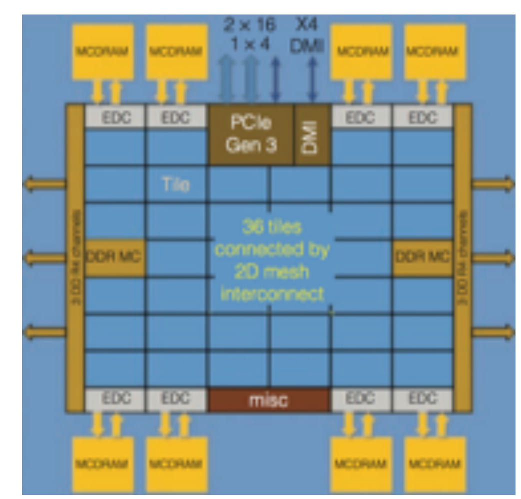
Andreas Prell

Manycore Processors

- “Cluster-on-chip” architectures
- Increasing thread- and data-level parallelism
- Growing importance of scalable communication





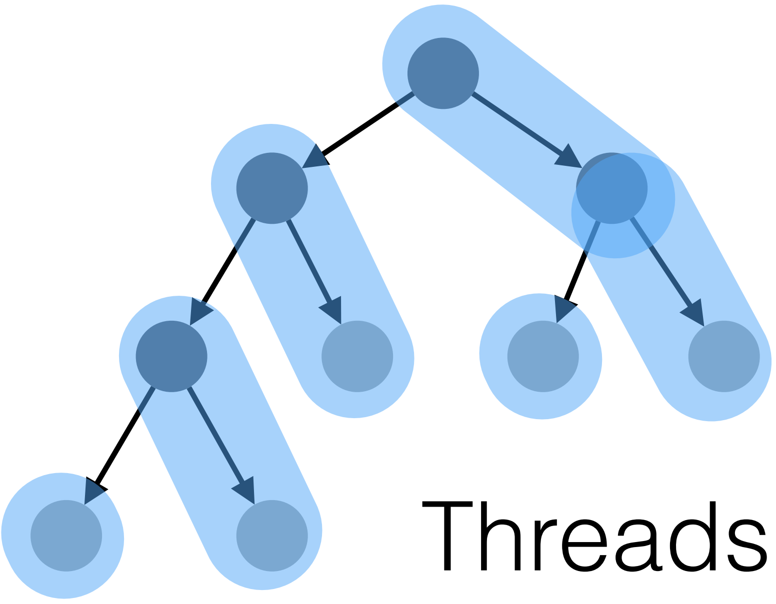


Left: S. Bell et al.,TILE64™ Processor: A 64-Core SoC with Mesh Interconnect, ISSCC 2008
Center: T. Mattson et al., The 48-Core SCC Processor: The Programmer’s View, SC 2010
Right: A. Sodani et al., Knights Landing: Second-Generation Intel Xeon Phi Product, IEEE Micro 2016

From Threads to Tasks

Make it easy to express fine-grained task parallelism

```
int recurse(int n)
{
    if (n < 2) return base_case();

    int x;

    std::thread t([&] {
        x = recurse(n-1);
    });

    int y = recurse(n-2);

    t.join();

    return x + y;
}
```

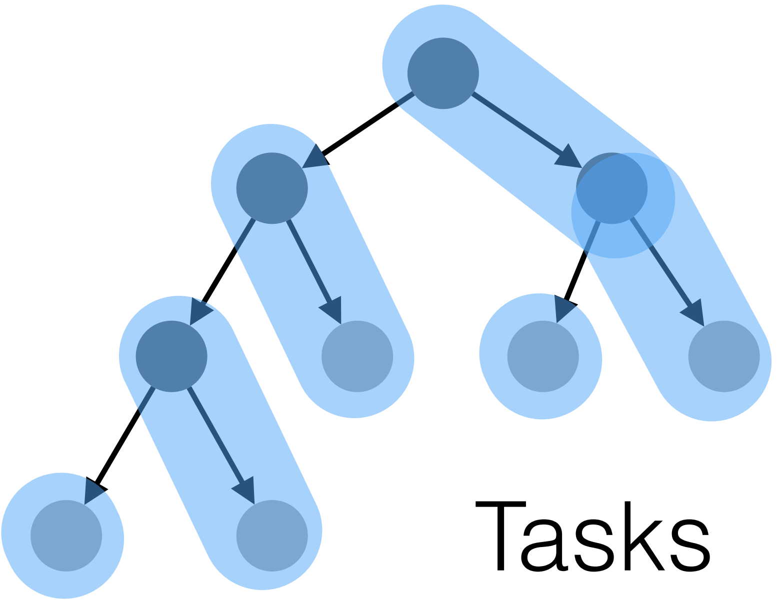


Threads

From Threads to Tasks

Make it easy to express fine-grained task parallelism

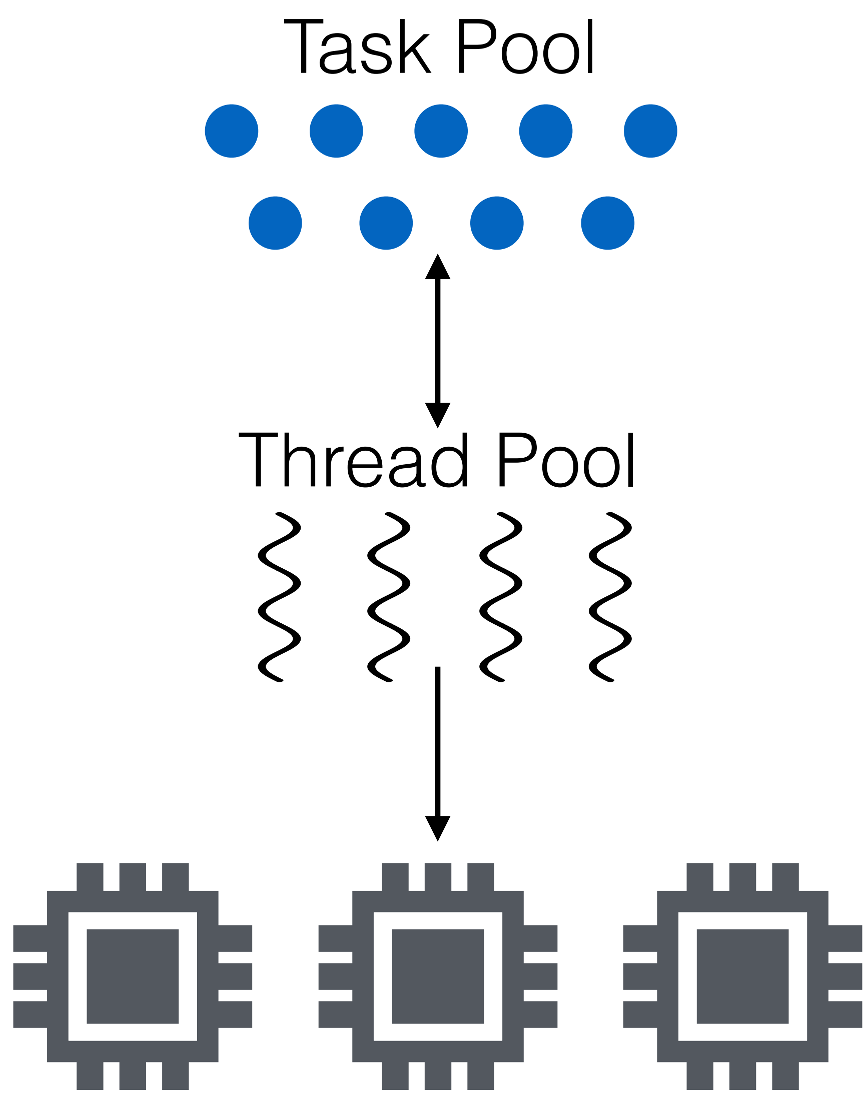
```
int recurse(int n)
{
    if (n < 2) return base_case();

    int x = spawn recurse(n-1);
    int y = recurse(n-2);

    sync;

    return x + y;
}
```



Tasks

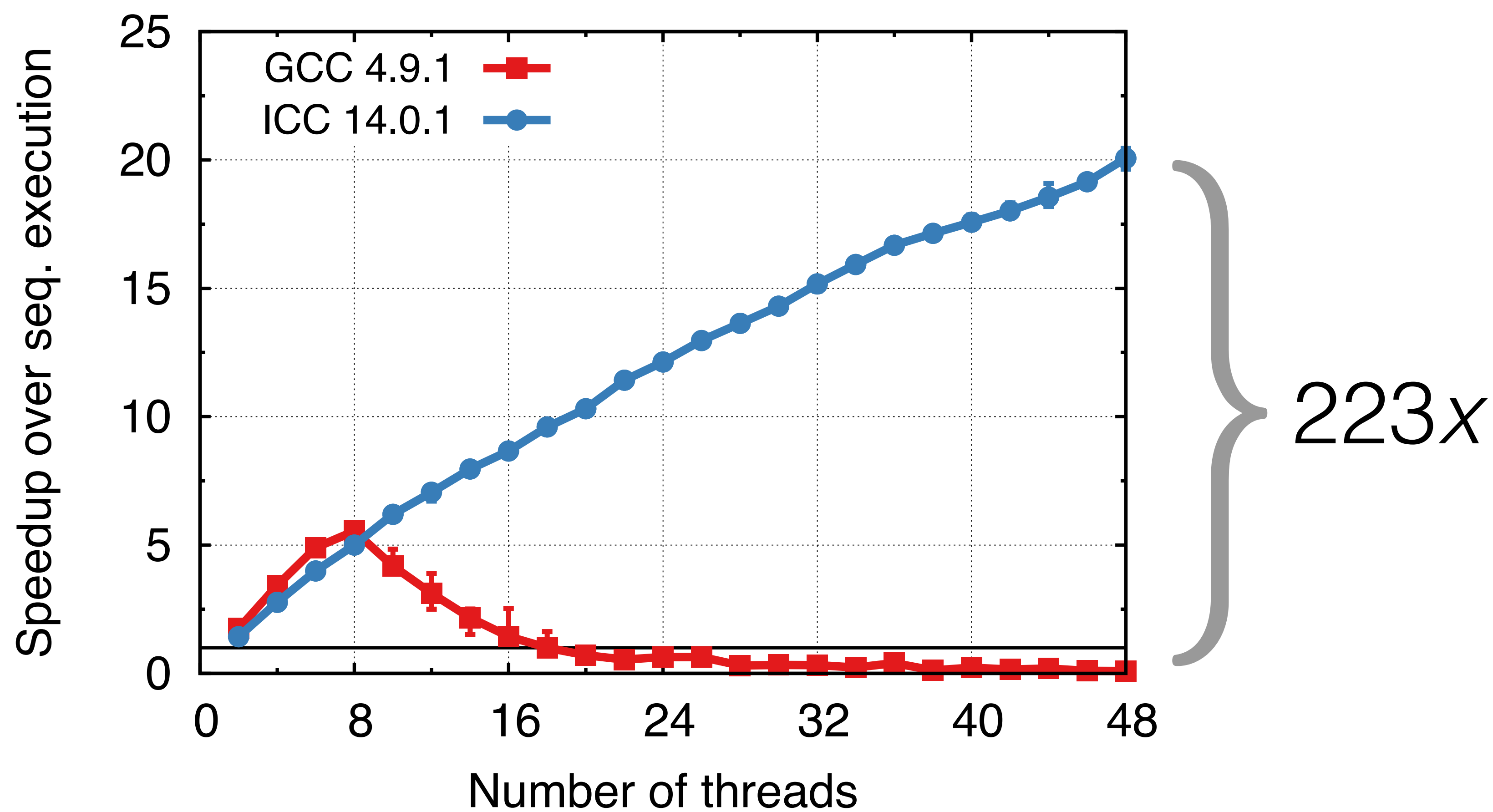
From Threads to Tasks

Runtime system manages parallel execution



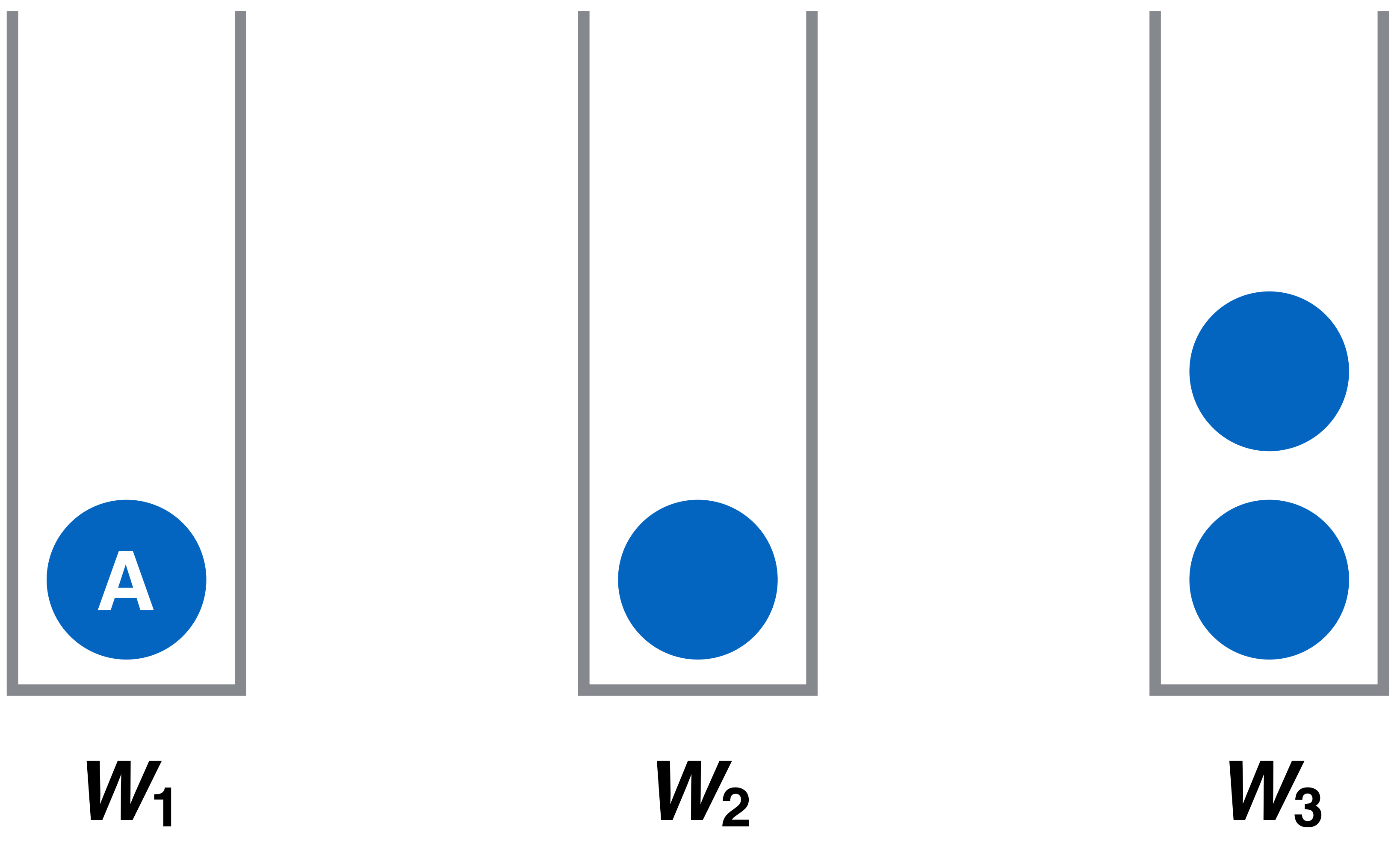
Central versus Distributed Task Pools

GCC: GNU libgomp, ICC: Intel OpenMP RTL

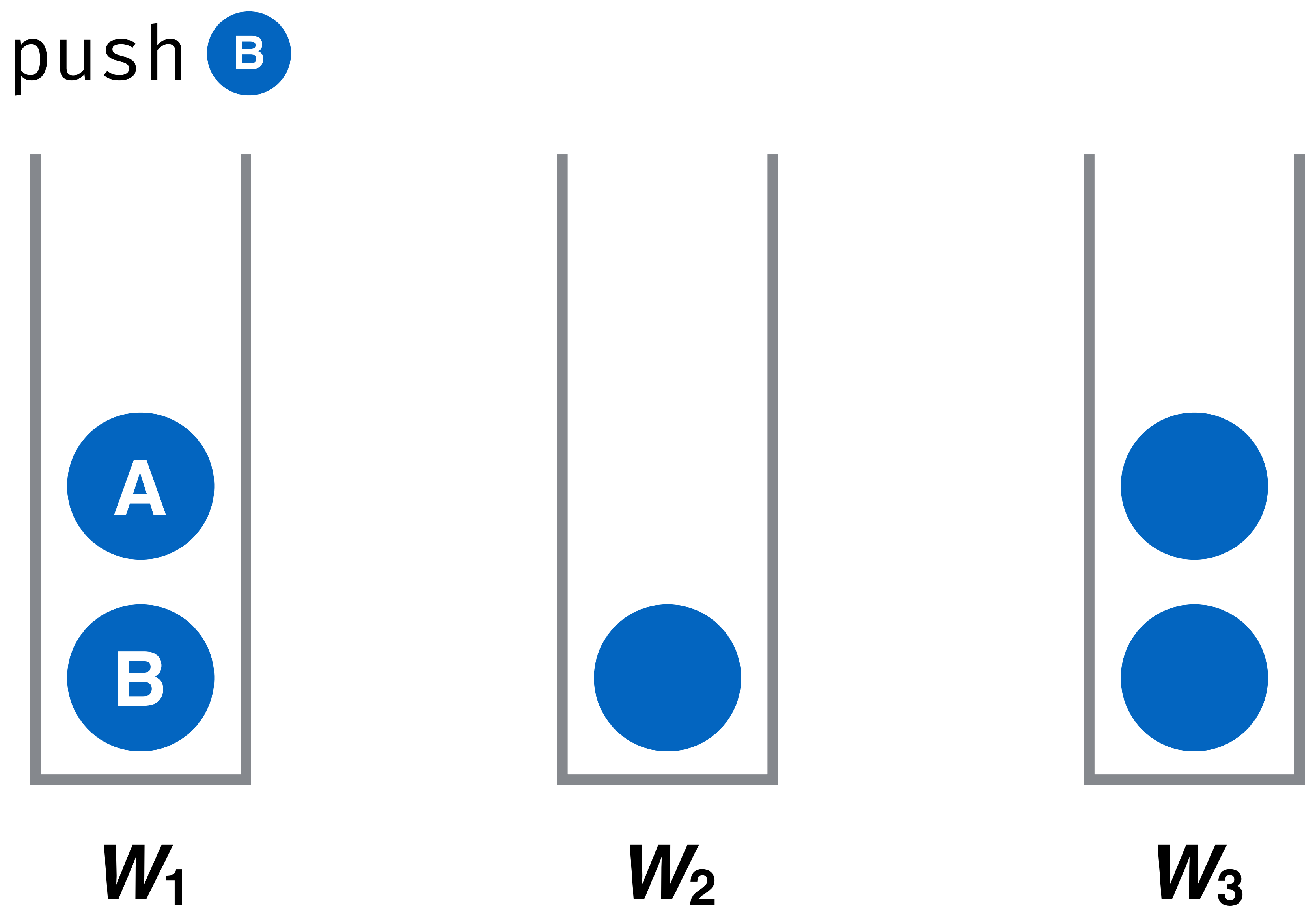


Benchmark: UTS T3L (binomial tree of ~111 million nodes)

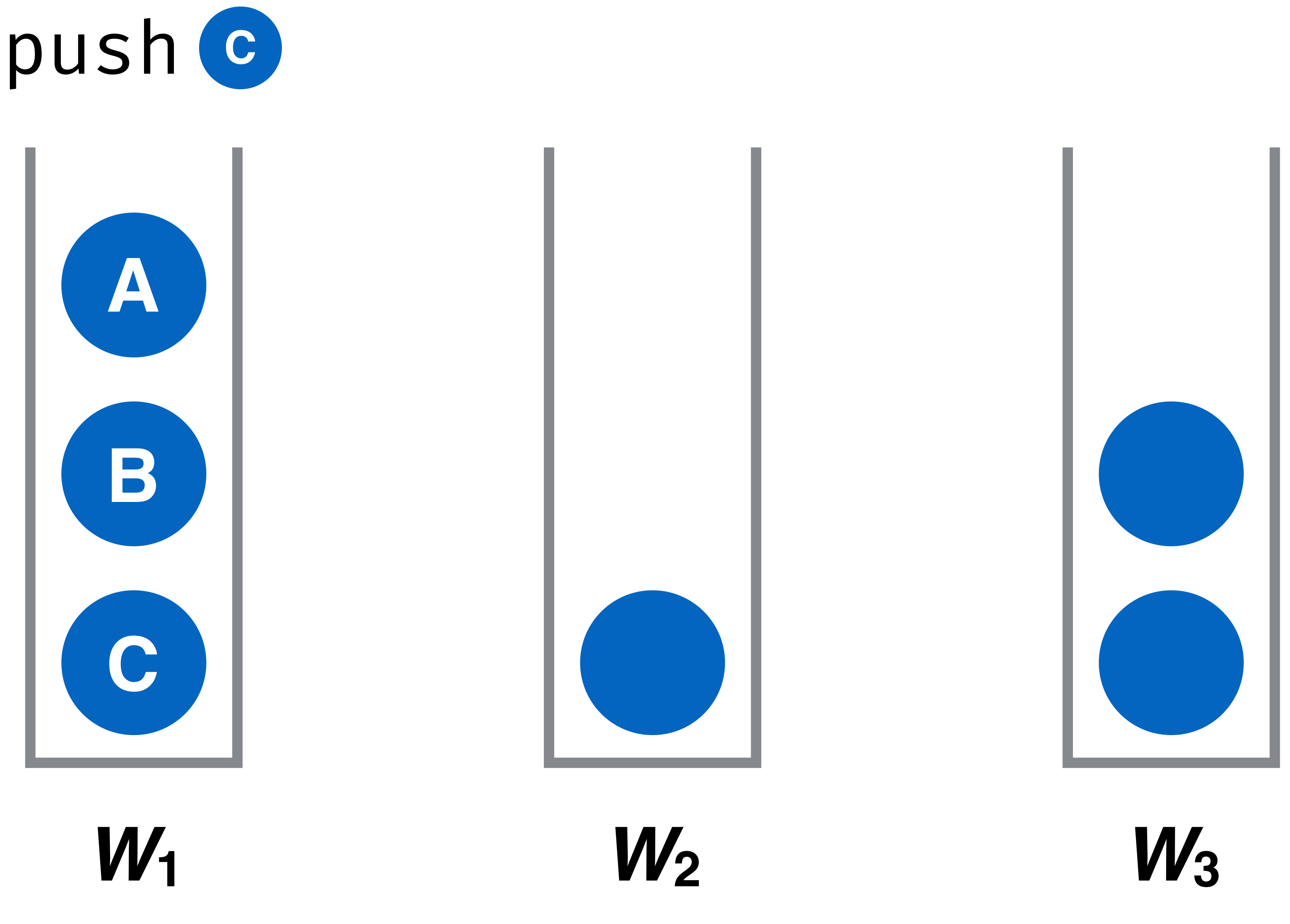
Load Balancing through Work Stealing



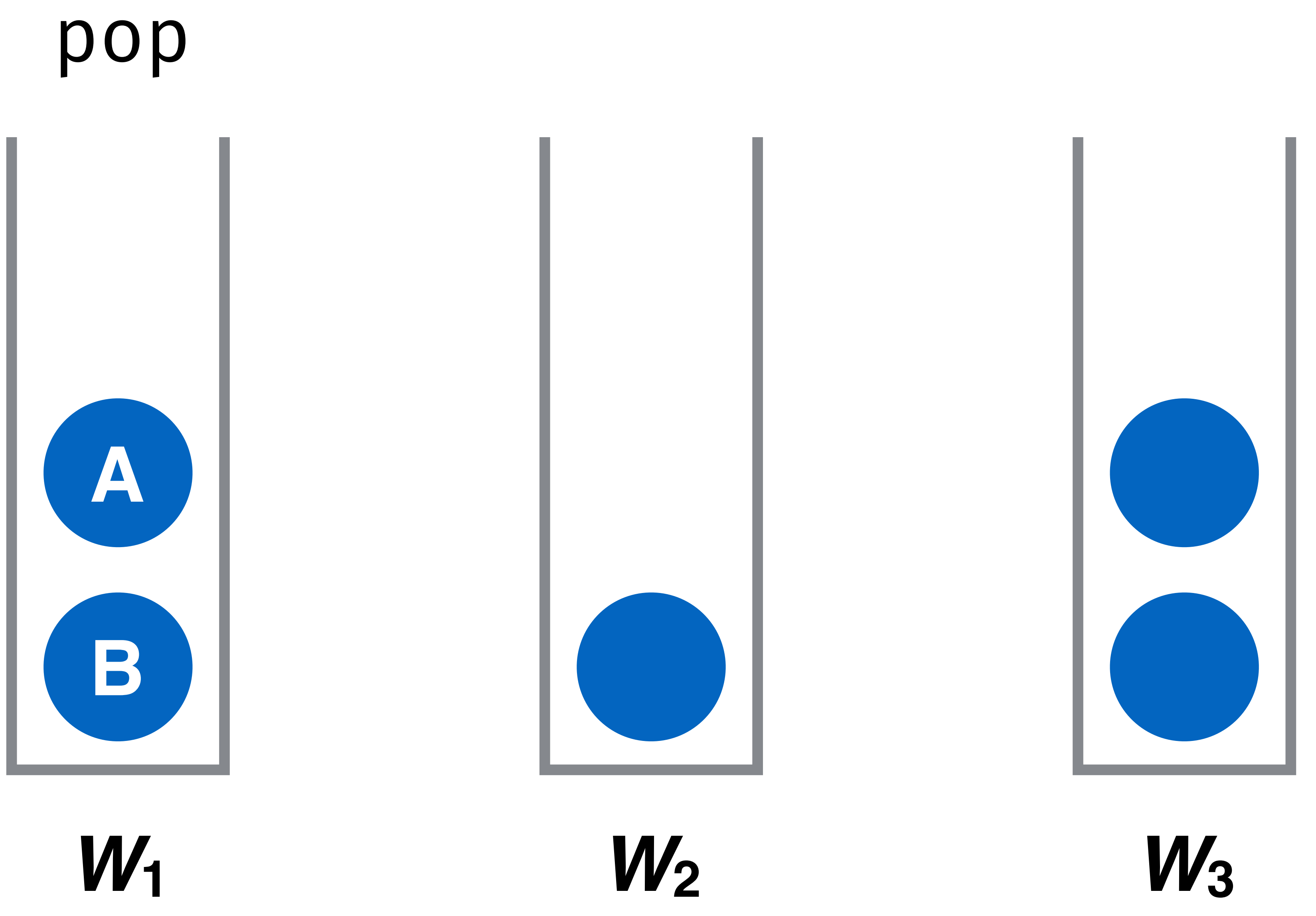
Load Balancing through Work Stealing



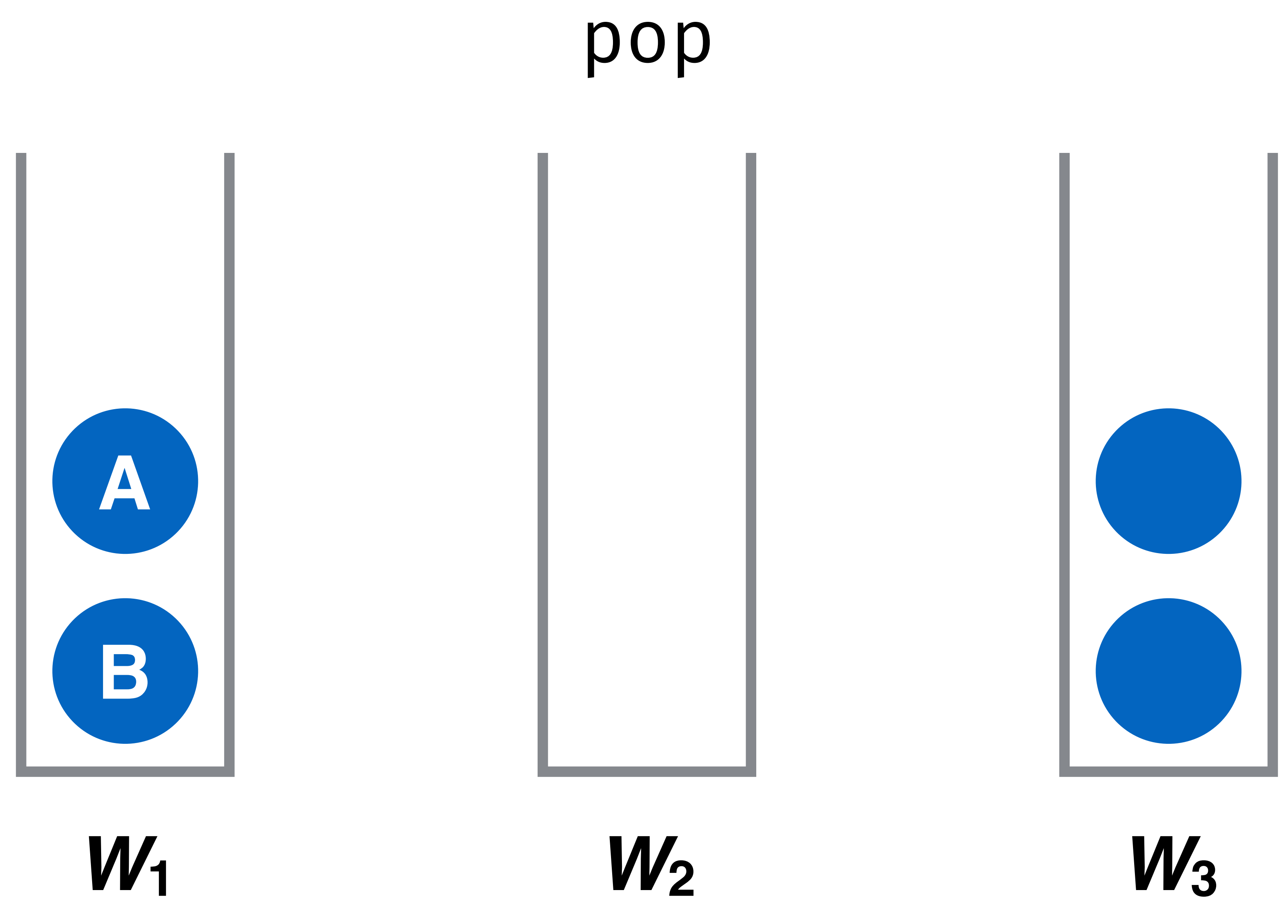
Load Balancing through Work Stealing



Load Balancing through Work Stealing

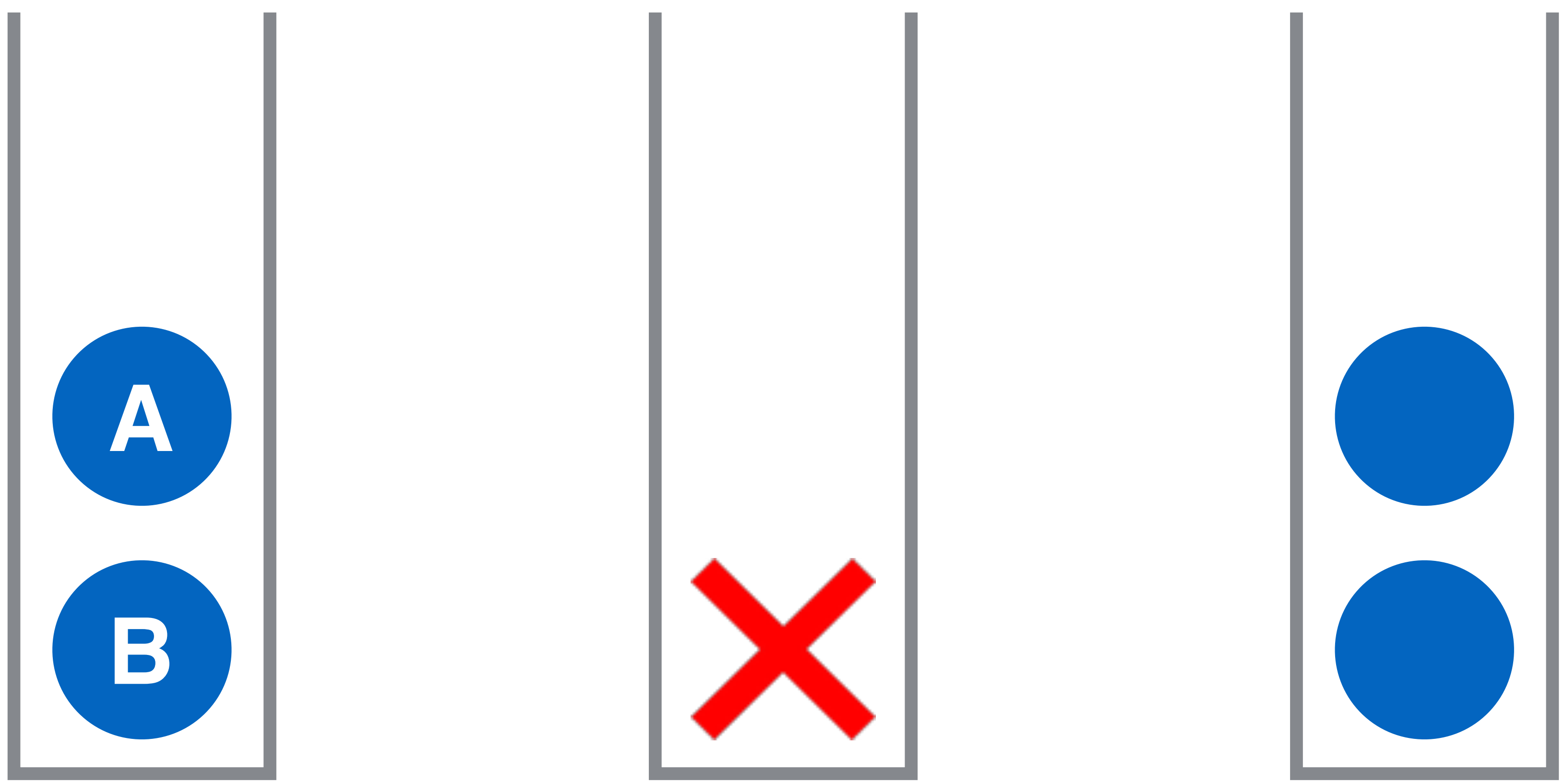


Load Balancing through Work Stealing



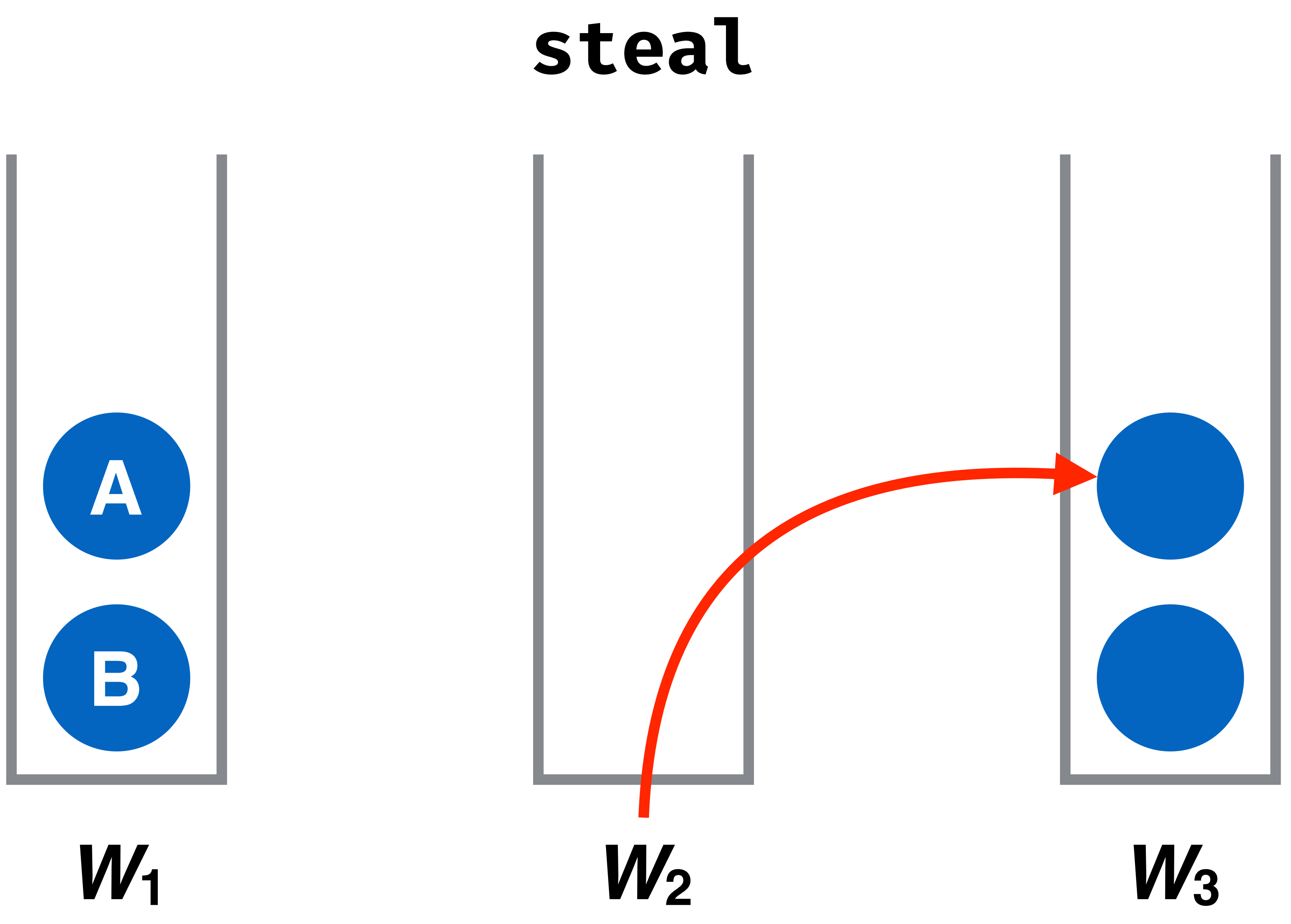
Load Balancing through Work Stealing

pop

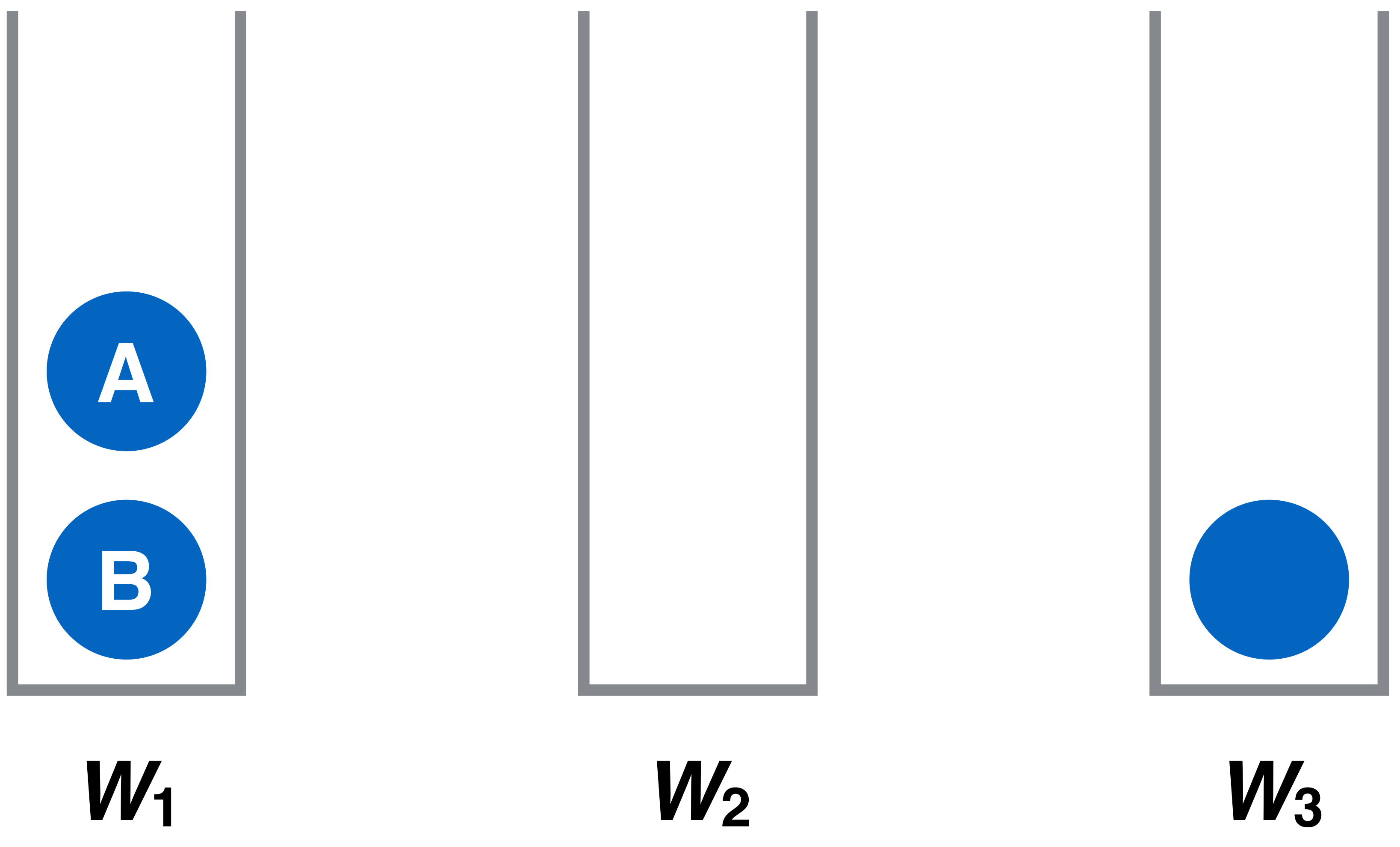


W_1 W_2 W_3

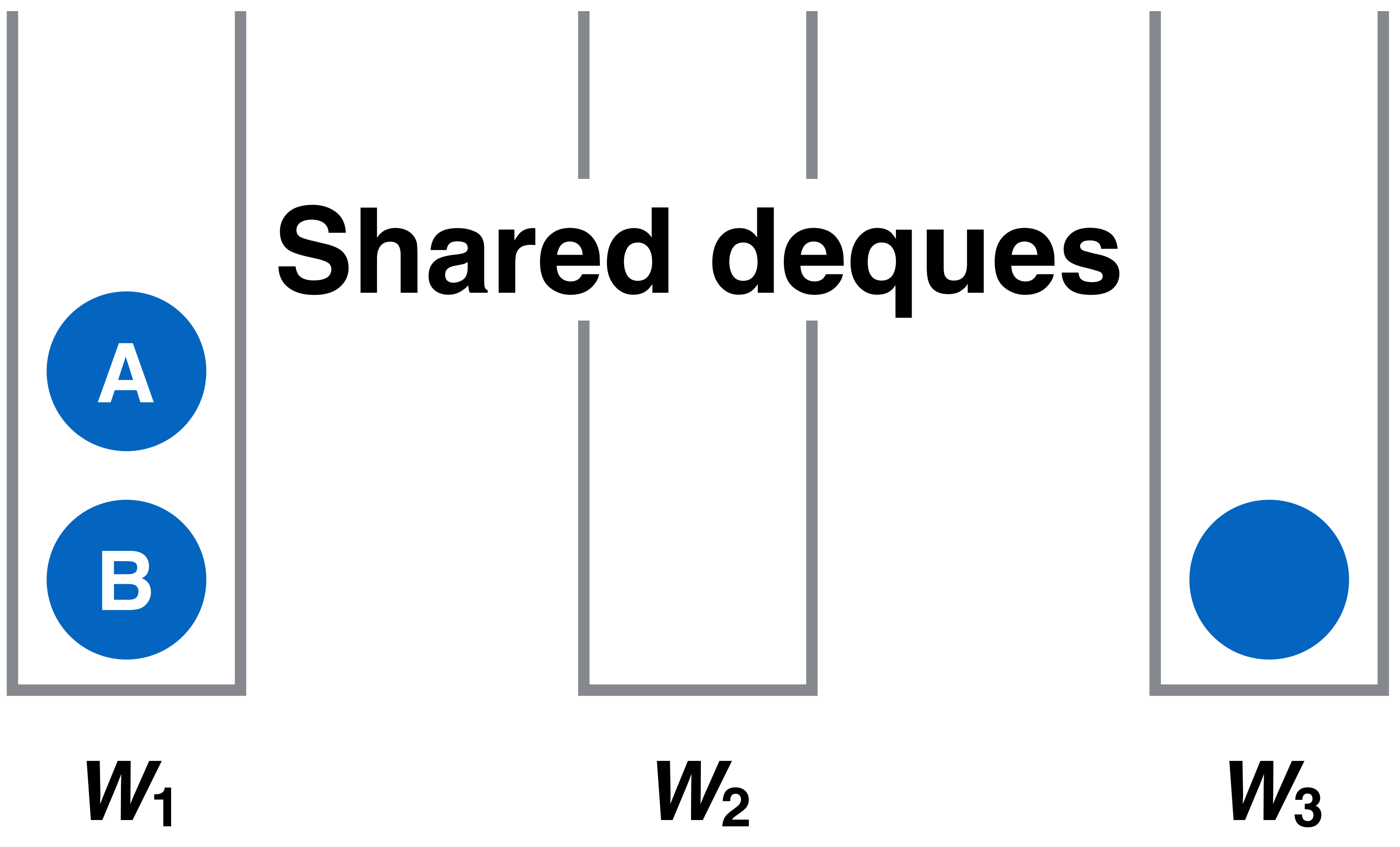
Load Balancing through Work Stealing



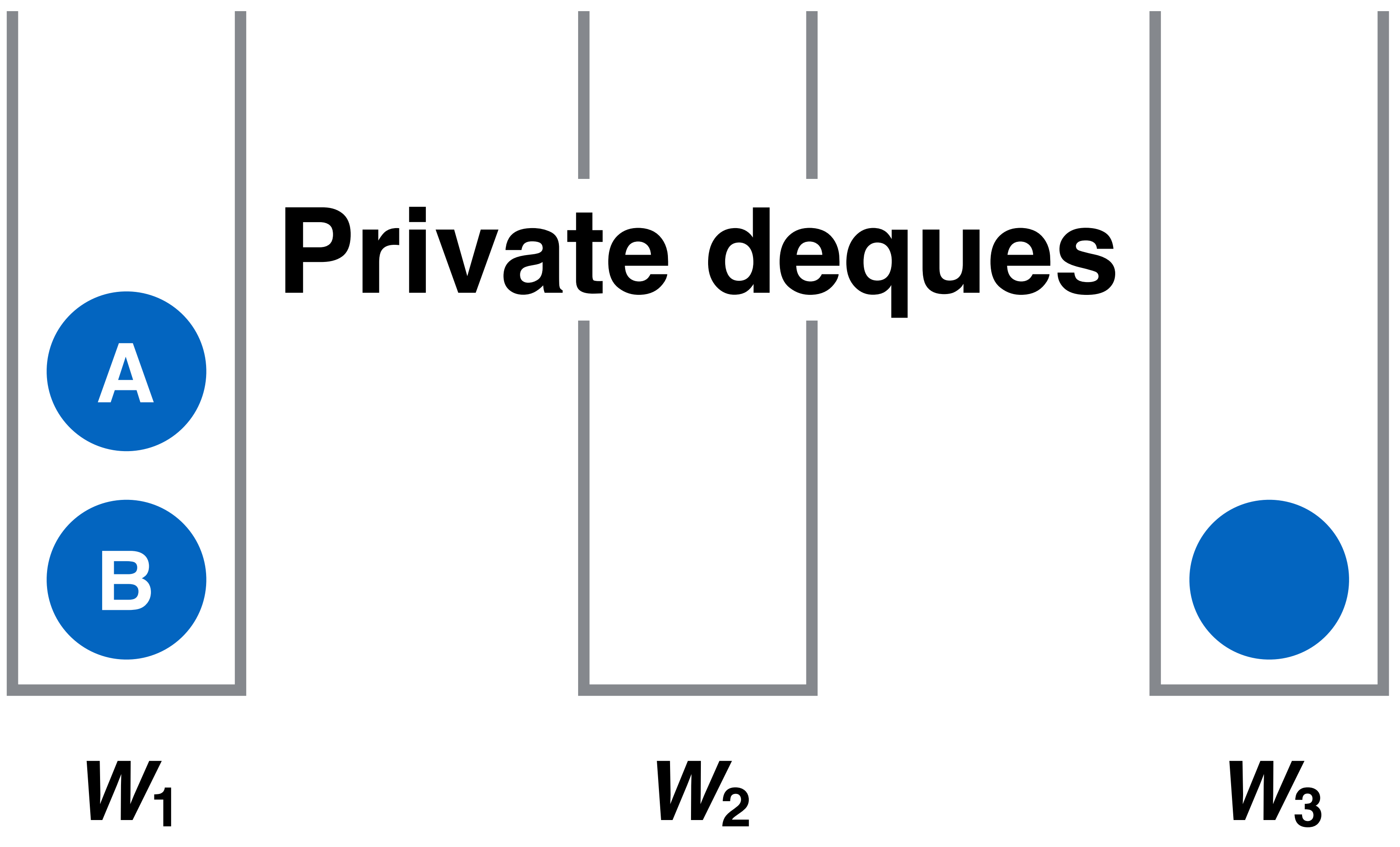
Load Balancing through Work Stealing



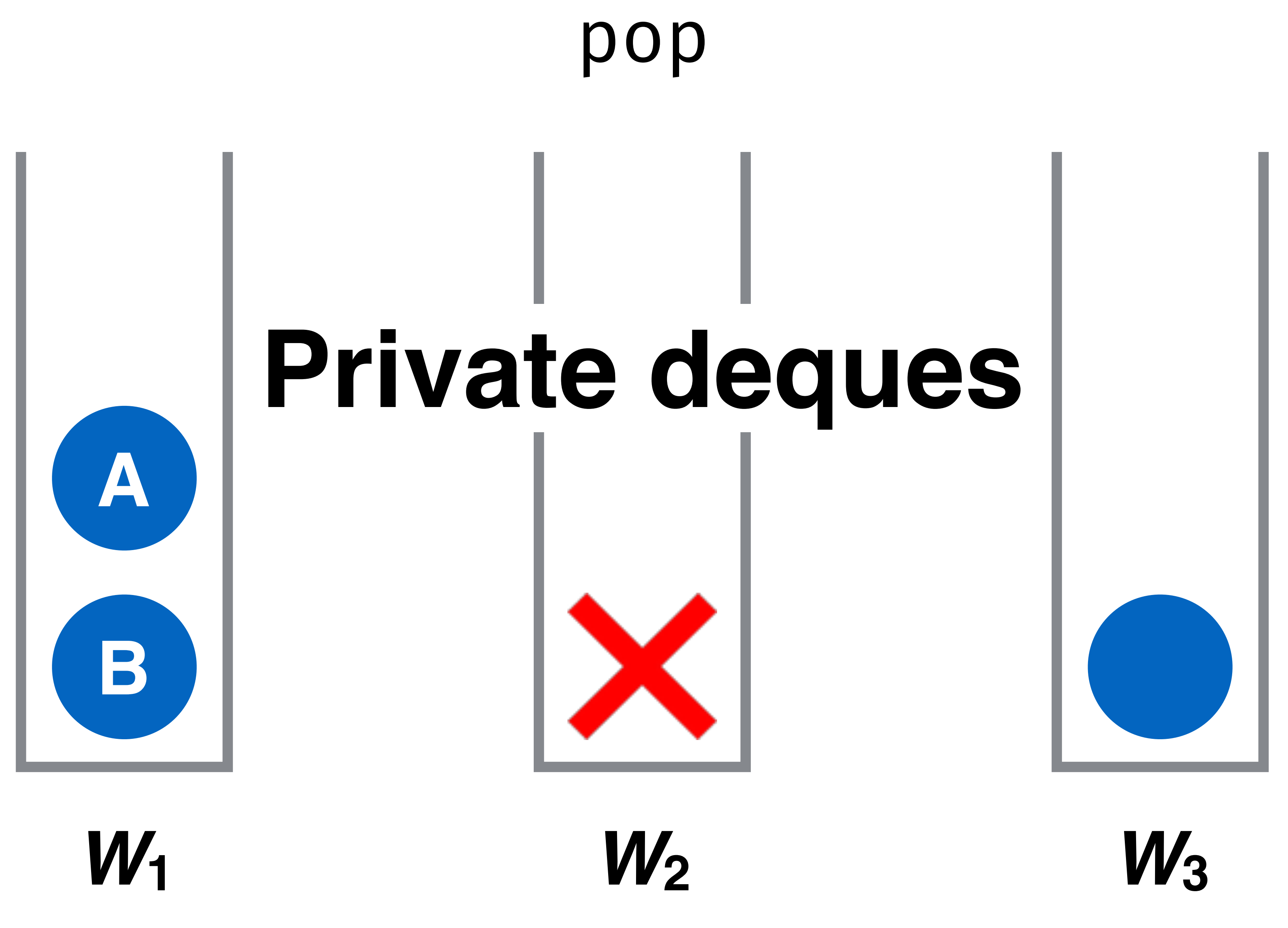
Load Balancing through Work Stealing



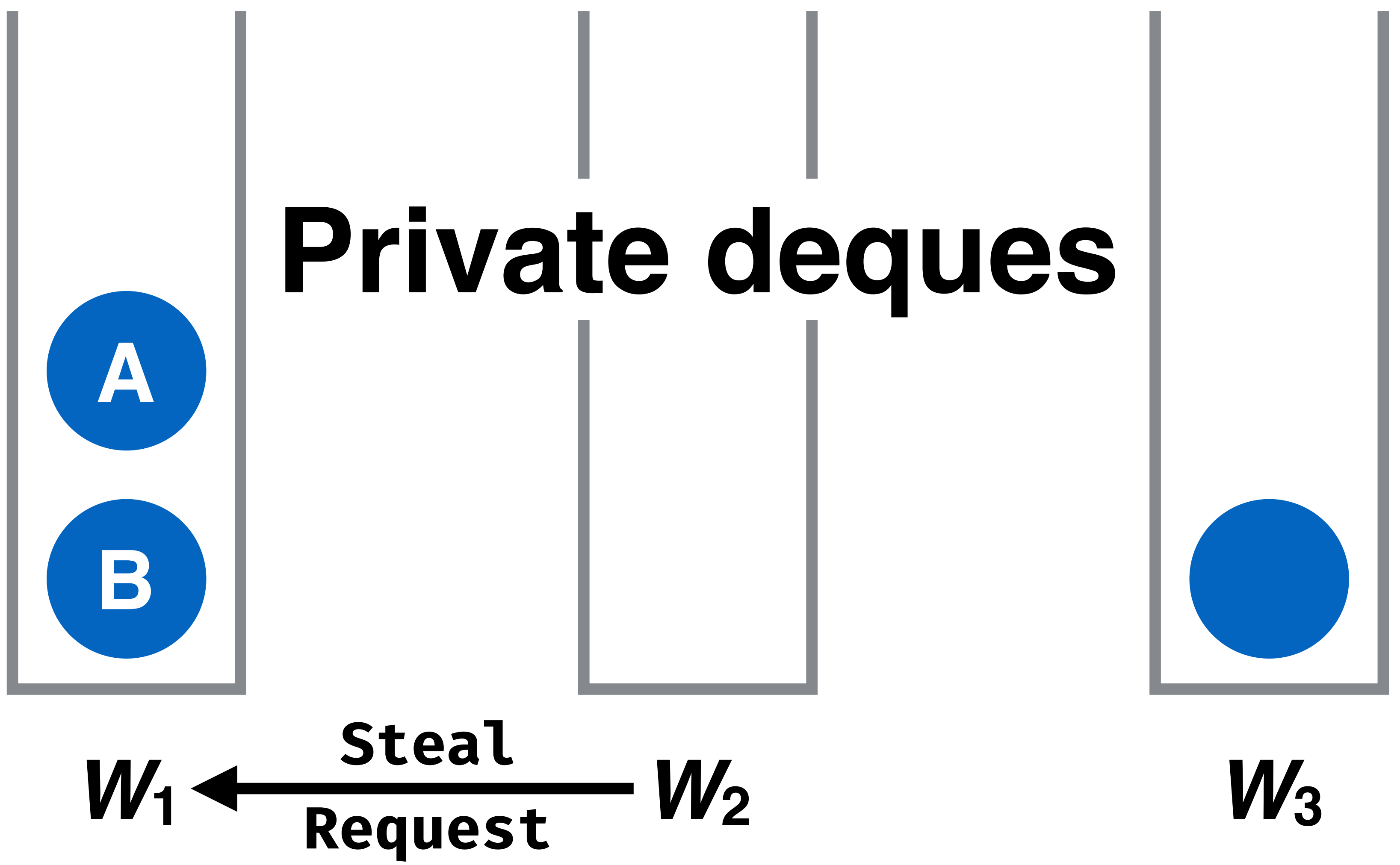
Load Balancing through Work Stealing



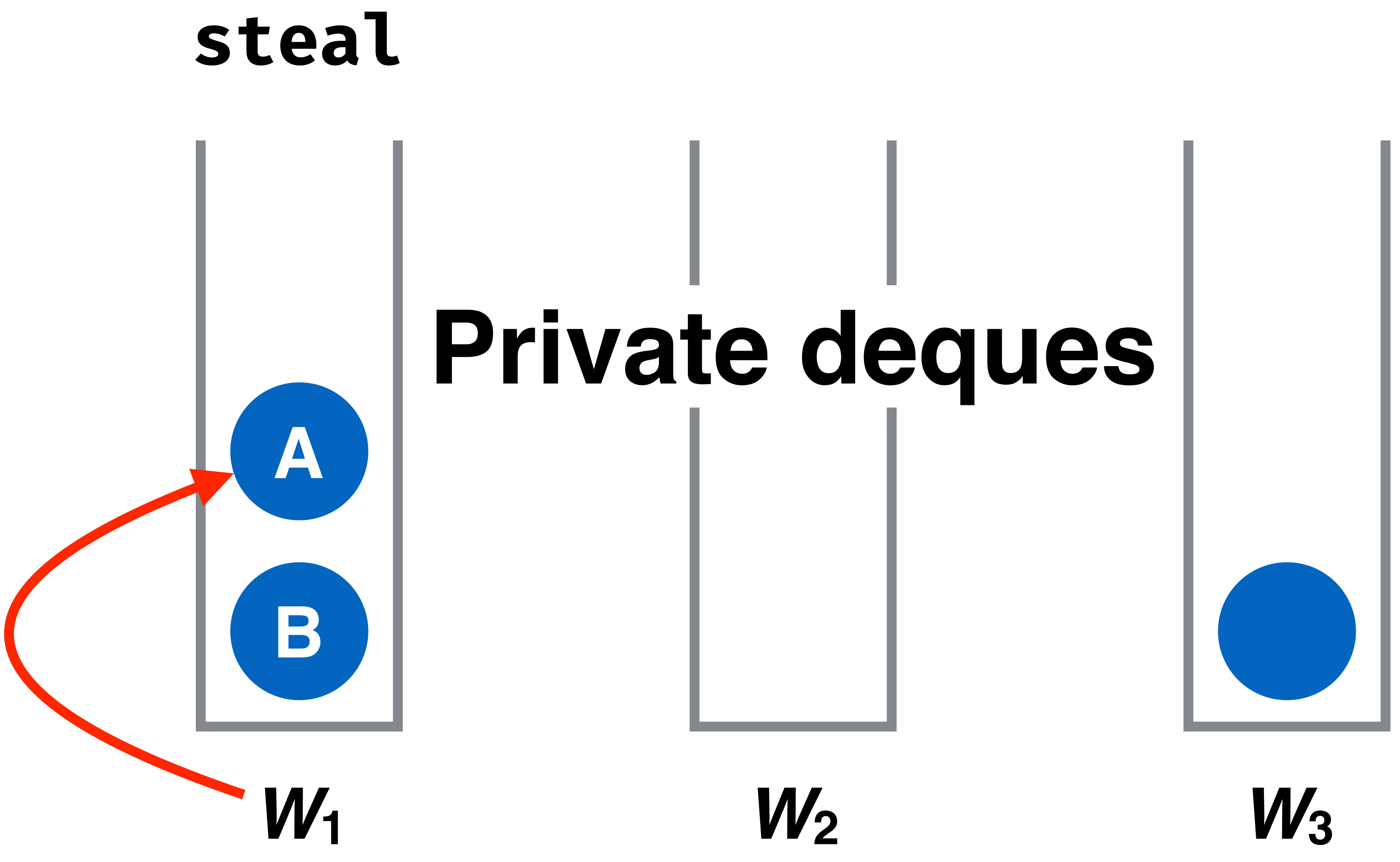
Load Balancing through Work Stealing



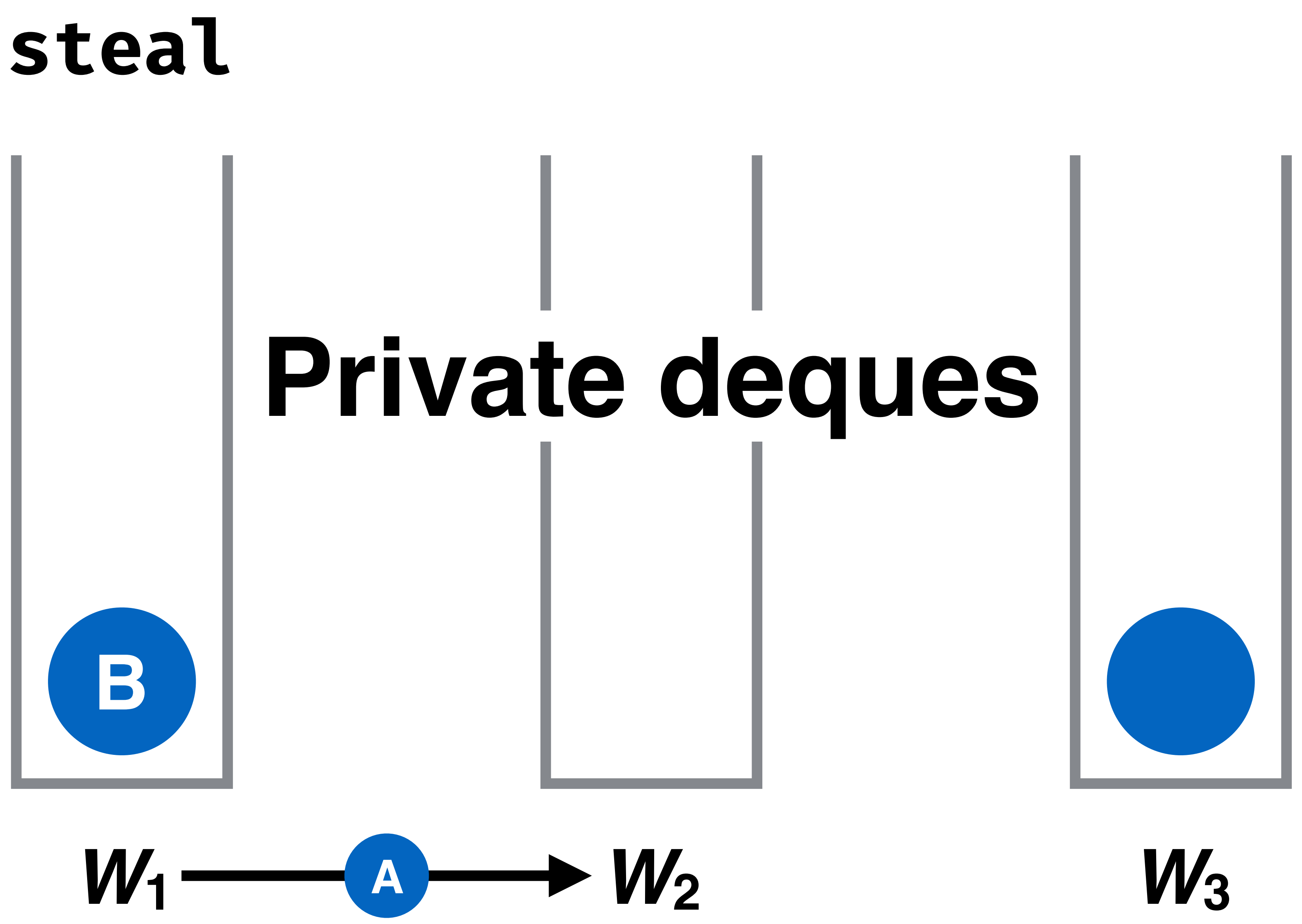
Load Balancing through Work Stealing



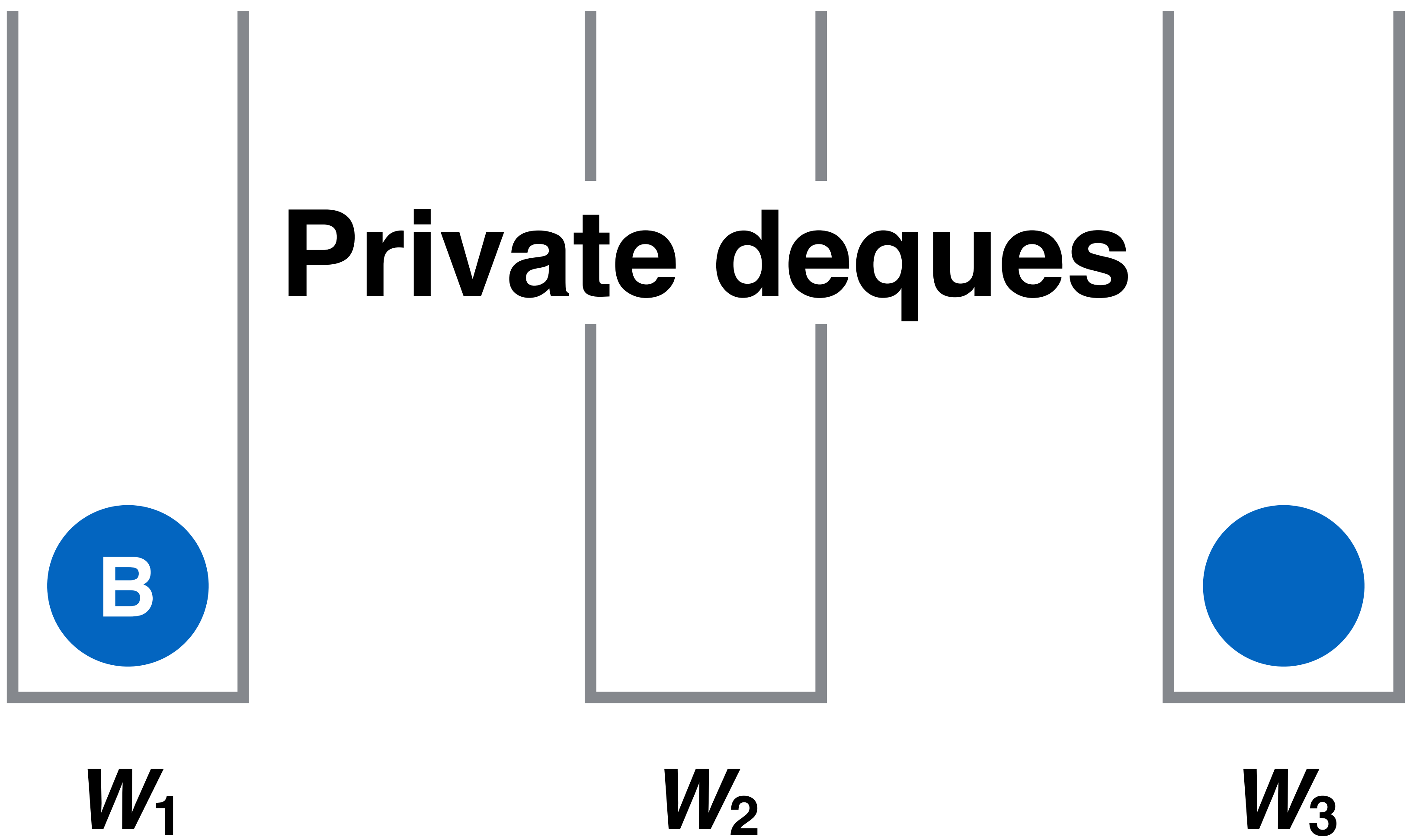
Load Balancing through Work Stealing



Load Balancing through Work Stealing



Load Balancing through Work Stealing



Embracing Explicit Communication

Requirements

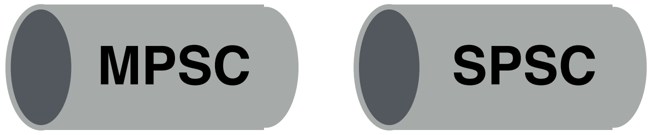
- Work-stealing scheduling
- Explicit communication → Efficient message passing, private-access deques
- Task synchronization → Collective, individual
- Coarse-grained parallelism → Polling
- Fine-grained parallelism → Adaptive stealing strategies, granularity control

Channels

Simple message passing abstraction:

```
bool channel_send(Channel *, void *, size_t);
bool channel_receive(Channel *, void *, size_t);
```

Bounded FIFO message queues

Building blocks: 

Steal Requests

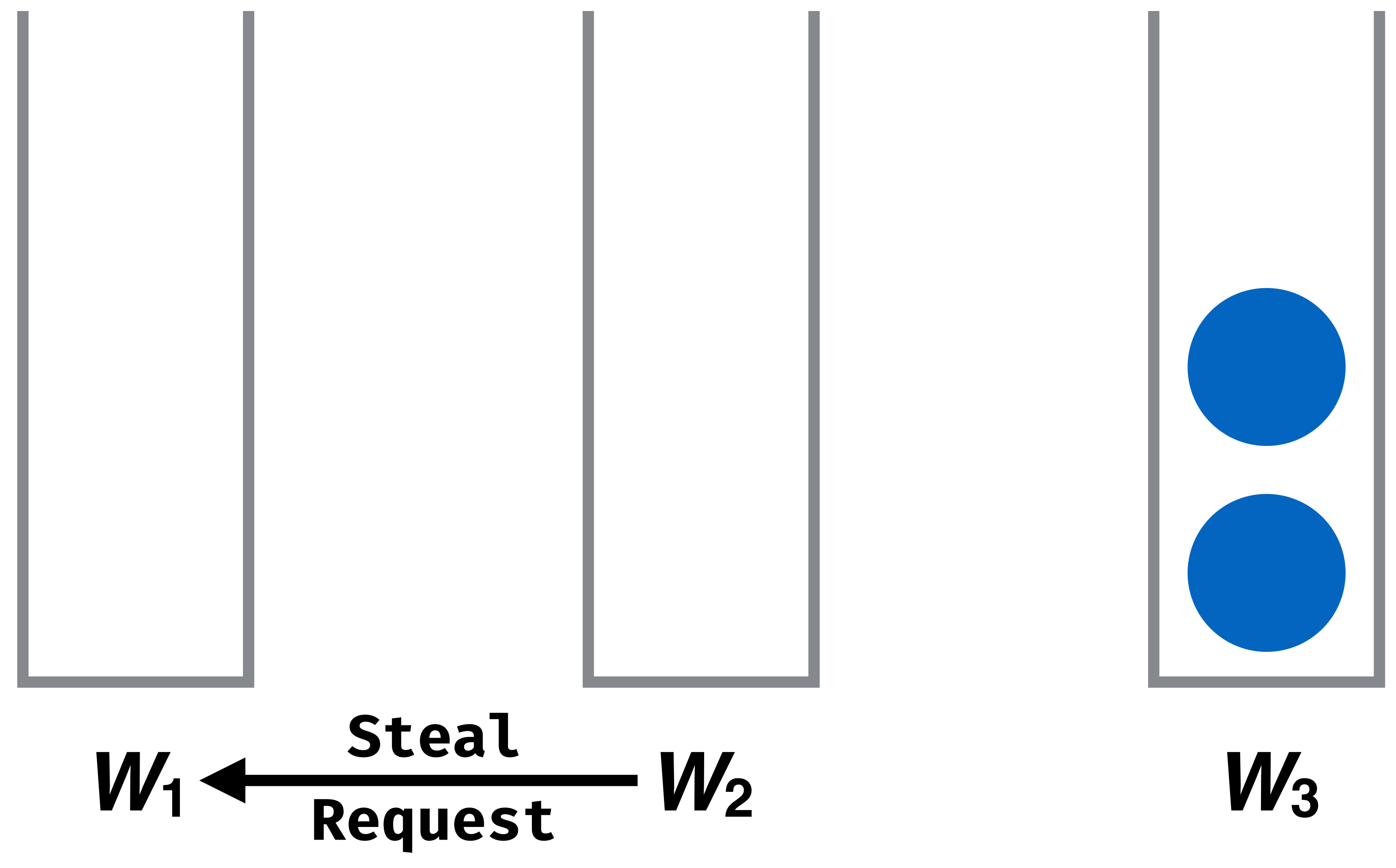
```
struct steal_request {
    Channel *chan;
    int thief;
    // ...
};
```

Steal Requests

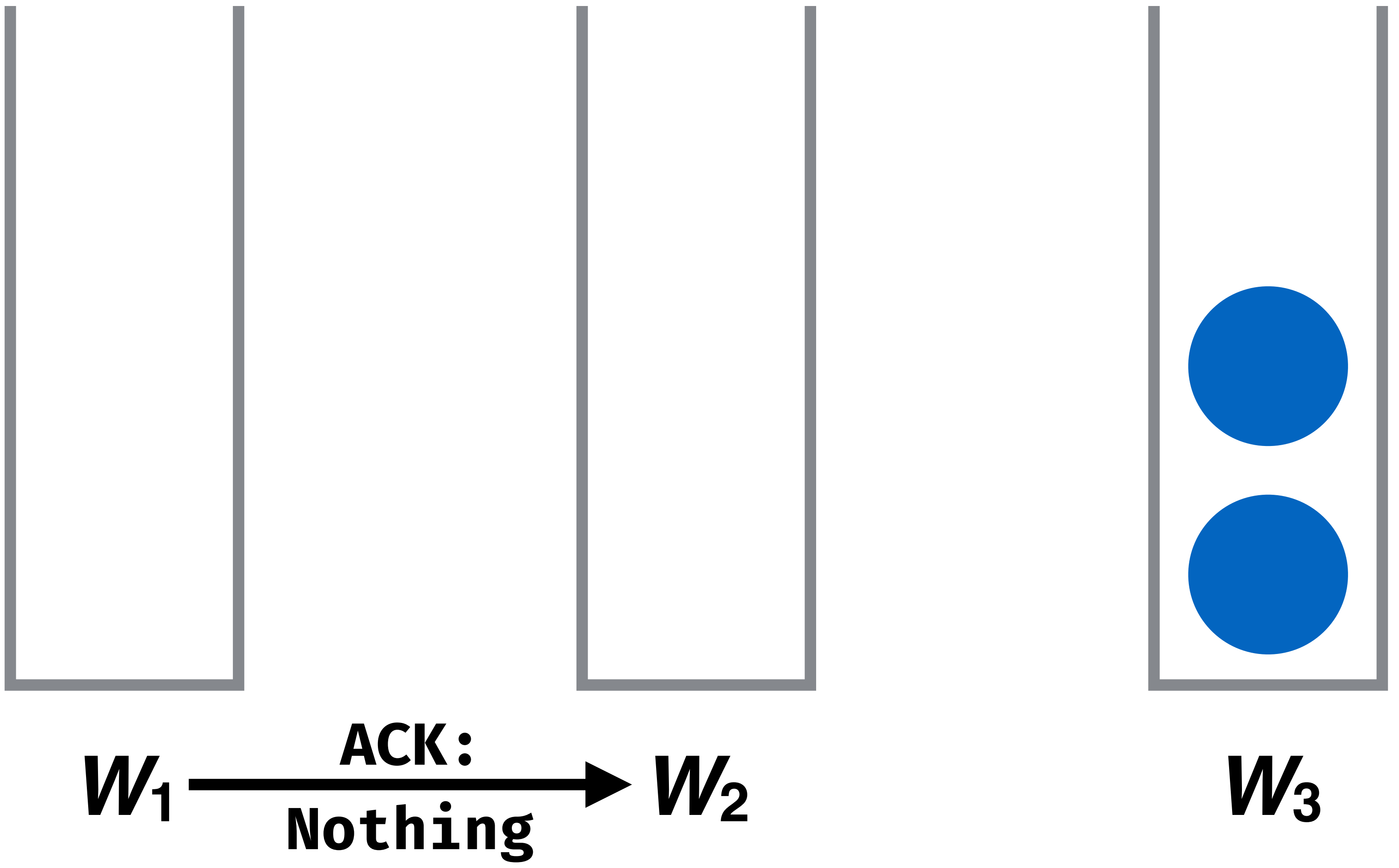
```
struct steal_request req = {
    .chan = SPSC,
    .thief = ID,
    // ...
};
int i = select_victim();
channel_send(MPSC[i], &req, sizeof(req));
```

Two channels per worker

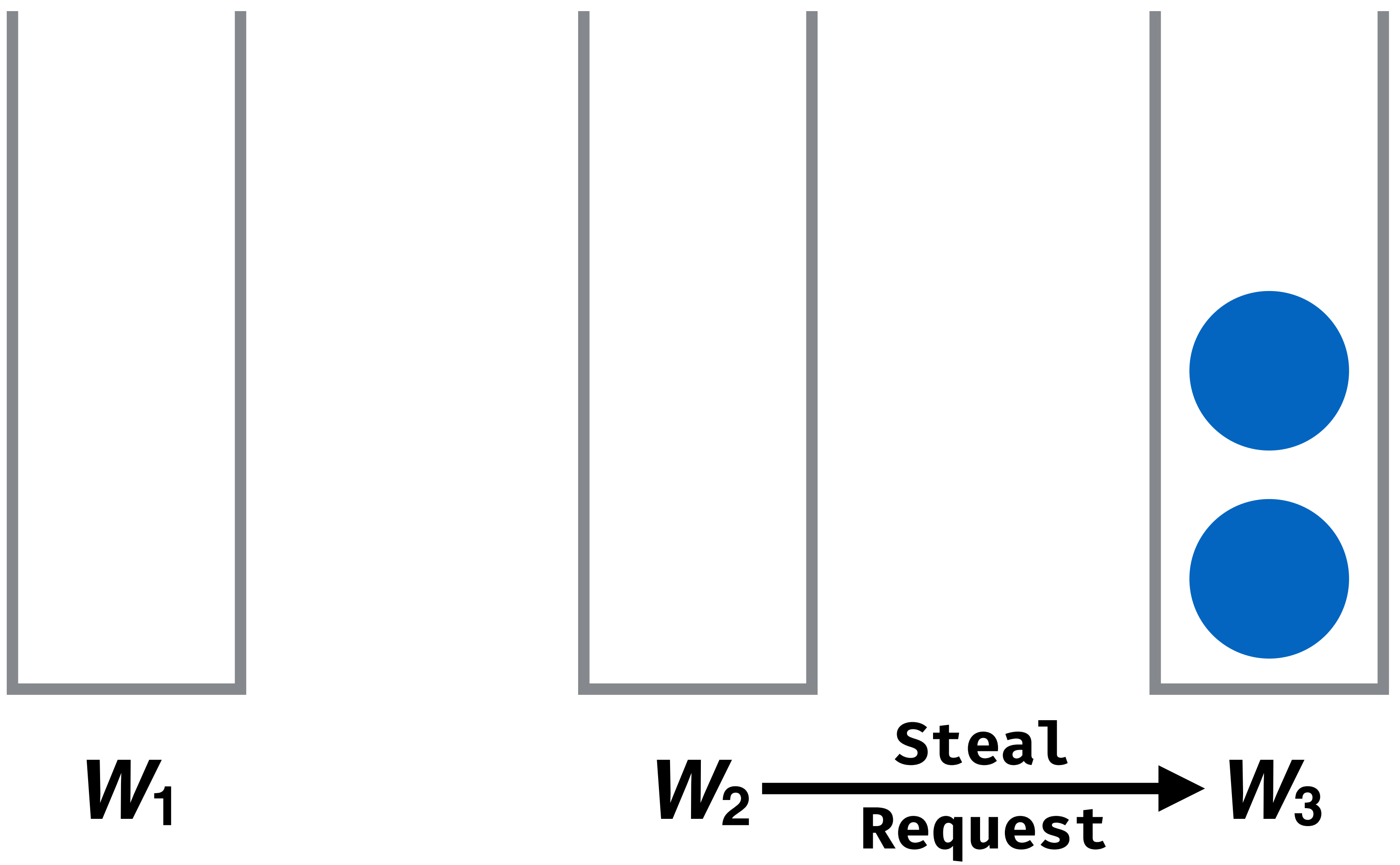
Steal Requests



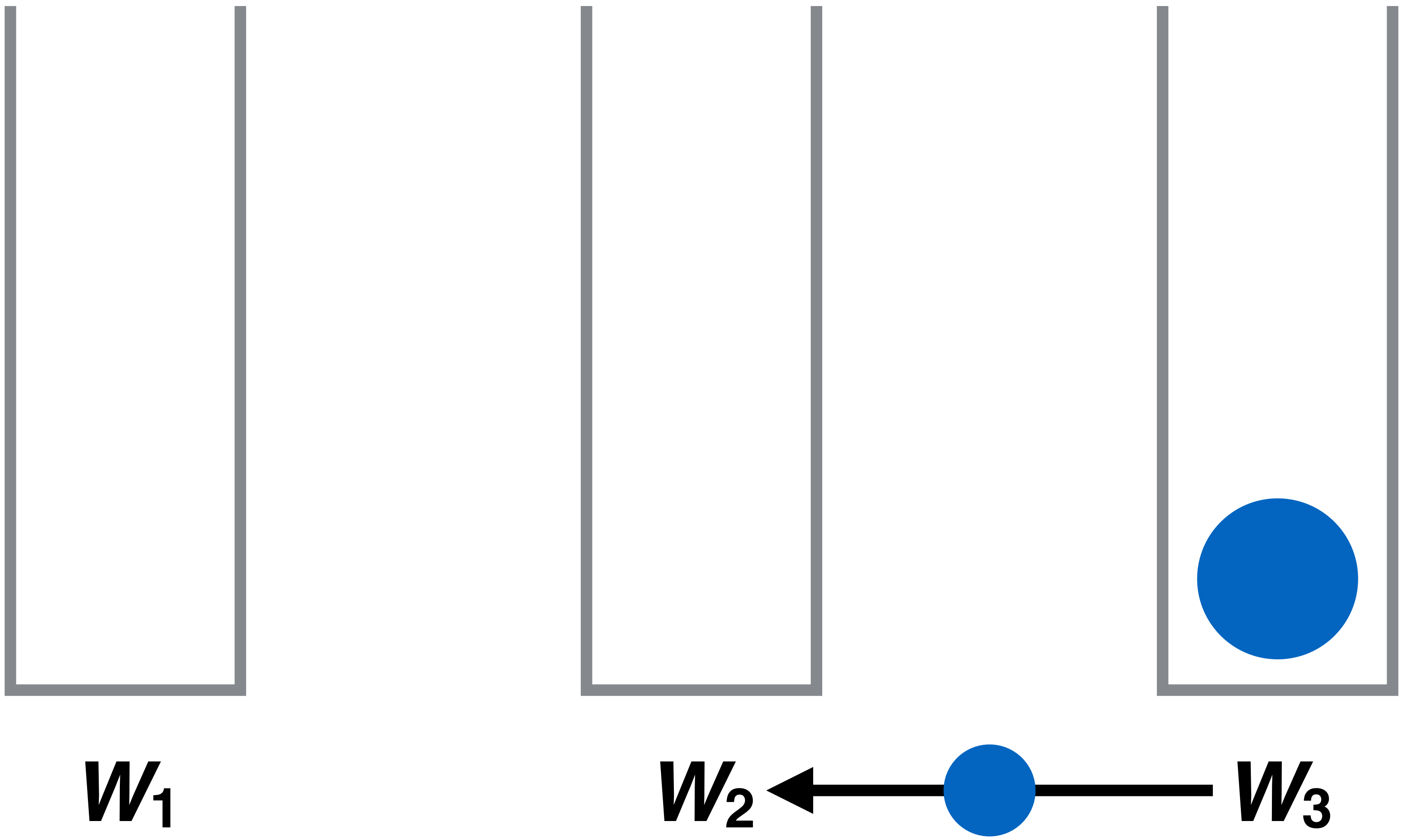
Steal Requests



Steal Requests

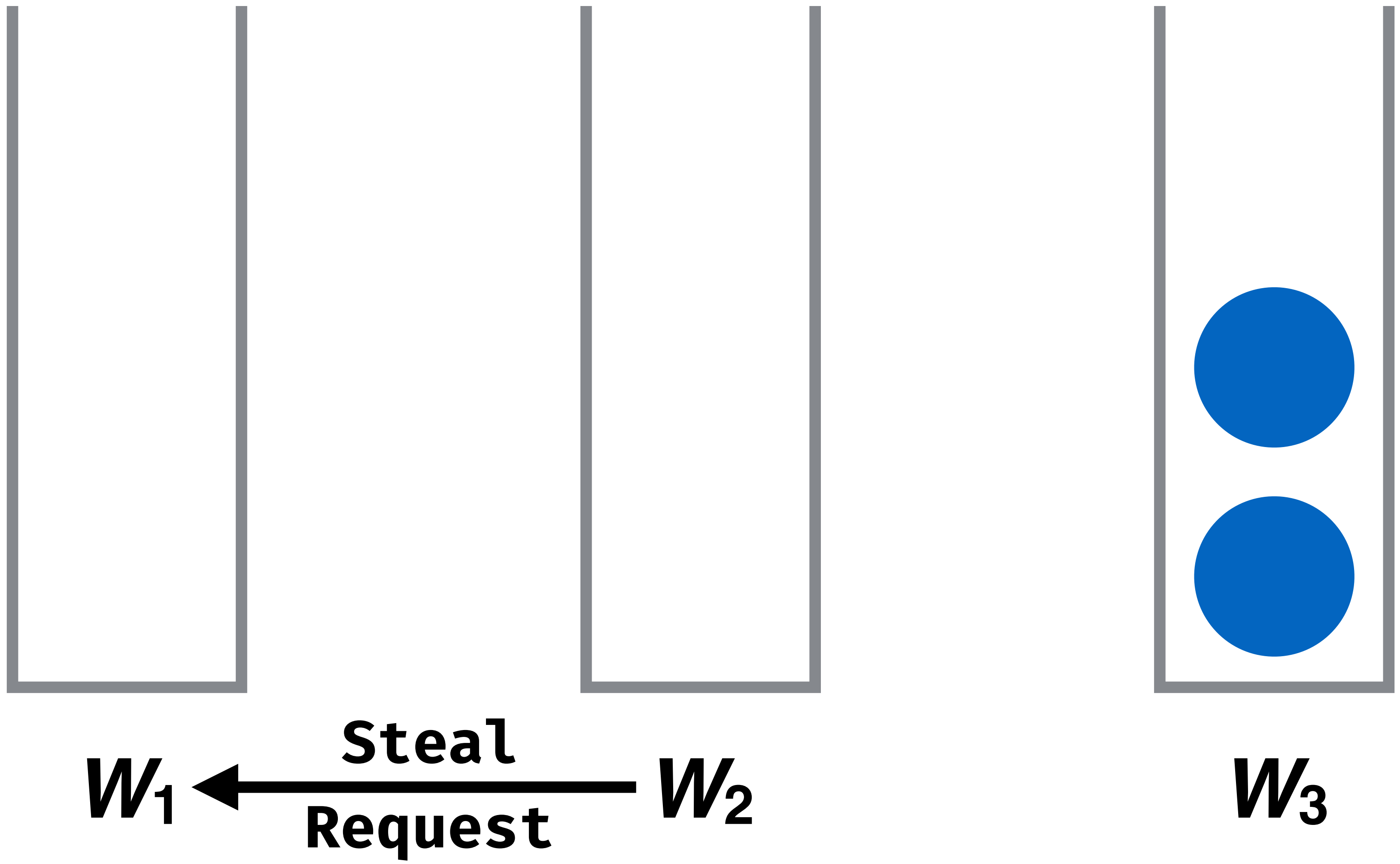


Steal Requests



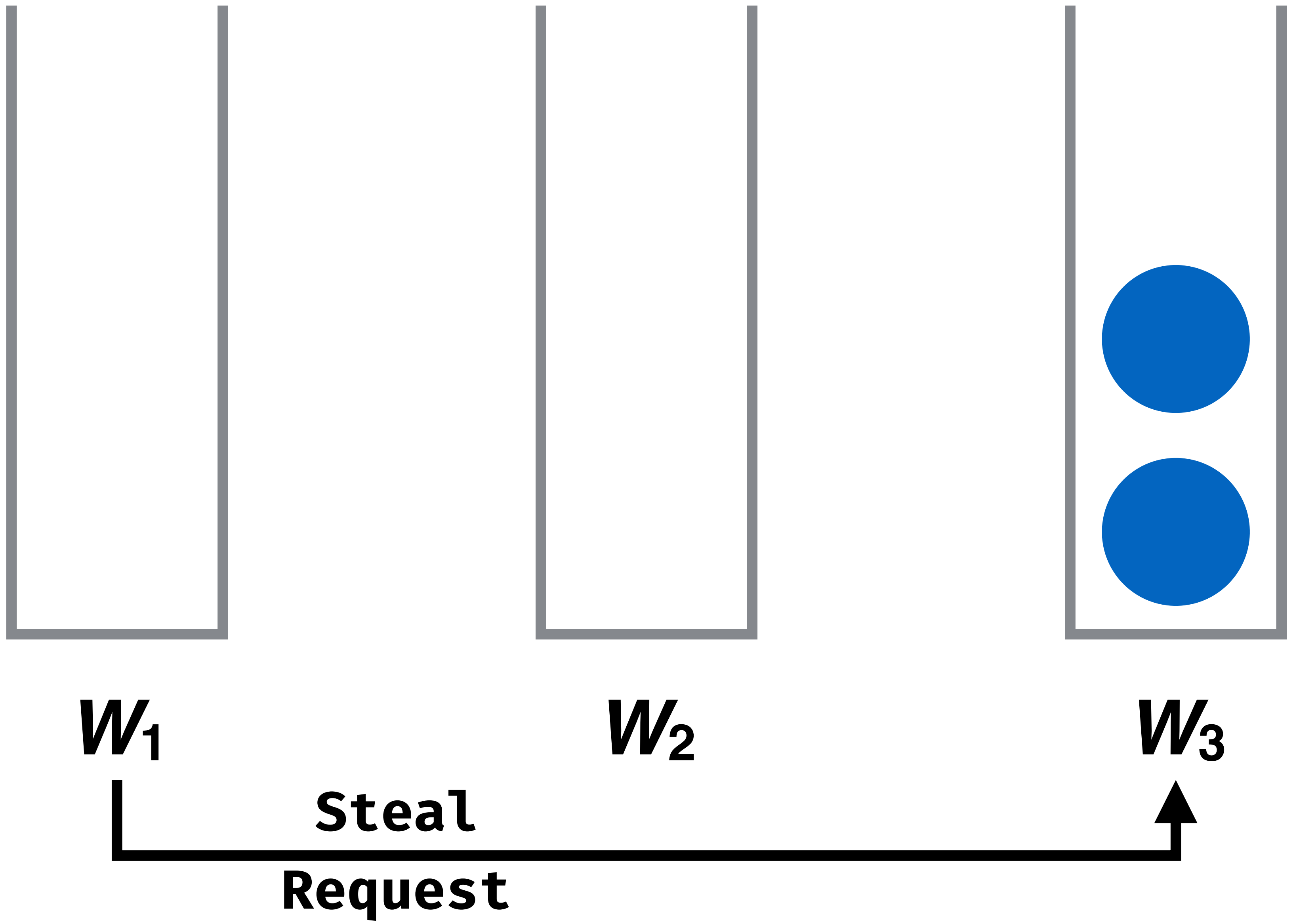
Steal Requests

Idea: Eliminate ACKs by forwarding steal requests



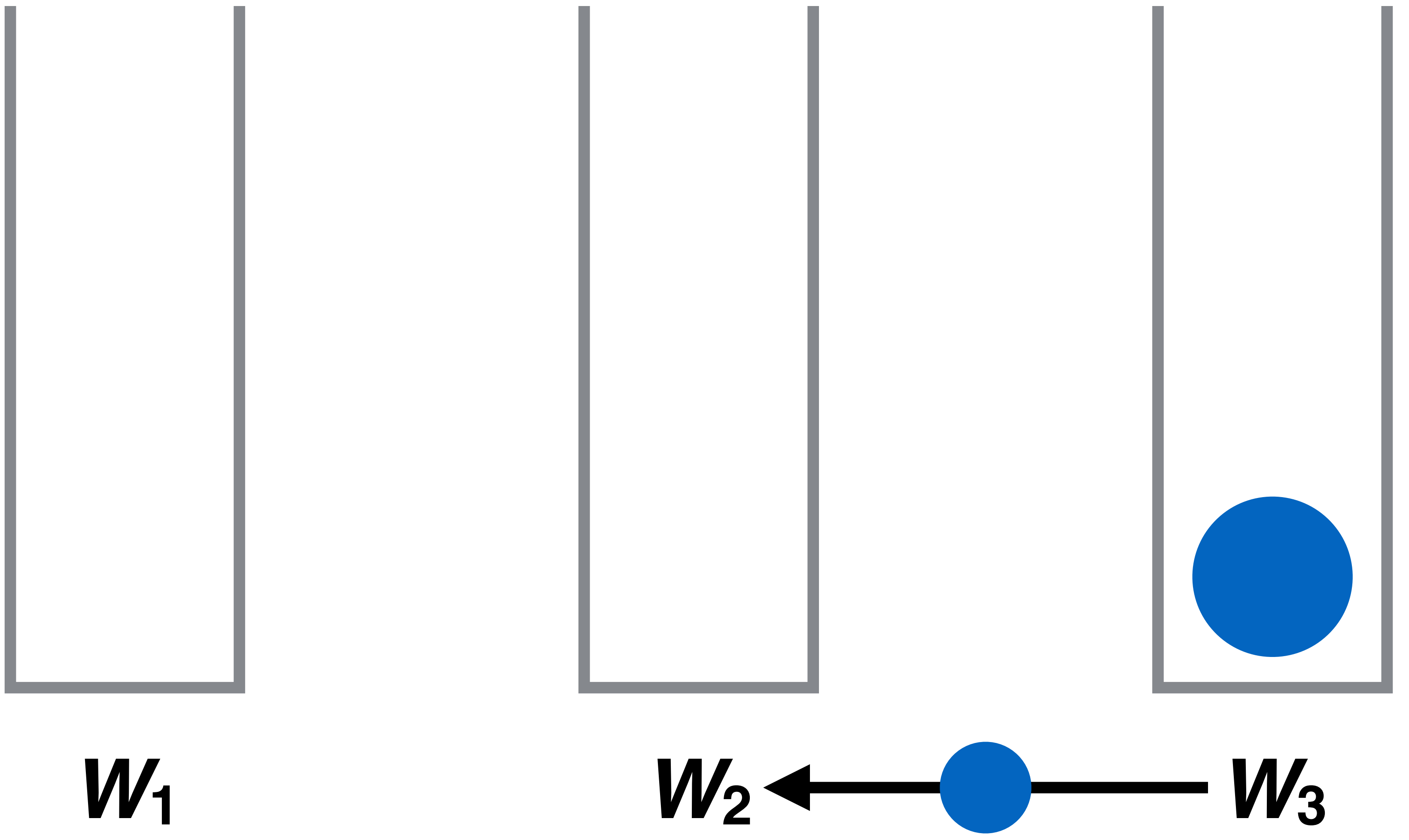
Steal Requests

Idea: Eliminate ACKs by forwarding steal requests



Steal Requests

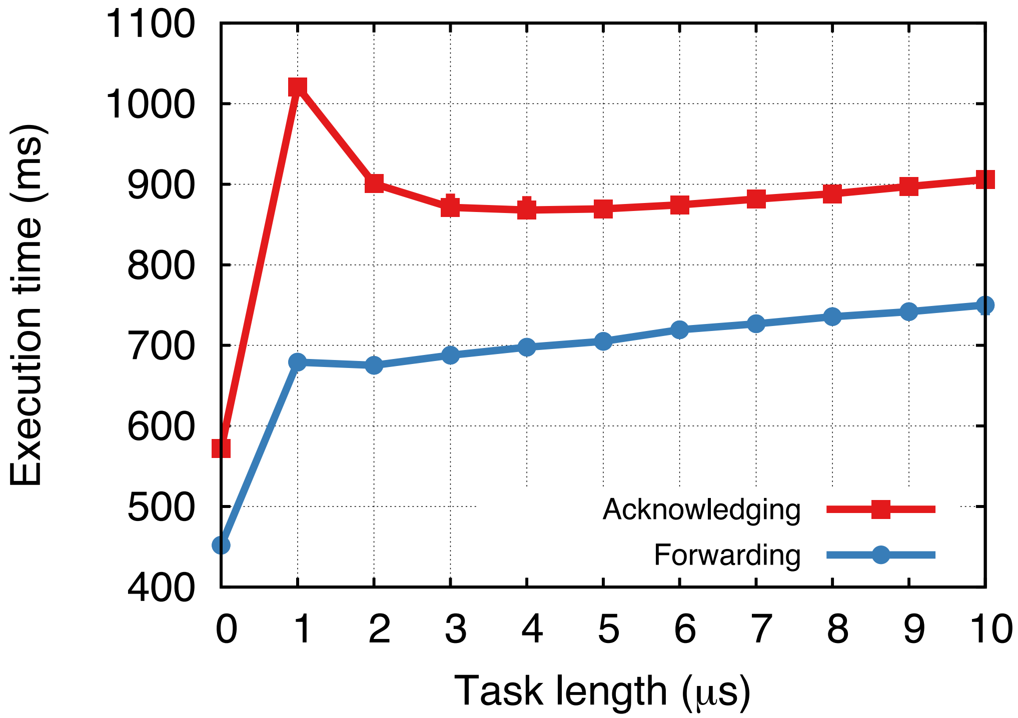
Idea: Eliminate ACKs by forwarding steal requests

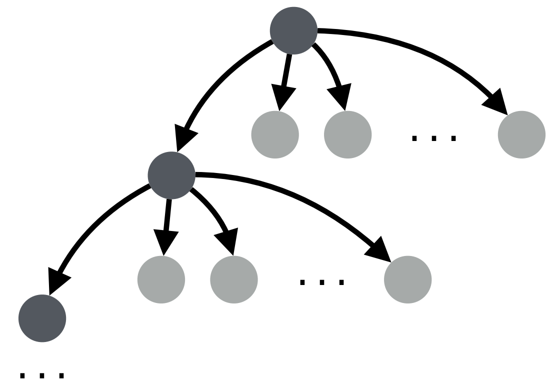


Steal Requests

Forwarding

- reduces number of messages
- facilitates asynchronous stealing
- improves performance

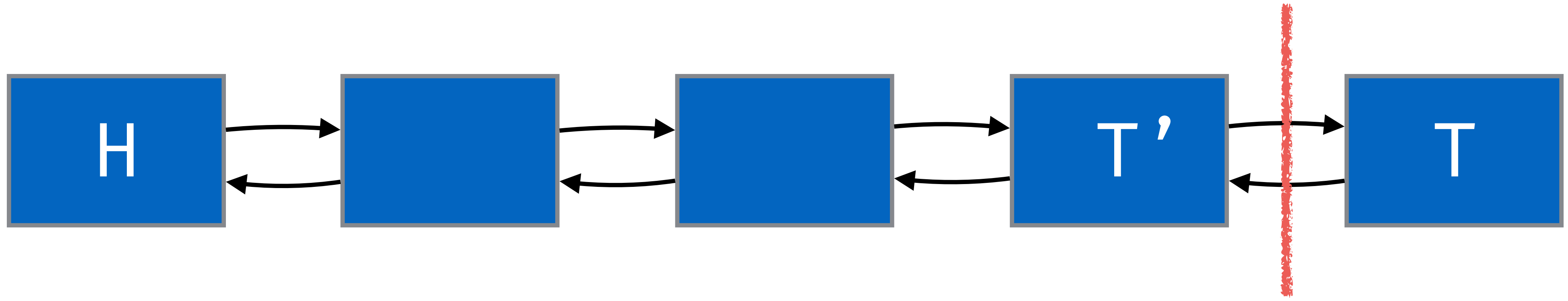




Benchmark: BPC with $d = 10^5$, $n = 9$, and t as shown (x-axis)

Stealing Tasks

One task



Send T or &T to thief

Stealing Tasks

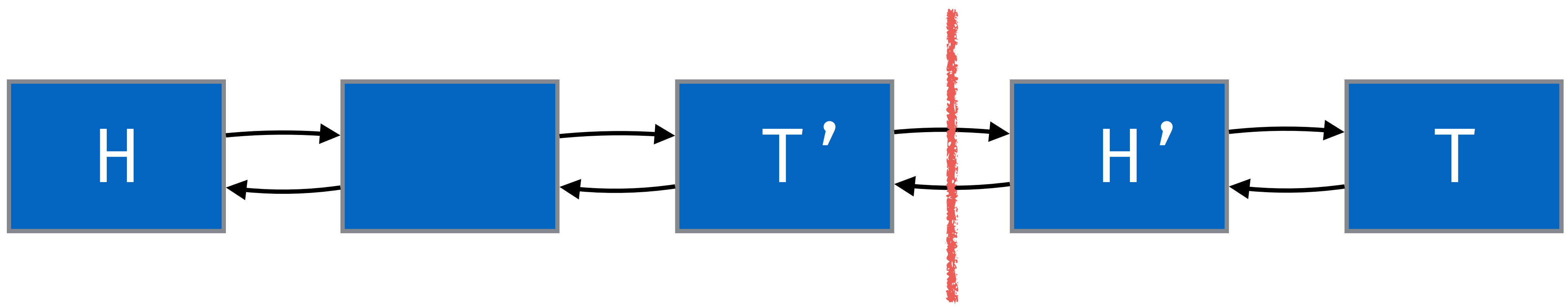
One task

Send T or &T to thief
Share memory by communicating*

*A. Gerrand, https://blog.golang.org/share-memory-by-communicating, 13 July 2010

Stealing Tasks

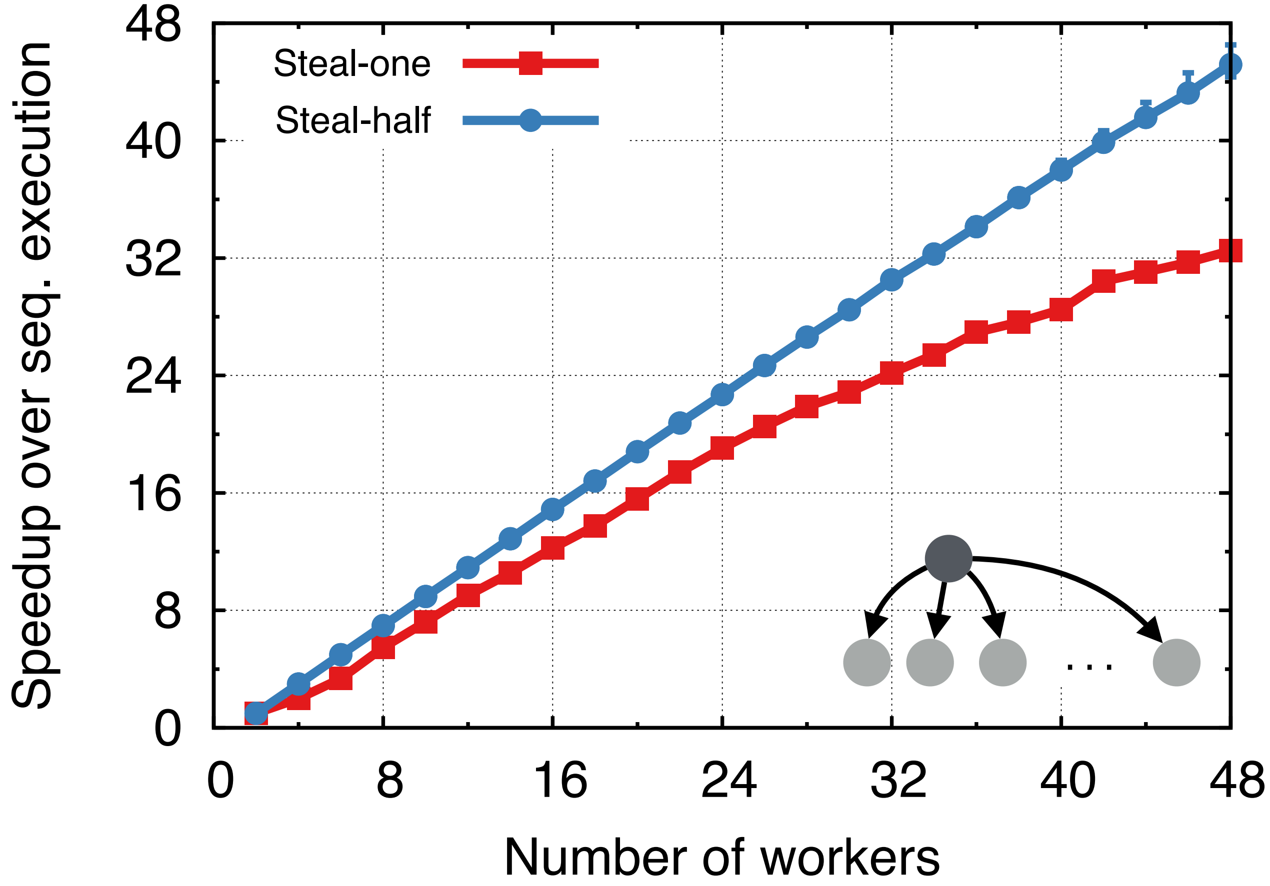
$\lfloor n/2 \rfloor$ tasks



Send `&H'` to thief

Stealing Tasks

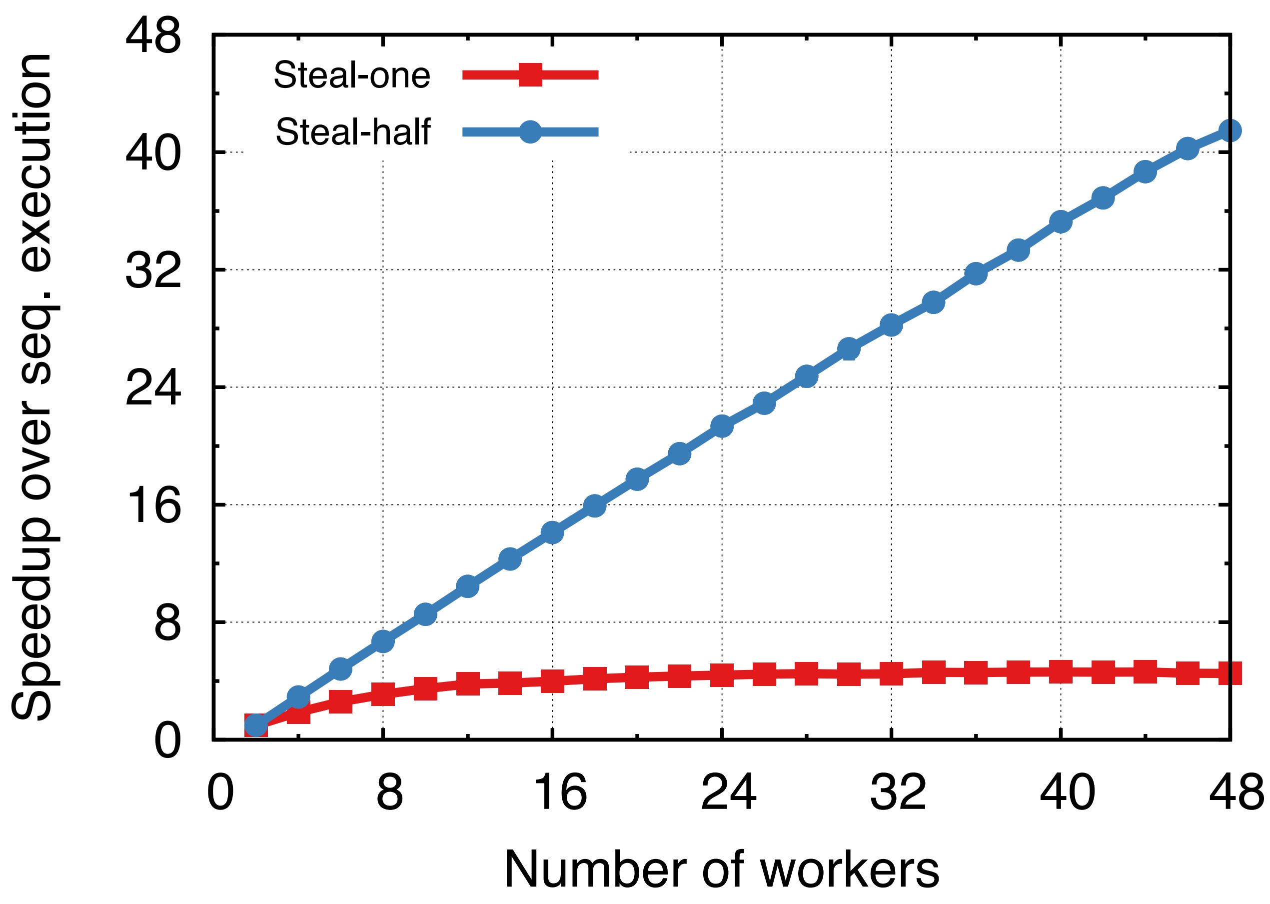
Task length 100 μs



Benchmark: SPC with n = 10^6 and t = 100 μs

Stealing Tasks

Task length 10 μs



Benchmark: SPC with $n = 10^6$ and t = 10 μs

Task Synchronization

Task Barrier

```c
#include <stdio.h>
#include "tasking.h"

ASYNC_VOID_DECL (
    puts, const char *s, s
);

int main(void)
{
    TASKING_INIT();

    ASYNC(puts, "Order");
    ASYNC(puts, "Undefined");

    TASKING_BARRIER();

    ASYNC(puts, "Last");

    TASKING_EXIT();
    return 0;
}
```

```c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            puts("Order");
            #pragma omp task
            puts("Undefined");
        }
        #pragma omp barrier
        #pragma omp master
        {
            #pragma omp task
            puts("Last");
        }
    }
    return 0;
}
```

Termination Detection with Steal Requests

Problem: Detect when all workers are idle without resorting to implicit communication

Idea: “Color” steal requests

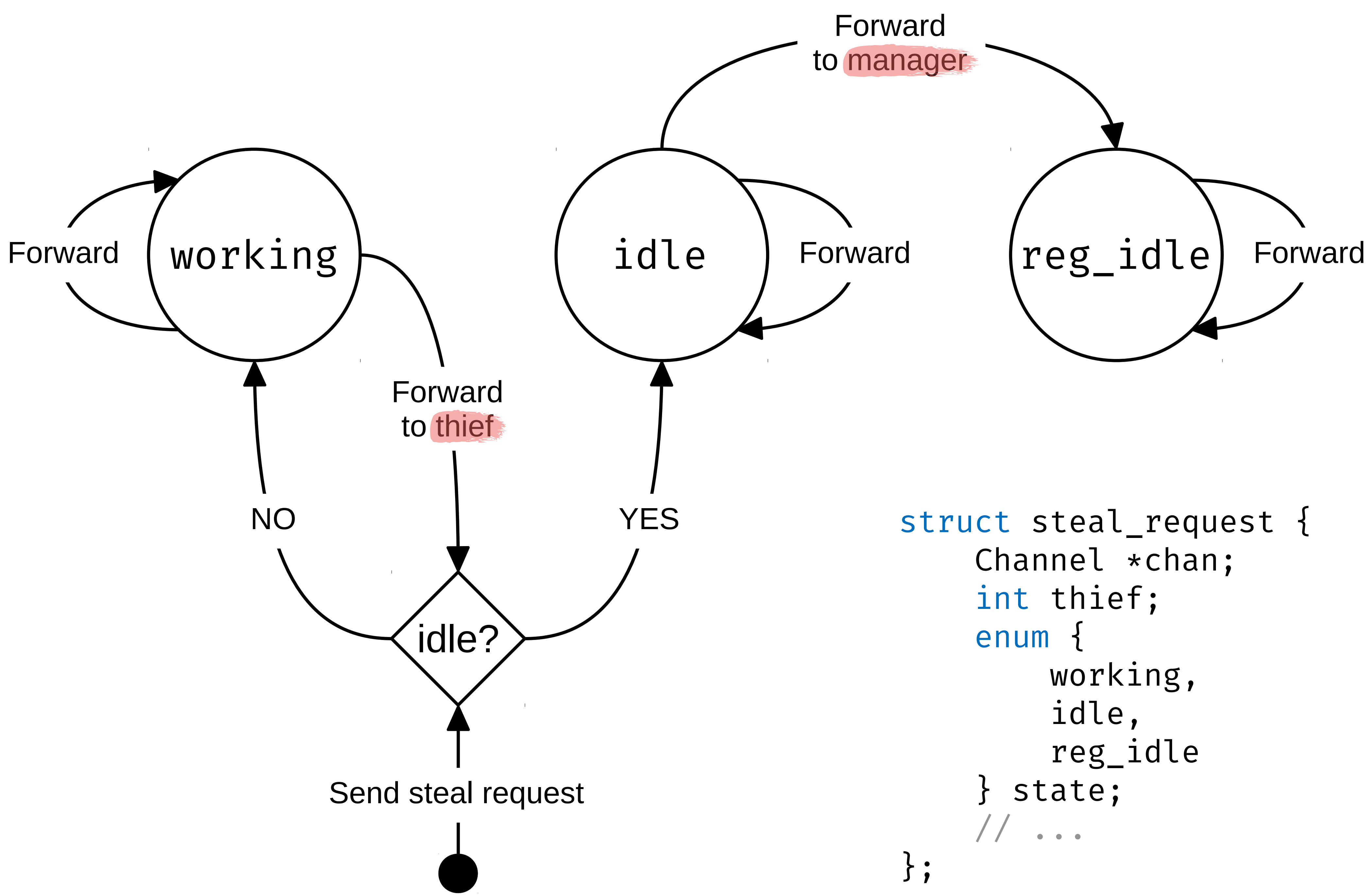
→ Avoids separate control messages

→ Termination follows from forwarding

Extended Steal Requests

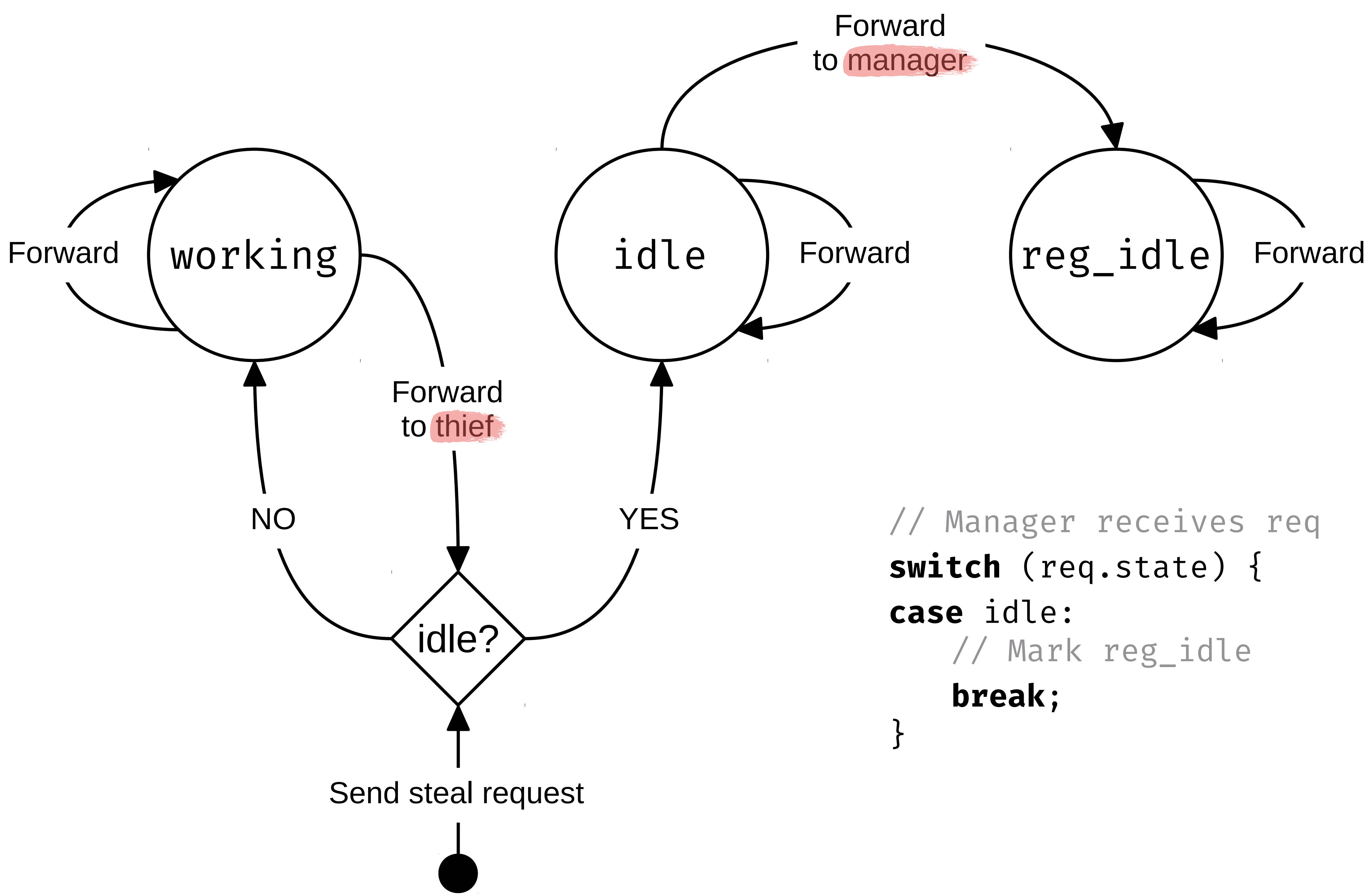
```
struct steal_request {
    Channel *chan;
    int thief;
    enum {
        working,
        idle,
        reg_idle
    } state;
    // ...
};
```

Extended Steal Requests




```
struct steal_request {
    Channel *chan;
    int thief;
    enum {
        working,
        idle,
        reg_idle
    } state;
    // ...
};
```

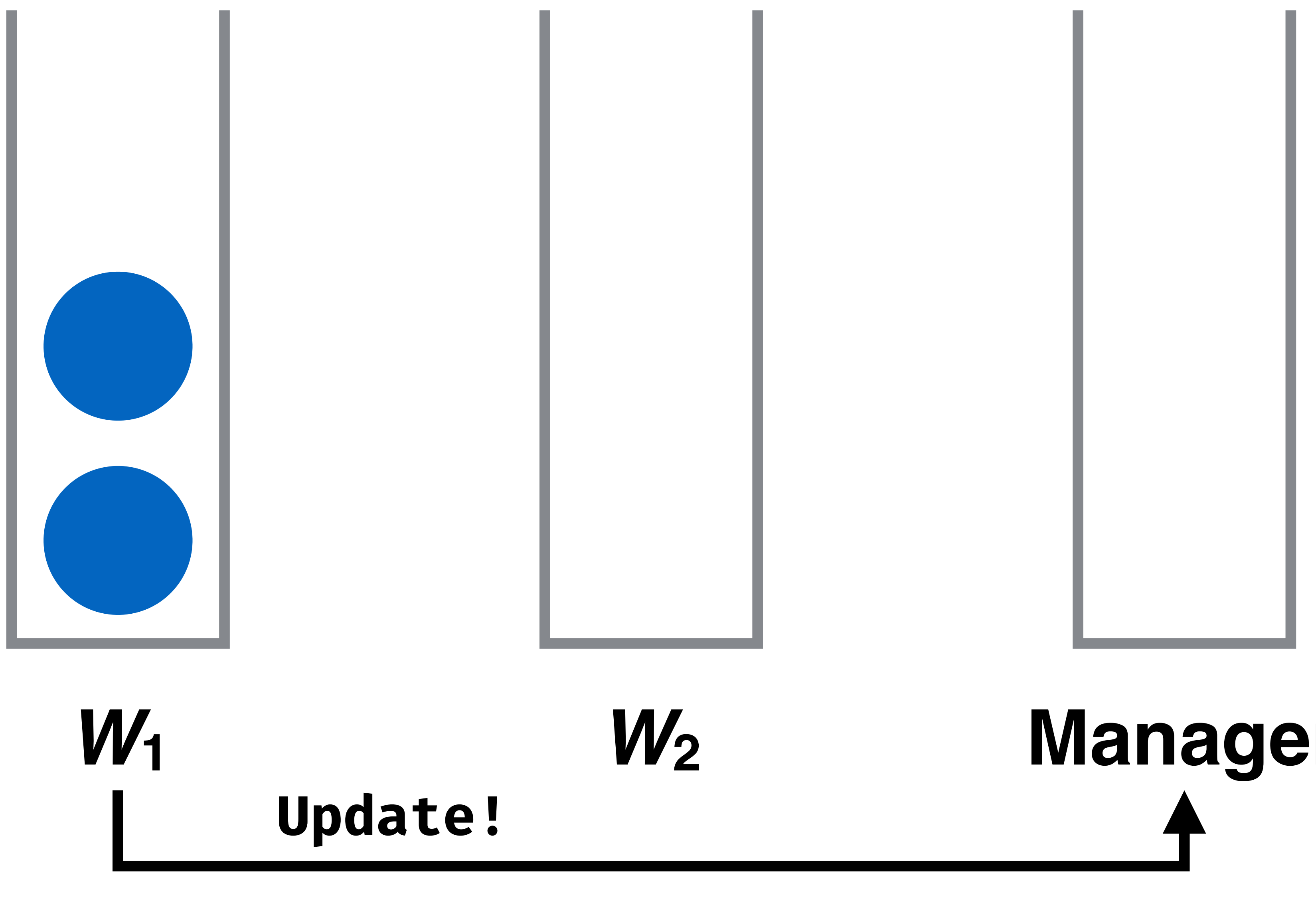
Extended Steal Requests



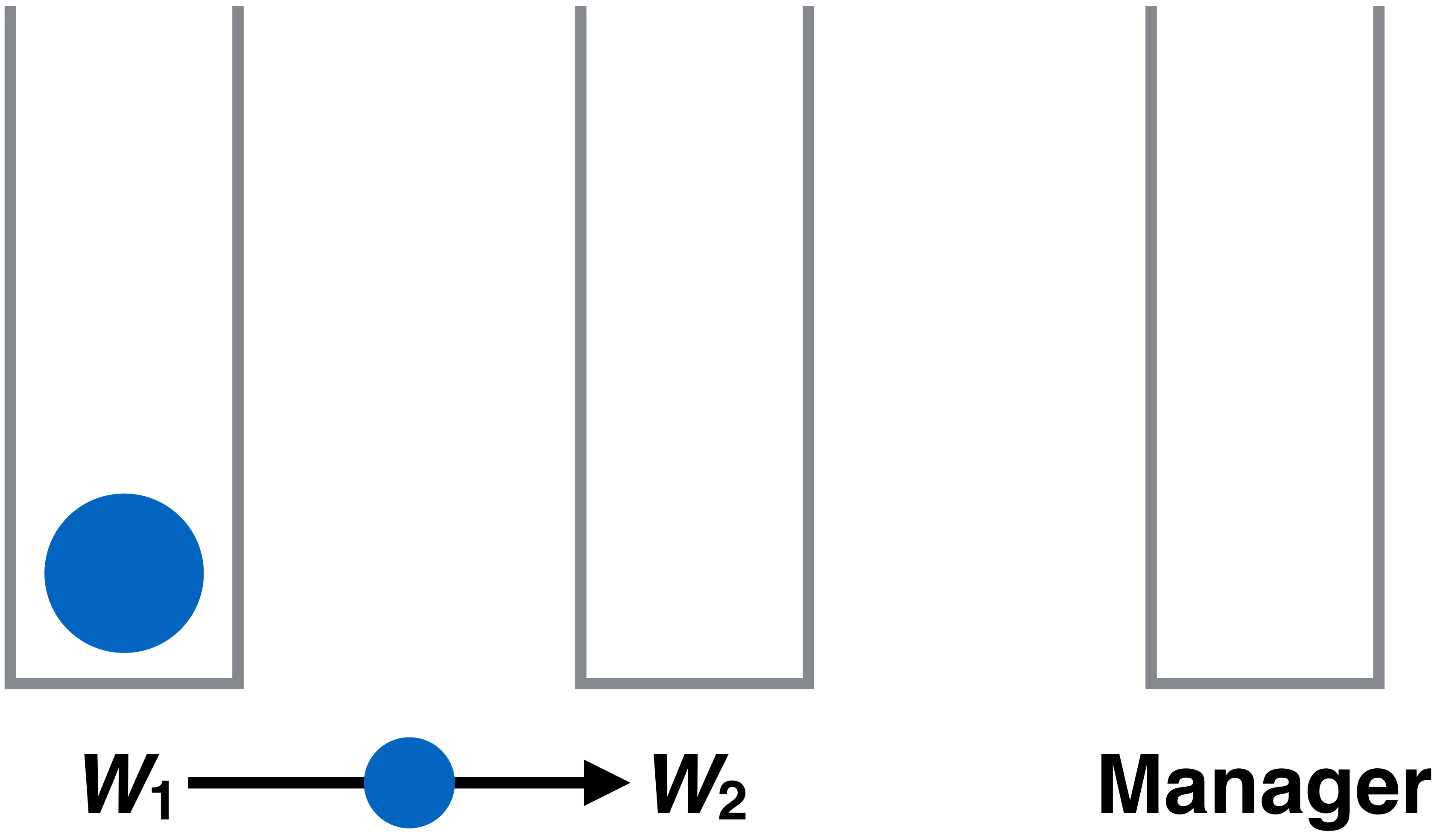

```
// Manager receives req
switch (req.state) {
case idle:
    // Mark reg_idle
    break;
}
```

Notifying the Manager

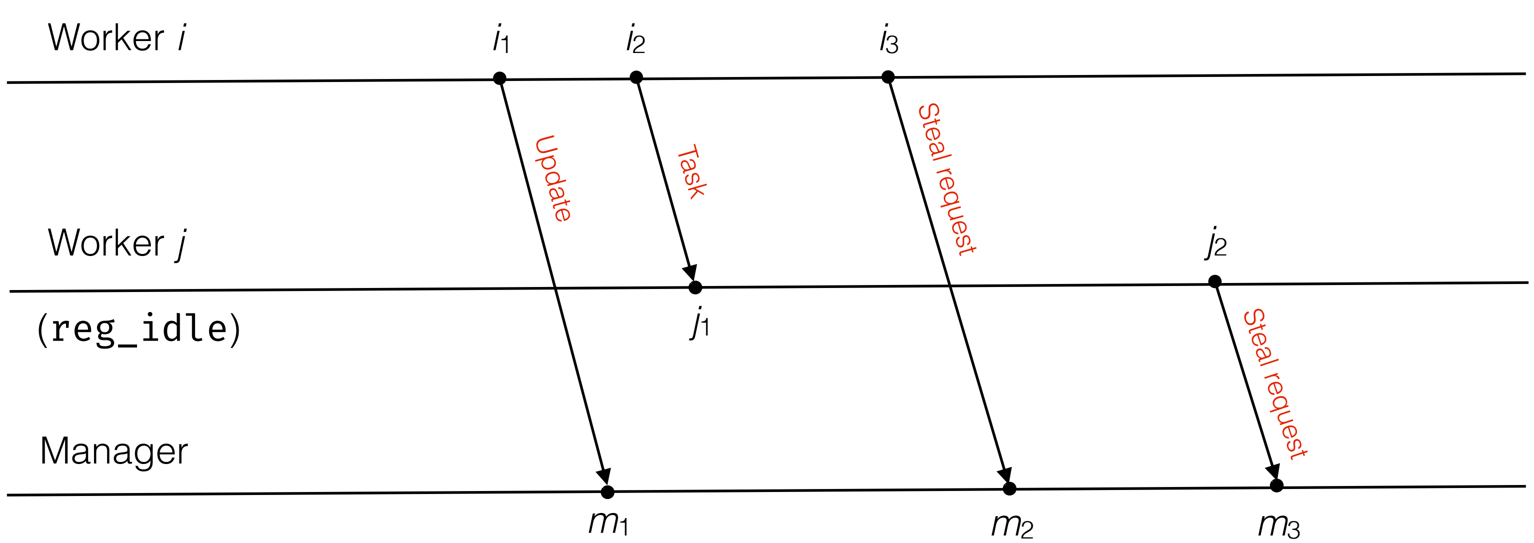
Notifying the Manager



Notifying the Manager



Updates




```c
struct steal_request {
    Channel *chan;
    int thief;
    enum {
        working,
        idle,
        reg_idle,
        update
    } state;
    // ...
};
```

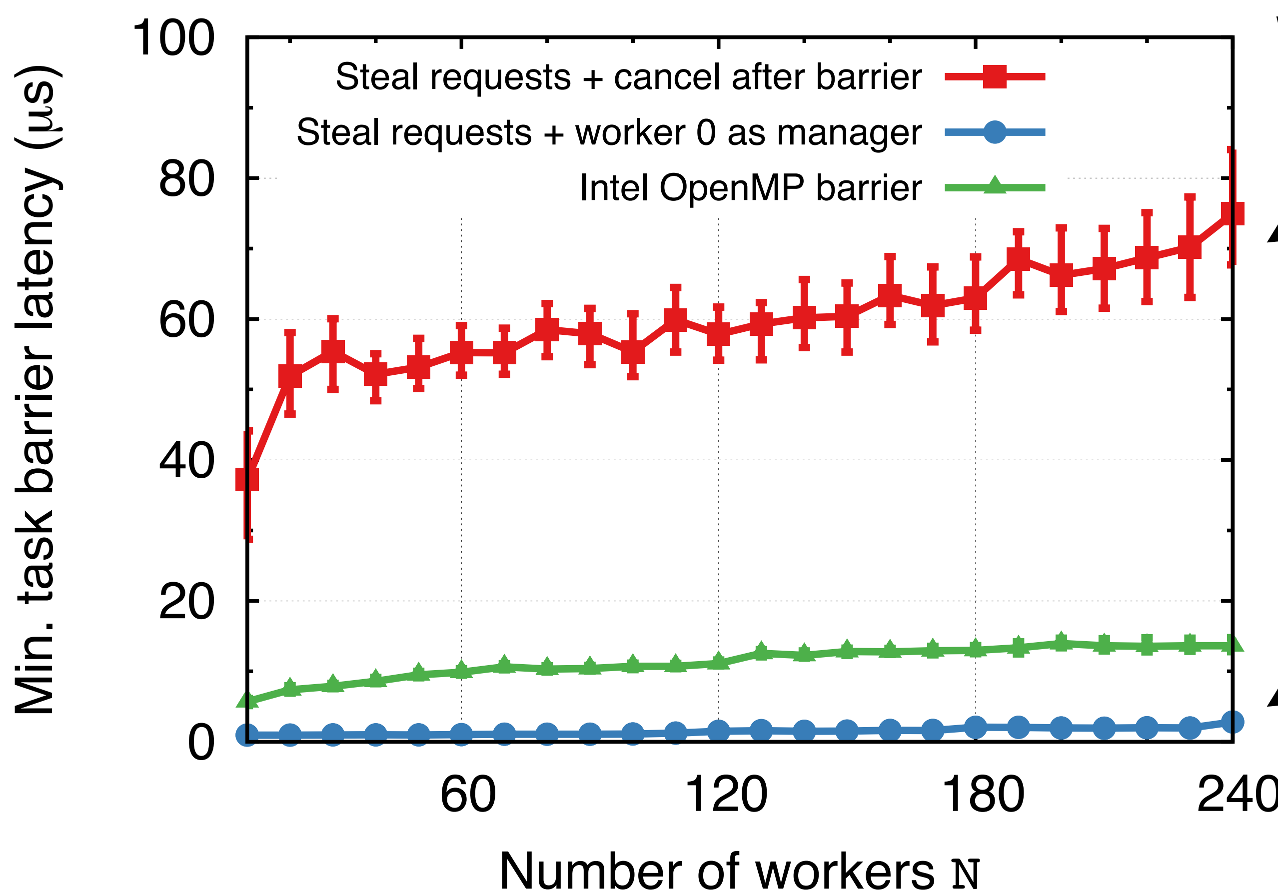
```c
// Manager receives req
switch (req.state) {
case update:
    // ...
    break;
case idle:
    // ...
    break;
}
```

Task Barrier

Impact of explicit communication

Task Barrier

Impact of explicit communication

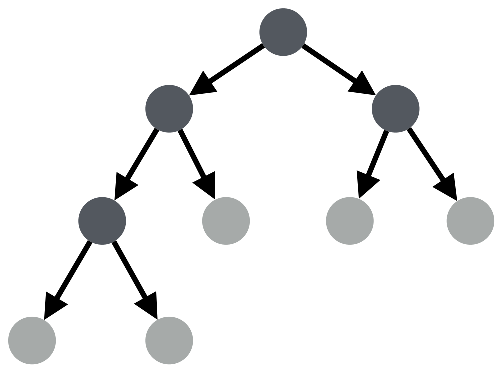


Worker 0 cancels a random worker's steal request

No additional communication required

Channel-based Futures

```
int x = spawn f(n-1);
int y = f(n-2);
sync;
```



```
future fx = FUTURE(f, n-1);
int y = f(n-2);
int x = AWAIT(fx, int);
```

Channel-based Futures

```
future fx = FUTURE(f, n-1);
int y = f(n-2);
int x = AWAIT(fx, int);
```

1. Allocates a one-element SPSC channel
2. Creates a task, passing the channel
3. Returns a handle to the channel (`future`)

Channel-based Futures

```
future fx = FUTURE(f, n-1);
int y = f(n-2);
int x = AWAIT(fx, int);
```

1. Waits for the task to send its result
2. Receives and returns the result
3. Frees or recycles the channel

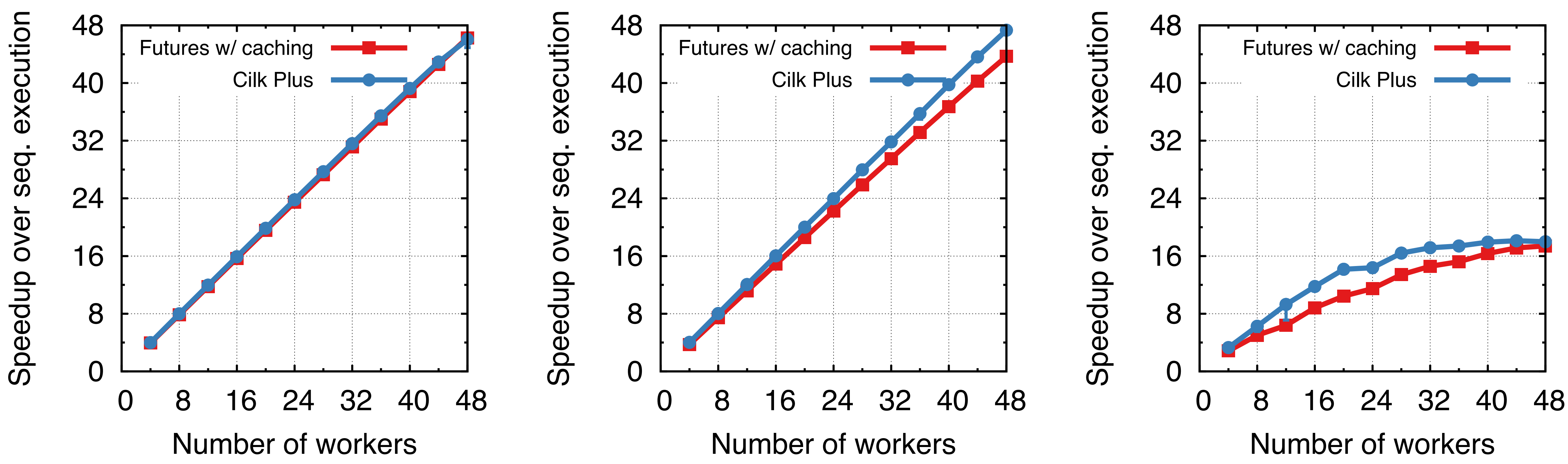
Channel-based Futures

```
future fx = FUTURE(f, n-1);
int y = f(n-2);
int x = AWAIT(fx, int);
```

Tries to schedule other work to avoid idling

Performance

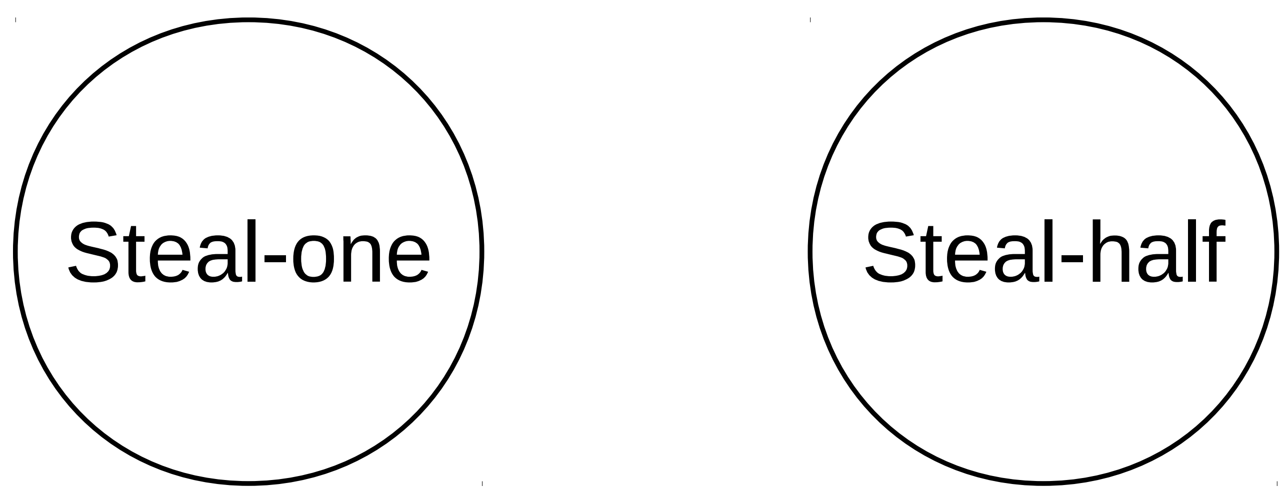
Fork/join parallelism



Benchmarks from left to right:
Tree recursion with n = 34 and t = 1 μs | 14 Queens Problem | Cilksort of 10^8 integers

Adaptive Strategies

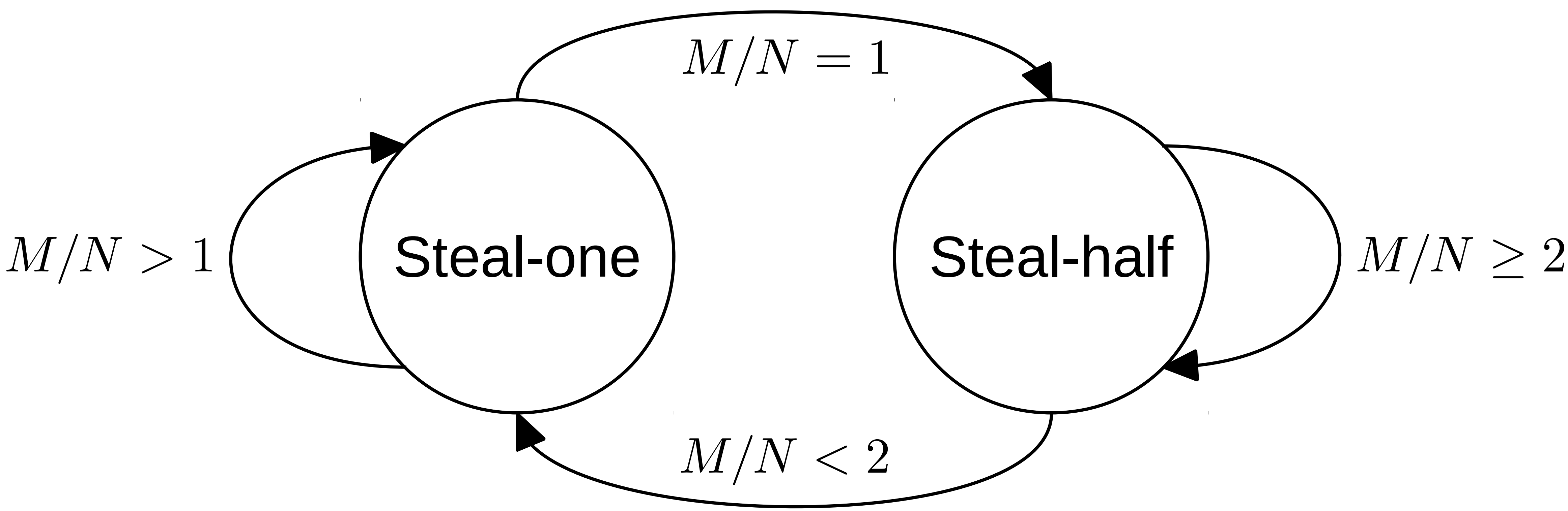
Adaptive Stealing



Adaptive Stealing

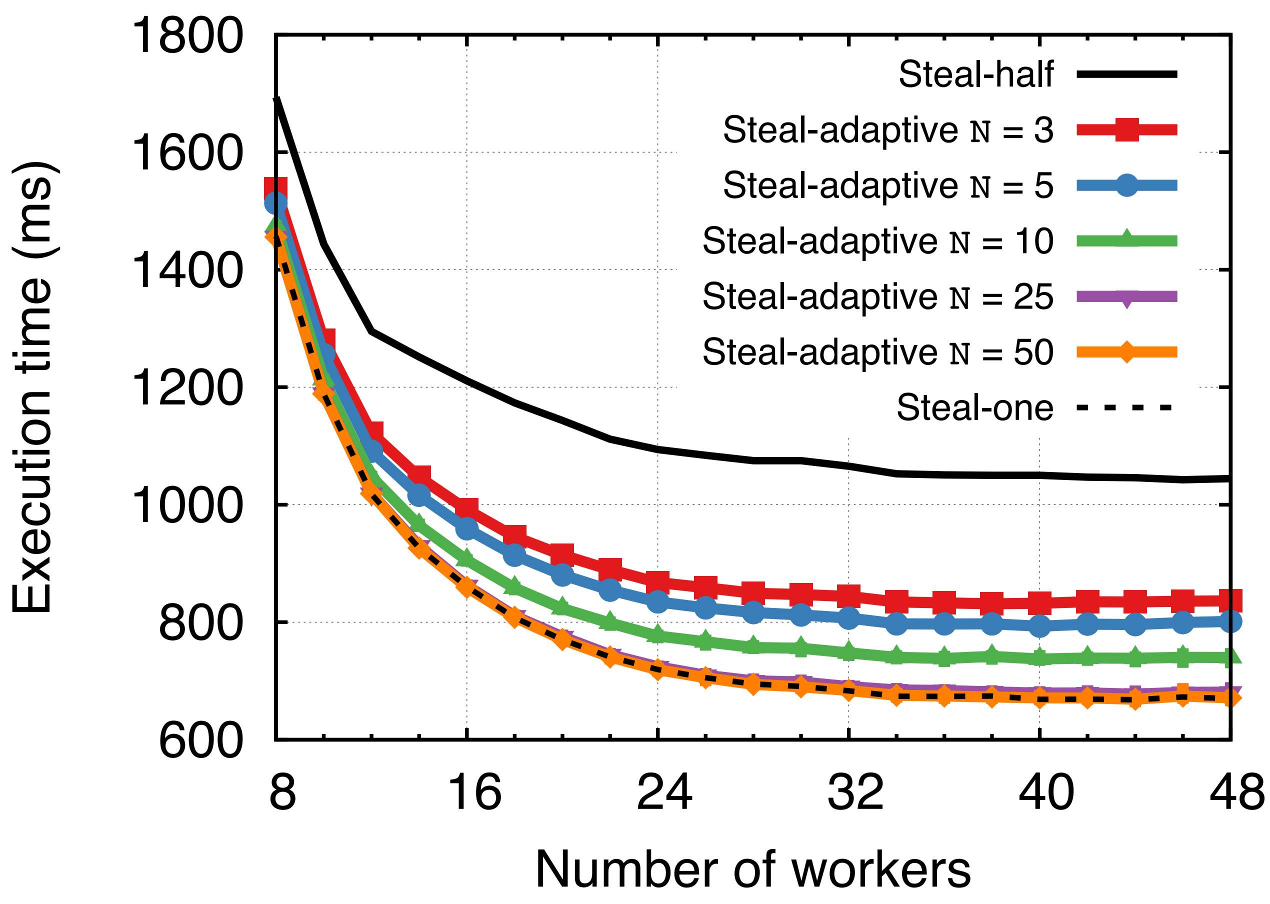
Idea: Reevaluate strategy after N steals

Count how many tasks have been executed: M



Adaptive Stealing

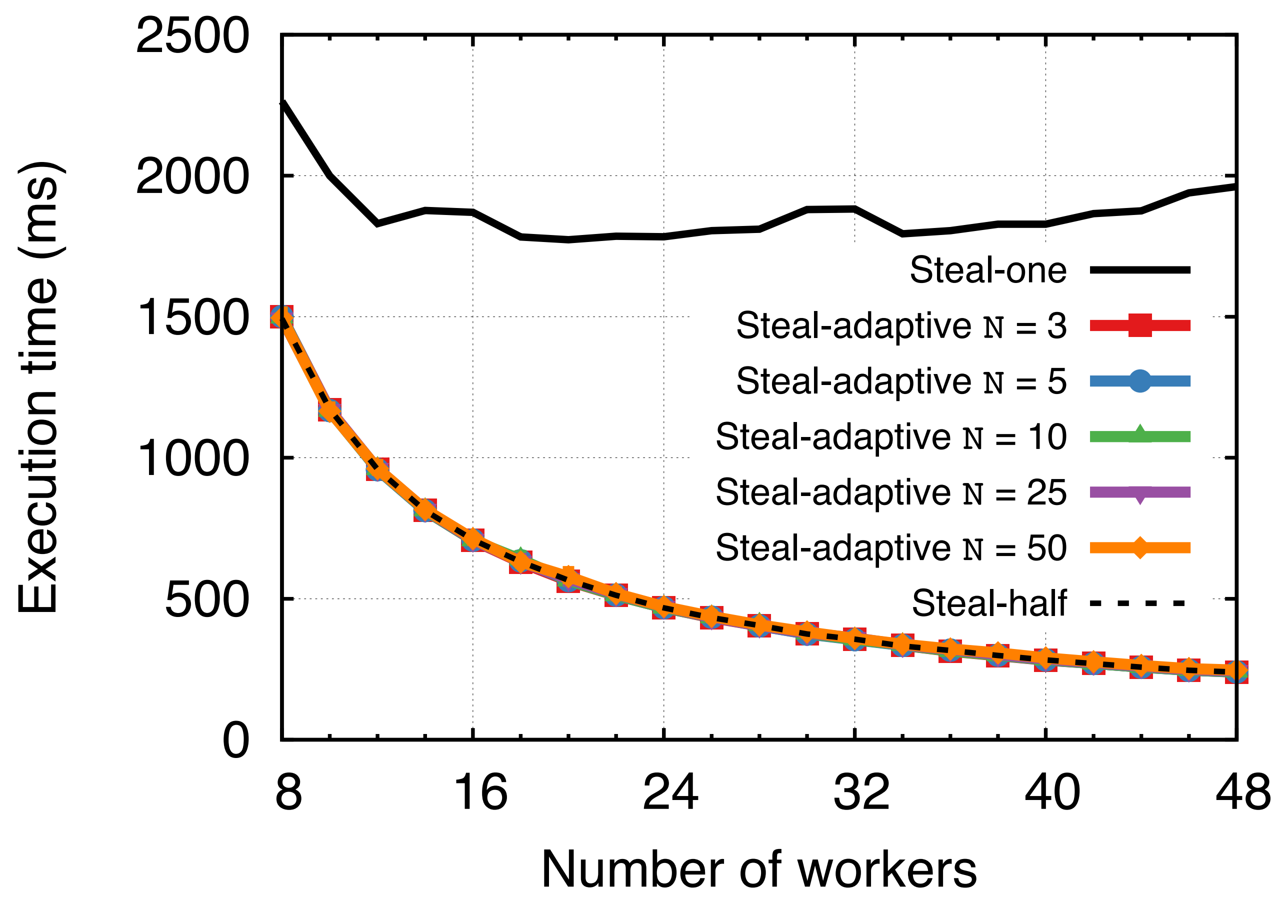
Task length 10 μs



Benchmark: BPC with $d = 10^5$, $n = 9$, and $t = 10$ μs

Adaptive Stealing

Task length 10 μs



Benchmark: BPC with d = 1, n = 999,999, and t = 10 μs

Very Fine-grained Tasks

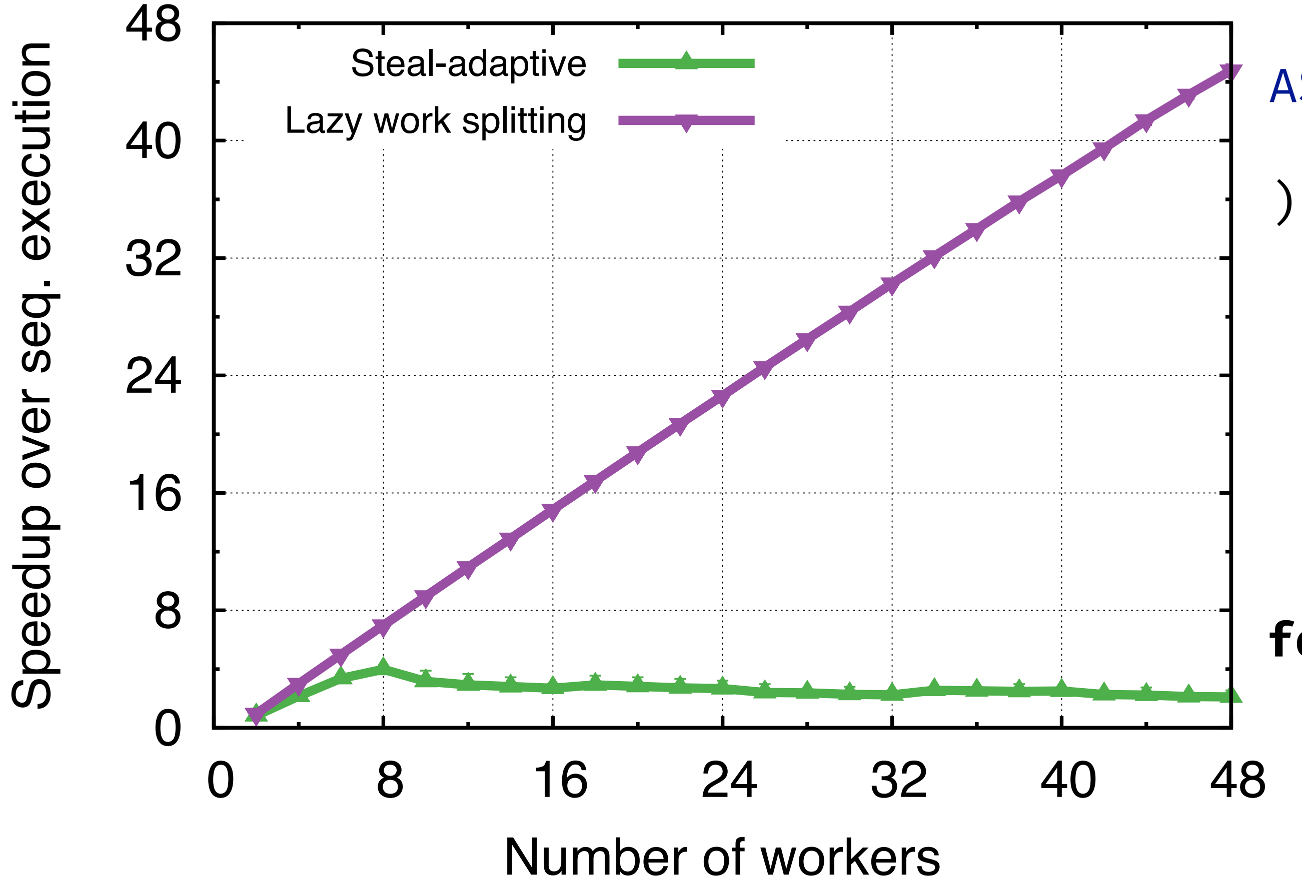
Task length 1 μs


```
for (i = 0; i < N; i++)
    ASYNC(f, i, ...);
```

Benchmark: SPC with $n = 10^6$ and $t = 1$ μs

Splittable Tasks

Task length 1 μs




```
ASYNC_FOR (
    f, 0, N, ...
);
```

```
for (i = 0; i < N; i++)
    ASYNC(f, i, ...);
```

Benchmark: SPC with $n = 10^6$ and $t = 1$ μs

Lazy Binary Splitting

Assumes concurrent work-stealing deques:

Worker splits when local deque is empty, otherwise executes tasks sequentially

→ Splitting is *lazy* as opposed to *eager*

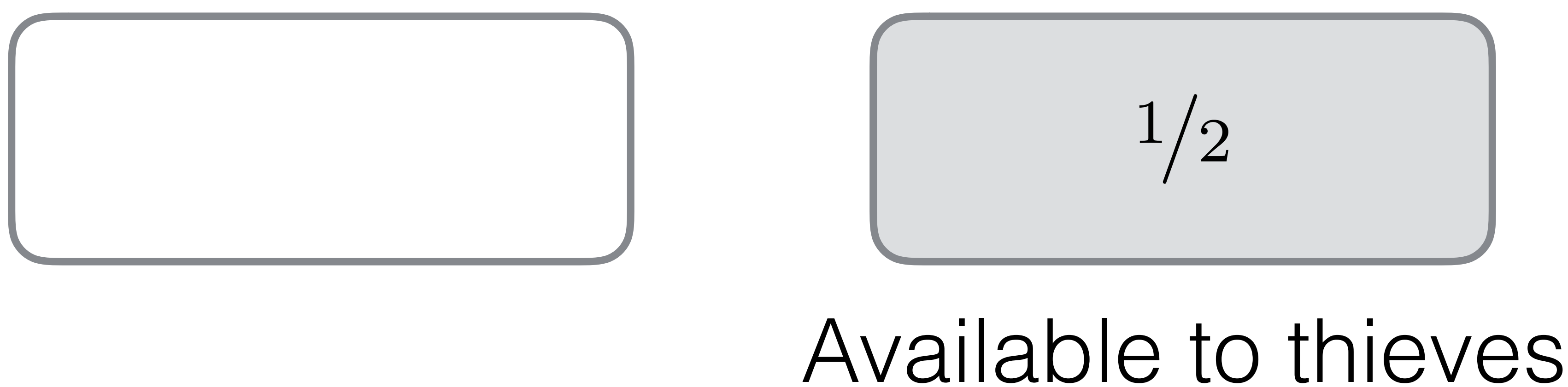
→ Chunking (granularity control) is *implicit*

A. Tzannes et al., Lazy Binary Splitting: A Run-Time Adaptive Work-Stealing Scheduler, PPoPP 2010

Lazy Binary Splitting

A. Tzannes et al., Lazy Binary Splitting: A Run-Time Adaptive Work-Stealing Scheduler, PPoPP 2010

Lazy Binary Splitting



A. Tzannes et al., Lazy Binary Splitting: A Run-Time Adaptive Work-Stealing Scheduler, PPoPP 2010

Lazy Guided Splitting

Example: Four workers

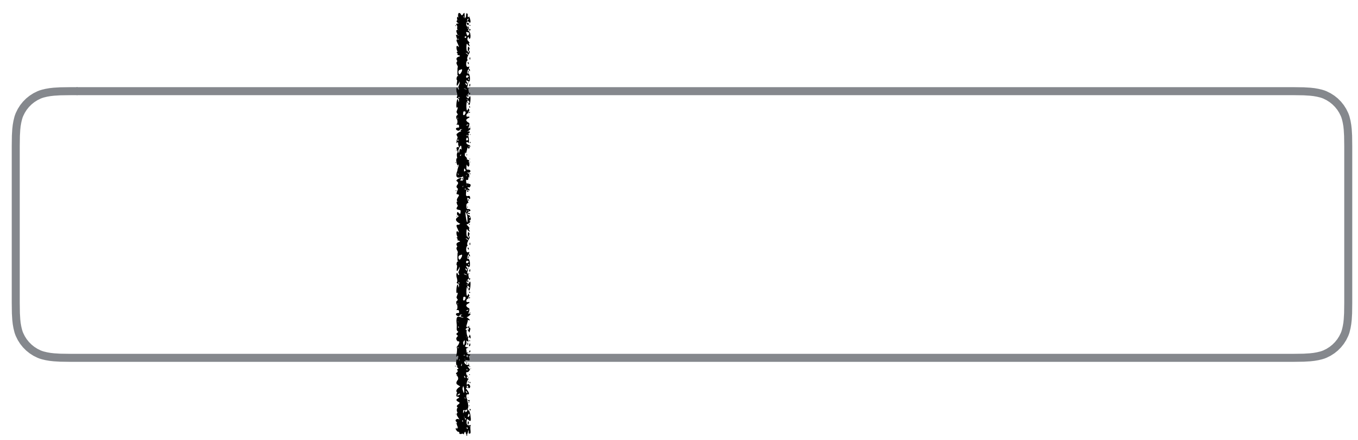
Lazy Guided Splitting

Example: Four workers

Available to thieves

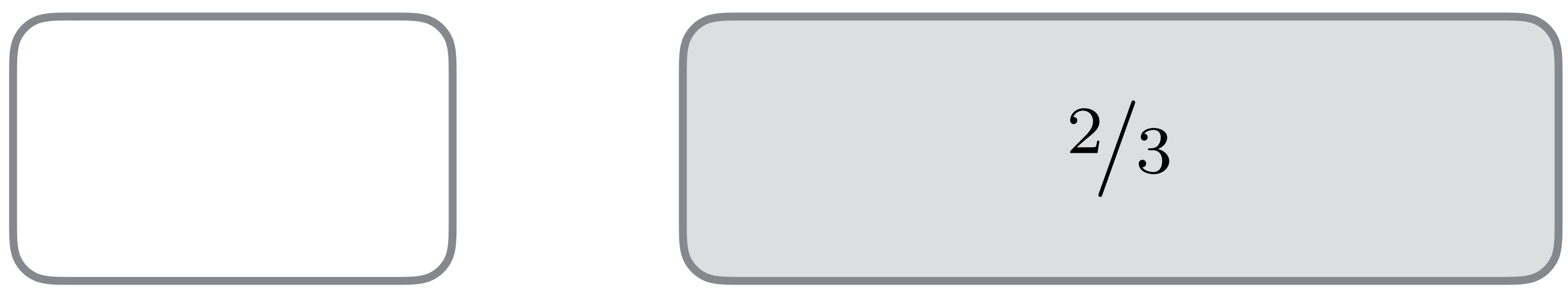
Lazy Adaptive Splitting

Example: Two workers are idle



Lazy Adaptive Splitting

Example: Two workers are idle



Available to thieves

Lazy Splitting

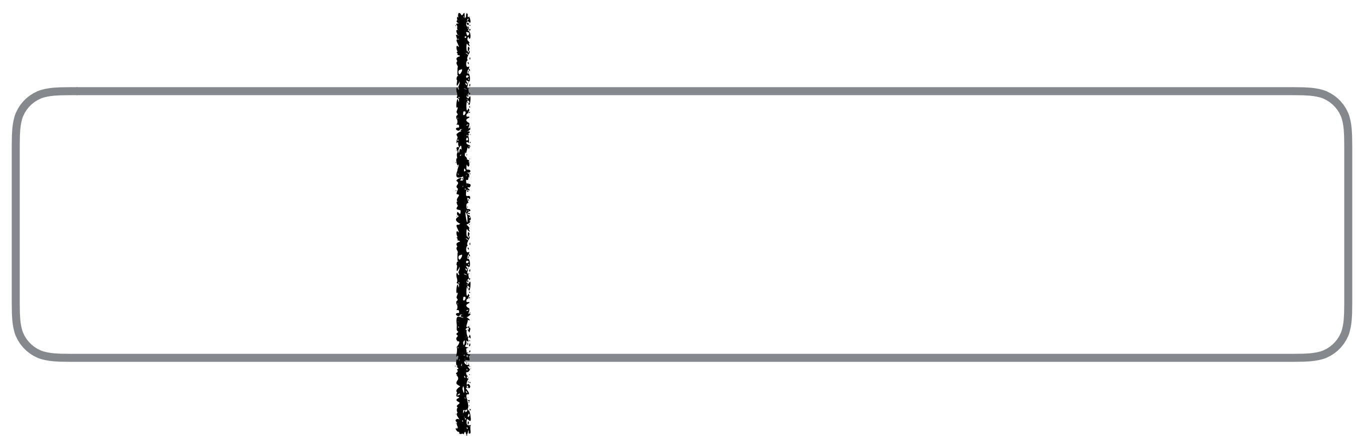
Neither strategy is *truly* lazy

Difficult to know which strategy works best

→ Explicit communication solves this problem

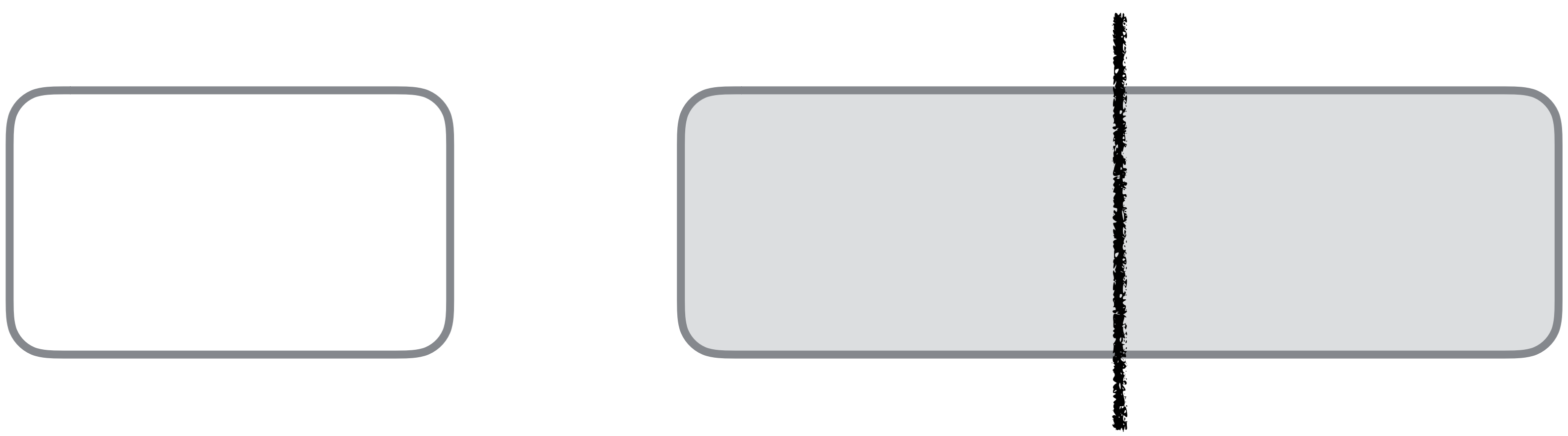
Lazy Adaptive Splitting

Example: Worker receives two steal requests



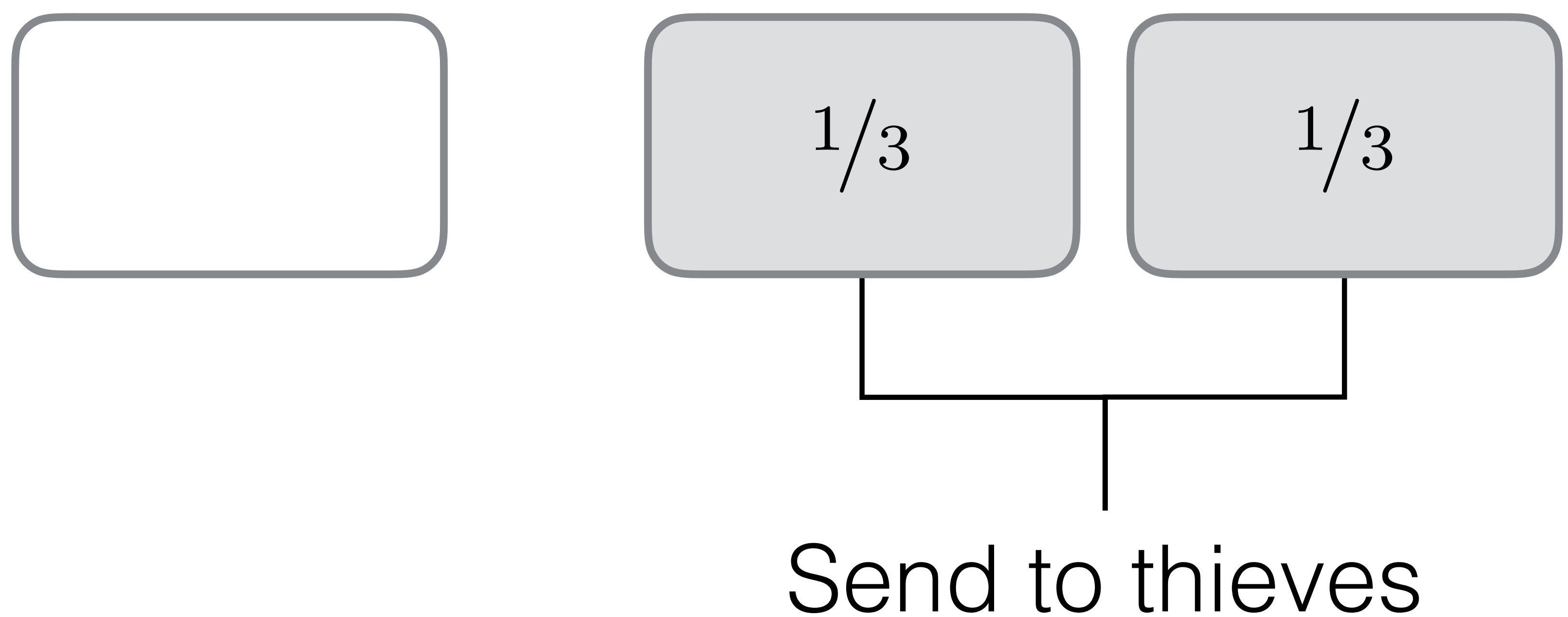
Lazy Adaptive Splitting

Example: Worker receives two steal requests



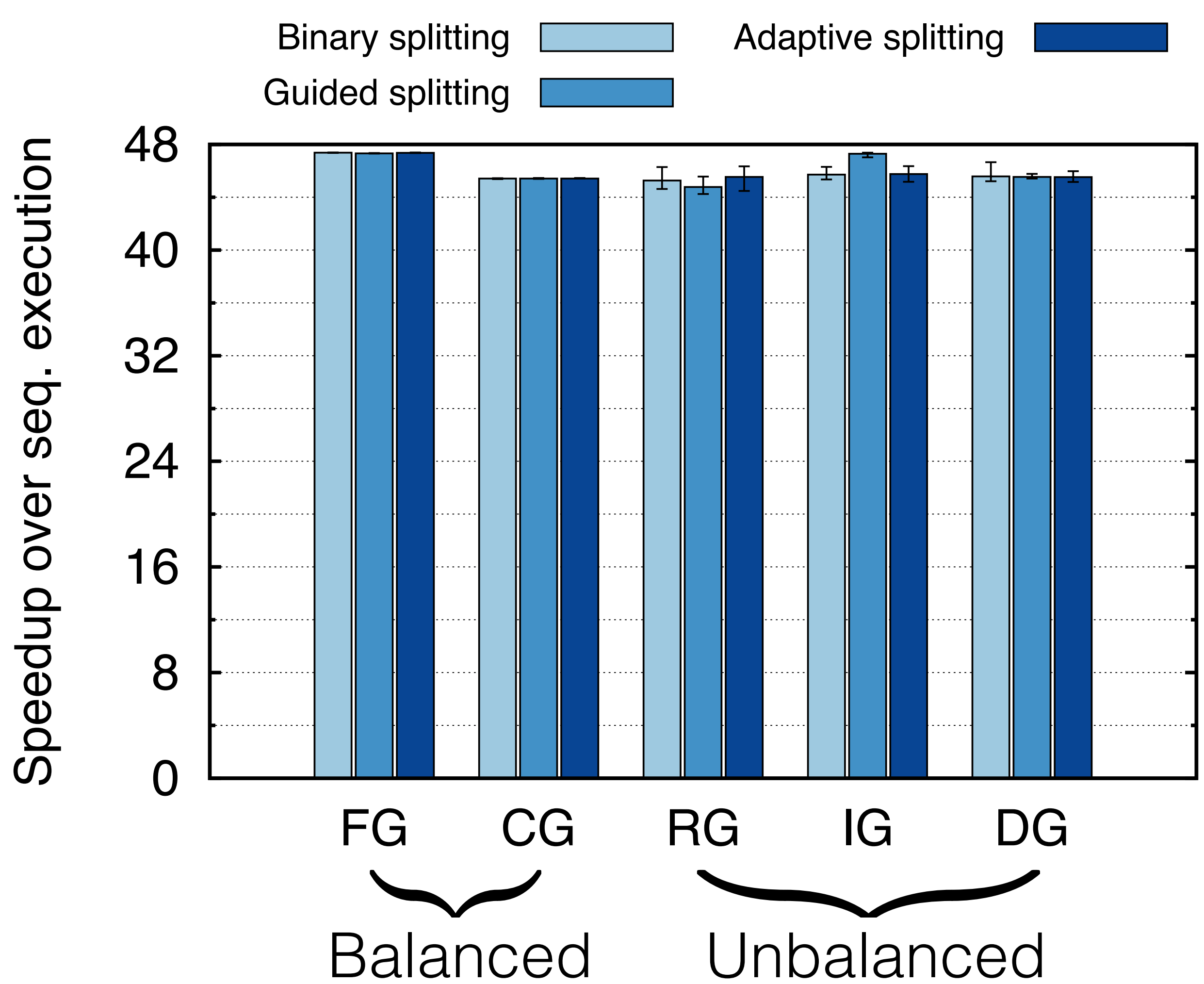
Lazy Adaptive Splitting

Example: Worker receives two steal requests



Performance

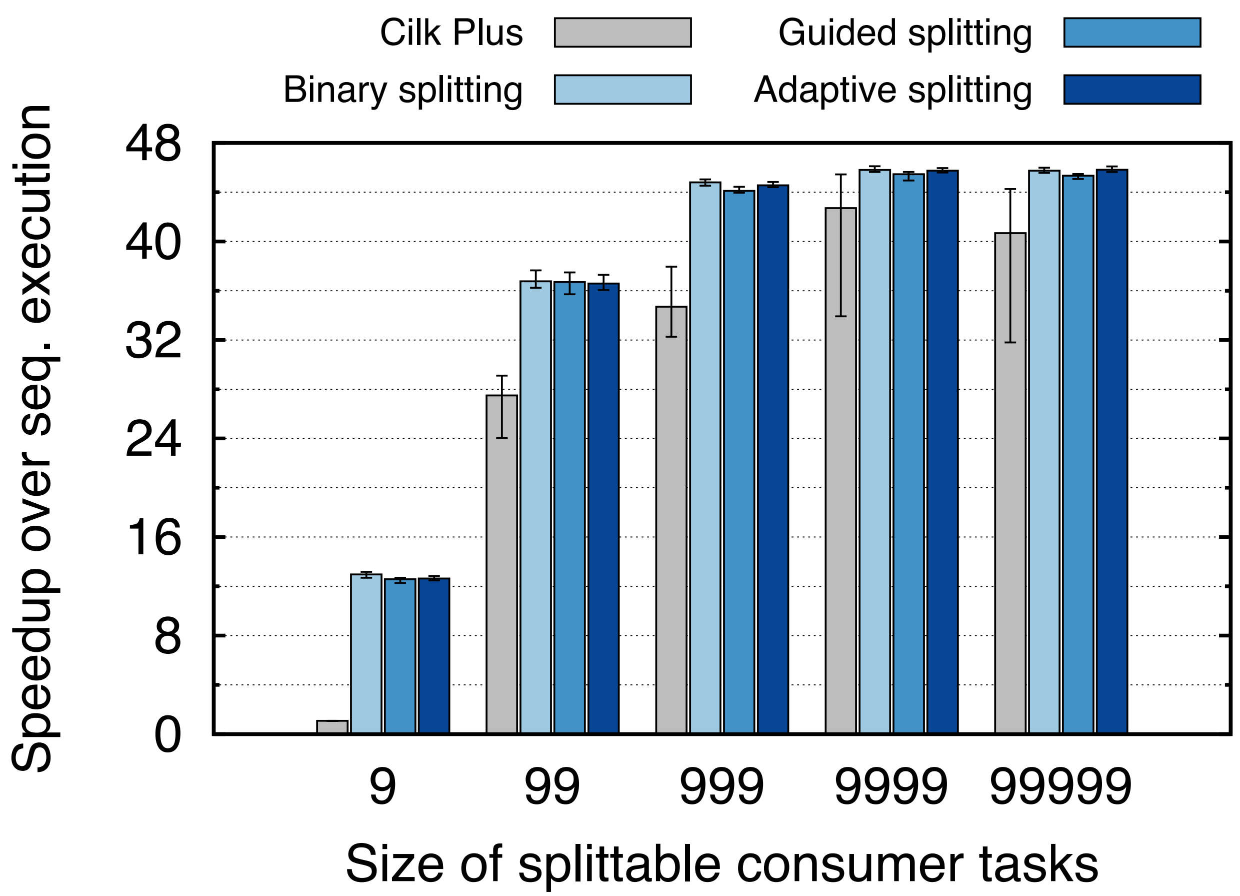
Lazy Splitting

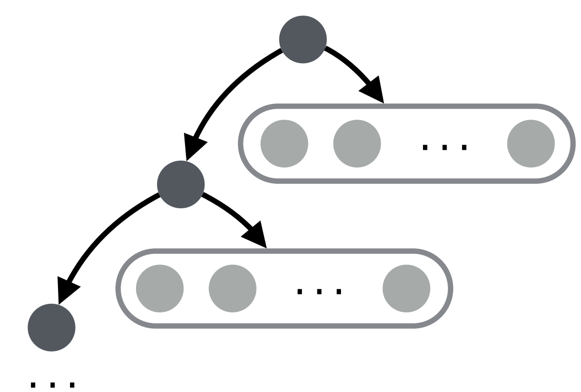


Benchmark: Parallel loops of **F**ine, **C**oarse, **R**andom, **I**ncreasing, and **D**ecreasing **G**ranularity

Performance

Mixing tasks and splittable tasks

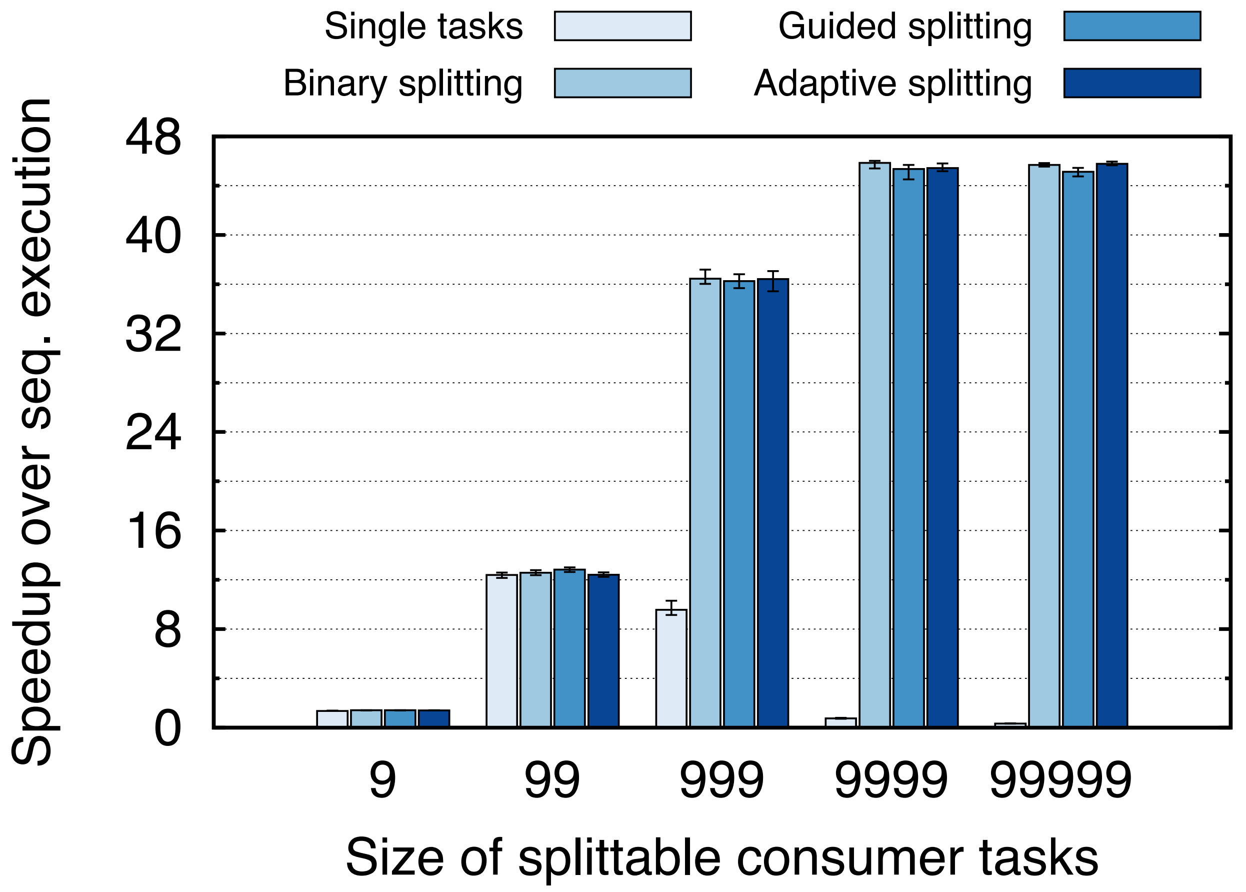


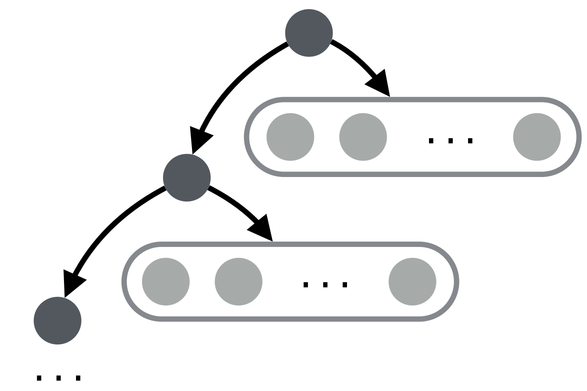


Benchmark: BPC with $d = [10^4, 10^3, \ldots, 1]$, $n = [9, 99, \ldots, 99999]$, and $t = 10\ \mu s$

Performance

Mixing tasks and splittable tasks





Benchmark: BPC with $d = [10^5, 10^4, \ldots, 10]$, $n = [9, 99, \ldots, 99999]$, and $t = 1$ μs

Conclusion

Performance Ranking

Average deviations from the best median speedups

2-socket Intel Xeon (24 threads)

1. Chase-Lev WS	-1.6 %
2. Channel WS	-2.4 %
3. Cilk Plus	-4.6 %
4. Intel OpenMP	-10.7 %

4-socket AMD Opteron (48 threads)

1. Chase-Lev WS	-2.2 %
2. Channel WS	-2.4 %
3. Cilk Plus	-6.9 %
4. Intel OpenMP	-21.8 %

60-core Intel Xeon Phi (240 threads)

1. Chase-Lev WS	-13.7 %
2. Channel WS	-13.7 %
3. Intel OpenMP	-22.2 %
4. Cilk Plus	-28.1 %

21 benchmarks/workloads (20 in the case of Cilk Plus)

David Chase and Yossi Lev, Dynamic Circular Work-Stealing Deque, SPAA 2005

Summary

- Work-stealing runtime system with
  - private deques
  - channel communication

  } Flexibility ✔
- Workers
  - forward steal requests
  - adapt their stealing strategy
  - split tasks lazily

  } Performance ✔