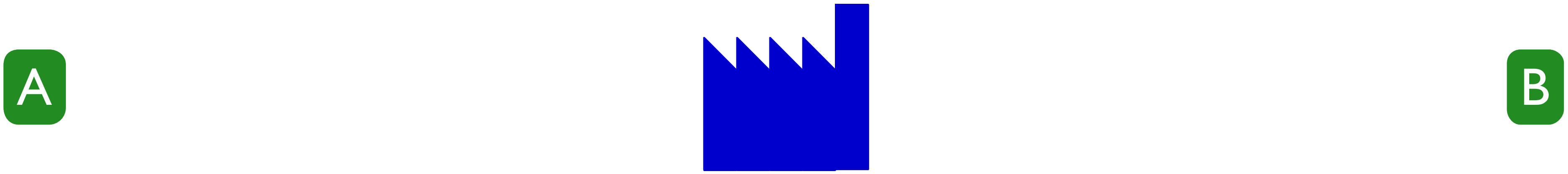
Introduction to Concurrency and Parallelism

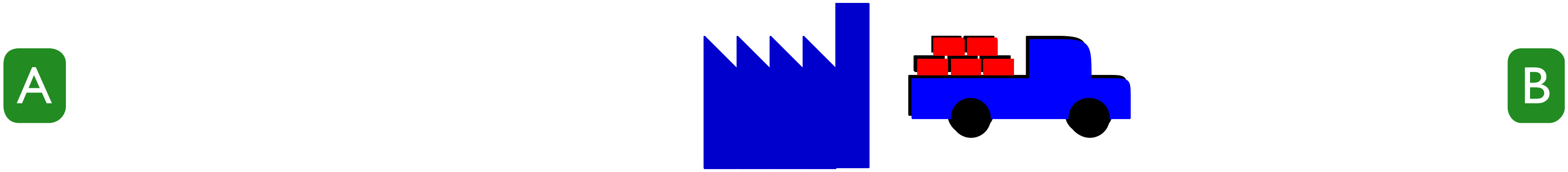
Concurrency

Concurrency: two tasks, any order



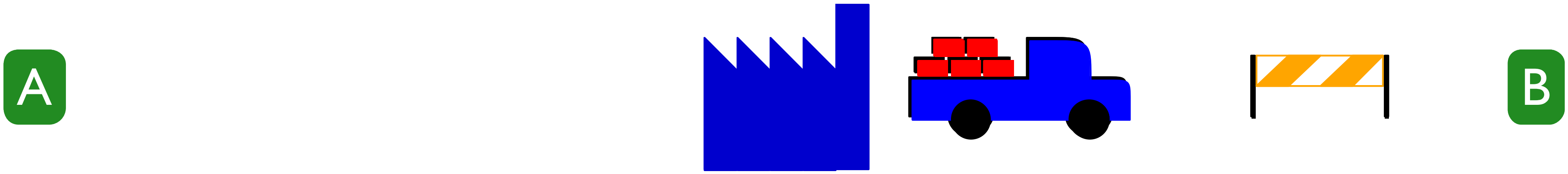
Concurrency

Concurrency: two tasks, any order



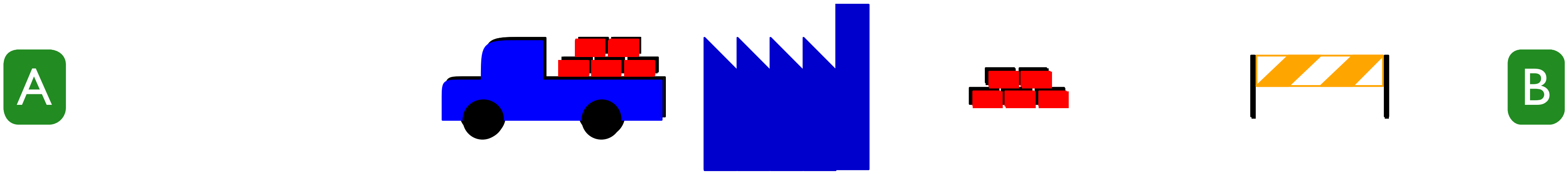
Concurrency

Concurrency: two tasks, any order



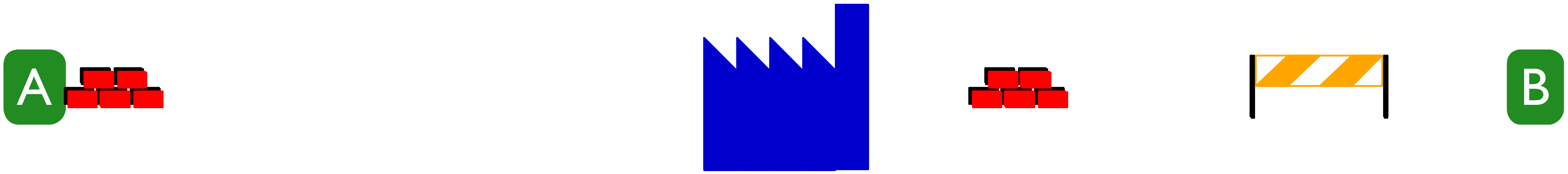
Concurrency

Concurrency: two tasks, any order



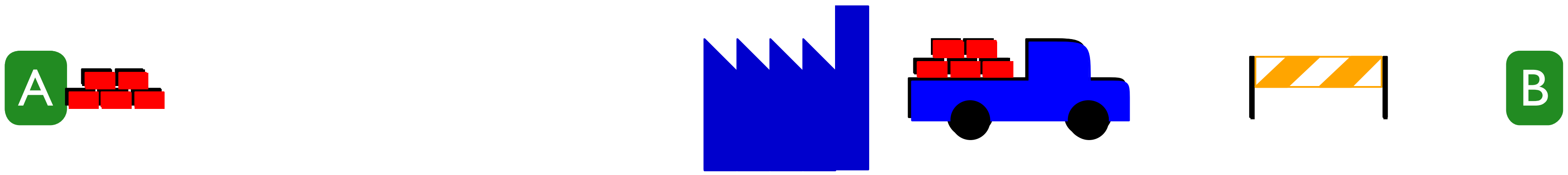
Concurrency

Concurrency: two tasks, any order



Concurrency

Concurrency: two tasks, any order

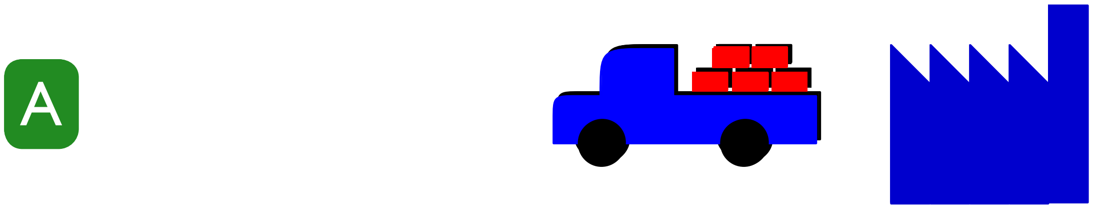


Concurrency is ***non-deterministic***

(whether A or B gets bricks first)

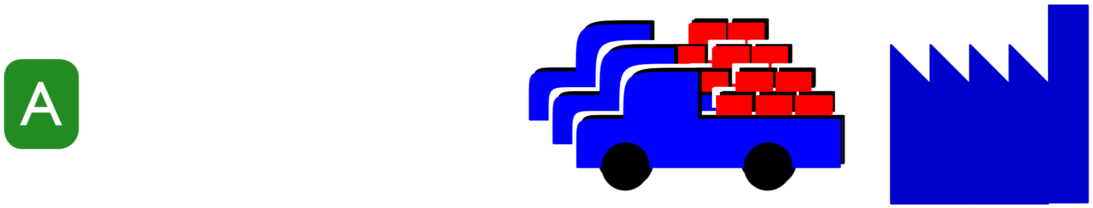
Parallelism

Parallelism: one task, faster



Parallelism

Parallelism: one task, faster

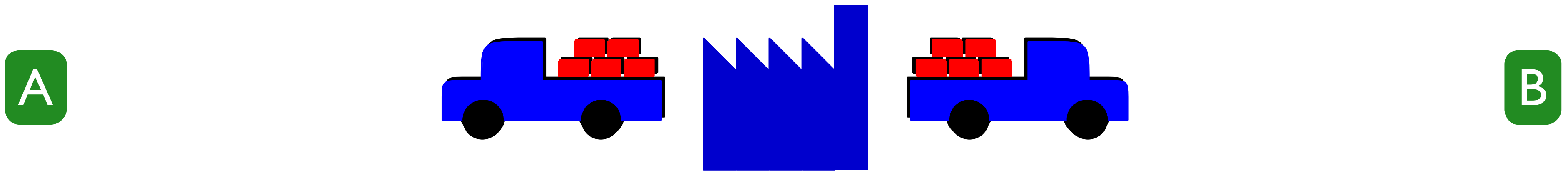


Parallelism can be ***deterministic***

(same bricks always delivered to A)

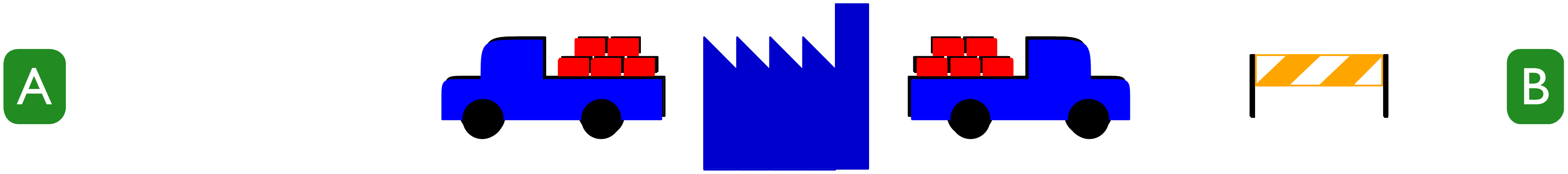
Parallelism vs. Concurrency

Bricks to both A and B as a single task:



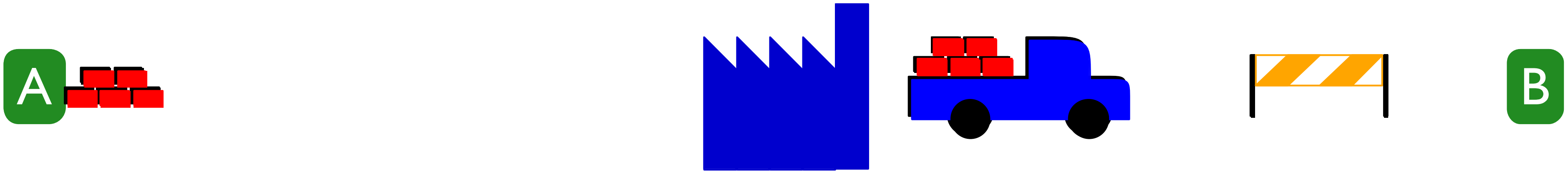
Parallelism vs. Concurrency

Bricks to both A and B as a single task:



Parallelism vs. Concurrency

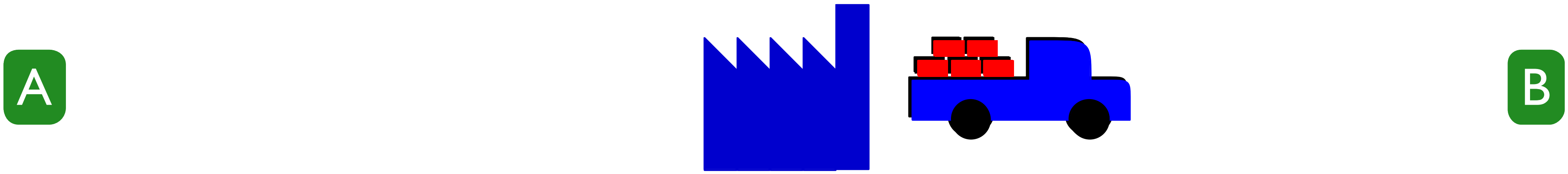
Bricks to both A and B as a single task:



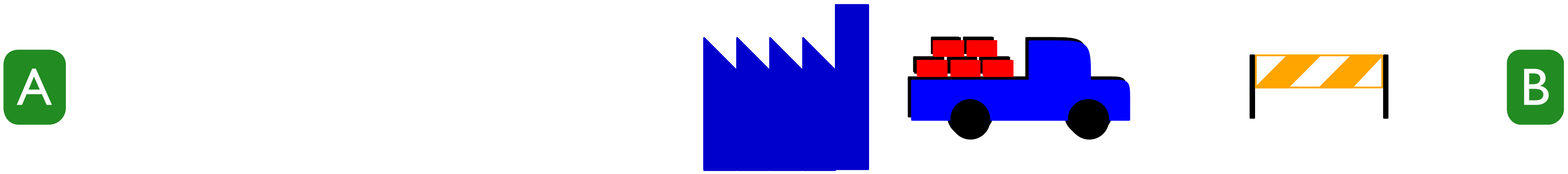
Parallelism may have internal concurrency!

Whether you see the concurrency depends on your layer of abstraction

Why Concurrency is Hard

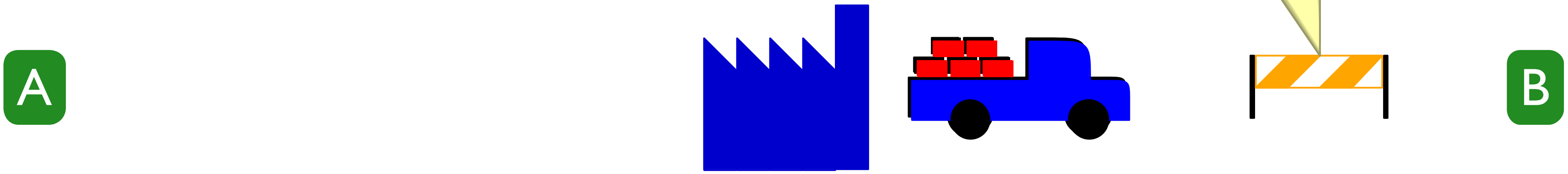


Why Concurrency is Hard

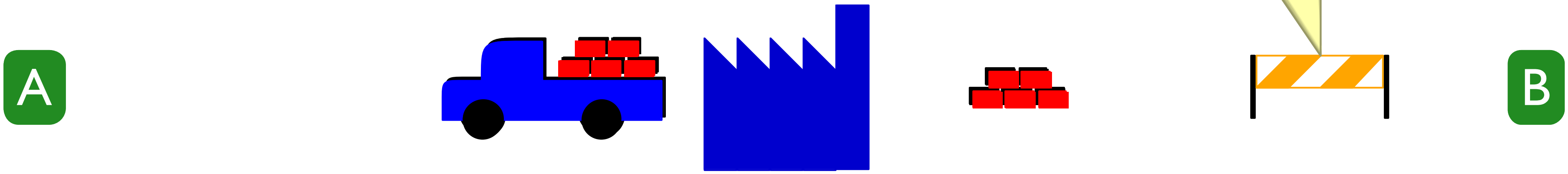


Why Concurrency is Hard

When barrier is removed, **`drive`**

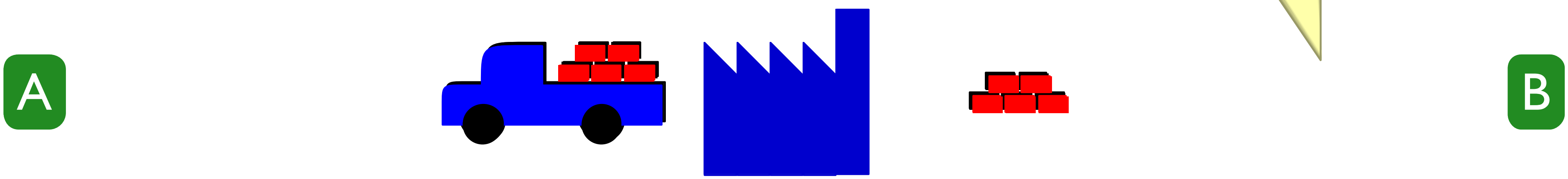


Why Concurrency is Hard



When barrier is removed, **`drive`**

Why Concurrency is Hard

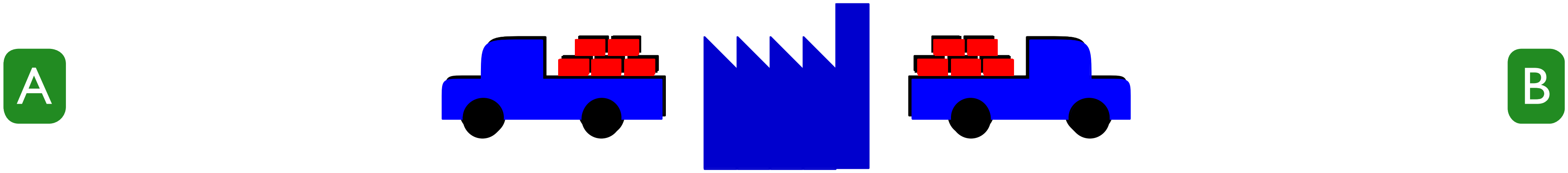


When barrier is removed, **`drive`**

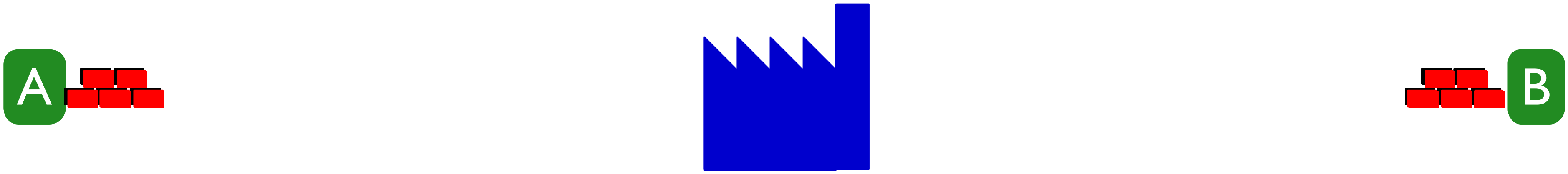
`no such method: drive in:` 🧱

General problem: ***shared resources***

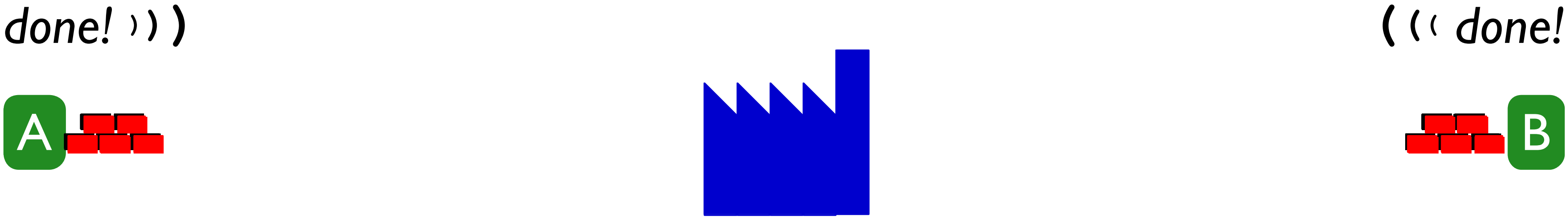
Why Parallelism is Hard: 1



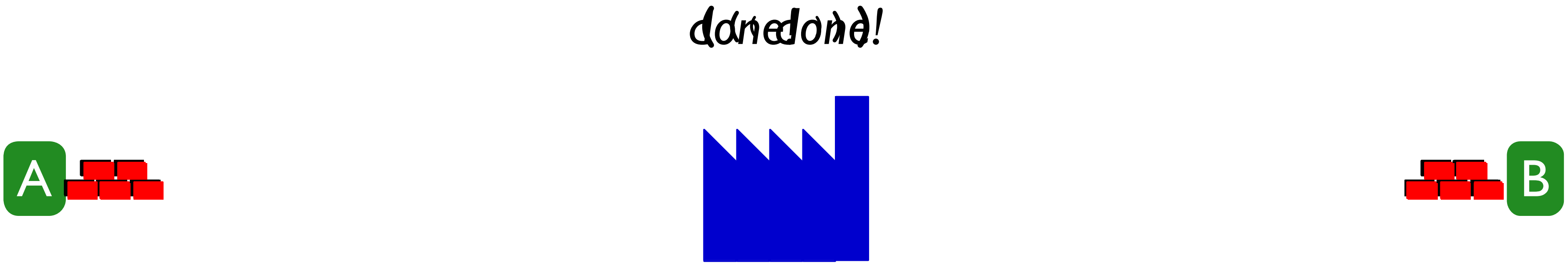
Why Parallelism is Hard: 1



Why Parallelism is Hard: 1



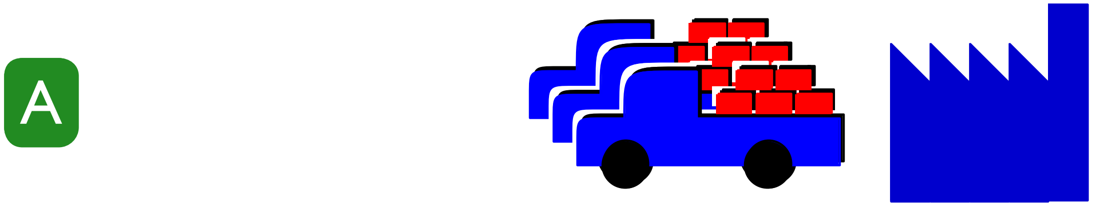
Why Parallelism is Hard: 1



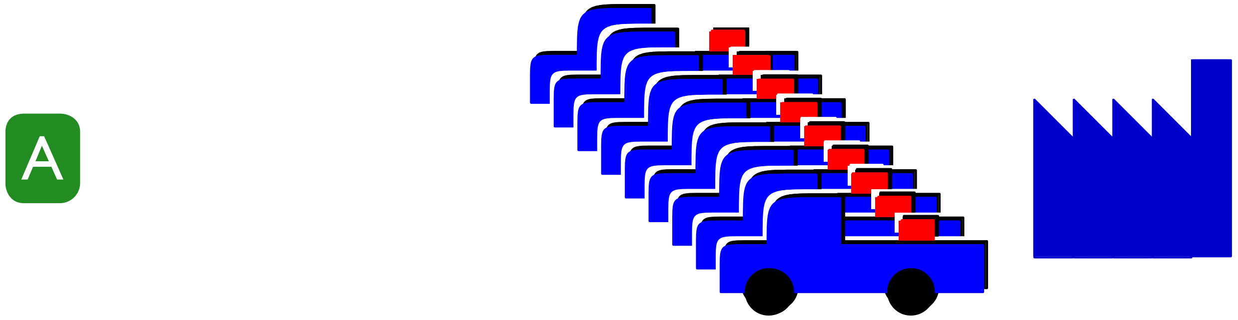
Concurrency is hard — including internal concurrency

“Systems” programmers deal with internal concurrency

Why Parallelism is Hard: 2



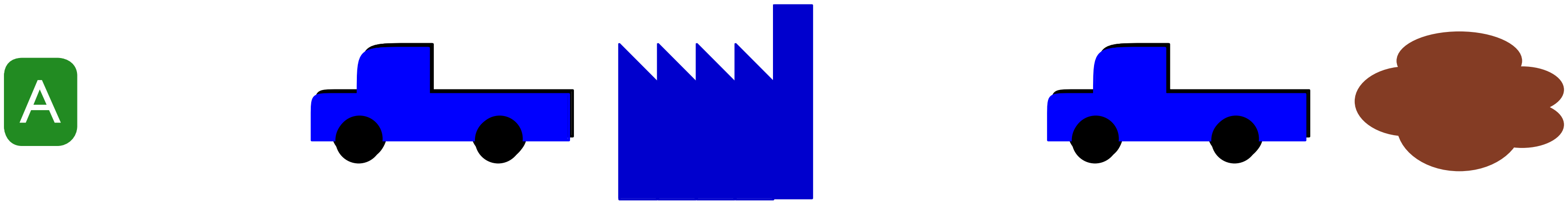
Why Parallelism is Hard: 2



It's easy to ask for too much parallelism

(Each truck adds overhead)

Why Parallelism is Hard: 3



Dependencies limit parallelism

Algorithm designers deal with dependencies

Parallelism in an Algorithm

```
(define (quicksort! vec n m)
  (when (> (- m n) 1)
    (let* ([pivot (vector-ref vec n)]
           [pre
            (for/fold ([pre n]) ([i (in-range (add1 n) m)])
              (let ([v (vector-ref vec i)])
                (cond
                  [(< v pivot)
                   (vector-set! vec pre v)
                   (vector-set! vec i (vector-ref vec (add1 pre)))
                   (values (add1 pre))]
                  [else (values pre)])))])
      (vector-set! vec pre pivot)
      ; Two recursive calls are independent:
      (quicksort! vec n pre)
      (quicksort! vec (add1 pre) m))))
```

Parallelism in an Algorithm

```
(define (quicksort! vec n m)
  (when (> (- m n) 1)
    (let* ([pivot (vector-ref vec n)]
           [pre
            (for/fold ([pre n]) ([i (in-range (add1 n) m)])
              (let ([v (vector-ref vec i)])
                (cond
                  [(< v pivot)
                   (vector-set! vec pre v)
                   (vector-set! vec i (vector-ref vec (add1 pre)))
                   (values (add1 pre))]
                  [else (values pre)])))])
      (vector-set! vec pre pivot)
      (parallel-begin ; ok, but...
       (quicksort! vec n pre)
       (quicksort! vec (add1 pre) m)))))
```

Request too much parallelism ⇒ management overload

Parallelism in an Algorithm

```
(define (quicksort! vec n m)
  (when (> (- m n) 1)
    (let* ([pivot (vector-ref vec n)]
           [pre
            (for/fold ([pre n]) ([i (in-range (add1 n) m)])
              (let ([v (vector-ref vec i)])
                (cond
                  [(< v pivot)
                   (vector-set! vec pre v)
                   (vector-set! vec i (vector-ref vec (add1 pre)))
                   (values (add1 pre))]
                  [else (values pre)])))])
      (vector-set! vec pre pivot)
      (if (> (- m n) (quotient (vector-length vec) 100)) ; ugh
          (parallel-begin
           (quicksort! vec n pre)
           (quicksort! vec (add1 pre) m))
          (begin
           (quicksort! vec n pre)
           (quicksort! vec (add1 pre) m))))))
```

Concurrency vs. Parallelism

In principle:

Parallelism ≠ Concurrency

- Parallelism is for higher ***throughput***
- Concurrency is for lower ***latency***

In practice (for now):

Parallelism ⇐⇒ **Concurrency**

- Parallelism via multiple processors
- Concurrency via multiple (virtual) processors

Threads

A ***thread*** is a virtual concurrent processor

- Racket: `thread` creates a thread

```
(define a
  (thread (lambda () (printf "a\n"))))
(define b
  (thread (lambda () (printf "b\n"))))
(sync a)
(sync b)
```

... but no parallelism!

Threads

A ***thread*** is a virtual concurrent processor

- C: **`pthread_create()`** creates a thread

```
void *go(void *s) {
  printf("%s\n", (char *)s);
  return NULL;
}

pthread_t a, b;
pthread_create(&a, NULL, go, "a");
pthread_create(&b, NULL, go, "b");

pthread_join(a, NULL);
pthread_join(b, NULL);
```

Futures

A ***future*** is a task that can run in parallel

- Racket: `future` creates a future

```
(define a
  (future (lambda () (+ 1 2))))
(define b
  (future (lambda () (+ 3 4))))
(touch a)
(touch b)
```

... but no guaranteed concurrency!

OpenMP Tasks

A ***task*** is a task that can run in parallel

- C + OpenMP: **`#pragma omp task`** creates a task

```
#pragma omp task
v1 = add_one_plus_two();
#pragma omp task
v2 = add_three_plus_four();
```

... and no guaranteed concurrency!