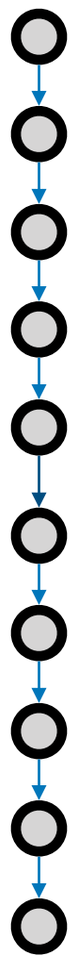
Lecture 11

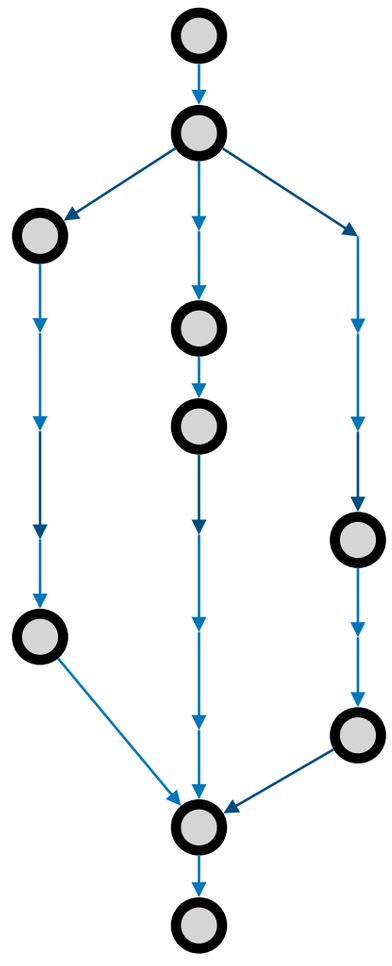
Asynchronous Calls

Previously, in cs2030..

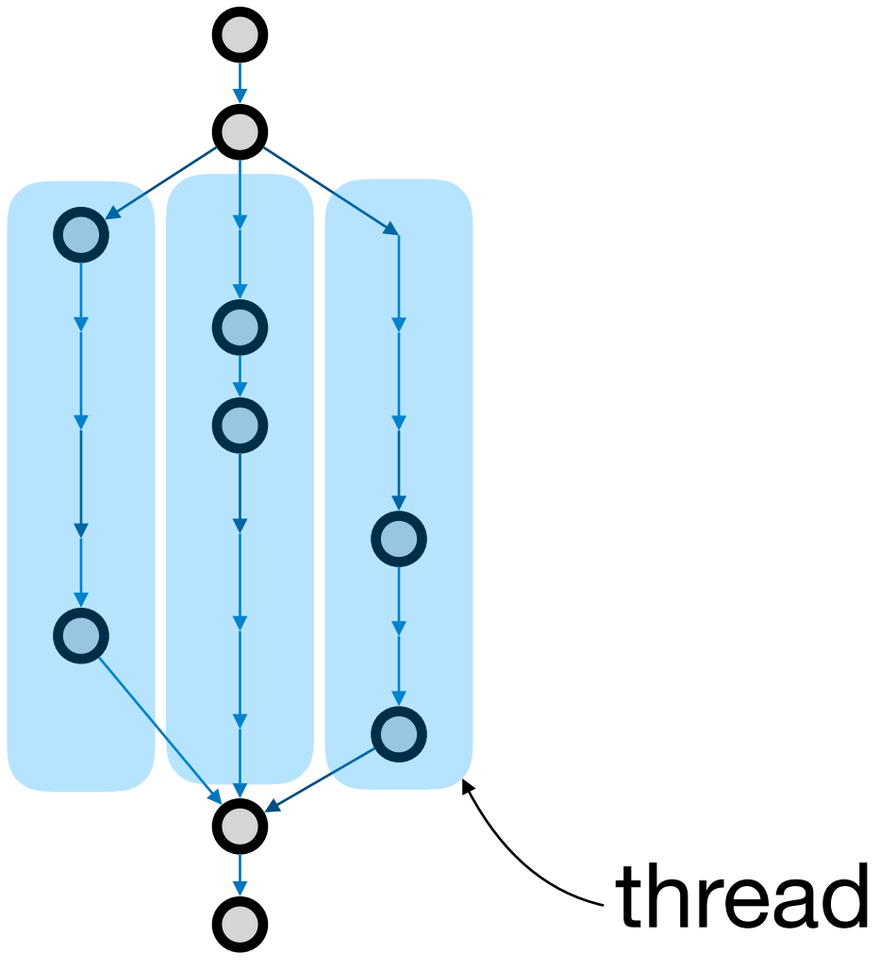
Sequential Program



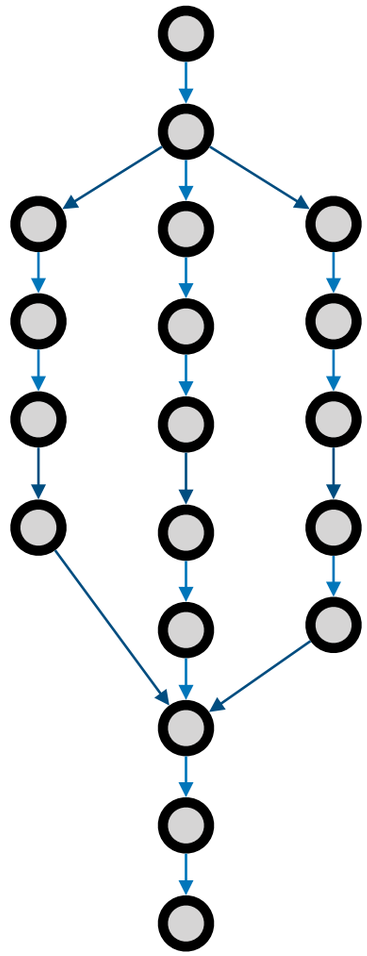
Concurrent Program



Concurrent Program



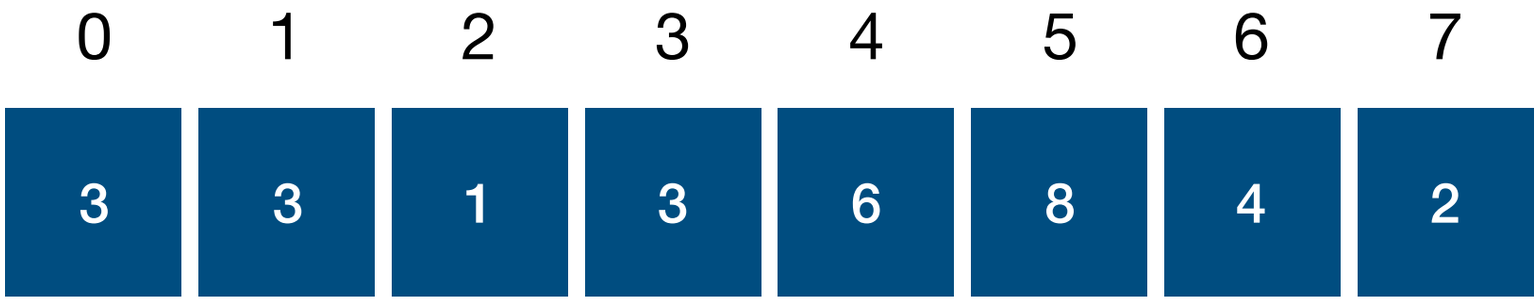
Parallel Program



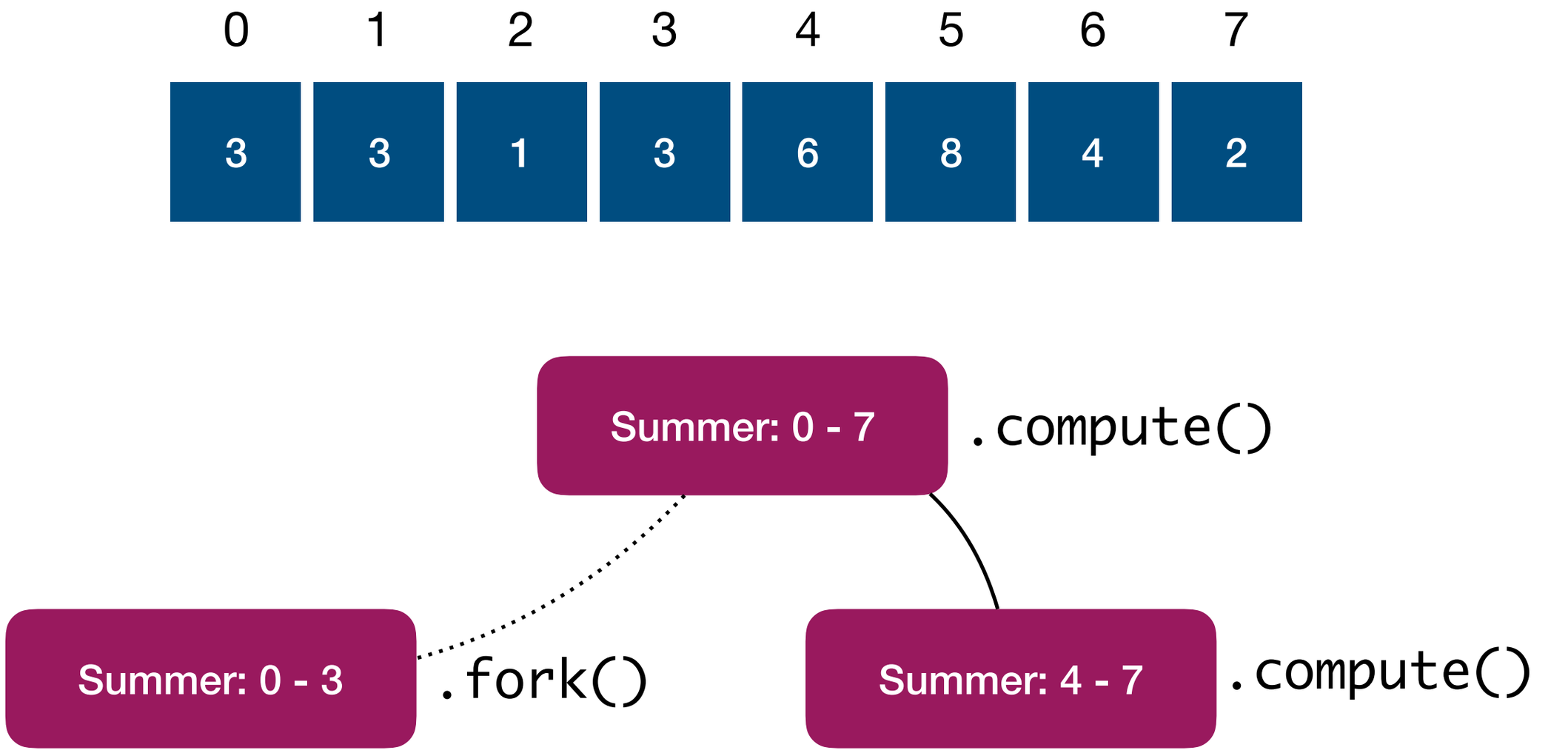
Good for Parallelization

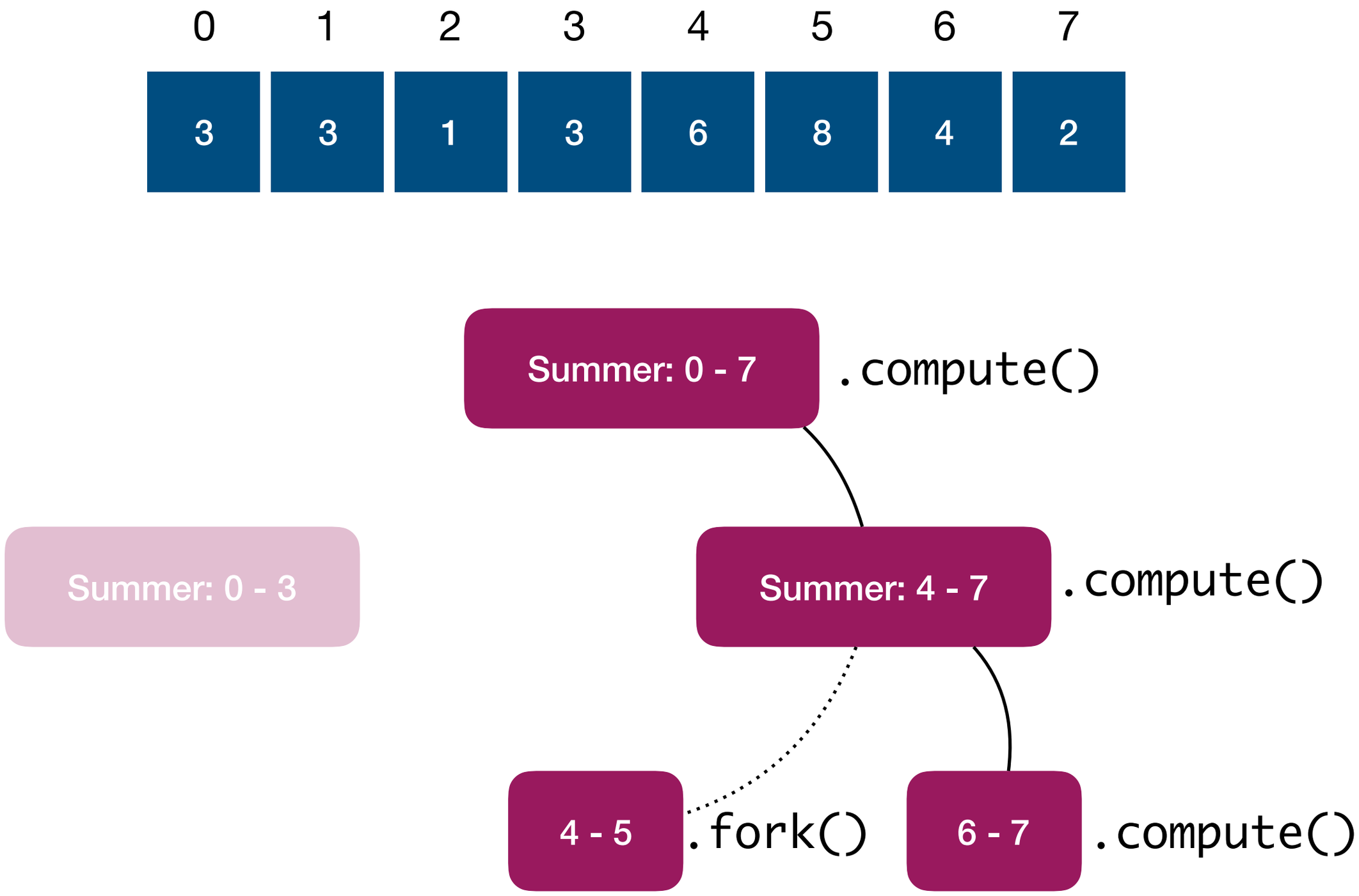
- Non-interference
- Stateless
- No side-effect
- Associative Reduction

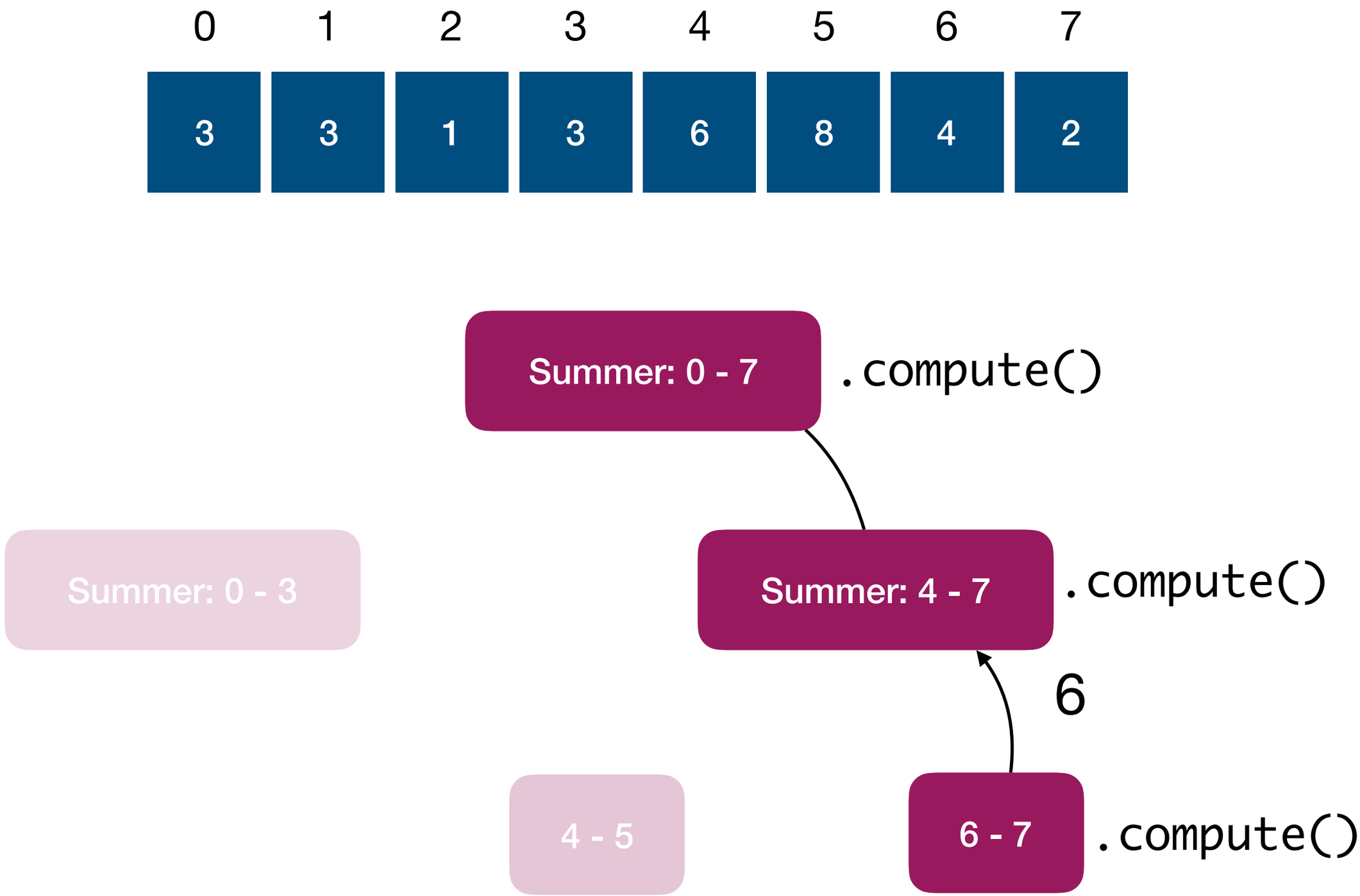
```
protected Integer compute() {
    // stop splitting if small enough
    if (high - low < FORK_THRESHOLD) {
      int sum = 0;
      for (int i = low; i < high; i++) {
        sum += array[i];
      }
      return sum;
    }

    int middle = (low + high) / 2;
    Summer left = new Summer(low, middle, array);
    Summer right = new Summer(middle, high, array);
    left.fork();
    return right.compute() + left.join();
}
```

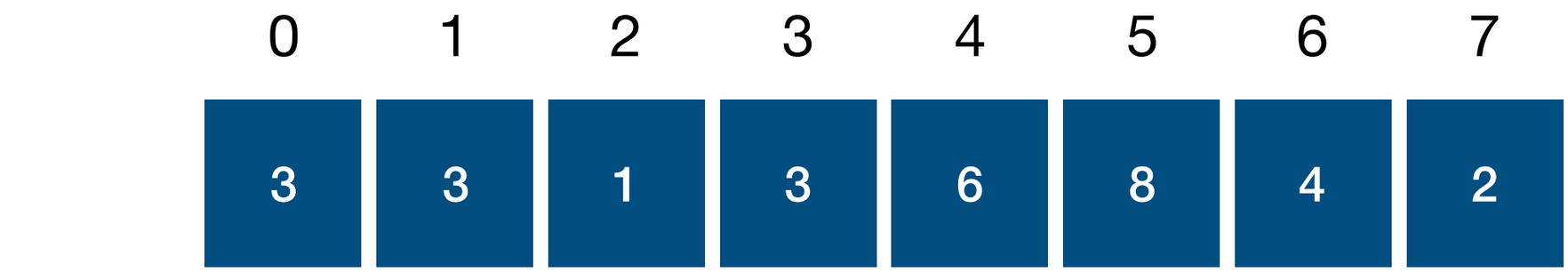


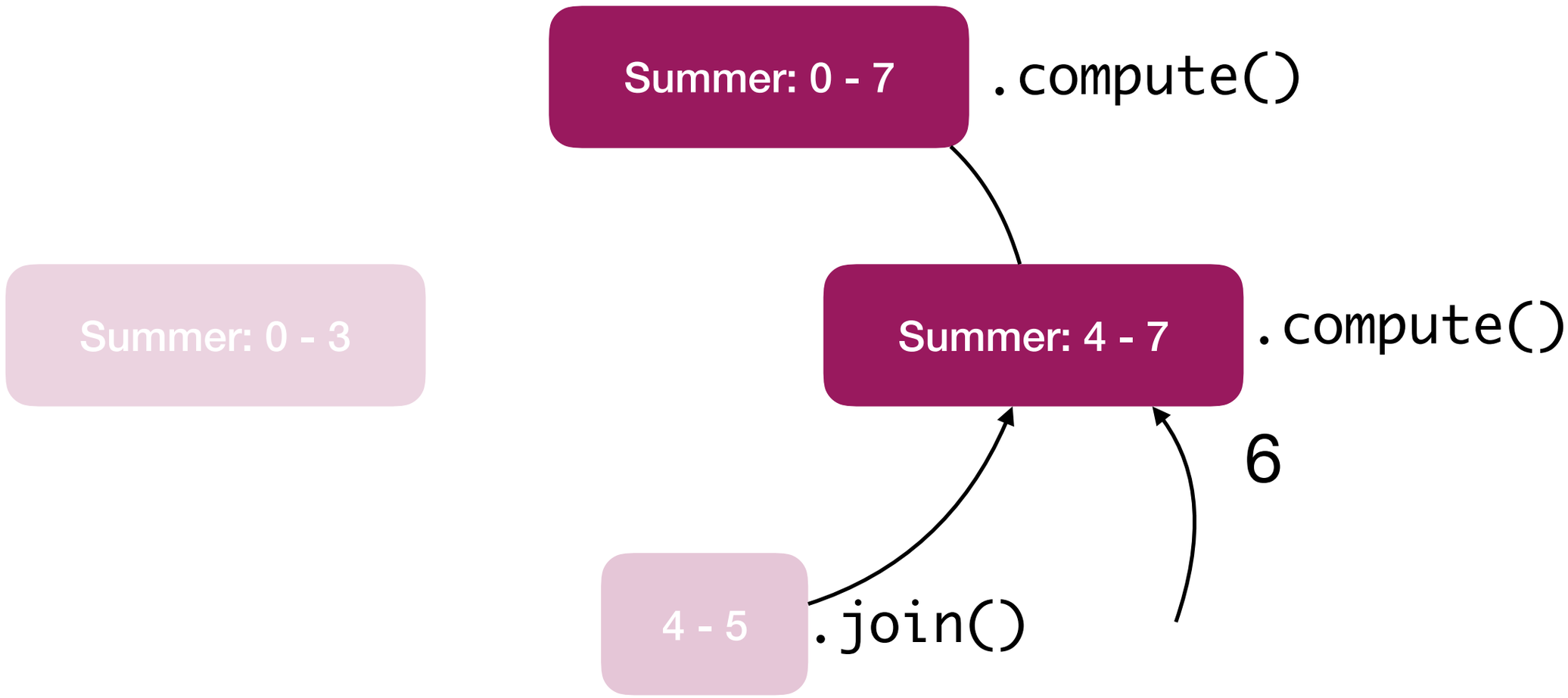
Summer: 0 - 7 `.compute()`

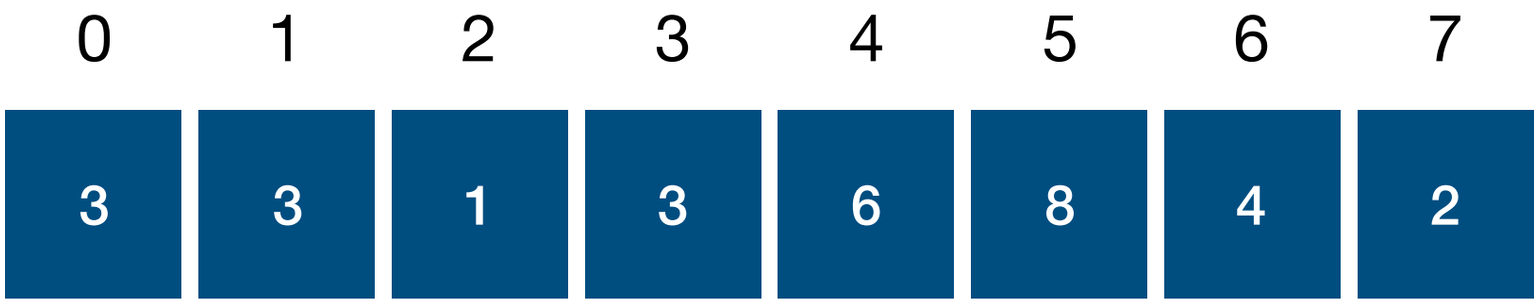

0
1
2
3
4
5
6
7
3
3
1
3
6
8
4
2
Summer: 0 - 7
.compute()
Summer: 0 - 3
.fork()
Summer: 4 - 7
.compute()

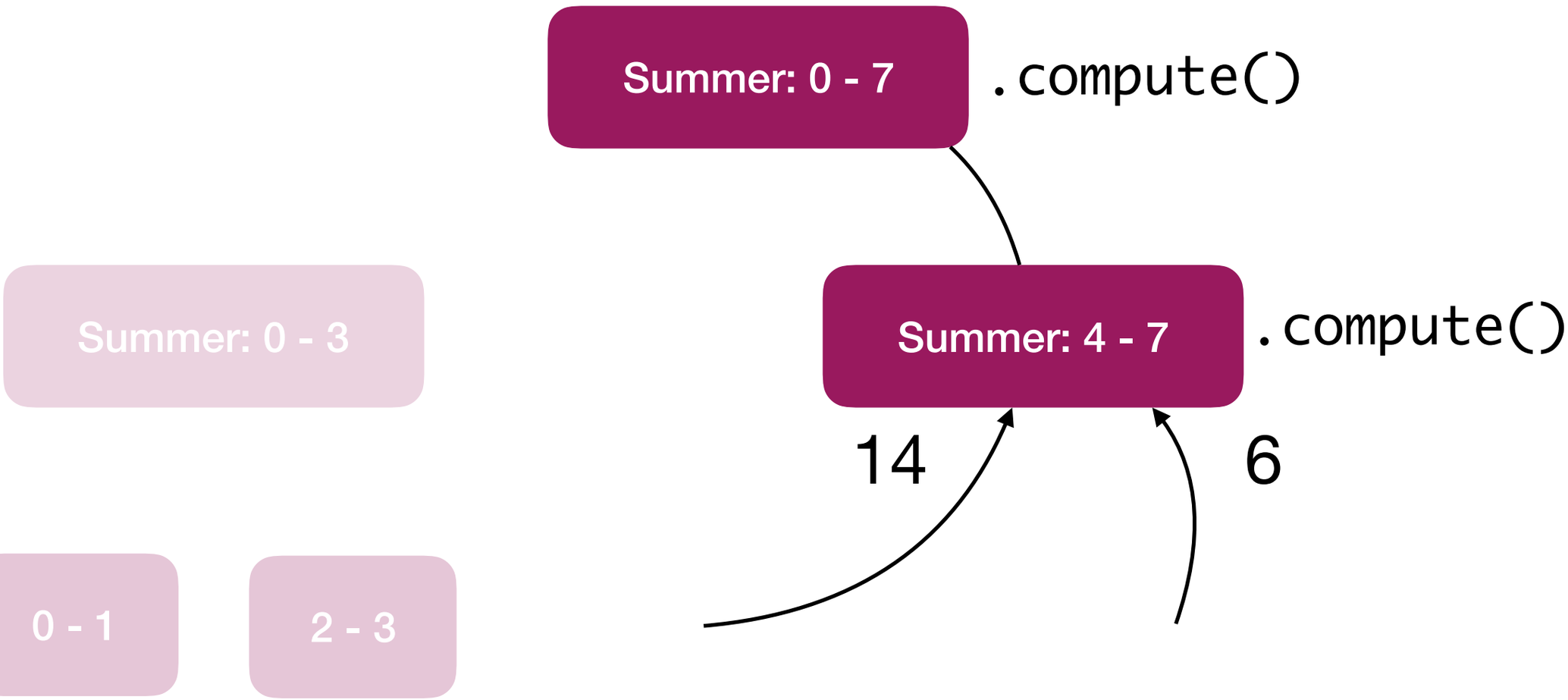

0
1
2
3
4
5
6
7
3
3
1
3
6
8
4
2
Summer: 0 - 7
.compute()
Summer: 0 - 3
Summer: 4 - 7
.compute()
4 - 5
.fork()
6 - 7
.compute()

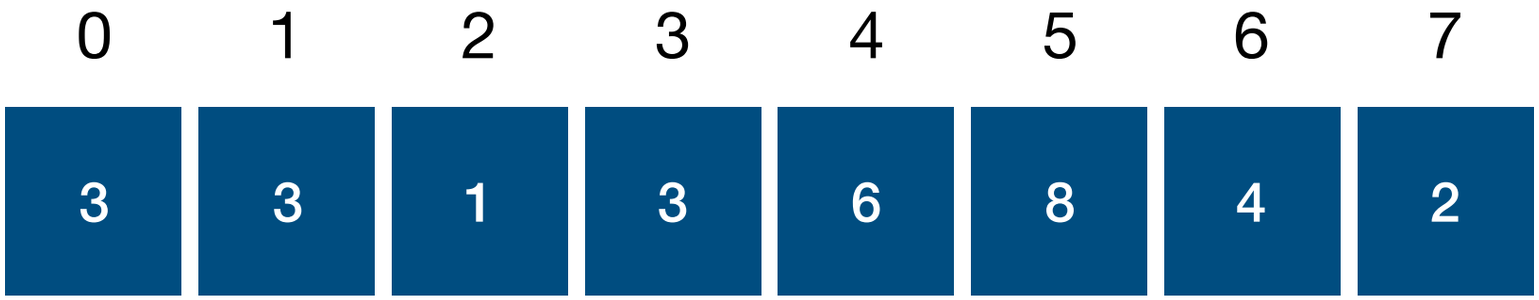

0
1
2
3
4
5
6
7
3
3
1
3
6
8
4
2
Summer: 0 - 7
.compute()
Summer: 0 - 3
Summer: 4 - 7
.compute()
6
4 - 5
6 - 7
.compute()

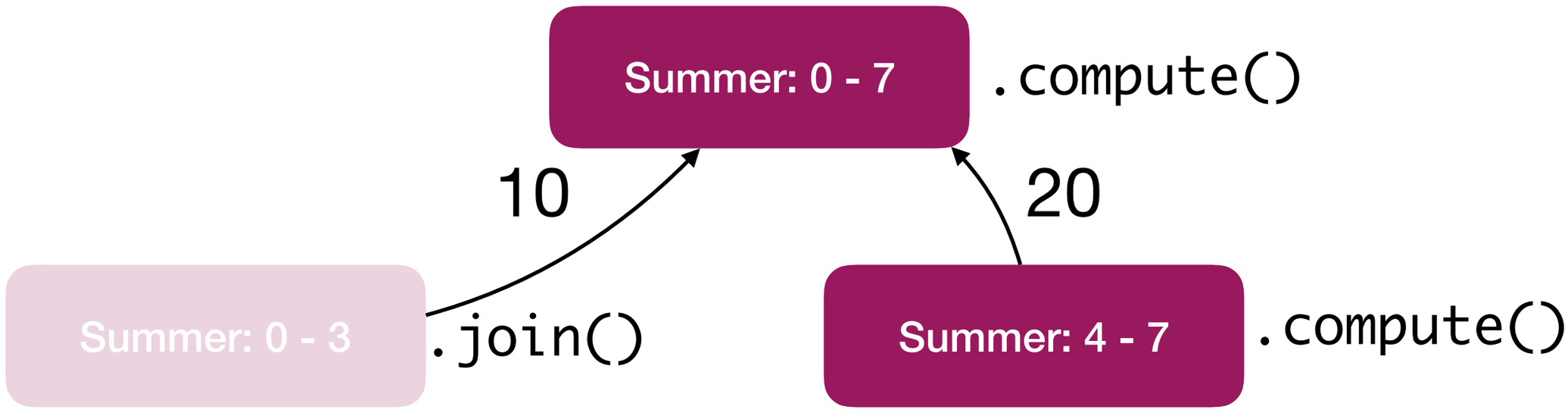

0
1
2
3
4
5
6
7
3
3
1
3
6
8
4
2


Summer: 0 - 7
.compute()
Summer: 0 - 3
Summer: 4 - 7
.compute()
6
4 - 5
.join()

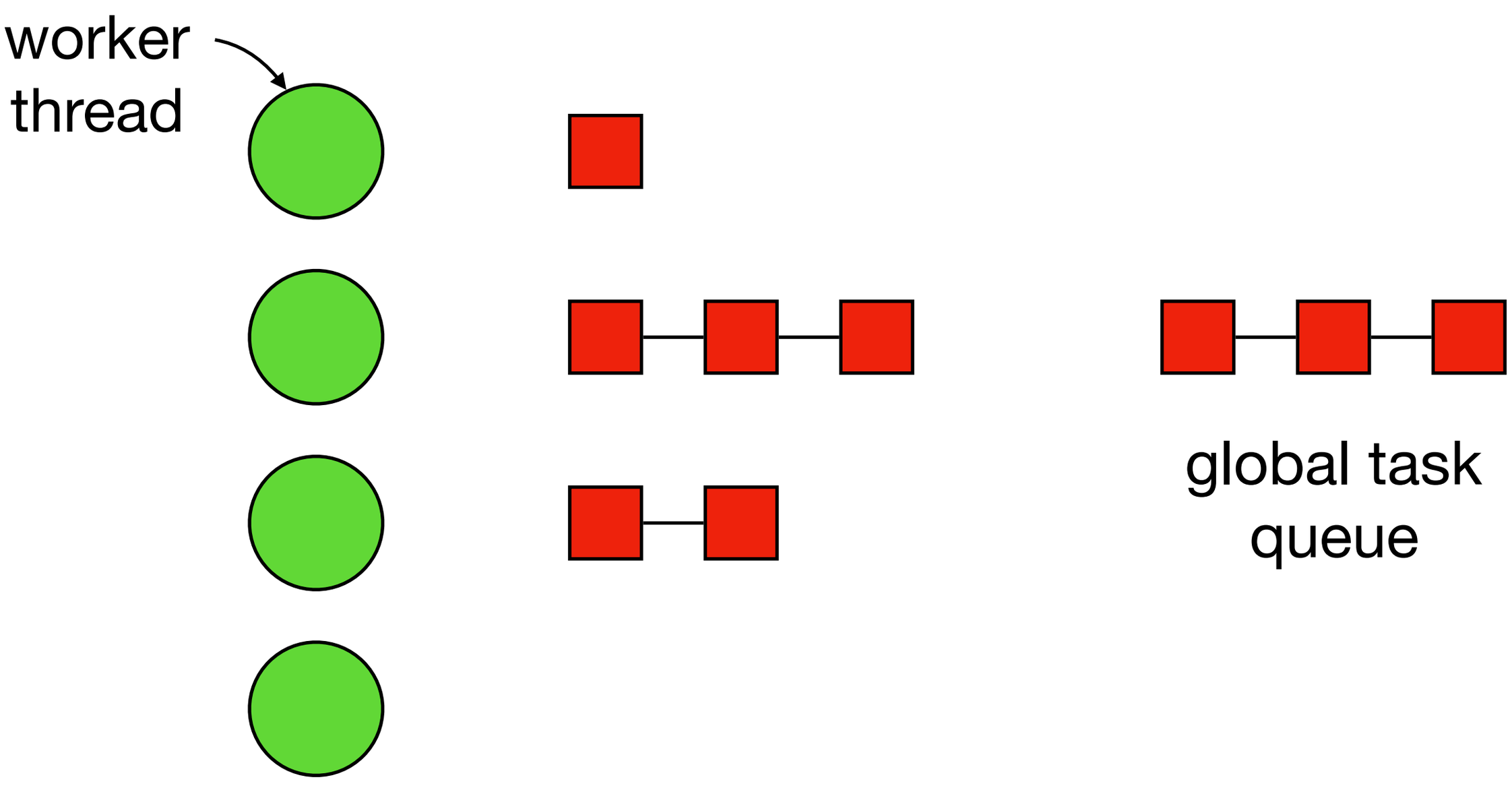

0
1
2
3
4
5
6
7
3
3
1
3
6
8
4
2


Summer: 0 - 7
.compute()
Summer: 0 - 3
Summer: 4 - 7
.compute()
14
6
0 - 1
2 - 3

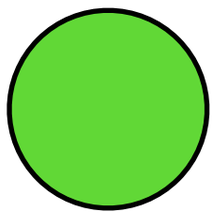

0
1
2
3
4
5
6
7
3
3
1
3
6
8
4
2

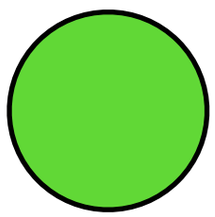

Summer: 0 - 7
.compute()
10
20
Summer: 0 - 3
.join()
Summer: 4 - 7
.compute()

ForkJoinPool



Task Summer(0, 3) arrives





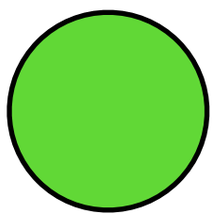
0 - 3

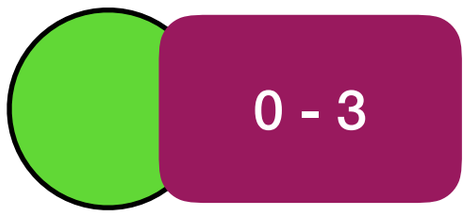
global task queue

thread pool

local task queue

Worker 1 is idle. Took Task 0-3 to compute( )

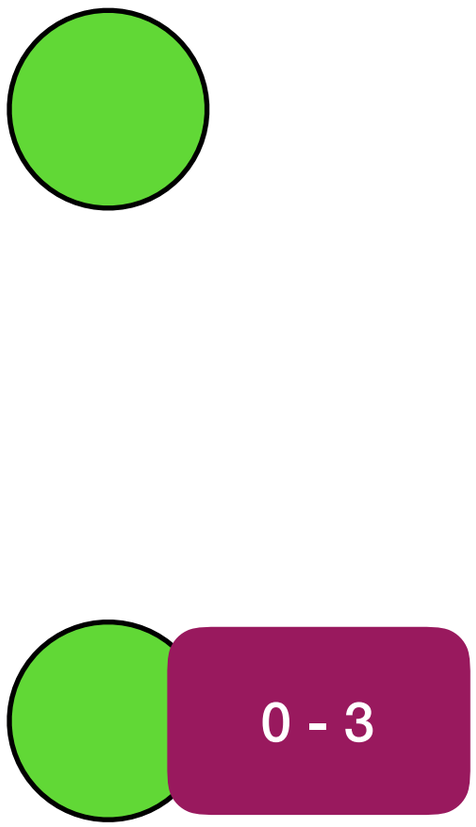




global task queue

thread pool local task queue

Meanwhile Task 4-5 enters queue

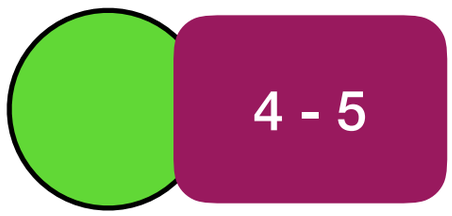


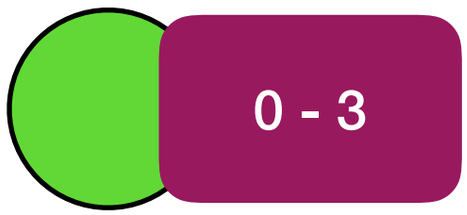
global task queue

thread pool

local task queue

Worker 2 is idle. Take Task 4-5 to compute( )

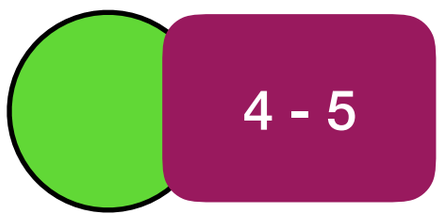




global task queue

thread pool local task queue

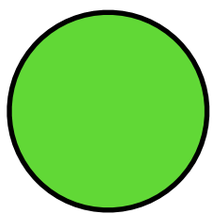
Task 0-3 fork() 0-1. 0-1 joins the task queue



global task queue

thread pool local task queue

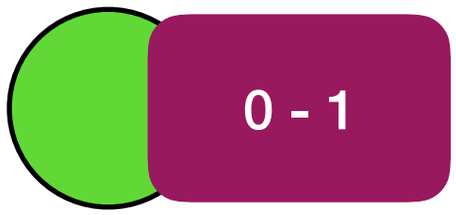
Worker 2 done with 4-5 and is now idle.
Look for tasks to do. Take 0-1

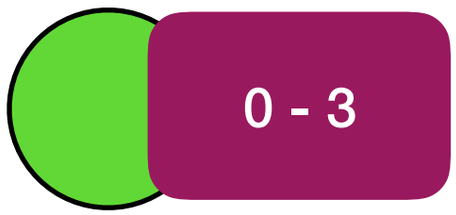


global task queue

thread pool local task queue

Now Task 0-3 calls 0-1.join() and is waiting.
(note Worker 1 is blocked)

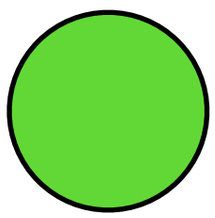


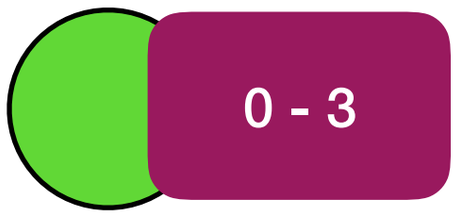


global task queue

thread pool local task queue

Task 0-1 is done. Task 0-3 can unblock.

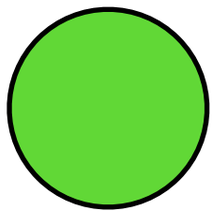


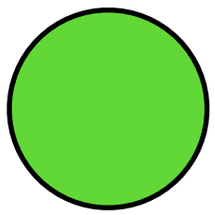


global task queue

thread pool local task queue

Task 0-3 is done. Worker 1 idle.

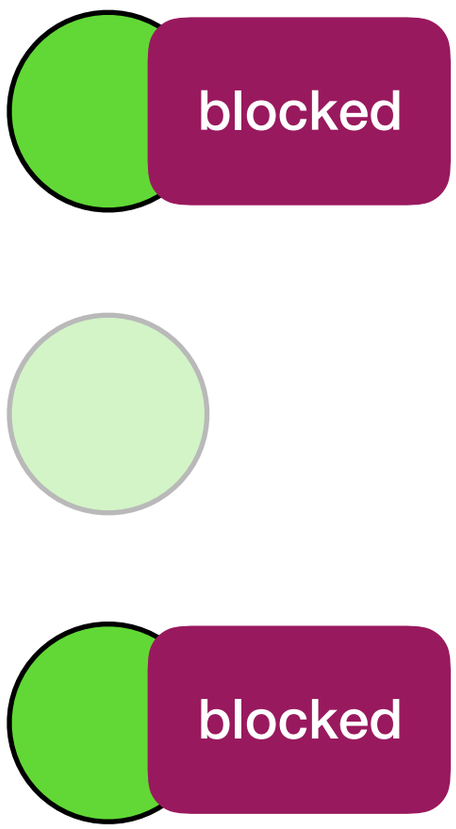




global task queue

thread pool local task queue

If all the tasks blocks, new worker thread is created.

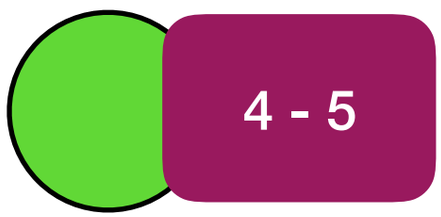


global task queue

thread pool local task queue

```
protected Integer compute() {
    // stop splitting if small enough
    if (high - low < FORK_THRESHOLD) {
      int sum = 0;
      for (int i = low; i < high; i++) {
        sum += array[i];
      }
      return sum;
    }

    int middle = (low + high) / 2;
    Summer left = new Summer(low, middle, array);
    Summer right = new Summer(middle, high, array);
    left.fork();
    right.fork();
    return right.join() + left.join();
}
```

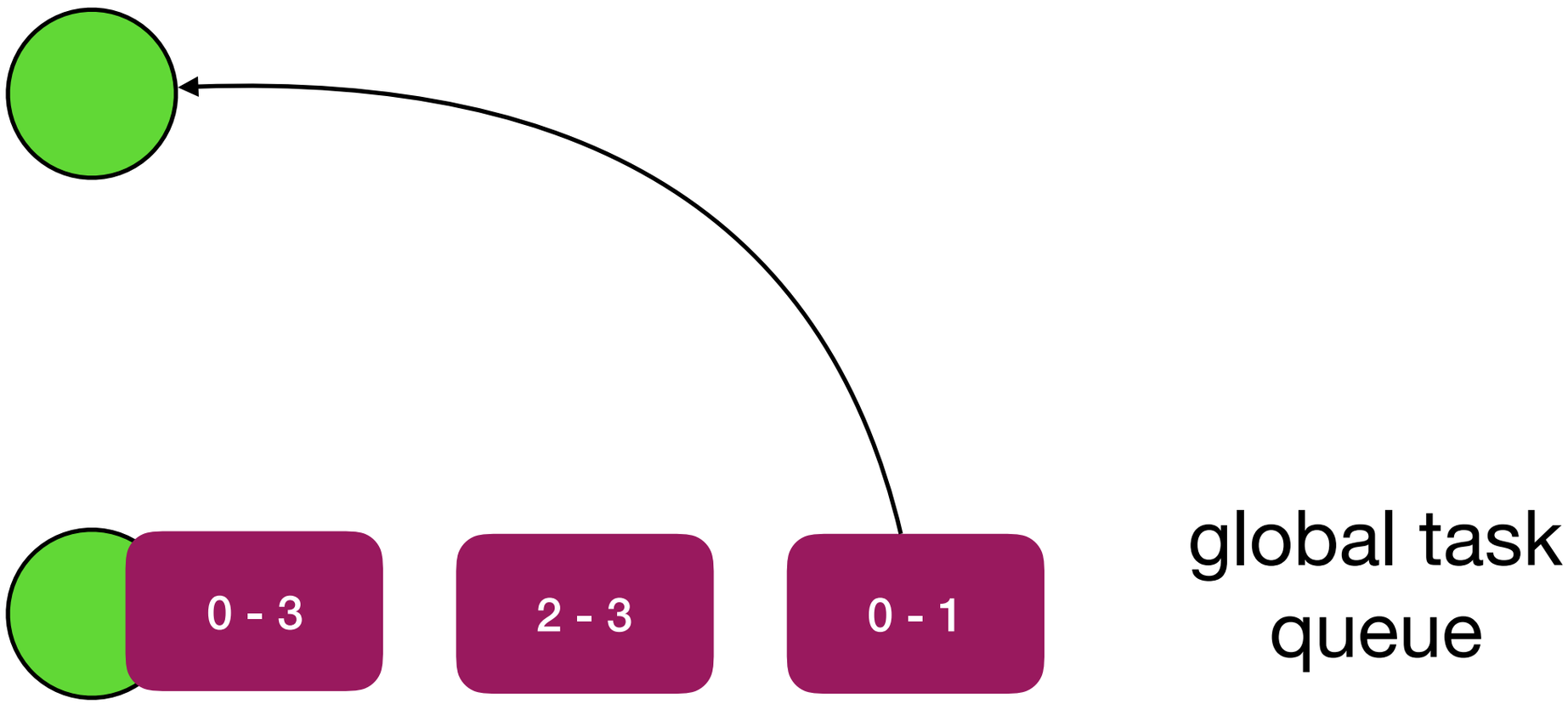
0-3 do a left.fork() then right.fork().
right goes to the head of the queue.



global task queue

thread pool local task queue

Worker 2 “steal” from the end of the queue

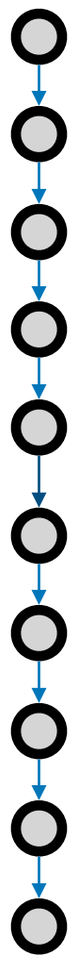


thread pool local task queue

```
protected Integer compute() {
    // stop splitting if small enough
    if (high - low < FORK_THRESHOLD) {
      int sum = 0;
      for (int i = low; i < high; i++) {
        sum += array[i];
      }
      return sum;
    }

    int middle = (low + high) / 2;
    Summer left = new Summer(low, middle, array);
    Summer right = new Summer(middle, high, array);
    left.fork();
    right.fork();
    return right.join() + left.join();
}
```

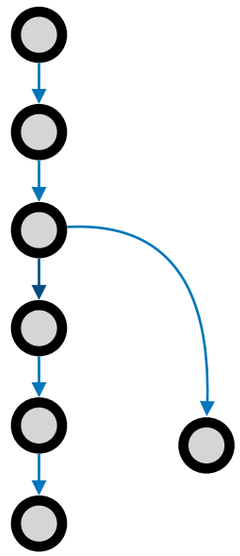
```
a.fork()
b.fork()
c.fork()
c.join()
b.join()
a.join()
```

Synchronous Call



Synchronous Call

Asynchronous Call



RecursiveAction

RecursiveTask

FutureTask

CompletableFuture

:

:

implements

Future<T>

- cancel()
- get()
- get(timeout, unit)
- isCancelled()
- isDone()


computation
fork()
isDone()
isDone()
print

```
task.fork();
while (!task.isDone()) {
    System.out.print(".");
}
System.out.print("done");
```

```
task.fork();
 if (!task.isDone()) {
      // do something
 } else {
      task.join();
 }
 if (!task.isDone()) {
      // do something else
} else {
      task.join();
}
if (!task.isDone()) {
      // do yet something else
} else {
      task.join();
```

Need a better way to coordinate async tasks

idea: go run this task, when you are done, run this other task, and then when that is done, run that task

callback: when some event happen, do something

```
CompletableFuture
    .supplyAsync(() -> doSomething())
    .thenAccept(System.out::println);
```

```
CompletableFuture
    .supplyAsync(() -> doSomething())
    .join();
```

```
CompletableFuture
    .supplyAsync(() -> doSomething())
```

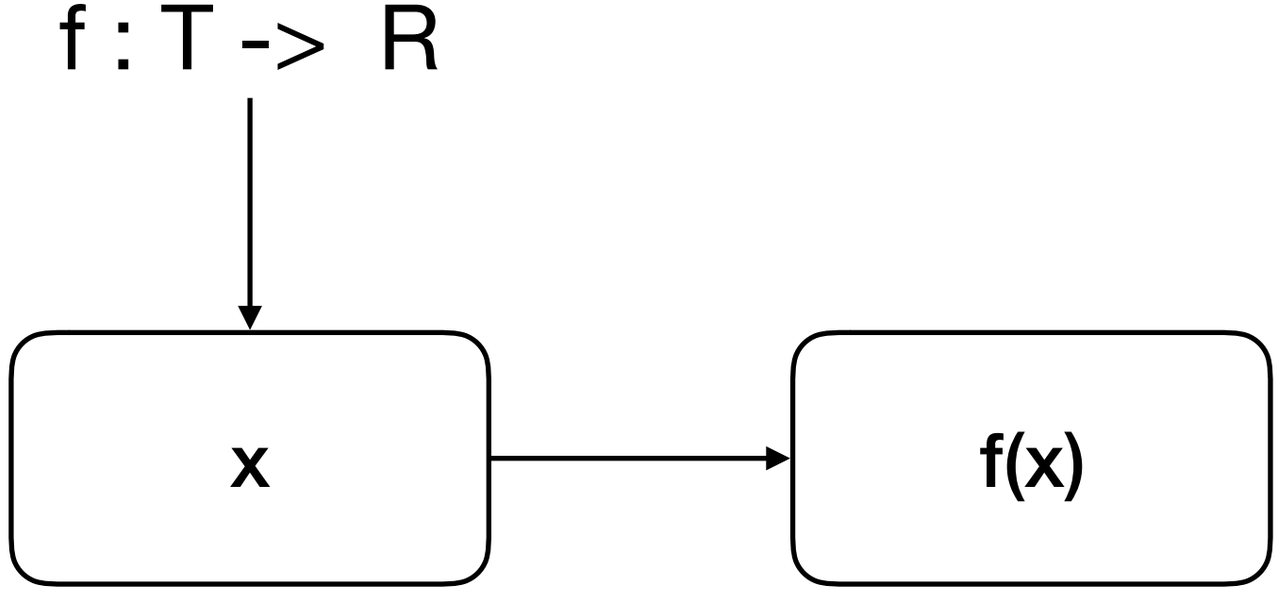
CompletableFuture is a Functor

```
<U> CompletableFuture<U> thenApply(
Function<? super T,? extends U> func)
```

and a Monad!

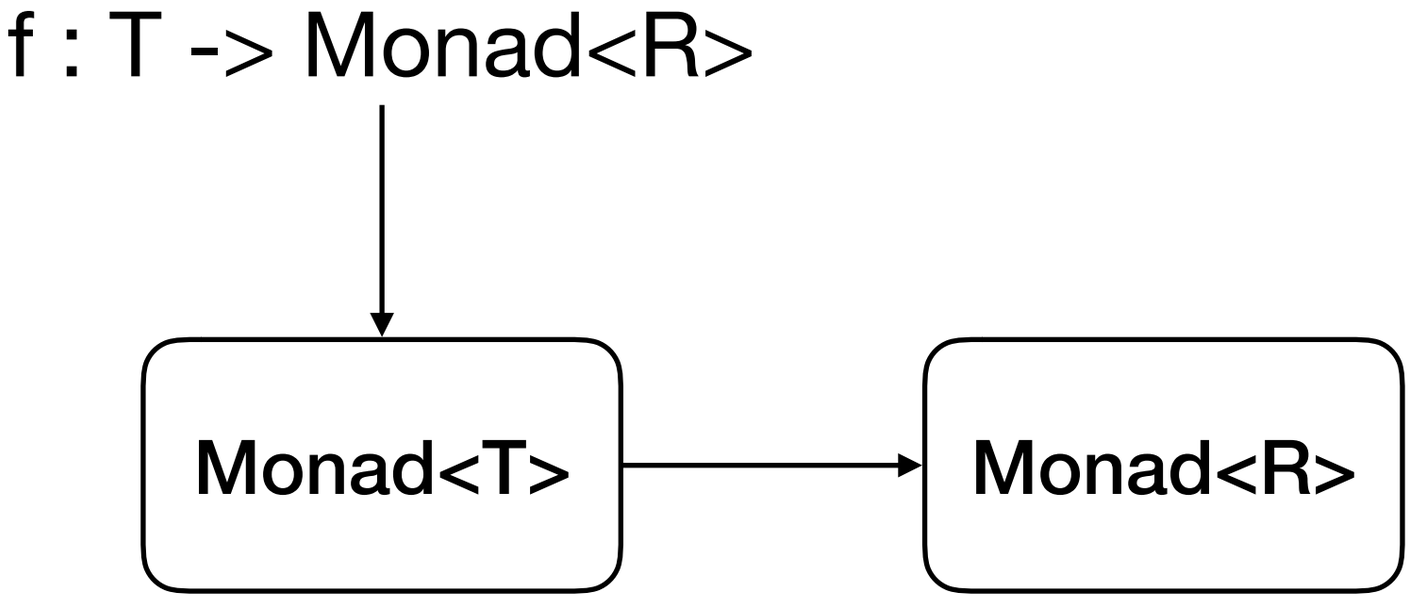
```
<U> CompletableFuture<U> thenCompose(
    Function<? super T,
    ? extends CompletionStage<U>> func)
```

“modify things in a box”



Monad

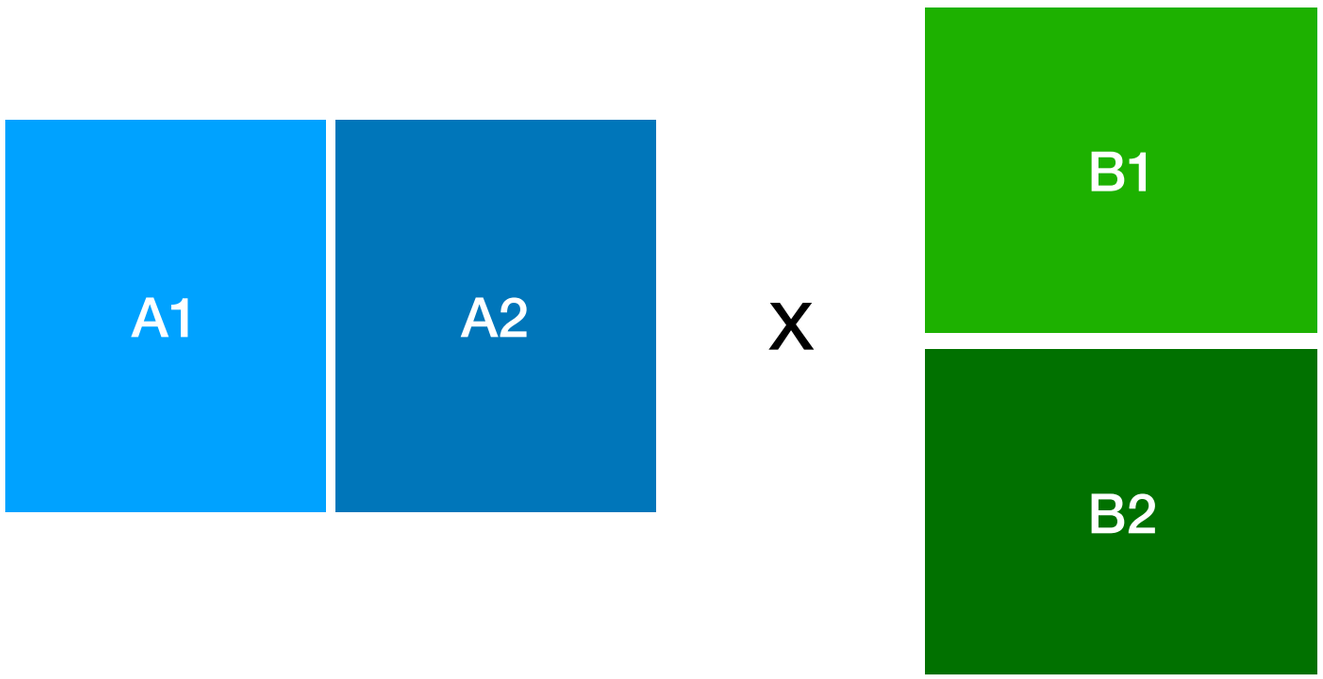
Explained in OO way

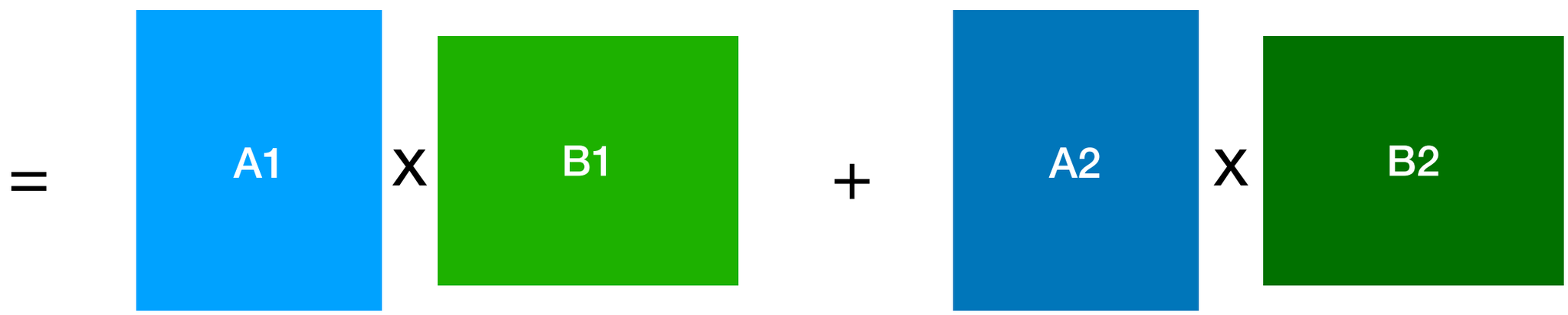


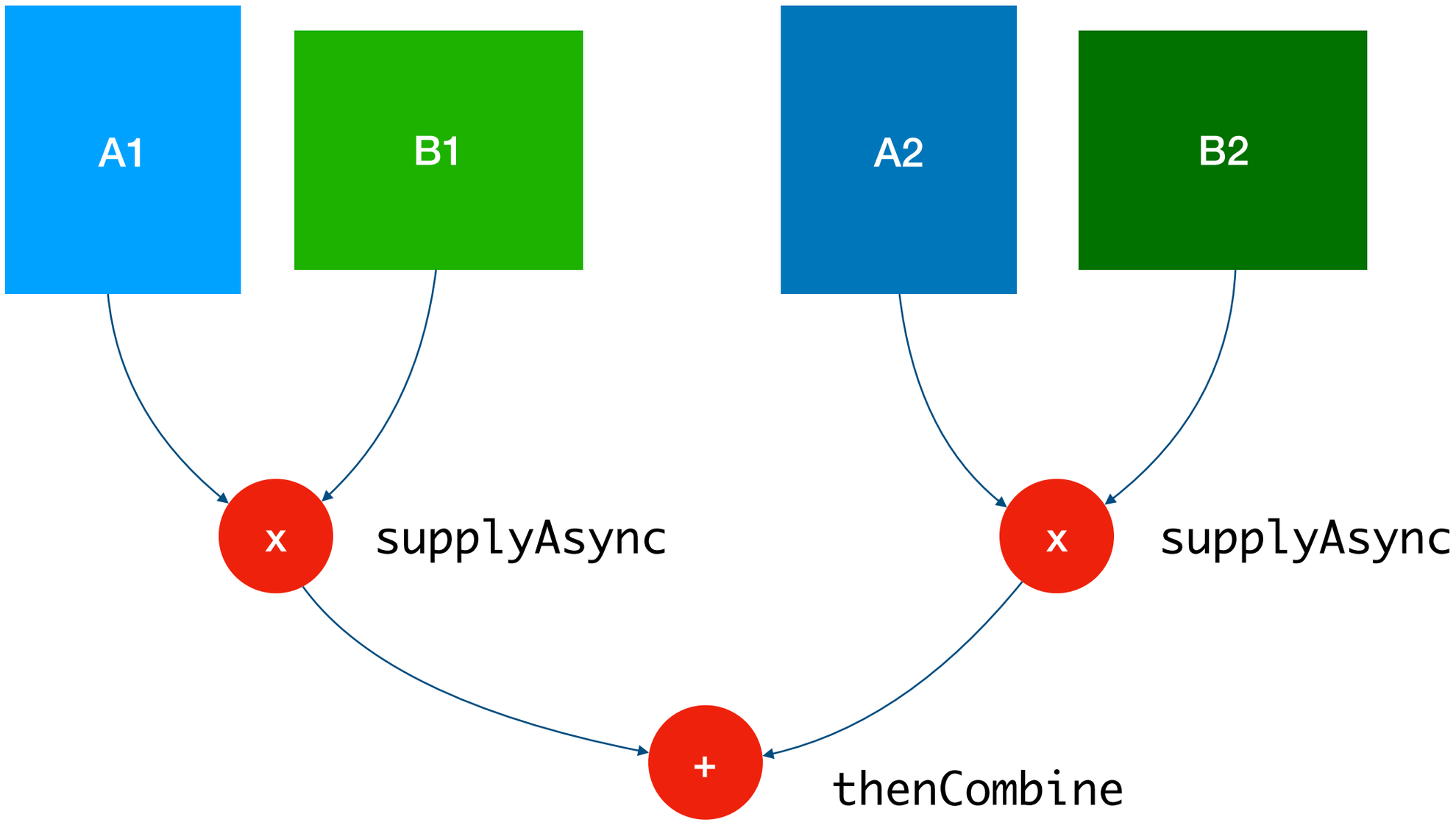
Monad

Explained in OO way

Class	Context / Box
Stream	A sequence of values
Optional	A value that maybe there
CompletableFuture	A value that will be available in the future


A1
A2
X
B1
B2


=
A1
X
B1
+
A2
X
B2


A1
B1
A2
B2
x
supplyAsync
x
supplyAsync
+
thenCombine

```
CompletableFuture left =
   .supplyAsync(() -> a1.multiply(b1));

CompletableFuture right =
.supplyAsync(() -> a2.multiply(b2))
   .thenCombine(left,
        (m1, m2) ->   m1.add(m2));
   .thenAccept(System.out::println);
```

- To create: runAsync, supplyAsync
- then<X><Y>:
    - X is Accept, Run, Combine, Compose
    - Y is nothing, Both, BothAsync, Either, EitherAsync