Parallel Programming

Parallel algorithms 1

Morgan Ericsson

Today

- Prefix sum/scan
- Sorting

Sum and scan

Remember: concurrent vs parallel

- Concurrency is about structure
- Parallelism is about execution
- For the algorithms so be beneficial
  - Parallelism to execute
  - Concurrency to structure

Computing the sum

```go
var s int
l := []int{3, 5, 2, 5, 7, 9, 4, 6}

for _, val := range l {
  s += val
}

fmt.Println("sum:", s)
```

Computing the sum

- Remember the algorithms course
- 8 add operations
- $O(N)$
- Can we do it in fewer with parallelism?

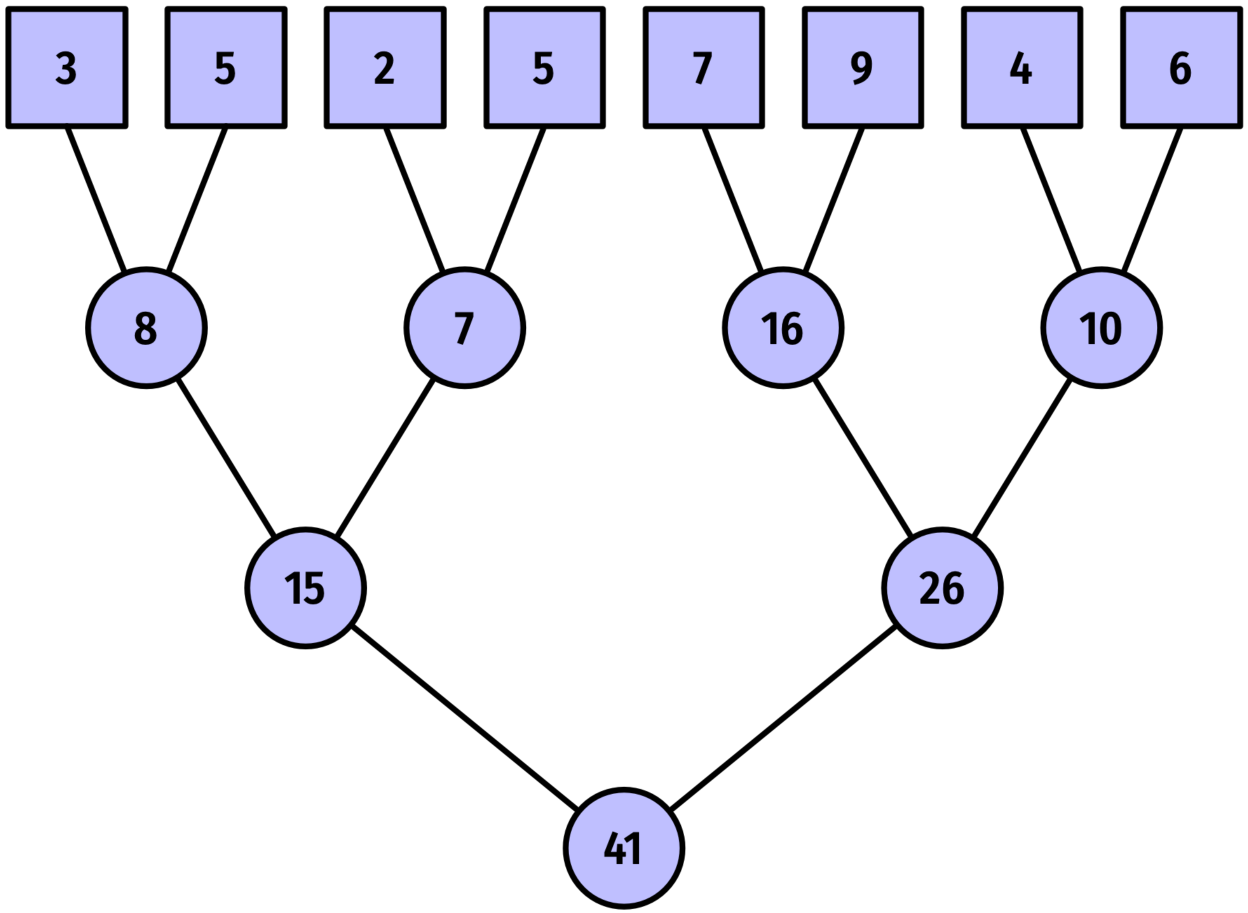
Aside: Not just the sum

- The previous example is a pattern
- We can use any binary commutative operation
  - sum, product, max, ...

Aside: Not just the sum

```go
s := 1
m := math.MinInt
l := []int{3, 5, 2, 5, 7, 9, 4, 6}

for _, val := range l {
  s *= val
  m = max(m, val)
}

fmt.Printf("prod: %d\nmax: %d", s, m)
```

How many operations do we need?



Aside: PRAM

- Parallel Random Access Machine
- Shared-memory multiprocessor model
- Unlimited number of processor with unlimited local and shared memory
- Each processor knows its ID
- Inputs and outputs are placed in shared memory
- Each instruction takes unit time
- Instructions are synchronised across processors

PRAM

- Unfeasible model
  - The interconnection network between processors and memory would require a very large area
  - The message routing would require time proportional to the network size
- Theoretical model
- Algorithm designers can forget about communication and focus on the parallel computation

Implementation

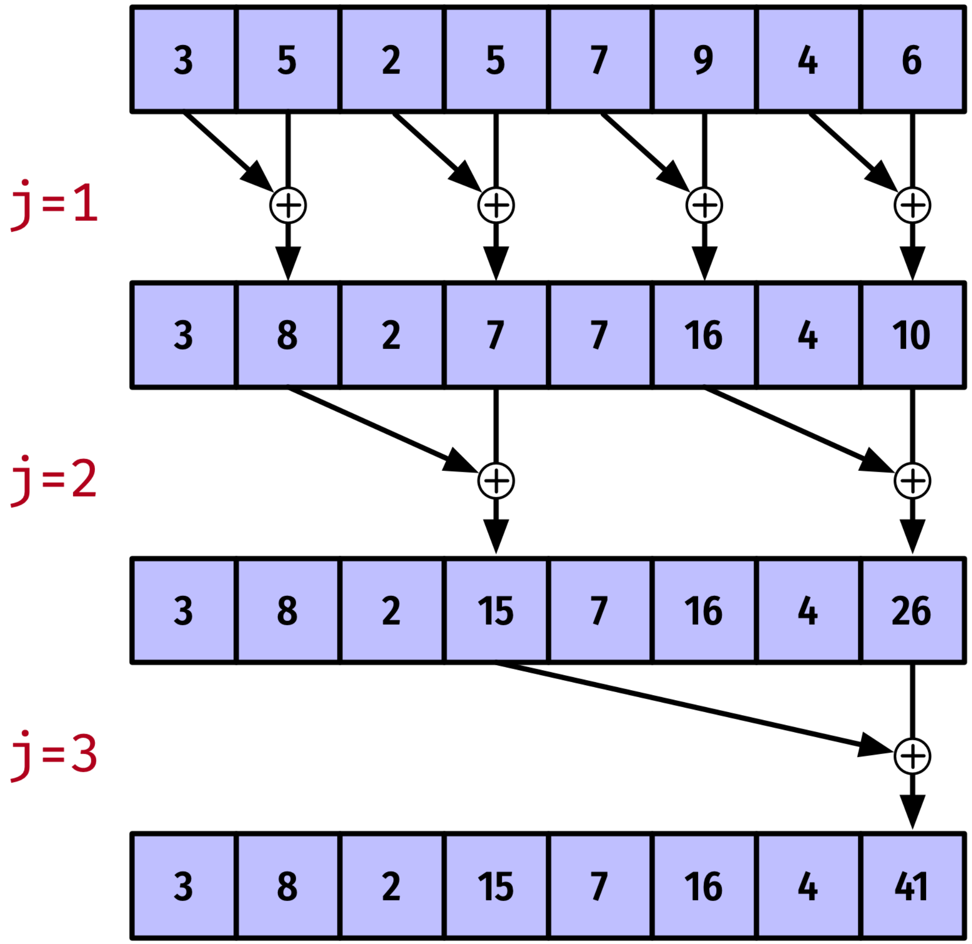
```go
l := []int{3, 5, 2, 5, 7, 9, 4, 6}
tmp := int(math.Log2(float64(len(l))))

for j := 1; j < tmp+1; j++ {
  for k := 0; k < len(l); k++ {
    if (k+1)%(1<<j) == 0 {
      l[k] = l[k-1<<(j-1)] + l[k]
    }
  }
}
```

```
l: [3 8 2 15 7 16 4 41]
sum: 41
```

<< and >>

- Shifts the bits in a number left or right a number of times
  - `a := 1 ; a <<= 3`
  - `0001` shifted three times to the left becomes `1000`
- Can be used to implement multiplication and division by two on integers
- Used here since explicit type conversion quickly becomes unreadable

Implementation



Implementation (makes little sense)

```go
var wg sync.WaitGroup
l := []int{3, 5, 2, 5, 7, 9, 4, 6}
tmp := int(math.Log2(float64(len(l))))

for j := 1; j < tmp+1; j++ {
  for k := 0; k < len(l); k++ {
    wg.Add(1)
    go func(x, y int) {
      defer wg.Done()
      if (x+1)%(1<<y) == 0 {
        l[x] = l[x-1<<(y-1)] + l[x]
      }
    }(k, j)
  }
}
wg.Wait()
```

Many issues

- We assume one processor per element
  - So 8 processors for 8 elements
- We cover races/sync by starting and stopping threads
  - This is equivalent to a barrier
  - but slower...

A more practical approach

1. Split the array into one part per thread (fork)
2. Compute the sum for each part
3. Sum the sums of the parts (join)

A more practical approach

```go
func summation(lst []int, wid int, numworks int, out chan<- int) {
    beg := len(lst) / numworks * wid
    end := len(lst) / numworks * (wid + 1)

    if wid == numworks - 1 {
        end = len(lst)
    }

    var ls int
    for i := beg; i < end; i++ {
        ls += lst[i]
    }

    out <- ls
}
```

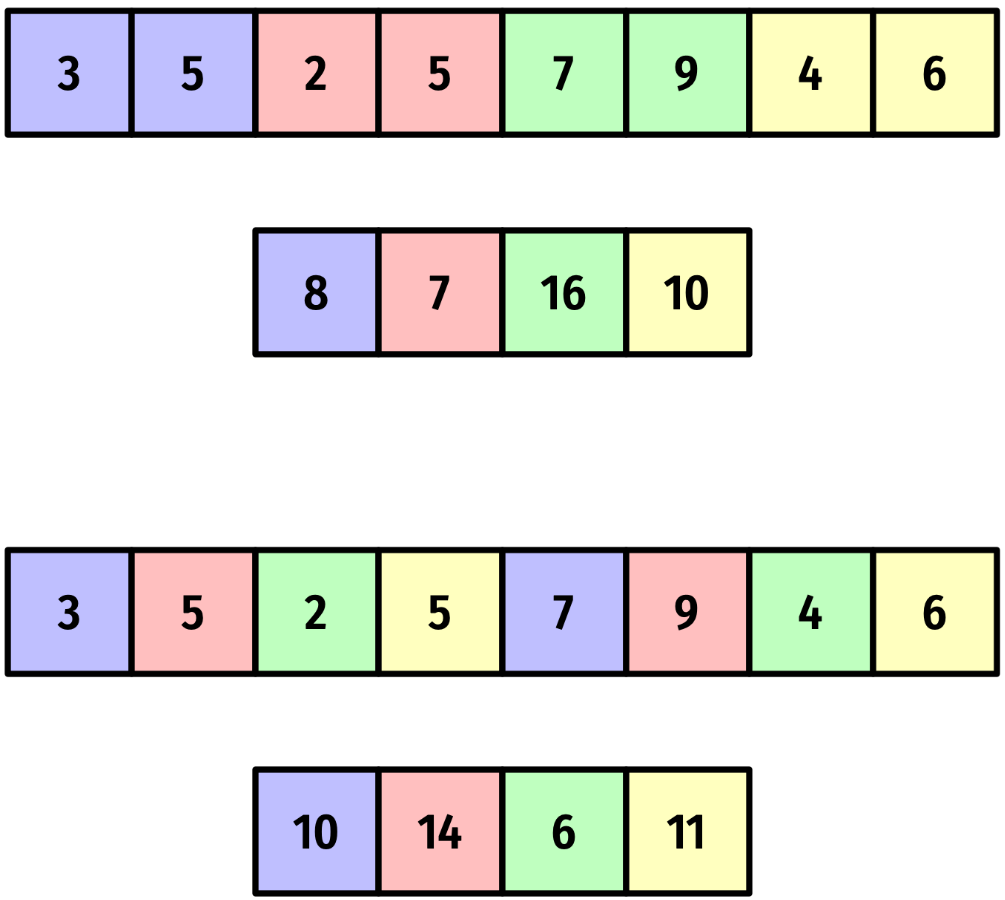
A more practical approach

```go
func main() {
  ch := make(chan int)
  l := make([]int, 10000)
  var tmp int
  for i := 0; i < 10000; i++ {
    l[i] = i
  }

  for i := 0; i < 8; i++ {
    go summation(l, i, 8, ch)
  }

  var gs int
  for i := 0; i < 8; i++ {
    gs += <-ch
  }
}
```

So much setup in summation!

```go
func summation2(lst []int, wid int, numworks int, out chan<- int) {
    var ls int
    for i := wid; i < len(lst); i += numworks {
        ls += lst[i]
    }

    out <- ls
}
```

Difference?



Prefix scan

- Computes all partial sums of a vector of values
- Inclusive and exclusive
  - Exclusive does not include “current” value
- Just as parallel sum, any associative combining operation can be used

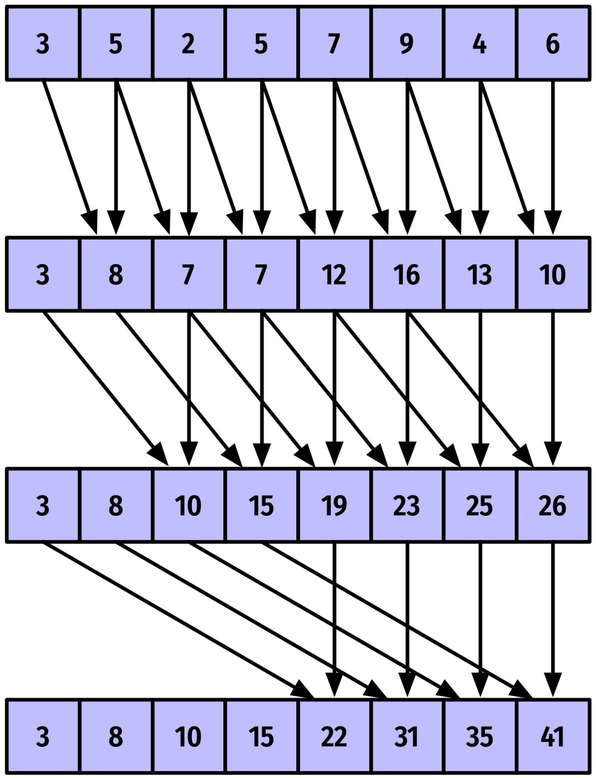
Prefix scan

Original	3	5	2	5	7	9	4	6
Inclusive	3	8	10	15	22	31	35	41
Exclusive	0	3	8	10	15	22	31	35

Again, simple serial

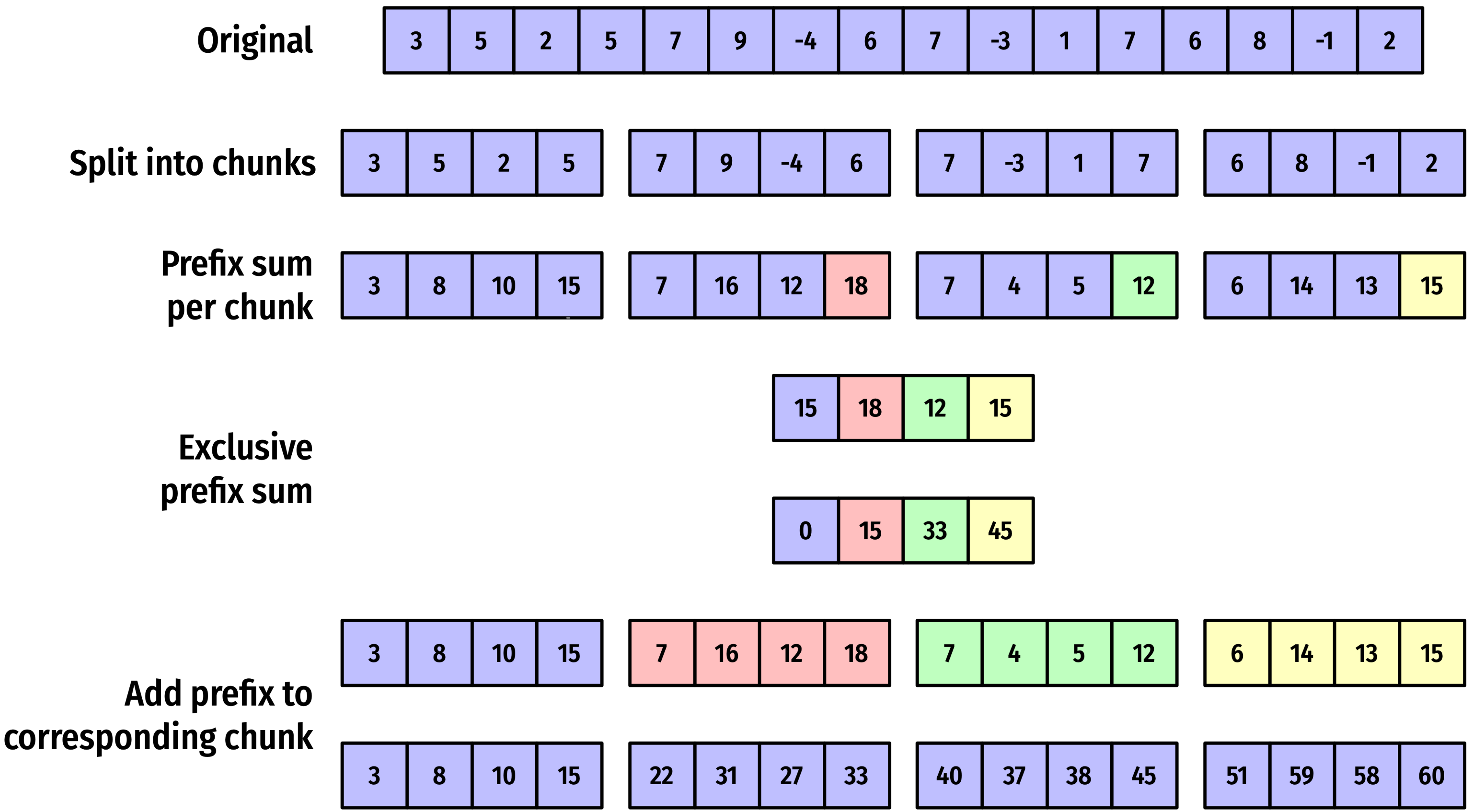
```go
lst := []int{3, 5, 2, 5, 7, 9, 4, 6}
ps := make([]int, len(lst))

ps[0] = lst[0]
for i := 1; i < len(lst); i++ {
  ps[i] = ps[i-1] + lst[i]
}
```

Unrealistic parallel version




```go
ll := int(math.log2(float64(len(l))))
for j:=1;j<ll+1;j++ {
  // This should run on
  // k processors
  for k:=0;k<len(l);k++ {
    go func(k int) {
      if k >= (1<<j - 1) {
        l[k] = l[k] + \
               l[k - (1<<j-1)]
      }
    }(k)
  }
}
```

A practical approach



Prefix scan

```go
func pscan(lst *[]int, wid int, ws int, sz int, ch chan int) {
    beg := sz / ws * wid
    end := sz / ws * (wid + 1)
    if wid == ws-1 {
        end = sz
    }

    for i := beg + 1; i < end; i++ {
        (*lst)[i] = (*lst)[i-1] + (*lst)[i]
    }

    ch <- (*lst)[end-1]
    tmp := <-ch

    for i := beg; i < end; i++ {
        (*lst)[i] = (*lst)[i] + tmp
    }
}
```

Main

```
l := []int{3, 5, 2, 6, 7, 9, -4, 6, 7, -3, 1, 7, 6, 8, -1, 2}
chs := make([]chan int, 4)
var wg sync.WaitGroup

for i := 0; i < 4; i++ {
    chs[i] = make(chan int)
    wg.Add(1)
    go func(ch chan int) {
        defer wg.Done()
        pscan(&l, i, 4, len(l), ch)
    }(chs[i])
}

var rc int
for i := 0; i < 4; i++ {
    tmp := <-chs[i]
    chs[i] <- rc
    rc += tmp
}

wg.Wait()
```

Sorting

Remember Bubble sort?

We can do this in parallel!

	0	1	2	3	4	5	6	7	8	
Iteration 0	1	4	8	2	5	9	3	7	6	odd
Iteration 1	1	4	8	2	5	3	9	7	6	even
Iteration 2	1	4	2	8	3	5	7	9	6	odd
Iteration 3	1	2	4	3	8	5	7	6	9	even
Iteration 4	1	2	3	4	5	8	6	7	9	odd
Iteration 5	1	2	3	4	5	6	8	7	9	even
Iteration 6	1	2	3	4	5	6	7	8	9	odd

Odd-even transposition sort (serial)

```
l := []int{3, 5, 2, 6, 7, 9, -4, 6, 7, -3, 1, 7, 6, 8, -1, 2}
exch := true
var offs int

for exch {
    exch = false
    for i := 1 + offs; i < len(l); i += 2 {
        if l[i-1] > l[i] {
            l[i-1], l[i] = l[i], l[i-1]
            exch = true
        }
    }
    offs = (offs + 1) % 2
}
```

Parallel version

- Split the array into *N* chunks (where *N* is the number of threads)
- Sorting on each chunk in parallel
- Challenges
  - Did we swap? Shared
  - The two phases (odd/even) should not overlap

Parallel version

```
func oets(lst *[]int, wid int, ws int, offs *int, exch *bool) {
    beg := len(*lst) / ws * wid
    end := (len(*lst) / ws * (wid + 1)) + 1
    if wid == ws-1 {
        end = len(*lst)
    }
    for exch {
        exch = false
        for i := beg + 1 + offs; i < end; i += 2 {
            if (*lst)[i-1] > (*lst)[i] {
                (*lst)[i-1], (*lst)[i] = (*lst)[i], (*lst)[i-1]
                exch = true
            }
        }
        ch <- exch
        exch = <-ch
        offs = (offs + 1) % 2
    }
}
```

Fails

- Problem with shared variables? Yes
  - Assignment in Python is atomic
  - All write the same value, so no race with flag
  - But offset is problematic!
- The threads can (and will) get out of sync
  - Some threads might not even enter the while-loop

Second parallel version

```go
func oets(lst *[]int, wid int, ws int, ch chan bool) {
    exch := true
    var offs int
    beg := len(*lst) / ws * wid
    end := len(*lst) / ws * (wid + 1)
    if wid == ws-1 {
        end = len(*lst)
    }
    for exch {
        exch = false
        for i := beg + 1 + offs; i < end; i += 2 {
            if (*lst)[i-1] > (*lst)[i] {
                (*lst)[i-1], (*lst)[i] = (*lst)[i], (*lst)[i-1]
                exch = true
            }
        }
        ch <- exch
        exch = <-ch
        offs = (offs + 1) % 2
    }
}
```

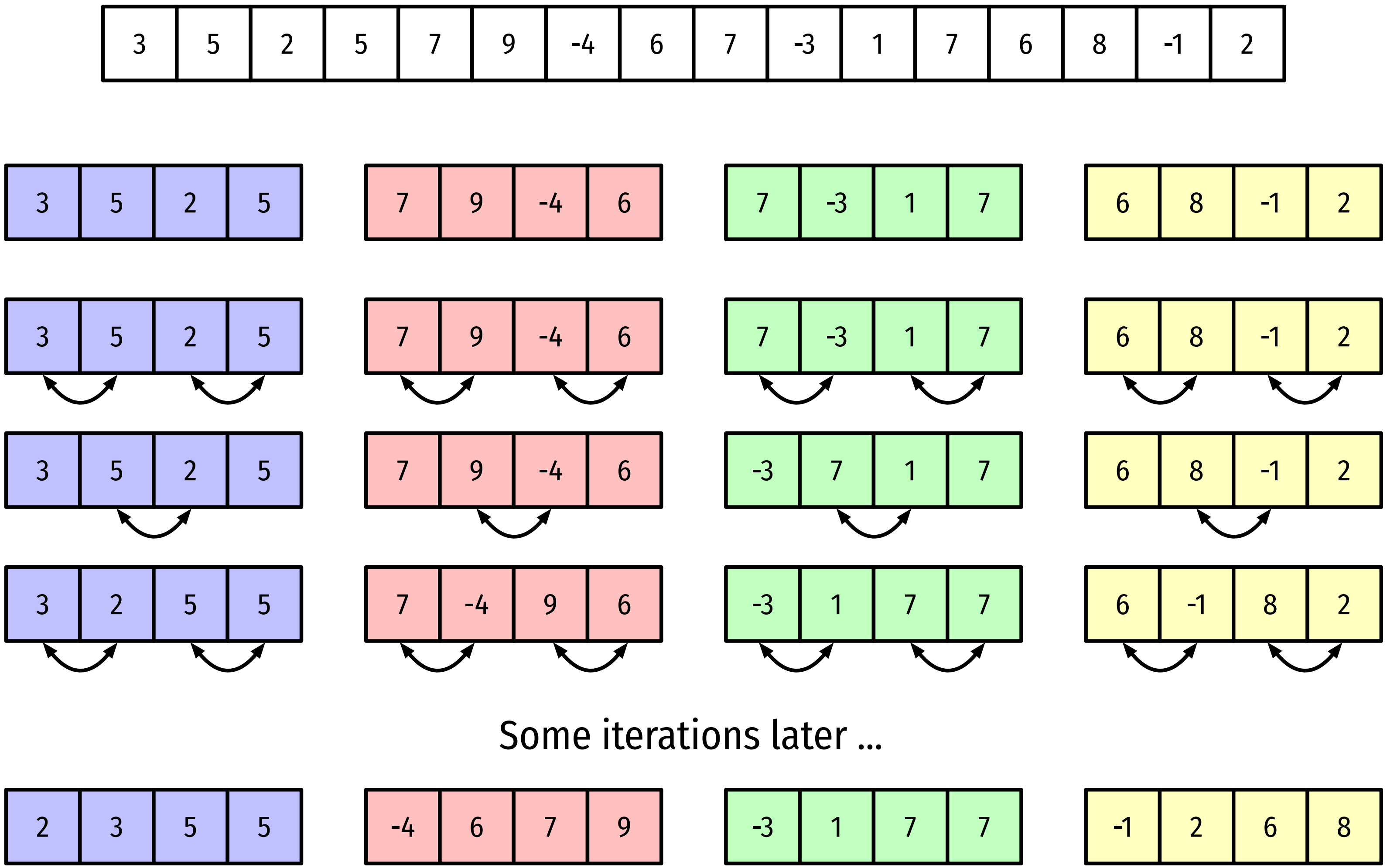
Second parallel version

```go
l := []int{3, 5, 2, 5, 7, 9, -4, 6, 7, -3, 1, 7, 6, 8, -1, 2}
chs := make([]chan bool, 4)

for i := 0; i < 4; i++ {
    chs[i] = make(chan bool)
    go func(ch chan bool) {
        oets(&l, i, 4, ch)
    }(chs[i])
}

// ...
```

Second parallel version

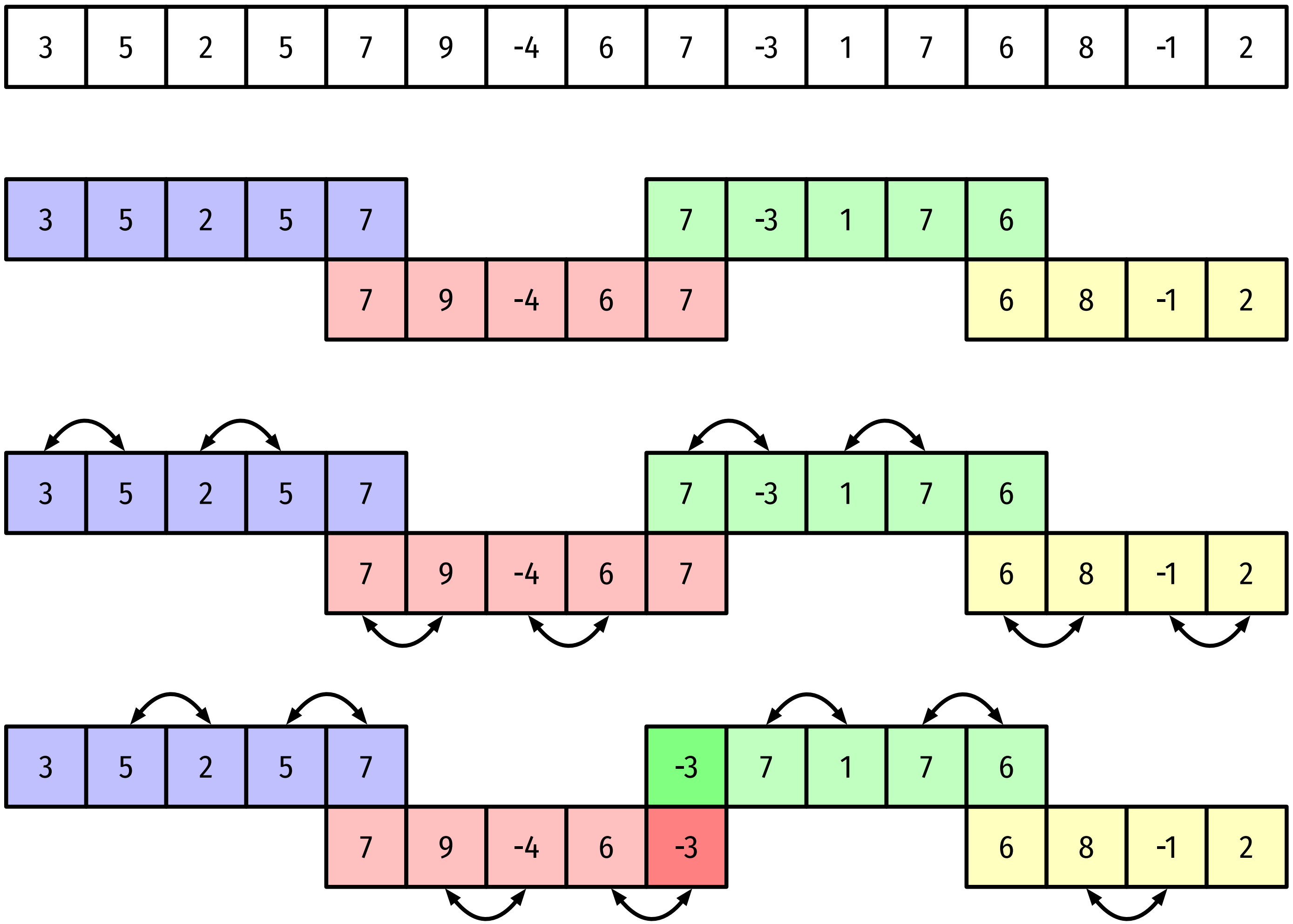
```go
// ...
for {
    var gexch bool
    for i := 0; i < 4; i++ {
        if tmp := <-chs[i]; tmp {
            gexch = true
        }
    }
    if gexch {
        for i := 0; i < 4; i++ {
            chs[i] <- gexch
        }
    } else {
        break
    }
}
```

Still fails



3 5 2 5 7 9 -4 6 7 -3 1 7 6 8 -1 2
3 5 2 5 | 7 9 -4 6 | 7 -3 1 7 | 6 8 -1 2
3 5 2 5 | 7 9 -4 6 | 7 -3 1 7 | 6 8 -1 2
3 5 2 5 | 7 9 -4 6 | -3 7 1 7 | 6 8 -1 2
3 2 5 5 | 7 -4 9 6 | -3 1 7 7 | 6 -1 8 2
Some iterations later ...
2 3 5 5 | -4 6 7 9 | -3 1 7 7 | -1 2 6 8

Solution



3 5 2 5 7 9 -4 6 7 -3 1 7 6 8 -1 2
3 5 2 5 7
7 9 -4 6 7
7 -3 1 7 6
6 8 -1 2
3 5 2 5 7
7 9 -4 6 7
7 -3 1 7 6
6 8 -1 2
3 5 2 5 7
7 9 -4 6 -3
-3 7 1 7 6
6 8 -1 2

Solution

```
func oets(lst *[]int, wid int, ws int, ch chan bool) {
    exch := true
    var offs int
    beg := len(*lst) / ws * wid
    end := (len(*lst) / ws * (wid + 1) + )
    if wid == ws-1 {
        end = len(*lst)
    }
    for exch {
        exch = false
        for i := beg + 1 + offs; i < end; i += 2 {
            if (*lst)[i-1] > (*lst)[i] {
                (*lst)[i-1], (*lst)[i] = (*lst)[i], (*lst)[i-1]
                exch = true
            }
        }
        ch <- exch
        exch = <-ch
        offs = (offs + 1) % 2
    }
}
```

Remember Quicksort

```python
def _sort(a:list[int], lo:int, hi:int) -> None:
  if hi <= lo:
    return
  j = _partition(a, lo, hi)
  _sort(a, lo, j - 1)
  _sort(a, j + 1, hi)

def sort(a:list[int]) -> None:
  _sort(a, 0, len(a) - 1)
```

Remember Quicksort

» We will do this one in Python

» To see how things work in Python

» And to leave room for you to implement it in Go!

Should be easy to parallelize?

```
def _sort(a:list[int], lo:int, hi:int) -> None:
  if hi <= lo:
    return
  j = self._partition(a, lo, hi)
  t1 = Thread(target=_sort, args=(a, lo, j - 1))
  t2 = Thread(target=_sort, args=(a, j + 1, hi))
  t1.start()
  t2.start()
  t1.join()
  t2.join()
```

Terrible approach

- How many threads will be created? Many!
- Why is that a problem?
  - Limit the number of threads
  - High overhead from creation
- Generally more difficult to deal with recursive algorithms

Make quicksort iterative

```python
from queue import Queue

class IQS:
  def __init__(self):
    self.Q = Queue()

  def _sort(self):
    while not self.Q.empty():
      lo, hi = self.Q.get()
      if lo < hi:
        mid = self._partition(lo, hi)  # Unchanged

        self.Q.put((lo, mid - 1))
        self.Q.put((mid + 1, hi))
```

Make quicksort iterative

```
def sort(self:IQS, lst):
  self.lst = lst
  self.Q.put((0, len(self.lst) - 1))
  self._sort()
```

Easier to parallelize?

- We use a queue instead of recursion
- Each call to partition is independent, so ...
- ... we can run them in parallel
- Rather than spawning threads for the calls, ...
- ... existing threads grab “jobs” from the queue
- Remember, channels in Go are essentially concurrent queues!

New `_sort`

```
def _sort(self):
  while True:
    self.qsem.acquire():
    lo, hi = self.Q.get()
    if lo < hi:
      mid = self._partition(lo, hi)
      self.Q.put((lo, mid - 1))
      self.Q.put((mid + 1, hi))
      self.qsem.release(2)
```

New `sort`

```python
def sort(self, l):
  self.lst = l
  self.Q.put((0, len(self.lst) - 1))
  self.qsem.release()
  ts = [Thread(target=self._sort)
        for _ in range(self.nt)]
  for t in ts:
    t.start()
  for t in ts:
    t.join()
```

New `init`

```python
def __init__(self):
    self.Q = Queue()
    self.qsem = Semaphore(0)
```

Unfortunately ...

- ... it is a bit more complicated
- How do we know when we are done?
- We should not make timing assumptions
  - e.g., end when there is nothing in the queue for a while
- So, we need a flag ...
- ... that is set once all elements are sorted

Adding a count

```
def _sort(self):
  while not self.done:
    self.qsem.acquire():
    lo, hi = self.Q.get()
    if lo < hi:
      mid = self._partition(lo, hi)
      self.sortcnt += 1
      self.Q.put((lo, mid - 1))
      self.Q.put((mid + 1, hi))
      self.qsem.release(2)
    if lo == hi:
      self.sortcnt += 1
      if self.sortcnt == self.N:
        self.done = True
```

Problem 1

```
def _sort(self):
  while not self.done:
    self.qsem.acquire():
    lo, hi = self.Q.get()
    if lo < hi:
      mid = self._partition(lo, hi)
      self.sortcnt += 1
      self.Q.put((lo, mid - 1))
      self.Q.put((mid + 1, hi))
      self.qsem.release(2)
    if lo == hi:
      self.sortcnt += 1
      if self.sortcnt == self.N:
        self.done = True
```

Problem 1

```
self.cntlock.acquire()
self.sortcnt += 1
self.cntlock.release()
```

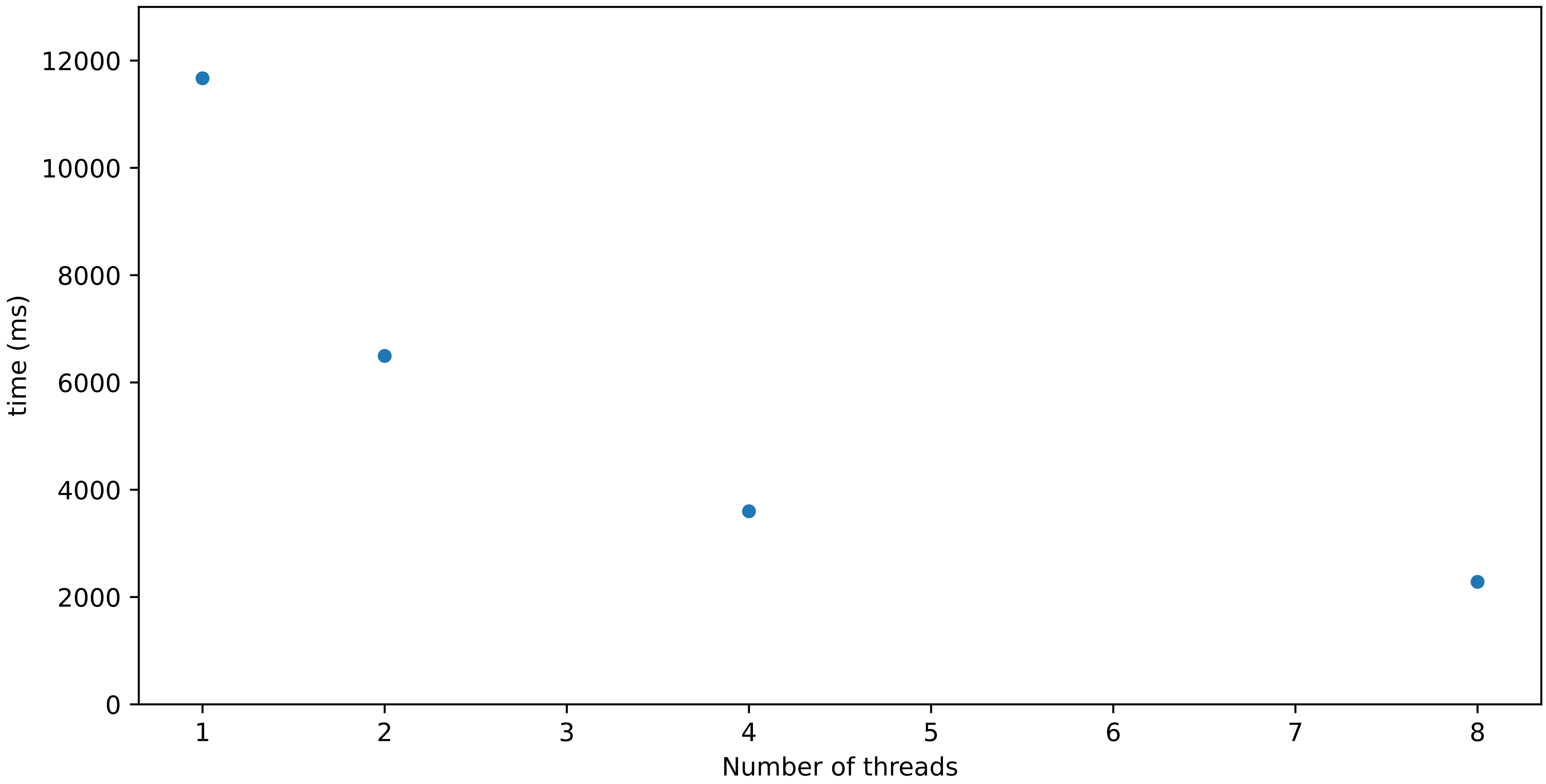
Problem 2

```
def _sort(self):
  while not self.done:
    self.qsem.acquire():
    lo, hi = self.Q.get()
    if lo < hi:
      mid = self._partition(lo, hi)
      self.sortcnt += 1
      self.Q.put((lo, mid - 1))
      self.Q.put((mid + 1, hi))
      self.qsem.release(2)
    if lo == hi:
      self.sortcnt += 1
      if self.sortcnt == self.N:
        self.done = True
```

Problem 2

```python
while True:
  while not self.qsem.acquire(timeout=10):
    if self.done:
      return
```

Scaling (Java version)



What did we do?

- N threads and a queue
- Basically an executor service

So, rewrite?

```python
class CCQsort:
  def __init__(self, nt):
    self.exec = ThreadPoolExecutor(nt)

  def sort(self, l):
    self.lst = l
    r = self.exec.submit(self._sort, 0, len(l) - 1)

  def _sort(self, lo, hi):
    if lo < hi:
      mid = self._partition(lo, hi)
      r1 = self.exec.submit(self._sort, lo, mid - 1)
      r2 = self.exec.submit(self._sort, mid + 1, hi)
```

Does not work

- Submit is not blocking, so when `sort(lst)` returns, the list is not sorted
- We need to get sort to block ...
- Or get something we can wait for ...

We had the same problem before!

```python
def _sort(self, lo, hi):
  if lo < hi:
    mid = self._partition(lo, hi)
    self.tl.acquire()
    self.tc += 1
    self.tl.release()
    self.exec.submit(self._sort, lo, mid - 1)
    self.exec.submit(self._sort, mid + 1, hi)
  elif lo == hi:
    self.tl.acquire()
    self.tc += 1
    tmp = self.tc
    self.tl.release()
    if tmp == self.N:
      self.tsig.set()
```

We had the same problem before!

```python
def sort(self, l):
    self.lst = l
    self.N = len(l)
    self.exec.submit(self._sort, 0, len(l) - 1)
    self.tsig.wait()
```

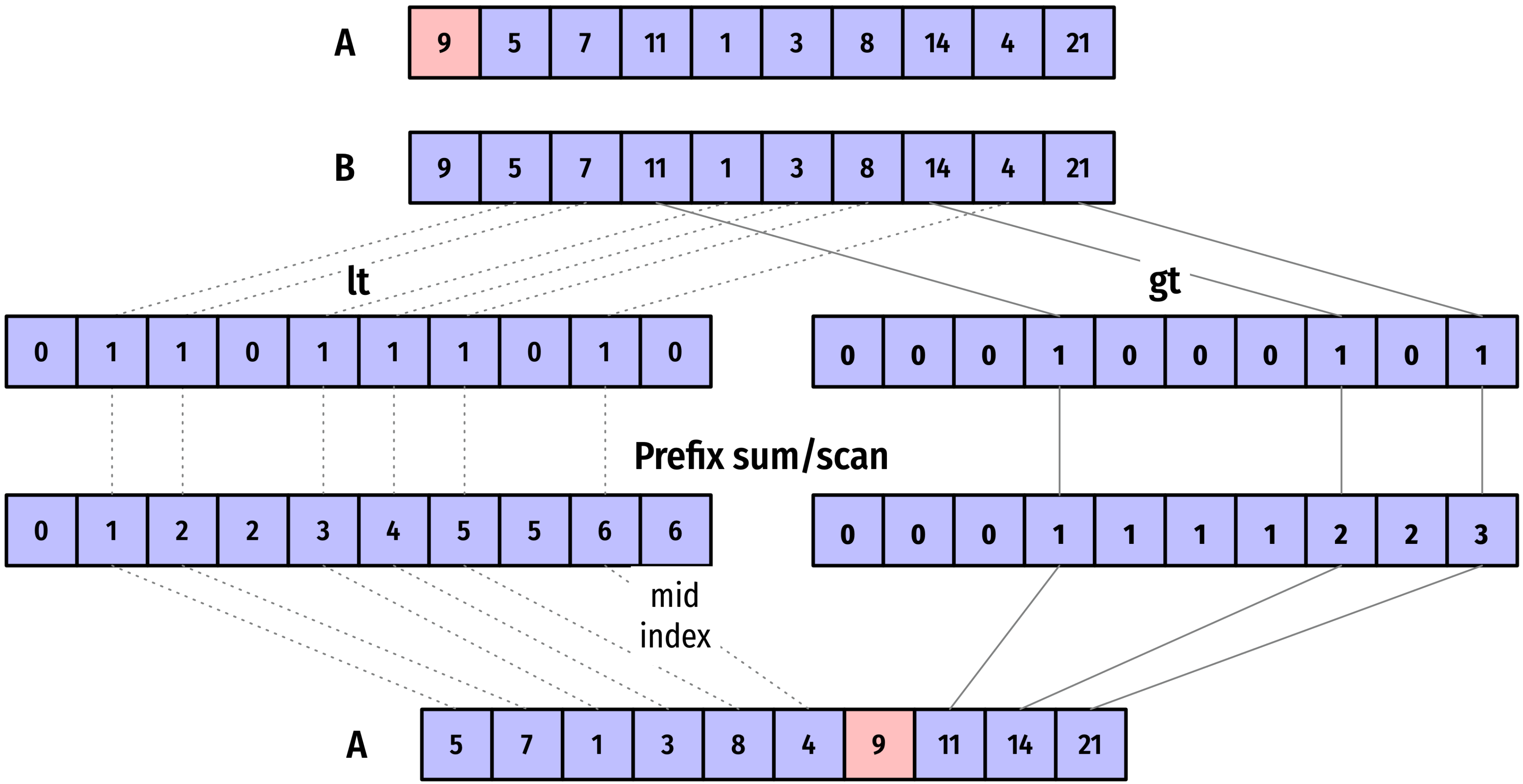
Signal/Event

- We can replace the signal/Event with a lock
- Aquire/lock when created
- Try to lock in main thread (which will block)
- Unlock when done (so main thread can lock)

What about partition?

- Can we partition concurrently?
- Single pivot, need to move “all” elements
- So, need more than split into N parts and process

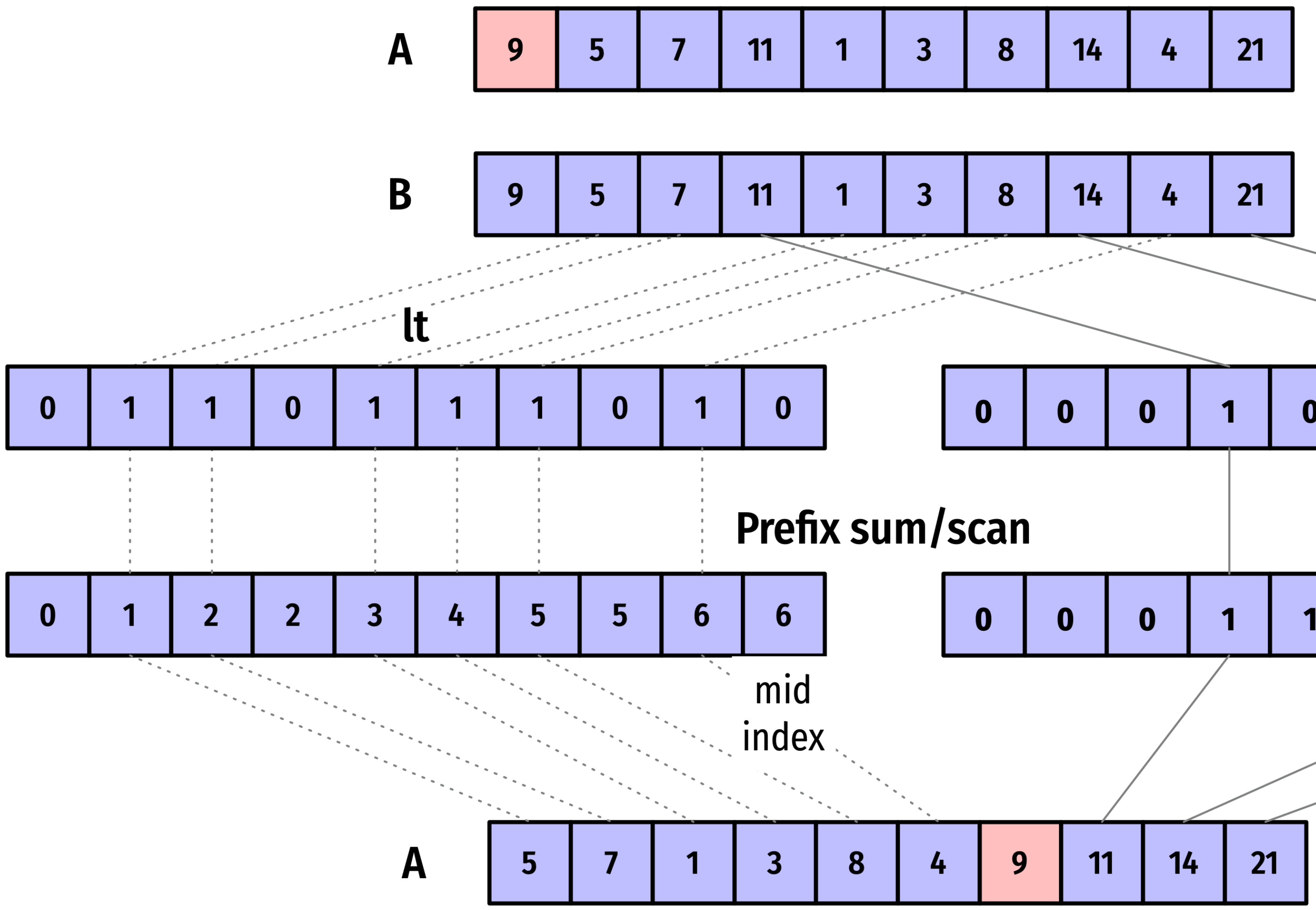
What about partition?



What about partition? (serial)

```
def ppartition(A, x):
  B, lt, gt = [0] * len(A), [0] * len(A), [0] * len(A)

  for i in range(len(A)):
    B[i] = lst[i]
    if B[i] < x: lt[i] = 1
    if B[i] > x: gt[i] = 1

  lt = prefixsum(lt)
  gt = prefixsum(gt)
  k = lt[n - 1]
  lst[k] = x

  for i in range(n):
    if B[i] < x:
      lst[lt[i] - 1] = B[i]
    elif B[i] > x:
      lst[k + gt[i]] = B[i]

  return k
```

What about partition?



What about partition?

- We can make the two for loops parallel
  - Chunk or different offsets
- We can use the parallel prefix scan
- Need a few synchronization points with barriers

Summary (part 1)

New designs/tools

- We can implement parallel for-loops by
  - Chunking
  - Offset based on thread id
- Barriers are useful to synchronize the execution

New designs/tools

- Locks, semaphores or events can be used to signal threads / control execution
- Thread id can also be used if something should be serial
  - Best used in combination with the other tools

Common building blocks

- Reusable building blocks
  - Prefix sum/scan
  - “Parallel for” (fork/join)
- Can be annoying to interleave sync
- Can be slower without interleaving
  - Since more syncronization, which has overhead
- Lack of barriers in Go is annoying!

Next time

- More algorithms
  - Searching
  - Graphs