Group Synchronisation and Parallel Go

DD1396
Ric Glassey
glassey@kth.se

Concepts

Synchronising goroutines

Mapreduce and concurrent design concerns

Synchronising Goroutines

Synchronising goroutines

Coordinating Go routines (so far...)

- waiting for user input
- waiting for an signal on a channel

Demo: Matching

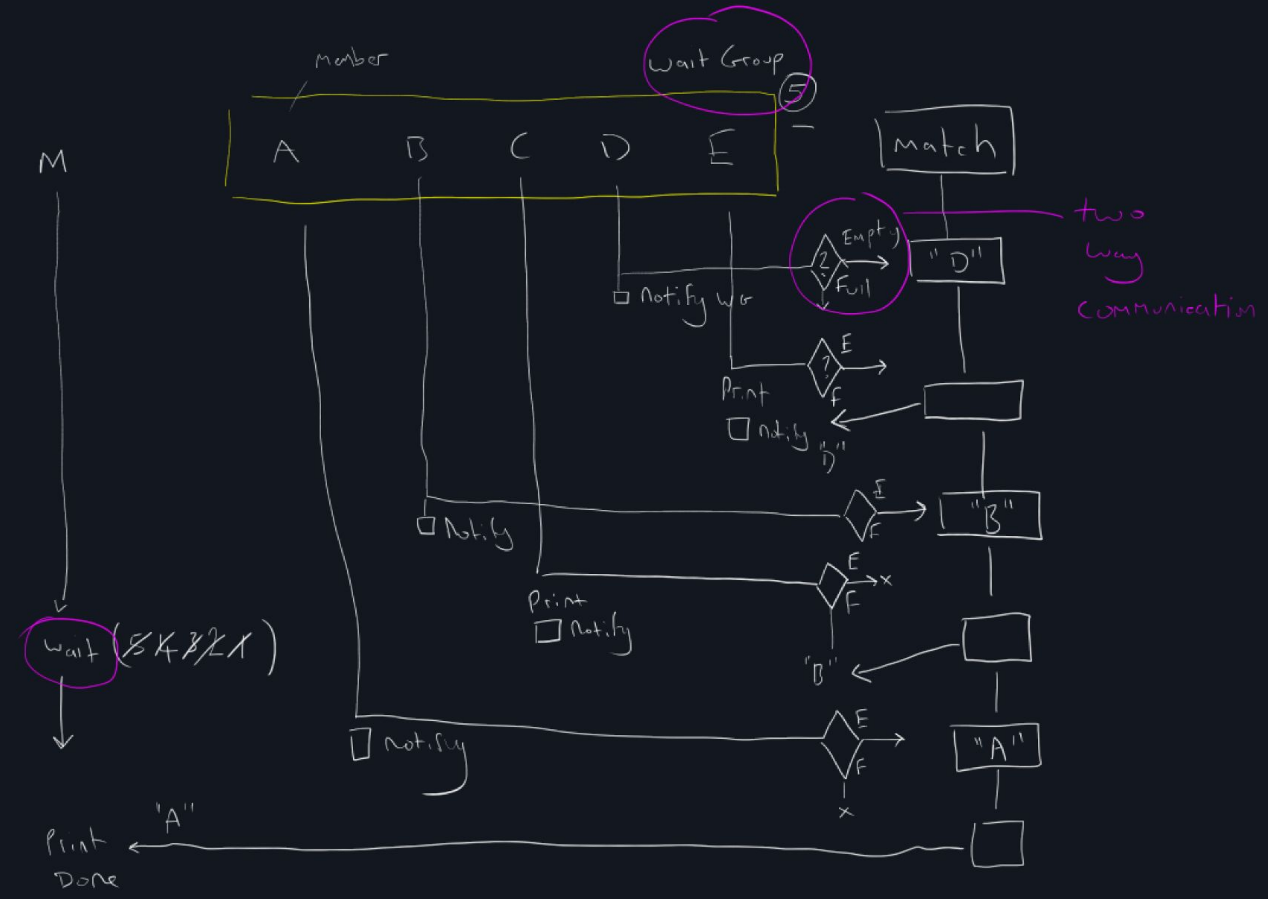
Matching setup

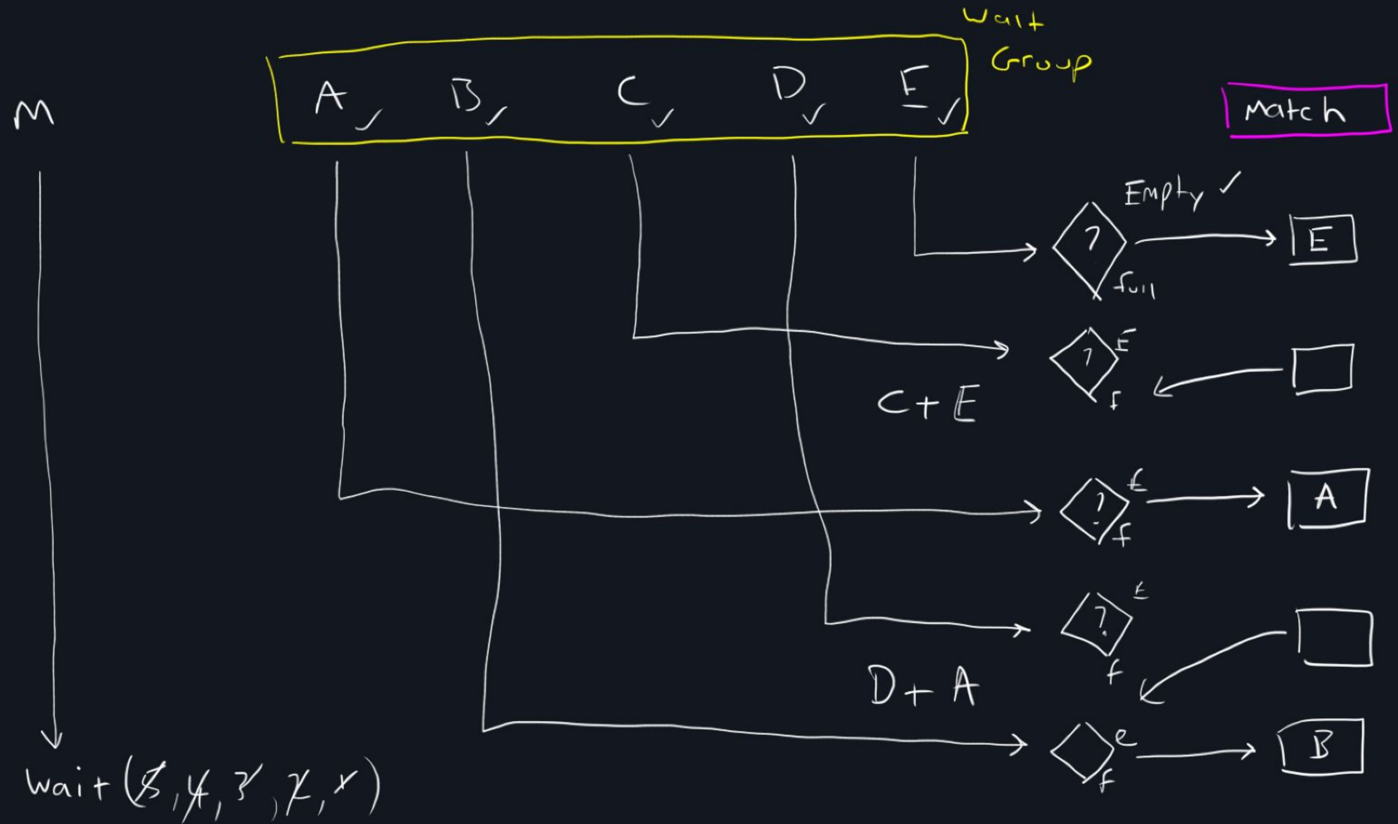
```go
// Demonstrates how a channel can be used for sending and
// receiving by any number of goroutines.
func main() {
    rand.Seed(time.Now().Unix())
    people := []string{"Anna", "Bob", "Cody", "Dave", "Eva"}
    match := make(chan string, 1) // Asynchronous channel.

    wg := new(sync.WaitGroup)
    wg.Add(len(people))

    for _, name := range people {
        go Seek(name, match, wg)
    }

    wg.Wait()

    select {
    case name := <-match:
        fmt.Printf("No one received %s's message.\n", name)
    default:
        // There was no pending send operation.
    }
}
```

Seek

```go
// Seek either sends or receives, whichever possible, a name on the match
// channel and notifies the wait group when done.
func Seek(name string, match chan string, wg *sync.WaitGroup) {
    time.Sleep(time.Duration(rand.Int31n(500)) * time.Millisecond)
    select {
    case peer := <-match:
        fmt.Printf("%s sent a message to %s.\n", peer, name)
    case match <- name:
        // for someone to receive my message.
    }
    wg.Done()
}
```

Draw: Matching


member
Wait Group
5
M
A
B
C
D
E
Match
two way communication
Empty
Full
"D"
Notify WG
E
F
Print
Notify
"D"
"B"
Notify
E
F
x
Print
Notify
"B"
Wait (~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~)
Notify
"A"
x
"A"
Print
Done


Wait Group
M
A ✓
B ✓
C ✓
D ✓
E ✓
Match
Empty ✓
?
full
E
?
E
f
C+E
?
E
f
A
?
E
f
D+A
e
f
B
wait(5, 4, 3, 2, 1)

Video

WaitGroups

Sync.WaitGroup

- **Add** a count of goroutines to wait for
- **Wait** (block point) until WaitGroup counter reaches zero
- **Done** decrements the count of a WaitGroup

```go
// Demonstrates how a channel can be used for sending and
// receiving by any number of goroutines.
func main() {
    rand.Seed(time.Now().Unix())
    people := []string{"Anna", "Bob", "Cody", "Dave", "Eva"}
    match := make(chan string, 1) // Asynchronous channel.

    wg := new(sync.WaitGroup)
    wg.Add(len(people))

    for _, name := range people {
        go Seek(name, match, wg)
    }

    wg.Wait()

    select {
    case name := <-match:
        fmt.Printf("No one received %s's message.\n", name)
    default:
    // There was no pendi
    }
}
```

```go
// Seek either sends or receives, whichever possible, a name on the match
// channel and notifies the wait group when done.
func Seek(name string, match chan string, wg *sync.WaitGroup) {
    time.Sleep(time.Duration(rand.Int31n(500)) * time.Millisecond)
    select {
    case peer := <-match:
        fmt.Printf("%s sent a message to %s.\n", peer, name)
    case match <- name:
        // for someone to receive my message.
    }
    wg.Done()
}
```

MapReduce Pattern

A simple task

Count the frequency of all unique words in a text file

Solution normally involves:

- Read in file as tokens
- Determine which tokens are words
- Count the frequency using a hashtable (key: string, value: int)

Can be greatly improved with concurrent solution

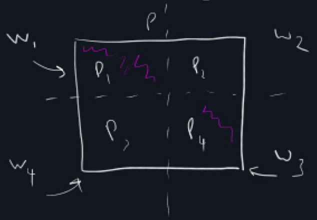
Draw: Concurrent Design Concerns

worker
problem

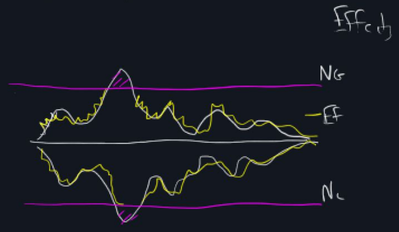
— TB of text, image (data)

Partitioning

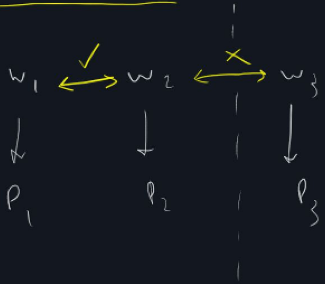
Domain (Apply Filter)



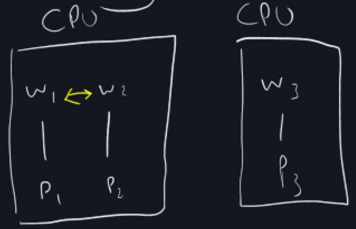
functional

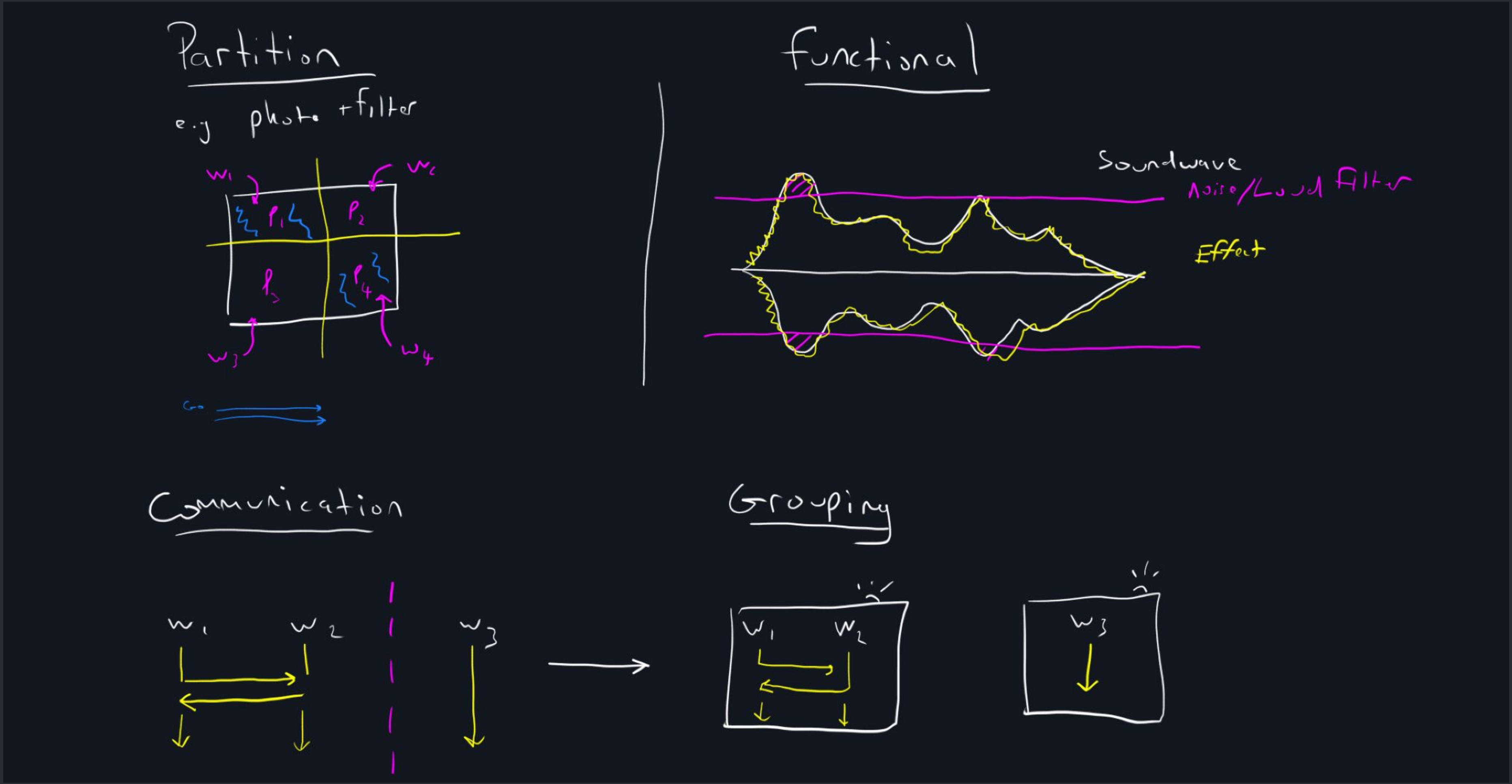


Communication



Grouping / Mapping




Partition
e.g photo +filter
w1
w2
p1
p2
p3
p4
w3
w4
Functional
Soundwave
Noise/Loud Filter
Effect
Communication
w1
w2
w3
Grouping
w1
w2
w3

Video

Design concerns: word frequency

Partitioning

- Domain decomposition: split text data into equal chunks

Communication concerns

- None, using map/reduce pattern

Grouping

- None, all tasks are independent

Mapping

- Go routine per processing unit (OS thread / CPU Core)

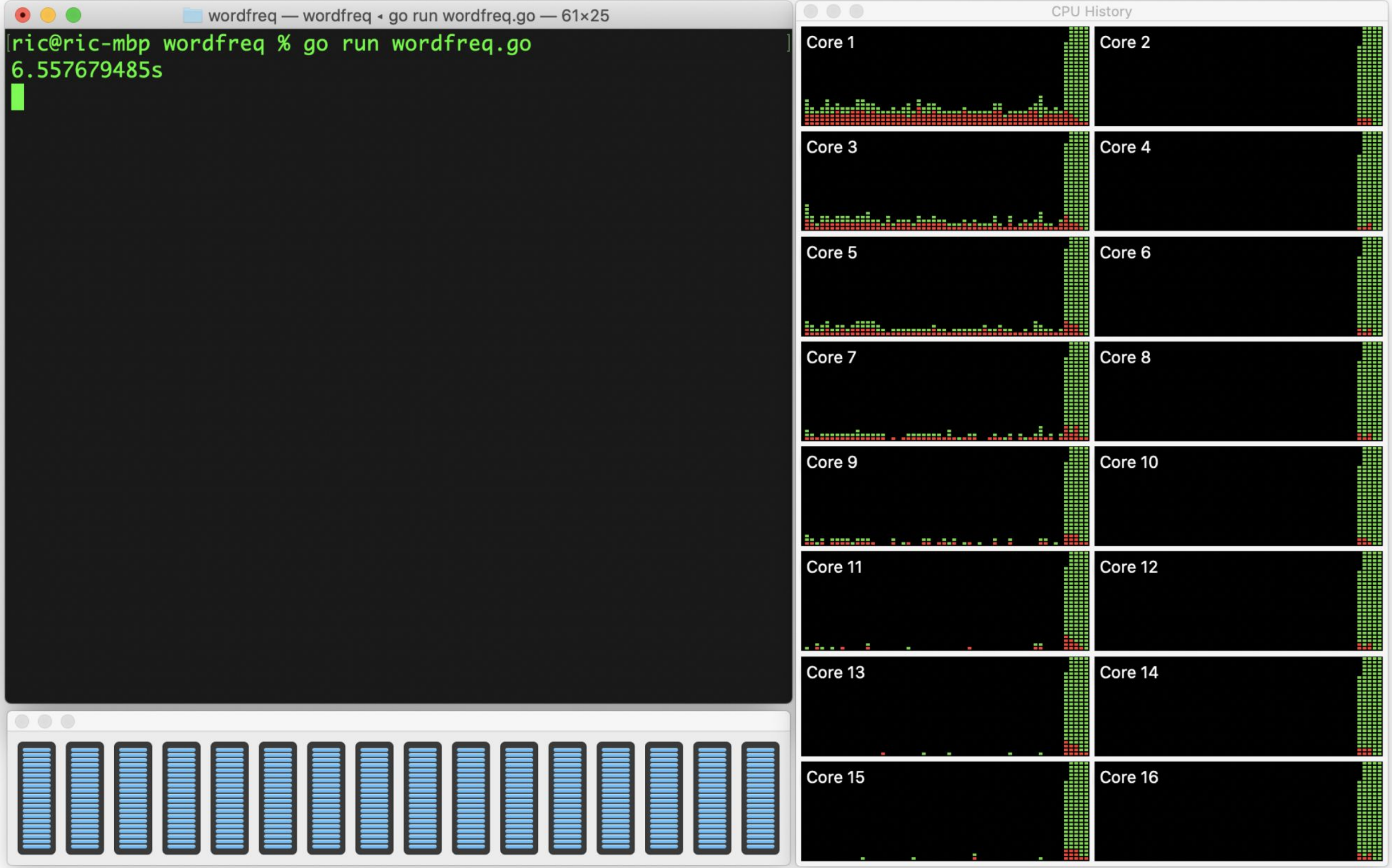
Word counters

```
func WordCount(s string, wg *sync.WaitGroup) map[string]int {
    // split input text by whitespace
    substrs := strings.Fields(s)
    counts := make(map[string]int)
    for _, word := range substrs {
        _, ok := counts[word]
        if ok == true {
            counts[word] += 1
        } else {
            counts[word] = 1
        }
    }
    // signal completion to main thread
    wg.Done()
    return counts
}
```

Sequential setup

```go
func main() {
    // Sequential - force 1 CPU core :)
    procs := 1
    runtime.GOMAXPROCS(procs)

    work := 1000000
    workers := 1

    // Wait for all workers to complete
    wg := new(sync.WaitGroup)
    wg.Add(workers)

    // Problem partitioning
    text := "Lorem ipsum dolor sit amet, consectetur …"
    problem := strings.Repeat(text, work/workers)

    // Start workers
    for i := 0; i < workers; i++ {
        go WordCount(problem, wg)
    }

    wg.Wait()
}
```

Demo: Word Frequency



```
ric@ric-mbp wordfreq % go run wordfreq.go
6.557679485s
```

Concept Review

Concept review

Channels support elegant communication and synchronisation operations with little syntax

Parallelism is simple in Go, after designing for concurrency from the start

In recent versions, parallelism is on by default :-)

Recommended Reading

Fundamentals of concurrent programming

- https://yourbasic.org/golang/wait-for-goroutines-waitgroup/
- https://yourbasic.org/golang/stop-goroutine/
- https://yourbasic.org/golang/time-reset-wait-stop-timeout-cancel-interval/
- https://yourbasic.org/golang/efficient-parallel-computation/