Concurrency in Go

9/22/17
Outline

Mapreduce (15 mins)

Two synchronization mechanisms

Locks (15 mins)

Channels (20 mins)
Hello my love. I love you, my dear. Goodbye.

hello: 1, my: 2, love: 2, i: 1, dear: 1, goodbye: 1
Application: Word count

Locally: tokenize and put words in a hash map

How do you parallelize this?

- Split document by half
- Build two hash maps, one for each half
- Merge the two hash maps (by key)
How do you do this in a distributed environment?
When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume, among the Powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.
When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume, among the Powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

Partition
When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume, among the Powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.
when: 1, in: 1, 
the: 1, course: 1, 
of: 1, human: 1, 
events: 1, it: 1

requires: 1, that: 1, 
they: 1, should: 1, 
declare: 1, the: 1, 
causes: 1, which: 1 ...

dissolve: 1, the: 2, 
the: 1, course: 1, 
of: 1, human: 1, 
events: 1, it: 1

among: 1, the: 2, 
powers: 1, of: 2, earth: 1, separate: 1, equal: 1, and: 1 ...

nature: 2, and: 1, of: 2, 
god: 1, entitle: 1, them: 1, 
decent: 1, respect: 1, 
mankind: 1, opinion: 1 ...

Compute word counts locally
Now what...

How to merge results?

Compute word counts locally
Merging results computed locally

Several options

Don’t merge — requires additional computation for correct results

Send everything to one node — what if data is too big? Too slow…

Partition key space among nodes in cluster (e.g. [a-e], [f-j], [k-p] …)

1. Assign a key space to each node
2. Split local results by the key spaces
3. Fetch and merge results that correspond to the node’s key space
when: 1, in: 1,
the: 1, course: 1,
of: 1, human: 1,
events: 1, it: 1

dissolve: 1, the: 2,
political: 1, bands: 1,
which: 1, have: 1,
connected: 1, them: 1

requires: 1, that: 1,
they: 1, should: 1,
declare: 1, the: 1,
causes: 1, which: 1

among: 1, the: 2,
powers: 1, of: 2, earth:
1, separate: 1, equal:
1, and: 1

god: 1, entitle: 1, them: 1,
decent: 1, respect: 1,
mankind: 1, opinion: 1

nature: 2, and: 1, of: 2,
when: 1, the: 1,
in: 1, it: 1, human: 1,
course: 1, events: 1,
of: 1,

bands: 1, dissolve: 1,
connected: 1, have: 1,
political: 1, the: 1,

the: 1, which: 1

causes: 1, declare: 1,
requires: 1, should: 1,
that: 1, they: 1, the: 1,

which: 1

nature: 2, of: 2,
mankind: 1, opinion: 1,
entitle: 1, and: 1,
decent: 1, god: 1,
them: 1, respect: 1,
among: 1, and: 1,
equal: 1, earth: 1,
separate: 1, the: 2,
powers: 1, of: 2

Split local results by key space
All-to-all shuffle
Note the duplicates...
Merge results received from other nodes
Mapreduce

Partition dataset into many chunks

**Map stage:** Each node processes one or more chunks locally

**Reduce stage:** Each node fetches and merges partial results from all other nodes
Mapreduce Interface

map(key, value) -> list(<k’, v’>)

Apply function to (key, value) pair
Outputs set of intermediate pairs

reduce(key, list<value>) -> <k’, v’>

Applies aggregation function to values
Outputs result
Mapreduce: Word count

map(key, value):
  // key = document name
  // value = document contents
  for each word w in value:
    emit (w, 1)

reduce(key, values):
  // key = the word
  // values = number of occurrences of that word
  count = sum(values)
  emit (key, count)
Mapreduce: Word count

How much wood would a woodchuck chuck if a woodchuck could chuck wood?

(a, 1), (woodchuck, 1), (would, 1), (a, 1), (lot, 1), (of, 1), (wood, 1), (if, 1), (a, 1), (woodchuck, 1), (could, 1), (chuck, 1), (wood, 1)

A woodchuck would chuck a lot of wood if a woodchuck could chuck wood.

(a, 1), (woodchuck, 1), (would, 1), (a, 1), (lot, 1), (of, 1), (wood, 1), (if, 1), (a, 1), (woodchuck, 1), (could, 1), (chuck, 1), (wood, 1)
Why is this hard?

Failure is common

Even if each machine is available 99.999% of the time, a datacenter with 100,000 machines still encounters failures \((1-(1-p)^n) = 63\%\) of the time

Data skew causes unbalanced performance across cluster

Problems occur at scale

Hard to debug!
Two synchronization mechanisms

**Locks** - limit access to a critical section

**Channels** - pass information across processes using a queue
Example: Bank account

Bob

Read b = 100
b = b + 10
Write b = 110

Bank Account

100

110

120

Alice

Read b = 110
b = b + 10
Write b = 120
Example: Bank account

Bob

Read $b = 100$

$b = b + 10$

Write $b = 110$

Alice

Read $b = 100$

$b = b + 10$

Write $b = 110$
What went wrong?

Changes to balance are not *atomic*

```java
func Deposit(amount) {
    lock balance
    read balance
    balance += amount
    write balance
    unlock balance
}
```
Semaphores

Allows at most $n$ concurrent accesses

Locks are a special case of semaphores, with $n = 1$
package account

import "sync"

type Account struct {
    balance int
    lock sync.Mutex
}

func NewAccount(init int) Account {
    return Account{balance: init}
}

func (a *Account) Deposit(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance += v
}

func (a *Account) CheckBalance() int {
    a.lock.Lock()
    defer a.lock.Unlock()
    return a.balance
}

func (a *Account) Withdraw(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance -= v
}

func (a *Account) Deposit(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance += v
}
package account

import "sync"

type Account struct {
    balance int
    lock sync.RWMutex
}

func NewAccount(init int) Account {
    return Account{balance: init}
}

func (a *Account) Deposit(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance += v
}

func (a *Account) Withdraw(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance -= v
}

func (a *Account) CheckBalance() int {
    a.lock.RLock()
    defer a.lock.RUnlock()
    return a.balance
}
Go channels

In Go, **channels** and **goroutines** are more idiomatic than locks.

```go
result := make(chan int, numWorkers)
// Launch workers
for i := 0; i < numWorkers; i++ {
    go func() {
        // ... do some work
        result <- i
    }()
}
// Wait until all worker threads have finished
for i := 0; i < numWorkers; i++ {
    handleResult(<-result)
}
fmt.Println("Done!")
```
Go channels

Easy to express asynchronous RPC

Awkward to express this using locks

```go
result := make(chan int, numServers)

// Send query to all servers
for i := 0; i < numServers; i++ {
    go func() {
        resp := // ... send RPC to server
        result <- resp
    }()
}

// Return as soon as the first server responds
handleResponse(<-result)
```
package account

type Account struct {
    // Fill in Here
}

func NewAccount(init int) Account {
    // Fill in Here
}

func (a *Account) CheckBalance() int {
    // What goes Here?
}

func (a *Account) Withdraw(v int) {
    // ???
}

func (a *Account) Deposit(v int) {
    // ???
}
package account

type Account struct {
    balance chan int
}

func NewAccount(init int) Account {
    a := Account{make(chan int, 1)}
    a.balance <- init
    return a
}

func (a *Account) CheckBalance() int {
    // What goes Here?
}

func (a *Account) Withdraw(v int) {
    // ???
}

func (a *Account) Deposit(v int) {
    // ???
}
package account

type Account struct {
    balance chan int
}

func NewAccount(init int) Account {
    a := Account{make(chan int, 1)}
    a.balance <- init
    return a
}

func (a *Account) CheckBalance() int {
    bal := <-a.balance
    a.balance <- bal
    return bal
}

func (a *Account) Withdraw(v int) {
    // ???
}

func (a *Account) Deposit(v int) {
    // ???
}
package account

type Account struct {
    balance chan int
}

func NewAccount(init int) Account {
    a := Account{make(chan int, 1)}
    a.balance <- init
    return a
}

func (a *Account) CheckBalance() int {
    bal := <-a.balance
    a.balance <- bal
    return bal
}

func (a *Account) Withdraw(v int) {
    bal := <-a.balance
    a.balance <- (bal - v)
}

func (a *Account) Deposit(v int) {
    //???
}
Bank Account Code (using channels)

```go
package account

type Account struct {
    balance chan int
}

func NewAccount(init int) Account {
    a := Account{make(chan int, 1)}
    a.balance <- init
    return a
}

func (a *Account) CheckBalance() int {
    bal := <-a.balance
    a.balance <- bal
    return bal
}

func (a *Account) Withdraw(v int) {
    bal := <-a.balance
    a.balance <- (bal - v)
}

func (a *Account) Deposit(v int) {
    bal := <-a.balance
    a.balance <- (bal + v)
}
```
Exercise: Locks and semaphores (using channels)

```go
type Lock struct {
    // ???
}

func NewLock() Lock {
    // ???
}

func (l *Lock) Lock() {
    // ???
}

func (l *Lock) Unlock() {
    // ???
}

type Semaphore struct {
    // ???
}

func NewSemaphore(n int) Semaphore {
    // ???
}

func (s *Semaphore) Acquire() {
    // ???
}

func (s *Semaphore) Release() {
    // ???
}
```
type Lock struct {
    ch chan bool
}

func NewLock() Lock {
    l := Lock{make(chan bool, 1)}
    l.ch <- true
    return l
}

func (l *Lock) Lock() {
    <-l.ch
}

func (l *Lock) Unlock() {
    l.ch <- true
}

type Semaphore struct {
    ch chan bool
}

func NewSemaphore(n int) Semaphore {
    s := Semaphore{make(chan bool, n)}
    for i := 0; i < n; i++ {
        s.ch <- true
    }
    return s
}

func (s *Semaphore) Acquire() {
    <-s.ch
}

func (s *Semaphore) Release() {
    s.ch <- true
}
Select statement

`select` allows a goroutine to wait on multiple channels at once

```go
for {
    select {
    case money := <-dad:
        buySnacks(money)
    case money := <-mom:
        buySnacks(money)
    }
}
```
Select statement

`select` allows a goroutine to wait on multiple channels at once

```go
for {
    select {
    case money := <-dad:
        buySnacks(money)
    case money := <-mom:
        buySnacks(money)
    case default:
        starve()
        time.Sleep(5 * time.Second)
    }
}
```
Handle timeouts using select

```go
result := make(chan int)
timeout := make(chan bool)

// Asynchronously request an answer from server, timing out after X seconds
askServer(result, timeout)

// Wait on both channels
select {
    case res := <-result:
        handleResult(res)
    case <-timeout:
        fmt.Println("Timeout!")
}
```

```go
func askServer(
    result chan int,
    timeout chan bool) {
    // Start timer
    go func() {
        time.Sleep(5 * time.Second)
        timeout <- true
    }()

    // Ask server
    go func() {
        response := // ... send RPC
        result <- response
    }()
}
```
Handle timeouts using select

```go
result := make(chan int)
timeout := make(chan bool)

// Asynchronously request an answer from server, timing out after X seconds
askServer(result, timeout)

// Wait on both channels
select {
    case res := <-result:
        handleResult(res)
    case <-timeout:
        fmt.Println("Timeout!")
}

func askServer(result chan int, timeout chan bool) {
    // Start timer
    go func() {
        time.Sleep(5 * time.Second)
        timeout <- true
    }()

    // Ask server
    go func() {
        response := // ... send RPC
        result <- response
    }()
}
```
Assignment 1.2 is out, due 9/28