Programming in Go

Matt Holiday
Christmas 2020

File Walk Example

What's the problem?

I want to find duplicate files based on their **content**

What's the problem?

I want to find duplicate files based on their **content**

Use a secure hash, because the names / dates may differ

```
f088913 2
    /Users/mholiday/Dropbox/Emergency/FEMA_P-320_2014_508.pdf
    /Users/mholiday/Dropbox/Emergency/nps61-072915-01.pdf
```

What's the problem?

I want to find duplicate files based on their **content**

Use a secure hash, because the names / dates may differ

```
f088913 2
    /Users/mholiday/Dropbox/Emergency/FEMA_P-320_2014_508.pdf
    /Users/mholiday/Dropbox/Emergency/nps61-072915-01.pdf
```

It takes nearly **5 minutes** to comb through my Dropbox folder
(2017 core i7 quad-core MacBook Pro)

Code at https://github.com/matt4biz/go-class-walk

Sequential Approach

How it works: Declarations

```
package main

import (
    "crypto/md5"
    "fmt"
    "io"
    "log"
    "os"
    "path/filepath"
)

type pair struct {
    hash, path string
}

type fileList []string
type results  map[string]fileList
```

How it works: Hashing

```
func hashFile(path string) pair {
    file, err := os.Open(path)

    if err != nil {
        log.Fatal(err)
    }

    defer file.Close()

    hash := md5.New() // fast & good enough

    if _, err := io.Copy(hash, file); err != nil {
        log.Fatal(err)
    }

    return pair{fmt.Sprintf("%x", hash.Sum(nil)), path}
}
```

How it works: Searching

```
func searchTree(dir string) (results, error) {
    hashes := make(results)

    err := filepath.Walk(dir, func(p string, fi os.FileInfo,
                                   err error) error {
        // ignore the error parm for now

        if fi.Mode().IsRegular() && fi.Size() > 0 {
            h := hashFile(p)
            hashes[h.hash] = append(hashes[h.hash], h.path)
        }

        return nil
    })

    return hashes, err
}
```

How it works: Output

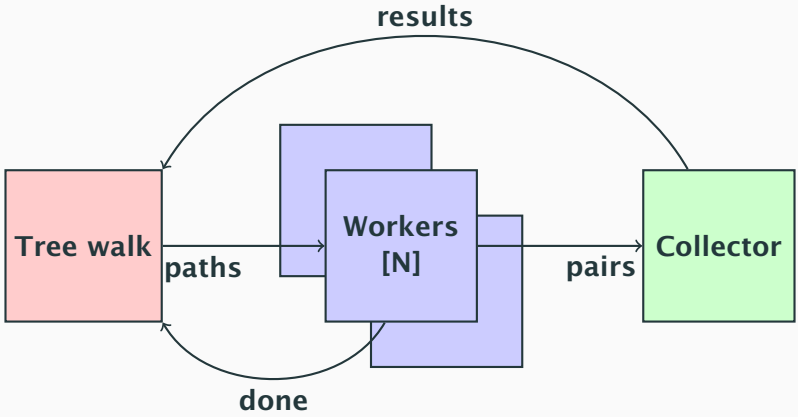
```
func main() {
    if len(os.Args) < 2 {
        log.Fatal("Missing parameter, provide dir name!")
    }
    if hashes, err := searchTree(os.Args[1]); err == nil {
        for hash, files := range hashes {
            if (len(files) > 1) {
                // we will use just 7 chars like git
                fmt.Println(hash[len(hash)-7:], len(files))

                for _, file := range files {
                    fmt.Println("   ", file)
                }
            }
        }
    }
}
```

Concurrent Approach #1

A concurrent approach (like map-reduce)

Use a fixed pool of goroutines and a collector and channels



How it works: Collecting the hashes

```
func collectHashes(pairs <-chan pair, result chan<- results) {
    hashes := make(results)

    for p := range pairs {
        hashes[p.hash] = append(hashes[p.hash], p.path)
    }

    result <- hashes
}
```

How it works: Replacing the processor

```
func processFiles(paths <-chan string, pairs chan<- pair,
                  done chan<- bool) {
    for path := range paths {
        pairs <- hashFile(path)
    }

    done <- true
}
```

How it works: Replacing the tree walk

```
workers := 2 * runtime.GOMAXPROCS(0)

paths := make(chan string)
pairs := make(chan pair)
done := make(chan bool)
result := make(chan results)

for i := 0; i < workers; i++ {
    go processFiles(paths, pairs, done)
}

// we need another goroutine so we don't block here

go collectHashes(pairs, result)

. . .
```

How it works: Replacing the tree walk

```
err := filepath.Walk(dir, func(p string, fi os.FileInfo,
                                err error) error {
    // again, ignore the error passed in

    if fi.Mode().IsRegular() && fi.Size() > 0 {
        paths <- p
    }

    return nil
})

if err != nil {
    log.Fatal(err)
}

// we must close the paths channel so the workers stop
close(paths)
. . .
```

How it works: Replacing the tree walk

```
. . .

// wait for all the workers to be done

for i := 0; i < workers; i++ {
    <-done
}

// by closing pairs we signal that all the hashes
// have been collected; we have to do it here AFTER
// all the workers are done

close(pairs)

hashes := <-result

return hashes
```

Evaluation #1

56.11s in the version shown above

Evaluation #1

56.11s in the version shown above

52.76s with a buffer to `pairs` (buffering `pairs` keeps the workers working)

Evaluation #1

56.11s in the version shown above

52.76s with a buffer to `pairs` (buffering `pairs` keeps the workers working)

51.36s with twice as many workers (32)

Concurrent Approach #2

Another concurrent approach

Add a goroutine for each directory in the tree

Another concurrent approach

Add a goroutine for each directory in the tree

This improves the performance slightly, we're not waiting on paths to be identified

How it works: Parallel tree walk

```
. . .

wg := new(sync.WaitGroup)

// multi-threaded walk of the directory tree; we need a
// waitGroup because we don't know how many to wait for

wg.Add(1)
err := walkDir(dir, paths, wg)

if err != nil {
    log.Fatal(err)
}

wg.Wait()
close(paths)

. . .
```

How it works: Parallel tree walk

```
func walkDir(dir string, paths chan<- string,
             wg *sync.WaitGroup) error {
    defer wg.Done()

    visit := func(p string, fi os.FileInfo, err error) error {
        // ignore the error passed in

        // ignore dir itself to avoid an infinite loop!
        if fi.Mode().IsDir() && p != dir {
            wg.Add(1)
            go walkDir(p, paths)
            return filepath.SkipDir
        }

        . . .
```

How it works: Parallel tree walk

```
        . . .

        if fi.Mode().IsRegular() && fi.Size() > 0 {
            paths <- p
        }

        return nil
    }

    return filepath.Walk(dir, visit)
}
```

Evaluation #2

51.14s in the basic version

Evaluation #2

51.14s in the basic version

50.03 adding buffers on all channels to/from workers

Evaluation #2

51.14s in the basic version

50.03 adding buffers on all channels to/from workers

48.75 with twice as many workers

Concurrent Approach #3

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

What could go wrong?

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

What could go wrong?

Without some controls, we'll run out of threads!

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

What could go wrong?

Without some controls, we'll run out of threads!

`GOMAXPROCS` doesn't limit threads blocked on syscalls (all our disk I/O)

We'll limit the number of **active** goroutines instead (the ones making syscalls)

Channels as counting semaphores

A goroutine can't proceed without *sending* on the channel

Channels as counting semaphores

A goroutine can't proceed without *sending* on the channel

A channel with buffer size N can accept N sends without blocking
(with no intervening reads)

Channels as counting semaphores

A goroutine can't proceed without *sending* on the channel

A channel with buffer size N can accept N sends without blocking (with no intervening reads)

The buffer provides a fixed upper bound (unlike a WaitGroup)

Channels as counting semaphores

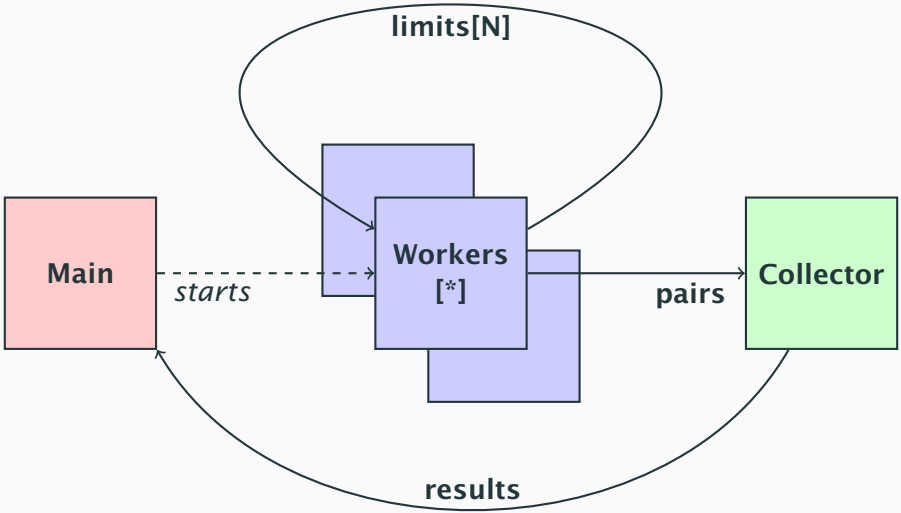
A goroutine can't proceed without *sending* on the channel

A channel with buffer size N can accept N sends without blocking
(with no intervening reads)

The buffer provides a fixed upper bound (unlike a WaitGroup)

One goroutine can start for each one that finishes
(each *reads* from the channel when done)

What that looks like



How it works: Limiting goroutines

```
// we don't need a channel for paths or to signal done but
// we need a buffered channel to act as a counting semaphore

wg := new(sync.WaitGroup)
limits := make(chan bool, workers)
pairs := make(chan pair, workers)
result := make(chan results)

go collect(pairs, result)

. . .
```

How it works: Limiting goroutines

```
. . .

wg.Add(1)
err := walkDir(dir, pairs, wg, limits)

if err != nil {
    log.Fatal(err)
}

wg.Wait()
close(pairs)

hashes := <-result

return hashes
```

How it works: Modified processing

```
func processFile(path string, pairs chan<- pair,
                 wg *sync.WaitGroup, limits chan bool) {
    defer wg.Done()

    limits <- true

    defer func() {
        <-limits
    }()

    pairs <- hashFile(path)
}
```

How it works: Modified tree walk

```
func walkDir(dir string, pairs chan<- pair, wg *sync.WaitGroup,
             limits chan bool) error {
    defer wg.Done()

    visit := func(p string, fi os.FileInfo, err error) error {
        // ignore the error passed in

        if fi.Mode().IsDir() && p != dir {
            wg.Add(1)
            go walkDir(p, pairs, wg, limits)
            return filepath.SkipDir
        }

        . . .
```

How it works: Modified tree walk

```
        . . .

        if fi.Mode().IsRegular() && fi.Size() > 0 {
            wg.Add(1)
            go processFile(p, pairs, wg, limits)
        }

        return nil
    }

    limits <- true

    defer func() {
        <-limits
    }()

    return filepath.Walk(dir, visit)
}
```

Evaluation #3

46.93s using 32 workers was the best time

Evaluation #3

46.93s using 32 workers was the best time

Increasing the `limits` buffer makes the time grow longer due to disk contention

Evaluation #3

46.93s using 32 workers was the best time

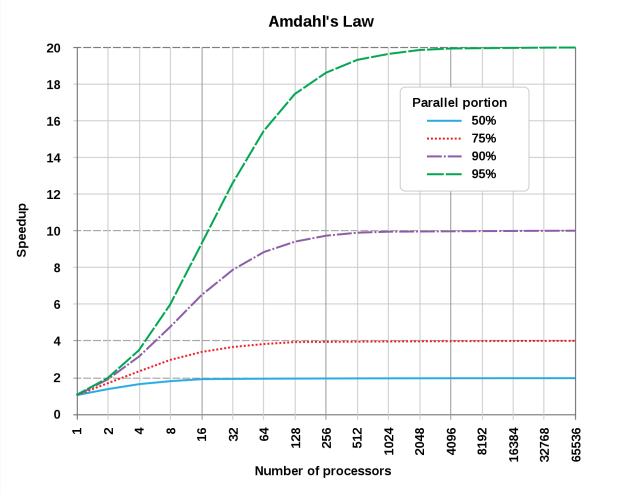
Increasing the `limits` buffer makes the time grow longer due to disk contention

Amdahl's law: speedup is limited by the part (not) parallelized

$$S = \frac{1}{1 - p + (p/s)}$$

Here we've managed about $S = 6.25$ on $s = 8$ processors, or about $p = 96\%$ parallel

Evaluation #3



Conclusions

We don’t need to limit *goroutines*

We need to limit *contention* for shared resources