Some Experiments with Goroutines and Channels

Matt Holiday

27 December 2018

Abstract

Given the problem of identifying duplicate files in a large directory tree, how should goroutines and channels be organized to yield the best performance?

1 Purpose

Given a large directory tree with many files, we would like to identify files with identical content by means of a secure hash function. In the present case, the tree contains roughly 6200 directories and over 42000 files that have been collected into it over a period of years[1]. A PDF file of some paper may exist in more than one subdirectory, possibly with a different name, as it was found to be of interest in one or more research projects, for example.

Note that there are some special cases to be considered, such as the fact that all zero-length files will yield identical hash values, or that certain "dot" directories or files should be ignored.

Given the size of the directory tree and the ubiquity of multi-core hardware, it is desirable to do as much work in parallel as possible. The go language makes this relatively easy with its concurrency primitives such as channels and goroutines. Still, some thought is required on how best to use those primitives. For example, simply creating and running a goroutine for every directory and file will fail in most non-trivial cases (exhaustion of OS kernel resources with a very large number of threads blocked on system calls).

The non-parallel version of this program took over five minutes to run on the original directory of over 60000 files. As we will see, using goroutines will give us a speedup of about 6, given that the program is allowed to saturate three of the four cores (six of eight hyperthreads) on our machine.

[1] Down from perhaps 60000 files when we began, before some were made redundant and others were identified as "to be ignored."

2 Walking the Tree

Let's first examine the non-parallel case. We will use the not-too-secure MD5 hash as it will be fast and sufficiently good to identify duplicate files, even if no longer used cryptographically. A file hashing function might then be:

```
type pair struct {
    hash string
    path string
}

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

    // we will allow that files might be deleted while we walk the tree
    if err != nil && !os.IsNotExist(err) {
        log.Fatal(err)
    }

    defer file.Close()

    // MD5 may not be cryptographically secure but it works for finding
    // matching files well enough
    hash := md5.New()

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

    // we need to format the hash since we're using string keys later in a map
    return pair{fmt.Sprintf("%x", hash.Sum(nil)), path}
}
```

The basic logic of walking the file tree uses a convenient go function based on the *visitor* pattern; that is, we provide it with a callback function to be called on each directory and file in the tree. In its simplest form it might appear as follows:

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

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

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

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

        return nil
    })

    return hashes, err
}
```

From the hashes result we can then trivially identify duplicate files from the map entries that have a slice of file names of length greater than one, for example:

```
    for hash, files := range hashes {
        if (len(files) > 1) {
            // we will use the last 7 chars as a short ID just like git
            fmt.Println(hash[len(hash)-7:], len(files))

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

Then, given output such as the following sample, we can find individually-duplicated files or even groups of files that can be merged or made redundant.

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

3 A First Approach

In making the first approach to parallelizing this code, we assume (rightly!) that most of the time is spent reading in files to calculate the hash value. Thus, we will keep a single-threaded tree walk, but use multiple goroutines to find the hashes, plus one more to collect the hash values together. Essentially, we have a "worker pool" of goroutines; they receive paths from a common channel and write their results to another channel read by the collector function.

So our tree walking code might look like this:

```
    workers := 2 * runtime.GOMAXPROCS(0) // a reasonable worker pool size

    paths := make(chan string)       // feed paths to workers
    pairs := make(chan pair)         // feed hashes to collector
    done := make(chan bool, workers) // worker semaphores
    result := make(chan results)     // storage we'll give to the collector

    // we want a pool of workers, not a goroutine per file; each worker gets
    // a path to a file that needs to be hashed and when there are no more
    // paths, it reports done
    for i := 0; i < workers; i++ {
        go processFiles(paths, pairs, done)
    }
    // we need another goroutine to act as a collector for the workers' output
    // so we don't block here
    go collectHashes(pairs, result)

    // single-threaded walk of the directory tree looking for files
    err := filepath.Walk(dir, func(p string, fi os.FileInfo, err error) error {
        // we will allow that files might be deleted concurrently
        if err != nil {
            if os.IsNotExist(err) {
                return nil
            }

            return err
        }

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

        return nil
    })

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

    // signal no more paths to be hashed, which will eventually stop all workers
    close(paths)

    // 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
    // can't do this in the workers themselves since they're not coordinated
    close(pairs)

    hashes := <-result

    // so now hashes will be complete (no longer mutating in the background)
    return hashes, err
```

The worker functions then appear as

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

    done <- true
}
```

and the collector

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

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

    result <- hashes
}
```

To recap, here we've created a goroutine workgroup whose size is tied to the number of available (logical) processors for the purpose of hashing files, with the results fed into another single collector. We use a buffered channel `done` to track completion; its capacity is just the number of workers, so that we can block until all of them have exhausted the shared channel of paths. We must then close the channel of hash values being collected, since none of the hashing workers have knowledge of each other and so cannot cooperate to close it deterministically. We then block again until the collector has exhausted its input and returns a complete map.

3.1 Performance and Small Improvements

These experiments were all run on an Intel core i7 quad-core CPU, which has eight logical processors when hyperthreading is taken into account; thus, `GOMAXPROCS` returns 8. We originally ran twice as many worker goroutines, for a total of 16. This version of the code ran in 56.11 seconds average (over three runs).

Looking at the code, we can see a couple of possible sticking points. Our channels to and from the worker goroutines aren't buffered, and if the collector is not scheduled or is busy, for example, workers will be blocked waiting for it to read from channel `pairs`. We can also block the tree-walk code trying to add paths to `paths`. Adding a small buffer to each of these channels can reduce such "friction" and keep the goroutines running:

```
    paths := make(chan string, 2 * workers) // feed paths to workers
    pairs := make(chan pair, workers)       // feed hashes to collector
```

These improvements reduced our time to 52.76 seconds on average. Finally, we tried increasing the goroutine pool from 16 to 32 workers, which only reduced our time to 51.36 seconds.

4 Take Two

The next approach was to keep the goroutine pool of workers hashing files, but to add goroutines during the directory walk, one per directory encountered. Note that these tree walk goroutines will be very short-lived, running just long enough to process a list of files and feed paths into the `paths` channel. The middle part of `searchTree` then becomes

```
    wg := new(sync.WaitGroup)

    // multi-threaded walk of the directory tree looking for files; walkDir
    // creates goroutines to handle directories in parallel

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

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

    // signal no more paths to be hashed, which will eventually stop all workers
    wg.Wait()
    close(paths)
```

Note the use of a `WaitGroup` to track the tree-walk goroutines, since we don't know how many will be created. Also note that we always add to the wait group *prior* to each call to `walkDir` as shown below:

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

    visit := func(p string, fi os.FileInfo, err error) error {
        if err != nil {
            if os.IsNotExist(err) {
                return nil
            }

            return err
        }

        // ignore dir itself or we'll end up in an infinite loop!
        if fi.Mode().IsDir() && p != dir {
            wg.Add(1)
            go walkDir(p, paths)
            return filepath.SkipDir
        }

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

        return nil
    }

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

At each level we ensure that `walkTree` will signal the wait group via `defer` and then walk the directory tree we've been given. Note that we return SkipDir to signal Walk that we've processed the directory — by handing it off to another goroutine.

4.1 Performance and Small Improvements

For this version, we again started with unbuffered channels to the workers and only 16 workers total. The average time was 51.14 seconds. Adding small buffers to the channels pairs and paths reduced the time to 50.03 seconds, while increasing the number of workers to 32 further reduced our time to 48.75 seconds. Given that we can generate paths more quickly by a parallel tree walk, there appears to be more utility in having a larger worker pool compared to the first version.

5 Goroutines Galore!

Our third approach took advantage of the "embarassingly parallel" nature of the problem, given that (in theory!) we could run a goroutine for every directory in the tree walk and for every file that needs to be hashed. We tried that, but unfortunately we were embarassed by a runtime panic with the message "failed to create new OS thread." This happens because GOMAXPROCS does not limit the number of threads blocked on a system call, and very quickly most threads were blocked waiting on disk access either for directory information or the contents of a file.

So our third approach takes a new tack — not to limit the number of goroutines we create, but to limit the number which may be active at any one time. We remove the worker pool of file-hashing goroutines and channel done, an instead use another buffered channel limits which acts as a counting semaphore. Each goroutine must get a "token" by attempting to add to limits and continues only when limits has capacity; it reads from limits when done in order to release its token to the next goroutine.

searchTree is simplified; instead of creating worker goroutines, we create some channels; the rest of it doesn't change from the code above.

```
wg := new(sync.WaitGroup)
limits := make(chan bool, nworkers) // limit in-progress work
pairs := make(chan pair, nworkers)  // feed hashes to collector
result := make(chan results)        // get results back from it
```

Our tree-walking code now looks like

```
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 {
        if err != nil {
            if os.IsNotExist(err) {
                return nil
            }

            return err
        }

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

        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)
}
```

On each call to `walkDir` we must attempt to write to `limits` and ensure that on exit we read from it, so our `defer` statement must now be a closure which does that as well as signal completion of the goroutine. The collector doesn't change, since there's only ever one, but the file-hashing function must now also take into account the `limits` channel.

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

    limits <- true

    defer func() {
        <-limits
    }()

    pairs <- hashFile(path)
}
```

Note that we still need the `WaitGroup` to let us know that all the "input" goroutines have completed; the `limits` channel doesn't convey that information. It's not safe to judge completion by waiting for `limits` to have full capacity, as (a) that count is transitory, and (b) doesn't ensure that another goroutine currently blocked won't wake up and add to `limits` while it runs.

5.1 Performance and Variations

Surprisingly, this version outperforms all others. Before we settled on it, we tried an approach with two "limiter" channels, one for the tree walk, and one for hashing files, so that we could control how many goroutiens of each type could run concurrently. With this interim version, whose code is not shown, we tried various combinations of goroutines, such as 20 of each, or 8 of each, or 12 of one and 20 of the other; all of them ran 50–52 seconds on average, which was not better than the second approach above.

Using a `limits` buffer allowing a total 20 workers of any type, we achieved 48.19 seconds on average, while 32 yielded our best time, 46.93 seconds. Thus, there's no value in trying to guess what proportion of the workers should be walking instead of hashing.

What happens if we use even more concurrent workers? A limit of 64 actually reduced performance slightly, while trying 200 caused thrashing and a very large increase in run time.

6 A Few More Details

We now turn to the main program logic, which hasn't been shown. As we noted in the introduction, there may be files and directories we don't want to inspect. We handle that with some additional maps and command-line flag logic.

First, we'll define some defaults; other than `.git` directories these are mostly Apple-specific.

```go
type ignores map[string]bool

var ignoreDirExts = ignores{
    ".app":         true,
    ".git":         true, // treated as an extension, not a name
    ".pkg":         true,
    ".idea":        true, // ditto
    ".lproj":       true,
    ".pbproj":      true,
    ".xcassets":    true,
    ".framework":   true,
    ".xcodeproj":   true,
    ".xcworkspace": true,
    ".xcdatamodel": true,
}

var ignoreDirs = ignores{}

var ignoreFiles = ignores{
    ".DS_Store":  true,
    ".gitignore": true,
}
```

In the main function, we'll handle possibly multiple flags which add directories to ignore. We must define a couple of routines on the `ignores` type for the flag logic to work:

```
func (i *ignores) String() string {
    dirs := make([]string, 0, len(*i))

    for v := range *i {
        dirs = append(dirs, v)
    }

    return fmt.Sprintf("%v", dirs)
}

func (i *ignores) Set(v string) error {
    (*i)[v] = true
    return nil
}

func main() {
    flag.Var(&ignoreDirs, "i", "ignore files under this path")
    flag.Parse()
    ...
```

And then our tree walk code must be changed to match:

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

        // ignore dir itself or we'll end up in an infinite loop!
        if fi.Mode().IsDir() && p != dir {
            if ignoreDirExts[filepath.Ext(p)] {
                return filepath.SkipDir
            }

            if ignoreDirs[p] {
                return filepath.SkipDir
            }

            wg.Add(1)
            go walkDir(p, pairs, wg, limits, verb)
            return filepath.SkipDir
        }

        if fi.Mode().IsRegular() && fi.Size() > 0 {
            if !ignoreFiles[filepath.Base(p)] {
                wg.Add(1)
                go process(p, pairs, wg, limits)
            }
        }

        return nil
    }
```

7 What About Directories?

We now extend this discussion to identifying duplicate directories that contain exactly the same content. We find them by constructing a *Merkle tree* in which each node has a hash representing the contents of a file or a directory; the hash of a directory is the hash of the hashes of its files (and subdirectories). If two directories have the same hash as defined here then they contain the same files, regardless of the file names or dates.

7.1 Merkle Tree Functions

Our tree will be simplified a bit, as there's no need for leaf nodes for files. Instead, a leaf node will be a directory that has no subdirectories. Each node may have any number of files, whose hashes will be kept in a slice. Since the output of a secure hash function is sensitive to the order of bytes fed into it, we will sort the hashes at each level before we combine them into a new hash in the parent node. Each node stores its own path and maps a subdirectory name to a child node.

```
type node struct {
    path  string
    hash  hash.Hash
    files []string
    child map[string]*node
}

// make a node to represent a directory in the tree
func makeNode(p string) *node {
    return &node{path: p, hash: md5.New(), child: make(map[string]*node)}
}
```

We use the insert routine only for files as we process and calculate their hashes. We append the hash into a slice of hashes in the node representing the directory in which they are found. Note that we don't feed these hashes into the node's hash function yet, because we need to sort them first once they're all found.

```
func (n *node) insert(path string, hash string) {
    s := strings.Split(path, "/")

    if len(s) == 1 {
        // we have a file in this directory

        n.files = append(n.files, hash)
    } else {
        // we need to insert into a child, which might need to be made
        c := s[0]

        if n.child[c] == nil {
            n.child[c] = makeNode(filepath.Join(n.path, c))
        }
        n.child[c].insert(filepath.Join(s[1:]...), hash)
    }
}
```

When file processing is complete, we do a post-order walk of the Merkle tree in order to calculate directory hashes.

```
func (n *node) walk(hashes results) string {
    for _, c := range n.child {
        n.files = append(n.files, c.walk(hashes))
    }

    // we need a consistent order to the hashes because we
    // don't know the order in which files were added

    sort.Strings(n.files)

    for _, h := range n.files {
        n.hash.Write([]byte(h))
    }

    // we need to format the hash since we're using string keys

    hash := fmt.Sprintf("%x", n.hash.Sum(nil))

    // we also update the master results hash; directories
    // always have a trailing slash so we recognize them

    hashes[hash] = append(hashes[hash], n.path+"/")
    return hash
}
```

7.2 Changes to Other Routines

The collector is updated to insert file hashes into the Merkle tree as they are received and then walk the tree before reporting out the results:

```
func collect(dir string, pairs <-chan pair, result chan<- results) {
    hashes := make(results)
    tree := makeNode(dir)

    for pair := range pairs {
        // we keep a multi-map of hash -> file paths that match, so we
        // need to append (not insert) any new data

        hashes[pair.hash] = append(hashes[pair.hash], pair.path)

        // and we insert this file hash into a directory tree

        rel, _ := filepath.Rel(dir, pair.path)
        tree.insert(rel, pair.hash)
    }

    tree.walk(hashes)
    result <- hashes
}
```

7.3 Reducing the Output

The hash results returned from the collector will now include directories. In some cases, two directories may be identical, possibly down one or more levels of subdirectories in addition to files. To simplify the listing, these files and subdirectories should be omitted from the final output if we are only looking for duplicates (using the -d option). However, if a file or subdirectory has a duplicate in some other part of the tree outside these parent duplicate directories, it should still be listed. For example, suppose directories A and B are duplicates, each having a subdirectory A1 and B1 respectively, where some directory C has a subdirectory C1 which duplicates B1 but is not itself a duplicate of B. Then we would like to identify A and B as duplicates while also identifying A1, B1, and C1 also as duplicates.

To do this we must find and remove directories or files which lie in or under directories whose hashes match. Unfortunately, this process is involved, as we must first proceed from the hashes and not from the directory tree structure. We find duplicate directories and sort them by level in the tree and remove the children to get a set of "root" directories that match, and then filter the hash results to remove items which have these roots in their paths.

```
func filterDupDirectories(hashes results) results {
    dupLevels := make(map[int][]string)
    dupDirs := make(map[string]bool)

    for _, files := range hashes {
        if len(files) == 1 {
            continue
        }

        for _, path := range files {
            if strings.HasSuffix(path, "/") {
                dupDirs[path] = true

                s := strings.Split(path, "/")
                t := len(s)
                dupLevels[t] = append(dupLevels[t], path)
            }
        }
    }

    // sort the paths by length, shortest first

    var keys []int
```

```
        for k := range dupLevels {
            keys = append(keys, k)
        }

        sort.Ints(keys)

        // starting with the shortest paths, remove any dup dir
        // that has the path as a proper prefix; we should end
        // up with only the shortest paths

        for _, k := range keys {
            for _, dir := range dupLevels[k] {
                for path := range dupDirs {
                    if path != dir && strings.HasPrefix(path, dir) {
                        delete(dupDirs, path)
                    }
                }
            }
        }

        // now remove duplicate entries from the overall list, but
        // only if ALL the entries under a hash have dup dirs

        for hash, files := range hashes {
            if len(files) == 1 {
                continue
            }

            j := 0

            for _, path := range files {
                for p := range dupDirs {
                    if path != p && strings.HasPrefix(path, p) {
                        j++
                    }
                }
            }

            if j == len(files) {
                delete(hashes, hash)
            }
        }

        return hashes
    }
```

Finally, we call `filterDupDirectories()` only if the user has requested a listing only of duplicates, before iterating the list to print out the results.

```
        if *dupFlag {
            filterDupDirectories(hashes)
        }
```