



Published in Better Programming



Yair Fernando

Follow

May 30 · 5 min read · [Listen](#)



File Processing Using Concurrency With GoLang

Master concurrency in Go





GoLang has incredible support for concurrent programs and in this article, we'll see how we can optimize a program that processes a CSV file to send SMS notifications to its users.

If you're new to using GoLang and want to get a better understanding of how concurrency works, I'd recommend reading this article first: [Concurrency in GoLang, Goroutines, and Channels Explained](#).

For this writing, we'll be using a CSV file and the purpose of this program is to read the file and process its data. The file content represents a list of 3,000 users.

The program should read this file and send a notification to each user and their friends as well.

You can find the source code in this Github [repository](#) for reference.

All right, let's get started. We'll first create a Go module, the `main.go` file, and a package called CSV. This package will have a method called `ProcessFile` which will be called from the main package. Let's create the Go module first.

```
go mod init github.com/GithubHandle/fileProcessing
```





```
csv/  
  csv.go  
  students.csv  
go.mod  
main.go
```

Folder Structure

As you can see there is also a `students.csv` file, you can grab this file from the repository. The `main.go` and the `csv.go` files look like this:

```
package main  
  
import "github.com/YairFernando67/fileProcessing/csv"  
  
func main() {  
    csv.ProcessFile()  
}
```

• • •

```
package csv  
  
func ProcessFile() {
```





. . .

Now, let's open the file and read its content:

```
1  package csv
2
3  import (
4      "bufio"
5      "fmt"
6      "log"
7      "os"
8      "strings"
9  )
10
11  const (
12      FILE_NAME = "csv/students.csv"
13  )
14
15  func ProcessFile() {
16      f, err := os.Open(FILE_NAME)
17      if err != nil {
18          log.Fatal(err)
19      }
20
21      users := scanFile(f)
22      fmt.Printf("users %v\n", users)
23  }
24
25  func scanFile(f *os.File) []*User {
26      s := bufio.NewScanner(f)
```





```
32         ids = ids[1 : len(ids)-1]
33         user := &User{
34             Id:      lineArray[0],
35             Name:     lineArray[1],
36             LastName: lineArray[2],
37             Email:    lineArray[3],
38             Phone:    lineArray[4],
39             FriendIds: ids,
40         }
41         users = append(users, user)
42     }
43     return users
44 }
```

csv1.go hosted with ❤ by GitHub

[view raw](#)

In the above code, we're first opening the `csv` file using the `os` package. Then the file is passed to another function called `scanFile`. This function uses the `bufio` package to initialize a new scanner and scan each line of the file. For each line, we extract the information about the user and append it to a slice of users. Once the scan finishes, the function returns the slice of users to the caller.

The user struct is defined in `csv/user.go`, and it looks like this.

```
package csv
```





If you run the program, you'll see the slice of users being printed out to the console.

. . .

Sweet, we now have the content of the file represented as a slice of users, we can now use this slice to send an SMS notification to each of these users and their friends.

For that, we'll first see how to do this sequentially, without using concurrency. Then we'll modify the program to make it faster.

```
1  package csv
2
3  import (
4      "bufio"
5      "fmt"
6      "log"
7      "os"
8      "strings"
9      "time"
10 )
11
12 const (
13     FILE_NAME = "csv/students.csv"
14 )
```





```
20     }
21
22     users := scanFile(f)
23
24     // sequential processing
25     sequentialProcessing(users)
26 }
27
28 func sequentialProcessing(users []*User) {
29     visited := make(map[string]bool)
30     for _, user := range users {
31         if !visited[user.Id] {
32             visited[user.Id] = true
33             sendSmsNotification(user)
34             for _, friendId := range user.FriendIds {
35                 friend, err := findUserById(friendId, users)
36                 if err != nil {
37                     fmt.Printf("Error %v\n", err)
38                     continue
39                 }
40
41                 if !visited[friend.Id] {
42                     visited[friend.Id] = true
43                     sendSmsNotification(friend)
44                 }
45             }
46         }
47     }
48 }
49
50 func sendSmsNotification(user *User) {
51     time.Sleep(10 * time.Millisecond)
```





```
57         if user.Id == userId {
58             return user, nil
59         }
60     }
61
62     return nil, fmt.Errorf("User not found with id %v", userId)
63 }
```

csv2.go hosted with ❤ by GitHub

[view raw](#)

For the sequential processing, we have created a function and passed the users into it. This function ranges over the users and for each user it checks if it has already been visited before, if it has not been visited, it marks the user as visited and sends the SMS notification. Then it also ranges over the user's friend `ids`, find each user, and performs the same steps, checking if it has been visited or not, marking them as visited, and sending the notification.

In the `sendSmsNotification` function, we are using the `time.Sleep` function to simulate some latency in sending the notification.

Let's run a benchmark test on this version of the program and see how fast it is. Here's the code:

```
1 package csv
```





```
8         for i := 0; i < b.N; i++ {  
9             ProcessFile()  
10        }  
11    }
```

csv_test.go hosted with ❤ by GitHub

[view raw](#)

Let's run `go test -bench=. github.com/YairFernando67/fileProcessing/csv -`
`benchtime=5x`

```
Sending sms notification to 1-389-100-5523  
Sending sms notification to (327)695-8002  
Sending sms notification to 1-375-489-0361  
Sending sms notification to 695-072-7369  
Sending sms notification to 1-330-843-8877  
Sending sms notification to 868.483.2936  
Sending sms notification to (175)806-6739  
Sending sms notification to 1-702-594-5038  
5          33281310842 ns/op  
PASS  
ok      github.com/YairFernando67/fileProcessing/csv    199.723s
```

Sequential processing benchmark

It took `199.723` seconds to process all users. Let's see how we can improve the performance of the program using concurrency. For that, we'll add another method to process the users concurrently.





```
6     "log"
7     "os"
8     "strings"
9     "time"
10 )
11
12 const (
13     FILE_NAME      = "csv/students.csv"
14     MAX_GOROUTINES = 10
15 )
16
17 func ProcessFile() {
18     f, err := os.Open(FILE_NAME)
19     if err != nil {
20         log.Fatal(err)
21     }
22
23     users := scanFile(f)
24
25     // sequential processing
26     // sequentialProcessing(users)
27
28     // concurrent processing
29     concurrentProcessing(users)
30 }
31
32 func concurrentProcessing(users []*User) {
33     usersCh := make(chan []*User)
34     unvisitedUsers := make(chan *User)
35     go func() { usersCh <- users }()
36     initializeWorkers(unvisitedUsers, usersCh, users)
37     processUsers(unvisitedUsers, usersCh, len(users))
}
```





```
43         for user := range unvisitedUsers {
44             sendSmsNotification(user)
45             go func(user *User) {
46                 friendIds := user.FriendIds
47                 friends := []*User{}
48                 for _, friendId := range friendIds {
49                     friend, err := findUserById(friendId, users)
50                     if err != nil {
51                         fmt.Printf("Error %v\n", err)
52                         continue
53                     }
54                     friends = append(friends, friend)
55                 }
56
57                 _, ok := <-usersCh
58                 if ok {
59                     usersCh <- friends
60                 }
61             }(user)
62         }
63     }()
64 }
65 }
66
67 func processUsers(unvisitedUsers chan<- *User, usersCh chan []*User, size int) {
68     visitedUsers := make(map[string]bool)
69     count := 0
70     for users := range usersCh {
71         for _, user := range users {
72             if !visitedUsers[user.Id] {
73                 visitedUsers[user.Id] = true
74                 count++

```





```
80         }  
81     }  
82 }
```

csv3.go hosted with ❤ by GitHub

[view raw](#)

For this concurrent implementation, we have created two channels, `usersCh` will hold the initial list of users and `unvisitedUsers` will hold individual unvisited users.

In line 35, we are feeding the first channel with the initial list of users that we got as a parameter into this function. This runs in a separate goroutine, because we don't want the main goroutine to be blocked, this is a concept that I talked about in this [article](#), you can go check it out if you're still confused about the blocking concepts.

Then we call `initializeWorkers`. This function essentially initializes N number of goroutines determined by the constant `MAX_GOROUTINES`, in this case, we'll start with 10. Each worker is a function that listens on the `unvisitedUsers` channel and for each user that it receives, it sends the SMS notification to the user and also processes its friend ids by finding each user in the list and then sending the users to the `usersCh` channel. The processing of the user's friend `ids` is run in a separate goroutine since we don't want to block the current goroutine here as





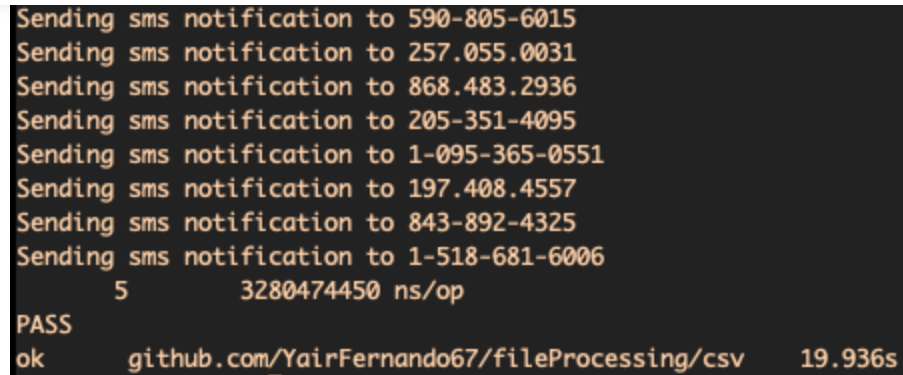
This function will allow the program to have 10 goroutines waiting for users to be sent to the `unvisitedUsers` channel. And each of these goroutines will run concurrently, thus improving the performance of the program.

Another important thing to note here is that in line 57 we are using this syntax to check if the `usersCh` channel is still open, this prevents us from sending data to a closed channel. In this example, this is important since we'll be closing the channel and we don't want other goroutines to attempt to send data to this channel.

Then in line 37, we have the `processUsers` function which ranges over the `usersCh` channel, and for each list of users that it receives, it checks if the user has already been visited, if not then it marks it as visited and sends the user to the `unvisitedUsers` channel. This function also keeps track of how many users have processed with the `count` variable, by doing this, we can check if we have reached the size in line 75 and close the `usersCh` channel which will terminate the program.

Let's run the benchmark test for this version of the program and see how much it improved.





```
Sending sms notification to 590-805-6015
Sending sms notification to 257.055.0031
Sending sms notification to 868.483.2936
Sending sms notification to 205-351-4095
Sending sms notification to 1-095-365-0551
Sending sms notification to 197.408.4557
Sending sms notification to 843-892-4325
Sending sms notification to 1-518-681-6006
5          3280474450 ns/op
PASS
ok          github.com/YairFernando67/fileProcessing/csv    19.936s
```

Concurrent processing benchmark

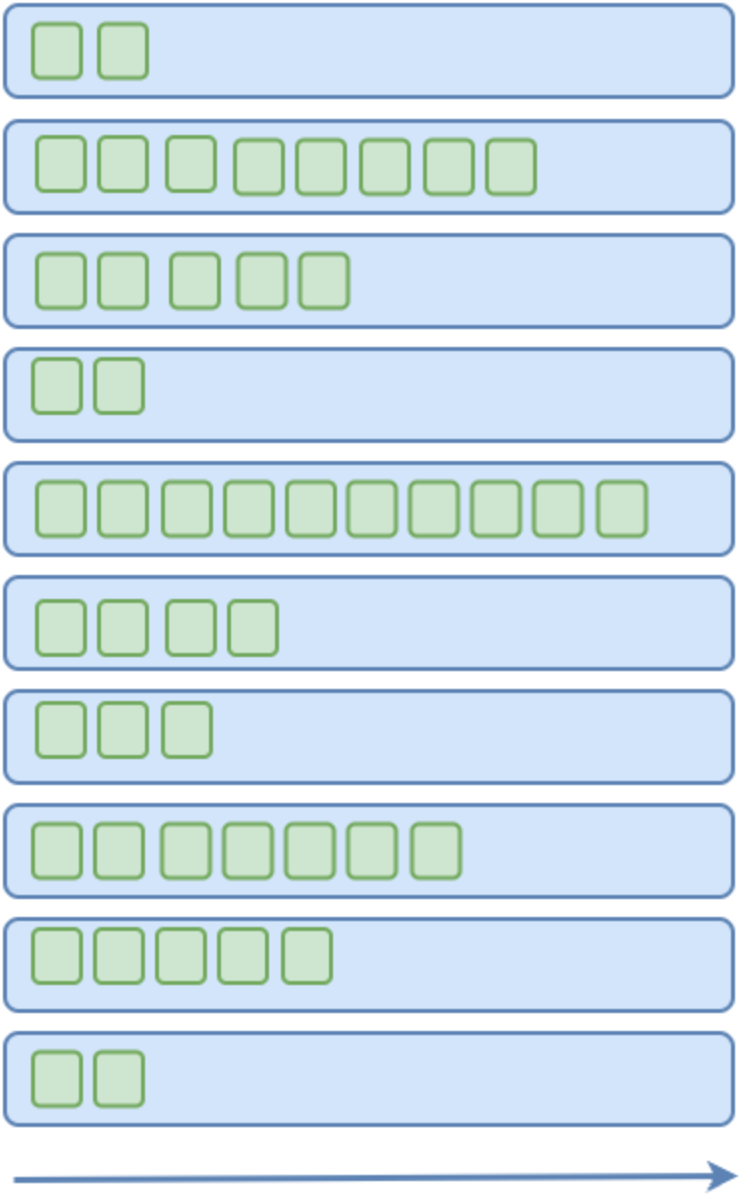
As you can see, it only took `19.936` seconds to finish!! This is a huge performance improvement. We can also control how many active goroutines/workers we want the program to have by increasing or decreasing the `MAX_GOROUTINES` constant.

If we increase the number of workers, our program will run faster since more tasks will be running concurrently.





Workers





• • •

I hope you found this writing useful and learn something new. Thank you for reading. Stay tuned.

Thanks to Anupam Chugh

Get an email whenever Yair Fernando publishes.

 **Subscribe**

Emails will be sent to jon_monts@hotmail.com.

[Not you?](#)

