Implementing **Concurrency** with **Go**

Gophers: Trevor English,
Nancy Everest, Allie Peterson

WHY GO?

WHY WAS GO INTRIGUING TO US?

- Becoming more popular as use of API’s grow
- Known for its concurrency features
- Known for being fast in compile time, coding time, and runtime
- Good at interacting with and connecting different systems

OUR IDEA

We knew we wanted to learn Go, so we built a project that would make use of its strengths.

Excited about implementing concurrency, we decided to attempt to pull an extremely large amount of data from a database serially and concurrently and compare how long each one took.

Our hope and expectation was to see the concurrent time be significantly less than the linear and to use Go’s built-in support to achieve that.

We also found that Go supports sqlite databases and has a way to display maps using Leaflet (cool!)

OUR PROJECT

- To try to maximize showing the difference of using concurrency, we chose to use a large dataset of US wildfires.
- This dataset has 2.3 million records and represents 180 million acres of burn area from 1992-2020
- Downloaded a sqlite zip file with the fire records
- Data source: Forest Service Fire Data from the US Department of Agriculture

INTERACTING WITH THE DATABASE

- Opened and viewed the data using DBeaver
- Opening connection to SQLite
- We then were able to query the rows for fire name, size, latitude, longitude, year it occurred

INTERACTING WITH THE DATABASE

```
import (
    "database/sql"
    "encoding/json"
```

```
func fetchSerialFireData() ([]Fire, error) {
    firedb, err := sql.Open("sqlite3", "../../internal/db/FPA_FOD_20221014.sqlite")
    if err != nil {
        return nil, err
    }
    defer firedb.Close()

    query, err := firedb.Query(`SELECT FIRE_NAME, FIRE_SIZE, LATITUDE, LONGITUDE, FIRE_YEAR,
                                       NWCG_REPORTING_UNIT_NAME, NWCG_GENERAL_CAUSE, FIPS_NAME
                                FROM Fires
                                LIMIT 1000`)
    if err != nil {
        return nil, err
    } else {
        fmt.Println("Serial query successful")
    }
    defer query.Close()

    var fires []Fire
    for query.Next() {
        var fire Fire
        query.Scan(&fire.Name, &fire.FireSize, &fire.Latitude, &fire.Longitude, &fire.Year,
                   &fire.Forest, &fire.Cause, &fire.County)
        log.SetFlags(0)
        fires = append(fires, fire)
    }
    return fires, nil
}
```

INTERACTING WITH THE DATABASE

```go
func fetchSerialFireData() ([]Fire, error) {
    firedb, err := sql.Open("sqlite3", "../../internal/db/FPA_FOD_20221014.sqlite")
    if err != nil {
        return nil, err
    }
    defer firedb.Close()

    query, err := firedb.Query(`SELECT FIRE_NAME, FIRE_SIZE, LATITUDE, LONGITUDE, FIRE_YEAR,
                                       NWCG_REPORTING_UNIT_NAME, NWCG_GENERAL_CAUSE, FIPS_NAME
                                FROM Fires
                                LIMIT 1000`)
    if err != nil {
        return nil, err
    } else {
        fmt.Println("Serial query successful")
    }
    defer query.Close()

    var fires []Fire
    for query.Next() {
        var fire Fire
        query.Scan(&fire.Name, &fire.FireSize, &fire.Latitude, &fire.Longitude, &fire.Year,
                   &fire.Forest, &fire.Cause, &fire.County)
        log.SetFlags(0)
        fires = append(fires, fire)
    }
    return fires, nil
}
```

```go
var wg sync.WaitGroup
defer wg.Wait()
wg.Add(1)
go sendToConcurrentWebSocket(conn, &wg)
```

```go
func fetchConcurrentFireData() ([]Fire, error) {
    firedb, err := sql.Open("sqlite3", "../../internal/db/FPA_FOD_20221014.sqlite")
    if err != nil {
        fmt.Println(err)
        return nil, err
    }
    defer firedb.Close()

    query, err := firedb.Query(`SELECT FIRE_NAME, FIRE_SIZE, LATITUDE, LONGITUDE, FIRE_YEAR,
                                       NWCG_REPORTING_UNIT_NAME, NWCG_GENERAL_CAUSE, FIPS_NAME
                                FROM Fires
                                LIMIT 1000
                                `)
    if err != nil {
        return nil, err
    } else {
        fmt.Println("Concurrent query successful")
    }
    defer query.Close()

    var wg sync.WaitGroup
    jobs := make(chan []Fire, 200)
    results := make(chan Fire, 200)

    for i := 0; i < 10; i++ {
        wg.Add(1)
        go worker(jobs, results, &wg)
    }
    var fires []Fire

    go func() {
        for query.Next() {
            var fire Fire
            query.Scan(&fire.Name, &fire.FireSize, &fire.Latitude, &fire.Longitude,
                       &fire.Year, &fire.Forest, &fire.Cause, &fire.County)
            data := []Fire{fire}
            jobs <- data
            fires = append(fires, fire)
        }
        close(jobs)
    }()

    go func() {
        wg.Wait()
        close(results)
    }()
    go func() {
        wg.Wait()
    }()

    var resultFires []Fire
    for result := range results {
        resultFires = append(resultFires, result)
    }
    return resultFires, nil
}
```

OUR MAIN

- Establish HTTP connection to localhost:8080/ and render the HTML template to it with net/http package
- Then establish both /serial and /concurrent connections that trigger the handlers. These are called in the JS script of mapPage.html

```go
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        http.ServeFile(w, r, "mapPage.html")
    })
    http.HandleFunc("/serial", serialWebsocketHandler)
    http.HandleFunc("/concurrent", concurrentWebsocketHandler)

    if err := http.ListenAndServe(":8080", nil); err !=
        log.Fatal(err)
    }
}
```

JS

```js
const wsConcurrent = new WebSocket("ws://localhost:8080/concurrent");
wsConcurrent.onmessage = (event) => handleConcurrentWS(event, mapConcurrent);

const wsSerial = new WebSocket("ws://localhost:8080/serial");
wsSerial.onmessage = (event) => handleSerialWS(event, mapSerial);
```

ESTABLISH WEBSOCKET CONNECTIONS

Using Gorilla Websocket

```go
var upgrader = websocket.Upgrader{
    ReadBufferSize:  1024,
    WriteBufferSize: 1024,
    CheckOrigin: func(r *http.Request) bool {
        return true
    },
}
```

```go
func serialWebsocketHandler(writer http.ResponseWriter, request *http.Request) {
    conn, err := upgrader.Upgrade(writer, request, nil)
    if err != nil {
        log.Println(err)      You, 12 hours ago • Working
        return
    }
    defer conn.Close()

    var wg sync.WaitGroup
    defer wg.Wait()

    wg.Add(1)
    go sendToSerialWebSocket(conn, &wg)
}

func concurrentWebsocketHandler(writer http.ResponseWriter, request *http.Request) {
    conn, err := upgrader.Upgrade(writer, request, nil)
    if err != nil {
        log.Println(err)
        return
    }
    defer conn.Close()

    var wg sync.WaitGroup
    defer wg.Wait()

    wg.Add(1)
    go sendToConcurrentWebSocket(conn, &wg)
}
```

SEND FIRE DATA THROUGH BOTH WEBSOCKETS

```go
func sendToSerialWebSocket(conn *websocket.Conn, wg *sync.WaitGroup) {
    defer wg.Done()

    fires, err := fetchSerialFireData()
    if err != nil {
        log.Println("Error fetching data serially from database:", err)
        return
    }

    firesJSON, err := json.Marshal(fires)
    if err != nil {
        log.Println("Error marshaling serial fires:", err)
        return
    }

    err = conn.WriteMessage(websocket.TextMessage, firesJSON)
    if err != nil {
        log.Println("Error sending serial fires through WebSocket:", err)
        return
    }
    fmt.Println("Websocket successfully sent serial data")

}
```

```go
func sendToConcurrentWebSocket(conn *websocket.Conn, wg *sync.WaitGroup) {
    defer wg.Done()

    fires, err := fetchConcurrentFireData()
    if err != nil {
        log.Println("Error fetching data concurrently from database:", err)
        return
    }

    firesJSON, err := json.Marshal(fires)
    if err != nil {
        log.Println("Error marshaling concurrent fires:", err)
        return
    }

    err = conn.WriteMessage(websocket.TextMessage, firesJSON)
    if err != nil {
        log.Println("Error sending concurrent fires through WebSocket:", err)
        return
    }
    fmt.Println("Websocket successfully sent concurrent data")
}
```

var err error

“CONCURRENT” FETCH

```go
func fetchConcurrentFireData() ([]Fire, error) {
    firedb, err := sql.Open("sqlite3", "../../internal/db/FPA_FOD_20221014.sqlite")
    if err != nil {
        fmt.Println(err)
        return nil, err
    }

    defer firedb.Close()

    query, err := firedb.Query(`SELECT FIRE_NAME, FIRE_SIZE, LATITUDE, LONGITUDE, FIRE_YEAR,
                                       NWCG_REPORTING_UNIT_NAME, NWCG_GENERAL_CAUSE, FIPS_NAME
                                FROM Fires
                                LIMIT 1000
                                `)
    if err != nil {
        return nil, err
    } else {      You, 1 hour ago • more columns & marker styling
        fmt.Println("Concurrent query successful")
    }

    defer query.Close()
```

THE WORKER POOL FUNCTION

```
var wg sync.WaitGroup
jobs := make(chan []Fire)
results := make(chan Fire)

for i := 0; i < 10; i++ {
    wg.Add(1)
    go worker(jobs, results, &wg)
}
```

```
func worker(jobs <-chan []Fire, results chan<- Fire, wg *sync.WaitGroup) {
    defer wg.Done()
    for job := range jobs {
        results <- job[0]
    }
}
```

THE WORKER POOL FUNCTION

```go
go func() {
    for query.Next() {
        var fire Fire
        query.Scan(&fire.Name, &fire.FireSize, &fire.Latitude, &fire.Longitude, &fire.Year, &fire.Forest, &fire.Cause, &fire.County)
        data := []Fire{fire}
        jobs <- data
    }
    close(jobs)
    wg.Wait()
    close(results)
}()

var fires []Fire

for result := range results {
    fires = append(fires, result)
}

return fires, nil
```

```go
var results chan<- Fire
```

```go
type Fire struct {
    Name      string
    FireSize  string
    Latitude  string
    Longitude string
    Year      string
    Forest    string
    Cause     string
    County    string
}
```

SETTING UP A WEBSERVER WITH GO

Connected leaflet to display maps on our local web server

- In HTML, CSS, and JS files

```
<head>
  <link rel="stylesheet" href="https://unpkg.com/leaflet@1.9.4/dist/leaflet.css"
        integrity="sha256-p4NxAoJBhIIN+hmNHrzRCf9tD/miZyoHS5obTRR9BMY="
        crossorigin=""/>
  <script src="https://unpkg.com/leaflet@1.9.4/dist/leaflet.js"
          integrity="sha256-20nQCchB9co0qIjJZRGuk2/Z9VM+kNiyxNV1lvTlZBo="
          crossorigin=""></script>
  <title>Go Gophers Maps</title>
```

```
<div class="map-container">
  <h1>Serialized Loading</h1>
  <div id="mapSerial" style="height:700px;"></div>
  <script type="text/javascript" charset="utf-8">

    var mapSerial = L.map("mapSerial").setView([39, -95], 4);

    L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
      maxZoom: 19,
      attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>
    }).addTo(mapSerial);

    const wsSerial = new WebSocket("ws://localhost:8080/serial");
    wsSerial.onmessage = (event) => handleWebSocketMessage(event, mapSerial);
  </script>
</div>
```

Used websockets to display our data on the graphs

DEMO

FILE OUTPUT VERSION (NO LIMIT)

```
Waiting for goroutines to finish...
Concurrent finished
callConcurrent took   5.906271 seconds
Linear finished
callLinear took   8.212647 seconds
Finished.
```

go.mod gogopherfires 2 | main.go app M | concurrentOut app

gogopherfires > cmd > app > concurrentOut

```
2303534  Fire: 2019-3238A 3000 39.21151 -96.76109 2019
2303535  Fire: KENNEDY PEAK 745 38.755556 -78.46694 2019
2303536  Fire: KAHANA RIDGE 931 20.965 -156.6693333 2019
2303537  Fire: SANDALWOOD 1011 33.99245999 -117.05921 2019
2303538  Fire: SANDHILL 350 38.06844 -97.62863 2019
                21 38.10375 -102.412 2019
                 32.13259 -110.4073 2019
                40.710117 -122.25451699 2019
                50 32.251667 -110.0686 2019
                .3372222 -119.05333329 2019
                5.3075 -119.964444 2019
                 38.46475 -104.8547 2019
                250 43.4767 -110.7794 2019
                 38.11222199 -107.6014 2019
                .1 48.01952 -120.8539 2019
                .7266 -112.0244 2019
2303550  Fire: BARREN HILL 1592 46.236944 -114.9828 2019
2303551  Fire: SAN RAFAEL 438 31.42304 -110.571 2019
2303552  Fire: ROCK 2422 37.4722222 -121.2494444 2019
2303553  Fire: BEAVER POND 168 31.49333 -88.74028 2019
2303554  Fire: CONNEX WF 970 30.52333329 -86.7816667 2019
2303555  Fire: 2019-3354 1000 36.71164 -96.74075 2019
2303556  Fire: BEN HOWARD HOLLOW 272 36.8502778 -83.5066667 2019
2303557  Fire: WALKER 54608 40.05325 -120.6689 2019
2303558  Fire: OK 745 413 32.99723 -87.30439 2019
2303559  Fire: 204 COW 9668 44.28505 -118.4598 2019
2303560  Fire: CAMERA 401 36.30383 -94.90382 2020
2303561  Fire: 22ND AVE SE 1000 26.1911111 -81.5238889 2020
2303562  Fire: JONES 1 39.03789 -108.9595 2020
2303563  Fire: POWER 100 37.1486111 -119.50305559 2020
2303564  Fire: 12 MILE 50 46.15137 -114.4428 2020
2303565  Fire: TAYLOR POND 24892 46.67034 -120.1145 2020
2303566  Fire: MIDDLE MOUNTAIN 105 38.5789 -79.14845 2020
2303567
```

go.mod gogopherfires 2 | main.go app M | linearOut app | FPA_FO

gogopherfires > cmd > app > linearOut

```
2303534  Fire: 2019-3238 3000 39.19603 -96.72559 2019
2303535  Fire: KAHANA RIDGE 931 20.965 -156.6693333 2019
2303536  Fire: KENNEDY PEAK 745 38.755556 -78.46694 2019
2303537  Fire: SANDALWOOD 1011 33.99245999 -117.05921 2019
2303538  Fire: SANDHILL 350 38.06844 -97.62863 2019
2303539  Fire: EAGLE NEST 4521 38.10375 -102.412 2019
2303540  Fire: WAR BONNET 1450 32.251667 -110.0686 2019
2303541  Fire: BUCKHORN 1900 32.13259 -110.4073 2019
2303542  Fire: MOUNTAIN 600 40.710117 -122.25451699 2019
2303543  Fire: MARIA 9999 34.3372222 -119.05333329 2019
2303544  Fire: BELMONT 835 35.3075 -119.964444 2019
2303545  Fire: STONE CITY 30 38.46475 -104.8547 2019
2303546  Fire: SADDLE BUTTE 250 43.4767 -110.7794 2019
2303547  Fire: COW CREEK 859 38.11222199 -107.6014 2019
2303548  Fire: MAPLE CREEK 0.1 48.01952 -120.8539 2019
2303549  Fire: JOMAX 1145 33.7266 -112.0244 2019
2303550  Fire: BARREN HILL 1592 46.236944 -114.9828 2019
2303551  Fire: SAN RAFAEL 438 31.42304 -110.571 2019
2303552  Fire: ROCK 2422 37.4722222 -121.2494444 2019
2303553  Fire: CONNEX WF 970 30.52333329 -86.7816667 2019
2303554  Fire: BEAVER POND 168 31.49333 -88.74028 2019
2303555  Fire: BEN HOWARD HOLLOW 272 36.8502778 -83.5066667 2019
2303556  Fire: 2019-3354 1000 36.71164 -96.74075 2019
2303557  Fire: WALKER 54608 40.05325 -120.6689 2019
2303558  Fire: OK 745 413 32.99723 -87.30439 2019
2303559  Fire: 204 COW 9668 44.28505 -118.4598 2019
2303560  Fire: CAMERA 401 36.30383 -94.90382 2020
2303561  Fire: 22ND AVE SE 1000 26.1911111 -81.5238889 2020
2303562  Fire: JONES 1 39.03789 -108.9595 2020
2303563  Fire: POWER 100 37.1486111 -119.50305559 2020
2303564  Fire: 12 MILE 50 46.15137 -114.4428 2020
2303565  Fire: TAYLOR POND 24892 46.67034 -120.1145 2020
2303566  Fire: MIDDLE MOUNTAIN 105 38.5789 -79.14845 2020
2303567
```

CONCURRENCY DIFFERENCE

For 1000 records to load (websocket limit)

- ~5 seconds for concurrency
  - Work is spread between 14 to 21 goroutines

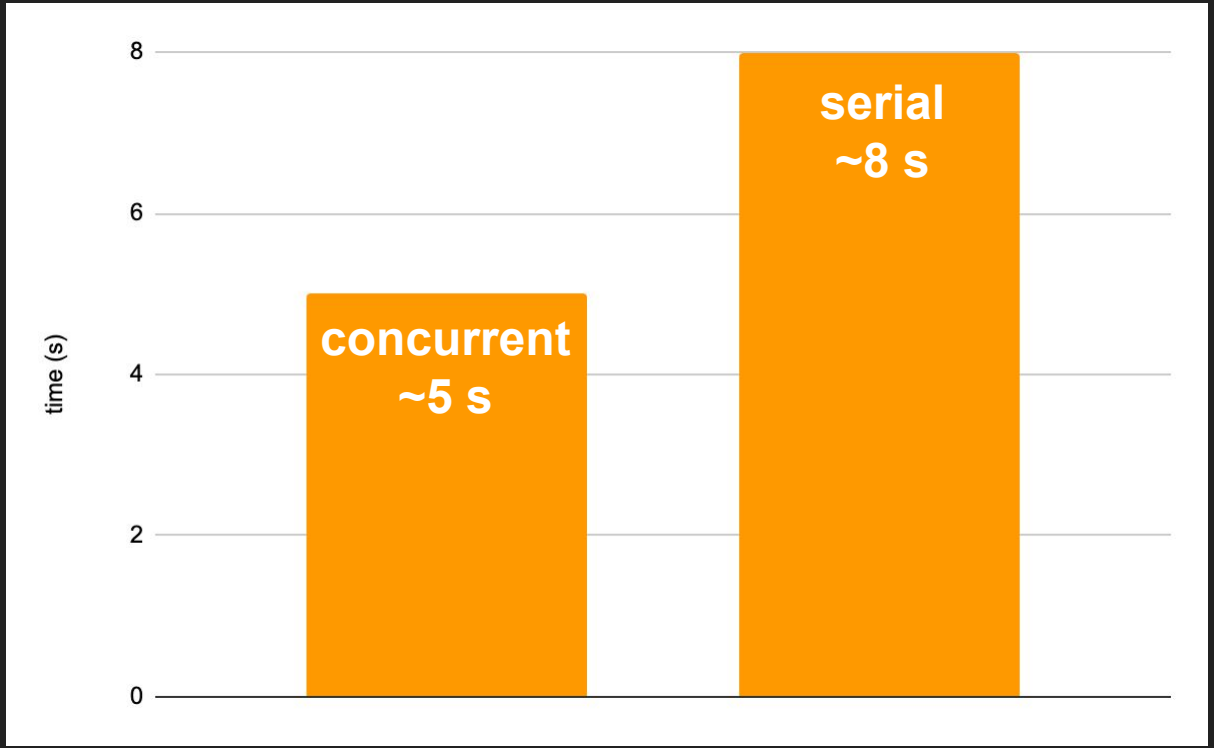
```
fmt.Printf("Goroutines: %d\n", runtime.NumGoroutine())
```

```
Goroutines: 21
```

- ~8 seconds for serial

CONCURRENCY DIFFERENCE

For 1000 records to load



SO, GO?

READABILITY– SIMPLE

- No operator overloading
- Clear what is going on and is generally similar to other high-level programming languages syntactically (like C)
- “go” with a function is used to start goroutines

WRITABILITY– SIMPLE:

- Fast coding time due to simplicity
- Orthography is similar to other languages and is as expected
- Concurrency was simple and clear to implement
- Wait groups
- Lots of packages

SO, GO?

RELIABILITY:

- Statically typed
- Exception handling– errors as return types!
- Aliasing– Go doesn’t allow direct aliasing but does allow type aliases
- Garbage collection

COST

- Garbage collection
- **Goroutines are used instead of threads– lightweight**

SO, GO?

STRENGTHS:

- Fast to code (lots of packages)
- Simplicity and speed of concurrency implementation (relatively)
- Easy integration with other systems (web pages, databases)

WEAKNESSES:

- Concurrency wait groups were a little difficult to understand load order and timers
- Using fmt.Println for debugging meddled with the concurrency (wait groups)

CONCLUSIONS
(some things we learned)

- Were able to accomplish our goal of creating a clickable map of a large amount of fire data
  - This could be reused for different datasets
- Limitation of Leaflet Markers
  - Testing was limited by about 1000 records due to the website being overloaded
  - Might try using geojson format to output to a file then load to map
- Difference between concurrent and serial not huge
  - Limited by webpage rendering

CREDITS & RESOURCES

- Go
  - Tutorial: Get started with Go: https://go.dev/doc/tutorial/getting-started
  - https://www.bairesdev.com/blog/why-golang-is-so-fast-performance-analysis/
- Creating a web server with Golang
  - https://blog.logrocket.com/creating-a-web-server-with-golang/
- Leaflet - free open-source javascript mapping library
  - leafletjs.com
  - go-leaflet: https://pkg.go.dev/github.com/ctessum/go-leaflet
- Websockets
  - Golang websockets video: https://www.youtube.com/watch?v=G8SKhZMqvsE
- Fire Data
  - Forest Service Fire Data from the US Department of Agriculture https://www.fs.usda.gov/rds/archive/catalog/RDS-2013-0009.6

Questions?

Thank you!