PSC Report

Massimiliano Baglioni

September 5, 2024

1 Assignment

Write a Google Go program that shuffle the words in a text by exploiting go-routines running in parallel and message passing communication only.

2 Report

In this brief report, I will explain the rationale behind my specific implementation, discuss the challenges encountered, and provide a short guide on how to use the program.

The primary objective was to shuffle an input text using Go routines and message-passing communication exclusively. The program begins by calculating the number of words each worker will handle, referred to as a segment. This ensures that each worker processes a roughly equal portion of the text. For example, if the text contains 22 words and the program utilizes 3 workers, two routines will handle 7 words each, while one will process 8.

```
1  func createSegments(listLen int, noWorkers int) []int {
2
3      if noWorkers <= 0 {
4          panic("Values error in slices function")
5      }
6
7      if listLen < noWorkers {
8          panic("less words than workers")
9      }
10
11     segmentsSize := listLen / noWorkers
12
```

```
13    //Extra remainder words.
14      extra := listLen % noWorkers
15
16      segmentsList := make([]int, noWorkers)
17
18      for i := 0; i < noWorkers; i++ {
19          end := segmentsSize
20
21      //Distriburtes the remainder to the first slices.
22          if extra > 0 {
23              end++
24              extra--
25          }
26          segmentsList[i] = end
27      }
28
29      return segmentsList
30  }
```

Listing 1: *Function that creates the slice with the number of words per routine.*

The assignment of words is the first layer of randomization. To distribute the words, the program creates a shared unbuffered channel. Each routine receives one word at a time from this channel. Since the operation is blocking, the distribution of words becomes non-deterministic due to concurrency. Initially, I attempted to use a buffered channel, but this led to words being assigned almost sequentially due to the behavior of the Go scheduler. Therefore, I opted for an unbuffered channel. However, even this approach was insufficient, as segments still contained many contiguous words. This is because the Go scheduler minimizes unnecessary swapping between routines.

```
1       go func() {
2           for _, word := range wordsList {
3               initialChan <- word
4           }
5           close(initialChan)
6       }()
```

Listing 2: *Routine that fills the channel with words for the routines.*

Once the words are assigned, each worker performs a local shuffle. After this local shuffle, the only task remaining is aggregation. To achieve this, a channel

is created for each worker, and these channels are grouped in a slice. These channels are then used for the aggregation process. Each routine writes every word to a randomly chosen channel. Once all words are written, the routines terminate, and the results are aggregated into a single list.

As mentioned earlier, the scheduler's tendency to minimize swapping between routines meant that the initial results were unsatisfactory, with many contiguous words appearing in blocks corresponding to the segments assigned to each worker. To achieve a more random and unpredictable shuffle, I applied the same algorithm a second time, using the already shuffled list and a different random number of workers.

```
shuffledText := textShuffle(textShuffle(wordsList,
    workersNumber1), workersNumber2)
```

Listing 3: *Second shuffle starting from the already shuffled text.*

3 Usage

The program writes the entire shuffled text to the file `output.txt`. The initial text to shuffle is located in the `text.txt` file, so if the user wants to shuffle custom text, they can simply edit the `text.txt` file.

The user can also choose the number of workers that will shuffle the text. Since the program performs two shuffles, it is possible to configure both sets of parameters. To do this, remove the comments from the following lines:

```
// Comment the random generation to hardcode the number of
    routines.
// workersNumber1 := 3
// workersNumber2 := 3
```

And then comment out the random assignment lines:

```
workersNumber1 := randomWorkerNumber(len(wordsList), 10)
workersNumber2 := randomWorkerNumber(len(wordsList), 10)
```

To run the code, simply use the command `go run main.go`, which will be sufficient to execute the program.

As mentioned earlier, the output will be appended to the `output.txt` file, with each shuffle separated by dashes. Inside the main function, there is a commented-out print statement that, if uncommented, will print the entire shuffled text to the terminal.

4 Conclusion

In summary, the primary challenge was that words tended to remain in blocks corresponding to the size of the worker segments, resulting in only local shuffling within these blocks. The interleaving of routines was insufficient to produce a completely unpredictable shuffle. To address this issue, I increased the computational complexity to enhance randomization. While the solution I implemented appears to be the most suitable for this case, I acknowledge that there may be more efficient approaches available.

Another approach I tried was using the switch statement, but it grouped words in bursts, making the output predictable and not as random as I was looking for.