Distributed and Parallel Technologies

Go Lab

0. Introduction

This Lab will

- familiarise you with some aspects of Go programming and concurrency in Go
- prepare you for the coursework

Important:

1. You should preserve the programs that you write for each exercise as they may be developed further in subsequent exercises.

2. Don't retype skeleton programs: you can paste program text from this lab sheet.

3. Strongly recommend that you follow the Go idiom of creating each program in it's own directory, e.g. factorial/factorial.go

1. Installing/Using Go

Go is installed and ready to use on the School Unix servers.

To install Go on other platforms follow the instructions at **https://golang.org/doc/install**.

Follow the "Test your Installation" instructions to check your installation by building and running a "hello world" program.

You can develop Go programs in pretty much any text editor, and compile and run from the command line (e.g. `%go build` or `%go run`), see lecture slides for examples. Alternately you may wish to use Go from an IDE. Go works well with VSCode and Visual Studio.

2. Go Programming

These exercises will familiarise you with Go programming.

2.1. A Factorial Function

Complete the `factorial` function in the following skeleton program.

```
package main

import (
  "fmt"
)

// Function to be written to compute the factorial of n
//
// factorial 0 = 1
// factorial n = n * factorial (n-1)
```

```
func factorial(n int64) int64 {
  fmt.Println("Should be a factorial function")
  return 0
}

func main() {
                                                    // Compute and output
factorial 16
  fmt.Println("Factorial 16", factorial(16))
}
```

2.2 A Perfect Number Function

A perfect number n is equal to the sum of the proper divisors of n. The first perfect number is 6, with divisors 1, 2 and 3. The next is 28 with divisors 1, 2, 4, 7 and 14.

Complete the `perfect` function in the following skeleton program.

```
package main

import (
  "fmt"
  "os"
  "strconv"
)

// Determine whether n is a perfect number

func perfect(n int64) bool {
    // To be written
  return false
}

func main() {
  var n int64
  var err error
                                                    // Read and argument
  if len(os.Args) < 2 {
    panic(fmt.Sprintf("Usage: must provide number as an argument"))
  }

  if n, err = strconv.ParseInt(os.Args[1],10,64) ; err != nil {
    panic(fmt.Sprintf("Can't parse first argument"))
  }

                            // Compute and output whether n is perfect
  fmt.Println(n, "is", perfect(n))
}
```

Hint: Do you need to consider every number less than `n` to be a potential factor?

2.3 Using Timing Information

Extend your perfect number program to record the elapsed time to compute and output the perfect function.

Hint 1: Use the `time` package, with `Now` and `Sub`

Hint 2: See lecture notes for an example

2.4 Perfect numbers up to n: `PerfectLoop`

Adapt your perfect number program to read a number n, and output all perfect numbers between 1 and n.

2.5 Performance Measurement

Record the times to compute all perfect numbers up to 1000, 2000, 4000, 8000, 16000

3. Concurrent Go Programming

These exercises will familiarise you with concurrency in Go.

3.1 Channels and deadlocks

In each case start from the following program with a 2-place buffered channel (from the Golang tutorial).

```
package main

import "fmt"

func main() {

	ch := make(chan int, 2)

	ch <- 1

	ch <- 2

	fmt.Println(<-ch)

	fmt.Println(<-ch)

}
```

a) Make the program deadlock

i) by adding a send

ii) by adding a receive

iii) by changing the buffering

b) Can you add the following *without* making the program deadlock

i) A send

ii) A receive

3.2 Simple Goroutine and Channel

Change your factorial program above as follows

1. Write a `factorialGo` function takes a channel argument, where it writes the result:

```
func factorialGo(n int64, c chan int64)
```

2. The main function

- creates a channel `c`
- launches `factorialGo` as a goroutine
- reads the result from `c` and prints it

Hint: See similar examples in the Golang (or other) tutorial.

3.3 Goroutines for Parallelism I: `perfectLoopPar`

Adapt your `perfectLoop` program from exercise 2.4 as follows.

1. It has a function `func perfectInterval(l,u int64, c chan int64)` that writes to channel `c` all perfect numbers between `l` and `u`.

2. The main function

- creates a channel `c`
- launches `perfectInterval` as a goroutine
- reads all results from `c` and prints them

Hint: You may need to use `range` and `close` the channel

3.4 Goroutines for Parallelism II

1. Now adapt your program from the previous section to launch 10 `perfectInterval` goroutines to compute `perfectInterval(1,1000,c), perfectInterval(1000,2000,c),… perfectInterval(9000,10000,c)`

Hint1: You will need to think carefully about how to indicate that each goroutine has completed.

Hint2: It's useful to print out the intervals that each `perfectInterval` goroutine is scanning

2. Compare the runtimes of `perfectLoop` and `perfectLoopPar`.