
# Universidad La Salle

## Fundamentos de Lenguajes de Programación

Informe de la Práctica 17

## Manipulación de Imágenes y Paralelismo en Go

Karlo Emigdio Pacha Curimayhua

Quinto Semestre - Ingeniería de Software

2022

# Ejercicio 1

## Enunciado

Implemente la adición de imágenes con los elementos de la Figura 1. Considere el problema de overflow en los pixeles, esto sucede cuando un pixel supera el valor de 255 o es inferior a 0; por ejemplo si un pixel resulta con un valor mayor a 255, solo le asigna el valor de 255.

Figure 1: Imágenes de muestra.

## Solución Secuencial

```go
package main

import (
  "fmt"
  "image"
  "image/color"
  "image/draw"
  "image/jpeg"
  "math"
  "os"
  "time"
)

func main() {
  path := "5.jpg"
  path2 := "6.jpg"

  start := time.Now()
  Sequential(path, path2)
  elapsed := time.Since(start)
  fmt.Println("Sequential time: ", elapsed)
}

func Sequential(path string, path2 string) {
  imagePath, err := os.Open(path)
  defer imagePath.Close()
  myImage, _, err := image.Decode(imagePath)

  imagePath2, err := os.Open(path2)
  defer imagePath2.Close()
  myImage2, _, err := image.Decode(imagePath2)
```

```
32
33    bounds := myImage2.Bounds()
34
35    size := myImage2.Bounds().Size()
36    rect := image.Rect(0, 0, size.X, size.Y)
37    wImg := image.NewRGBA(rect)
38
39    for x := 0; x < size.X; x++ {
40      for y := 0; y < size.Y; y++ {
41        pixel := myImage2.At(x, y)
42        originalColor := color.RGBAModel.Convert(pixel).(color.RGBA)
43
44        r := uint8(math.Min(255, float64(originalColor.R)*3.5))
45        g := uint8(math.Min(255, float64(originalColor.G)*3.5))
46        b := uint8(math.Min(255, float64(originalColor.B)*3.5))
47
48        c := color.RGBA{
49          R: r, G: g, B: b, A: originalColor.A,
50        }
51        wImg.Set(x, y, c)
52      }
53    }
54
55    union := image.NewAlpha(bounds)
56
57    for x := 0; x < bounds.Dx(); x++ {
58      for y := 0; y < bounds.Dy(); y++ {
59        union.SetAlpha(x, y, color.Alpha{uint8(75)})
60      }
61    }
62
63    myImage3 := image.NewRGBA(bounds)
64    draw.Draw(myImage3, myImage3.Bounds(), myImage, image.ZP, draw.
        Src)
65    draw.DrawMask(myImage3, bounds, wImg, image.ZP, union, image.ZP,
        draw.Over)
66
67    result, _ := os.Create("resultSequential1.jpg")
68    defer result.Close()
69    err = jpeg.Encode(result, myImage3, nil)
70    check(err)
71  }
72
73  func check(err error) {
74    if err != nil {
75      panic(err)
76    }
77  }
```

Figure 2: Imágenes de entrada y resultado.

## Solución Paralela

```
package main

import (
  "fmt"
  "image"
  "image/color"
  "image/draw"
  "image/jpeg"
  "math"
  "os"
  "sync"
  "time"
)

func main() {
  path := "1.jpg"
  path2 := "2.jpg"

  start := time.Now()
  Parallel(path, path2)
  elapsed := time.Since(start)
  fmt.Println("Parallel time: ", elapsed)
}

func Parallel(path string, path2 string) {
  imagePath, err := os.Open(path)
  defer imagePath.Close()
  myImage, _, err := image.Decode(imagePath)

  imagePath2, err := os.Open(path2)
  defer imagePath2.Close()
  myImage2, _, err := image.Decode(imagePath2)

  bounds := myImage2.Bounds()

  size := myImage2.Bounds().Size()
  rect := image.Rect(0, 0, size.X, size.Y)
  wImg := image.NewRGBA(rect)

  waitGroup := new(sync.WaitGroup)

  goroutines := 7

  for i := 0; i < goroutines; i++ {
    startX := (i) * size.X / goroutines
```

```
    endX := (i + 1) * size.X / goroutines
    waitGroup.Add(1)
    go func() {
      for x := startX; x < endX; x++ {
        for y := 0; y < size.Y; y++ {
          pixel := myImage2.At(x, y)
          originalColor := color.RGBAModel.Convert(pixel).(color.
    RGBA)

          r := uint8(math.Min(255, float64(originalColor.R)*3.5))
          g := uint8(math.Min(255, float64(originalColor.G)*3.5))
          b := uint8(math.Min(255, float64(originalColor.B)*3.5))

          c := color.RGBA{
            R: r, G: g, B: b, A: originalColor.A,
          }
          wImg.Set(x, y, c)
        }
      }
      defer waitGroup.Done()
    }()
  }
  waitGroup.Wait()

  union := image.NewAlpha(bounds)

  for i := 0; i < goroutines; i++ {
    endX := (i + 1) * bounds.Dx() / goroutines
    startX := (i) * bounds.Dx() / goroutines
    waitGroup.Add(1)
    go func() {
      for x := startX; x < endX; x++ {
        for y := 0; y < bounds.Dy(); y++ {
          union.SetAlpha(x, y, color.Alpha{uint8(75)})
        }
      }
      defer waitGroup.Done()
    }()
  }
  waitGroup.Wait()

  myImage3 := image.NewRGBA(bounds)
  draw.Draw(myImage3, myImage3.Bounds(), myImage, image.ZP, draw.
    Src)
  draw.DrawMask(myImage3, bounds, wImg, image.ZP, union, image.ZP,
    draw.Over)

  result, _ := os.Create("resultParallel1.jpg")
  defer result.Close()
  err = jpeg.Encode(result, myImage3, nil)
  check(err)
}

func check(err error) {
  if err != nil {
    panic(err)
  }
}
```

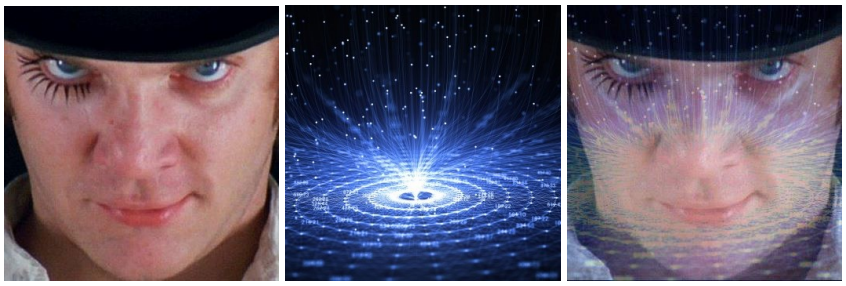

Figure 3: Imágenes de entrada y resultado.

### Análisis

| Implementación | Tiempo |
|---|---|
| Secuencial | 36.6312ms |
| Paralelo | 12.9794ms |

Table 1 : Tiempos de ejecución

La unión de imágenes no presenta difucultad alguna, la imagen que hará de máscara se necesita quemar. Esto se logra con la manipulación de los valores RGBA, aunque sólo basta los 3 primeros valores.
A pesar de varias pruebas, la implementación paralela es 3 veces mejor que la secuencial, se saca mayor provecho de los threads en los bucles, aunque sólo se debe limitar a un número pequeño (en este caso 7) dado la enorme tamaño de las imágenes.

## Ejercicio 2

### Enunciado

El operador blending, es un operador que suma dos imágenes, pero nosotros podemos decidir que imágen tendrá más presencia en el resultado. Implemente el operador Blending (ecuación 1) y evalue sus resultados con imágenes de su preferencia, tambien pruebe diferentes valores de x.

$$Q(i,j) = X * P_1(i,j) + (1 - X) * P_2(i,j) \quad (1)$$

Donde: $Q(i,j)$, es un pixel de la imagen resultado; $P_1(i,j)$, es un pixel de la imagen de entrada; $P_2(i,j)$, es un pixel de la otra imagen de entrada. // Por ejemplo, con las imágenes de entrada de la Figura 4, se puede obtener el resultado de la Figura 5, para un $x = 0,25, x = 0,5 y x = 0,75$.

Figure 4: Imágenes de entrada.

Figure 5: Imagen resultado de blending, para $x = 0,25, x = 0,5 y x = 0,75$.

## Solución Secuencial

```
package main

import (
  "fmt"
  "image"
  "image/color"
  "image/draw"
  "image/jpeg"
  "strconv"
  "time"

  "os"
)

func main() {
  path := "9.jpg"
  path2 := "10.jpg"

  start := time.Now()
  Sequential(path, path2)
  elapsed := time.Since(start)
  fmt.Println("Sequential time: ", elapsed)
}

func Sequential(path string, path2 string) {
```

```
  imagePath, err := os.Open(path)
  defer imagePath.Close()
  myImage, _, err := image.Decode(imagePath)

  imagePath2, err := os.Open(path2)
  defer imagePath2.Close()
  myImage2, _, err := image.Decode(imagePath2)

  bounds := myImage2.Bounds()
  level := 0
  union := image.NewAlpha(bounds)

  for i := 0; i < 3; i++ {
    level += 25
    for j := 0; j < bounds.Dx(); j++ {
      for k := 0; k < bounds.Dy(); k++ {
        union.SetAlpha(j, k, color.Alpha{uint8(255 * level / 100)})
      }
    }
    myImage3 := image.NewRGBA(bounds)
    draw.Draw(myImage3, myImage3.Bounds(), myImage, image.ZP, draw.
    Src)
    draw.DrawMask(myImage3, bounds, myImage2, image.ZP, union,
    image.ZP, draw.Over)

    result, _ := os.Create("resultSequential2_" + strconv.Itoa(
    level) + ".jpg")
    defer result.Close()
    err = jpeg.Encode(result, myImage3, nil)
    check(err)
  }
}

func check(err error) {
  if err != nil {
    panic(err)
  }
}
```

Figure 6: Resultados.

## Solución Paralela

```
package main

import (
```

```
  "fmt"
  "image"
  "image/color"
  "image/draw"
  "image/jpeg"
  "strconv"
  "sync"
  "time"

  "os"
)

func main() {
  path := "3.jpg"
  path2 := "4.jpg"

  start := time.Now()
  Parallel(path, path2)
  elapsed := time.Since(start)
  fmt.Println("Parallel time: ", elapsed)
}

func Parallel(path string, path2 string) {
  imagePath, err := os.Open(path)
  defer imagePath.Close()
  myImage, _, err := image.Decode(imagePath)

  imagePath2, err := os.Open(path2)
  defer imagePath2.Close()
  myImage2, _, err := image.Decode(imagePath2)

  level := 0

  wg := new(sync.WaitGroup)

  goroutines := 3

  for i := 0; i < goroutines; i++ {
    wg.Add(1)
    level += 25
    go func(mylevel int) {
      bounds := myImage2.Bounds()
      union := image.NewAlpha(bounds)

      waitGroup := new(sync.WaitGroup)

      for h := 0; h < 2; h++ {
        startX := (h) * bounds.Dx() / 2
        endX := (h + 1) * bounds.Dx() / 2
        waitGroup.Add(1)
        go func() {
          for j := startX; j < endX; j++ {
            for k := 0; k < bounds.Dy(); k++ {
              union.SetAlpha(j, k, color.Alpha{uint8(255 * mylevel
    / 100)})
            }
          }
          defer waitGroup.Done()
```

```
61          }()
62        }
63        waitGroup.Wait()
64
65        myImage3 := image.NewRGBA(bounds)
66        draw.Draw(myImage3, myImage3.Bounds(), myImage, image.ZP,
      draw.Src)
67        draw.DrawMask(myImage3, bounds, myImage2, image.ZP, union,
      image.ZP, draw.Over)
68
69        result, _ := os.Create("resultParallel2_" + strconv.Itoa(
      mylevel) + ".jpg")
70        defer result.Close()
71        err = jpeg.Encode(result, myImage3, nil)
72        check(err)
73        defer wg.Done()
74      }(level)
75    }
76    wg.Wait()
77  }
78
79  func check(err error) {
80    if err != nil {
81      panic(err)
82    }
83  }
```

Figure 7: Imágenes de entrada.

Figure 8: Resultados.

### Análisis

| Implementación | Tiempo |
|---|---|
| Secuencial | 45.6795ms |
| Paralelo | 18.5831ms |

Table 2 : Tiempos de ejecución

A la imagen de máscara se le necesita manipular los valores Alpha, en este caso a menor valor tenga $X$, mayor transparencia.
A pesar de varias pruebas, nuevamente la implementación paralela es 2.5 veces mejor que la secuencial, se saca mayor provecho de los threads en los bucles, aunque sólo se debe limitar a un número pequeño (en este caso 3 y 2) dado la enorme tamaño de las imágenes y los 3 niveles de transparencia.

## Ejercicio 3

### Enunciado

Obtenga el histograma de una imagen, este representa la frecuencia de intensidad de un colores de los pixeles.Como un pixel tiene tres canales de colores, se puede obtener tres histogramas como se ve en la Figura 9.

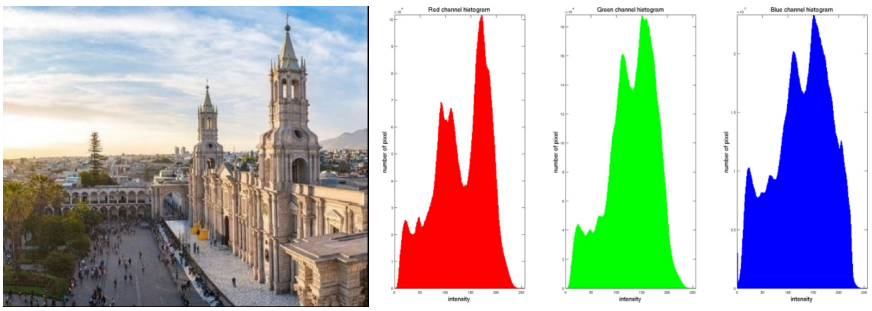


Figure 9: Ejemplo de histograma de una imagen.

### Solución Secuencial

```
package main

import (
  "fmt"
  "image"
  "image/color"
  "os"
  "time"

  "gonum.org/v1/plot"
  "gonum.org/v1/plot/plotter"
```

```
  "gonum.org/v1/plot/vg"
)

func main() {
  path := "8.jpg"

  start := time.Now()
  Sequential(path)
  elapsed := time.Since(start)
  fmt.Println("Sequential time: ", elapsed)
}

func Sequential(path string) {
  imagePath, err := os.Open(path)
  check(err)
  defer imagePath.Close()

  myImage, _, err := image.Decode(imagePath)
  check(err)

  size := myImage.Bounds().Size()

  var redValues plotter.Values
  var greenValues plotter.Values
  var blueValues plotter.Values

  for x := 0; x < size.X; x++ {
    for y := 0; y < size.Y; y++ {
      pixel := myImage.At(x, y)
      originalColor := color.RGBAModel.Convert(pixel).(color.RGBA)
      r := float64(originalColor.R)
      g := float64(originalColor.G)
      b := float64(originalColor.B)
      redValues = append(redValues, r)
      greenValues = append(greenValues, g)
      blueValues = append(blueValues, b)
    }
  }

  Histogram(redValues, "Red channel histogram (Sequential)", color.
    RGBA{R: 255, G: 0, B: 0, A: 255}, "redHistogramSequential.png")
  Histogram(greenValues, "Green channel histogram (Sequential)",
    color.RGBA{R: 0, G: 255, B: 0, A: 255}, "
    greenHistogramSequential.png")
  Histogram(blueValues, "Blue channel histogram (Sequential)",
    color.RGBA{R: 0, G: 0, B: 255, A: 255}, "
    blueHistogramSequential.png")
}

func Histogram(values plotter.Values, title string, color color.
    Color, name string) {
  myPlot := plot.New()
  myPlot.Title.Text = title
  myPlot.X.Max = 260
  myPlot.X.Label.Text = "intensity"
  myPlot.Y.Label.Text = "number of pixel"

  histogram, err := plotter.NewHist(values, 300)
```

```
  check(err)
  histogram.Color = color

  myPlot.Add(histogram)
  myPlot.Save(4*vg.Inch, 6*vg.Inch, name)
}

func check(err error) {
  if err != nil {
    panic(err)
  }
}
```

Figure 10: Imagen de entrada.

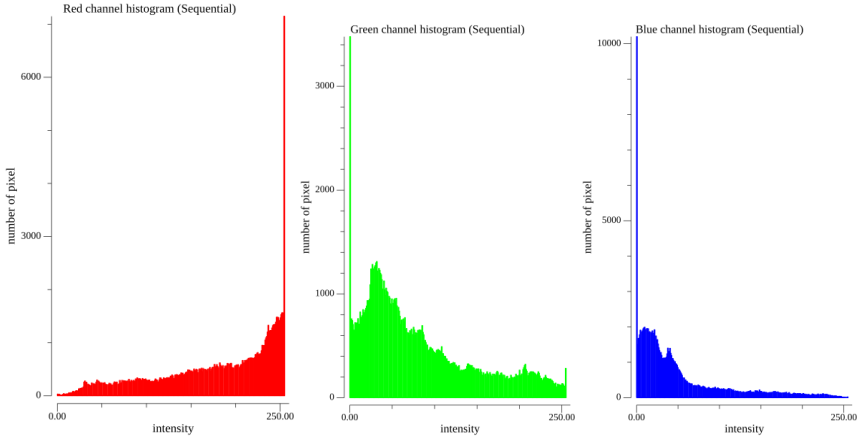


Figure 11: Histogramas resultantes.

## Solución Paralela

```
package main

import (
  "fmt"
```

```
  "image"
  "image/color"
  "image/draw"
  "image/jpeg"
  "math"
  "os"
  "sync"
  "time"
)

func main() {
  path := "1.jpg"
  path2 := "2.jpg"

  start := time.Now()
  Parallel(path, path2)
  elapsed := time.Since(start)
  fmt.Println("Parallel time: ", elapsed)
}

func Parallel(path string, path2 string) {
  imagePath, err := os.Open(path)
  defer imagePath.Close()
  myImage, _, err := image.Decode(imagePath)

  imagePath2, err := os.Open(path2)
  defer imagePath2.Close()
  myImage2, _, err := image.Decode(imagePath2)

  bounds := myImage2.Bounds()

  size := myImage2.Bounds().Size()
  rect := image.Rect(0, 0, size.X, size.Y)
  wImg := image.NewRGBA(rect)

  waitGroup := new(sync.WaitGroup)

  goroutines := 7

  for i := 0; i < goroutines; i++ {
    startX := (i) * size.X / goroutines
    endX := (i + 1) * size.X / goroutines
    waitGroup.Add(1)
    go func() {
      for x := startX; x < endX; x++ {
        for y := 0; y < size.Y; y++ {
          pixel := myImage2.At(x, y)
          originalColor := color.RGBAModel.Convert(pixel).(color.
    RGBA)

          r := uint8(math.Min(255, float64(originalColor.R)*3.5))
          g := uint8(math.Min(255, float64(originalColor.G)*3.5))
          b := uint8(math.Min(255, float64(originalColor.B)*3.5))

          c := color.RGBA{
            R: r, G: g, B: b, A: originalColor.A,
          }
          wImg.Set(x, y, c)
```

```
62          }
63        }
64        defer waitGroup.Done()
65      }()
66    }
67    waitGroup.Wait()
68
69    union := image.NewAlpha(bounds)
70
71    for i := 0; i < goroutines; i++ {
72      endX := (i + 1) * bounds.Dx() / goroutines
73      startX := (i) * bounds.Dx() / goroutines
74      waitGroup.Add(1)
75      go func() {
76        for x := startX; x < endX; x++ {
77          for y := 0; y < bounds.Dy(); y++ {
78            union.SetAlpha(x, y, color.Alpha{uint8(75)})
79          }
80        }
81        defer waitGroup.Done()
82      }()
83    }
84    waitGroup.Wait()
85
86    myImage3 := image.NewRGBA(bounds)
87    draw.Draw(myImage3, myImage3.Bounds(), myImage, image.ZP, draw.
        Src)
88    draw.DrawMask(myImage3, bounds, wImg, image.ZP, union, image.ZP,
        draw.Over)
89
90    result, _ := os.Create("resultParallel1.jpg")
91    defer result.Close()
92    err = jpeg.Encode(result, myImage3, nil)
93    check(err)
94  }
95
96  func check(err error) {
97    if err != nil {
98      panic(err)
99    }
100 }
```

Figure 12: Imagen de entrada.

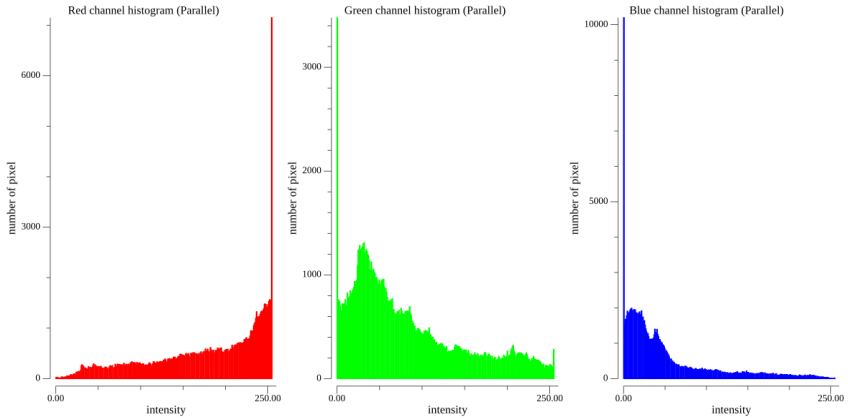


Figure 13: Imágenes de entrada y resultado.

### Análisis

| Implementación | Tiempo |
|---|---|
| Secuencial | 68.0478ms |
| Paralelo | 50.7469ms |

Table 3 : Tiempos de ejecución

Para la obtención de los valores RGB se usa la misma lógica. Para graficar los histogramas se necesita el módulo ***"gonum.org/v1/plot"***, este también usa otros como : ***"gonum.org/v1/plot/plotter"*** y ***"gonum.org/v1/plot/vg"***. Se debe crear un función especial (Histogram).
A pesar de varias pruebas, la implementación paralela es mejor que la secuencial, aunque esta vez no por tanto. En este caso no se puedo sacar mucho provecho a los threads dado errores al exportar o generar los histogramas, sólo se pudo usar uno para obtener uno de los histogramas.

## Conclusión

Go demostró ser tan bueno como python para manipular imágenes y generar gráficos, aunque quizás no tiene tanta facilidad como python. Para el manejo de threads se muestra simple aunque si se debe prestar atención a como se programa estos.

GitHub : `https://github.com/KEPCU/FundamentalsOfProgrammingLanguages/tree/master/go/practice17`