Práctica 8

Modelo de urnas

Introducción

En esta práctica se simula un sistema de coagulación y fragmentación de partículas que se puede aplicar en un sistema de filtrado. La práctica consiste en tener una cantidad de partículas disponibles (n) con la cual se podrán formar una cantidad determinada de cúmulos (k).

Objetivos

1. Analizar el código original y determinar las partes que se pueden paralelizar para ahorrar el mayor tiempo posible.
2. Determinar si el ahorro en el tiempo de ejecución de la paralelización es estadísticamente significativo y determinar a partir de que valores k se observa este cambio.

Simulación y Resultados

Para la tarea base se analizaron las partes del código que era necesario realizar muchas veces y que eran independientes entre sí. Las partes que se paralelizaron en este código fueron las etapas de romper, unir y juntarse. Estas fases necesitan realizarse muchas veces dependiendo de la cantidad de cúmulos *k*, se agregaron 3 funciones una para cada una de las etapas que se deben realizar, donde primero se rompen todos los cúmulos con la función *rompiendo ()*, luego se unen todos los cúmulos con las funciones *uniendo ()* y *uniendose ()*, en el código se agregaron tres *foreach* para cada una de las funciones, las partes del código modificado se pueden apreciar a continuación:

```
#Función Rompiendo
rompiendo <- function(){
  cumulos <- integer()
  urna <- freq[i,]
  if (urna$tam > 1) { # no tiene caso romper si no se puede
    cumulos <- c(cumulos, romperse(urna$tam, urna$num))
  } else {
    cumulos <- c(cumulos, rep(1, urna$num))
  }
  return(cumulos)
}
```

```r
#Función Uniendo
uniendo <- function(){
  cumulos <- integer()
  urna <- freq[i,]
  cumulos <- c(cumulos, unirse(urna$tam, urna$num))
  return(cumulos)
}
#Función juntar
uniendose<- function(){
  tcumulos <- juntarse[2*i-1] + juntarse[2*i]
  return(tcumulos)
}
```

Además de las funciones, en la parte de las afirmaciones de la función *uniendo* se modificaron nombres de variables para no tener que leer y reescribir la variable *cumulos* constantemente y así ahorrar algo más de tiempo de ejecución total. También para evitar un error que salía durante la simulación donde la suma de los *cumulos* no era igual a n cuando la variable *cu* daba un número impar. Por esta razón se cambió el nombre a los cúmulos mayores a 0 por la variable *positivos* y el *foreach* que devuelve los pares se le puso *cu*. Al final se hace una suma de estas dos variables y se integran a la variable *cumulos*, el código se modificó como se muestra a continuación:

```r
#Uniendo
  cumulos=foreach(i=1:dim(freq)[1], .combine = c) %dopar% uniendo ()
  assert(sum(abs(cumulos)) == n)
  assert(length(cumulos[cumulos == 0]) == 0) # que no haya vacios
  juntarse <- -cumulos[cumulos < 0]
  positivos <- cumulos[cumulos > 0]#
  assert(sum(positivos) + sum(juntarse) == n)#
  nt <- length(juntarse)
  if (nt > 0) {
    if (nt > 1) {
      juntarse <- sample(juntarse)
       cu=foreach(i=1:floor(nt / 2),.combine = c)%dopar% uniendose()
        cumulos<-c(positivos,cu)
    }
    if (nt %% 2 == 1) {
```

```
      cumulos <- c(cu,positivos, juntarse[nt])
    } }
```

Posteriormente, se realizó un código nuevo en donde por medio de la función *source* se mandó a llamar ambos programas y con ayuda de dos *for*, uno para controlar el valor de *k* otro para controlar las réplicas se simuló el programa, el código se muestra a continuación:

```
x=data.frame()
Resultados=data.frame()
for (i in 1:5){
  source('~/GitHub/SimulacionComputacional/P8/p8original/p8original.R', encoding = 'UTF-
8')
  source('~/GitHub/SimulacionComputacional/P8/p8para/parap8.R')
  Resultados=cbind(TiempoO,TiempoT)
  x=rbind(x,Resultados)
}
colnames(x)=c("Programa Original", "Programa Paralelizado")
png("Prac8t1.png",width=600, height=800,pointsize = 20)
boxplot(x,col=c("Blue","Red"),ylab="Tiempo (min)")
graphics.off()
```

Los resultados obtenidos se pueden observar en la figura 1 donde cabe mencionar que la k se modificó a un valor de 100,000 ya que para un valor de 10,000 el programa secuencial era más eficiente que el programa paralelizado. En la figura 1 se muestran los programas comparados para una k de 100,000 y cinco réplicas para cada uno de los programas. Se puede observar que el programa paralelizado tiene un tiempo de ejecución menor.

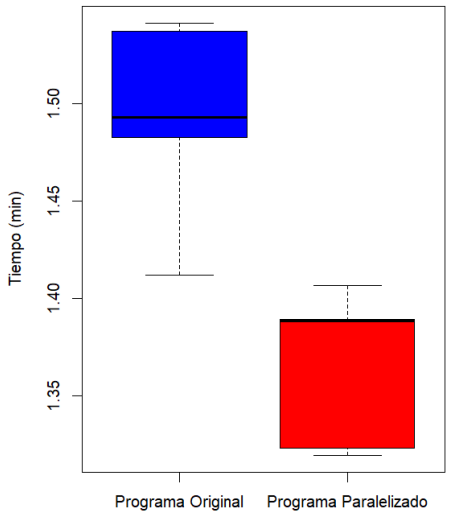
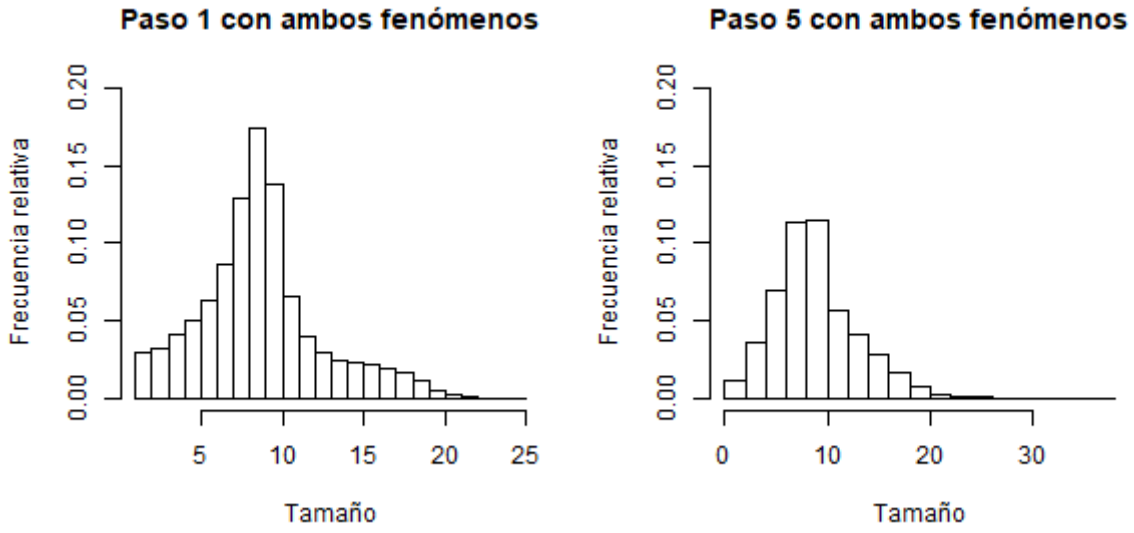
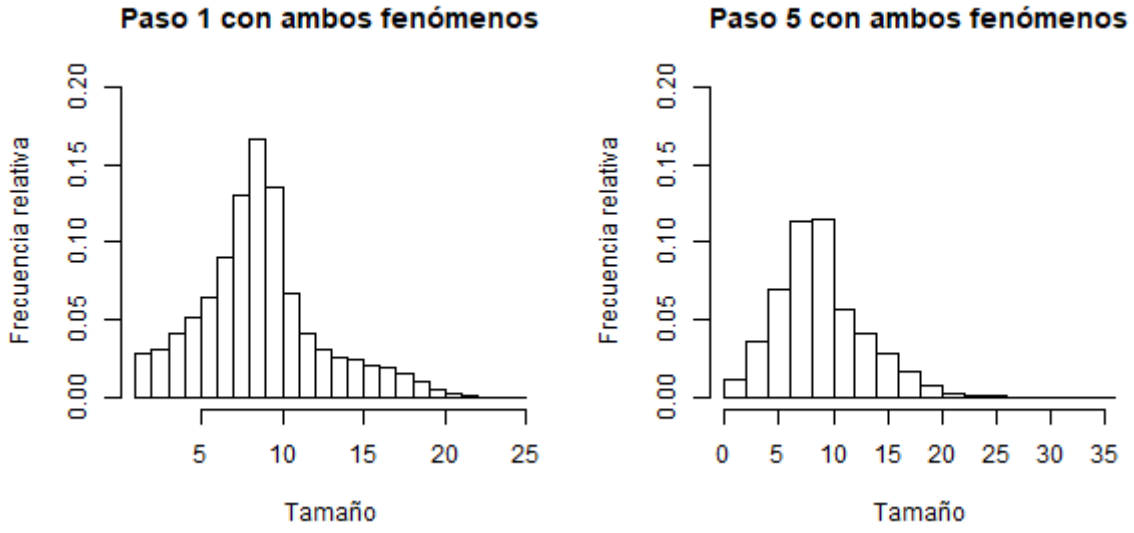


Figura 1. Gráfico de Caja-bigote tiempos de ejecución para el programa original.

En la figura 2 se pueden observar el primer y último gráfico de distribución para ambos programas mostrando a la izquierda el paso uno y a la derecha el último paso. Se pueden observar que las distribuciones son similares e indica que la paralelización no altero los resultados del programa original.



a)



b)

Figura 2. Gráfico comparativo de distribuciones

a) Programa original, b) Programa paralelizado.

Reto 1

Para el primer reto se corrió la simulación con distintos valores de *k*, para analizar a partir de que valores de *k* el ahorro del tiempo era estadísticamente significativo. Para esta simulación se mantuvo el valor de n = 30k, se realizó un código nuevo en donde por medio de la función *source* se mandó a llamar ambos programas y con ayuda de dos *for*, uno para controlar el valor de *k* (80000, 100000, 150000, 200000) y otro para controlar las réplicas en este caso fueron cinco para cada valor de *k* como se muestra a continuación:

```
for (k in c(80000,100000,150000,200000)){
  for (r in 1:5){
    source('~/GitHub/SimulacionComputacional/P8/p8original/p8original.R', encoding =
'UTF-8')
    Toriginal=cbind(k,"o",Tiempo)
    source('~/GitHub/SimulacionComputacional/P8/p8para/parap8.R')
    Tparalelo=cbind(k,"p",Tiempo)
    Resultados=rbind(Resultados,Toriginal,Tparalelo)
  }
```

Además se realizó una prueba estadística para determinar si la diferencia entre el tiempo de ejecución entre ambos programas era significativa. Esta prueba se realizó con t student con la función *test* en R, se filtraron los valores del data frame *Resultados* para cada valor de k y para cada programa, *o* para el original y *p* para el paralelizado con el código que se muestra a continuación:

```
PruebaTO<-Resultados[Resultados$k == k & Resultados$tipo=="p",]
PruebaTP<-Resultados[Resultados$k == k & Resultados$tipo=="o",]
vecO<-PruebaTP$Tiempo
vecP<-PruebaTO$Tiempo
student<-t.test(vecO,vecP)
print(student)
```

En la figura 3 se muestra visualmente como la brecha de los tiempos entre ambos programas aumenta conforme aumentamos la cantidad de cúmulos *k.* Esto quiere decir que entre mayor sea la cantidad de cúmulos el programa paralelizado brilla más. Así mismo, se verificó que por debajo de un valor de k = 80,000, el programa secuencial es más eficiente que el paralelizado.

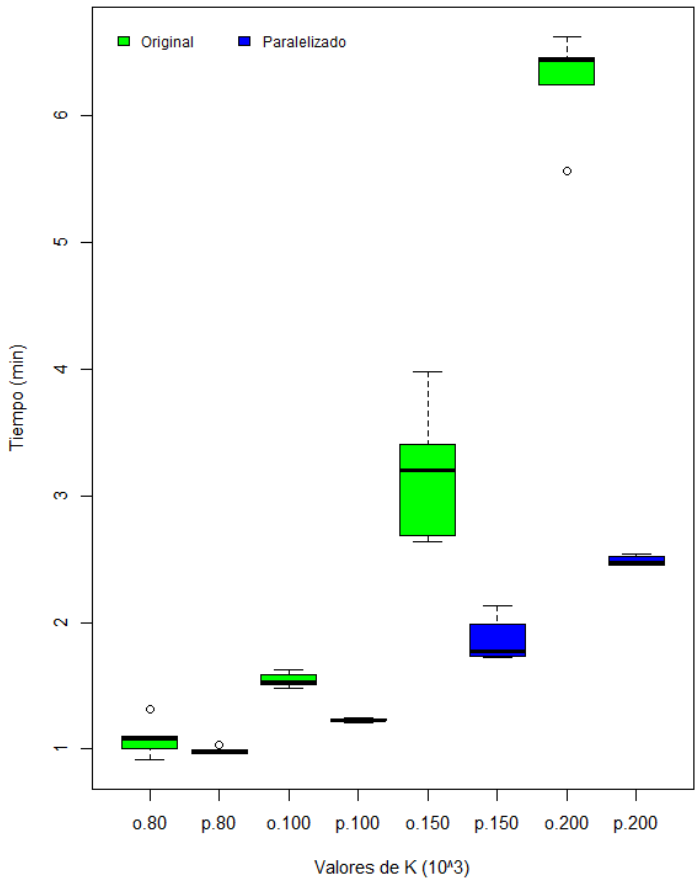


Figura 3. Gráfico comparativo de los tiempos de ejecución para diferentes valores de k

Se realizó un análisis estadístico calculando p-value para cada una de las *k* simuladas, dónde para cada una la hipótesis alternativa establece que el aumento en los valores de *k* dará como resultado que la diferencia del tiempo de ejecución entre el programa paralelizado y el original es estadísticamente significativa. Los resultados se muestran en el cuadro 1.

Cuadro 1. Análisis estadístico comparativo entre los valores de *k* por *t* student.

Valor de k	p-value
80,000	0.2339
100,000	1.68×10^{-4}
150,000	4.23×10^{-3}
200,000	3.043×10^{-5}

Se puede observar que en el análisis estadístico se corrobora lo que se aprecia en la figura 3, para el primer caso de k = 80,000 el valor de la p-value es mayor al 5% de significancia por lo tanto, la diferencia del tiempo de ejecución entre el programa paralelizado y original no es significativa. Sin embargo para los demás casos al ser un valor por debajo del 5% de significancia, la hipótesis alternativa es aceptada, es decir, la diferencia del tiempo de ejecución entre el programa paralelizado y el original es estadísticamente significativa.