Práctica No 3: Teoría de colas

Héctor Hernán Montes García

29 de agosto de 2017

Resumen

En la presente práctica se se estudian los tiempos de ejecución de un Código en R consistente en un método para descubrir qué números dentro de una secuencia de números enteros son primos. Como el problema no es puramente secuencial, los tiempos se estudian para diferentes formas de paralelización del código haciendo uso de la librería de R denominada *Do-Parallel*. La tarea de evaluar la secuencia completa puede ser considerada como la realización paralela de un conjunto de sub-tareas de dificultad variable (cada número a evaluar es una subtarea), así que el tiempo total de ejecución se mide bajo diversos escenarios de cantidad de núcleos asignados y de ordenamiento de sub-tareas. Finalmente se aplican pruebas estadísticas formales para estudiar la existencia de diferencias significativas en los tiempos totales de ejecución para cada escenario, y se concluye que el orden de asignación de las sub-tareas, y la cantidad de núcleos asignado a la realización de la tarea completa impacta significativamente el tiempo total de ejecución.

1. Planteamiento del problema

En esta práctica estudiamos el problema de identificar qué números son primos dentro de una secuencia de número enteros. Para decidir si un número n particular es primo, una alternativa es revisar si él es divisible entre algún entero menor que él. En general es posible acotar el grupo de enteros candidatos (por ejemplo enteros no superiores a $\sqrt{n}$).La tarea de encontrar todos los números que son primos dentro de una secuencia de enteros especificada pasa entonces por aplicar esta sencilla regla sobre cada entero n de la secuencia. Si ningún entero actúa como divisor exacto el número se declara primo. Por supuesto, para evaluar si un número de la secuencia es primo (digamos n_1), no se requiere esperar los resultados de la evaluación sobre otro (digamos n_2), así que las evaluaciones se pueden hacer en paralelo, y podemos disponer varios núcleos de la computadora para estudiar varios números en simultáneo.

El problema anterior tiene todas las características de un problema de teoría de colas, donde cada núcleo actúa como un servidor (prestador del servicio de identificar si un n dado es primo), la longitud completa de la cola en el tiempo t_0 es la longitud de la secuencia de números a etiquetar como primos o no. Aunque pudiera ser de interés evaluar cuánto tarda la tarea en ser completada conforme más números se agregan a la cola de trabajo a una tasa de llegada dada, y considerando la tasa de servicio de los núcleos (problema típico en teoría de colas), por lo pronto trataremos un problema más bien sencillo: consideraremos

que no existen llegadas nuevas, y que nos interesa evaluar cuánto tarda el servicio completo en ser prestado, es decir, tiempo transcurrido entre la evaluación del primer número, y la evaluación del último medido a través de una instrucción que registre el momento en que la función de etiquetado ha finalizado con todos los números. Esto dependerá, en general, del orden de llegada de los casos, y la forma cómo estos son enviados para ser atendidos por cada servidor, puesto que cada núcleo puede recibir una carga de trabajo diferente y no tienen porque finalizar en tiempos parejos.

En la presente situación práctica, nosotros asumimos no saber cómo opera internamente la librería de R *DoParallel* en lo que atañe a la distribución del esfuerzo de cómputo entre los núcleos (aunque siempre sea posible consultarlo por revisión cuidadosa del código original). La implementación actual de la instrucción de etiquetado es como sigue:

```
primo <- function(n)
{
  #Evaluamos si el entero es el primo 1 o 2 (casos triviales)
  if (n == 1 || n == 2)
  {
    return(TRUE)
  }
  #Evaluamos si es par, si lo es, etiquetamos falso (no primo)
  if (n %%2 == 0)
  {
    return(FALSE)
  }
  #Si es impar mayor a 3 y divisible entre un entero candidato,
      ponemos falso
  for (i in seq(3, max(3, ceiling(sqrt(n))), 2))
  {
    if ((i<n)&&(n %%i) == 0)
    {
      return(FALSE)
    }
  }
  return(TRUE)
}
```

Una mirada rápida al código nos hace conscientes de que el tiempo que tarda un núcleo en evaluar un número es variable, pues depende de factores como: pertenecer a un grupo trivial (por ejemplo: 1,2), ser éste un número par, o ser impar. Si es impar, la función de etiquetado evalúa los enteros candidatos a divisores, y el tiempo total de etiquetado dependerá del tamaño y naturaleza del número (números primos grandes son más tardados de evaluar).

De acuerdo con lo anterior, nuestro objetivo será, no sólo estudiar los tiempos de etiquetado, sino también realizar conjeturas sobre el mecanismo usado por la librería *DoParallel* para distribuir entre los núcleos la tarea de etiquetar toda un secuencia completa de números.Para ello convendrá tomar control de las secuencias de números que se le pasarán al programa, en forma tal que se conozca previamente la dificultad de evaluación de los números suministrados y el orden en que se están pasando al comando *foreach*. Cada número puede interpretarse como una sub-tarea de complejidad variable, y de lo que se trata es de evaluar el impacto que diferentes ordenamientos y diferentes cantidades de núcleos tienen sobre los tiempos totales de ejecución, así como explicar las

razones de estas diferencias y establecer por métodos estadísticos formales si éstas son significativas. Los objetivos precisos se formulan en la siguiente sección.

2. Objetivos

1. Examinar cómo cambian las diferencias en los tiempos de ejecución de cinco diferentes ordenamientos (ascendente, descendente,aleatorio, dificultad creciente, dificultad decreciente) cuando se varía el número de núcleos asignados a la tarea.

2. Argumentar, apoyándose con visualizaciones, las posibles causas de las diferencias en los tiempos de ejecución y razonar cómo y por qué le afecta el número de núcleos disponibles a la diferencia.

3. Aplicar en R pruebas estadísticas para determinar si las diferencias observadas entre los cinco ordenamientos son significativas.

3. Metodología

Para atacar cada objetivo se usan una serie de herramientas, basadas esencialmente en estrategias que controlan el orden en que las secuencias de números que se van a evaluar es pasada al algoritmo, usando consideraciones de tamaño del número, y la naturaleza del mismo (si es par, impar, o impar primo, por ejemplo). También se tomó en consideración la cantidad de núcleos físicos que tenía disponible la computadora usada en la experimentación[1], para medir el tiempo de ejecución total cuando se usan dos núcleos y compararlo contra el tiempo total obtenido cuando se usan tres. A continuación describimos como se procedió en cada objetivo

3.1. Discusión Resultados Objetivo 1

En el caso del primer objetivo, construimos un vector de enteros desde el número 10000 hasta el 30000, y planteamos cinco formas distintas de ordenar sus elementos: creciente, decreciente, aleatorio, en orden creciente de dificultad de etiquetado, y por último en orden decreciente de dificultad de etiquetado.

Para construir el orden de dificultad creciente se partió de la secuencia de números en su orden natural (orden creciente de magnitud), y se rastrearon los pares en su orden de aparición. Estos se pusieron en las primeras posiciones. Luego se rastrearon los impares no primos y se pusieron en las siguientes posiciones nuevamente en el orden de aparición. Por último, se ubicaron los primos en orden de aparición. El orden de dificultad decreciente simplemente fue construido mediante una inversión del vector anterior. Todos estos procedimientos fueron automatizados como se describe a continuación.

En primer lugar, se construyó una rutina en R que informara sobre el tipo de número con el que estábamos tratando. Para el efecto la rutina debía etiquetar

[1]Se usó una computadora con sistema operativo Windows 10 Pro de 64bits y procesadorx64 Intel(R) core(TM) i5-4210U, CPU@ 1.70GHz 2.40GHz y memoria instalada de 6.00GB. Una llamada a la función *detectcores(logical=FALSE)* de R, permitió además constatar que la computadora posee dos núcleos físicos disponibles

los números que eran pares, aquellos que eran impares pero no primos y por supuesto los números primos.

El código resulta de ligeras modificaciones al código original que se ofreció en clase para medir tiempos de ejecución del etiquetado de números. En lugar de simplemente etiquetar el número como primo o no primo mediante los valores de retorno *false* o *true*, se cambió el código para que retornara los textos *Trivial*, *Par*, *Impar no primo* e *Impar primo*. La etiqueta *Trivial* se refiere a si el número es 1,o 2 pero para la secuencia de números que elegimos esto no sucede, así que se dejó dentro del código simplemente con fines de generalización.

El código anterior se corrió en forma paralelizada para acelerar un poco el proceso, aunque este sólo se fuese a ejecutar una única vez, y se usó para el efecto la totalidad de núcleos físicos de la máquina. La paralelización se hizo para almacenar, ya no los tiempos de ejecución del etiquetado, sino las etiquetas mismas, a través de una sentencia *foreach* que recorriendo todos los números de la secuencia arroja como salida el vector de etiquetas. Finalmente un llamado a la función *table* sobre el vector de etiquetas creado, permitió obtener toda la información necesaria para construir la Tabla 1 que se presenta más abajo. El código usado para el etiquetado del tipo de número es el siguiente:

```
suppressMessages(library(doParallel))
nucleos<-(detectCores(logical=FALSE)-1)#logical=False nos
    garantiza que estamos usando los cores fisicos
etiquetas <- function(n)
{
  if (n == 1 || n == 2)
  {
    return("Trivial")
  }
  if (n %%2 == 0)
  {
    return("Par")
  }
  for (i in seq(3, max(3, ceiling(sqrt(n))), 2))
  {
    if ((i<n)&&(n %%i) == 0)
    {
      return("Impar no primo")
    }
  }
  return("Impar primo")
}
desde <- 10000
hasta <-  30000
original <- desde:hasta
registerDoParallel(makeCluster(nucleos))
ot <- foreach(n = original, .combine=c) %dopar% etiquetas(n)
stopImplicitCluster()
table(ot)
```

Así mismo, el procesamiento de la información contenida en una base de datos disponible en internet[2] que provee los primeros 50 millones de números primos, pero de la que nosotros sólo extrajimos los que caían en el intervalo bajo consideración, nos permitió confirmar la consistencia de los resultados presentados en la Tabla 1.

[2]https://primes.utm.edu/lists/small/millions/

Tabla 1: Conteo de números según tipo y tamaño

Tipo de número	Tamaño				Totales
	[10000-15000]	(15000-20000]	(20000-25000]	(25000-30000]	
Impar	2500	2500	2500	2500	10000
No primo	1992	2000	2017	1975	7984
Primo	508	500	483	525	2016
Par	2500	2500	2500	2501	10001
Totales	5000	5000	5000	5001	20001

Por último, para construir los vectores que en consonancia con el etiquetado anterior, almacenan los enteros de acuerdo con su dificultad de etiquetado, bien sea en orden creciente o decreciente, se usó el código que sigue:

```
1  dif_crec_t<-numeric()
2  dif_dec_t<-numeric()
3  tipos<-c("Trivial","Par","Impar no primo","Impar primo")
4  k<-1
5  for (i in tipos)
6  {
7    for (j in desde:hasta)
8    {
9      if (o_num[j-desde+1]==i)
10     {
11       dif_crec_t[k]<-j
12       k<-k+1
13     }
14   }
15 }
16 dif_dec_t<-rev(dif_crec_t)
```

En el código anterior, el ciclo *for* entre las líneas *5* y *15*, compara cada componente del vector contra los tipos almacenados en el vector *tipos*, en forma tal que resulten primero ubicado las componentes con etiqueta *Trivial*, luego aquellas con etiqueta *Par*, y así sucesivamente en el orden creciente de dificultad definido por *tipos*. La última instrucción simplemente invierte el vector de dificultad creciente para construir el de dificultad decreciente.

Por último, estudiamos para estos cinco vectores de enteros, escritos bajo ordenamientos diferentes, los tiempos totales de clasificación de nuestra rutina de detección de primos. Es decir, nos propusimos cumplir con el objetivo 1 de examinar cómo cambian las diferencias en los tiempos de ejecución de cinco diferentes ordenamientos (ascendente, descendente,aleatorio, dificultad creciente, dificultad decreciente) cuando se varía el número de núcleos asignados a la tarea.

Para lo anterior, propusimos nuevamente algunas modificaciones al código original entregado en clase, de suerte que recorriera un ciclo sobre los cores disponibles (lamentablemente sólo 2 o 3 para el equipo que se usa). Para garantizar consistencia de nuestros resultados, efectuamos varias réplicas (50 por cada combinación de núcleos y ordenamientos). Finalmente, una vez obtenidos los tiempos y almacenados en formato matricial conveniente para cada tipo de ordenamiento definido, ejecutamos resúmenes estadísticos para facilitar las comparaciones de los tiempos totales mínimo, máximo, y promedio obtenidos a lo largo de las 50 réplicas del experimento[3]. El código usado para estas comparaciones fue el siguiente:

[3]Por supuesto, somos conscientes de que más réplicas pueden haberse propuesto así cómo

```
nucleos<-seq(2:(detectCores(logical=FALSE)))
invertido <- hasta:desde
replicas <- 50
m_ot <-  matrix(nrow = length(nucleos),ncol=replicas)
m_it <-  matrix(nrow = length(nucleos),ncol=replicas)
m_at <-  matrix(nrow = length(nucleos),ncol=replicas)
m_dif_crec_t <-  matrix(nrow = length(nucleos),ncol=replicas)
m_dif_dec_t <-  matrix(nrow = length(nucleos),ncol=replicas)
names_var<-paste(nucleos," n\u{FA}cleos")
for (i in nucleos)
{
  registerDoParallel(makeCluster(i))
  ot<-numeric()
  it<-numeric()
  at<-numeric()
  df_c_t<-numeric()
  df_d_t<-numeric()
  for (r in 1:replicas)
  {
    ot<- c(ot, system.time(foreach(n = original, .combine=c) %
        dopar% primo(n))[3]) # de menor a mayor
    it <- c(it, system.time(foreach(n = invertido, .combine=c) %
        dopar% primo(n))[3]) # de mayor a menor
    at <- c(at, system.time(foreach(n = sample(original), .
        combine=c) %dopar% primo(n))[3]) # orden aleatorio
    df_c_t <- c(df_c_t, system.time(foreach(n = dif_crec_t, .
        combine=c) %dopar% primo(n))[3]) # orden menor a mayor
        dificultad
    df_d_t <- c(df_d_t, system.time(foreach(n = dif_dec_t, .
        combine=c) %dopar% primo(n))[3]) # orden mayor a menor
        dificultad
  }
  m_ot[i,]<-ot #Resultados orden ascendente
  m_it[i,]<-it #Resultados orden descendente
  m_at[i,]<-at #Resultados orden aleatorio
  m_dif_crec_t[i,] <- df_c_t #Resultados orden dificultad
      creciente
  m_dif_dec_t[i,] <-  df_d_t #Resultados orden dificultad
      decreciente
  stopImplicitCluster()
}
#Imprimo los resultados en pantalla
print(paste("Resultados de tiempos de ejecuci\u{F3}n para ",i," n\u
    {FA}cleos"))
print(apply(m_ot,1,summary)) #El apply respecto a 1 porque las
    repeticiones van en filas
print(apply(m_it,1,summary))
print(apply(m_at,1,summary))
print(apply(m_dif_crec_t,1,summary))
print(apply(m_dif_dec_t,1,summary))
```

Los resultados obtenidos de la anterior implementación del código para los diferentes ordenamientos y cantidades de núcleos, se resumen en la Tabla 2.

Se observa que los ahorros en tiempos de ejecución cuando se pasa de 2 núcleos a 3, es de 1,6042 minutos (tomando promedios generales sobre todos los

un análisis más extenso en término de cantidad de núcleos y tamaño del vector de partida, pero por restricciones de Hardware del equipo disponible esto no fue posible. Para trabajos futuros queda la evaluación cuidadosa de las reducciones adicionales de tiempos que se pueden lograr en una implementación de nuestro código si se tiene libertad de aumentar aun más la cantidad de núcleos, y la forma como la asignación de ordenamientos puede llegar a afectarle

Tabla 2: Resultado de tiempos de ejecución para diferentes núcleos y ordenamientos

Orden	**Original**		**Invertido**		**Aleatorio**		**Dif Creciente**		**Dif Decreciente**	
Núcleos	**2**	**3**	**2**	**3**	**2**	**3**	**2**	**3**	**2**	**3**
Min.	9.250	7.650	9.270	7.64	9.230	7.610	9.120	7.610	9.200	7.610
1st Qu.	9.300	7.680	9.300	7.68	9.303	7.650	9.190	7.700	9.263	7.690
Median	9.330	7.700	9.320	7.69	9.320	7.670	9.225	7.720	9.295	7.710
Mean	9.323	7.708	9.324	7.70	9.327	7.676	9.234	7.717	9.322	7.708
3rd Qu.	9.340	7.720	9.340	7.71	9.347	7.690	9.268	7.747	9.377	7.720
Max.	9.530	7.850	9.460	7.92	9.530	7.940	9.420	7.810	9.550	7.890

ordenamientos), así que se puede afirmar que este es el efecto de pasar de usar dos núcleos a usar 3 independientemente del orden de trabajo que se elija.

Por otro lado, el ordenamiento que en promedio más mejora produce es el aleatorio, con una mejoría de 1,651 minutos, y el que más baja mejoría tiene es el de dificultad creciente el cual posee números primos altos al final de la cola y genera un ahorro de 1,517 minutos. Creemos que estos resultado puede estar explicado por el hecho de que, para el orden aleatorio, a lo largo de varias réplicas, las diferencias en balances de carga que fueron debidas al azar se compensan, y permiten que en general los núcleos trabajen casi la misma cantidad de tiempo, lo que repercute en una disminución estable de los tiempos de espera, o tiempos ociosos de núcleos cuando se están finalizando las últimas tareas, y a su vez en el ahorro de tiempo general de ejecución. Note que esta conjetura viene a ser reforzada por el hecho de que el ahorro más pobre se presente justo en aquel orden que es más proclive a generar desbalances al final del proceso, es decir, por el orden de dificultad creciente cuyas últimas tareas son números primos de tamaño grande que tienen más tiempo de etiquetado.

En la gráfica 1 y la gráfica 2, se da un apoyo para la visualización de resultados anteriores:

3.2. Discusión Resultados Objetivo 2

En el caso del segundo objetivo, interesa una aproximación vía simulación y análisis gráfico a la explicación de las diferencias detectadas en el primer objetivo, así que esto requerirá proceder en una forma un poco más sistemática en el diseño y asignación de secuencias, con el fin de conjeturar sobre la forma de asignación de tareas a la cantidad de núcleos disponibles y sus efectos en los tiempos de ejecución.

3.3. Discusión Resultados Objetivo 3

Finalmente el último objetivo añade formalismo a la explicación del segundo objetivo, así que será necesario el planteamiento de hipótesis sobre los tiempos de ejecución y su contraste a través de procedimientos con el debido rigor estadístico.

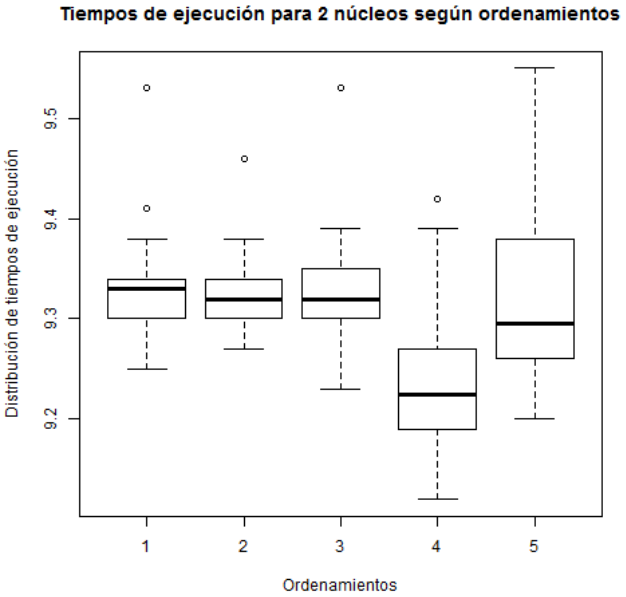

Figura 1:

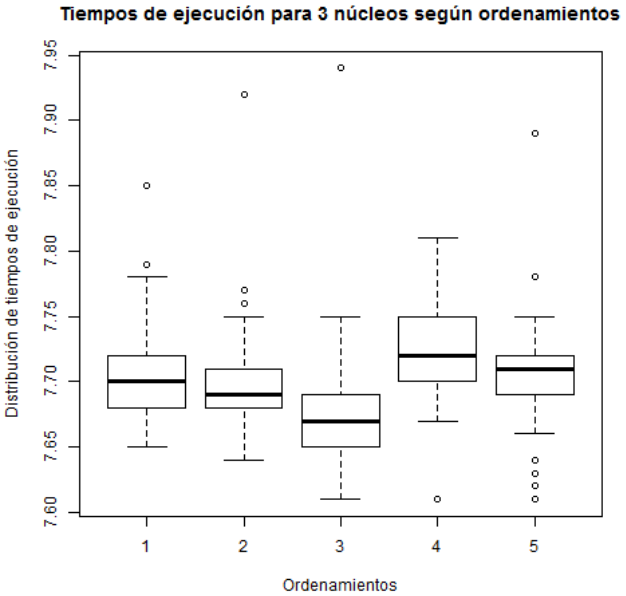

Figura 2: