Práctica 2

Guillermo Marquínez

En esta práctica tenemos que implementar el siguiente algoritmo, considerando la notación por columnas de las matrices, es decir, guardamos la matriz cuadrada A de dimensión l*l como un vector de l*l elementos donde el elemento (i,j) de la matriz A será el elemento i+l*j. Dadas tres matrices cuadradas A,B y C vamos actualizando C de la siguiente manera.

```
for i = 1 : l
        for i = 1 : l
                for k = 1 : l
                        C[i, j] = C[i,j] + A[i,k] · B[k,j]
                end
        end
end
```

Este programa tiene 6 maneras diferentes de ser implementado, pues podemos permutar los bucles for. Como ya pregunté en clase, este algoritmo no es lo mismo que decir que actualizamos la matriz C como C=C+A*B, pues en el algoritmo que hemos escrito, estamos actualizando el valor C(i,j) con cada iteración de k. Es decir, si queremos escribir el algoritmo que actualiza C=C+A*B deberíamos escribir el siguiente programa:

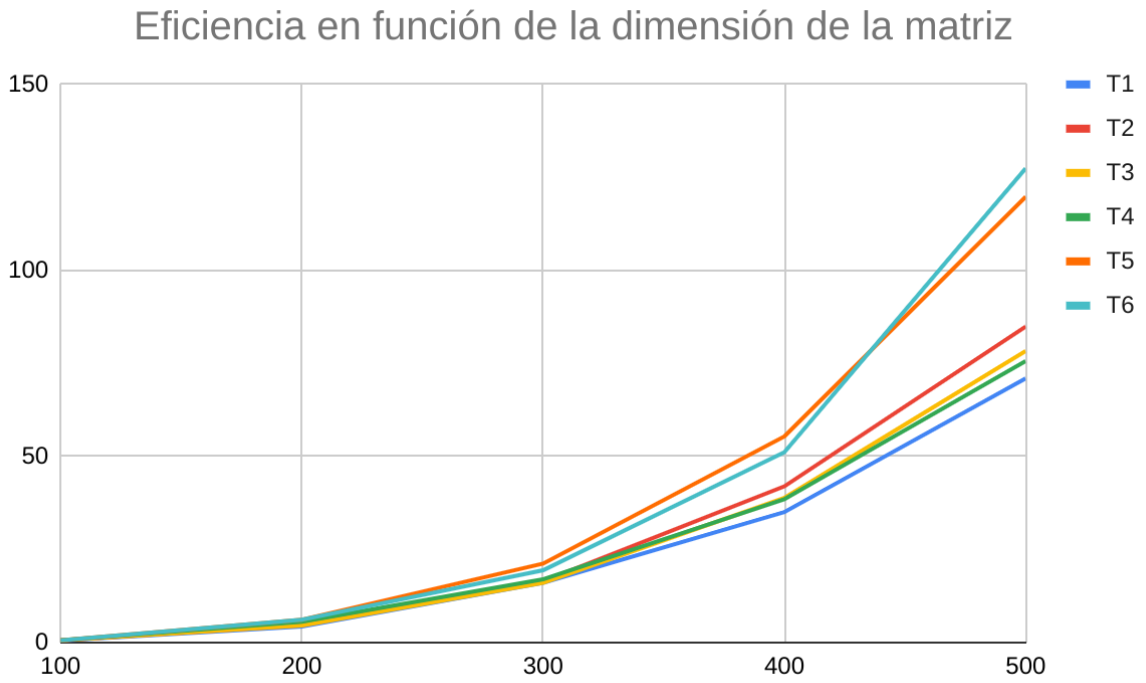
```
for i = 1 : l
        for i = 1 : l
                c=0
                for k = 1 : l
                        c= c + A[i,k] · B[k,j]
                C[i,j]=C[i,j]+c
                end
        end
end
```

En este programa primero hacemos la multiplicación de matrices A*B y después sumamos al resultado la matriz C. Aunque en clase dijimos que nos estábamos refiriendo a este segundo algoritmo, he decidido implementar el otro para que sea compatible con la práctica, pues como ahora hay valores que se actualizan fuera del bucle k, ya no tenemos 6 implementaciones posibles sino solamente 2 (los bucles i y j sí son permutables). Así que para que pueda comparar los tiempos en las 6 implementaciones diferentes, he decidido escoger el primer algoritmo.

Si ejecutamos el programa en el ordenador para matrices de 100*100, 200*200, 300*300, 400*400 y de 500*500 vemos que, en el siguiente gráfico, donde las líneas representan el tiempo que tarda cada uno de los programas según va aumentando la dimensión de las matrices cuadradas. Tal y como indica el gráfico, la línea azul es la más baja, y se corresponde con el siguiente programa:

```
def prod_1(A,B,C):
        l=int(len(A)**(1/2))
```

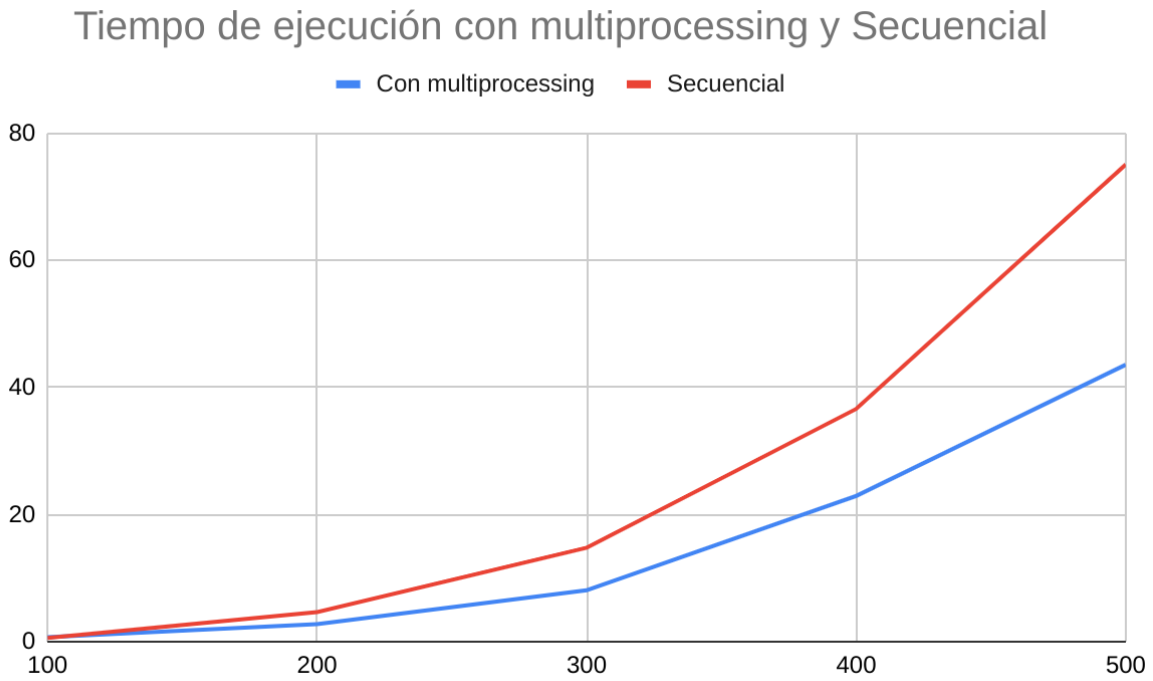
```
for i in range(0, l):
        for j in range (0,l):
                for k in range (0,l):
                        C[l*j+i]+= A[i+k*l]*B[j*l+k]
return(C)
```



Ejecutando el programa en Heracles para matrices 1000*1000, 2000*2000, 3000*3000 y 4000*4000 aparecen unos tiempos de ejecución muy bajos sin que ninguno sea mayor que otro. Además, para matrices de dimensión mayor da error por esperar más de 7 segundos.

Para la tercera pregunta de la práctica he implementado el algoritmo a bloques. He dividido la matriz en cuatro bloques diferentes. Para que sea más sencillo, las matrices cuadradas aleatorias creadas van a ser de dimensión l*l, donde l es par, así podemos dividir en cuatro submatrices de misma dimensión. Ahora, hay cuatro procesos totalmente independientes donde cada uno se encarga de dar el valor final del bloque que le corresponda. Después, hay que juntar todos esos resultados independientes y ver que el resultado final es el mismo que en el programa secuencial. Para recuperar el valor que devuelve cada proceso hay que crear una matriz de ceros en la memoria compartida, y que cada proceso edite su bloque correspondiente en esa matriz.

Una vez comprobado que el programa en paralelo está bien hecho y nos da el mismo resultado que en secuencial, vamos a comparar los tiempos. Como se ve en el programa, por cada bloque hay un proceso totalmente independiente. Como hemos dividido las matrices en 4 bloques, como máximo podremos paralelizar en cuatro procesos. Mi ordenador tiene 2 CPU así que no aprovechará al máximo la paralelización. Aún así, vamos a comparar tiempos.



A medida que aumenta la dimensión de las matrices, vemos que el tiempo con multiprocessing se aproxima a la mitad. Esto se debe a que mi ordenador tiene 2 CPUs y puede tener dos procesos en paralelo cada vez. En general no llega a ser nunca igual a la mitad, pues compartir memoria entre procesos lo hace un poco más lento.