Computo de Alto Rendimiento con lenguajes de alto nivel: Miniproyecto 1

Max B. Austria

March 19, 2022

Abstract

El presente documento evalua la concurrencia de un código python que el paquete "threading" para la multiplicación de matrices cuadradas de tamaño $n = 50, 100, 200, 300$ y 400.

Palabras clave: Computo concurrente, Python, threading, Multiplicación de matrices.

1 Introducción

La computación concurrente es una forma de cómputo en la cual varios cálculos se realizan simultaneamente, y no uno a la vez de forma secuencial.

El objetivo de este trabajo es evaluar el desempeño de un código python para la multiplicación de matrices a partir de la definición.

Dadas dos matrices $A_{m\times n}$ y $B_{n\times p}$, la operación a realizar será la siguiente:

$$(C)_{i,j} = \sum_{r-1}^{n} a_{i,r} b_{r,j}$$

2 Algoritmo Secuencial

El algoritmo secuencial que se programó es el más natural posible.

0 Inicializar una matriz C de tamaño apropiado con entradas nulas.

1 Para cada fila, $i = 1, \ldots, n$ hacer:

  2 Para cada columna $j = 1, \ldots, n$ hacer:

    3 Para $k == 1, \ldots, n$ hacer:

      4 $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$

3 Algoritmo Concurrente

El algoritmo concurrente inicializa cada hilo de manera idependiente a partir de los siguientes inputs:

- h un número natural que representa identifica al hilo, estos deben ser consecutivos y empezar desde el cero.
- N_h número de hilos.
- n Dimensión de las matrices cuadradas.

La idea principal consiste en asignar un conjunto de filas de la matriz A a cada hilo. Estos conjuntos no deben intersectarse y cubrir todos los renglones.

Para hacer la asignación, cada hilo calcula el tramo de las filas que le corresponde, de esta manera evitamos usar el "lock" de python.

El algoritmo queda como a continuación:

0. Inicializar una matriz C de tamaño apropiado con entradas nulas.
1. Para cada hilo, calcular su segmento como: $h[\frac{n}{N_h}], \ldots, (h+1)[\frac{n}{N_h}]$. En caso de que $h = N_h$ tomar hasta la fila n (es decir, todo el resto de las filas).
2. Realizar la multiplicación de las filas (de A) y columnas (de B) de manera secuencial.

4 Evaluación

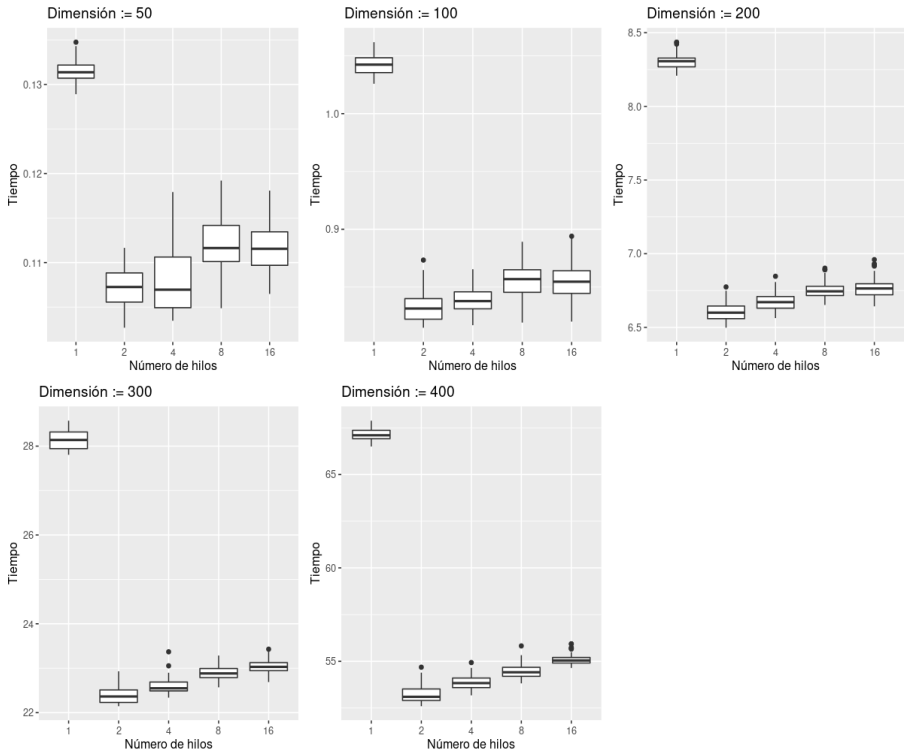
En terminos generales se puede decir que el tiempo de ejecución con el algoritmo concurrente fue menor (significativamente menor) que el secuencial. Pero aumentar el número de thread no necesariamente produce una ganancia.

En algunos casos la diferencia en el tiempo que producia aumentar el número de threads no era significativa (sobre todo para los tamaños de entrada más pequeños) pero se observa una clara tendencia a empeorar el desempeño conforme aumentan los threads (al menos para el algoritmo y código que aquí se presenta).

4.1 Tiempo de ejecución

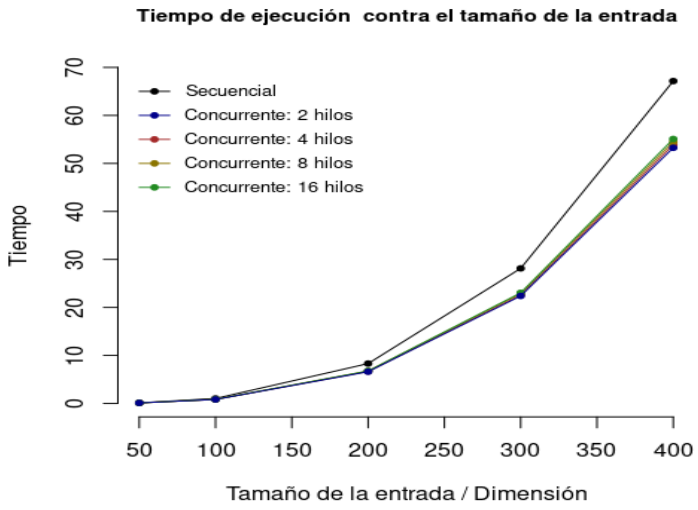
Los algoritmos se ejecutaron 100 veces cada uno para diferente tamaño de entrada y diferente número de hilos sobre una Intel(R) Xeon(R) CPU X5472 @ 3.00 GHz de 64 bits y 4 nucleos en la versión de 3.3 de Python.

El tiempo de ejecución del algoritmo concurrente siempre fue mejor al secuencial, independientemente del número de hilos que se usaran. En la siguiente imaágen se muestra lo anterior (el algoritmo secuencial se encuentra en la posición donde sólo se usa un thread). También se puede observar una tendencia "no decreciente" al aumentar el número de hilos.



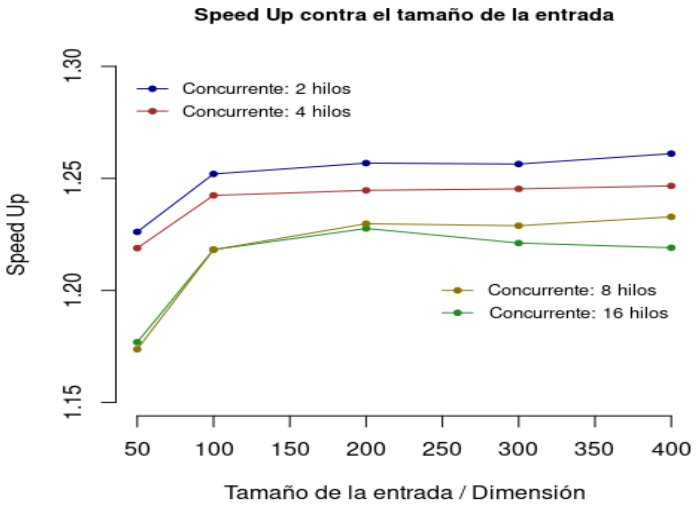
La gráfica anterior omite los intervalos de confianza pues se pierden por el rango (la varianza es muy pequeña).

Y se observo una tendencia creciente en el tiempo conforme crecia el tamaño de la entrada.



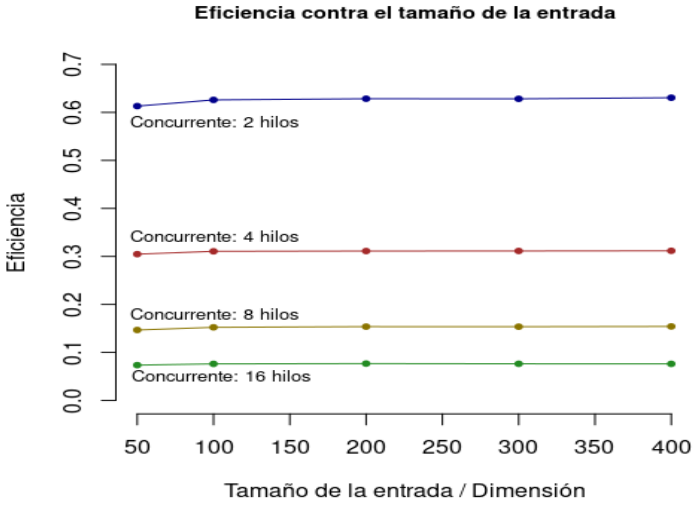
4.2 Speed Up

Respecto del Speed Up de las ejecuciones concurentes, se observo una ligera tendencia creciente para $2, 4$ y 8 hilos, mientras que 16 hilos decrece a partir de un tamaño de entrada de 200.



4.3 Eficiencia

Como se mencionó anteriormente, todas las ejecuciones concurrentes tienen tiempos menores que la secuencial. Pero en terminos de eficiencia, tener más de 2 hilos no representa una mejora. Probablemente se deba a las condiciones particulares de hardware, softwere ó a la implementación del algoritmo.



4.4 Escalabilidad

Incluso si aumentar el número de hilos no sea eficiente, la implementación concurrente permite incrementar los recursos para manterner un tiempo aceptable de ejecución.

5 Conclusiones

Después de evaluar el algoritmo de multiplicación concurrente y secuencial se puede concluir lo siguiente:

- La implementación concurrente puede permitir disminuir los tiempos de ejecución.
- Aumentar el número de hilos no garantiza minimizar el tiempo de ejecución.
- La programación concurrente representa un grado mayor de dificultad con respecto al paradigma secuencial.

6 Bibliografía

Lynch, N. A. (1996). Distributed algorithms. Elsevier.

Real Python. (2022, 8 marzo). An Intro to Threading in Python. Python Real. https://realpython.com/intro-to-python-threading/starting-a-thread

7 Anexo 1: Tabla de tiempos medios

Ejecución	$n = 50$	$n = 100$	$n = 200$	$n = 300$	$n = 400$
Secuencial	0.13142	1.04187	8.30558	28.13282	67.14461
Concurrente 2 hilos	0.10718	0.83217	6.60840	22.39172	53.24510
Concurrente 4 hilos	0.10782	0.83858	6.67302	22.59071	53.86004
Concurrente 8 hilos	0.11197	0.85526	6.75380	22.89327	54.46354
Concurrente 16 hilos	0.11167	0.85525	6.76547	23.03827	55.08006

Procesamiento secuencial

Se evaluara los tiempos del algoritmo secuencial.

La multiplicacion será de dos matrices cuadradas de tamaño n (llamadas A y B)

Cada evaluación se realizara con 100 simulaciones.

In [1]:
```
import numpy as np
import time

#Tamanio de la matriz
N = [50 100 200 300 400]
```

In [2]:
```
#Declarando la matriz TS
TS = np.zeros((100,len(N)))
```

Realizando la ejecución.

In [3]:
```
#s recorre el numero de simulaciones
for s in range(100):

    #n corre sobre los tamanios de la matriz
    for nt in range(len(N)):

        #Dimension de la matriz
        n = N[nt]

        print("Simulación: " + str(s) + " | Dim = " + str(n))

        #Declarando dos matrices aleatorias de tamanio n.
        A = np.random.rand(n,n)
        B = np.random.rand(n,n)

        #Declarando la matriz respuesta
        C = np.zeros((n,n))

        #Tomando el tiempo inicial
        t = time.time()

        #Realizando la multiplicacion

        #i corre sobre las filas de A
        for i in range(n):

            #j corre sobre las columnas de B
            for j in range(n):

                #k recorre la fila y la columna
                for k in range(n):

                    #Haciendo la multiplicacion y suma
                    C[i,j] = C[i,j] + A[i,k]*B[k,j]

        #Guardando el tiempo
        TS[s,nt] = time.time() - t
```

```
Simulación: 0 | Dim = 50
Simulación: 0 | Dim = 100
Simulación: 0 | Dim = 200
Simulación: 0 | Dim = 300
Simulación: 0 | Dim = 400
Simulación: 1 | Dim = 50
Simulación: 1 | Dim = 100
Simulación: 1 | Dim = 200
Simulación: 1 | Dim = 300
Simulación: 1 | Dim = 400
Simulación: 2 | Dim = 50
Simulación: 2 | Dim = 100
Simulación: 2 | Dim = 200
Simulación: 2 | Dim = 300
```

Exportando la matriz con los tiempos

In [4]:
```
np.savetxt("minip_sec.csv", TS, delimiter=",")
```

Procesamiento Concurrente

Se evaluara los tiempos del algoritmo concurrente. Cada evaluación se realizara con 100 simulaciones.

La multiplicacion será de dos matrices cuadradas de tamaño n (llamadas A y B).

El número de hilos seran $n_h := 2, 4, 8, 16$.

In [1]:
```
import threading
import time
import numpy as np

#Producto fila-columna
#h:= id del hilo
#n:= dimension de las matrices cuadradas
#Nh:= numero de hilos
def prodFC(h,Nh,n):

    #Las matrices son variables globales
    #global A,B
    #global C

    #numero de filas a procesar
    nf = int(n / Nh)

    #fila inicial
    fi = h*nf

    #fila final
    if h == Nh-1:
        ff = n
    else:
        ff = (h+1)*nf


    #i corre sobre las filas de A
    for i in range(fi,ff):

        #j corre sobre las columnas de B
        for j in range(n):

            #k recorre la fila y la columna
            for k in range(n):

                    #Haciendo la multiplicacion y suma
                    C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

Dimensiones de las matrices

In [2]:
```
#Tamanio de la matriz
N = [50,100,200,300,400]
```

Numero de hilos

In [3]:
```
#Numero de hilos a evaluar
NH = [2,4,8,16]
```

Generando la matriz para guardar los tiempos

In [4]:
```
#Declarando la matriz TS
```

```
TC = np.zeros((100,len(N),len(NH)))
```

Realizando la ejecución.

In [5]:
```
TC.shape
```

Out[5]:
```
(100, 5, 4)
```

In [6]:
```
#s recorre el numero de simulaciones
for s in range(100):

    #n corre sobre los tamanios de la matriz
    for nt in range(len(N)):

        #Dimension de las matrices
        n = N[nt]

        for nh_ind in range(len(NH)):

            #nh es el numero de hilos
            nh = NH[nh_ind]

            print("Simulación: " + str(s) + " | Dim = " + str(n)+ " | Nhilos = "


            global A, B, C

            #Declarando dos matrices aleatorias de tamanio n.
            A = np.random.rand(n,n)
            B = np.random.rand(n,n)

            #Declarando la matriz respuesta
            C = np.zeros((n,n))

            #Tomando el tiempo inicial
            t = time.time()

            #Realizando la multiplicacion
            threads = list()

            for h in range(nh):
                th = threading.Thread(target = prodFC, args = (h,nh,n,))
                threads.append(th)
                th.start()

            for th in threads:
                th.join()

            #Guardando el tiempo
            TC[s,nt,nh_ind] = time.time() - t
```

```
Simulación: 0 | Dim = 50 | Nhilos = 2
Simulación: 0 | Dim = 50 | Nhilos = 4
Simulación: 0 | Dim = 50 | Nhilos = 8
Simulación: 0 | Dim = 50 | Nhilos = 16
Simulación: 0 | Dim = 100 | Nhilos = 2
```

In [7]:
```
TC.shape
```

Out[7]:
```
(100, 5, 4)
```

In [8]:
```
TC[:,:,0].shape
```

Out[8]:
```
(100, 5)
```

In [9]:
```
for nh_ind in range(len(NH)):
    np.savetxt("minip_conc_"+ str(NH[nh_ind]) + ".csv", TC[:,:,nh_ind], delimite
```