PRÁCTICA 1:

MULTIPLICACIÓN DE MATRICES

MARC INFANTE GARCÍA
IÑIGO ARRIAZU GARCIA
PRACTICA 1 CONVOCATORIA
Sistemas Distribuidos 2019/20
17/04/2020

Índice:

1. OBJETIVO

En esta práctica se pondrá a prueba la potencia de cálculo de un servicio como es en este de caso de IBM. Además, comprobaremos que el multithreading puede aportar mejoras significativas en cuanto a tiempo de ejecución. Durante la ejecución de este proceso es indispensable incluir en la implementación elementos del proyecto IBM-PyWren

2. FUNCIONES PRINCIPALES

matrizMultCloud(casilla_ini,num_casillas)

Función: Se encarga de calcular un cierto número de casillas de la matriz final

Parámetros de entrada

casilla_ini -> número de la primera casilla de la matriz final a calcular

num_casillas -> número de casillas a calcular

Parámetros de salida

resultados -> todos los valores calculados con el formato "[[fila,columna,valor],[...],[...]]"

Precondiciones:

- (casilla_ini >= 0) y (casilla_ini < (num_filas*num_colunas))
- ((casilla_ini +num_casillas) > 0) y ((casilla_ini +num_casillas)=< (num_filas*num_colunas))

reunirResultados(results)

Función: Recoge todos los resultados calculados y rellena una matriz que devuelve como parámetro de salida

Parámetros de entrada

results -> Lista de resultados con formato "[[RESULTADOS W_0],[...],[RESULTADOS $W_{num\_workers-1}$]

Parámetros de salida

matriz_resultado -> Matriz con todos los resultados escritos dentro

Precondiciones

Los valores de fila y columna deberán estar dentro de las dimensiones escogidas para M (número de filas) y L (número de columnas)

Postcondiciones

Si aplicamos get_result() después del map_reduce() podremos obtener la matriz que devuelve esta función.

3. FUNCIONES AUXILIARES

inicializarMatriz(rows,cols)

Función: Devuelve una matriz inicializada con números del 0 al 100 del tamaño que indican los parámetros de entrada.

Parámetros de entrada

Rows -> Numero de filas de la matriz deseada

Columns -> Numero de columnas de la matriz deseada

Parámetros de salida

matriz_resultado-> Matriz inicializada con valores de 0 a 100

guardarMatrices(mA, mB, filasA, columnasB)

Función: Guarda las filas de la matriz A y las columnas de la matriz B que se le pasan por parámetro.

Parámetros de entrada

mA -> Matriz A

mB -> Matriz B

filasA -> Número de filas de la matriz A

columnas -> número de columnas de la matriz B

Parámetros de salida

Ninguno

Precondiciones

Los valores de filasA y columnasB deben estar correctamente

CalcPosMatrix(num_casilla, rows, columns)

Función: Dado el numero de una casilla de la matriz (siendo la 0 la fila 0, columna 0 y avanzando de izquierda a derecha y de arriba a abajo) devuelve su posición, pero en formato fila y columna.

Parámetros de entrada

num_casilla -> Casilla a modificar

rows -> Filas de la matriz contenedora de num_casilla

columns -> Columnas de la matriz contenedora de num_casilla

Parámetros de salida

fila -> numero de fila de la matriz menos 1

col-> numero de columna de la matriz menos 1

Postcondiciones

Si num_casilla >= (rows*columns) la función no retornara nada.

CalcNumCasillas(workers)

Función: Dados cuantos trabajadores se quieren para el map genera una lista de valores iterables con los parámetros de entrada para matrizMultCloud() ([[casilla_ini,num_casillas],[...],[...]])

Parámetros de entrada

workers -> Número de trabajadores seleccionados para repartir la tarea

Parámetros de salida

iterdata -> Lista iterable con los parámetros de entrada del mapper

Precondiciones

Es necesario tener inicializadas los variables globales M y L que son número de filas y número de columnas de la matriz final a calcular.

sacarCasillaConcreta(fila,col)

Función: Calcula una casilla determinada de la matriz final

Parámetros de entrada

fila -> Fila donde reside la casilla a calcular

col -> Columna donde reside la casilla a calcular

Parámetros de salida

val -> Valor de la casilla

Precondiciones

Es necesario haber ejecutado anteriormente guardarMatrices() y tener inicializado el valor N (columnas de la primera matriz y filas de la segunda).

mostratMatriz(A,B,matriz)

Función: Escribe por pantalla la matriz de una forma más visible

Parámetros de entrada

A -> Número de filas de la matriz

B -> Número de columnas de la matriz

matriz -> Matriz a representar

Parámetros de salida

Ninguno

Precondiciones

A y B deben ser correctos.

matrizMultiplication(filas,columnas,comun)

Función: Calcula la multiplicación de matrices local y secuencialmente.

Parámetros de entrada

filas -> Número de filas de la matriz A

columnas - > Número de columnas de la matriz B

común -> Número de columnas de la matriz A / Numero de filas de la matriz B

Parámetros de salida

matriz_resultado -> Matriz filas*columnas con los valores ya calculados

Precondiciones

Se debe haber usado guardarMatrices() anteriormente.

4.VARIABLES GLOBALES Y CONFIGURACIONES

- MÓDULOS USADOS:

```
import random                          # CREAR NUMEROS ALEATORIOS
from time import time                  # CALCULAR TIEMPO EJECUCION
from cos_backend import COSBackend     # MANIPULACION OBJETOS EN OBJECT STORAGE
import pickle                          # MANIPULAR LISTAS
import numpy as np                     # GENERAR MATRIZ INICIALIZADA A 0
import pywren_ibm_cloud as pywren      # USO DE HERRAMIENTAS DEL CLOUD FUNCTIONS
```

- VARIABLES DE CONFIGURACIÓN PARA OBJECT STORAGE Y CLOUD FUNCTIONS

```
config_os = {'endpoint': '',
             'secret_key': '',
             'acces_key': ''}
config_cf = {'pywren': {'storage_bucket': ''},

          'ibm_cf': {'endpoint': '',
                     'namespace': '',
                     'api_key': ''},

          'ibm_cos': {'endpoint': '',
                      'private_endpoint': '',
                      'api_key': ''}}
```

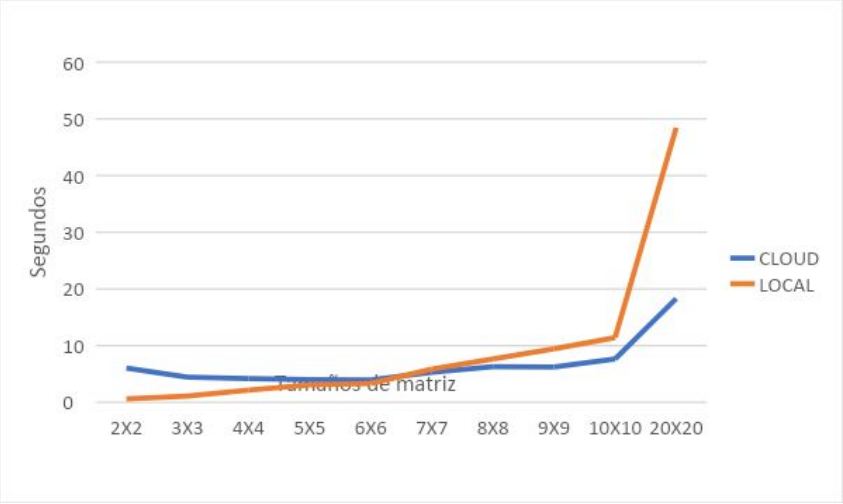
- VARIABLES GLOBALES

```
# Numero de Workers
W = 200
# tamaño de filas matriz 1 (M) y columnas matriz 1(N)/filas matriz 2(N)
M = 1000
N = 1000
# tamaño de columnas matriz 2(L)
L = 1000
```

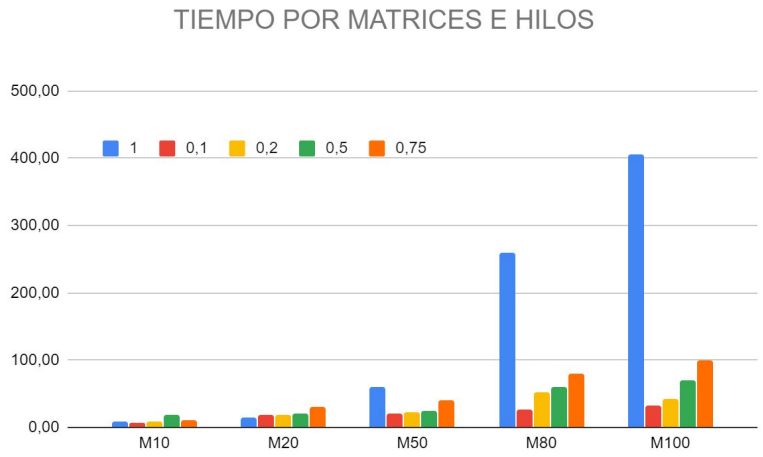
5.RESULTADOS

5.1 SECUENCIAL LOCAL VS CLOUD

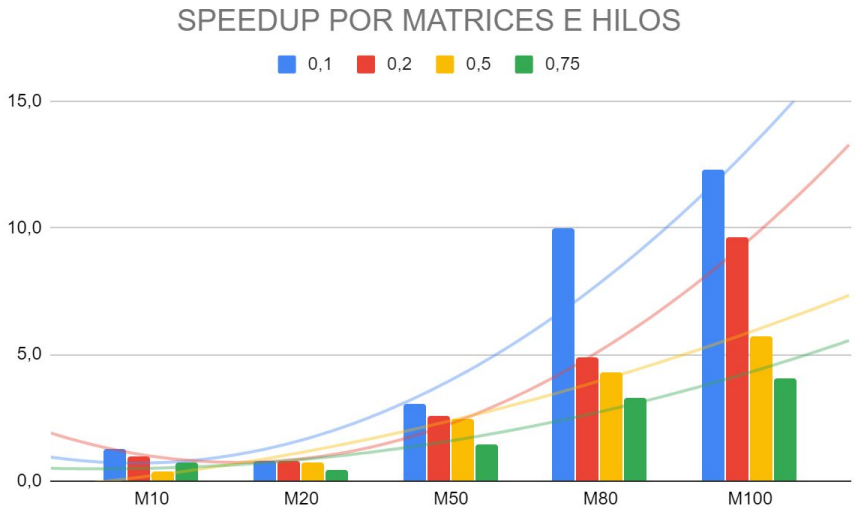
Como se puede ver claramente en el gráfico, la ejecución secuencial en el cloud puede ser un poco ineficiente para una carga de cálculo muy pequeña. Esto es debido a que solo el hecho de ejecutarlo en la nube ya tiene un coste mínimo de tiempo que posiblemente sea superior al tiempo de ejecución de la resolución local, lo que deriva en que localmente se calcule antes. De la misma forma, cuando aumentamos la carga se va viendo justamente lo contrario. Cuanta más carga se añade, más eficiente se vuelve la ejecución en el cloud respecto a la local llegando a ejecutarse a más del doble de rápido, algo que agradeceremos cuando el tiempo de ejecución sea muy largo.



Al hacer las comparaciones respecto al tiempo, se puede apreciar como el tiempo está directamente relacionado con el tamaño de la matriz. En el gráfico titulado "TIEMPO POR MATRICES E HILOS" podemos apreciar como el tiempo va aumentando de forma exponencial al tomar como referencia al procesamiento de datos de forma secuencial (azul). Cuando tomamos por referencia el porcentaje de hilos tomados vemos una notable mejoría cuando los hilos oscilan en torno al 10%.



Posteriormente se procede a analizar el SpeedUp, una medida que se emplea para obtener la mejora del rendimiento obtenida. podemos apreciar cómo de forma muy notable el SpeedUp se dispara cuando los workers oscilan el 10% de la cantidad de datos a tratar



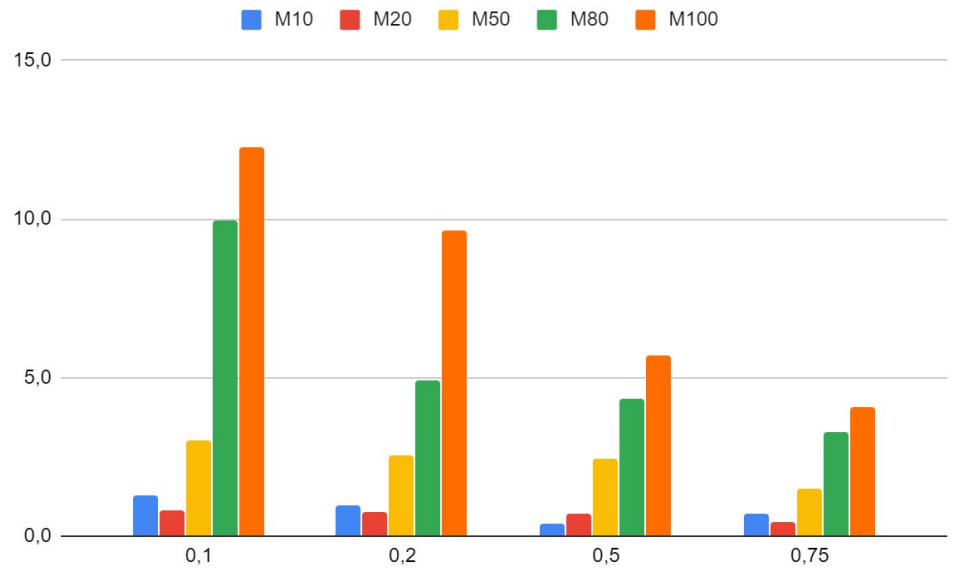
6. ANEXO I

Primero queremos comentar que los resultados obtenidos han sido muy variables tanto en la máquina local como en el servidor de IBM, en la siguiente gráfica a pesar de no aportar mucha información extra hemos añadido el tiempo de la máquina local para que se pueda apreciar este hecho. Tras hacer varias pruebas hemos optado por los siguientes valores que responden a unos valores mas próximos a la media.

Matrices	%Threads	Threads	Tiempo Cloud	Tiempo Local	SpeedUp
10 x 10	secuencial	1	8,01	15,71	1,0
100	0,1	10	6,18	20,53	1,3
	0,2	20	8,10	15,61	1,0
	0,5	50	19,50	15,76	0,4
	0,75	75	11,17	17,23	0,7
20 x 20	secuencial	1	14,40	56,60	1,0
400	0,1	40	18,34	57,05	0,8
	0,2	80	18,64	58,26	0,8
	0,5	200	19,93	70,34	0,7
	0,75	300	30,88	59,40	0,5
50 x 50	secuencial	1	59,92	363,55	1,0
2500	0,1	250	19,80	336,79	3,0
	0,2	500	23,38	363,74	2,6

	0,5	1250	24,59		2,4
	0,75	1875	40,60		1,5
80 x 80	secuencial	1	259,00	1.210,00	1,0
6400	0,1	640	26,00		10,0
	0,2	1280	53,00		4,9
	0,5	3200	60,00		4,3
	0,75	4800	79,00		3,3
100 x 100	secuencial	1	405,00	1.516,00	1,0
10000	0,1	1000	33,00		12,3
	0,2	2000	42,00		9,6
	0,5	5000	71,00		5,7
	0,75	7500	100,00		4,1

En el siguiente gráfico se muestra como para alcanzar el máximo rendimiento en función de workers diferentes se obtiene un máximo cuando estos se aproximan al 10% del número de los datos a tratar. Este hecho es aplicable únicamente a nuestro programa en cuestión.



7. REFERENCIAS ACLARACIONES Y BIBLIOGRAFÍA

Este documento y el completo de la práctica ha sido desarrollada de forma simultánea por ambos integrantes. En un inicio nos centramos principalmente en entender como funciona python. Posteriormente optamos por realizar el programa de forma local, donde las matrices se multiplican en nuestro ordenador. Por último tratamos de buscar informacion en github, sobre como podríamos llevar a cabo la practica. El código inicial ha sido modificado de forma continua e ininterrumpida hasta el día previo a la entrega.

BIBLIOGRAFIA:

[1] - Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, Pedro García-López, “Serverless Data Analytics in the IBM Cloud”. Middleware Industry 2018: 1-8
[2]IBM-PyWren - PyWren on IBM Cloud, URL: https://github.com/pywren/pywren-ibm-cloud
[3] - Asignatura Sistemas Distribuidos y apuntes asociados