Lab 1: Pruebas sobre el comportamiento de la memoria caché

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN
FACULTAD DE INGENIERÍA Y PRODUCCIÓN DE SERVICIOS
ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

CURSO: COMPUTACION PARALELA Y DISTRIBUIDA

NIFLA LLALLACACHI, MANUEL ANGEL
mnifla@unsa.edu.pe

***Abstract*—En este documento tipo informe se explica la implementación, resultados y análisis de la ejecución de diferentes problemas para el Laboratorio N° 1 del curso de Computación Paralela y Distribuida.**

Palabras clave: costo computacional, K-means.

I. EJERCICIO N°1

Implementación y comparación de 2-bucles anidados FOR presentados en el cap. 2 del libro, pag 22 del libro [1].

A. Implementación

En el código presentado en el libro se plantea la implementación de una multiplicación de matrices A y un vector x para obtener un nuevo vector y.

```
#include <iostream>
#include <chrono>
using namespace std;

const int MAX = 1000;

double A[MAX][MAX], x[MAX], y[MAX];

int main() {
    // Inicializa A y x, asigna y = 0
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            A[i][j] = 0.0;
        }
        x[i] = 0.0;
        y[i] = 0.0;
    }

    //tiempo
    auto start_time = std::chrono::
    high_resolution_clock::now();

    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            y[i] += A[i][j] * x[j];
        }
    }

auto end_time = std::chrono::
high_resolution_clock::now();
auto duration = std::chrono::
duration_cast<std::chrono::
microseconds>(end_time - start_time);

std::cout << "Tiempo primer bucle: "
<< duration.count() << " microsegundos"
<< std::endl;

// Asigna y = 0
for (int i = 0; i < MAX; i++) {
    y[i] = 0.0;
}

// Tiempo
start_time = std::chrono::
high_resolution_clock::now();

for (int j = 0; j < MAX; j++) {
    for (int i = 0; i < MAX; i++) {
        y[i] += A[i][j] * x[j];
    }
}

end_time = std::chrono::
high_resolution_clock::now();
duration = std::chrono::duration_cast
<std::chrono::microseconds>
(end_time - start_time);

std::cout << "Tiempo segundo bucle: "
<< duration.count()
<< " microsegundos" << std::endl;
```

```
    return 0;
}
```

B. Resultados

```
// Con 100 datos:
Tiempo primer bucle: 68 microsegundos
Tiempo segundo bucle: 92 microsegundos
// Con 500 datos:
Tiempo primer bucle: 1360 microsegundos
Tiempo segundo bucle: 3033 microsegundos
// Con 1000 datos:
Tiempo primer bucle: 3704 microsegundos
Tiempo segundo bucle: 9218 microsegundos
// Con 10000 datos con par de bucles:
Tiempo para el primer bucle:
425059 microsegundos
Tiempo para el segundo bucle:
2170458 microsegundos
```

C. Análisis

El primer bucle tiene menor tiempo de ejecucion debido a que accede a los datos de la matriz bidimensionalmente en bloques contiguos como se vé en la figura 1, lo que no ocurre con el segundo buble debido a que lo recorremos en forma de columnas y filas como se ve en la figura 2, ocurriendo lo contrario.

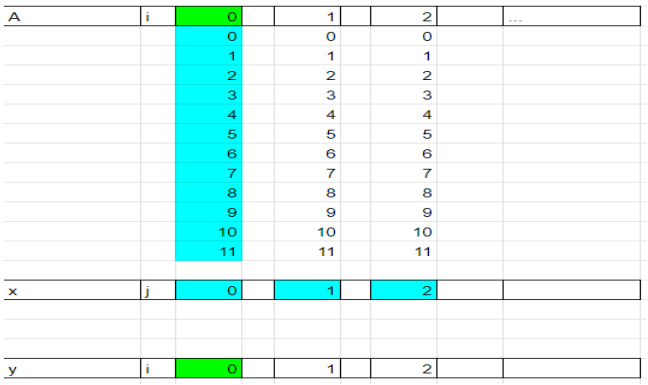

Fig. 1: Bucle con bloques contiguos.

II. Ejercicio N° 2

Implementar en C/C++ la multiplicación de matrices clásica, la versión de tres bucles anidados y evaluar su desempeño considerando diferentes tamaños de matriz.

A. Implementación

La multiplicación de matrices clásica es un algoritmo básico que sigue la definición matemática de la multiplicación de matrices. Dado dos matrices A y B de dimensiones n×n, el elemento en la posición C[i][j] de la matriz resultante C se obtiene sumando los productos de los elementos correspondientes de la fila i de A y la columna j de B.

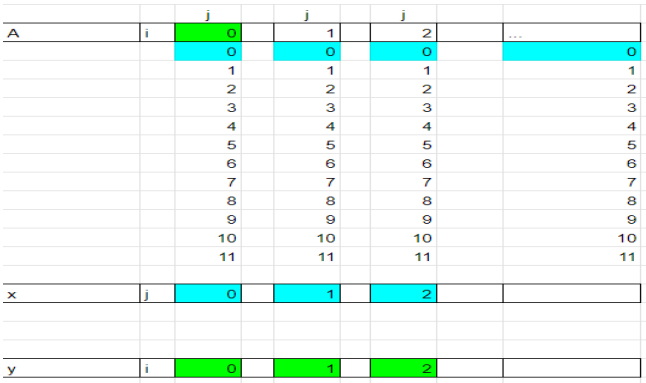

Fig. 2: Bucle de columnas y filas.

```
#include <iostream>
#include <chrono>
using namespace std;

const int MAX = 2000;

double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];

int main() {
    // Inicializa A y x, asigna y = 0
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            A[i][j] = B[i][j] = 1.0;
            C[i][j] = 0.0;
        }
    }

    //tiempo
    auto start_time = std::chrono::
    high_resolution_clock::now();

    for (int i = 0; i < MAX; ++i) {
        for (int j = 0; j < MAX; ++j) {
            for (int k = 0; k < MAX; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    auto end_time = std::chrono::
    high_resolution_clock::now();
    auto duration = std::chrono::
    duration_cast<std::chrono::
    microseconds>(end_time - start_time);

    std::cout << "Tiempo primer bucle: "
    << duration.count() << " microsegundos"
    << std::endl;
```

```
    return 0;
}
```

B. Resultados

```
//Con 100 datos:
Tiempo primer bucle: 4995 microsegundos
//Con 500 datos:
Tiempo primer bucle: 438969 microsegundos
//Con 1000 datos:
Tiempo primer bucle: 4129539 microsegundos
//Con 2000 datos
Tiempo primer bucle: 50934419 microsegundos
```

C. Análisis

Este algoritmo tiene una complejidad de O(n$\hat{3}$) porque cada uno de los tres bucles anidados recorre n elementos, siendo n el tamaño de las matrices.

Este método sencillo no es optimizado para la jerarquía de memoria (caché), por lo que puede tener un rendimiento subóptimo en implementaciones grandes sin optimizaciones adicionales.

III. Ejercicio N° 3

Implementar la versión por bloques (investigar en internet), seis bucles anidados, evaluar su desempeño y compararlo con la multiplicación de matrices clásica.

A. Implementación

La multiplicación de matrices por bloques es una optimización de la multiplicación de matrices clásica, diseñada para mejorar el rendimiento en sistemas con jerarquía de memoria (memoria caché). En lugar de multiplicar matrices elemento por elemento, el algoritmo divide las matrices en submatrices (bloques) más pequeños y realiza operaciones sobre estos bloques, lo que permite que los datos se mantengan más tiempo en la caché, reduciendo la cantidad de accesos a la memoria principal.

```
#include <iostream>
#include <chrono>
using namespace std;

const int MAX = 2000;

double A[MAX][MAX], B[MAX][MAX],
C[MAX][MAX];

int main()
{
    int Bloques = 64;
    // Inicializa A y x, asigna y = 0
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            A[i][j] = B[i][j] = 1.0;
            C[i][j] = 0.0;
        }
    }

    auto start_time = std::chrono::
    high_resolution_clock::now();

    for (int I = 0; I < MAX; I += Bloques)
    {
    for (int J = 0; J < MAX; J += Bloques)
    {
    for (int K = 0; K < MAX; K += Bloques)
    {
    for (int i = I; i < min(I + Bloques, MAX); ++i
    {
    for (int j = J; j < min(J + Bloques, MAX); ++j
    {
    int sum = 0;
    for (int k = K; k < min(K + Bloques, MAX); ++k
    {
        sum += A[i][k] * B[k][j];
    }
    C[i][j] += sum;
    }
    }
    }
    }
    }

    auto end_time = std::chrono::
    high_resolution_clock::now();

    auto duration = std::chrono::
    duration_cast<std::chrono::
    microseconds>(end_time - start_time);
    std::cout << "Tiempo primer bucle: "
    << duration.count() << " microsegundos"
    << std::endl;

    return 0;
}
```

B. Resultados

```
//Con 100 datos:
Tiempo primer bucle: 6997 microsegundos
//Con 500 datos:
Tiempo primer bucle: 625204 microsegundos
//Con 1000 datos:
Tiempo primer bucle: 4975500 microsegundos
//Con 2000 datos:
Tiempo primer bucle: 37875690 microsegundos
```

C. Análisis

Codigo:
https://github.com/AngelNifla/EPCC_COMPUTACION_PARALELA_Y_DISTRIBUIDA_2024/tree/master/LABS/LAB01

REFERENCES

[1] P. S. Pacheco, *An Introduction to Parallel Programming*. Burlington, MA, USA: Morgan Kaufmann, 2011.