# Universidad Nacional de San Agustín de Arequipa

## Escuela Profesional de Ciencia de la Computación
## Computación Paralela y Distribuida



---

# Laboratorio 1

---


***Presentado por:***
Merisabel Ruelas Quenaya

***Docente :***
Alvaro Mamani Aliaga

# Laboratorio 01

## 1. Ejercicio 1

Implementar y comparar los 2-bucles anidados FOR presentados en el cap. 2 del libro, pag 22.

```
#include <chrono>
#include <iostream>
#include <sys/time.h>
#include <ctime>
#include <cstdlib>

using namespace std;

const int MAX = 500;
double A[MAX][MAX], x[MAX], y[MAX];

void init() {
    for(int i = 0; i < MAX ; i++) {
        y[i] = 0;
        x[i] = 1 + rand() % 9; // Genera n meros entre 1 y 9
        for(int j = 0 ; j < MAX ; j++)
            A[i][j] = 1 + rand() % 100; // Genera n meros entre 1 y 100
    }
}

double calcTime(clock_t t0, clock_t t1) {
    return static_cast<double>(t1 - t0) / CLOCKS_PER_SEC;
}

void nestedLoop1() {
    clock_t t0, t1;
    init();
    t0 = clock();
    for (int i = 0; i < MAX; i++)
        for (int j = 0; j < MAX; j++)
            y[i] += A[i][j] * x[j];
    t1 = clock();
    cout << "Tiempo de Ejecuci n del Bucle Anidado 1: " << calcTime(t0,
    t1) << " segundos" << endl;
}

void nestedLoop2() {
    clock_t t0, t1;
    init();
    t0 = clock();
    for (int j = 0; j < MAX; j++)
        for (int i = 0; i < MAX; i++)
            y[i] += A[i][j] * x[j];
    t1 = clock();
    cout << "Tiempo de Ejecuci n del Bucle Anidado 2: " << calcTime(t0,
    t1) << " segundos" << endl;
```

```
}

int main() {
    srand(time(0)); // Semilla para n meros aleatorios
    nestedLoop1();
    nestedLoop2();
    return 0;
```

Bucle Anidado 1 (por filas): 0.001205 segundos Bucle Anidado 2 (por columnas): 0.002199 segundos

El Bucle Anidado 1 es claramente más rápido, tomando aproximadamente la mitad del tiempo que el Bucle Anidado 2.

Ambos bucles muestran cierta variabilidad en sus tiempos de ejecución, lo cual es normal.

El Bucle Anidado 1, que recorre la matriz por filas, es significativamente más eficiente que el Bucle Anidado 2, que la recorre por columnas.

## 2. Ejercicio 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.

```
#include <iostream>
#include <vector>
#include <chrono>

using namespace std;
using namespace std::chrono;

void classicMatrixMultiplication(vector<vector<int>>& A, vector<vector<
    int>>& B, vector<vector<int>>& C, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    vector<int> sizes = {128, 256, 512, 1024};

    for (int N : sizes) {
        vector<vector<int>> A(N, vector<int>(N, 1));
        vector<vector<int>> B(N, vector<int>(N, 2));
        vector<vector<int>> C(N, vector<int>(N, 0));
```

```
        cout << "Multiplicacion clasica para tamanio de matriz " << N <<
    "x" << N << ":" << endl;

        auto start = high_resolution_clock::now();
        classicMatrixMultiplication(A, B, C, N);
        auto stop = high_resolution_clock::now();

        auto duration = duration_cast<milliseconds>(stop - start);
        cout << "Duracion: " << duration.count() << " ms" << endl <<
   endl;
    }

    return 0;
}
```

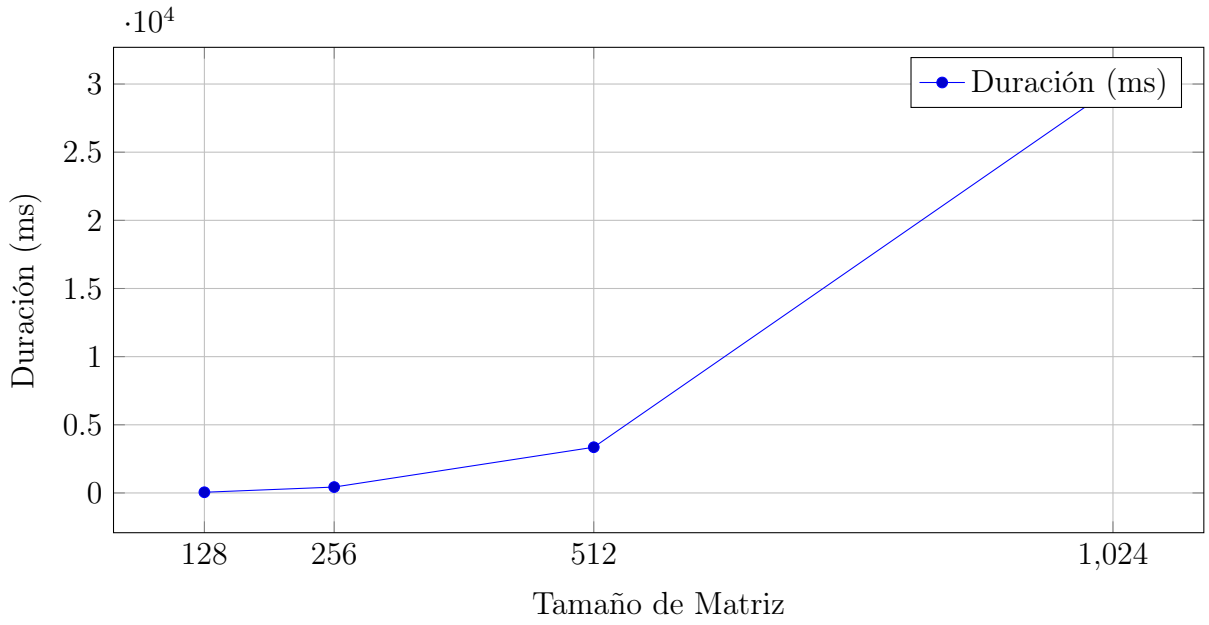
## Resultados de la Multiplicación Clásica de Matrices

### Tabla

| Tamaño de Matriz | Duración (ms) |
|---|---|
| 128x128 | 51 |
| 256x256 | 432 |
| 512x512 | 3355 |
| 1024x1024 | 29725 |

Cuadro 1: Duración de la multiplicación clásica de matrices para distintos tamaños.

**Gráfico**




# 3. Ejercicio 3

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

```
#include <iostream>
#include <vector>
#include <chrono>

using namespace std;
using namespace std::chrono;

void classicMatrixMultiplication(vector<vector<int>>& A, vector<vector<
    int>>& B, vector<vector<int>>& C, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void blockMatrixMultiplication(vector<vector<int>>& A, vector<vector<int
    >>& B, vector<vector<int>>& C, int N, int blockSize) {
    for (int ii = 0; ii < N; ii += blockSize) {
        for (int jj = 0; jj < N; jj += blockSize) {
            for (int kk = 0; kk < N; kk += blockSize) {
                // Multiplicar los bloques
                for (int i = ii; i < min(ii + blockSize, N); i++) {
```

```
                    for (int j = jj; j < min(jj + blockSize, N); j++) {
                        for (int k = kk; k < min(kk + blockSize, N); k
    ++) {
                            C[i][j] += A[i][k] * B[k][j];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    vector<int> sizes = {128, 256, 512, 1024};

    int blockSize = 64;

    for (int N : sizes) {
        cout << "Probando multiplicaci n para tamanio de matriz " << N
    << "x" << N << ":" << endl;

        vector<vector<int>> A(N, vector<int>(N, 1));
        vector<vector<int>> B(N, vector<int>(N, 2));
        vector<vector<int>> C(N, vector<int>(N, 0));

        auto start = high_resolution_clock::now();
        classicMatrixMultiplication(A, B, C, N);
        auto stop = high_resolution_clock::now();
        auto durationClassic = duration_cast<milliseconds>(stop - start)
    ;
        cout << "Duracion (clasica): " << durationClassic.count() << "
    ms" << endl;

        fill(C.begin(), C.end(), vector<int>(N, 0));

        start = high_resolution_clock::now();
        blockMatrixMultiplication(A, B, C, N, blockSize);
        stop = high_resolution_clock::now();
        auto durationBlock = duration_cast<milliseconds>(stop - start);
        cout << "Duracion (por bloques): " << durationBlock.count() << "
     ms" << endl << endl;
    }

    return 0;
}
```

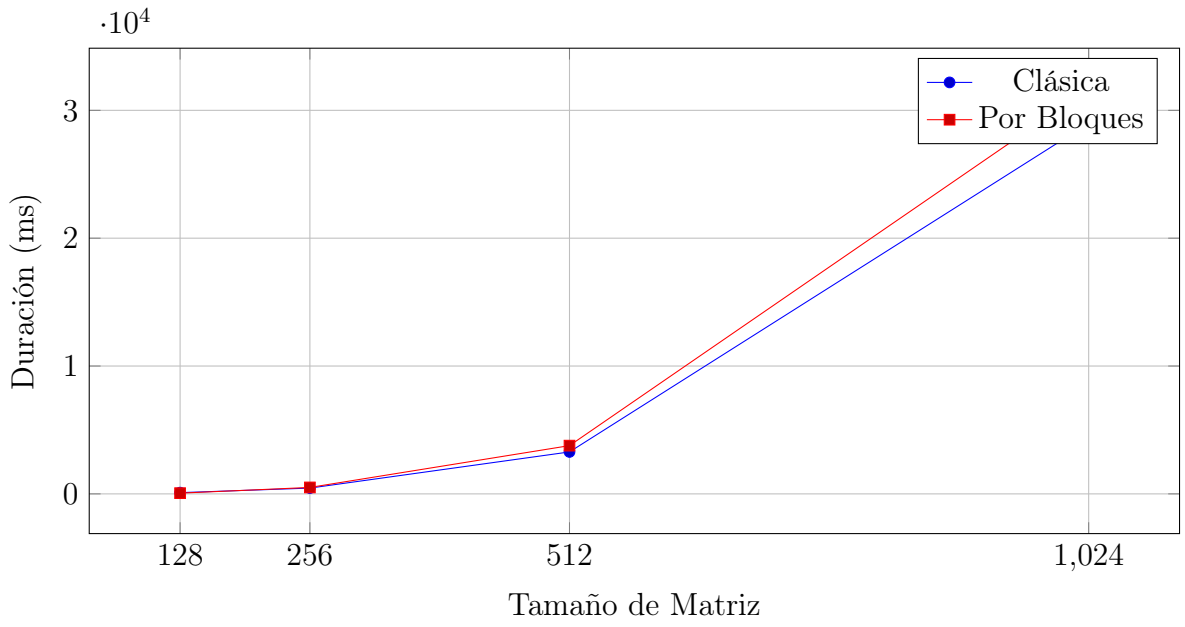
## Comparación de Multiplicación Clásica vs. Multiplicación por Bloques

### Tabla

| Tamaño de Matriz | Duración Clásica (ms) | Duración por Bloques (ms) |
|---|---|---|
| 128x128 | 97 | 57 |
| 256x256 | 453 | 504 |
| 512x512 | 3281 | 3774 |
| 1024x1024 | 28986 | 31695 |

Cuadro 2: Comparación de duración entre la multiplicación clásica y la multiplicación por bloques.

**Gráfico**



# 4. Ejercicio 4

Ejecutar ambos algoritmos paso a paso, y analizar el movimiento de datos entre la memoria principal y la memoria cache. Hacer una evaluación de acuerdo a la complejidad algorítmica.

## 4.1. Multiplicación Clásica

- Accesos a Memoria: Accede a elementos de ‘A‘ y ‘B‘ repetidamente.
- Caché: Acceso secuencial puede causar fallos de caché.
- Complejidad Temporal: $O(N^3)$. Tres bucles anidados, cada uno recorriendo N elementos.
- Complejidad Espacial: $O(N^2)$. Se necesitan matrices ‘A‘, ‘B‘ y ‘C‘, cada una con $N \times N$ elementos.

## 4.2. Multiplicación por Bloques

- Accesos a Memoria: Menos accesos a memoria principal, mejora el uso de la caché. Cada bloque se multiplica dentro de la caché, lo que reduce el número de accesos a memoria principal.
- Caché: Mejor localización temporal y espacial dentro de bloques. La localización espacial es mejorada porque los datos accesibles en un bloque se mantienen en caché durante más tiempo, reduciendo fallos de caché.
- Complejidad Temporal: $O(N^3)$.
- Complejidad Espacial: $O(N^2)$.