Maestría:

Inteligencia Artificial Aplicada (MNA)

Curso:

Cómputo en la nube

Tarea 1

Programación de una solución paralela

Estudiantes:

Alexys Martín Coate Reyes A01746998

Profesores:

Dr. Gilberto Echeverría Furió

Fecha de Entrega:

1 de febrero del 2026

Contenido

Tabla de Ilustraciones

Tarea 1 - Programación de una solución paralela

Introducción

La computación en paralelo es una técnica de optimización de software para que el código pueda ejecutarse de manera más rápida y eficiente, aprovechando todos los hilos del procesador. De esta manera podemos lograr reducir tiempos de computo de gran manera en cualquier tipo de sistema, sea local o en la nube. Este tipo de optimizaciones tiene gran impacto para tareas más complejas y tardadas.

En esta actividad se utiliza OpenMP (Open Multi-Processing) que es una API que permite escribir este código en paralelo en C/C++ y Fortran sin cambiar la base entera del código.

En este caso demostraremos la utilización de la librería aplicada a la suma de 2 arreglos A y B, que es la suma resultante de uno tercero llamado C. Esta suma se realiza mediante los subíndices de cada arreglo de la siguiente forma:

- a[0] + b[0] = c[0]
- a[1] + b[1] = c[1]
- a[2] + b[2] = c[2]
- ...
- a[N] + b[N] = c[N]

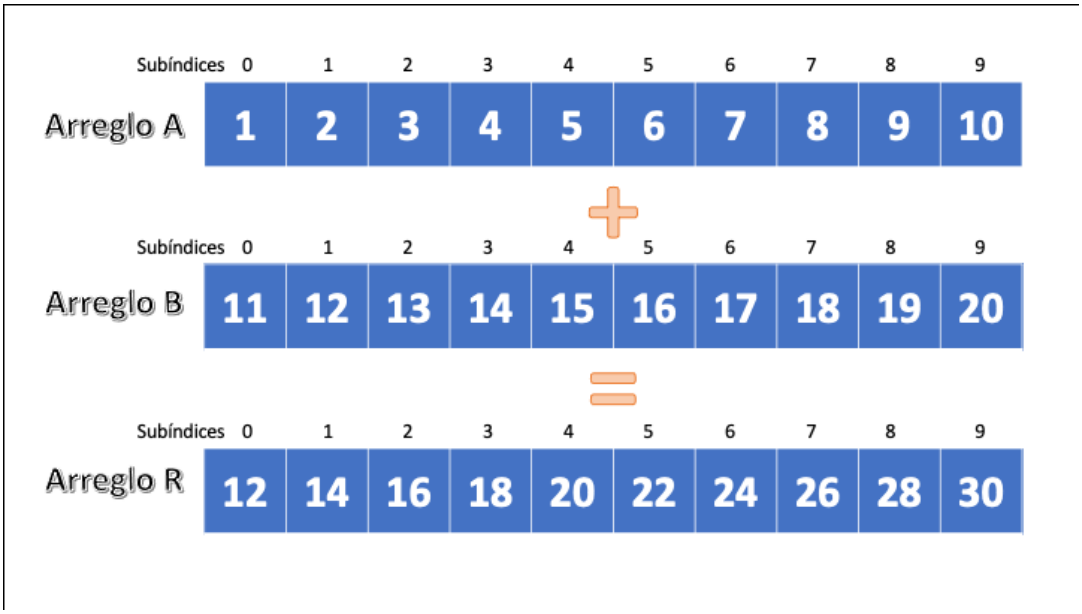

Figura 1 - Suma de arreglos A y B y resultado (Arreglo C)

La manera en que implementaremos la paralelización dentro del programa será utilizando un ciclo for que aproveche todos los hilos del procesador de la computadora para ejecutar la instrucción de suma por partes del arreglo resultante.

Figura 2 - Ejemplo de distribución de la suma en los hilos del procesador

Repositorio de Github

Link: https://github.com/AlexysCR/MNA_Computo-en-la-nube/tree/main/Tarea%201%20-%20Programaci%C3%B3n%20de%20una%20soluci%C3%B3n%20paralela

Capturas de pantalla

En todas las capturas de pantalla con las ejecuciones podemos comprobar que se está cumpliendo que la suma del elemento “n” del arreglo C, corresponde a la suma del arreglo A y B en la posición “n”.

Ejecución 1

Parámetros

```
#define N 1000          // Número de elementos en los arreglos
#define chunk 100       // Tamaño del chunk para la distribución estática
#define mostrar 10      // Número de elementos a mostrar en la terminal
```

```
Sumando Arreglos en Paralelo!
Imprimiendo los primeros 10 valores del arreglo a:
66 - 20 - 15 - 66 - 28 - 21 - 29 - 58 - 29 - 52 -
Imprimiendo los primeros 10 valores del arreglo b:
48 - 17 - 97 - 42 - 46 - 60 - 83 - 72 - 56 - 30 -
Imprimiendo los primeros 10 valores del arreglo c:
114 - 37 - 112 - 108 - 74 - 81 - 112 - 130 - 85 - 82 -
```

Ejecución 2

Parámetros

```
#define N 50000          // Número de elementos en los arreglos
#define chunk 2000       // Tamaño del chunk para la distribución estática
#define mostrar 20      // Número de elementos a mostrar en la terminal
```

```
Sumando Arreglos en Paralelo!
Imprimiendo los primeros 20 valores del arreglo a:
69 - 94 - 64 - 97 - 48 - 5 - 4 - 42 - 88 - 57 - 43 - 60 - 37 - 72 - 10 - 27 - 71 - 67 - 20 - 81 -
Imprimiendo los primeros 20 valores del arreglo b:
48 - 85 - 33 - 28 - 49 - 95 - 47 - 2 - 78 - 31 - 49 - 50 - 70 - 83 - 78 - 5 - 73 - 53 - 80 - 65 -
Imprimiendo los primeros 20 valores del arreglo c:
117 - 179 - 97 - 125 - 97 - 100 - 51 - 44 - 166 - 88 - 92 - 110 - 107 - 155 - 88 - 32 - 144 - 120 - 100 - 146 -
```

Ejecución 3

Parámetros

```
#define N 7500          // Número de elementos en los arreglos
#define chunk 10       // Tamaño del chunk para la distribución estática
#define mostrar 15      // Número de elementos a mostrar en la terminal
```

```
Sumando Arreglos en Paralelo!
Imprimiendo los primeros 15 valores del arreglo a:
95 - 10 - 43 - 82 - 38 - 60 - 51 - 68 - 9 - 91 - 31 - 15 - 20 - 77 - 45 -
Imprimiendo los primeros 15 valores del arreglo b:
37 - 54 - 45 - 57 - 18 - 87 - 98 - 55 - 19 - 15 - 12 - 88 - 62 - 60 - 73 -
Imprimiendo los primeros 15 valores del arreglo c:
132 - 64 - 88 - 139 - 56 - 147 - 149 - 123 - 28 - 106 - 43 - 103 - 82 - 137 - 118 -
```

Código

```
#include <iostream>
#include <omp.h>
#include <ctime>    // Necesario para time()
#include <cstdlib>  // Necesario para rand() y srand()

#define N 1000          // Número de elementos en los arreglos
#define chunk 100       // Tamaño del chunk para la distribución estática
#define mostrar 10      // Número de elementos a mostrar en la terminal

void imprimeArreglo(float* d);

int main()
{
    std::cout << "Sumando Arreglos en Paralelo!\n";
    float a[N], b[N], c[N];
    int i;

    // Inicializar la semilla para números aleatorios reales
    srand(time(0));

    // Llenado de arreglos con números aleatorios
    for (i = 0; i < N; i++)
    {
        // Genera números aleatorios entre 0 y 99
```

```
        a[i] = static_cast<float>(rand() % 100);
        b[i] = static_cast<float>(rand() % 100);
    }

    int pedazos = chunk;

    // Directiva de OpenMP para paralelizar el ciclo for
#pragma omp parallel for \
    shared(a, b, c, pedazos) private(i) \
    schedule(static, pedazos)

    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i];

    // Impresión de resultados
    std::cout << "Imprimiendo los primeros " << mostrar << " valores del
arreglo a: " << std::endl;
    imprimeArreglo(a);
    std::cout << "Imprimiendo los primeros " << mostrar << " valores del
arreglo b: " << std::endl;
    imprimeArreglo(b);
    std::cout << "Imprimiendo los primeros " << mostrar << " valores del
arreglo c: " << std::endl;
    imprimeArreglo(c);

    return 0;
}

void imprimeArreglo(float* d)
{
    for (int x = 0; x < mostrar; x++)
    {
        std::cout << d[x] << " - ";
    }
    std::cout << std::endl;
}
```

Explicación del código

1. Inicialización de variables

```
   #define N 1000          // Número de elementos en los arreglos
   #define chunk 100       // Tamaño del chunk para la distribución estática
   #define mostrar 10      // Número de elementos a mostrar en la terminal
```

2. Generación de números aleatorios

```
std::cout << "Sumando Arreglos en Paralelo!\n";
    float a[N], b[N], c[N];
    int i;
```

```
    // Inicializar la semilla para números aleatorios reales
    srand(time(0));

    // Llenado de arreglos con números aleatorios
    for (i = 0; i < N; i++)
    {
        // Genera números aleatorios entre 0 y 99
        a[i] = static_cast<float>(rand() % 100);
        b[i] = static_cast<float>(rand() % 100);
    }
```

3. Creación de la paralelización

```
   int pedazos = chunk;

   // Directiva de OpenMP para paralelizar el ciclo for
   #pragma omp parallel for \
       shared(a, b, c, pedazos) private(i) \
       schedule(static, pedazos)

       for (i = 0; i < N; i++)
           c[i] = a[i] + b[i];
```

#pragma omp parallel for: Esta línea le dice al compilador que el siguiente ciclo for no debe ser ejecutado por un solo hilo, sino repartido.

shared(a, b, c, pedazos): Indica que todos los hilos comparten y pueden ver estos arreglos distribuyéndolos en los chunks que les corresponden.

Private(i): Indica que cada hilo tiene su propio contador en el ciclo for, por eso se define como privada

schedule(static, pedazos): Divide las 1000 (N) iteraciones en bloques de 100 (chunks) o valor que corresponda a las variables iniciales. El hilo 0 toma de la 0 a la 99, el hilo 1 de la 100 a la 199, y así sucesivamente dependiendo el valor de los chunks y la longitud del arreglo final, así como la cantidad de hilos del procesador donde se está ejecutando el programa.

4. Impresión de resultados

```
// Impresión de resultados
    std::cout << "Imprimiendo los primeros " << mostrar << " valores del
arreglo a: " << std::endl;
    imprimeArreglo(a);
    std::cout << "Imprimiendo los primeros " << mostrar << " valores del
arreglo b: " << std::endl;
    imprimeArreglo(b);
    std::cout << "Imprimiendo los primeros " << mostrar << " valores del
arreglo c: " << std::endl;
    imprimeArreglo(c);

    return 0;
```

```
}

void imprimeArreglo(float* d)
{
    for (int x = 0; x < mostrar; x++)
    {
        std::cout << d[x] << " - ";
    }
    std::cout << std::endl;
}
```

Resultados

Tras realizar diversas pruebas con distintas escalas de datos (desde N=1,000 hasta N=50,000), se verificó que la suma de cada elemento en la posición “N” es consistente y correcta en todos los casos. Gracias a esto podemos determinar que el procesador puedo paralelizar de manera correcta la tarea y realizar la suma exitosamente, aumentando la eficiencia y minimizando el tiempo de ejecución en comparación de un algorimto serializado.

Reflexión

La programación en paralelo nos obliga a nosotros como programadores a pensar de manera distinta los problemas. comúnmente pensamos en como resolverlos de manera serializada (paso 1, paso 2, paso 3), mientras que, al solucionar problemas en paralelo, no solo nos involucramos con la ejecución de instrucciones simultaneas, sino también manejo de memoria independiente y compartido. Así como en la distribución más efectiva de los trabajos entre los diferentes hilos por medio de la utilización del tamaño de chunks adecuados para cada tamaño distinto de instrucción (granularidad)