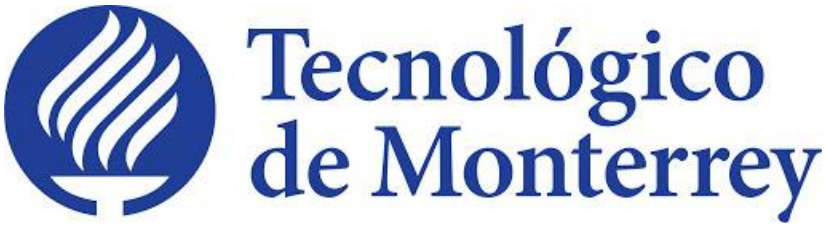


INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Tarea 1. Programación de una solución paralela

(Suma de Arreglos en Paralelo)

Lázaro Romel Ceja Rodríguez

A01795989

Cómputo en la Nube

26 de Enero 2025

Introducción

La programación paralela nos permite hacer que los programas sean más rápidos dividiendo el trabajo entre varios hilos. En este proyecto, quise exploré esta técnica desarrollando un programa en C++ que utiliza OpenMP para sumar dos arreglos en paralelo. La idea es aprovechar al máximo los recursos de hardware moderno para resolver problemas de manera más eficiente.

Fue interesante ver cómo la programación paralela nos da la oportunidad de tomar tareas comunes, como sumar elementos de arreglos, y optimizarlas para que sean más rápidas y escalables. Esto es especialmente útil cuando trabajamos con grandes cantidades de datos o procesadores con múltiples núcleos. La programación paralela es una técnica utilizada para acelerar la ejecución de programas mediante la distribución de tareas entre varios hilos o procesos. En esta tarea, se desarrolló un programa en C++ con la librería OpenMP para sumar dos arreglos en paralelo, dividiendo las operaciones entre varios hilos de ejecución.

El objetivo principal es demostrar cómo la programación paralela puede ser aplicada para resolver problemas comunes de manera eficiente, aprovechando las capacidades de los procesadores modernos.

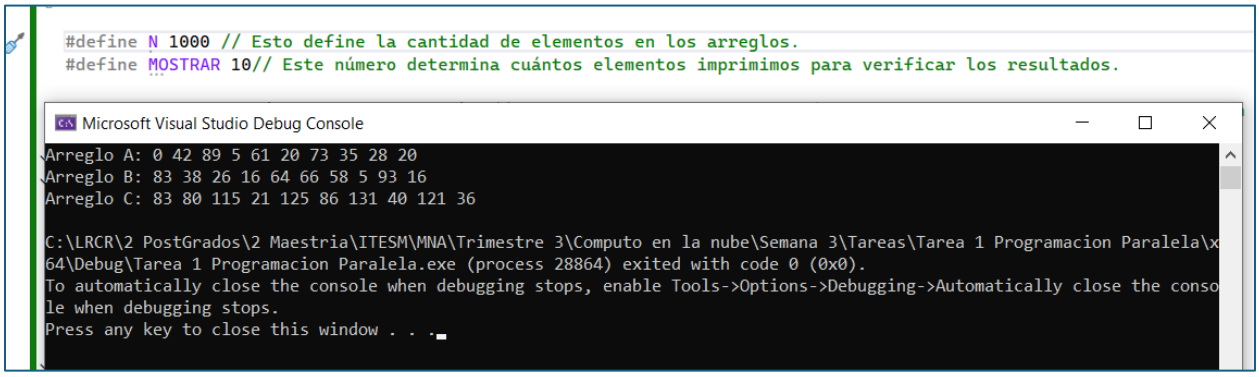
Liga al repositorio de GitHub

https://github.com/A01795989/TC4031

Capturas de Pantalla

Primer Ejecución.

Captura de los primeros elementos de los arreglos A, B y C con un valor de N = 1000 y MOSTRAR = 10.



Aquí lo que noté es que si no se defino el CHUNK el valor default sería:

$$\text{Tamaño del bloque (CHUNK)} = \frac{\text{N}}{\text{Número de hilos disponibles}}$$

Lo que sería 4 hilos de 250 iteraciones.

Segunda Ejecución.

Captura mostrando la salida cuando se modificó el valor de CHUNK. En este caso, CHUNK controla cómo OpenMP divide las iteraciones del bucle en bloques (chunks) que serán procesados por cada hilo. Por ejemplo, al cambiar #define CHUNK 100 a #define CHUNK 50 y 25, podemos observar cómo afecta la distribución del trabajo entre los hilos.

CHUNK 100 , N 100

Solo un bloque, probablemente ejecutado por un único hilo.

```
#include <ctime>   // Para inicializar la semilla aleatoria

#define N 100 // Esto define la cantidad de elementos en los arreglos.
#define MOSTRAR 10// Este número determina cuántos elementos imprimimos para verificar los resultados.
#define CHUNK 100 // Tamaño de los bloques asignados a cada hilo.
```

```
Microsoft Visual Studio Debug Console
Arreglo A: 59 71 75 64 93 88 59 77 37 70
Arreglo B: 43 36 78 31 81 75 55 26 57 91
Arreglo C: 102 107 153 95 174 163 114 103 94 161

C:\LRCR\2 PostGrados\2 Maestria\ITESM\MNA\Trimestre 3\Computo en la nube\Semana 3\Tareas\Tarea 1 Programacion Paralela\x
64\Debug\Tarea 1 Programacion Paralela.exe (process 19192) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

CHUNK 50 , N 100

Dos bloques, divididos entre los hilos disponibles.

```
#define N 100 // Esto define la cantidad de elementos en los arreglos.
#define MOSTRAR 10// Este número determina cuántos elementos imprimimos para verificar los resultados.
#define CHUNK 50 // Tamaño de los bloques asignados a cada hilo.
```

```
Microsoft Visual Studio Debug Console
Arreglo A: 16 61 90 8 25 20 85 49 89 83
Arreglo B: 98 67 30 7 37 80 64 46 28 39
Arreglo C: 114 128 120 15 62 100 149 95 117 122

C:\LRCR\2 PostGrados\2 Maestria\ITESM\MNA\Trimestre 3\Computo en la nube\Semana 3\Tareas\Tarea 1 Programacion Paralela\x
64\Debug\Tarea 1 Programacion Paralela.exe (process 9488) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

CHUNK 25 , N 100

Cuatro bloques, lo que distribuye aún más las iteraciones entre los hilos.

```
#define N 100 // Esto define la cantidad de elementos en los arreglos.
#define MOSTRAR 10// Este número determina cuántos elementos imprimimos para verificar los resultados.
#define CHUNK 25 // Tamaño de los bloques asignados a cada hilo.
```

```
Microsoft Visual Studio Debug Console
Arreglo A: 79 34 64 1 62 63 78 54 55 8
Arreglo B: 14 2 75 0 84 76 20 7 99 34
Arreglo C: 93 36 139 1 146 139 98 61 154 42

C:\LRCR\2 PostGrados\2 Maestria\ITESM\MNA\Trimestre 3\Computo en la nube\Semana 3\Tareas\Tarea 1 Programacion Paralela\x
64\Debug\Tarea 1 Programacion Paralela.exe (process 20284) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Explicación del código y los resultados

Declaración de constantes y variables

- **Constantes**: Se definieron tres constantes principales:
    - **N:** Define el tamaño de los arreglos y el número de iteraciones del bucle paralelo. Por ejemplo, N = 1000 significa que cada arreglo tiene 1000 elementos.
    - **CHUNK:** Controla cómo OpenMP divide el bucle en bloques de iteraciones para distribuir el trabajo entre los hilos. Si no se define, OpenMP calcula automáticamente el tamaño de los bloques.
    - **MOSTRAR:** Determina cuántos elementos del arreglo se imprimen para verificar los resultados.
- **Arreglos**: Se declararon tres arreglos:
    - **a y b:** Contienen los datos iniciales generados aleatoriamente.
    - **c:** Almacena el resultado de la suma de los elementos de a y b.

Generación de datos iniciales

- Los valores en los arreglos **a** y **b** se generan de manera aleatoria utilizando la función rand() en un rango de 0 a 99. Esto asegura que los datos sean variados en cada ejecución.

```
// Llenamos los arreglos "a" y "b" con números aleatorios entre 0 y 99.
for (int i = 0; i < N; i++) {
    a[i] = rand() % 100; // Aquí llenamos "a".
    b[i] = rand() % 100; // Y aquí llenamos "b".
}
```

Paralelización del bucle

El bucle que realiza la suma de los arreglos se paralelizó utilizando OpenMP:

```
    // Ahora hacemos la suma en paralelo.
    // OpenMP distribuye las operaciones entre varios hilos, así que cada hilo suma una parte de los arreglos.
#pragma omp parallel for shared(a, b, c) private(i)
    for (i = 0; i < N; i++) {
        c[i] = a[i] + b[i]; // Aquí sumamos los elementos correspondientes de "a" y "b".
    }
```

Directiva #pragma omp parallel for:

- Esta directiva indica que el bucle debe ser ejecutado en paralelo.
- OpenMP divide las iteraciones entre los hilos disponibles en el sistema.

Cláusulas utilizadas:

- **shared(a, b, c):** Los arreglos son compartidos entre los hilos para garantizar que todos accedan a los mismos datos.
- **private(i):** La variable de control del bucle (i) es privada para cada hilo, evitando conflictos durante la ejecución.
- **schedule(static, CHUNK):** Controla cómo se distribuyen las iteraciones entre los hilos. Si CHUNK no está definido, OpenMP lo calcula automáticamente.

Resultados obtenidos

El programa suma correctamente los elementos correspondientes de a y b, almacenando los resultados en c. Esto se valida imprimiendo los primeros elementos de cada arreglo.

- **Comportamiento con CHUNK**:
  - Cuando CHUNK = 100 con N = 1000, OpenMP divide las iteraciones en 10 bloques de 100 elementos cada uno.

- Con CHUNK = 50, se generarían 20 bloques más pequeños, lo que mejora la granularidad de la distribución.
- Al usar CHUNK = 25, OpenMP distribuiría 40 bloques, maximizando el balanceo de carga entre los hilos.

(Las capturas ejemplo fueron con N=100)

Reflexión sobre la programación paralela

Trabajar con programación paralela fue una experiencia muy interesante. Al principio, puede parecer complicado entender cómo dividir el trabajo entre varios hilos, pero una vez que comprendes cómo funcionan las directivas como #pragma omp, el proceso se vuelve más intuitivo.

Lo que más me gustó fue ver cómo pequeños ajustes, como modificar el valor de CHUNK, pueden tener un impacto tan grande en la ejecución del programa. Por ejemplo, al usar bloques más grandes, algunos hilos terminan más rápido y otros quedan inactivos, mientras que bloques más pequeños ayudan a equilibrar mejor la carga. Esto me hizo pensar sobre lo importante que es conocer el hardware en el que corre tu programa para aprovechar al máximo los recursos disponibles.

Otro punto que me llamó la atención fue lo fácil que OpenMP hace la paralelización. Con solo unas pocas líneas de código, pude dividir el trabajo y usar múltiples núcleos de mi procesador.

Me quedo con la idea de que, aunque la paralelización puede requerir un poco más de planeación, los beneficios en términos de rendimiento valen completamente la pena. Además, me motiva a seguir explorando herramientas más avanzadas para proyectos más complejos.