Instituto Tecnológico y de Estudios Superiores de Monterrey

Escuela de Ingeniería y Ciencias

Maestría en Inteligencia Artificial Aplicada

Dr. Eduardo Antonio Cendejas Castro

Profesora Yetnalezi Quintas Ruiz

Cómputo en la nube

Tarea 1. Programación de una solución paralela

Francisco Antonio Enríquez Cabrera – A01795006

26-01-2025

Introducción

La programación paralela permite dividir un problema en tareas independientes que se ejecutan simultáneamente en múltiples núcleos o procesadores. Este enfoque aprovecha la capacidad de las arquitecturas multicore modernas para reducir tiempos de ejecución y mejorar la eficiencia del sistema (Quinn, 2004).

En C++, OpenMP es una herramienta ampliamente utilizada para implementar paralelismo de manera eficiente. Proporciona directivas de alto nivel que simplifican la distribución de tareas entre hilos sin necesidad de gestionar manualmente los detalles complejos de creación, sincronización y destrucción de hilos (Chapman, Jost, & Pas, 2007).

El proyecto presentado consiste en la suma de dos arreglos utilizando programación paralela con OpenMP. Los arreglos se llenan con valores generados aleatoriamente y se procesan en paralelo utilizando dos hilos. Este ejemplo destaca la combinación de flexibilidad, reproducibilidad y control sobre la ejecución paralela, demostrando los beneficios del paralelismo.

Liga del repositorio de GitHub

El código fuente, el ejecutable y la documentación se encuentran en el siguiente repositorio:

https://github.com/FranciscoEnriquez/openmpA01795006

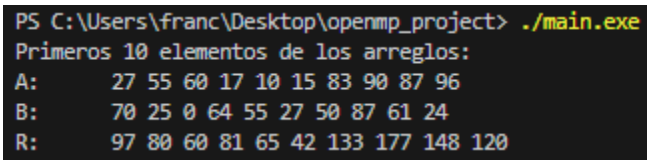
En dicho repositorio se incluyen:

- El archivo main.cpp con el código fuente principal.
- El archivo main.exe (o la configuración para generarlo en local).
- La configuración de OpenMP.
- Esta documentación.

Capturas de pantalla de dos ejecuciones del proyecto

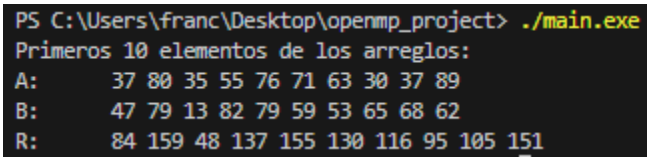
A continuación, se presentan dos ejecuciones distintas del programa, mostrando los primeros 10 elementos de cada arreglo (A, B y R):

Ejecución 1



En esta salida, los valores generados para A y B son aleatorios, y R contiene la suma correspondiente de cada elemento.

Ejecución 2



En esta segunda ejecución, los valores en los arreglos A, B y R son diferentes debido a la semilla aleatoria basada en el tiempo del sistema. Esto demuestra la flexibilidad del programa para generar nuevos valores en cada corrida.

Explicación del código y los resultados

main.cpp > ...

```cpp
#include <iostream>   // Para std::cout
#include <vector>     // Para std::vector
#include <cstdlib>    // Para rand(), srand()
#include <ctime>      // Para time()
#include <omp.h>      // Para programación paralela con OpenMP

constexpr std::size_t SIZE = 1000;  // Tamaño de los arreglos
constexpr std::size_t CHUNK = 100; // Tamaño de cada pedazo (chunk) para OpenMP
constexpr std::size_t SHOW = 10;   // Número de elementos a imprimir

/**
 * @brief Imprime los primeros elementos de un arreglo.
 *
 * @param arr Arreglo a imprimir (std::vector<float>).
 * @param label Etiqueta descriptiva del arreglo (ej. "A").
 */
void imprimeArreglo(const std::vector<float>& arr, const std::string& label) {
    std::cout << label << ":\t";
    for (std::size_t i = 0; i < SHOW; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    // Declaración de los arreglos como std::vector para seguridad y flexibilidad
    std::vector<float> A(SIZE), B(SIZE), R(SIZE);

    // Inicializar el generador de números aleatorios con una semilla variable
    std::srand(static_cast<unsigned int>(std::time(nullptr)));

    // Llenar los arreglos A y B con valores aleatorios entre 0 y 99
    for (std::size_t i = 0; i < SIZE; i++) {
        A[i] = static_cast<float>(std::rand() % 100); // Convertir a float
        B[i] = static_cast<float>(std::rand() % 100);
    }

    /**
     * @brief Suma paralela de los arreglos A y B en C.
     *
     * Directiva OpenMP:
     * - `parallel for`: Divide el bucle entre hilos.
     * - `num_threads(2)`: Usa exactamente 2 hilos.
     * - `schedule(static, CHUNK)`: Divide las iteraciones en bloques de tamaño CHUNK.
     * - `default(none)`: Obliga a declarar explícitamente las variables.
     * - `shared`: Variables compartidas entre hilos.
     */
    #pragma omp parallel for num_threads(2) schedule(static, CHUNK) \
    default(none) shared(A, B, R)
    for (std::size_t i = 0; i < SIZE; i++) {
        R[i] = A[i] + B[i];
    }

    // Imprimir los primeros 10 elementos de cada arreglo
    std::cout << "Primeros " << SHOW << " elementos de los arreglos:\n";
    imprimeArreglo(A, "A");
    imprimeArreglo(B, "B");
    imprimeArreglo(R, "R");

    return 0; // Indica que la ejecución fue exitosa
}
```

Bibliotecas y definición de constantes

El código utiliza:

- <iostream> y <vector>: Para manejo moderno de entrada/salida y estructuras dinámicas en C++.
- <cstdlib> y <ctime>: Para generar valores aleatorios controlados mediante la función std::srand().
- <omp.h>: Biblioteca específica de OpenMP para paralelizar bucles y manejar hilos.

Constantes:

- SIZE: Define el tamaño de los arreglos.
- CHUNK: Controla el tamaño de cada bloque procesado por cada hilo.
- SHOW: Número de elementos a imprimir de cada arreglo.

Generación de valores aleatorios

```
std::srand(static_cast<unsigned int>(std::time(nullptr)));

for (std::size_t i = 0; i < SIZE; i++) {

   A[i] = static_cast<float>(std::rand() % 100);

   B[i] = static_cast<float>(std::rand() % 100);

}
```

- std::srand() inicializa el generador de números aleatorios con una semilla basada en el tiempo del sistema (std::time(nullptr)).
- Cada valor se genera entre 0 y 99, asegurando una distribución uniforme.

Suma paralela de arreglos con OpenMP

```
#pragma omp parallel for num_threads(2) schedule(static, CHUNK) \

default(none) shared(A, B, R)
```

```
for (std::size_t i = 0; i < SIZE; i++) {
    R[i] = A[i] + B[i];
}
```

- num_threads(2): Utiliza exactamente 2 hilos para dividir el trabajo.
- schedule(static, CHUNK): Divide las iteraciones en bloques de tamaño CHUNK para garantizar una carga balanceada.
- default(none): Obliga a declarar explícitamente qué variables son compartidas (shared) o privadas (private), mejorando la seguridad del código.

Impresión de resultados

```
void imprimeArreglo(const std::vector<float>& arr, const std::string& label) {
    std::cout << label << ":\t";
    for (std::size_t i = 0; i < SHOW; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

- La función imprimeArreglo es reutilizable y mejora la legibilidad del programa principal.

Conclusión de los resultados

- Eficiencia: El uso de dos hilos permite realizar la suma más rápido en sistemas multicore.
- Flexibilidad: Los valores generados aleatoriamente muestran la adaptabilidad del programa para diferentes escenarios.

- Control: La división en bloques y el uso de OpenMP permiten ajustar el comportamiento de la paralelización.

Reflexión sobre la programación paralela

La programación paralela, como se implementa con OpenMP, ofrece un enfoque eficaz para optimizar aplicaciones con alta demanda computacional (Grama, Gupta, Karypis, & Kumar, 2003). Las ventajas incluyen:

- Reducción del tiempo de ejecución: Dividir tareas entre hilos mejora el rendimiento, especialmente en sistemas con múltiples núcleos (Chapman, Jost, & Pas, 2007).
- Escalabilidad: La capacidad de ajustar el número de hilos o el tamaño de los bloques asegura un uso eficiente del hardware.

Sin embargo, también presenta desafíos:

- Condiciones de carrera: Es crucial gestionar las variables compartidas para evitar resultados inconsistentes.
- Sobrecarga de sincronización: La coordinación entre hilos puede introducir costos adicionales.
- Análisis detallado: Determinar qué partes del código se benefician del paralelismo es fundamental para evitar esfuerzos innecesarios.

A pesar de estos retos, el paralelismo es un pilar esencial en la computación moderna, donde el rendimiento ya no depende únicamente de la frecuencia del procesador, sino de la capacidad de aprovechar múltiples núcleos de manera eficiente (Sterling, Anderson, & Brodowicz, 2018).

Conclusión

Este proyecto demuestra cómo la programación paralela, implementada mediante OpenMP, puede optimizar tareas computacionales como la suma de dos arreglos. Utilizando dos hilos y una planificación estática con bloques (schedule(static, CHUNK)), se logra un reparto balanceado de las iteraciones, lo que permite un aprovechamiento eficiente de los recursos del hardware multicore.

El uso de std::vector en lugar de arreglos estáticos brinda flexibilidad y seguridad en la gestión de memoria, mientras que la generación de valores aleatorios asegura versatilidad en las pruebas, manteniendo la posibilidad de reproducir los resultados mediante el ajuste de la semilla. Adicionalmente, la introducción de constantes (SIZE, CHUNK, SHOW) y funciones auxiliares como imprimeArreglo contribuyen a mejorar la claridad y la modularidad del código.

En conclusión, este proyecto ilustra cómo una combinación adecuada de paralelismo, control sobre los hilos y diseño estructurado puede abordar problemas computacionales con mayor eficiencia. La programación paralela es un componente esencial en la computación moderna y sigue siendo una herramienta indispensable en la resolución de problemas científicos, tecnológicos e industriales.

Referencias

Chapman, B., Jost, G., & Pas, R. (2007). Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press.

Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). Introduction to Parallel Computing (2nd ed.). Addison-Wesley.

Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. McGraw-Hill.

Sterling, T. L., Anderson, M., & Brodowicz, M. (2018). *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann.