Memoria:
Práctica final

Sira González-Madroño y Miriam Blanco Ponce
Arquitectura de Computadores

Índice

Detalles técnicos del entorno

- **Lenguaje:** C (estándar 11).
- **Sistema Operativo:** macOS 15 (Apple Silicon - ARM64).
- **Entorno de desarrollo:** CLion + CMake.
- **Compilador:** Clang.
- **Paralelismo:** OpenMP habilitado con `libomp`.
- **Gestor de paquetes:** Homebrew.

⚠️ El enunciado pedía compilación con `gcc - fopenmp`, pero por incompatibilidad entre GCC y los SDK de Apple (problemas con `.tbd` y `ld`), hemos optado por compilar con **Clang + libomp**, que configuramos correctamente en `CMakeLists.txt`.

Introducción

Link al repositorio: https://github.com/mbp4/proyectoFinal.git

En este trabajo se nos solicitó realizar una simulación de la difusión del calor sobre una placa metálica bidimensional utilizando dos enfoques: uno secuencial y otro paralelo.

Para realizar el programa usamos el lenguaje de programación C ya que es un lenguaje compatible con el ejercicio y con el que ya estamos familiarizadas, además de esto se hizo uso de OpenMP.

OpenMP es una API para la programación multiproceso de memoria compartida y permite la concurrencia en programas en C o C++.

La simulación modela una placa como una matriz N x N, donde se propaga el calor desde los bordes hacia el interior, y se ejecuta hasta alcanzar un umbral de estabilidad térmica o un número máximo de iteraciones.

Resolución

La solución de este ejercicio se divide en dos partes: una implementación secuencial y una implementación paralela con uso de OpenMP.

A continuación queremos dejar una captura del resultado que hemos conseguido con cada una de las implementaciones, para más adelante explicar con un poco más de detalle como funciona cada una de ellas.

En el comando de ejecución vemos `./cmake-build-debug/proyectoFinal 50 2.0 1000`, esto quiere decir que estamos otorgando los siguientes valores a nuestra simulación:

- N = 50
- Umbral = 2.0
- Iteraciones = 1000

```
(base) Ordenador-Sira:proyectoFinal siragonzalez-madrono$ ./cmake-build-debug/proyectoFinal 50 2.0 1000
Ejecutando simulación secuencial...
Secuencial: iteraciones = 18, tiempo = 0.0013 s
Ejecutando simulación paralela con OpenMP...
Paralela: iteraciones = 18, tiempo = 0.0049 s

Prueba 1: (2 * N, umbral, 2 * iteraciones)
Paralela: iteraciones = 18, tiempo = 0.0030 s

Prueba 2: (N^2, 2 * umbral, 2 * iteraciones)
Paralela: iteraciones = 10, tiempo = 0.2221 s
```

Parte 1: Implementación Secuencial

En primer lugar se implementó una base de la simulación sin paralelismo. La lógica que hicimos es la siguiente:

- ➔ Se crean dos matrices dinámicas: **actual** (estado actual de temperaturas) y **siguiente** (estado siguiente).
- ➔ Se inicializan los **bordes** de la matriz a *100ºC* (simulando una fuente de calor constante) y el **interior** a *0ºC*.
- ➔ En cada iteración, se actualiza cada celda interior con el promedio de sus 4 vecinas (arriba, abajo, izquierda y derecha).
- ➔ Se calcula el **cambio máximo** entre valores antiguos y nuevos para decidir si continuar o detenerse.
- ➔ Se usa la función `omp_get_wtime()` para medir el tiempo de ejecución (aunque sin usar paralelismo aún).
- ➔ Se copia el contenido de la matriz **siguiente** a **actual** para la próxima iteración.

Estructura del código:

- `main.c` → recibe los argumentos (N, umbral, iteraciones).
- `utilidades.c/h` → funciones auxiliares para crear, inicializar y liberar matrices.
- `difusión_secuencial.c` → simulación base, sin paralelismo.

Parte 2: Implementación Paralela con OpenMP

Para realizar la segunda parte de la simulación donde se pide una implementación paralela haciendo uso de OpenMP seguimos la siguiente lógica:

- ➔ Mantenemos las matrices dinámicas anteriores donde se guarda el estado actual (matriz **actual**) y el siguiente (matriz **siguiente**).
- ➔ De nuevo inicializamos los bordes a 100ºC y el interior a 0ºC, consiguiendo así una simulación de calor en la placa.
- ➔ Haciendo uso de `#pragma omp parallel for` se actualizará cada celda interior con el promedio de su placas vecinas (arriba, abajo, izquierda y derecha). Al hacer uso de esto logramos paralelizar el bucle.
- ➔ Con `reduction(max:cambio_max)` calculamos el cambio máximo entre la temperatura antigua y la nueva, de nuevo se calcula en un bucle paralelo para poder obtener el valor máximo de una manera segura entre los distintos hilos.
- ➔ Con `omp_get_wtime()` medimos el tiempo de ejecución y lo podemos comparar con el rendimiento de la primera parte.
- ➔ Por último se intercambian los punteros entre **actual** y **siguiente**, esto se hace sin copiar los datos por lo que mejoramos la eficiencia.

Estructura del código:

- `main.c` → recibe los argumentos (N, umbral, iteraciones).
- `utilidades.c/h` → funciones auxiliares para crear, inicializar y liberar matrices.
- `difusión_paralela.c` → simulación haciendo uso de la paralelización con OpenMP.

Conclusión

La práctica nos ha permitido comprender cómo una simulación sencilla basada en mallas, como la que se nos pedía programar, puede beneficiarse significativamente del paralelismo mediante el uso de OpenMP.

- ★ El paralelismo con OpenMP es simple de implementar pero muy poderoso para operaciones matriciales.
- ★ El uso de `reduction` y `collapse` mejora la eficiencia y evita errores comunes de concurrencia.
- ★ A partir de cierto tamaño de matriz, la versión paralela supera claramente la secuencial.
- ★ La transición entre GCC y Clang en macOS ha sido una barrera técnica importante que resolvimos correctamente y de forma eficiente utilizando `libomp` y ajustes en `CMakeLists.txt`.
- ★ El diseño modular del código (división en archivos independientes) facilita el mantenimiento y la extensión futura.

En resumen, hemos conseguido cumplir con los objetivos del enunciado, incluyendo pruebas de escalabilidad, comparación de tiempos y uso correcto de herramientas de programación paralela, todo esto a pesar de haber tenido dificultades técnicas por la incompatibilidad de GCC con nuestro sistema operativo.

Bibliografía

https://es.wikipedia.org/wiki/OpenMP