Documentacion_paralelo

1. Propósito y Objetivos

Esta versión del simulador está optimizada para el rendimiento en sistemas modernos con múltiples núcleos de CPU. El objetivo principal es superar el cuello de botella computacional inherente a la versión secuencial, cuya complejidad es de **$O(n^2)$**, para permitir la simulación de sistemas más grandes o durante períodos de tiempo más largos de manera eficiente.

Los objetivos específicos de esta implementación son:

- **Reducir el tiempo de ejecución:** Distribuir el trabajo más costoso entre múltiples hilos de ejecución.
- **Asegurar la correctitud:** Garantizar que los resultados físicos (energías, trayectorias) sean idénticos a los de la versión secuencial.
- **Evaluar la escalabilidad:** Medir cómo mejora el rendimiento (speed-up) a medida que se añaden más núcleos al cómputo.

2. Tecnología Utilizada: OpenMP

Para lograr el paralelismo, se utiliza **OpenMP (Open Multi-Processing)**. Esta es una API estándar de la industria para la programación paralela en arquitecturas de memoria compartida (la mayoría de las computadoras y portátiles modernos).

¿Por qué OpenMP?

- **Simplicidad:** Permite paralelizar código existente de forma incremental utilizando **directivas** (`#pragma`), que son instrucciones especiales para el compilador sin alterar drásticamente la estructura del código.
- **Portabilidad:** Es un estándar soportado por la mayoría de los compiladores de C/C++ modernos (GCC, Clang, MSVC), lo que hace que el código sea portable entre diferentes sistemas operativos.
- **Eficiencia:** Está diseñado para tener una sobrecarga (overhead) baja, siendo muy eficiente para paralelizar bucles computacionalmente intensivos.

3. Arquitectura y Estrategia de Paralelización

La estrategia se centra en el **paralelismo de datos** aplicado a la función más costosa del programa: `calculate_forces_and_potential_parallel`.

- **Identificación del Cuello de Botella:** El doble bucle anidado que calcula las interacciones entre todos los pares de partículas es, con diferencia, la parte más lenta del programa. El número de cálculos crece cuadráticamente con el número de partículas.
- **Modelo de Paralelismo de Datos:** La estrategia consiste en dividir las iteraciones del bucle `for` **exterior** (`for (int i = 0; i < N_PARTICLES; i++)`) entre un equipo de hilos (threads).
  - **Ejemplo:** Si hay 1000 partículas y 4 hilos, OpenMP puede asignar las iteraciones `i = 0` a `249` al hilo 0, `i = 250` a `499` al hilo 1, y así sucesivamente.
  - Cada hilo calcula de forma independiente las fuerzas para su subconjunto de partículas `i`, trabajando en paralelo.
- **Gestión de Condiciones de Carrera (Race Conditions):** Cuando varios hilos escriben en la misma memoria compartida, pueden ocurrir conflictos. Este código enfrenta dos condiciones de carrera principales:
  1. **Acumulación de `potential_energy`**: Varios hilos suman a esta variable global.
  2. **Actualización de `particles[j].force`**: Múltiples hilos pueden intentar actualizar el vector de fuerza de la misma partícula `j` al mismo tiempo.
     Estas se solucionan con directivas específicas de OpenMP, como se detalla a continuación.

4. Desglose de la Implementación

La paralelización se logra con tres directivas clave dentro de `calculate_forces_and_potential_parallel`:

- **`#pragma omp parallel for`**
  - Esta es la directiva principal que combina dos conceptos:
    - `parallel`: Crea un equipo de hilos.
    - `for`: Le dice a esos hilos que se repartan las iteraciones del siguiente bucle `for`.
  - Se coloca justo antes del bucle `for` exterior para dividir el trabajo principal.

- `reduction(+:potential_energy)`
  - Esta cláusula se añade a la directiva `parallel for` para solucionar de forma segura y eficiente la condición de carrera en `potential_energy`.
  - **Cómo funciona:** OpenMP crea una copia privada de `potential_energy` para cada hilo, inicializada en cero. Cada hilo suma sus resultados locales a su copia privada. Al finalizar la región paralela, OpenMP realiza una operación de "reducción", sumando los valores de todas las copias privadas en la variable global original.
- `#pragma omp atomic`
  - Esta directiva soluciona la condición de carrera más compleja: la actualización de las fuerzas en las partículas.
  - Cuando un hilo que procesa la partícula `i` calcula su interacción con `j`, debe actualizar tanto `particles[i].force` como `particles[j].force`. El problema es que otro hilo podría estar actualizando `particles[j].force` simultáneamente.
  - `atomic` garantiza que una operación simple de memoria (como `x += valor`) se ejecute como una unidad indivisible, sin que pueda ser interrumpida. Es una forma de bajo costo para proteger estas actualizaciones críticas, asegurando que todas las contribuciones a la fuerza se sumen correctamente sin corromper los datos.

5. Compilación y Medición de Rendimiento

- **Compilación:** Es indispensable incluir la bandera `fopenmp` al compilar con `gcc`. Esto le indica al compilador que reconozca y procese las directivas `#pragma omp`.
- **Ejecución y Pruebas:** El número de hilos se controla en tiempo de ejecución a través de la variable de entorno `OMP_NUM_THREADS`. Esto permite realizar pruebas de rendimiento de manera flexible para evaluar la escalabilidad del código. El rendimiento se mide con:
  - **Speed-up:** Sp=T1/Tp (Tiempo de ejecución secuencial / Tiempo de ejecución con p hilos).
  - **Eficiencia:** Ep=Sp/p (Qué tan bien se aprovechan los núcleos).