Documentacion_secuencial

Propósito

Esta versión sirve como la implementación base y de referencia del simulador. Su propósito es validar la corrección de la física y la lógica del algoritmo de Velocity Verlet. Computacionalmente, su rendimiento está limitado por una complejidad de **$O(n^2)$** debido al cálculo de fuerzas de "fuerza bruta". Sirve como el punto de comparación (T1) para medir el rendimiento de la versión paralela.

Estructura General

El programa está diseñado para ser flexible y modular, separando la configuración, la lógica de simulación y los cálculos físicos.

- **Entrada:** El programa se ejecuta desde la línea de comandos y requiere una única entrada: la ruta a un archivo de configuración (`.txt`).
- **Configuración Dinámica:** Todos los parámetros de la simulación (número de partículas, tamaño de la caja, constantes físicas, etc.) se cargan en tiempo de ejecución desde el archivo de configuración, lo que permite realizar múltiples experimentos sin necesidad de recompilar el código.
- **Flujo Principal:** El `main` orquesta todo el proceso: carga la configuración, inicializa el sistema, ejecuta el bucle de simulación principal y libera los recursos al final.

Estructuras de Datos Clave

- `struct Vector` : Una estructura simple para almacenar coordenadas 3D (x, y, z), utilizada para posiciones, velocidades y fuerzas.
- `struct Particle` : Agrupa toda la información de una partícula: un identificador (`id`), y tres `Vector` para su posición, velocidad y la fuerza neta que actúa sobre ella.
- `struct Config` : Almacena todos los parámetros de la simulación leídos del archivo. Se pasa por referencia a casi todas las funciones, centralizando la configuración.

Desglose de Funciones Principales

- **`main(int argc, char *argv[])`**
  - Punto de entrada del programa.
  - Valida los argumentos de la línea de comandos.
  - Orquesta el flujo: llama a `load_config`, `initialize_particles_lattice`, y luego entra en el bucle de simulación.
  - Dentro del bucle, implementa el algoritmo de Velocity Verlet en el orden correcto.
  - Gestiona la impresión periódica de datos y la liberación de memoria.
- **`load_config(Config *config, const char* filename)`**
  - Abre y lee el archivo de configuración especificado.
  - Parsea cada línea para extraer pares clave-valor (ej. `N_PARTICLES 512`).
  - Puebla la estructura `Config` con los valores leídos.
- **`initialize_particles_lattice(Particle particles[], const Config* config)`**
  - Coloca las partículas en una red cúbica simple y uniforme dentro de la caja de simulación.
  - Este método es preferible a una inicialización aleatoria porque previene que las partículas comiencen demasiado cerca, lo que podría causar fuerzas de repulsión extremadamente grandes y hacer "explotar" la simulación.
- **`calculate_forces_and_potential(Particle particles[], const Config* config)`**
  - Es el corazón computacional del programa.
  - Utiliza un doble bucle anidado para iterar sobre todos los pares únicos de partículas (`for i... for j = i + 1...`).
  - Para cada par, calcula la distancia aplicando la **convención de la imagen mínima**.
  - Si la distancia está dentro del radio de corte, calcula la fuerza y la energía potencial de Lennard-Jones y las acumula.
  - La complejidad de esta función es O(n2).

- `update_positions(...)` y `update_velocities(...)`
  - Implementan las dos mitades del integrador de **Velocity Verlet**.
  - `update_positions` usa las fuerzas del tiempo t para calcular las nuevas posiciones en t+Δt.
  - `update_velocities` usa el promedio de las fuerzas en t y t+Δt para calcular las nuevas velocidades.

2. Documentación del Código Paralelo (`simulador_paralelo.c`)

Propósito

Esta versión del simulador está optimizada para el rendimiento en sistemas modernos de múltiples núcleos. El objetivo es reducir drásticamente el tiempo de ejecución del cálculo de fuerzas, que es el cuello de botella de la versión secuencial, y así permitir simulaciones más grandes o más largas.

Tecnología Utilizada: OpenMP

Se utiliza **OpenMP (Open Multi-Processing)**, una API estándar para la programación de memoria compartida. Permite paralelizar el código de forma incremental y portable mediante directivas `#pragma`, que son instrucciones para el compilador.

Estrategia de Paralelización

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

- **Identificación del Trabajo:** El trabajo a dividir son las iteraciones del bucle `for` exterior (`for (int i = 0; i < config->N_PARTICLES; i++)`).
- **División:** El equipo de hilos (threads) de OpenMP se reparte estas iteraciones. Por ejemplo, con 4 hilos y 1000 partículas, el hilo 0 podría tomar `i` de 0 a 249, el hilo 1 de 250 a 499, y así sucesivamente.

Implementación y Directivas OpenMP

Se introducen directivas clave en `calculate_forces_and_potential_parallel` para gestionar el paralelismo y evitar errores:

- **`#pragma omp parallel for`**
  - Es la directiva principal que le dice a OpenMP: "Toma el siguiente bucle `for` y distribuye sus iteraciones entre los hilos disponibles".
- **`reduction(+:potential_energy)`**
  - Soluciona una **condición de carrera** (race condition) en la variable `potential_energy`. Sin esta cláusula, todos los hilos intentarían sumar a la misma variable al mismo tiempo, corrompiendo el resultado.
  - `reduction` crea una copia privada de `potential_energy` para cada hilo. Cada hilo acumula sus resultados en su copia local. Al final de la región paralela, OpenMP suma (reduce) los valores de todas las copias privadas en la variable global original de forma segura.
- **`#pragma omp atomic`**
  - Soluciona la condición de carrera más crítica: la actualización de los vectores de fuerza de las partículas (`particles[i].force` y `particles[j].force`).
  - Por ejemplo, el hilo 0 podría estar calculando la interacción (i, j) y necesita actualizar `particles[j].force`. Simultáneamente, el hilo 1 podría estar calculando la interacción (k, j) y también necesita actualizar `particles[j].force`.
  - La directiva `atomic` garantiza que una operación de memoria simple (como `+=`) se ejecute como una unidad indivisible, evitando que los hilos se "pisen" entre sí y asegurando que todas las contribuciones a la fuerza se sumen correctamente.