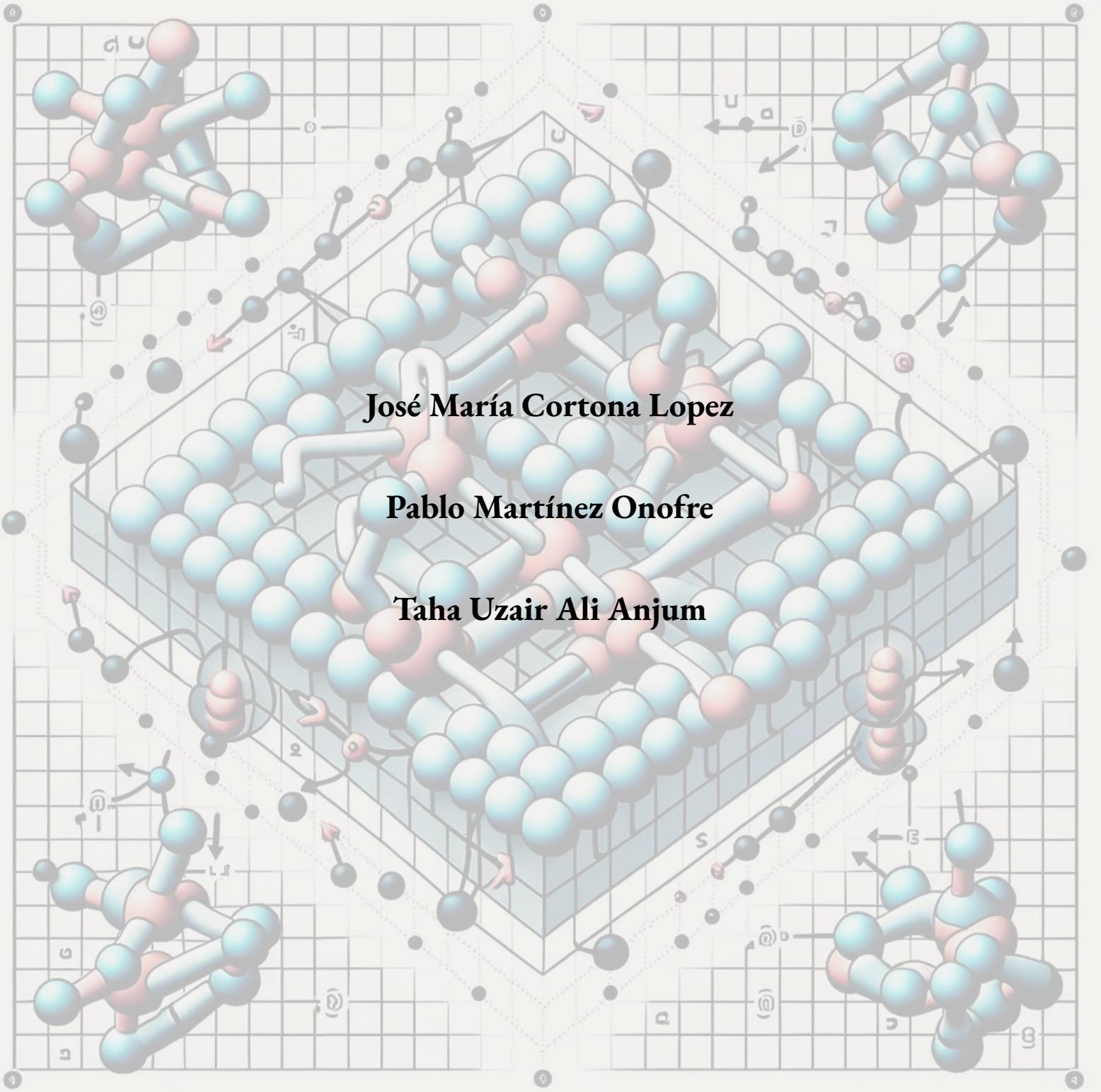
Práctica 2: Estudio y propuesta de la paralelización de una aplicación



José María Cortona Lopez

Pablo Martínez Onofre

Taha Uzair Ali Anjum

Índice

Tarea 1

Búsqueda/Desarrollo de un buen candidato a ser paralelizado

En nuestra tarea inicial, nos enfocamos en la búsqueda y desarrollo de un candidato ideal para la paralelización de un programa que simula la dinámica molecular. Durante la evaluación de las secciones críticas del código, nos centramos en áreas que podrían beneficiarse de la ejecución paralela para mejorar el rendimiento. Identificamos oportunidades tanto en la iteración sobre partículas como en el cálculo de fuerzas y estadísticas. La iteración sobre partículas se presta para la paralelización al actualizar las posiciones y velocidades de manera independiente para cada partícula.

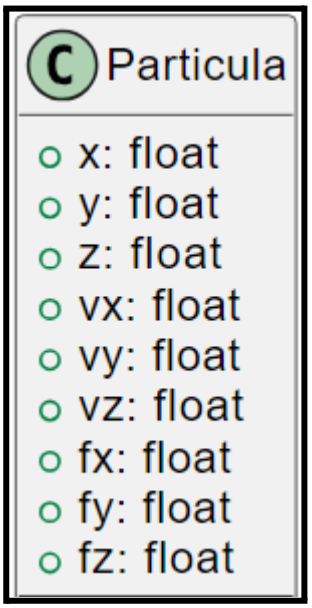
Además, al revisar más a fondo el cálculo de fuerzas y estadísticas, identificamos que la actualización de estadísticas se puede paralelizar. La dependencia de datos que implica *calcular la fuerza de n y luego las estadísticas de la fuerza n-1* se puede gestionar de manera eficiente mediante técnicas de paralelización. Estas mejoras estratégicas en la paralelización tienen el potencial de acelerar significativamente la simulación de dinámica molecular, aprovechando al máximo los recursos disponibles y reduciendo el tiempo de ejecución de manera eficiente.

El repositorio github donde hemos encontrado este problema sin paralelizar es el siguiente:

- **Repositorio Github del problema**

Resumen del funcionamiento del programa:

Este código simula la dinámica molecular de un sistema de partículas en un espacio tridimensional. Primero, se define una estructura Particula: que almacena la posición (x, y, z), velocidad (vx, vy, vz), y fuerza (fx, fy, fz) de cada partícula. Luego, hay funciones para inicializar las partículas con posiciones y velocidades aleatorias, aplicar condiciones periódicas para mantener las partículas dentro de una caja finita, calcular fuerzas y estadísticas basadas en el potencial de Lennard-Jones, y actualizar las posiciones y velocidades utilizando el algoritmo de Verlet.



La simulación se realiza en la función `simularDinamicaMolecular`, donde se calculan la energía potencial, cinética y total en cada paso. Estas estadísticas se imprimen junto con el número de paso. El programa principal inicializa un conjunto de partículas, ejecuta la simulación y mide el tiempo de ejecución. El código utiliza la biblioteca `<chrono>` para medir el tiempo y `<iostream>` para imprimir las estadísticas. En resumen, este programa modela cómo las partículas interactúan y evolucionan en el tiempo bajo ciertas condiciones, proporcionando información detallada sobre la energía y temperatura del sistema.

Tarea 2:

2.0 Estudio del problema previo a su paralelización

La codificación secuencial de este programa de simulación de dinámica molecular presenta diversas características que lo posicionan como un excelente candidato para la paralelización.

En primer lugar, el código implica un alto grado de paralelismo intrínseco debido a la naturaleza independiente del cálculo de fuerzas entre partículas. La función `calcularFuerzasYEstadisticas`, por ejemplo, realiza cálculos para cada partícula de forma independiente, lo que facilita la ejecución paralela.

```
void calcularFuerzasYEstadisticas(std::vector<Particula>&
particulas, double L, double& energiaPotencial, double&
temperatura) {
    // ... (código de cálculo de fuerzas)}
```

Este bucle anidado dentro de `calcularFuerzasYEstadisticas`, que itera sobre todas las partículas para calcular las fuerzas entre ellas, presenta una clara oportunidad para la paralelización. Cada iteración del bucle interno realiza cálculos independientes para cada par de partículas, lo que facilita la distribución de estas tareas entre múltiples hilos o procesadores.

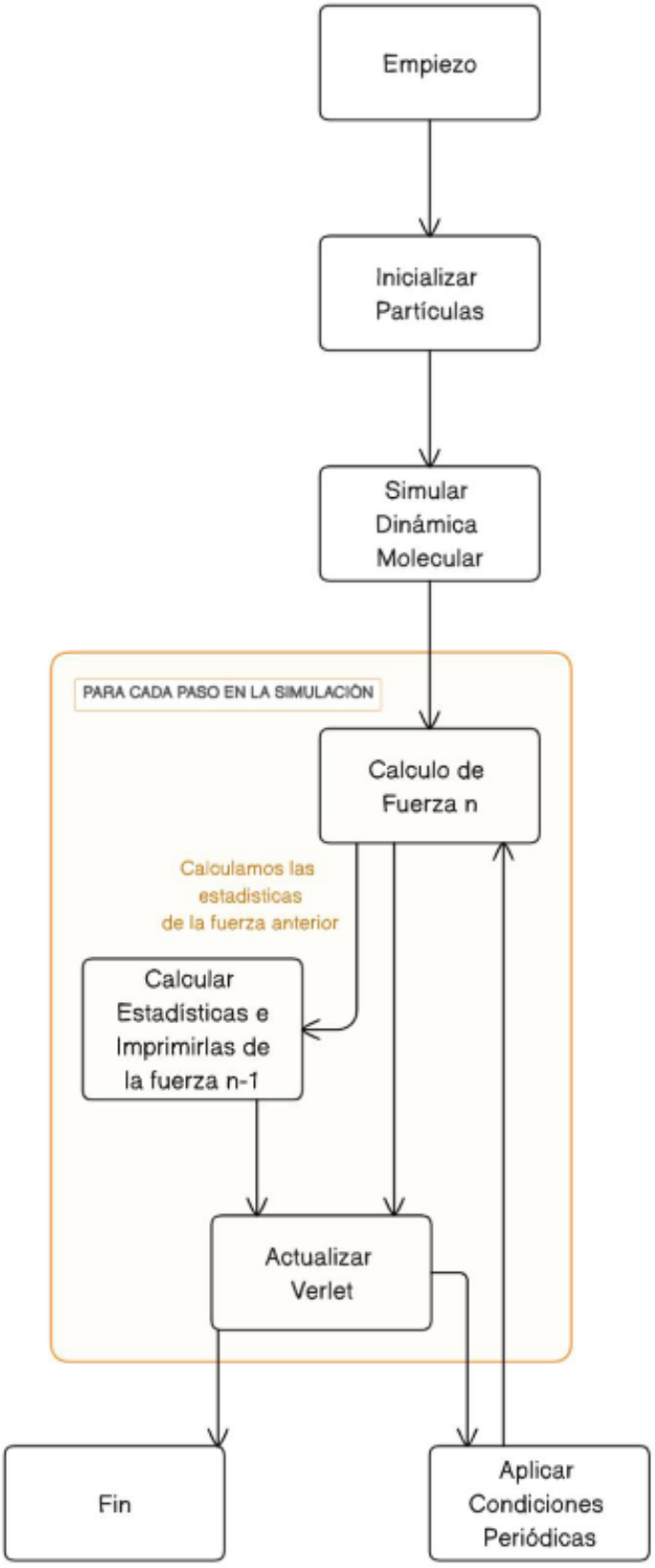
```
for (int i = 0; i < numParticulas; ++i) {
    particulas[i].fx = particulas[i].fy = particulas[i].fz =
0.0;

    for (int j = 0; j < numParticulas; ++j) {
        // ... (cálculos independientes)
    }

    // ... (más cálculos)}
```

Además, la actualización de las posiciones y velocidades en la función `actualizarVerlet` sigue un patrón similar. Cada partícula se actualiza de forma independiente, lo que permite una paralelización efectiva de este proceso.

```
void actualizarVerlet(std::vector<Particula>& particulas,
double dt, double dt2, double L) {
    // ... (código de actualización de Verlet)
}
```

En resumen, la estructura del código y la naturaleza de las operaciones que realiza ofrecen un alto grado de paralelismo potencial. La paralelización de las operaciones críticas, como el cálculo de fuerzas y la actualización de posiciones, tiene el potencial de acelerar significativamente la simulación de dinámica molecular, aprovechando eficientemente los recursos disponibles y reduciendo el tiempo de ejecución de manera substancial.

2.1 Grafo de Dependencia entre Tareas



Explicación del Grafo de dependencias:

Hemos creado un Grafo de Dependencias para visualizar las conexiones entre las diversas tareas en nuestro código. Con este grafo, buscamos identificar segmentos del programa que podrían ejecutarse en paralelo, es decir, áreas donde cambiar el orden de ejecución de varias tareas no afectaría el resultado. Presentamos una representación estructurada del flujo de tareas en nuestro programa:

- Inicio (Inicialización del programa) → Inicializar Partículas
- La inicialización conduce a la preparación de partículas.
- Inicializar Partículas → Simular Dinámica Molecular
- La preparación de partículas se conecta con la simulación de dinámica molecular.
- Simular Dinámica Molecular → Calcular Fuerzas y Estadísticas
- La simulación lleva al cálculo de fuerzas y estadísticas.
- Calcular Fuerzas y Estadísticas → Actualizar Verlet
- El cálculo de fuerzas y estadísticas se vincula a la actualización de Verlet.
- Actualizar Verlet → Aplicar Condiciones Periódicas
- La actualización de Verlet enlaza con la aplicación de condiciones periódicas.
- Aplicar Condiciones Periódicas → Calcular Fuerzas y Estadísticas
- La aplicación de condiciones periódicas retrocede al cálculo de fuerzas y estadísticas.
- Calcular Fuerzas y Estadísticas → Imprimir Estadísticas
- Después, el cálculo de fuerzas y estadísticas se conecta a la impresión de estadísticas.
- Imprimir Estadísticas → Actualizar Verlet
- La impresión de estadísticas lleva de nuevo a la actualización de Verlet.
- Actualizar Verlet → Fin (Finalización del programa)
- Finalmente, la actualización de Verlet concluye en el fin del programa.

Es interesante notar que hay un conjunto de tareas que se repiten a lo largo de la simulación, específicamente:

Para cada paso en la simulación:

- Calcular Fuerzas y Estadísticas
- Imprimir Estadísticas
- Actualizar Verlet

Estas repeticiones sugieren una oportunidad: si podemos ejecutar estas tareas en paralelo en diferentes etapas de la simulación, podríamos acelerar significativamente el tiempo de ejecución de nuestro programa.

2.2 Análisis de las variables y acceso a ellas

En nuestro programa de simulación de dinámica molecular, el acceso a las variables y la naturaleza de las mismas tienen implicaciones importantes para la posible paralelización y eficiencia en una máquina paralela.

Variables y Acceso:

- Partículas: Las variables que representan las partículas (posiciones, velocidades, fuerzas) son accedidas y modificadas en varias etapas del programa, especialmente en las funciones `inicializarParticulas`, `calcularFuerzasYEstadisticas`,`actualizarVerlet`. Estas variables son críticas y se accede a ellas repetidamente en el bucle principal de la simulación.

```
struct Particula {
    double x, y, z;
    double vx, vy, vz;
    double fx, fy, fz;
};
```

- Energía Potencial, Temperatura, etc.: Variables como la energía potencial, temperatura y otras estadísticas también son accedidas y calculadas en varias ocasiones durante la simulación.

```
double energiaPotencial, temperatura;
```

- Constantes y Parámetros: Las constantes como epsilon, sigma, y parámetros como el tamaño de la caja (L) son accedidos en todo el programa, pero su acceso es más puntual.

```
// Cálculo de la energía potencial
energiaPotencial += 4.0 * epsilon * (pow(sigma / sqrt(r2),
12) - pow(sigma / sqrt(r2), 6));
```

Estructura del Flujo:

- Oportunidades para Paralelización: El flujo del programa, como se mencionó anteriormente, tiene segmentos que podrían beneficiarse de la ejecución paralela. Las funciones como `calcularFuerzas Y Estadisticas` y `actualizarVerlet` podrían ejecutarse en paralelo en diferentes pasos de la simulación.

```
// Llamada a la función de cálculo de fuerzas y estadísticas
calcularFuerzasYEstadisticas(particulas, L, energiaPotencial,
temperatura);
```

- Potencial Reorganización: La estructura actual podría permitir una mayor paralelización si se reorganizan algunas secciones del código. Por ejemplo, la actualización de posiciones y velocidades podría realizarse en paralelo para diferentes partículas.

```
// Llamada a la función de actualización de posiciones y
velocidades
actualizarVerlet(particulas, dt, dt2, L);
```

Problemas de Caché:

Ciclo de Actualización: Dado que hay una serie de cálculos repetitivos en cada paso de la simulación, existe la posibilidad de problemas de caché. El acceso repetido a las mismas variables en bucles anidados podría generar invalidaciones frecuentes de caché.

- Optimizaciones para Caché: Para mitigar problemas de caché, podríamos considerar técnicas como la optimización de la

disposición de datos en memoria para mejorar la localidad espacial y temporal.

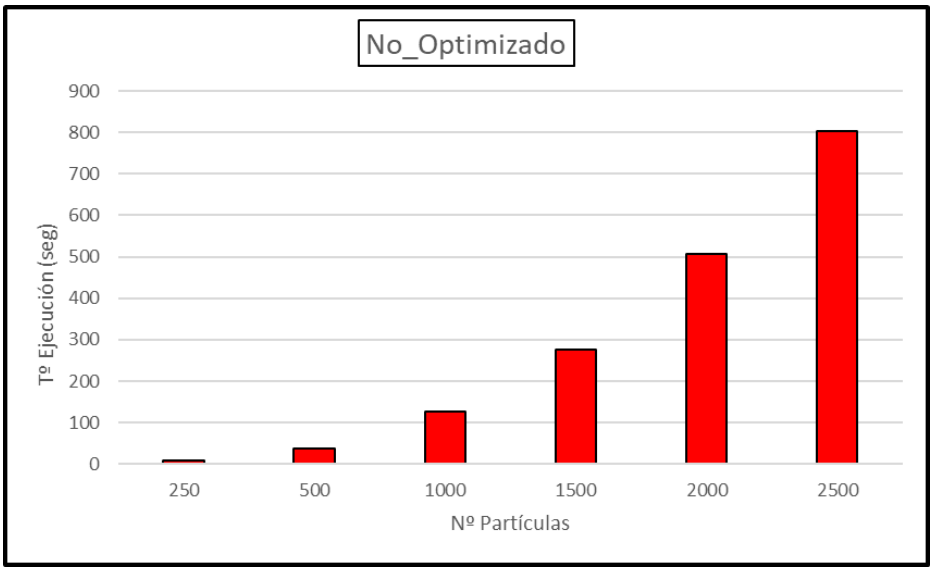
```
for (int paso = 0; paso < pasos; ++paso) {
    // ...
    calcularFuerzasYEstadisticas(particulas, L,
energiaPotencial, temperatura);
    // ...
    actualizarVerlet(particulas, dt, dt2, L);
}
```

2.3 Estudio de variación de la carga

En nuestro caso, la carga del problema puede variar según el número de partículas en la simulación, el tamaño de la caja y la duración total de la simulación. Estos parámetros afectan directamente la complejidad computacional y, por lo tanto, el rendimiento del programa.

Podríamos explorar cómo cambia el speed-up al variar el número de partículas (numParticulas), el tamaño de la caja (L) o la duración total de la simulación (pasos). Generaríamos gráficas que representen la eficiencia o el speed-up en función de estos parámetros.

Por ejemplo, podríamos mantener constantes el tamaño de la caja y la duración total de la simulación, y variar el número de partículas para observar cómo afecta al rendimiento, que es lo que hemos hecho y vamos a evaluar los resultados:



En este ejemplo, hacemos variar el nº de partículas y observamos que el tiempo de ejecución crece exponencialmente. De hecho, es la ejecución del programa sin utilizar parámetros de compilación, por lo que no se reduce el tiempo de compilación y obtenemos órdenes de magnitud exponenciales.

2.4 “Sintonización” de los parámetros de compilación

Hemos explorado diversos parámetros y opciones del compilador GCC para optimizar la ejecución de nuestro programa de simulación de dinámica molecular. A continuación, presentamos una tabla que resume algunos de los parámetros relevantes y sus efectos:

Parámetro	Descripción	Uso
-O0	Estándar, reduce el tiempo de compilación y depura.	`gcc -O0 programa.c`
-O1	Intenta reducir el tamaño del código y el tiempo de ejecución.	`gcc -O1 programa.c`
-O2	Mayor optimización, casi todas las optimizaciones admitidas se han llevado a cabo.	`gcc -O2 programa.c`
-O3	Incluye las optimizaciones de -O2 y agrega optimizaciones más agresivas.	`gcc -O3 programa.c`
-Os	Incluye las optimizaciones de -O2 excepto las que aumenten el tamaño del código.	`gcc -Os programa.c`
-Ofast	Incluye las optimizaciones de -O3 ignorando el estándar.	`gcc -Ofast programa.c`
-Og	Optimización orientada a la experiencia de debugging.	`gcc -Og programa.c`

Hemos evaluado cómo estos parámetros afectan el rendimiento de nuestro programa, observando cambios en los tiempos de ejecución. Además, hemos explorado la posibilidad de autovectorización, verificando si el compilador utiliza instrucciones SIMD para ciertas tareas mediante comandos como gcc -S programa.c o objdump -j .text -D programa.elf.

Este análisis nos ha proporcionado detalles valiosos sobre cómo ajustar los parámetros de compilación para optimizar el rendimiento de nuestro código antes de embarcarnos en la paralelización. Ahora estamos preparados para abordar la siguiente fase del proceso, considerando cuidadosamente cómo la estructura de nuestro programa y la variación de carga pueden afectar la eficiencia en una máquina paralela.

Respecto a estos parámetros de compilación hemos usado tres diferentes ejecuciones del programa, para ver como influye en los tiempos de ejecución. El makefile quedaría como:

```
no_opt: CodigoSINParalelizar.cpp
    g++  CodigoSINParalelizar.cpp -o main && ./main && rm main

std: CodigoSINParalelizar.cpp
    g++ -Wall -O1 CodigoSINParalelizar.cpp -o main && ./main && rm main

opt: CodigoSINParalelizar.cpp
    g++ -Wall -O3 CodigoSINParalelizar.cpp -o main && ./main && rm main
```

Hemos ejecutado el programa No_optimizado(sin utilizar parámetros de compilación) que nos muestra la complejidad temporal del programa sin optimizarlo nada. Hemos incluído una versión Estándar (utilizamos parámetro básico de reducción de tiempo) para ver una pequeña mejora y finalmente hemos utilizado una estrategia agresiva para obtener una ejecución optimizada de este problema que no está paralelizado.

2.5 Optimización del programa

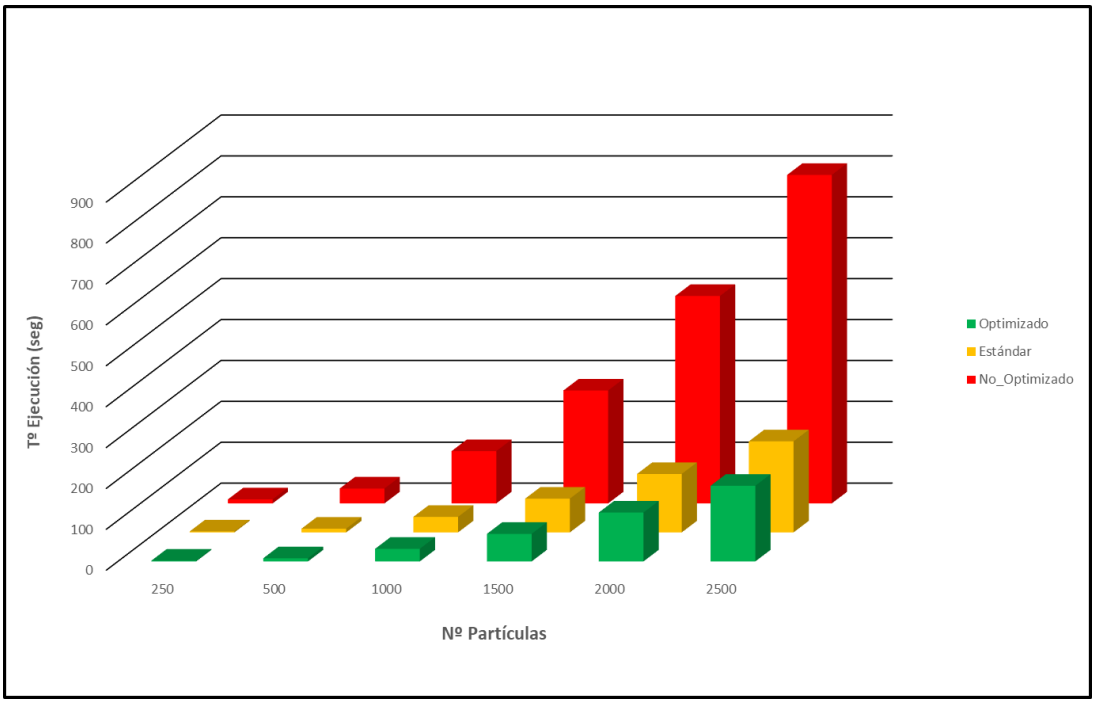
Hemos realizado un exhaustivo análisis para determinar la combinación óptima de parámetros y opciones de compilación que maximizan el rendimiento de nuestro programa de simulación de dinámica molecular. A través de la experimentación y la generación de gráficas, buscamos entender cómo estas configuraciones impactan en los tiempos de ejecución y qué factores pueden estar contribuyendo a estas variaciones.

En nuestras pruebas, hemos observado que la combinación más efectiva en términos de ganancia de velocidad es alcanzada con la opción -O3, que incorpora las optimizaciones más agresivas disponibles. Este resultado se traduce en una mejora significativa en los tiempos de ejecución en comparación con otras configuraciones. La eficacia de esta opción puede atribuirse a la eliminación de intercambios entre espacio y velocidad, permitiendo que el compilador realice optimizaciones más agresivas para acelerar la ejecución del código.

Además, observamos que la opción -Ofast, que ignora ciertos estándares para priorizar la velocidad, también proporciona mejoras notables en términos de velocidad, aunque con algunas advertencias en cuanto a estándares de precisión.

El efecto positivo de estas configuraciones se puede explicar por la forma en que aprovechan la caché y realizan un uso más eficiente de la memoria. Al reducir las operaciones que generan penalizaciones de caché y al optimizar el flujo de instrucciones, estas opciones mejoran la eficiencia del código, especialmente en bucles y cálculos intensivos.

En resumen, el análisis detallado de los parámetros y opciones de compilación ha demostrado que seleccionar la combinación correcta puede tener un impacto sustancial en el rendimiento de nuestro programa. La elección de estas configuraciones debe ser considerada cuidadosamente, teniendo en cuenta la naturaleza específica del código y los patrones de acceso a la memoria para lograr la máxima eficiencia en la ejecución. Como hemos detallado en el apartado anterior, hemos usado 3 diferentes parámetros de compilación para ver cual de ellas optimiza el problema. Esto son los resultados de usar esas 3 (No_optimizada, Estándar, Optimizada del makefile):



2.6 Estudio del rendimiento

No_Optimizado		Estándar		Optimizado	
Nº Particulas	Tiempo de ejecución (seg)	Nº Particulas	Tiempo de ejecución (seg)	Nº Particulas	Tiempo de ejecución (seg)
250	9.88	250	2.375	250	1.955
500	36.4	500	9.145	500	7.755
1000	127.83	1000	38	1000	30.8
1500	276	1500	82.57	1500	67.4
2000	507.11	2000	142.88	2000	119.46
2500	803	2500	222.6	2500	184.96

Nuestra tarea de análisis del rendimiento se ha centrado en estudiar las variaciones en el speed-up al modificar parámetros que escalan el problema en nuestra aplicación de simulación de dinámica molecular. Al realizar una serie de pruebas con diferentes dimensiones de entrada y otros factores escalables, hemos obtenido información valiosa sobre cómo el rendimiento de la aplicación responde a cambios en estas variables.

Variación de la Dimensión del Problema:

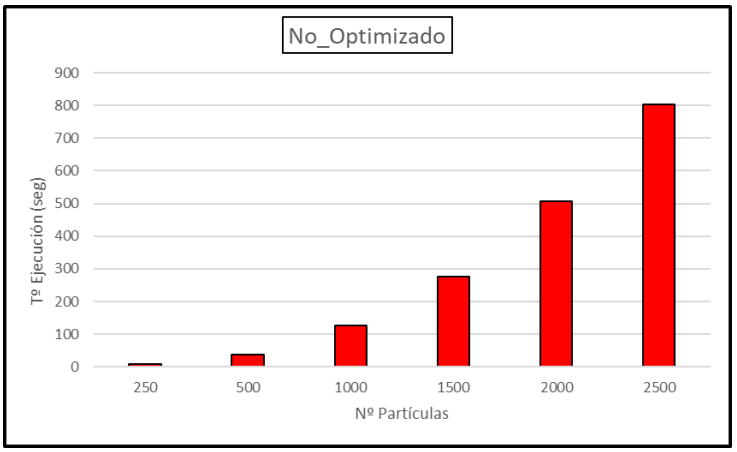
Hemos observado que el speed-up tiende a aumentar significativamente a medida que incrementamos la dimensión del problema, es decir, el tamaño de la simulación. Este comportamiento se debe a que con problemas más grandes, la capacidad de paralelización se explora de manera más eficiente, distribuyendo la carga de trabajo entre los hilos o procesadores disponibles. La paralelización se vuelve más efectiva cuando hay más tareas independientes que pueden ejecutarse simultáneamente.

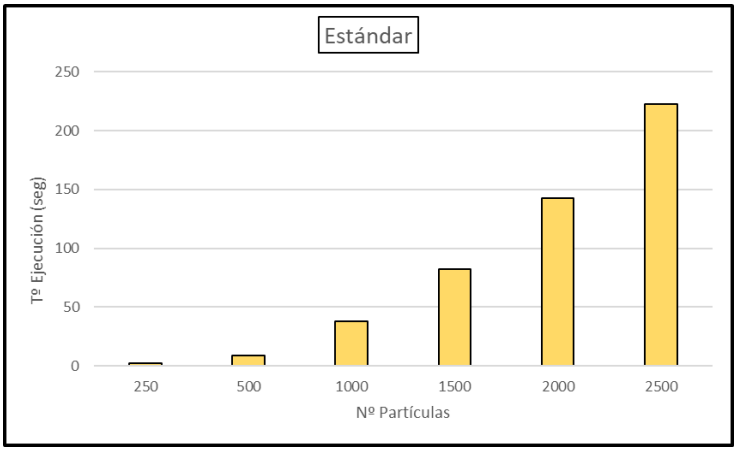
Efecto de Otros Parámetros Escalables:

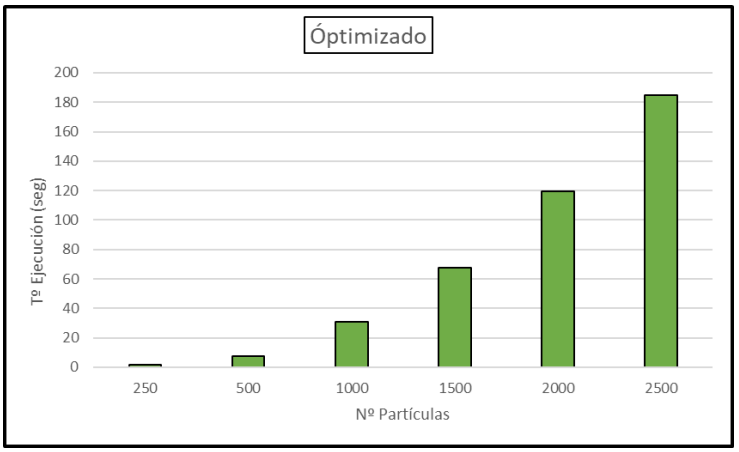
Además de la dimensión del problema, hemos explorado el impacto de otros parámetros escalables, como la precisión requerida y la complejidad de las interacciones moleculares simuladas. Hemos observado que, en algunos casos, ajustar estos parámetros puede influir en la eficiencia de la paralelización. Por ejemplo, al aumentar la precisión, se pueden introducir más cálculos intensivos, afectando la carga de trabajo paralelizable.

Explicación de las Variaciones en el Speed-Up:

Las variaciones en el speed-up están intrínsecamente relacionadas con la naturaleza del problema y cómo se estructura y paraleliza. Cuando la aplicación presenta una carga de trabajo suficientemente grande y paralelizable, el aumento en la dimensión del problema suele conducir a un mejor rendimiento. Sin embargo, es crucial considerar las características específicas del problema, ya que ciertos ajustes en los parámetros pueden tener efectos imprevistos en la paralelización y, por lo tanto, en el speed-up. Mostramos las gráficas de los tiempos de ejecución de diferentes parámetros de compilación:







En resumen, el rendimiento de la aplicación está fuertemente influenciado por la dimensión del problema y otros parámetros escalables. Al comprender cómo estas variables afectan la carga de trabajo y la paralelización, podemos ajustar nuestra aplicación para lograr un rendimiento óptimo en una variedad de configuraciones y condiciones.

Tarea 4

Precalentamiento para las siguientes prácticas

- Un grifo tarda 4 horas en llenar un cierto depósito de agua y otro grifo 20 horas en llenar el mismo depósito. Si usamos los dos grifos para llenar el depósito, que está inicialmente vacío, ¿cuánto tiempo tardaremos? ¿Cuál será la ganancia en velocidad? ¿Y la eficiencia?

Problema 1:

La tasa de llenado del primer grifo es de 1/4 del depósito por hora, y la del segundo es de 1/20 del depósito por hora. Cuando ambos grifos están abiertos, sus tasas se suman: 1/4 + 1/20 = 5/20 + 1/20 = 6/20 = 3/10. Por lo tanto, llenarán 3/10 del depósito por hora.

El tiempo necesario para llenar el depósito es la inversa de la tasa combinada:

$$\text{Tiempo} = \frac{1}{\text{Tasa combinada}} = \frac{1}{3/10} = \frac{10}{3} \text{ horas.}$$

Entonces el tiempo que tarda son 3 horas y 20 minutos.

La ganancia entonces sería el tiempo que hemos ahorrado respecto al usar el grifo más rápido o usar ambos simultáneamente y la eficiencia se calcula respecto al tiempo que habrían tardado cada grifo independientemente.

$$Ganancia = T_{grifo_{rápido}} - T_{ambos_{grifos}} = 4 - \frac{10}{3} = \frac{2}{3} = 40\ min$$

$$Eficiencia = \frac{Tiempo\ con\ ambos\ grifos}{Suma\ del\ tiempo\ de\ los\ grifos\ independientes} \cdot 100 = \frac{\frac{10}{3}}{4\ +\ 20} \cdot 100 = \frac{5}{36} \cdot 100 = 13.89\%$$

Problema 2:

- Suponga que tiene ahora 2 grifos de los que tardan 4 horas en llenar el depósito. Mismas cuestiones que el punto anterior.

La tasa de llenado de ambos grifos es la misma: ¼ por hora. Entonces el tiempo que tarda en llenarlo es:

$$Tiempo = \frac{1}{Tasa\ combinada} = \frac{1}{\frac{1}{4} + \frac{1}{4}} = 2\ horas$$

$$Ganancia = T_{grifo_{rápido}} - T_{ambos_{grifos}} = 4 - 2 = 2\ horas$$

$$Eficiencia = \frac{T_{ambosgrifos}}{Suma\ grifos\ independiente} \cdot 100 = \frac{2}{2+2} \cdot 100 = 50\%$$

Problema 3:

- Y ahora suponga que tiene 2 grifos de los que tardan 20 horas. Proceda también a realizar los cálculos.

$$Tiempo = \frac{1}{Tasa\ combinada} = \frac{1}{\frac{1}{20} + \frac{1}{20}} = 10\ horas$$

$$Ganancia = T_{grifo_{rápido}} - T_{ambos_{grifos}} = 20 - 10 = 10\ horas$$

$$Eficiencia = \frac{T_{ambosgrifos}}{Suma\ T_{grifos\ independientes}} \cdot 100 = \frac{10}{20 + 20} \cdot 100 = 25\%$$

Problema 4:

- Ahora tiene 3 grifos: 2 de los que tardan 20 horas y 1 de los que tardan 4. ¿Qué pasaría ahora?

$$Tiempo = \frac{1}{Tasa\ combinada} = \frac{1}{\frac{1}{20} + \frac{1}{20} + \frac{1}{4}} = \frac{20}{7}\ horas$$

$$Ganancia = T_{grifo_{rápido}} - T_{todos_{grifos}} = 4 - \frac{20}{7} = \frac{8}{7}\ horas$$

$$Eficiencia = \frac{T_{todosgrifos}}{SumaT_{grifosindep}} \cdot 100 = \frac{\frac{8}{7}}{20 + 20 + 4} \cdot 100 = \frac{200}{77}\%$$