

Trabajo final de Computación paralela: *tiny md*

Francisco Fernández
email: ffernandez@famaf.unc.edu.ar

28 de enero de 2021

Resumen

AGREGAR RESUMEN CUANDO TERMINE¹

1. Introducción teórica al problema

La dinámica molecular (MD, de sus siglas en inglés, Molecular Dynamics) es una técnica de simulación computacional que considera la interacción entre partículas atómicas para obtener una evolución temporal de las mismas. Esto se logra resolviendo numéricamente las ecuaciones de movimiento de Newton. A partir de cantidades microscópicas (posiciones (x), velocidades (v), fuerzas (f)) se pueden obtener propiedades termodinámicas macroscópicas del sistema en equilibrio (temperatura (T), presión (P)). Esta técnica tiene aplicaciones en muchas áreas del conocimiento, tales como física, química, biofísica, ciencias de los materiales, etc.

1.1. Programa de MD

Una descripción simple de un programa de dinámica molecular se introduce a continuación:

- *Inicialización del sistema*: se especifican la cantidad de partículas N , la temperatura de referencia T y la densidad ρ , de donde puede obtenerse el volumen V , largo de la caja L . Además, se elige r_{cut} , el paso temporal dt , las posiciones y velocidades iniciales.
- *Cálculo de fuerzas*: se computan las fuerzas de todas las partículas.
- *Integración de las ecuaciones de movimiento*: se integran las ecuaciones de Newton, con algún integrador que a partir de la condición anterior obtiene las posiciones y velocidades del paso temporal siguiente.
- *Mediciones*: se realizan cálculos de distintas cantidades de interés (energía potencial, cinética, presión, temperatura).
- *Evolución temporal*: $t = t + dt$.

A continuación se amplía cada una de estas secciones específicamente para el programa presentado. Ya que hay distintas formas de inicializar el sistema, de calcular las fuerzas con distintos potenciales, algoritmos de evolución, etc.

¹Directorio de GitHub: https://github.com/fernandezfran/tiny_md

1.1.1. Inicialización del sistema

En este caso las posiciones se inicializan dentro de una red cristalina *FCC* y las velocidades se dan aleatoriamente entre $-0,5$ y $0,5$, se les resta la velocidad del centro de masa para que el sistema no se esté desplazando y se las multiplica por un factor que involucra la temperatura.

1.1.2. Condiciones periódicas de contorno e imagen mínima

En el caso simulado acá se utilizan condiciones periódicas de contorno (pbc, periodic boundary conditions), las mismas buscan reproducir un sistema infinito para que no haya efectos de borde y consisten en considerar las N partículas como una celda primitiva de una red infinita de celdas idénticas, en donde, si una partícula sale por un extremo de la caja, ingresa por el opuesto.

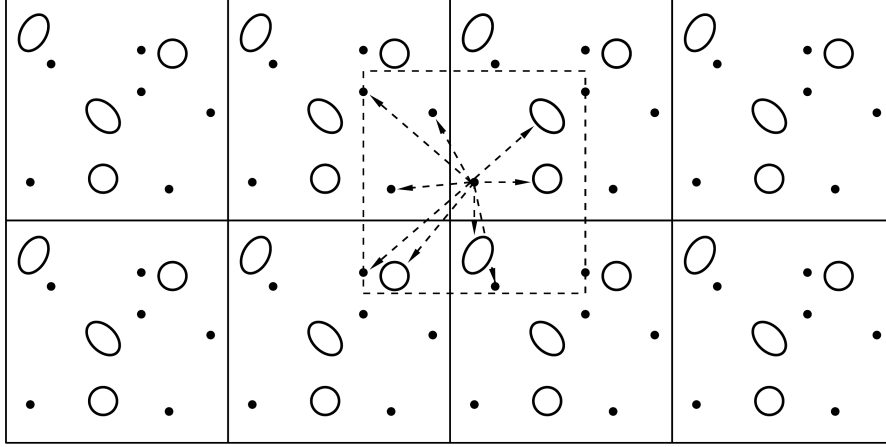


Figura 1. Condiciones periódicas de contorno e imagen mínima.

En la figura 1 puede verse como se replica una misma celda en todas las direcciones y como al estar centrado en una partícula es necesario salirse de la celda hacia celdas vecinas para encontrar la imagen mínima, que es la distancia más cercana a una partícula.

1.1.3. Cálculo de fuerzas

Para el cálculo de las fuerzas se utiliza un potencial aditivo de pares de Lennard–Jones (12–6), dado por la siguiente expresión,

$$V_{LJ}(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right],$$

donde r es la distancia entre dos partículas, σ es el “tamaño de la partícula” y ε indica que tan profundo es el potencial en el mínimo $r_m = 2^{1/6}\sigma$. La forma funcional se muestra en la figura 2, si la distancia entre dos pares de partículas es menor a r_m se repelen, si la distancia es mayor, se atraen, y no interactúan si la distancia es infinita.

Antes de calcular las fuerzas, se necesita la distancia entre dos partículas i y j , utilizando la regla de la imagen mínima para considerar la distancia a la imagen más cercana, y calculando la fuerza sólo si dicha distancia es menor a un radio de corte, r_{cut} , de la siguiente forma

$$f_x(r) = -\frac{\partial u(r)}{\partial x} = -\left(\frac{x}{r}\right) \left(\frac{\partial u(r)}{\partial r}\right),$$

que para el caso del potencial de Lennard–Jones queda

$$f_x(r) = \frac{24x}{r^2} \left(\frac{2}{r^{12}} - \frac{1}{r^6} \right)$$

tanto para x como para y y z .

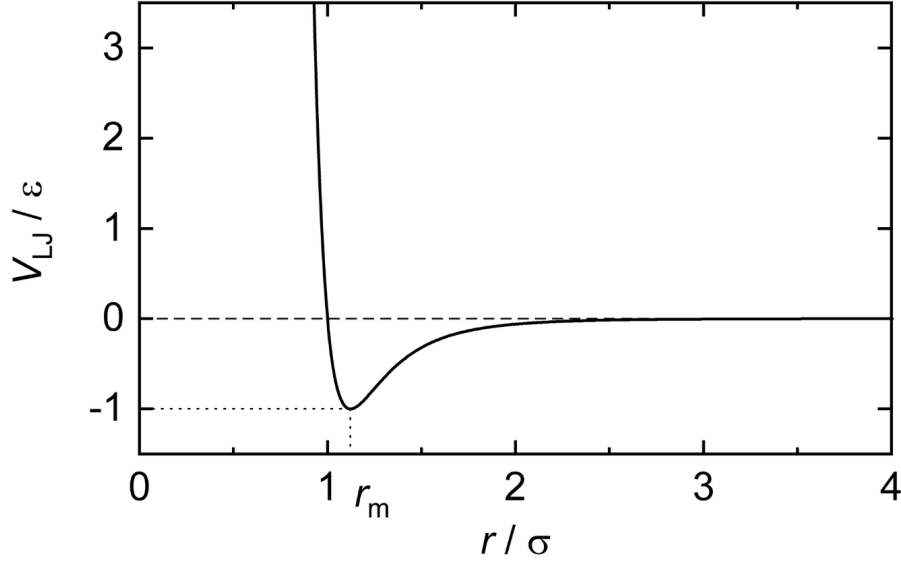


Figura 2. Gráfico del potencial de Lennard–Jones.

Dentro de esta misma sección de código se calcula la energía potencial y la presión instantáneas.

1.1.4. Integración de las ecuaciones de movimiento

Se utiliza el algoritmo *Velocity Verlet*, el mismo conserva la energía total del sistema si se encuentra en el ensamble NVE, que para las posiciones se ve como un desarrollo de Taylor,

$$r(t + \Delta t) = r(t) + v(t)\Delta t + \frac{f(t)}{2m}\Delta t^2,$$

y a las velocidades se las actualiza como

$$v(t + \Delta t) = v(t) + \frac{f(t + \Delta t) + f(t)}{2m}\Delta t,$$

esto exige calcular las velocidades una vez que se obtuvieron las nuevas posiciones y, a partir de ellas, las nuevas fuerzas.

Aquí se calcula la energía cinética y la temperatura instantáneas.

1.1.5. Mediciones

Tanto a la energía potencial como a la presión es necesario sumarles una contribución de cola debido al *truncado y desplazado* que se realiza en r_{cut} , para que el potencial se anule en este punto, esto asumiendo que la energía potencial aportada por una partícula es dominada por las interacciones de las partículas más cercanas. Para la presión se tiene

$$P_{tail} = \frac{16}{3}\pi\rho^2\varepsilon\sigma^3 \left[\frac{2}{3} \left(\frac{\sigma}{r} \right)^9 - \left(\frac{\sigma}{r} \right)^3 \right],$$

y para la energía

$$U_{tail} = \frac{16}{3}N\pi\rho\varepsilon\sigma^3 \left[\frac{2}{3} \left(\frac{\sigma}{r} \right)^9 - \left(\frac{\sigma}{r} \right)^3 \right].$$

La presión total será una suma de esta contribución de cola y la presión del virial

$$P = \rho k_B T + \frac{1}{dV} \left\langle \sum_{i < j} \mathbf{f}(\mathbf{r}_{ij}) \cdot \mathbf{r}_{ij} \right\rangle,$$

donde k_B es la constante de Boltzmann y d la dimensión del sistema.

La energía total es la suma de la contribución de cola más la suma que se obtiene a través de la interacción entre las partículas a través del potencial de Lennard–Jones.

Por otro lado, tanto la energía cinética como la temperatura se obtienen utilizando las velocidades de la siguiente manera

$$E_{kin} = \frac{1}{2} \sum_{i=1}^N m_i v_i^2,$$

$$k_B T = \frac{1}{3N} \sum_{i=1}^N m_i v_i^2$$

donde m_i es la masa de la partícula i

1.2. Ecuación de estado

La ecuación de estado relaciona las variables de un sistema bajo ciertas condiciones físicas. En este caso se presenta la presión en función de la densidad del sistema, es decir, se realizan simulaciones en el ensamble canónico (NVT) en las que inicialmente se fija un volumen y la temperatura se mantiene reescalando las velocidades, para cada una de ellas se calcula la presión para obtener la ecuación de estado P vs ρ que se muestra en la figura 3. Para densidades altas el sistema se comporta como un sólido, las posiciones iniciales del cristal FCC se mantienen, y para densidades bajas se comporta como un líquido en el cual las partículas están desorganizadas.

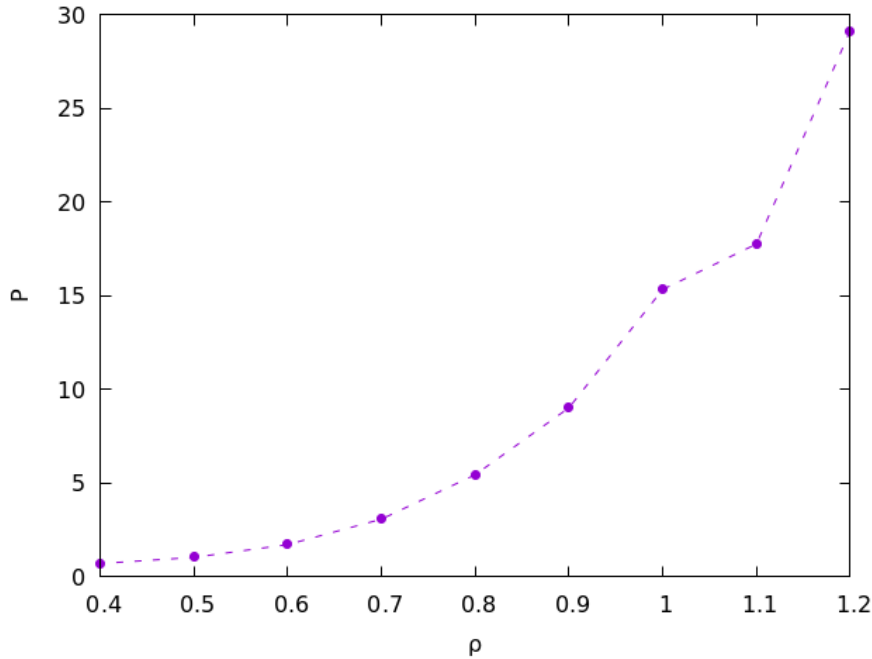


Figura 3. Ecuación de estado para la isoterma $T = 2,0$.

2. Resultados y discusiones

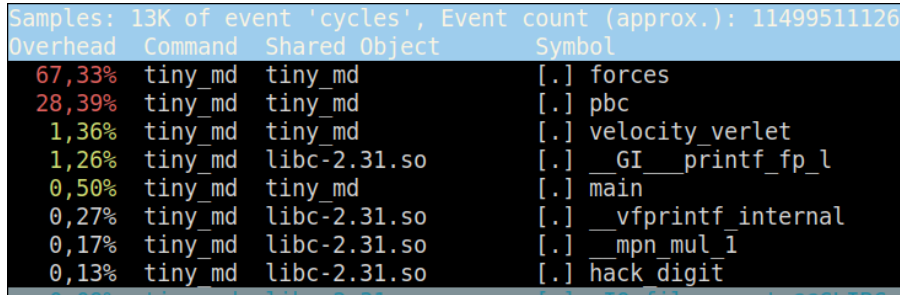
Unidades y métrica

Las unidades en las que se encuentran las cantidades físicas dentro del programa son unidades reducidas de Lennard–Jones. Esto quiere decir que se considera $k_B = 1$, que la masa está medida en unidades de m , que la distancia está medida en unidades de σ y la energía en unidades de ε . A partir de esto se pueden convertir los valores de todas las propiedades físicas, por ejemplo, $T^* = k_B T / \varepsilon$ y $t^* = t \sqrt{\varepsilon / m \sigma^2}$, donde $*$ como superíndice quiere decir que es la forma reducida.

Una métrica usual de dinámica molecular es *ns/day*, es decir, cuántos nanosegundos evoluciona el sistema en un día de simulación. Esta métrica está implementada en el código pero depende del tamaño del problema. En algunas gráficas se presenta la misma y en otras el tiempo total de simulación.

perf

En la figura 4 se muestra lo obtenido al realizar `perf record ./tiny_md && perf report`. Más del 95 % del tiempo se computo se consume en las funciones `forces` y `pbk` (que es llamaba por `forces` y `velocity_verlet`), así que estas son las secciones de código en las que se necesitan hacer optimizaciones.



The image shows a terminal window displaying the output of the 'perf report' command. The output is a table with four columns: 'Overhead', 'Command', 'Shared Object', and 'Symbol'. The data is as follows:

Overhead	Command	Shared Object	Symbol
67,33%	tiny_md	tiny_md	[.] forces
28,39%	tiny_md	tiny_md	[.] pbk
1,36%	tiny_md	tiny_md	[.] velocity_verlet
1,26%	tiny_md	libc-2.31.so	[.] __GI___printf_fp_l
0,50%	tiny_md	tiny_md	[.] main
0,27%	tiny_md	libc-2.31.so	[.] __vfprintf_internal
0,17%	tiny_md	libc-2.31.so	[.] __mpn_mul_i
0,13%	tiny_md	libc-2.31.so	[.] hack_digit

Figura 4. Porcentaje del tiempo simulado en las funciones del código.

2.1. Optimizaciones secuenciales

AoS

Inicialmente en el código las posiciones, velocidades y fuerzas se almacenan en memoria de la forma *Array of Structure*. En el vector de las posiciones (velocidades o fuerzas) se tienen almacenadas en memoria las coordenadas de la misma partícula de forma consecutiva, es decir que, si se lo piensa como matriz, las coordenadas de cada una de las direcciones se encuentran en columnas.

SoA

Esta otra implementación del código cambia la forma del almacenamiento en memoria, aquí como *Structure of Array*. En el vector de las posiciones se tiene primero las coordenadas x de todas las partículas, luego las y y luego las z , es decir, que en forma de matriz las coordenadas de cada una de las direcciones están almacenadas en filas.

Mixed precision

En este otro caso en vez de usar `double` para los vectores de las posiciones, velocidades y fuerzas, se utilizan `float`.

En esta parte del trabajo se realizó una exploración con los distintos flags de optimización que se encuentran listados en la tabla 1 del compilador `gcc`. Las mediciones se realizaron en `zx81`² utilizando `SLURM`, `script` de `python3` y `perf stat`. Los resultados obtenidos se muestran en la figura 5, donde se gráfica los `ns/day` en función de los distintos flags de optimización para las tres versiones del programa explicadas anteriormente.

La tendencia que se observa en la figura 5 es que para la versión SoA se obtienen mejores resultados que para la AoS y que la versión mixed precision tiene menor performance que las otras dos. Para los tres casos puede afirmarse, debido a que las barras de error son menores al tamaño de los puntos, que la combinación de flags que mejor resultados da es `-O3 -funroll-loops -ffast-math -march=native`. De la cual se obtiene $(1,60 \pm 0,02)$ `ns/day` para la versión SoA. Inicialmente se tenía $(0,500 \pm 0,003)$ `ns/day`, lo que da un speed-up de 3.2x.

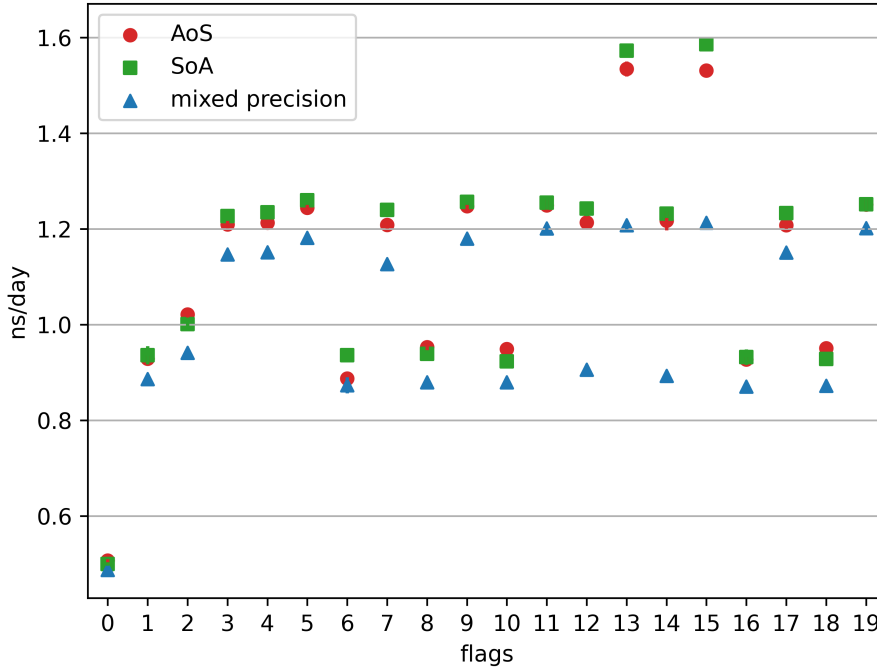


Figura 5. Métrica `ns/day` para los distintos flags de optimización listados en la tabla 1. Para la obtención de cada punto se realizaron 30 mediciones y el número de partículas fue $N = 256$. Las barras de error en cada punto son menores que el tamaño del mismo.

2.2. Vectorización

Autovectorización

Se utilizó el flag de autovectorización, `-ftree-vectorize`, en la misma búsqueda de la tabla 1. Además de las versiones AoS y SoA, se agregó una versión *Naive-loops*, a partir de la

²ver Apéndice A

Tabla 1: Flags de optimización utilizados en el siguiente orden para la obtención de los datos presentados en la figura 5.

	flag
0	-O0
1	-O1
2	-Os
3	-O2
4	-O3
5	-Ofast
6	-O1 -funroll-loops
7	-O3 -funroll-loops
8	-O1 -ffast-math
9	-O3 -ffast-math
10	-O1 -funroll-loops -march=native
11	-O3 -funroll-loops -march=native
12	-O1 -ffast-math -march=native
13	-O3 -ffast-math -march=native
14	-O1 -funroll-loops -ffast-math -march=native
15	-O3 -funroll-loops -ffast-math -march=native
16	-O1 -floop-block -floop-interchange
17	-O3 -floop-block -floop-interchange
18	-O1 -funroll-loops -floop-block -floop-interchange -march=native
19	-O3 -funroll-loops -floop-block -floop-interchange -march=native

versión SoA, en la cual se juntan las operaciones que se hacen en x , y , z dentro de un loop que va de 0 a 2, en todas las partes del código en las que fue posible. Se presentan los datos obtenidos para la mejor combinación de flags en la tabla 2. Tanto para AoS como para SoA la mejor combinación sigue siendo la encontrada anteriormen, para *Naive-loops* se obtuvo que la mejor es `-O1 -funroll-loops -ffast-math -march=native -ftree-vectorize`, que da, comparando con el proyecto inicial, un speed-up de 3.46x.

Tabla 2: Resultados de la autovectorización en ns/day, se realizaron 10 mediciones para dar cada uno de los valores.

versión	ns/day
AoS	$1,55 \pm 0,02$
SoA	$1,577 \pm 0,004$
naive loops	$1,73 \pm 0,01$

Utilizando el flag de compilación `-fopt-info-vec-optimized` se ve que tanto para AoS como para SoA los loops autovectorizados son básicamente los mismos, parte de la inicialización de las velocidades, el reescalo de las mismas y `velocity_verlet`. Como se aprecia en los datos mostrados en la tabla 2, estas optimizaciones no son significativas en la simulación con respecto al laboratorio anterior. Por otro lado, en la versión *Naive-loops* no se optimiza `velocity_verlet` pero sí `pb` y parte de `forces` (el loop más interno y el cálculo de la distancia, pero no el de la fuerza y el cargado de estos datos al vector).

SSE

2.3. Paralelización: OpenMP

Apéndice A **zx81**

Características de **zx81**.