

Herramientas Informáticas Avanzadas: Molecular Dynamics Simulation of a Van der Waals Gas

Grupo 1:

Jon López, Asier Zabalo, Manuel Cánovas, Didac Martí, Joan Giralt, Sergio Granados

31 de marzo de 2019

Índice

1. Introducción y distribución del trabajo	3
2. Inicialización del sistema	3
2.1. Código en serie	4
2.2. Código en paralelo	4
3. Cálculo de fuerzas	5
3.1. Programación en serie	6
3.2. Programación en paralelo	6
4. Integración de las ecuaciones del movimiento	7
4.1. Programación en serie	7
4.1.1. Velocity Verlet	7
4.1.2. Termostato	7
4.2. Programación en paralelo	8
4.2.1. Velocity Verlet	8
4.2.2. Termostato	10
5. Condiciones periódicas de contorno	10
5.1. Programación en serie	11
5.2. Programación en paralelo	11
6. Función de distribución radial	12
6.1. Programación en serie	13
6.2. Programación en paralelo	13
7. Desplazamiento cuadrático medio	15
7.1. Programación en serie	15
7.2. Programación en paralelo	15
8. Main	16
8.1. Programación en serie	16
8.2. Programación en paralelo	17
9. Promedios estadísticos <i>Binning</i>	17

10.Resultados de la dinámica molecular	18
10.1. Sistema 1: gas de Helio	19
10.2. Sistema 2: líquido de Argón	20
11.Estudio de escalabilidad	21

1. Introducción y distribución del trabajo

El objetivo principal de este proyecto es desarrollar en grupo un código de dinámica molecular para el estudio de un fluido de Van der Waals. El código se ha implementado tanto en serie como en paralelo, y su desarrollo se ha realizado empleando la plataforma GitHub. Para comprobar el funcionamiento del código, se han simulado dos sistemas diferentes: un gas de helio y un líquido de argón, cuyas condiciones se especificarán más adelante.

Todo el código se ha escrito utilizando el lenguaje Fortran 90, y para la programación en paralelo se ha utilizado la versión 1.10.4. del programa OpenMPI. Por último, para la visualización de los diferentes resultados se utilizó Gnuplot.

Ya que el proyecto se ha realizado en grupo, lo primero es explicar cómo se ha distribuido el trabajo entre los diferentes componentes del grupo.

- **Jon López.** Se ha encargado de la coordinación del proyecto, siendo por lo tanto el administrador del repositorio de GitHub. También ha realizado la programación en serie del integrador de las ecuaciones del movimiento (*Velocity Verlet*) y del termostato (*Andersen*), así como la programación en serie del integrador.
- **Asier Zabalo.** Ha sido el encargado de crear los Makefile para compilar y ejecutar los códigos creados. En cuanto a la programación, ha escrito el código correspondiente a la inicialización del sistema (posiciones y velocidades), tanto en serie como en paralelo.
- **Manuel Cánovas.** Ha programado el cálculo de las fuerzas, energías y presión, tanto en serie como en paralelo.
- **Didac Martí.** Se ha encargado de programar y paralelizar la subrutina para imponer las condiciones periódicas de contorno.
- **Joan Giralt.** Ha llevado a cabo el cálculo de varias magnitudes a partir de la trayectoria, como son la función de distribución radial y el desplazamiento cuadrático medio. También ha realizado un módulo para el análisis estadístico de los datos obtenidos de la simulación mediante el método de *binning*.
- **Sergio Granado.** Ha sido el encargado de programar los *script* necesarios para la visualización en Gnuplot de los resultados de la simulación. Por otro lado, ha escrito el programa principal y ha paralelizado la subrutina del termostato de *Andersen*.

En las siguientes secciones se irá describiendo cada una de las partes del código, tanto los fundamentos para la programación del algoritmo en serie, como los algoritmos empleados para la paralelización del mismo. Tras esto, se procederá al estudio de la escalabilidad de la implementación en serie y por último se mostrarán los resultados obtenidos.

2. Inicialización del sistema

Como primer paso de la dinámica molecular, hemos creado dos subrutinas, *sub_initialize_v.f90* y *sub_initialize_r.f90*, que inicializan las velocidades y las posiciones de las partículas, respectivamente. Debido a que estas subrutinas solamente son utilizadas una vez en todo el programa, el tiempo de cálculo que ahorraremos paralelizándolas será prácticamente nulo. Aún así, tratándose de un ejercicio con fines académicos, la subrutina de inicialización de las velocidades también ha sido implementada en paralelo.

2.1. Código en serie

En primer lugar comentaremos la subrutina *sub_initialize_r.f90*. Para inicializar la posición de las partículas pedimos como parámetros de entrada la densidad del sistema (en unidades reducidas) y con cuántas partículas estamos trabajando, N . Como salida obtenemos una matriz de dimensiones $(N,3)$ donde cada fila i contiene las coordenadas x,y,z de la partícula número i . Por otro lado, también se devuelve la dimensión de la caja donde se encuentra el sistema, L , siendo el volumen total $V = L^3$. Respecto al posicionamiento de las partículas, se crea una estructura cristalina *fcc* (*face-centered cubic*) completamente perfecta a partir de los parámetros de entrada. Claramente, si se desea simular un líquido o un gas como es nuestro caso, tendremos que realizar una pequeña simulación para que los átomos se desplacen ligeramente de las coordenadas iniciales.

En cuanto a la subrutina de inicialización de las velocidades *sub_initialize_r.f90*, como parámetros de entrada pedimos la temperatura del sistema T y el número de partículas N y devolvemos, siguiendo la misma notación que con las posiciones, una matriz de dimensiones $(N,3)$. En un primer lugar las velocidades se inicializan utilizando el generador de números aleatorios de Fortran90^[1], y tras calcular la suma de las velocidades en las tres direcciones x,y,z se fija el momento total del sistema a cero. Finalmente, tras calcular la energía cinética del sistema, se escalan todas las velocidades con la temperatura. El pseudocódigo podría expresarse de la siguiente manera:

Algorithm 1 Inicialización de las velocidades, código en serie

```
1: procedure INITIALIZE_v(Input:T,N Output:v)
2:   # Inicializamos las velocidades con números aleatorios entre [0,1]
3:   v ← random numbers
4:   # Centramos los números aleatorios alrededor del cero
5:   v ← v - 0,5
6:   Se exige que el momento total del sistema sea nulo
7:   v(:,i) ← v(:,i) - sum(v(:,i))/N   for i=x,y,z
8:   # Calculamos la energía cinética del sistema
9:   kinetic ← 0,5 · sum(v2)
10:  # Escalamos las velocidades con la temperatura del sistema
11:  v ← v · √(1,5 · N · T/kinetic)
```

2.2. Código en paralelo

La subrutina *sub_initialize_v.f90* ha sido implementada en paralelo. El cálculo es exactamente el mismo que el explicado en la sección anterior, pero esta vez varios procesadores realizarán tareas simultáneamente. Al inicio se divide la matriz de velocidades, a la que llamaremos v , entre el número de procesadores, *numproc*. De esta manera, cada procesador trabajará con matrices de dimensión $(N/numproc, 3)$, las cuales denominamos *local*.

En primer lugar, cada procesador inicializará su matriz *local* con números aleatorios independientes a los demás procesadores. Esto se consigue mediante una *seed* (semilla del generador de números aleatorios) que dependa del identificador del procesador (*taskid* en nuestro código). A continuación, cada procesador calcula el momento de su vector *local* para las direcciones x, y, z y mediante la subrutina *MPI_AllReduce* sumamos todas las contribuciones, de manera que

¹A lo largo de todo el código, siempre que ha sido necesario generar números aleatorios, hemos utilizado la subrutina intrínseca *random_number*.

todos los procesadores conozcan la suma de todos los momentos parciales. Es decir, todos los procesadores conocen el momento total del sistema. De esta manera, al igual que en la versión en serie, exigimos que el momento total sea cero, pero esta vez cada procesador impone esta condición a su vector *local*. Del mismo modo, cada procesador calcula la contribución de su vector a la energía cinética del sistema, y una vez más, mediante *MPI_AllReduce* sumamos todas las contribuciones haciendo que todos los procesadores conozcan la energía cinética total. En este punto, cada procesador escala su vector *local* de velocidades con la temperatura. Finalmente, lo único que nos queda por hacer es juntar todas las matrices *local* en una matriz *v* que contenga las velocidades de todas las partículas del sistema y que todos los procesadores conozcan esta matriz. Esto se consigue mediante la subrutina *MPI_AllGather*.

Cabe destacar que, en general, el número de partículas *N* no podrá ser dividido por el número de procesadores (no obtendremos un número entero). Por lo tanto, a lo largo del programa tendremos que realizar comprobaciones para completar las entradas de la matriz total *v* correspondientes a las partículas que no se han podido asignar a ningún procesador. Por simplificar el pseudocódigo, estas comprobaciones no se muestran en Algorithm 2, pero es algo que ha sido implementado en nuestro código.

Algorithm 2 Inicialización de las velocidades, código en paralelo

```

1: procedure INITIALIZE_v(Input:T,N,taskid,numproc Output:v)
2:   local(partition,3)  $\leftarrow$  partition = N/numproc
3:   local _seed  $\leftarrow$  377 · (1 + taskid)
4:   local  $\leftarrow$  random numbers
5:   local  $\leftarrow$  local - 0,5
6:   suma _local(i)  $\leftarrow$  sum(local(:,i)), for i=x,y,z
7:   MPI_ALL_REDUCE(SUM,suma_local,suma)
8:   local(:,i)  $\leftarrow$  local(:,i) - suma(i)/N for i=x,y,z
9:   kinetic_local = sum(local2)
10:  MPI_ALL_REDUCE(SUM,kinetic_local,kinetic)
11:  local = local ·  $\sqrt{1,5 \cdot N \cdot T / \text{kinetic}}$ 
12:  MPI_ALLGATHER(local,v)

```

3. Cálculo de fuerzas

En esta parte del código tenemos como objetivo calcular las fuerzas de interacción entre las partículas a partir del potencial de Lennard-Jones, la presión y las energías potenciales y cinéticas. Para las fuerzas seleccionamos 2 partículas *i,j*. La distancia se calcula usando el criterio de imagen mínima. Una vez se obtiene la distancia *i,j* al cuadrado para calcular la fuerza solo se tiene en cuenta si no supera al cutoff al cuadrado, con el cual restringimos la distancia para el cual la interacción se tiene en cuenta para evitar interacciones muy largas. Utilizando las formulas del potencial de Lennard-Jones y su derivada calculamos la fuerza y el potencial con las siguientes expresiones:

$$Pot_{ij}^{LJ} = 2\left(\frac{1}{r_{ij}^{12}} - \frac{1}{r_{ij}^6}\right) \quad (1)$$

La fuerza en la dirección x, las otras se consiguen cambiando el dx por dy o dz:

$$F_{ij}^{LJ} = \left(\frac{48}{r_{ij}^{14}} - \frac{24}{r_{ij}^8}\right) * dx \quad (2)$$

La energía cinética se calcula usando las velocidades al cuadrado con la siguiente formula:

$$E_{cin} = \frac{1}{2} \sum_{i=1}^N v_i^2 \quad (3)$$

La presión tiene 2 componentes. Una de origen cinético y otro de potencial. La calcularemos usando la siguiente fórmula:

$$P = \frac{1}{3L^3} [2E_{cin} + \sum_{i=1}^N \sum_{j=1}^3 x(i,j)F(i,j)] \quad (4)$$

3.1. Programación en serie

La subrutina force en serie tiene como valores de entrada el número de partículas (N), la longitud de la caja (L), un cutoff, las matrices de posiciones y velocidades. Como valores de salida con esta subrutina se obtienen la matriz de Fuerzas, la presión cinética y las energías cinéticas y potenciales.

Una vez entramos se recorren 2 bucles asignando i, j donde cada uno va de 1 a N. A continuación se calcula la distancia al cuadrado entre la partícula i y la j, pero el caso de i=j no se tiene en cuenta ya que no hay auto interacción. Calculamos la fuerza con las ecuaciones previas y la añadimos a la matriz de fuerzas para la partícula i en las 3 direcciones; mientras que la energía potencial la vamos sumando en cada interacción de i, j que se acepte por el cutoff.

La energía cinética y la presión se calcula usando la formula previa. Con esto ya tenemos todas as variables de salida calculadas en serie.

3.2. Programación en paralelo

En el caso de en paralelo a diferencia de en serie hemos separado el calculo en 2 subrutinas diferentes. En force calculamos la matriz de fuerzas y la energía potencial; mientras que en ekinpress la cinética y la presión. Esto se debe a que usamos las matrices de indices diferentes. En la primera hacemos el calculo con parejas de interacción para evitar cálculos repetidos con table_index2 que lleva los indices iniciales y finales de las parejas para cada nodo, mientras que en el otro con las de partículas con table_index1.

La subrutina force usa como variables de entrada el numero de procesadores (numproc), el nodo de cada procesador (taskid), table_index2, el numero de partículas (N), la matriz de pares (Pair), la dimensión de las matrices (dim), la longitud de la caja (L), un cutoff, las matrices de posiciones y velocidades. Mientras que las variables de salida son la matriz de fuerza total (Ftot) que estará en todos los nodos; y la energía potencial (resepot) que solo está en el nodo 0.

Cada nodo cuando entra usa el table_index2 para saber cuales son sus indices de parejas inicial (Nini) y final(Nfin) para él usando:

$$\begin{aligned} Nini &= \text{table_index2}(taskid, 1) \\ Nfin &= \text{table_index2}(taskid, 2) \end{aligned} \quad (5)$$

De esta manera con un único bucle recorremos todas las parejas donde cada nodo va desde su Nini a Nfin. Luego con la matriz de parejas cargamos los indices en las variables i, j y repetimos el cálculo de las fuerzas y la energía potencial en cada nodo de manera análoga a en serie, con la diferencia de que anotamos la fuerza de i sobre j y la de j sobre i con signo cambiado, ya que calculamos la mitad que antes. Por este mismo motivo, también en cada iteración multiplicamos por 2 la contribución a la energía potencial. A continuación, necesitamos sumar la matriz de fuerzas de cada uno de los nodos y enviárselos a todos ellos. Para ello usamos el comando

Mpi_AllReduce con lo que sumamos todas las matrices entre ellas y se envía a todos los nodos. Finalmente, la energía potencial que ha calculado cada nodo se envía al nodo 0 donde se suman y obtenemos solo en ese nodo la energía potencial total del sistema usando el Mpi_Reduce.

La otra subrutina calcula la energía cinética y la presión donde tenemos como variables de entrada el número de procesadores (numproc), el nodo de cada procesador (taskid), table_index1, el numero de partículas (N), la matriz de pares (Pair), la dimensión de las matrices (dim), la longitud de la caja (L) y las matrices de posiciones, velocidades y fuerzas. Como variables de salida tenemos solo en el nodo 0 la energía cinética y la presión. Los indices de las partículas de cada nodo se inicializan como antes pero con la matriz table_index1 y no con la 2 ya que no son parejas sino solo partículas. La energía cinética y la presión se calculan en cada nodo de forma local y finalmente usando el Mpi_reduce se envían y se suman en el nodo 0 de manera conjunta.

4. Integración de las ecuaciones del movimiento

Una de las partes vitales del código es integrar las ecuaciones del movimiento para describir las trayectorias del conjunto de las partículas. La simuación se ha realizado en el conjunto canónico, es decir, manteniendo el volumen y la temperatura constantes. Para ello, se ha decidido implementar el algoritmo de *Velocity Verlet*, que se complementa con el termostato de Andersen para mantener la temperatura constante.

La principal característica de este método de integración es que trabaja tanto con las posiciones de las partículas $\{\mathbf{r}_i\}$ en el instante t como con las velocidades $\{\mathbf{v}_i\}$ en ese mismo momento, por lo que es ideal para la termalización. La actualización de las coordenadas en un instante $t + \Delta t$ se hace de la siguiente manera:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{\mathbf{f}_i(t)}{2m_i}\Delta t^2 \quad (6)$$

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \frac{\mathbf{f}_i(t) + \mathbf{f}_i(t + \Delta t)}{2m_i}\Delta t \quad (7)$$

para todas las $i = 1, \dots, N$, siendo N el número de partículas.

4.1. Programación en serie

En primer lugar, se describirá en qué consisten los algoritmos empleados y cómo es su implementación en serie.

4.1.1. Velocity Verlet

Comenzando por el integrador de las ecuaciones, el pseudocódigo a seguir es el siguiente:

En primer lugar se toman como entrada las posiciones, velocidades y fuerzas calculadas para el instante t . Con esto se actualizan las posiciones para el instante $t + \Delta t$, mediante la expresión de la ecuación 6. Al mismo tiempo, se hace una actualización parcial de las velocidades, sumando la contribución de la fuerza en el instante t de la ecuación 7. Con las posiciones actualizadas se pueden calcular las fuerzas sobre todas las partículas en el siguiente instante, para así poder calcular las velocidades definitivas sumando la contribución que falta.

4.1.2. Termostato

Por otro lado, el termostato de Andersen se basa en el resultado de la mecánica estadística que dicta que la distribución de velocidades de un sistema en contacto con un baño térmico viene

Algorithm 3 Paso del algoritmo de integración Velocity Verlet.

```
1: procedure VELOCITY VERLET( $\{\mathbf{r}(t)\}$ ,  $\{\mathbf{v}(t)\}$ ,  $\{\mathbf{F}(t)\}$ )
2:   for  $i = 1, N$  do:
3:      $\mathbf{r}_i(t + \Delta t) \leftarrow \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{\mathbf{f}_i(t)}{2m_i}\Delta t^2$ 
4:      $\mathbf{v}_i(t + \Delta t) \leftarrow \mathbf{v}_i(t) + \frac{\Delta t}{2m_i}\mathbf{f}_i(t)$ 
5:   end for
6:    $\{\mathbf{F}(t + \Delta t)\} \leftarrow \text{call forces } (\{\mathbf{r}(t + \Delta t)\})$ 
7:   for  $i = 1, N$  do:
8:      $\{\mathbf{v}(t + \Delta t)\} \leftarrow \mathbf{v}_i(t + \Delta t) + \frac{\Delta t}{2m_i}\mathbf{f}_i(t + \Delta t)$ 
9:   end for
```

dada por una distribución de Maxwell-Boltzmann:

$$P(v_{i,\alpha}) = \sqrt{\frac{m_i}{2\pi k_B T}} e^{-m_i v_{i,\alpha}^2 / 2k_B T}, \quad \forall i = 1, \dots, N \quad \forall \alpha = 1, 2, 3 \quad (8)$$

Es decir, cada componente de la velocidad de cada partícula debe seguir una distribución gausiana con desviación estándar de $\sigma = \sqrt{k_B T / m_i}$.

El efecto del baño térmico se simula de manera a cada paso de tiempo, cada partícula tiene una probabilidad ν determinada de interactuar con el baño, y en el caso de interacción, la velocidad de dicha partícula se cambia por una nueva que esté distribuida de acuerdo a la temperatura deseada.

Algorithm 4 Termostato de Andersen.

```
1: procedure THERMOSTAT( $v(t)$ ,  $\nu$ )
2:   for  $i = 1, N$  do:
3:     Extraer un número aleatorio uniforme  $R \sim U[0, 1]$ 
4:     if  $R < \nu$  then
5:       La partícula  $i$  interactúa con el baño
6:       for  $\alpha = 1, 2, 3$  do:
7:          $v_{i,\alpha} \leftarrow \text{Gausiana } (\sigma = \sqrt{k_B T})$ 
8:       end for
9:     end if
10:   end for
```

En la práctica, después de haber actualizado todas las posiciones y velocidades mediante el integrador *Velocity Verlet*, se pasan las velocidades por el termostato de Andersen.

4.2. Programación en paralelo

A la hora de parallelizar el programa, se ha decidido parallelizar cada una de las dos subrutinas mencionadas anteriormente por separado, ya que la tarea de parallelización de cada una de ellas se ha llevado a cabo por un miembro diferente. Por lo tanto, comenzaremos por el algoritmo de *Velocity Verlet*.

4.2.1. Velocity Verlet

El principal problema que se plantea a la hora de la paralización de esta subrutina, es el hecho de que se deben calcular las fuerzas en la mitad del proceso, por lo en dicho punto todos los

workers deberán conocer las posiciones de todas las partículas (ya que en principio se contemplan las interacciones entre todas las partículas). Además toda la información debe ser compartida también al final de la subrutina; es decir, en la salida todos los *workers* deben conocer las posiciones y velocidades de todo el sistema.

Si almacenamos las posiciones de las partículas en una matriz de dimensión $N \times 3$, de manera que la posición de la partícula número i se almacena como un vector de tres componentes en la fila i de la matriz, y hacemos lo mismo con sus velocidades y con las fuerzas totales ejercidas sobre cada partícula, las ecuaciones 6 y 7 se puede interpretar como la suma de varias matrices. Por lo tanto, las operaciones matriciales que se deben hacer se pueden descomponer como sumas elemento-elemento independientes entre sí.

Aunque en un principio parece intuitivo dividir la actualización de las coordenadas de manera que cada *worker* actualice las de un conjunto de partículas (es decir, las coordenadas x, y, z de cada una de las partículas), siguiendo el razonamiento anterior esto no es necesario. De hecho, teniendo en cuenta que el lenguaje Fortran almacena las matrices por columnas, es más sencillo que la distribución del trabajo se haga siguiendo las columnas de las matrices.

Por lo tanto, la división del trabajo para la suma de las matrices se hará siguiendo cada matriz por columnas, y cada *worker* manipulará un conjunto de elementos de una de las columnas; o en el caso que la columna llegue a su fin, dicho *worker* trabajará con parte de los elementos del final de una columna y parte de los del inicio de la siguiente. Es decir, el proceso se podría imaginar como si desplegáramos cada matriz de dimensión $N \times 3$ en un vector de dimensión $3N$, de manera que se colocan las columnas una tras otra; y es este vector el que se divide entre los *workers*.

En la práctica, cada *worker* recibirá un conjunto de vectores de dimensión menor $N_{\text{local}} < 3N$ (que en adelante llamaremos *vectores locales*) que contendrán las componentes de las posiciones, velocidades y fuerzas con las que deba trabajar, es decir, que recibirá tres de estos vectores locales. Una vez realizadas las operaciones pertinentes, se debe hacer una comunicación para reunir todas estas piezas de la matriz global, lo cual se hace mediante el comando *MPI_ALLGATHERV()*. A esta subrutina se le debe indicar cada *worker* que elementos de la matriz original ha modificado, y devuelve a cada uno de los *workers* la matriz global de posiciones/velocidades. Además, permite asignar a cada uno de los *workers* una cantidad diferente de elementos con los que trabajar, que será necesario ya que en general la cantidad de entradas de la matriz no será divisible por el número de procesadores.

Lo primero que se debe hacer es asignar con qué vectores locales trabajará cada uno de los *workers*, información que se creará y almacenará al inicio del programa principal, y la subrutina lo tomará como entrada. En cuanto a la propia subrutina, los pasos a seguir son los siguientes:

- En un primer paso, cada *worker* genera sus vectores locales a partir de las matrices globales que tiene como entrada.
- Una vez se tienen los vectores locales, se procede a calcular las posiciones locales nuevas y las velocidades locales provisionales.
- El siguiente paso es compartir las posiciones locales de cada *worker* para que cada uno tenga toda la matriz de posiciones actualizada.
- Se procede al cálculo de las fuerzas, y tras el cada uno de los *workers* conocerá la nueva matriz de fuerzas.
- El último paso es actualizar las velocidades provisionales a partir de estas nuevas fuerzas, para finalizar compartiendo de nuevo las velocidades con todos los *workers*.

Algorithm 5 Paso del algoritmo de integración en paralelo.

```
1: procedure VELOCITY VERLET( $\{\mathbf{r}(t)\}$ ,  $\{\mathbf{v}(t)\}$ ,  $\{\mathbf{F}(t)\}$ )
2:   Construir los vectores locales  $\mathbf{r}_{local}(t)$ ,  $\mathbf{v}_{local}(t)$ 
3:   for  $i = 1, N_{local}$  do:
4:      $\mathbf{r}_{local}(t + \Delta t) \leftarrow$  actualización de posiciones locales.
5:      $\mathbf{v}_{local}(t + \Delta t) \leftarrow$  actualización de velocidades locales (provisionales).
6:   end for
7:   Comunicar las posiciones:
8:    $\{\mathbf{r}(t + \Delta t)\} \leftarrow$  call MPI_GATHERV ( $\mathbf{r}_{local}(t + \Delta t)$ )
9:   Calculo de fuerzas:
10:   $\{\mathbf{F}(t + \Delta t)\} \leftarrow$  call forces ( $\mathbf{r}(t + \Delta t)$ )
11:  for  $i = 1, N_{local}$  do:
12:     $\mathbf{v}_{local}(t + \Delta t) \leftarrow$  actualización de velocidades locales.
13:  end for
14:  Comunicar las velocidades:
15:   $v(t + \Delta t) \leftarrow$  call MPI_GATHERV ( $v_{local}$ )
```

4.2.2. Termostato

Para el termostato se han considerado distintas alternativas de paralelización. Cómo se ha explicado anteriormente, en el termostato existe una cierta probabilidad de interacción entre el baño termico y las partículas. Esta probabilidad debe ser baja para no crear un ruido considerable en las simulaciones. En nuestro caso se ha escogido 0.1 de probabilidad. Esto sugiere la idea de que el mecanismo de paralelización mas optimo podria ser dividir el numero de atomos entre el numero de procesadores y que cada procesador recorriera los atomos que le pertocan, modificando con una probabilidad de 0.1 las velocidades acorde a una distribucion gaussiana y comunicando unicamente los resultados de las velocidades que han sido cambiadas al resto de procesadores. Este algoritmo requiere la utilizacion de un bucle de envios no bloqueantes mediante *IBSEND()* y un bucle de recibos bloqueantes *RECV()*. Este algoritmo ha sido programado pero ha sido descartado debido a su mala escalabilidad: Cuando el sistema contiene 10000 partículas, se deben hacer del orden de 1000·3 comunicaciones individuales entre procesadores, y cuando el numero de procesadores o el de partículas es incrementado el algoritmo se hace muy lento.

Por este motivo se ha optado finalmente por utilizar la estructura de array y la division de trabajo local creada para la subrutina integracion. De esta manera se evitan comunicaciones extra y las dos subrutinas se pueden unificar en una sola que comparte la misma particion y comunicacion. Es decir: Cada vez que se llama a la subrutina del termostato cada uno de los workers recorre su parte correspondiente al vector $3N$ dimensional y modifica las velocidades acorde con una probabilidad 0.1. Despues se informa al resto de workers con la subrutina MPI *MPI_ALLGATHERV()*. La escalabilidad del algoritmo ha sido comparada con el descrito anteriormente y la mejora es considerable.

5. Condiciones periódicas de contorno

Las condiciones periódicas de contorno que se usan en nuestro caso, la simulación de un sistema NVT, nos mantienen dentro de nuestra caja en el espacio un numero constante de partículas. Para poder entender este concepto hay que pensar que las paredes de nuestra caja estan delimitadas con cajas identicas, así si una partícula sale por una pared entra por la pared orientada igual pero que esta en el lado opuesto de la caja, consiguiendo así tener el mismo numero de partículas

dentro de nuestra caja durante toda la simulación (así mantenemos el volumen y la densidad constantes también durante todo el experimento). Con el objetivo de ilustrar este concepto se añade la siguiente figura 1 en este documento.

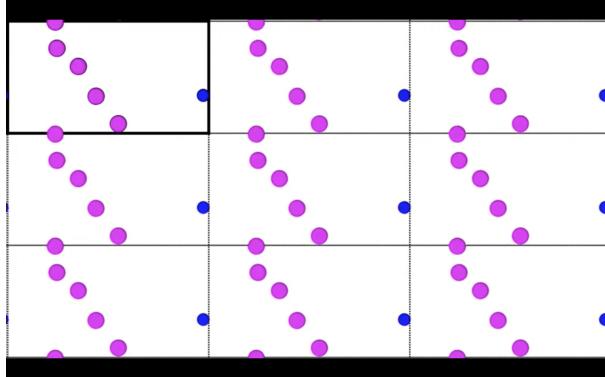


Figura 1: Ejemplo de condiciones periódicas de contorno.

5.1. Programación en serie

La aplicación de las condiciones periódicas de contorno en fortran 90 se implementan de 2 formas posibles dependiendo de como definamos el rango de posiciones de los átomos. Estos 2 tipos de rangos posibles son de $[0, L]$ (nuestro caso donde todas las posiciones son reales positivos) o de $[-L/2, L/2]$ (el caso en que el centro de la caja este en $[0, 0, 0]$), donde la variable L es la longitud de la aresta del cubo que contiene nuestras partículas. Como se puede ver en la formula 9 usamos una comanda distinta dependiendo de que definicion de rango usemos.

$$X(i, j) = \begin{cases} X(i, j) - \text{floor}(X(i, j)/L)L, [0, L] \\ X(i, j) - \text{nint}(X(i, j)/L)L, [-L/2, L/2] \end{cases} \quad (9)$$

En la formula anterior X es la matriz de posiciones de dimension $n_{atomos} \times 3$, donde la i es el atomo que estamos comprobando si esta dentro de la caja y j es la coordenada x,y o z.

Entonces para la implementación, hacemos un doble bucle para que pase por todas las partículas en un contador y en el otro por todas las coordenadas.

Algorithm 6 Condiciones periódicas de contorno em serie.

```

1: procedure PBC_SERIE( $X(n_{atomos}, 3)$ ,  $L$ )
2:   for  $i = 1, N$  do:
3:     for  $j = 1, 3$  do:
4:        $X(i, j) = X(i, j) - \text{floor}(X(i, j)/L)L$ 
5:     end for
6:   end for

```

5.2. Programación en paralelo

En el momento de la implementación en paralelo usamos la comanda **CALL MPI_ALLGATHERV**. Pero para poder usarla no se puede reaprovechar el bucle anterior porque fortran lee las matrices por columnas. Por la razón anterior tuvimos que pensar otro metodo para pasar por todas coordenadas de todos los atomos dependiendo del "task_id" del procesador. Así que cambiamos

el bucle en función de unas variables nuevas definidas en el main. Para entender esto hay que ver X como un vector de longitud $3 \times n_{\text{atomos}}$ donde las primeras n_{atomos} posiciones son la primera columna, las segundas la segunda columna y lo mismo para la tercera. Entonces la variable *displs* almacena dependiendo del *task_id* en qué sitio de la matriz empieza si fuese un vector, la variable *counts* dice a cada procesador qué longitud de la matriz tiene que recorrer desde allí. Con estas 2 variables ajustamos un poco el bucle para conseguir que pase por todos almacenandolos en una variable local (un vector) y luego cada procesador manda su vector con la comanda MPI mencionada arriba que se encarga de juntar los trozos de la matriz posición y mandar toda la información a todos los procesadores. Así las subrutinas de fuerzas pueden usar la información de matriz actualizada ya que necesita todas las posiciones de todos los átomos.

Algorithm 7 Condiciones periódicas de contorno en paralelo.

```

1: procedure PBC_PARALELO( $X(n_{\text{atomos}}, 3)$ ,  $L, \text{displs}(\text{numproc}), \text{counts}(\text{numproc})$ )
2:    $\text{my\_N\_elem} = \text{counts}(\text{taskid} + 1)$ 
3:    $kk = 1 + (\text{displs}(\text{taskid} + 1) + 1)/n_{\text{atomos}}$ 
4:    $jj = \text{displs}(\text{taskid} + 1) + 1 - (kk - 1) * n_{\text{atomos}}$ 
5:   for  $ii = 1, \text{my\_N\_elem}$  do:
6:      $\text{local\_X}(ii) = X(jj, kk) - \text{floor}(X(jj, kk)/L)L$ 
7:     if  $jj == n_{\text{atomos}}$  then:
8:        $jj = 0$ 
9:        $kk = kk + 1$ 
10:    end if
11:     $jj = jj + 1$ 
12:  end for
13:  call MPI_ALLGATHERV( $\text{local\_X}, \text{my\_N\_elem}$ , MPI_REAL, &
14:   $X, \text{counts}, \text{displs}$ , MPI_REAL, MPI_COMM_WORLD, ierror)

```

Esta es la subrutina para hacer el ejercicio de parallelizar aunque luego juntamos esta subrutina, la del termostato y el integrador para que evitar comunicaciones en cada subrutina. Con esto solo se comunican una vez y se agiliza el programa.

6. Función de distribución radial

En la colectividad canónica se considera un sistema de N átomos en un volumen V a temperatura T (ensamble $\{N, V, T\}$). En este marco, la probabilidad de encontrar cada átomo i en $d\vec{r}_i$ en la posición \vec{r}_i está determinada según:

$$P^{(N)}(\vec{r}_1, \dots, \vec{r}_N) d\vec{r}_1 \dots d\vec{r}_N = \frac{e^{-\beta U_N} d\vec{r}_1 \dots d\vec{r}_N}{Q_N} \quad (10)$$

Donde Q_N es la función de partición del sistema y U_N la energía interna. Dado que la probabilidad de encontrar n átomos en $d\vec{r}_i$ en la posición \vec{r}_i es independiente de la posición de los otros $N - n$ átomos, se pueden integrar las coordenadas de los átomos $n + 1$ hasta N . De este modo, se pueden definir las funciones de distribución de n partículas como la probabilidad de que cualquier átomo esté en $d\vec{r}_1$ en la posición \vec{r}_1 hasta en $d\vec{r}_n$ en la posición \vec{r}_n independientemente de la posición del resto de partículas:

$$\rho^{(n)}(\vec{r}_1, \dots, \vec{r}_n) = \frac{N!}{(N - n)!} \frac{\int \dots \int e^{-\beta U_N} d\vec{r}_{n+1} \dots d\vec{r}_N}{Q_N} \quad (11)$$

Por otro lado, la función de correlación de n partículas $g^{(n)}$ se define como:

$$g^{(n)}(\vec{r}_1 \dots \vec{r}_n) = \frac{\rho^{(n)}(\vec{r}_1, \dots, \vec{r}_n)}{\prod_{i=1}^n \rho^{(i)}(\vec{r}_i)} \quad (12)$$

Esta función da información de como se encuentra correlacionado un sistema. Para un caso homogéneo:

$$\rho^{(n)}(\vec{r}_1, \dots, \vec{r}_n) = \rho^n g^{(n)}(\vec{r}_1 \dots \vec{r}_n) \quad (13)$$

donde ρ es la densidad del sistema.

En esta sección se estudia concretamente la función de distribución radial, que se define como la función de correlación de pares de partículas en el sistema $g^{(2)}(\vec{r}_1, \vec{r}_2)$. Dado que el sistema que se está estudiando es, a priori, un sistema homogéneo e isotrópico la función de distribución radial se puede definir en función del módulo de la distancia entre partículas $g^{(2)}(\vec{r}_1, \vec{r}_2) = g^{(2)}(\|\vec{r}_1 - \vec{r}_2\|) \equiv g(r)$ e integrando sobre todo el espacio se puede encontrar la expresión:

$$g(r) \simeq \frac{n(r, dr)}{\rho \frac{4\pi}{3} [(r + dr)^3 - r^3]} \quad (14)$$

donde $n(r, dr)$ se define como el número de partículas que se encuentran en una cascara esférica de grueso dr a una cierta distancia r de una partícula de referencia. Esta función de distribución radial nos permite conocer la probabilidad de encontrar una partícula a una distancia r respecto a una partícula de referencia en un sistema. Se considera que es una importante propiedad de estudio en las simulaciones de dinámica molecular dado que nos da acceso a caracterizar la estructura local del fluido que se pretende simular y se puede comparar con datos experimentales.

6.1. Programación en serie

En el código de dinámica molecular de este proyecto se implementó una subrutina para poder calcular esta propiedad. Es importante destacar que para poder obtener un resultado representativo se calculó la función de distribución radial para numerosos *snapshots* de la simulación y se promedió el resultado a lo largo de toda la dinámica. Además, para reducir los efectos de las correlaciones entre pasos de tiempo consecutivos, solamente se realizaron medidas cada un cierto número de pasos `outg = 500`. La subrutina recibía las posiciones de las N partículas del sistema, el radio máximo hasta el cual se quería calcular la función de distribución radial `rmax` y el número de puntos `nbox` en los cuales se quería realizar el cálculo de $g(r)$ en el intervalo $[0, \text{rmax}]$. Como *output*, se extrae un histograma que indica para cada punto r el número de parejas de partículas que se encuentran a esa distancia de separación. El peso computacional de esta parte del programa es del orden de N^2 (más concretamente $N(N - 1)/2$) debido a que se deben recorrer, para cada *snapshot* en los que se realiza una medida, todas las posibles parejas de partículas. La subrutina internamente funciona de la siguiente manera:

Una vez realizado todo el proceso en un numero `conteg` de *snapshots*, para obtener la función de distribución radial se recorren las `nbox` donde se ha realizado el cálculo del histograma y se realiza el cálculo de la ecuación (14) conociendo que $n(r)$ es el histograma en la posición r y, además, dividiendo por el número de *snapshots*.

6.2. Programación en paralelo

Para la implementación en paralelo de este código, en primer lugar se construyó un vector de conexiones que guarda las $N(N - 1)/2$ parejas de partículas $\{i, j\}$. La división del trabajo

Algorithm 8 Cálculo del histograma de $g(r)$ para un *snapshot*.

```
1: procedure HISTOGRAMA( $r(nparticles)$ ,  $Histo(nbox)$ )
2:   for  $i = 1, nparticles-1$  do:
3:     for  $j = i+1, nparticles$  do:
4:        $\vec{r}_{ij} = \vec{r}(i) - \vec{r}(j)$ 
5:       Se recorren las imágenes de estos pares de partículas en la caja actual y las vecinas
       más cercanas:
6:       for  $Nx = -1, 1$  do:
7:         for  $Ny = -1, 1$  do:
8:           for  $Nz = -1, 1$  do:
9:              $\vec{r}_{ij} = \vec{r}_{ij} + L \cdot (Nx, Ny, Nz)$ 
10:            Se calcula el módulo de la distancia:
11:             $r_2 = \sqrt{r_x^2 + r_y^2 + r_z^2}$ 
12:            Se estudia en que caja  $nbox$  del intervalo  $[0, r_{max}]$  cae conociendo la
       medida  $dr$  de la caja:
13:             $caja = \text{int}(r_2/dr) + 1$ 
14:            if ( $caja < nbox$ ).and.( $caja > 0$ ) then
15:              Se suma 2 al histograma en esa caja (hay 2 parejas  $r_{ij}$  y  $r_{ji}$ ):
16:               $Histo(caja) = Histo(caja) + 2$ 
17:            end if
18:          end for
19:        end for
20:      end for
21:    end for
22:  end for
23: end procedure
```

entre los distintos procesadores se realiza de manera que cada *worker* estudia la función de distribución radial de $N(N - 1)/(2\text{numproc})$ parejas de partículas (se distribuye el vector de conexiones entre el número de *workers* y se recorre esa porción de vector dentro de la subrutina en lugar del doble bucle de las N partículas). De este modo, cada *worker* guardará un histograma para su porción de pares de partículas. Es importante observar que esta magnitud no necesita realizar comunicaciones entre los distintos procesadores en cada *snapshot* ya que no se necesitan las posiciones del resto de partículas que no considera el procesador ni el valor del histograma que se obtiene para el resto, en otras palabras, es como si cada procesador estudiara la función de distribución radial de sistemas independientes con $N(N - 1)/(2\text{numproc})$ parejas de partículas. Por lo tanto, cada procesador irá acumulando el histograma de todos los *snapshots* en los que se realiza el cálculo. Por último, una vez realizada toda la simulación de dinámica molecular, para realizar el cálculo de la función de distribución radial $g(r)$ según la ecuación (14) es necesaria la comunicación entre procesadores del valor de los distintos histogramas. Para ello se usó el comando `MPI_REDUCE` para sumar los `numproc` histogramas independientes y mandarlos a uno de los procesadores (el de `taskid = 0` en este caso) que realiza el cálculo de la $g(r)$ y escribe los datos en un fichero de salida.

7. Desplazamiento cuadrático medio

El desplazamiento cuadrático medio (MSD o $\langle \Delta r^2(t) \rangle$) es una medida de la desviación de la posición de una partícula respecto a una posición fijada a lo largo de el tiempo. Comúnmente esta propiedad se estudia para determinar si el movimiento de una partícula en un sistema es de carácter difusivo o si se debe a fuerzas advectivas que contribuyen al desplazamiento del *bulk* del sistema. Si el movimiento es de tipo difusivo o Browniano se debe observar que esta magnitud se comporta linealmente con el tiempo según la relación de Einstein (en tres dimensiones en este caso): $\langle \Delta r^2(t) \rangle = 6Dt$ donde D es el coeficiente de difusión del sistema.

7.1. Programación en serie

Dado que en una simulación el tiempo se encuentra discreteado por N_{steps} pasos de intervalos de tiempo de valor dt , se puede estudiar para cada *snapshot* el promedio de las correlaciones de las posiciones $r(t)$ de las N partículas del sistema respecto una posición fijada $r(t = 0)$:

$$\langle \Delta r^2(t) \rangle = \frac{1}{N_{steps}} \sum_{i=1}^{N_{steps}} \| \vec{r}_i(t) - \vec{r}_i(t=0) \|^2 \longrightarrow 6Dt \quad (15)$$

De este modo la subrutina procede según:

Finalmente, para este *snapshot* que corresponde al instante de tiempo t se guarda en un fichero el valor del *MSD*. En el código, el cálculo del *MSD*, al igual que el de la función de distribución radial, no se realiza cada paso de tiempo sino que se realizaron medidas cada un cierto número de pasos `outg = 500`.

7.2. Programación en paralelo

El coste computacional de esta subrutina es de orden N ya que solamente se realiza un bucle del número de partículas. Para la implementación en paralelo de esta sección se dividieron las coordenadas de las N partículas entre el numero de procesadores para que cada *worker* calculara el *MSD* de un sistema de $N/\text{numproc}$ partículas. En este caso, para cada *snapshot* es necesaria una comunicación de la información de los distintos procesadores para poder guardar en el fichero de datos el valor total del *MSD* del sistema. Para ello, después de obtener la

Algorithm 9 Cálculo del desplazamiento cuadrático medio para un instante t de tiempo.

```
1: procedure MEAN_SQUARE_DISPLACEMENT( $r_0(nparticles), r_t(nparticles)$ ,  $MSD$ )
2:   for  $i = 1$ ,  $nparticles$  do:
3:      $\vec{R}_i(t) = \vec{r}_i(t) - \vec{r}_i(0)$ 
4:     Se aplican condiciones periódicas de contorno
5:      $\vec{R}_i(t) = \vec{R}_i(t) - L \cdot nint(\vec{R}_i(t)/L)$ 
6:     Se suma el módulo del vector al  $MSD$ 
7:      $MSD = MSD + \|\vec{R}_i(t)\|$ 
8:   end for
9:   Se calcula el  $MSD$  promediando por el número de partículas
10:   $MSD = MSD/nparticles$ 
11: end procedure
```

suma de los desplazamientos cuadráticos de cada subsistema de cada *worker*, se usó la subrutina `MPI_REDUCE` para sumar los distintos valores obtenidos y mandarlos a uno de los procesadores (el de `taskid = 0` en este caso) que se encarga de escribir el valor obtenido dividido entre el número de partículas en la unidad de memoria.

8. Main

En el main.f90 es donde se unifica y controla el flujo del programa y donde las subrutinas son llamadas. El programa requiere de un input en formato .txt en el cual se introducen una serie de parámetros que se pueden encontrar en el readme.txt para controlar la simulación. Las unidades del input.txt han sido escogidas en un sistema de unidades que permitan trabajar sin salirse de la precisión del programa. Una vez los parámetros del gas-líquido a simular son introducidos, una subrutina dentro del main.f90 se encarga de pasar las magnitudes a unidades reducidas para que los cálculos y ecuaciones sean más cómodos de manejar. Una vez el calculo ha terminado, los resultados son pasados otra vez a las mismas unidades del input.txt. La estructura del main es muy sencilla y muy similar tanto en serie como en paralelo.

8.1. Programación en serie

Inicialmente se inicializan las posiciones siguiendo una estructura cristalográfica FCC y las velocidades se inicializan a una temperatura elevada, usando las subrutinas `INITIALIZE_R` y `INITIALIZE_V`. Seguidamente, se inicia una simulación de dinámica molecular a una temperatura mucho mayor que la del baño para asegurarse que el sistema ha fundido su estructura inicial.

Un bucle de dinámica molecular consiste en calcular las fuerzas a través de `FORCE`, hacer un paso de Verlet con `V_VERLET_STEP`, reintroducir las partículas que han salido del sistema con `PBC`, y llamar al termostato de Andersen `THERMOSTAT`.

Una vez se ha fundido el sistema se inicializa otra serie de bucles a la temperatura introducida en el input para llegar a *samplear* un espacio de configuraciones asociado a la temperatura deseada.

Finalmente empieza un tercer bucle con la misma estructura, a la temperatura del baño en el cual se calculan, ademas, las magnitudes físicas importantes como el *mean square displacement* `MSDISPLACEMENT`, la función de distribución radial `GR` y las energías. Todas las magnitudes son escritas a medida que se van calculando, excepto la función de distribución radial, que se acumula y se promedia al final de la simulación.

8.2. Programación en paralelo

La estructura general del main en paralelo es exactamente la misma que la estructura en serie, pero con algunos cambios para garantizar la eficacia del algoritmo en paralelo. En el main.f90 se reparte el trabajo de los procesadores entre el numero de átomos. Hay diferentes maneras de repartir el trabajo en función de la subrutina paralelizada. Para esto se utilizan las diferentes tablas que se inicializan. La tabla 1 (*TABLE_INDEX1*) asigna a cada procesador un cierto numero de átomos. La tabla 2 (*TABLE_INDEX2*) asigna a cada procesador un cierto numero de pares de interacción para el calculo de las fuerzas. Por ultimo, existe una tabla para dividir el trabajo en la manera en la que se ha explicado en los apartados de integración y termostato.

Para reducir el coste computacional de las comunicaciones las subrutinas de integración, condiciones periódicas y termostato han sido unificadas en una sola *V_VERLET_PBC_THERM*. En el tercer bucle, después de la inicialización del sistema, se inician las medidas de las magnitudes físicas en subrutinas que también han sido paralelizadas y cuando la simulación acaba el procesador asociado al numero 0 escribe los resultados.

9. Promedios estadísticos *Binning*

El análisis de resultados mediante promedios estadísticos se realiza de manera externa al programa de dinámica molecular para no influir en la velocidad del código y para trabajar de manera más modular. De este modo el programa principal de MD escribe en un archivo *data_EK_EP_T_P.dat* los valores de tiempo, energía cinética, energía potencial, temperatura y presión para todos los *snapshots* de la simulación para ser analizados a posteriori.

El método *Binning* se usa para obtener promedios y aproximaciones de errores estadísticos de variables cuyos valores se encuentran correlacionados. En este caso, dado que la simulación de MD es secuencial, las configuraciones de cada instante de tiempo que dan los valores del estado macroscópico de energía cinética, energía potencial, temperatura y presión; están fuertemente correlacionadas, es decir, para poder considerar que dos microestados correspondientes a dos *snapshots* están descorrelacionados deben pasar un número determinado de pasos que se define como el tiempo de autocorrelación. El método *Binning* se basa en realizar promedios estadísticos de variables por bloques de medida m de manera recursiva. En este caso, el aumento de la medida de los bloques m es de un factor dos, de manera que de cada conjunto de N valores para los cuales se estudia el promedio estadístico, se agrupan al siguiente paso del método en $N/2$ bloques que promedian los valores anteriores dos a dos, y así recursivamente hasta un número determinado de iteraciones. De este modo se puede conocer en función de la medida de los bloques el valor del promedio estadístico y de la varianza de las variables que se estudian. Para poder sacar aproximaciones del tiempo de autocorrelación y del error estadístico de los datos descorrelacionados se puede realizar un *fitting* no lineal de las varianzas en función de la medida m de los bloques según una función exponencial:

$$f(\sigma) = a - b e^{-\sigma/\tau} \quad (16)$$

donde a es el valor del error estadístico y τ el tiempo de autocorrelación.

La implementación de este código se realiza mediante una subrutina que tiene como parámetros de entrada el número final de bloques hasta los cuales se va hacer el proceso de Binning (se va a repetir el promedio de conjuntos de datos dos a dos de manera recursiva hasta obtener este número de bloques); el archivo de entrada de los cuales se lee el valor de energía cinética, energía potencial, temperatura y presión para todos los pasos de tiempo de la simulación; el archivo de salida en los cuales se escribe para estas variables y para la energía mecánica el valor de su

promedio estadístico y de su varianza en función de la medida de los bloques m ; y el número de pasos iniciales de la simulación que se omiten. Esta omisión de los primeros pasos se realiza dado que los instantes iniciales de la simulación sirven para termalizar y llegar a un estado de equilibrio dinámico del sistema y, por lo tanto, no deben considerarse para realizar cálculos de promedios estadísticos. Por lo tanto, es una buena estrategia representar gráficamente la evolución de las variables en función del tiempo para estudiar en qué instante entran en un régimen estacionario con pequeñas fluctuaciones debidas a las interacciones del sistema en equilibrio. Es importante considerar que para iniciar la simulación se parte de una inicialización de las posiciones de las partículas en una red cristalina que se derrite realizando una simulación de MD a muy alta temperatura. Las medidas de la simulación se realizan cuando las configuraciones del sistema ya se encuentran lejos del estado cristalino y se introduce el efecto del termostato de Andersen para fijar la temperatura T . Por lo tanto, la rapidez con la que el sistema entra en el régimen estacionario y, por lo tanto, la cantidad de pasos que no se consideran en el proceso de *Binning* viene fuertemente determinado por la efectividad del termostato que se implementa en la dinámica.

Esta parte de análisis del código no se paralelizó dado que para ficheros de datos de más de 10^6 pasos de tiempo este algoritmo tardaba escasos instantes a sacar los resultados y, además, la mayor parte del tiempo computacional proviene de la lectura del fichero de entrada de datos (ya que la velocidad de comunicación con la memoria es mucho menor que la de procesado de datos) y este es un proceso no paralelizable ya que hay conflictos si varios *workers* acceden a la misma porción de memoria para leer ficheros y porque la lectura es siempre secuencial. Además, el coste computacional de procesado de datos de la subrutina es de como mucho del número de pasos de la simulación para cada iteración del *Binning* y solamente se realizan del orden de 10-20 iteraciones; en consecuencia, no es de vital importancia su paralelización.

10. Resultados de la dinámica molecular

En esta sección mostraremos los resultados de la dinámica molecular realizada con el programa en serie y en paralelo. Como es de esperar, los resultados son idénticos, ratificando de este modo la buena implementación del código en paralelo. En este punto, tenemos que comentar que a la hora de ejecutar el código en paralelo, notamos una pérdida de precisión en las comunicaciones. Por ello, el uso de reales de doble precisión dejó de ser algo optativo (en el programa en serie, para ambos casos, reales de precisión simple y doble, los resultados no variaban). Los cálculos se realizaron para dos sistemas diferentes: un gas de Helio y un líquido de Argón. La elección de este segundo sistema, el cual no se especifica en la descripción del proyecto de la asignatura, fue motivada por el hecho de que, para un gas de Helio en las condiciones que se piden, las interacciones entre las partículas son prácticamente nulas. Además, disponíamos los resultados del segundo sistema compuesto de Argón que mostraremos. Por lo tanto, nos fue útil para verificar el buen funcionamiento del programa tanto en serie como en paralelo.

Los parámetros usados fueron los especificados en la Tabla 1.

	Sistema 1: Helio	Sistema 2: Argón
N. partículas	500	500
Temperatura (K)	300	1200
Presión (atm)	1.7	3500
LJ: σ (\AA)	2.556	3.4
LJ: ϵ (J/mol)	84.928	998.0

Tabla 1: Parámetros utilizados para las simulaciones de los dos sistemas.

10.1. Sistema 1: gas de Helio

La Figura 2 muestra la evolución de las energías respecto al tiempo. Como vemos, todas las magnitudes fluctúan alrededor del valor de equilibrio. Esto mismo se aplica a la temperatura y la presión.

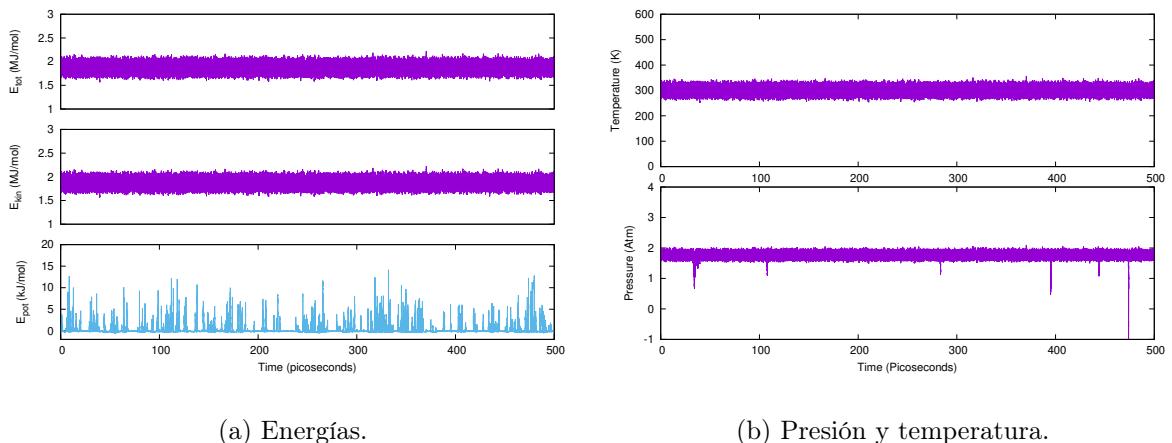


Figura 2: Resultados para el gas de Helio.

Por otro lado, en la Figura 3 se muestran la función de distribución radial y el desplazamiento cuadrático medio. Como era de esperar, el desplazamiento cuadrático medio se puede ajustar casi perfectamente a una línea recta, tal como se explicó en la sección 7, de la cual se puede deducir el coeficiente de difusión del Helio. Además, la función de distribución radial se asemeja a la de un gas: observamos un máximo alrededor del valor de σ y se vuelve prácticamente constante a partir de ahí aproximándose a uno. Esto se debe a que el Helio tiene una interacción a muy corta distancia y para distancias grandes el desorden es total. Además, el Helio en estas condiciones es casi un gas ideal, por lo que el máximo que hemos mencionado es apenas visible.

Utilizando el método binning explicado anteriormente, pudimos obtener valores medios de las magnitudes y sus correspondientes errores estadísticos y tiempos de autocorrelación. Éstos se muestran en la Tabla 2.

Como podemos observar, los valores medios de la temperatura y presión están de acuerdo con los valores fijados al inicio de la simulación. Además, tal y como se esperaba, el tiempo de autocorrelación de la energía cinética y de la temperatura son prácticamente iguales porque la temperatura se deriva directamente de la energía cinética. Por otro lado, la energía potencial muestra el máximo tiempo de autocorrelación ya que, al ser una interacción tan débil, el número de pasos que hay que realizar para conseguir muestras sin correlación es muy grande.

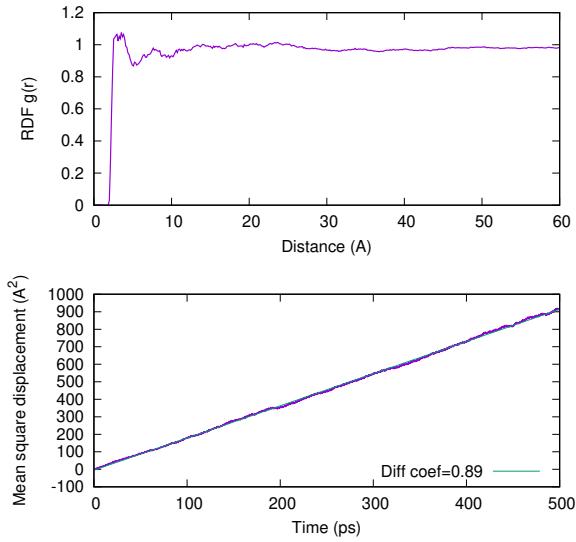


Figura 3: Función de distribución radial y desplazamiento cuadrático medio del Helio.

	Media	σ_E	τ
Energía cinética (J/mol)	1867600	300	15.7973
Energía potencial (J/mol)	23	9	463.559
Temperatura (K)	299.65	0.04	15.8035
Presión (atm)	1.7628	0.0007	199.896

Tabla 2: Valor medio, error estadístico (σ_E) y tiempo de autocorrelación τ de las diferentes magnitudes medidas para el primer sistema.

10.2. Sistema 2: líquido de Argón

En este caso simularemos un líquido de Argón, por lo tanto, esperamos que la función de distribución radial presente diferencias significativas respecto a un sistema gaseoso. En cuanto a la interpretación de los gráficos referentes a la evolución de las energías, presión y temperatura, lo dicho para el Helio es aplicable en este caso: los resultados cuadran con lo esperado.

Para la función de distribución radial se puede observar que hay un máximo más definido alrededor del valor de σ y el carácter ondulatorio se extiende más con la distancia. Es decir, el orden a distancias pequeñas es mayor, aunque a distancias mayores el desorden sigue dominando.

En la Tabla 3 se muestran los resultados estadísticos obtenidos mediante la técnica del binning.

	Media	σ_E	τ
Energía cinética (J/mol)	7475000	6000	2.73349
Energía potencial (J/mol)	-220000	20000	153.291
Temperatura (K)	1199	1	2.73277
Presión (atm)	3530	70	235.327

Tabla 3: Valor medio, error estadístico (σ_E) y tiempo de autocorrelación τ de las diferentes magnitudes medidas para el segundo sistema.

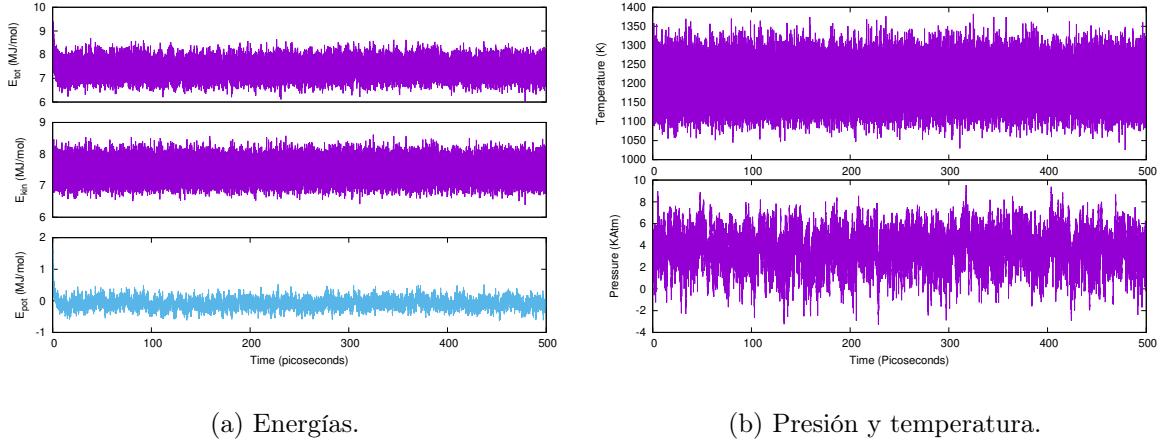


Figura 4: Resultados para el gas de Argón.

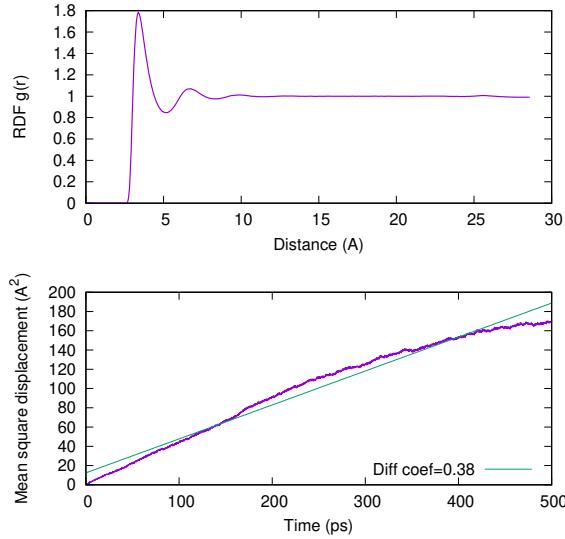


Figura 5: Función de distribución radial y desplazamiento cuadrático medio del Argón.

Al igual que en el caso anterior los valores medios se corresponden a lo esperado.

Todos los resultados mostrados en esta sección se han realizado con el código paralelizado porque esto nos permite hacer los cálculos con más partículas siendo el coste computacional mucho menor. En cualquier caso, en el repositorio de GitHub se pueden encontrar los resultados para el Helio hechos en estas mismas condiciones con el código en serie, siendo los resultados compatibles con los aquí mostrados. Se debe tener en mente que aquí los cálculos están hechos con 500 partículas y en el caso en serie tan solo se utilizaron 108.

11. Estudio de escalabilidad

Antes de hacer un estudio de escalabilidad del programa completo, se decidió estudiar por separado las subrutina de integración de las ecuaciones del movimiento, termostato y condiciones periódicas de contorno. La razón es que estas tres subrutinas están estrechamente ligadas entre sí, y se quiso comprobar cómo afecta el paralelizar cada una de ellas por su cuenta. El problema de

paralelizar cada una de ellas independientemente del resto es que se deben realizar muchas más comunicaciones entre los diferentes procesos. Teniendo en cuenta que ninguna de estas subrutinas conlleva un gran coste computacional, se prevé que al aumentar el número de procesadores sea más el tiempo que se pierde en las comunicaciones que el que se gana al paralelizar.

Para comprobar si esto es cierto, se hizo una prueba en la que se trabaja con un sistema de $N = 400000$ partículas en el que tan solo se implementó estas tres subrutinas. Por una parte se emplearon las tres subrutinas independientes mencionadas anteriormente, y por otro se hizo una implementación en la que se combinan las tres en una única subrutina, minimizando así el número de comunicaciones. En la Figura 6 se muestran las diferencias en cuanto a las escalabilidad de ambas implementaciones. Tal y como se puede observar, con hasta 4 procesadores, el tiempo de cálculo es similar en ambos casos. Sin embargo, a partir de ese punto, el tiempo de cálculo de la implementación de las 3 subrutinas independientes empieza a aumentar, mientras que en el segundo caso el tiempo se mantiene más o menos constante.

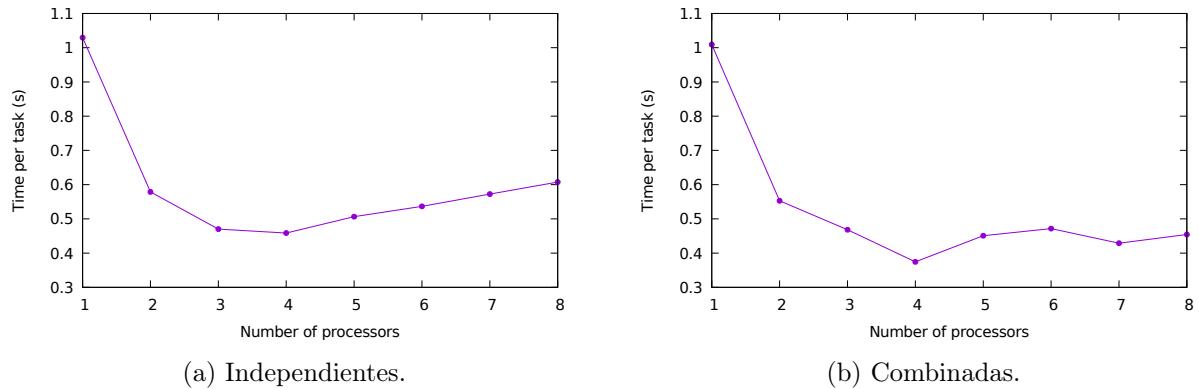


Figura 6: Tiempo medio empleado por cada llamada a las subrutinas para diferentes cantidades de procesadores para (a) las subrutinas independientes y (b) la subrutina unificada.

A la vista de este primer resultado, se decidió que era mejor optar por la implementación en la que se combinan las tres subrutinas, por lo que se trabajó con esa opción.

A la hora de hacer el análisis de la escalabilidad de la paralelización, se debe elegir un tamaño para el sistema (es decir, fijar el número de partículas) y la cantidad de pasos que se darán en la simulación; y repetir el cálculo aumentando cada vez la cantidad de procesadores utilizados. En nuestro caso, se ha decidido hacer dos pruebas, una con un sistema de $N_{par} = 108$ partículas y otra con $N_{par} = 500$ partículas, haciendo en ambos casos $N_{steps} = 10^7$ pasos en la simulación. Este estudio se ha realizado en el servidor del *BSC*, lo que no permite trabajar con más procesadores de los que podríamos usar en un ordenador corriente. Así, se ha hecho el cálculo para una cantidad entre $N_{proc} = 1$ y 32. Cabe destacar que para comprobar la escalabilidad del código, los cálculos se han ejecutado omitiendo las partes en las que se escriben los resultados en archivos, ya que esto ralentiza mucho la ejecución.

Para estudiar la eficiencia del código, se definen las siguientes magnitudes:

- *Speedup*. Se define como el ratio entre el tiempo de cálculo utilizando un procesador y el tiempo utilizando P procesadores:

$$\text{speedup}(P) = \frac{\text{Tiempo}(P=1)}{\text{Tiempo}(P)}$$

En el caso ideal, esta relación debería ser lineal.

- *Eficiencia*. Se define como la división del *speedup* y el número de procesadores, por lo que idealmente debería tomar el valor 1 para cualquier cantidad de procesadores.

En primer lugar, en la Figura 7 se muestran los resultados obtenidos para el sistema con un número menor de partículas. De las figuras mostradas, se deduce rápidamente que empezando por un procesador, al ir aumentando la cantidad el comportamiento no se aleja demasiado de la expectativa ideal: el tiempo de ejecución se reduce drásticamente en los primeros casos. Sin embargo, al llegar alrededor de 12-16 procesadores, el tiempo de ejecución deja de bajar más y el *speedup* llega a su límite (alrededor de 7). Por lo tanto, para una cantidad mayor de procesadores la eficiencia baja mucho, ya que el aumentar la cantidad de procesadores no da lugar a una mejora sustancial en cuanto al tiempo de ejecución.

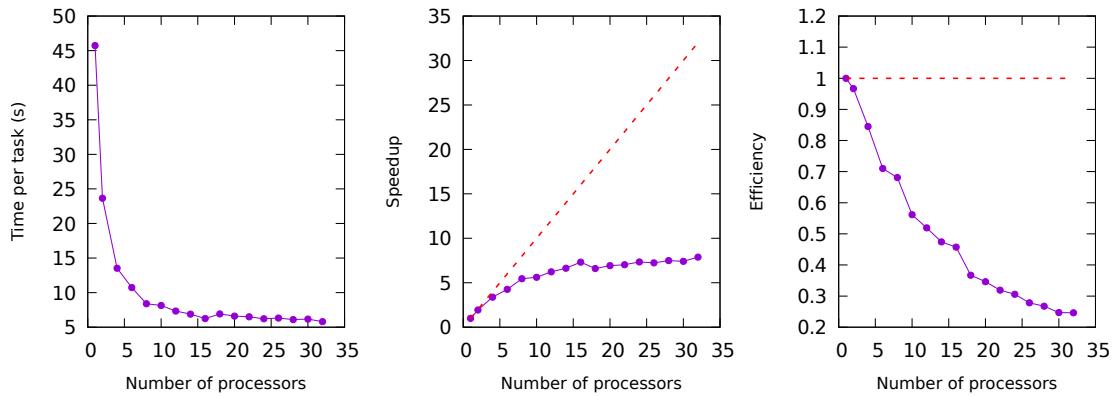


Figura 7: Tiempo de cálculo, *speedup* y eficiencia en función del número de procesadores para el sistema con 108 partículas.

Por otro lado, en la Figura 8 se muestran los resultados correspondientes al caso en el que el número de partículas en el sistema es mayor. En esta figura se puede observar una mejoría notable respecto al resultado anterior. El tiempo de ejecución continúa disminuyendo al aumentar el número de procesadores para un número de procesadores mucho mayor que en el caso anterior. De hecho, en la representación del *speedup* se ve claramente que en este caso es mucho mayor, y que se asemeja mucho más al caso ideal. Lo mismo ocurre con la eficiencia, que es mucho mayor que en el caso del sistema menor.

Lo más adecuado habría sido continuar aumentando el número de procesadores aún más para comprobar hasta qué cantidad de procesadores sigue siendo rentable. Sin embargo, debido a problemas con el servidor no se ha conseguido utilizar más de 32 procesadores.

En cualquier caso, mediante este estudio hemos comprobado que el código en paralelo implementado funciona correctamente. Por una parte, se ha verificado que el tiempo de cálculo para un problema de tamaño fijo disminuye al incrementar el número de procesadores, aunque por supuesto, tiene su límite. Por otro lado, también se ha visto como afecta el tamaño del problema a la eficiencia obtenida de la parallelización: cuanto mayor es el sistema más cantidad de operaciones se deben realizar (sobre todo el cálculo de la interacción a parejas), por lo que la división del trabajo se nota aún más.

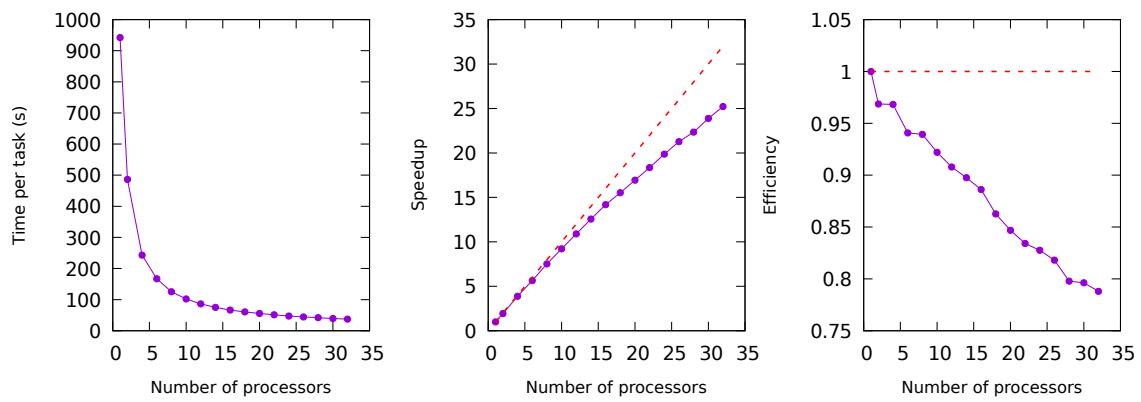


Figura 8: Tiempo de cálculo, *speedup* y eficiencia en función del número de procesadores para el sistema con 500 partículas.