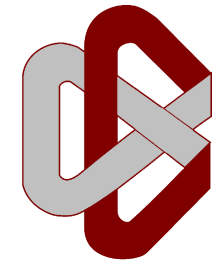

Centro de Investigación en Matemáticas, A.C.

CIMAT

Complejidad Lineal y Alta Escalabilidad Para Una Simulación En Paralelo Del Método de Monte Carlo

T E S I S

Que para obtener el grado de :

Maestro en Ciencias con Especialidad en Computación y Matemáticas Industriales

P r e s e n t a :

Guillermo Amaro Rico

Codirectores de Tesis:

Dr. Salvador Botello Rionda
Dr. Mauricio Carrillo Tripp

Guanajuato, Gto. Diciembre de 2013

CIMAT
CENTRO DE INVESTIGACION EN MATEMATICAS A. C.

Centro de Investigación en Matemáticas, A.C.

Acta de Examen de Grado

Acta No.: 066

Libro No.: 003

Foja No.: 066

En la Ciudad de Guanajuato, Gto., siendo las 15:30 horas del día 2 de diciembre del año 2013, se reunieron los miembros del jurado integrado por los señores:

DR. ARTURO HERNÁNDEZ AGUIRRE (CIMAT)
DR. JOSÉ ANDRÉS CHRISTEN GRACIA (CIMAT)
DR. MAURICIO CARRILLO TRIPP (CINVESTAV-LANGEBIO)

bajo la presidencia del primero y con carácter de secretario el segundo, para proceder a efectuar el examen que para obtener el grado de

MAESTRO EN CIENCIAS
CON ESPECIALIDAD EN COMPUTACIÓN
Y MATEMÁTICAS INDUSTRIALES

Sustenta

GUILLERMO AMARO RICO

en cumplimiento con lo establecido en los reglamentos y lineamientos de estudios de posgrado del Centro de Investigación en Matemáticas, A.C., mediante la presentación de la tesis

"COMPLEJIDAD LINEAL Y ALTA ESCALABILIDAD PARA UNA SIMULACIÓN EN PARALELO DEL MÉTODO DE MONTE CARLO".

Los miembros del jurado examinaron alternadamente al (la) sustentante y después de deliberar entre sí resolvieron declararlo (a):

APROBADO

A. Hdez. Aguirre

DR. ARTURO HERNÁNDEZ AGUIRRE
Presidente

DR. JOSÉ ANDRÉS CHRISTEN GRACIA
Secretario

DR. MAURICIO CARRILLO TRIPP
Vocal

CIMAT
DIRECCION GENERAL

DR. JOSÉ ANTONIO STEPHAN DE LA PEÑA MENA
Director General

Dedicatoria:

Para esa gran señora que fue mi Abuelita...

Agradecimientos:

A mi Madre: Virginia Rico Hernández, por que siempre estuvo para darme ánimos y seguir adelante.

A mis Hermanos: Jorge y Linda, que siempre me apoyaron sin importar las condiciones.

A todas esas personas que han ayudado a que esto sea realidad, en espacial a todos mis amigos, Edward, Flor, Adrián, Carmina, Manuel, Gerardo, Flor, Judith, Gilberto, Abel, Mary, Paco, Edgar, y a todos los compañeros con los que compartí el salón de clases y los pasillos del CIMAT.

A mis Asesores: El Dr. Salvador Botello y el Dr. Mauricio Carrillo, por todo el apoyo académico y personal que siempre me brindaron. Gracias por la confianza que tuvieron en mi.

Al Consejo Nacional de Ciencia y Tecnología y al Centro de Investigación en Matemáticas A.C. por el apoyo económico recibido para la realización de mis estudios de maestría.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Simulación de Fenómenos Físicos

El desarrollo de las computadoras a partir de la década de los 50's y su aplicación a la resolución de problemas científicos, ha introducido lo que algunos han llamado una tercera metodología a la investigación científica: la simulación computacional. Este método, de carácter complementario y muchas veces alternativo a los métodos convencionales de hacer ciencia, el experimental y el teórico, ha ejercido un fuerte impacto en prácticamente todos los campos de la ciencia. El objetivo de la simulación computacional es resolver los modelos teóricos en su total complejidad, mediante la resolución numérica de las ecuaciones involucradas, haciendo uso intensivo (y extensivo) de computadoras.

En el área de la física, la simulación computacional fue introducida como una herramienta para tratar sistemas de muchos cuerpos a comienzo de la década de los 50's, con el trabajo pionero de Metrópolis. Más tarde, auspiciosos resultados iniciales obtenidos en mecánica estadística clásica, en particular en el estudio de líquidos, dieron credibilidad a la simulación computacional, extendiéndose rápidamente su uso a temas tan diversos como dinámica cuántica, física de fluidos, relatividad general, física del plasma, materia condensada, física nuclear y ciencia de materiales.

Actualmente, gracias al vertiginoso desarrollo de la tecnología de las computadoras, cuya velocidad crece aproximadamente con factor 2 cada dieciocho meses, la simulación computacional se ha constituido en una herramienta de cálculo esencial, tanto para experimentalistas como para teóricos. Mediante un buen modelo computacional no solo se pueden reproducir experimentos de laboratorio, sino que además, gracias a que se pueden variar libremente los parámetros usados, permite probar modelos teóricos existentes en rangos de parámetros imposibles de alcanzar experimentalmente por ahora, resolviendo así conflictos entre explicación teórica y observación. Un papel fundamental también lo juega hoy en día la visualización de los resultados obtenidos. No solo obtenemos datos numéricos que pueden ser contrastados con los experimentos, sino también obtenemos una imagen gráfica del proceso en cuestión.

Obtener modelos matemáticos que reflejen fielmente el funcionamiento de los fenómenos físicos, es por lo general una tarea difícil, y requiere conocer las diferentes teorías que hasta la fecha mejor los aproximan. Si abordamos el estudio de los fenómenos físicos en los estados de agregación de la materia, sólido, líquido y gaseoso, es necesario realizar un análisis a nivel atómico, lo que implica conocer al detalle las interacciones atómicas de las partículas.

Los dos métodos de simulación computacional más usados en física actualmente son el de la Dinámica Molecular, que es de carácter determinista, y el Método de Monte Carlo, que es de carácter estocástico. Ambos pueden considerarse como métodos para generar configuraciones diferentes de un sistema de partículas, es decir, realizar un muestro en el espacio fase de tal manera que cada punto que se obtenga sea compatible con las condiciones externas.

El método de la Dinámica Molecular y el método de Monte Carlo han sido empleados con éxito para simular gases, líquidos y sólidos. Los sistemas estudiados van desde cientos a miles y últimamente incluso

a cientos de miles de átomos.

Los aspectos estudiados incluyen propiedades estructurales, termodinámicas, mecánicas y cinéticas. Cabe señalar que estas técnicas de simulación tienen aplicaciones mucho más amplias que las aquí mencionadas. En particular el método de Monte Carlo se emplea con éxito en Física de Partículas, así como para calcular sistemas cuánticos.

En este trabajo abordamos la simulación de sistemas de partículas utilizando el Método de Monte Carlo apoyándonos en las tecnologías de computo en paralelo OpenMP y CUDA.

1.2. Elección del Modelo

El punto de partida para simular un sistema físico es definir con claridad el problema en cuestión: qué tipo de propiedades nos interesa estudiar, dentro de que rango de parámetros, con que precisión, y es de gran importancia determinar la infraestructura computacional con la que contamos. En función de ello debemos decidir el número de partículas en el sistema, cuáles serán las variables de control, que potencial interatómico utilizar, que tipo de promedios debemos calcular, en que ensemble conducir la simulación.

1.3. Ensembles de la Simulación

La información que genera una corrida de Monte Carlo es la posición de cada partícula del sistema en cada paso de la simulación. Empleando las técnicas tradicionales de la Mecánica Estadística podemos pasar de esta información microscópica a la obtención de magnitudes macroscópicas que nos permitan conectar con el experimento.

Supongamos que estamos tratando un sistema puro compuesto por N partículas, encerrado en un volumen V y con una energía fija E. Las posiciones definen un espacio fase de $3N$ dimensiones. Obtener la posición de cada una de las partículas, en cada paso de la simulación, significa obtener la trayectoria de un punto P del espacio fase en función del tiempo, esto es, $P(t)$. Denotemos por $\mathcal{A}$ el valor instantáneo de un cierto observable. El promedio de esta cantidad $\mathcal{A}$ está dado por:

$$\langle \mathcal{A} \rangle_{obs} = \langle \mathcal{A} \rangle_{tiempo} = \frac{1}{\tau_{obs}} \sum_{\tau=1}^{\tau_{obs}} \mathcal{A}(P(\tau)) \tag{1.1}$$

Donde τ representa un tiempo discreto y τ_{obs} son los pasos totales de la corrida, suponiendo que el sistema es ergódico (promedios temporales y promedios de ensemble coinciden en el límite de tiempos muy largos), podemos asociar directamente este promedio con el promedio usual sobre el ensemble de la Mecánica Estadística,

$$\langle \mathcal{A} \rangle_{obs} = \langle \mathcal{A} \rangle_{tiempo} = \langle \mathcal{A} \rangle_{ensemble} \tag{1.2}$$

En otras palabras, por medio del formalismo de la mecánica estadística lo que se hace es generar una sucesión de diferentes estados (configuraciones) del espacio fase (que se suponen no–correlacionados), compatibles con las condiciones externas (N, V, E) es este caso, sobre los cuales se toman los promedios.

La elección del ensemble bajo el cual llevar a cabo la simulación está determinado fundamentalmente por el tipo de problema a tratar. Los promedios estadísticos pueden llevar a pequeñas diferencias en los diferentes ensembles, pero estas desaparecen en el límite, que se alcanza incluso con unos pocos cientos de partículas.

Ambas técnicas se pueden realizar en todos los ensambles. No hay ninguna restricción que force a asociar a alguna de las técnicas a un ensamble particular.

1.4. El Potencial de Interacción Entre Partículas

Un punto de importancia en una simulación de un sistema de partículas es la elección del potencial interatómico del sistema a simular. De la fidelidad con que éste represente las interacciones reales entre las partículas dependerá la calidad de los resultados ya que mientras más detalles de la interacción posea el potencial, los resultados de la simulación serán mejores. La contrapartida de esto es que mientras mayor sea la complejidad funcional del potencial, mayor también será el tiempo de computación requerido para la simulación.

Evidentemente, si lo que se busca es solo probar ciertos aspectos de un modelo teórico, lo mejor será emplear un potencial lo más simple posible que reproduzca la esencia de ese modelo. Diferente es la situación si lo que se desea es simular materiales reales: entonces el potencial deberá contener el máximo de información posible de modo de reproducir los resultados no solo cualitativamente, sino también cuantitativamente.

En todos los estados de la materia, el mejor método para obtener las fuerzas que actúan sobre los átomos es por medio de la Mecánica Cuántica, resolviendo las ecuaciones de Schrödringer para un sistema de N partículas interactuantes. De hecho, ya se han desarrollado métodos para realizar esta tarea desde primeros principios, como el Método de Car-Parrinello, que combina Dinámica Molecular con teoría del funcional de densidad. Sin embargo el costo computacional de esto es alto, pudiendo realizarse en la actualidad simulaciones con a lo mas cientos de partículas. Si se quiere ir mas allá, se debe establecer un compromiso entre la calidad del potencial y las posibilidades de cálculo, esto es lo que mantiene viva la vigencia de los llamados potenciales empíricos y semi-empíricos y la búsqueda de nuevos métodos para mejorarlos.

1.4.1. La Forma del Potencial

En general, la energía potencial $\mathcal{V}$ de un sistema de N partículas puede expresarse en función de las coordenadas de los átomos individuales, de la distancia entre dos de ellos, de la posición relativa entre tres átomos, etc.

$$\mathcal{V} = \sum_{i=1}^{N} v_1(r_i) + \sum_{i=1}^{N}\sum_{j>i}^{N} v_2(r_i, r_j) + \sum_{i=1}^{N}\sum_{j>i}^{N}\sum_{k>j}^{N} v_3(r_i, r_j, r_k) + ... \tag{1.3}$$

Donde el primer término v_1 representa las interacciones de una partícula (fuerza externa), v_2 las interacciones entre dos partículas, v_3 interacciones de tres partículas y así sucesivamente.

El término v_2, sólo depende de la distancia interátomica $r_{ij} = |r_i - r_j|$. Este término es muy importante pues se ha demostrado que por sí solo describe muy bien ciertos sistemas físicos, como es el caso del potencial de Lennard–Jones para los gases nobles. El término de tres cuerpos $v_3(ri, rj, rk)$ es de mucha importancia en el caso de sólidos covalentes, por los enlaces direccionales que ellos poseen.

1.5. El Potencial de Lennard-Jones

Un par de partículas están sujetas a dos fuerzas distintas en términos de la distancia entre ellas: una fuerza atractiva actúa a grandes distancias (fuerza de Van Der Waals, o fuerza de dispersión) y una fuerza repulsiva actuando a pequeñas distancias (conocido como la repulsión de Pauli). El potencial de Lennard-Jones es un modelo matemático sencillo para representar este comportamiento. Su formulación es la siguiente:

$$V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{1.4}$$

Donde ε es la profundidad del potencial, σ es la distancia en la que el potencial entre partículas es cero y r es la distancia entre partículas. En la Figura (1.1) se muestra la forma de este potencial.

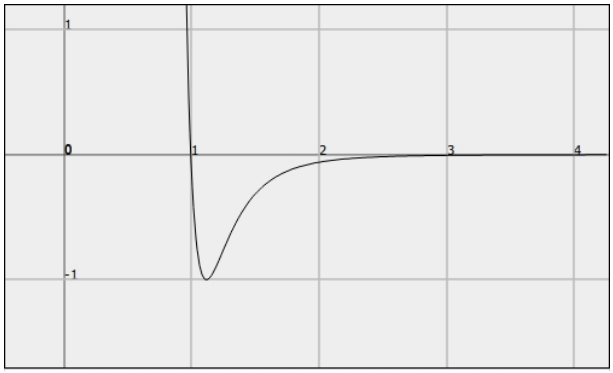

Figura 1.1: Potencial de Lennard-Jones

En general, para ahorrar tiempo computacional, el potencial de Lennard-Jones es truncado en la distancia límite de $r_c = 2.5\sigma$.

Capítulo 2

Mecánica Estadística

2.1. Antecedentes

El nexo entre las propiedades macroscópicas que trata la Termodinámica y las propiedades moleculares y atómicas de la materia está dado por la Mecánica Estadística, cuyos principios fundamentales fueron introducidos por James Clerk Maxwell y Ludwig Boltzmann durante la segunda mitad del Siglo XIX, y cuya estructura matemática fue establecida por Gibbs en su libro Elementary Principles in Statistical Mechanics (1902). Aquí nos limitaremos a presentar las ideas básicas y su aplicación al método de Monte Carlo, ya que un tratamiento completo de la Mecánica Estadística excede los propósitos de esta tesis.

Alrededor de 1870, el gran logro de Boltzmann fue el de relacionar el concepto de entropía, que es una propiedad macroscópica del sistema, con las propiedades moleculares del mismo. La idea básica es que la especificación del estado macroscópico de un sistema es incompleta. En efecto, un sistema que se encuentra en un determinado estado macroscópico puede estar en uno cualquiera de un número inmensamente grande de estados microscópicos. La descripción macroscópica no permite distinguir entre ellos.

Consideremos por un momento un sistema en equilibrio termodinámico (macroscópico). El estado microscópico del sistema cambia continuamente (debido a la agitación térmica y a los choques entre las moléculas). No todos los estados microscópicos por los que pasa el sistema corresponden al equilibrio, pero el número de estados microscópicos que corresponden al equilibrio macroscópico es enormemente grande en comparación con el número de los demás estados microscópicos (que no son de equilibrio). Por lo tanto, la probabilidad de que se observen desviaciones apreciables desde el equilibrio es absolutamente insignificante.

2.2. Macroestados

Consideraremos un sistema de N moléculas idénticas (una sola especie molecular). El macroestado del sistema se puede especificar de varias maneras, según sean las restricciones que se impongan. Por ejemplo, el volumen de un gas dentro de un cilindro de paredes rígidas tiene un valor fijo, pero si el cilindro está cerrado por un pistón que se mueve libremente, sobre el cual aplicamos una presión externa constante, el gas (si está en equilibrio) está obligado a mantener la misma presión. En general, llamaremos restricciones a las condiciones impuestas sobre el sistema que obligan que una o más de sus variables tomen valores definidos.

Consideraremos por ahora un sistema totalmente aislado. Entonces E, V y N son fijos y determinan por completo el macroestado de equilibrio del sistema. Para un macroestado que no sea de equilibrio hay que especificar otros parámetros macroscópicos. Por ejemplo, si nuestro sistema es un gas, hay que especificar (entre otras cosas) la densidad de partículas en todo punto y en todo instante. O, si queremos ser más realistas, tendremos que especificar la densidad de partículas promediada sobre elementos de volumen finitos e intervalos de tiempo finitos, según sean las resoluciones espacial y temporal de nuestras medidas.

En el equilibrio estas variables adicionales tienen valores definidos: por ejemplo la densidad de partículas de un gas es N/V. Indicaremos con α al conjunto de estas variables adicionales (que pueden ser muchas) de modo que un macroestado se especifica dando (E, V, N, α). Se debe notar que para los estados de no equilibrio, nuestra especificación depende del cuán exactas sean nuestras observaciones.

2.3. Microestados

La descripción completa de un microestado es muy complicada, pues en ella interviene un número de parámetros del orden de N (para un mol $N = N_0 \approx 6 \times 10^{23}$). Afortunadamente, para la discusión del equilibrio no hace falta un conocimiento detallado de los microestados, basta con saber cuántos microestado corresponden a cada macroestado. La idea básica es pues simple, pero es necesario recurrir a un artificio. Consideremos por ejemplo un gas en una caja de volumen V, completamente aislado. Claramente, hay un número enorme de maneras de asignar las posiciones $(r_1, r_2, ..., r_N)$ y las cantidades de movimiento $(p_1, p_2, ..., p_N)$ de las moléculas de modo que correspondan a un dado macroestado (por ejemplo, el estado de equilibrio con energía E). Además, cada posición puede tomar una infinidad continua de valores (todos los puntos dentro de V), y lo mismo ocurre con las cantidades de movimiento. En una descripción clásica, los diferentes microestados forman un continuo no numerable, lo cual plantea una dificultad.

Se puede eludir la dificultad, si recordamos que ninguna medición tiene precisión matemática absoluta: Toda medida de la posición de una molécula tiene un margen de incertidumbre δr y toda medida de la cantidad de movimiento tiene una incertidumbre δp (que dependen de la resolución de nuestros instrumentos). Por lo tanto, los valores de $(r_1, r_2, ..., r_N)$ y $(p_1, p_2, ..., p_N)$ que se pueden observar forman en realidad un conjunto discreto, y entonces los microestados también forman un conjunto discreto y los podemos contar. Sin embargo, esta forma de contarlos contiene un elemento de arbitrariedad. El problema de contar los microestados se aclara si adoptamos el punto de vista de la Mecánica Cuántica. De acuerdo con la Mecánica Cuántica, los microestados no forman un conjunto continuo, sino discreto. En consecuencia existe un número entero de microestados y éstos se pueden contar sin incurrir en arbitrariedad. En vista de lo anterior, haremos la suposición que en general todo macroestado de un sistema comprende un número perfectamente definido de microestados.

2.4. Peso estadístico de un macroestado

Dejando por ahora de lado el problema de cómo proceder para contar los microestados, supongamos que lo sabemos hacer. Indicaremos entonces con $\Omega(E, V, N, \alpha)$ el número de microestados que corresponden al macroestado especificado por (E, V, N, α) , y cuya energía está comprendida en el pequeño intervalo entre E y $E + \delta E$. El número $\Omega(E, V, N, \alpha)$ se llama peso estadístico del macroestado (a veces se lo llama también "probabilidad termodinámica" pero no es un buen nombre, pues Ω no es una probabilidad en el sentido ordinario del término). Del mismo modo, indicaremos con $\Omega(E, V, N)$ el peso estadístico del estado de equilibrio.

Uno de los objetivos de la mecánica estadística es la estimación de las propiedades medias de un sistema como por ejemplo un sistema de muchos cuerpos o el comportamiento de un gas encerrado, estos sistemas se pueden encontrar en uno de una gran cantidad de estados. Por ejemplo, en un litro de un gas encerrado a temperatura y presión estándar hay cerca de $3x10^{22}$ moléculas; el solo movimiento de una de ellas hace que se genere un nuevo estado en el sistema por lo que la cantidad de estados en ese sistema es enorme. Por tanto, estos sistemas son muy difíciles de resolver de manera analítica por lo que se hace uso de simulaciones por computadora para poder estimar dichas propiedades.

2.5. Simulaciones Monte Carlo - Cadenas de Markov

Como un primer intento de estimar propiedades de un sistema podemos tomar un subconjunto de los posibles estados del sistema como muestra y darle a cada uno de estos un peso y promediar de manera ponderada la propiedad deseada. El principal problema es como tomamos ese subconjunto de estados, por ejemplo, no podemos tomarlos de manera aleatoria pues es muy probable que solo una muy pequeña fracción de esos estados contribuyan de manera significativa al promedio (por ejemplo, para un fluido de 100 esferas solidas en su punto de congelamiento, el factor de Boltzmann seria apreciable para 1 de cada 10^{260} configuraciones). Por lo tanto el tomar subconjuntos de estados de manera aleatoria no es un método confiable.

Un método funcional es generar un conjunto de estados del sistema mediante los algoritmos Monte Carlo - Cadenas de Markov en el que, en resumen, se inicia con un estado arbitrario del sistema y se genera un nuevo estado dependiendo del estado actual. Estos nuevos estados no sos generados de manera arbitraria sino que obedecen a ciertos criterios para poder alcanzar un equilibrio. Así, la propiedad se podrá estimar como un promedio simple.

2.5.1. Equilibrio

Se dice que una simulación o un sistema ha llegado a un equilibrio cuando cada estado de dicho sistema ocurre con una frecuencia que es proporcional a su peso. Por ejemplo, fue Gibbs quien mostró que para que un sistema se encuentre en equilibrio térmico a una temperatura T, la probabilidad de encontrar al sistema en un estado μ debe ser:

$$\omega_\mu = \frac{\exp(-E_\mu/k_\beta T)}{\sum_\nu \exp(-E_\nu/k_\beta T)} \tag{2.1}$$

Donde la suma se efectúa sobre todos los estados del sistema, E_μ es la energía del estado μ, k_β es la constante de Boltzmann y T es la temperatura del sistema. En este ejemplo el peso de un estado debe ser proporcional a $\exp(-E_\mu/k_\beta T)$. Para asegurar que la simulación llegara al equilibrio es necesario cumplir con dos condiciones: La ergodicidad y el balance detallado.

Condicion de Ergodicidad

La condición de ergodicidad es el requerimiento que hará que nuestra simulación de Monte Carlo - Cadenas de Markov llegue a cualquier estado del sistema desde cualquier otro para una simulación suficientemente grande. El concepto también contempla el hecho de visitar todos los estados posibles. Esto es necesario para lograr nuestro objetivo de generar estados con probabilidades correctas. Por ejemplo, en el algoritmo Metrópolis cada estado ν aparece con alguna probabilidad p_ν en la distribución de Boltzmann, y si ese estado fuera inaccesible desde otro estado μ no importa que tanto continuemos nuestra simulación pues nuestro objetivo sera inalcanzable si empezamos en el estado μ: la probabilidad de encontrar el estado ν en nuestra simulación Monte Carlo - Cadenas de Markov sera cero, y no p_ν como requerimos que sea.

Balance detallado

Esta condición es la que nos asegura que la distribución de probabilidad que generamos después de que nuestro sistema se encuentra en equilibrio es precisamente la distribución de Boltzmann. El concepto también contempla el que se garantice que exista la misma probabilidad de brincar del estado 1 al estado 2 y viceversa, (Ecuación 3.4). Para ocupar un estado con la frecuencia correcta los algoritmos Monte Carlo - Cadenas de Markov generan de forma aleatoria un movimiento de prueba desde el estado actual μ al nuevo estado ν. En el caso de sistemas moleculares, por ejemplo, tal movimiento puede ser el desplazamiento de una sola partícula o el desplazamiento de varias partículas.

Si p_μ es la probabilidad de encontrar al sistema en el estado μ y $t_{\mu\nu}$ es la probabilidad condicional de generar un movimiento de prueba (lo cual lleva al sistema a un nuevo estado ν) dado que el estado

actual del sistema es μ, entonces la probabilidad $w_{\mu\nu}$ de aceptar el movimiento de prueba desde μ a ν esta relacionado con $w_{\nu\mu}$ mediante:

$$p_\mu t_{\mu\nu} w_{\mu\nu} = p_\nu t_{\nu\mu} w_{\nu\mu} \tag{2.2}$$

de esta manera existen diversos algoritmos Monte Carlo - Cadenas de Markov que difieren en la forma en la que los movimientos de prueba son generados (es decir, que usan diferente elección de $t_{\mu\nu}$). Un ejemplo de este tipo de algoritmos es el algoritmo de Metrópolis pero se pueden diseñar otros métodos.

Capítulo 3

Método de Monte Carlo

Supongamos que deseamos calcular una cierta propiedad $\mathcal{A}$ de un sistema de partículas en el ensemble canónico (N, V, T). Según la mecánica estadística, el valor promedio de A esta dado por:

$$\langle \mathcal{A} \rangle = \int \mathcal{A}(r^N) \mathcal{Z}(r^N) dr^N \tag{3.1}$$

Donde:

$$\mathcal{Z}(r^N) = \frac{exp[-\beta \mathcal{U}(r)]}{\int exp[-\beta \mathcal{U}(r)] dr^N} \tag{3.2}$$

Entonces nuestro problema se reduce a que dada la energía potencial $\mathcal{U}(r^N)$, debemos calcular la función de probabilidad $\mathcal{Z}(r^N)$ y luego hacer la integral. Para ello podríamos utilizar el método tradicional de cuadraturas, sin embargo la dimensión del problema hace que utilizar este método sea realmente lento, una alternativa a esto es hacer la integral por el Método de Monte Carlo.

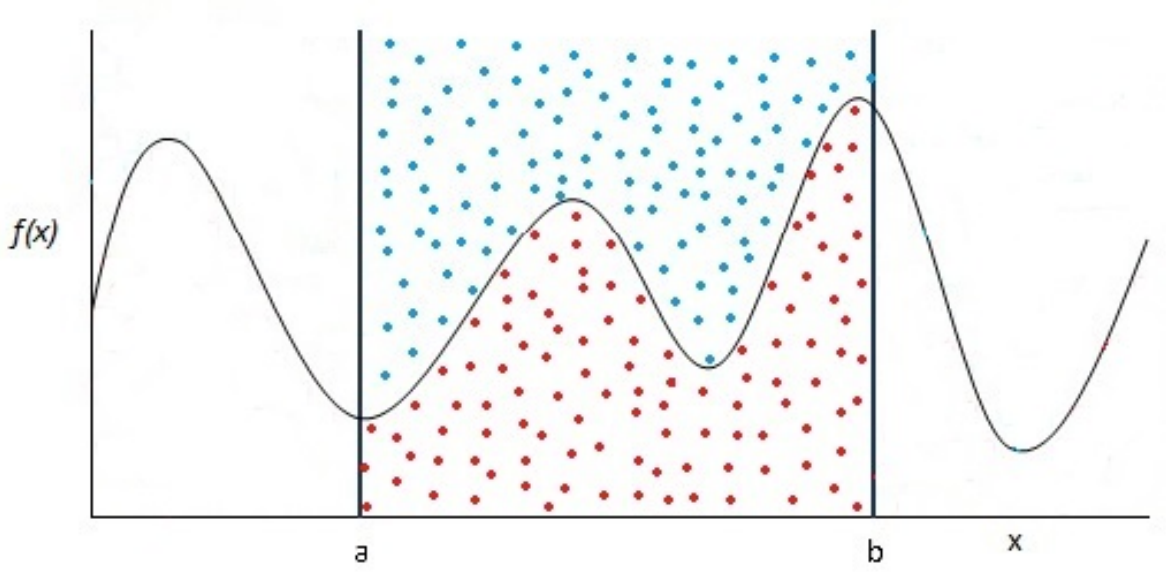

Figura 3.1: Integral en 2D por el método de Monte Carlo

Supongamos que podemos generar aleatoriamente puntos del espacio de configuraciones de acuerdo a la distribución de probabilidades $\mathcal{Z}(r^N)$. Esto significa que en promedio, el número de puntos n_i generados por unidad de volumen alrededor del punto r_i^N es igual a $L\mathcal{Z}(r_i)$, donde L es el número total de puntos generados, es decir:

$$\langle \mathcal{A} \rangle = \frac{1}{L} \sum_{i=1}^{L} n_i \mathcal{A}(r_i^N) \tag{3.3}$$

Sin embargo generar aleatoriamente los puntos en el espacio de configuraciones no es el mejor método, pues como $\mathcal{Z}(r^N)$ es proporcional al factor de Boltzmann $exp(-\beta\mathcal{U})$, los puntos que solo tienen baja energía contribuirán significativamente, mientras que los de alta energía tendrán un bajo peso relativo. La clave entonces esta en idear un método que sólo genere puntos que tengan un peso relativo alto. Esto fue precisamente lo que resolvió Metrópolis en los años cincuenta, con el algoritmo que lleva su nombre.

Consideremos la transición entre los estados a y b, donde la probabilidad que el sistema esté en el estado a es p_a y en el estad b es p_b. En equilibrio se cumple la condición del balance detallado, y por tanto:

$$W_{ab} p_a = W_{ba} p_b \tag{3.4}$$

Donde W_{ab} es la probabilidad de que el sistema pase del estado a al estado b y W_{ba} es la probabilidad de que el sistema pase del estado b al estado a, pero sabemos que en equilibrio la probabilidad de que el sistema se encuentre en el estado a es proporcional a $exp[-\beta E_a]$ y la probabilidad de que el sistema se encuentre en el estado b es proporcional a $exp[-\beta E_b]$, entonces:

$$\frac{p_a}{p_b} = exp[-\beta \Delta E_{ab}] \tag{3.5}$$

Y luego

$$\frac{W_{ba}}{W_{ab}} = exp[-\beta \Delta E_{ab}] \tag{3.6}$$

Así vemos que la probabilidad de transición entre dos estados posibles también está controlada por el factor de Boltzmann. Esto nos da la clave de como escoger la configuración para tener siempre un peso relativo alto: Si al pasar del estado a al estado b el cambio de energía $\Delta E_{ab} < 0$, entonces la probabilidad W_{ab} es 1 (decimos que la transición es factible). Pero si el cambio $\Delta E_{ab} > 0$, entonces $W_{ab} \propto exp[-\beta \Delta E_{ab}]$. En este caso comparamos W_{ab} con un número r entre $[0, 1]$ escogido al azar, y si $W_{ab} > r$, se acepta la transición, y si es menor se rechaza.

El método de Monte Carlo es utilizado para generar configuraciones independientes con peso relativo alto en el espacio de configuraciones, este es un método de carácter estocástico, que hace uso intensivo de un generador de números aleatorios en su funcionamiento.

En la práctica, si tenemos un sistema de N partículas sometido a una temperatura T , el algoritmo de Metrópolis se implementa así:

1. *Calcular la energía del sistema $\mathcal{U}(r^N)$*
2. *Seleccionar una partícula i al azar.*
3. *Dar a la partícula elegida un desplazamiento aleatorio.*
4. *Calcular la nueva energía del sistema $\mathcal{U}_i(r^N)$*
5. *Calcular $\Delta\mathcal{U} = \mathcal{U}_i(r^N) - \mathcal{U}(r^N)$*
6. *Si $\Delta\mathcal{U} < 0$ aceptamos la transición y regresamos a 2.*
   Si $\Delta\mathcal{U} > 0$, elegimos un número $\alpha \in [0, 1]$ al azar.
   Si $\alpha < exp[-\beta\Delta\mathcal{U}]$, aceptamos la transición y regresamos a 2.
   Si $\alpha > exp[-\beta\Delta\mathcal{U}]$, rechazamos la transición y regresamos a 2.
7. *Luego de obtener la cantidad de configuraciones deseadas, guardamos la trayectoria obtenida*

3.1. Generación de configuraciones

3.1.1. Evaluación del potencial

La evaluación de la energía potencial es la parte más costosa en una simulación computacional. Para un sistema de N partículas evaluar en forma directa la interacción de dos cuerpos requiere $O(N^2)$ de operaciones, mientras que evaluar la parte de tres cuerpos requiere en principio $O(N^3)$ de operaciones. De allí la necesidad de elaborar técnicas que permitan ahorrar tiempo en esta tarea.

3.1.2. Radio de corte

Supongamos que tenemos un sistema de N partículas interactuando a través de un potencial de pares y necesitamos evaluar la energía potencial entonces la contribución de cada partícula a la energía potencial proviene de la interacción de la partícula con las $N-1$ partículas restantes del sistema, sin embargo en las simulaciones se acostumbra utilizar potenciales de corto alcance, que generalmente no van más allá de quintos vecinos, de esta forma la contribución principal de la partícula a la energía potencial proviene de sus vecinos más cercanos por lo que podemos introducir un radio de corte r_c , más allá del cual el potencial es prácticamente nulo, ver imagen [1.1]. De este modo, la suma ya no se realiza sobre las $N-1$ partículas restantes, sino que queda restringida a los vecinos que están dentro de la esfera definida por el radio de corte, que llamaremos la influencia del potencial.

3.1.3. Lista de vecinos

Con la introducción del radio de corte el tiempo de computación que ahorramos consiste en que para calcular la energía potencial que aporta cada partícula a la evaluación de la energía potencial total del sistema no es necesario considerar la interacción con las $N-1$ partículas restantes, sino que por la naturaleza del potencial de interacción es suficiente con considerar un número mucho menor. Sin embargo, para saber cuáles partículas son las que están a distancia mayor del radio de corte, y por tanto no contribuyen a la energía potencial, debemos examinar la distancia entre todos los pares de partículas cada vez que necesitemos calcular la energía total del sistema y el tiempo de esta operación es proporcional a N^2.

En otras palabras, para calcular la energía potencial total del sistema necesitamos conocer la distancia entre todos los pares de partículas, eso considerando un potencial de interacción entre partículas que solo considera las interacciones a pares. Al introducir el concepto de radio de corte ya no necesitamos calcular todas las interacciones entre todos los pares de partículas, si no que es suficiente calcular la interacción entre una cantidad mucho menor de pares de partículas.

Para reducir el tiempo de computación Verlet ideo un ingenioso sistema de vecinos de cada partícula, que se renueva cada cierto número de pasos [1]. El método supone que los vecinos con los cuales interactúa la partícula i, es decir, aquellos que están dentro de la esfera de radio r_c y con centro en la posición actual de la partícula i, no varían mucho entre paso y paso de la simulación. A lo más cada cierto número de pasos, digamos m, algunas partículas entran y otras salen, quedando a distancias menores y mayores que r_c respectivamente. Lo que propuso Verlet es hacer una lista para cada partícula i de todos los vecinos que están dentro de su esfera de radio r_c , y así en vez de examinar la distancia de la partícula i con todas las $N-1$ restantes, se examinan esas distancias solo con las partículas de su lista. Esta lista se construye cada cierto m número de pasos. El valor de m estará determinado por las características del problema que es motivo de estudio.

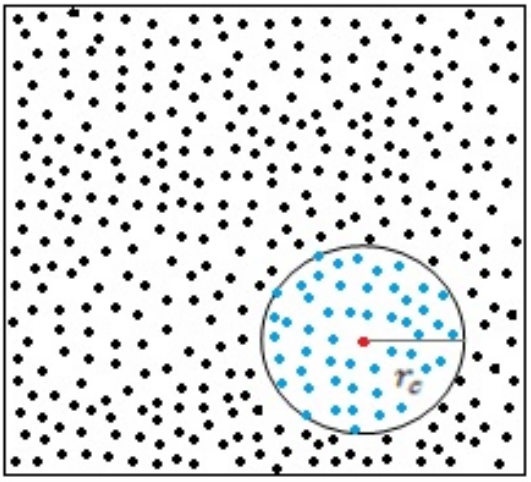

Figura 3.2: Vecindad de interacción determinada por el radio de corte, las partículas de color azul están interactuando con la partícula de color rojo

3.2. Condiciones iniciales

La especificación de las condiciones iniciales para la posición de cada partícula puede realizarse en una variedad de formas, dependiendo de las características del sistema a simular. En el caso de sólidos perfectos es costumbre poner las partículas inicialmente en sus posiciones de equilibrio en la red, tomando como caja de simulación un múltiplo de la celda unitaria en cada una de las direcciones x, y, z. Si se trata de un sólido con defectos en la red cristalina, por ejemplo un borde de grano, entonces las partículas se ponen inicialmente en posiciones que se suponen cercanas al equilibrio y se ocupa alguna técnica de minimización, por ejemplo, descenso del gradiente, recocido simulado u otro, para minimizar la energía del sistema, que al inicio de la simulación (a temperatura $T = 0K$)corresponde a la energía de cohesión.

Para el caso de sistemas líquidos y amorfos (vidrios) se puede proceder de forma similar. Calentando gradualmente, i.e. variando la temperatura y las dimensiones de la caja de simulación para obtener la densidad deseada

3.3. Periodicidad

En nuestro caso, para realizar la simulación de un sistema de partículas estamos utilizando el ensemble canónico (N, V, T), en el que tenemos una cantidad de partículas N finita y un volumen V finito, lo que esto significa es que solo estamos observando una pequeña ventana de lo que sucede en la realidad, ver imagen [3.3].

La correcta elección de las condiciones de frontera es un aspecto que se debe considerar en la simulación. Es una práctica habitual imponer condiciones de frontera periódicas cuando se requiere eliminar la distorsión que introducen las fronteras en el sistema, especialmente cuando las dimensiones lineales que se usan en el modelo son finitas y pretenden obtener propiedades universales.

La justificación experimental es sencilla, en un sistema macroscópico, la influencia termodinámica de la superficie es muy pequeña comparada con el interior del medio.

Sin embargo en la caja de simulación, el número de puntos de superficie en relación a los internos es considerable dado el tamaño pequeño de las cajas, para evitar esto se rodea la caja de simulación mediante replicas idénticas en las tres dimensiones espaciales, de manera similar a como se muestra en la figura

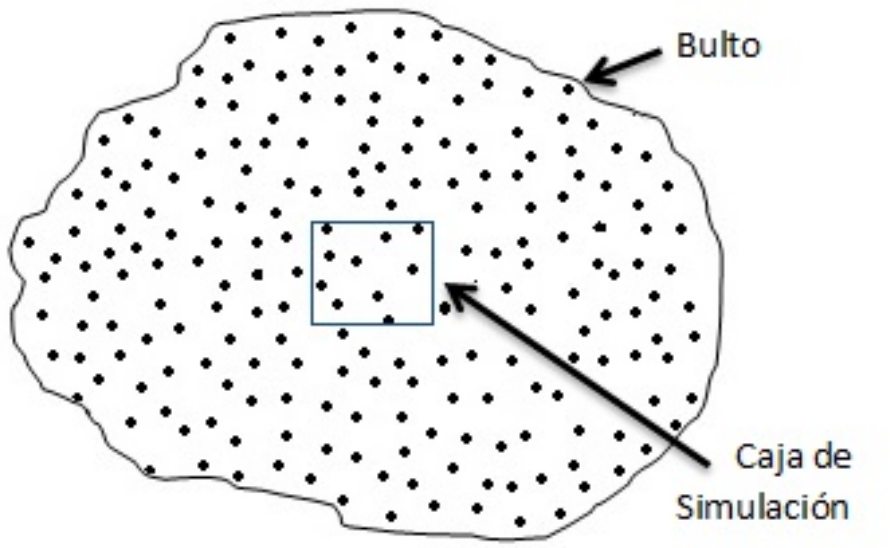

Figura 3.3: Caja de simulación en comparación con el entorno continuo de la realidad.

bidimensional [3.4].

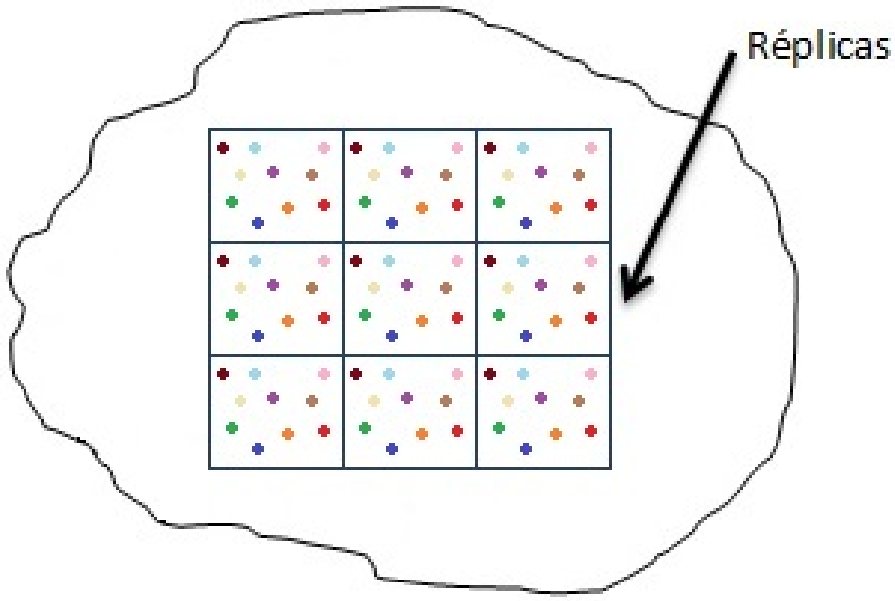

Figura 3.4: Caja de simulación replicada para evitar la distorsión por los bordes y obtener resultados más coherentes con la realidad.

De esta forma, cuando una partícula elegida al azar intenta atravesar una cara de la caja, esta misma partícula entra por la cara opuesta de la caja de simulación y garantizamos la conservación de materia dentro del sistema.

Con esta replica infinita se consigue que las interacciones del sistema sean homogéneas y que la suma

de atracciones y repulsiones que experimentan las partículas se deban a sus vecinos adyacentes y no a la presencia de las fronteras de la caja de simulación.

La elección de un cubo como caja de simulación presenta las ventajas de ser intuitivo y que las condiciones de frontera periódicas son muy sencillas de implementar en el lenguaje de programación.

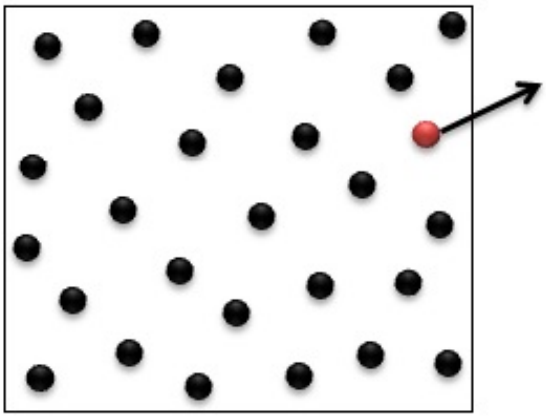

Figura 3.5: Condiciones de frontera periódicas, en este caso la partícula de color rojo que es perturbada como muestra la flecha, sale de la caja de simulación por el lado derecho.

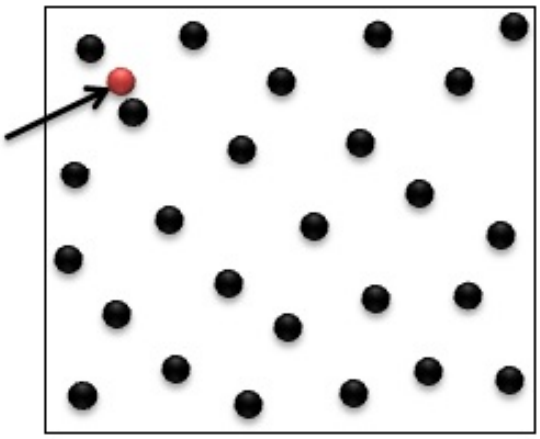

Figura 3.6: La partícula de color rojo que ha salido por el lado derecho, entra por el lado izquierdo simulando las condiciones de frontera periódicas.

Capítulo 4

Algoritmos

El Método de Monte Carlo es a simple vista sencillo de implementar en un algoritmo, sin embargo, el cálculo de la energía potencial del sistema y el cálculo de la diferencia en la energía potencial del sistema al perturbar una partícula son de orden cuadrático para el potencial de interacción que hemos elegido, lo cual hace que el problema sea intratable con un esquema convencional para valores de N en el orden de millones. En esta sección abordaremos las ideas de Verlet para desarrollar un algoritmo que nos ayude a implementar el método de Monte Carlo de forma eficiente y poder trabajar con valores de N suficientemente grandes. También presentaremos el algoritmo propuesto que mejora en gran medida las implementaciones que utilizan la metodología de Verlet, nuestras propuestas de modificaciones para el algoritmo de Verlet son la contribución principal de esta tesis.

Para poder entender mejor el problema en el cálculo de la energía potencial del sistema recordemos que la energía depende directamente de la interacción de cada una de las partículas con todas las partículas del sistema, a su vez, la interacción entre partículas depende de la distancia entre las mismas, la expresión mas completa para calcular la energía potencial del sistema seria:

$$\mathcal{V} = \sum_{i=1}^{N} v_1(r_i) + \sum_{i=1}^{N}\sum_{j>i}^{N} v_2(r_i, r_j) + \sum_{i=1}^{N}\sum_{j>i}^{N}\sum_{k>j}^{N} v_3(r_i, r_j, r_k) + ... \tag{4.1}$$

Donde el primer término v_1 representa las interacciones de una partícula (fuerza externa), v_2 las interacciones entre dos partículas, v_3 interacciones de tres partículas y así sucesivamente.

El potencial de interacción interatómica puede ser tan complejo como deseemos o necesitemos, la contrapartida de esto es que mientras mayor complejidad matemática tenga el potencial, mayor será la complejidad asintótica del algoritmo implementado lo que se ve reflejado en la capacidad de computo y en el tiempo necesario para realizar los cálculos.

En esta tesis hemos decidido trabajar con el potencial de Lennard-Jones que considera únicamente la interacción entre pares de partículas, este potencial es ampliamente utilizado ya que por sí solo describe muy bien la interacción interatómica de los gases nobles.

Utilizando el potencial de Lennard-Jones la expresión para calcular la energía potencial del sistema es la siguiente:

$$\mathcal{U}(r^N) = \sum_{i=1}^{N}\sum_{j>i}^{N} 4\varepsilon \left[\left(\frac{\sigma}{r_{i,j}} \right)^{12} - \left(\frac{\sigma}{r_{i,j}} \right)^{6} \right] \tag{4.2}$$

Donde $r_{i,j}$ es la distancia euclidiana entre la partícula i y la partícula j, σ y ε son parámetros que dependen del elemento con el que se esté trabajando, por ejemplo para un sistema de átomos de argón estos parámetros son: $\varepsilon = 125.7K/K_B$, $\sigma = 0.3345nm$, es muy importante tener cuidado con las unidades de los parámetros, en el apéndice se puede ver con detalle cuales son las unidades de cada variable involucrada en el Método de Monte Carlo.

En la ecuación (4.2) podemos ver claramente que para poder calcular la energía potencial del sistema necesitamos calcular la interacción entre cada par de partículas y para calcular esta interacción necesitamos conocer la distancia entre todos los pares de partículas. Podríamos decir que el cálculo de la energía potencial del sistema se reduce a calcular la distancia entre todos los pares de partículas y la resolución de este subproblema tiene una complejidad cuadrática lo cual tiene como consecuencia que el cálculo de la energía potencial del sistema también tenga complejidad cuadrática, a continuación mostraremos dos maneras de implementar el método de Monte Carlo reduciendo la complejidad del método y así mismo reducir el tiempo necesario para realizar los cálculos.

Siempre que se dice que se ha utilizado una simulación por computadora para dar solución a un problema debemos tener claro que el resultado que obtenemos en una aproximación al resultado real del problema, debido principalmente a la aritmética finita de la computadora y en el caso del Método de Monte Carlo esta aproximación también se debe a que el potencial de interacción interatómica es una aproximación a la realidad. Al tener claro que lo que estamos haciendo es una aproximación a la solución exacta o real del comportamiento dinámico de un sistema de partículas, podemos hacer algunas suposiciones para reducir el costo computacional y la complejidad de cálculo necesarios para resolver el problema.

En el caso del potencial de Lennard-Jones es importante notar que el potencial converge asintóticamente a 0*, como se muestra en la imagen (4.1).*

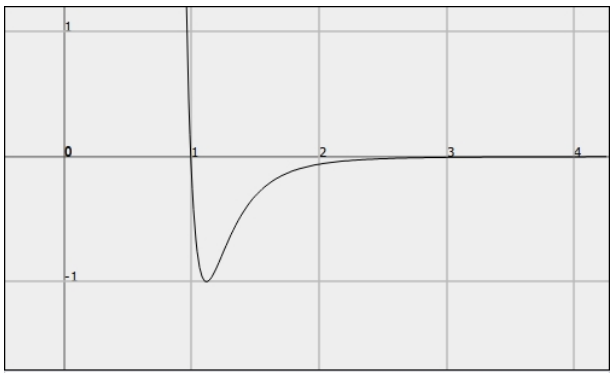

Figura 4.1: Potencial de Lennard-Jones con $\sigma = 1$, $\varepsilon = 1$

Si tomamos un valor de $r_c = 2.5\sigma$ *tenemos que:*

$$LJ(r_c) = 4\varepsilon\left[\left(\frac{\sigma}{2.5\sigma}\right)^{12} - \left(\frac{\sigma}{2.5\sigma}\right)^{6}\right] \approx -0.0163\varepsilon \tag{4.3}$$

Recordemos que $-\varepsilon$ *es la profundidad del potencial, por lo tanto a la distancia de* $r_c = 2.5\sigma$ *tenemos que el potencial vale* $1/60$ *veces el valor mínimo del potencial, lo cual es una insignificancia que se puede descartar. Al valor de* r_c *se le conoce como radio de corte, nos otros hemos decidido tomar un valor* $r_c = 4.5\sigma$ *para nuestras simulaciones para reducir el error numérico.*

Como el valor del potencial para distancias mayores a r_c *es prácticamente* 0 *podemos descartar estas distancias en el cálculo de la energía potencial del sistema y de esta forma ya no será necesario calcular la interacción entre todos los pares de partículas ya que será suficiente con calcular la interacción entre los pares de partículas que se encuentren a una distancia menor o igual que el radio de corte. El uso del radio de corte es lo que nos permitirá reducir los cálculos y el tiempo necesario para nuestras simulaciones.*

Es importante notar que el radio de corte solo podrá ser utilizado cuando el potencial de interacción interatómica converja a 0*, el valor del radio de corte adecuado dependerá de los parámetros del potencial.*

4.1. Algoritmo utilizando las ideas de Verlet

Al introducir el concepto de radio de corte podemos calcular la energía potencial del sistema de una forma más eficiente y rápida. Al fijar un valor para i en la ecuación (4.1) estaremos calculando la aportación de la partícula i al potencial total del sistema, esta aportación se calculara considerando únicamente las partículas j que se encuentren a una distancia de la partícula i que no sea mayor que el radio de corte, a las partículas que cumplen con esta condición las llamaremos vecinas de interacción de la partícula i, al conjunto de las vecinas de interacción lo llamaremos vecindad de interacción.

Como el sistema se encuentra en distintas configuraciones espaciales entre los distintos estados muestreados, las vecindad de interacción de cada una de las partículas va cambiando por lo que sería necesario calcular las vecindad de interacción de cada partícula que se perturbe durante la simulación de Monte Carlo lo cual sería altamente costoso, es por eso que Verlet propone calcular las vecindades de interacción cada m pasos, para m alrededor de 10, con la suposición de que las vecindades de interacción cambian muy poco de un paso a otro de la simulación, sin embargo al no actualizar las vecindades de interacción durante estos m pasos, estaremos cometiendo errores en el cálculo del potencial total del sistema por lo que será necesario ajustar el potencial calculando nuevamente el potencial total del sistema cada m pasos. Esto nos ahorra tener que calcular las vecindades de interacción en cada paso y reducir el tiempo de ejecución de la simulación m veces ya que en vez de calcular las vecindades de interacción en cada uno de los m pasos de la simulación solo calculamos el potencial total del sistema cada m pasos. Notar que calcular el potencial total del sistema es equivalente a calcular las vecindades de interacción ya que en ambos casos en necesario calcular la distancia entre todos los pares de partículas.

Aplicando las ideas de Verlet podemos desarrollar el siguiente algoritmo:

Algoritmo 1 Método de Monte Carlo con Vecindades de Verlet

```
for k = 1,2,..,iter_global do
  -Construir la vecindad de interacción de cada partícula.
  -Calcular la energía potencial del sistema U.
  for j = 1,2,3,...,iter_local do
    -Elegir al azar una partícula n_i.
    -Dar un desplazamiento aleatorio a la partícula elegida.
    -Calcular la nueva energía potencial del sistema U_i
    -Calcular ΔU = U_i − U,
    if ΔU < 0 then
      -Aceptar la transición
    else
      -Tomar aleatoriamente α ∈ [0, 1]
      if α < exp[−βΔU] then
        -Aceptar la transición
      else
        -Rechazar la transición
      end if
    end if
  end for
end for
```

En este algoritmo el valor para m es el número de iteraciones locales. Este algoritmo tiene una complejidad cuadrática por que en cada iteración global es necesario calcular las vecindades de interacción de las partículas lo cual involucra calcular la distancia entre todos los pares de partículas, esto si deseamos hacer una implementación clásica, pero también podremos hacer más eficiente este proceso aplicando las ideas que se proponen más adelante en esta tesis.

En términos de programación, para poder guardar las vecindades de interacción durante las m iteraciones locales, será necesario contar con la memoria para guardar los índices de las partículas que interactúan en cada vecindad de interacción. Para hacer esto necesitamos saber primero cuantas partí-

culas hay en cada vecindad de interacción para después solicitar la memoria correspondiente y llenar la memoria solicitada con los índices de las partículas. También podríamos utilizar listas ligadas para guardar las vecindades de interacción, sin embargo las listas ligadas no son adecuadas para el cómputo en paralelo, pues la memoria asignada para almacenar los datos de cada una de las partículas se encontrara en localidades distintas de la memoria RAM lo cual hace que el acceso y el procesamiento sea mas lento [5].

4.2. Algoritmo Con Rejilla y Búffer

La principal desventaja computacional de las ideas de Verlet es el hecho de tener que calcular las vecindades de interacción y el potencial del sistema cada m pasos ya que durante los pasos intermedios dentro de cada m pasos podemos cometer errores por que las vecindades de interacción no están actualizadas.

Los errores ocurrirán cuando una transición sea aceptada por que la partícula perturbada cambiara de posición y por consiguiente cambiara su vecindad de interacción ya que las distancias entre la partícula perturbada y el resto de las partículas cambiaran. Dicho de otra manera, las partículas que se encontraban a una distancia menor del radio de corte de la partícula perturbada, antes de ser perturbada, ya no serán exactamente las mismas partículas que estarán a una distancia menor del radio de corte de la partícula perturbada, después de ser perturbada.

Para tener más claridad de cómo es que ocurren los errores en los cálculos, veamos el siguiente caso. Es probable que en dos pasos consecutivos se elijan dos partículas que interactúan, digamos las partículas a y b, en el paso en el que se elije la partícula a para ser perturbada, suponemos que la distancia $d(a,b)$ entre las partículas a y b es tal que $d(a,b) < r_c$ por lo que la partícula b se encuentra en la vecindad de interacción de la partícula a y viceversa, supongamos que damos una perturbación a la partícula a de tal forma que la transición es aceptada y por lo tanto modificamos las coordenadas de la partícula a, supongamos que las nuevas coordenadas de la partícula a son tales que $d(a,b) > r_c$ lo que indicaría que la partícula a ya no está en la vecindad de interacción de la partícula b sin embargo como las vecindades de interacción aun no son actualizadas, estaremos considerando que la partícula a si esta dentro de la vecindad de interacción de la partícula b, lo cual acarrearía un error en el cálculo en el siguiente paso en el que suponemos que será elegida la partícula b.

De igual forma que en el caso en el que la partícula a se encuentra dentro de la vecindad de interacción de la partícula b antes de ser perturbada y después de ser perturbada ya no se encuentra en la vecindad de interacción de la partícula b, puede ser que si consideramos una tercer partícula c, sea el caso tal que, $d(a,c) > r_c$ antes de que la partícula a sea perturbada y $d(a,c) < r_c$ después de que la partícula a sea perturbada, lo que nos llevaría a cometer errores en el cálculo de la nueva energía potencial.

La hipótesis de Verlet es que los sucesos descritos en los párrafos anteriores aunque son probables casi no ocurren por lo que las vecindades de interacción cambiaran muy poco de un paso a otro, esto es cierto pero aunque las probabilidades son pocas los sucesos ocurren y acarrean errores.

Para no cometer los errores de cálculo que se cometen en el algoritmo 1, la idea propuesta es actualizar las vecindades de interacción en cada paso de la simulación de Monte Carlo y contar siempre con el valor correcto para la energía potencial del sistema y así no tener que realizar el cálculo de la energía potencial cada m pasos.

Como ya hemos mencionado anteriormente, generar las vecindades de interacción conlleva el cálculo de las distancias entre todos los pares de partículas la cual es una tarea muy costosa. Para reducir el costo computacional de este proceso, proponemos el uso de un sistema de rejilla para hacer un particionamiento del dominio que nos ayudara a generar las vecindades de interacción iniciales. También se hace uso de un búffer para cada rejilla del particionamiento para poder actualizar de forma sencilla las vecindades de interacción en cada paso de Monte Carlo. Estas ideas se describen con mayor detalle en el capítulo de implementación.

Si utilizamos el particionamiento del dominio y el búffer para cada rejilla del particionamiento podemos implementar el Método de Monte Carlo con el siguiente algoritmo:

Algoritmo 2 Método de Monte Carlo con Rejilla y Búffer

-Hacer el particionamiento del dominio.
-Construir las vecindades de interacción
-Calcular la energía potencial total del sistema $\mathcal{U}$.
for j = 1,2,3,...,iter_global **do**
  -Elegir aleatoriamente una partícula n_i
  -Dar un desplazamiento aleatorio a la partícula elegida.
  -Calcular la nueva energía potencial del sistema$\mathcal{U}_i$
  -Calcular $\Delta\mathcal{U} = \mathcal{U}_i - \mathcal{U}$,
  if $\Delta\mathcal{U} < 0$ **then**
    -Aceptar la transición
  else
    -Elegir aleatoriamente un numero $\alpha \in [0, 1]$
    if $\alpha < exp[-\beta\Delta\mathcal{U}]$ **then**
      -Aceptar la transición
    else
      -Rechazar la transición
    end if
  end if
end for

A diferencia del algoritmo 1, en el algoritmo 2 solo es necesario generar las vecindades de interacción y calcular la energía potencial total del sistema una sola vez al inicio del algoritmo, posteriormente solo realizamos el algoritmo de metrópolis el numero de pasos que deseemos.

En el capítulo de implementación veremos con detalle que el algoritmo 2 es de complejidad lineal en comparación con el algoritmo 1 que es de complejidad cuadrática. Esta es la mayor contribución de este trabajo de Tesis, reducir la complejidad del Método de Monte Carlo a una complejidad Lineal.

4.3. Descomposición de Dominio

El método de descomposición de dominio es ampliamente utilizado para resolver ecuaciones diferenciales, esencialmente consiste en descomponer el dominio de estudio $\mathcal{D}$, en n subdominios $\mathcal{D}_i$, tales que:

$$\bigcup_{i=1}^{n} \mathcal{D}_i = \mathcal{D} \tag{4.4}$$

Y además:

$$\mathcal{D}_i \cap \mathcal{D}_j = \emptyset \quad \forall i \neq j \tag{4.5}$$

Después de realizar el particionamiento del dominio se procede a resolver la ecuación diferencial en cada subdomio e integrar la solución general considerando las interacciones en las fronteras de los subdomios.

Este método generalmente se utiliza en clusters de computadoras, aunque también se puede utilizar en computo paralelo de memoria compartida. La idea es descomponer el dominio en n subdomios, donde n es el numero de computadoras en el cluster y el numero de cores de procesamiento, así, cada dominio se resuelve en cada una de las computadoras o en cada core. Esto permite poder resolver problemas en

dominios mucho mas grandes que los que podríamos trabajar en computadoras convencionales.

El método de descomposición de dominios se a utilizado para estudiar sistemas de partículas, ya que se puede trabajar con mayor numero de partículas. Generalmente es utilizado en Dinámica Molecular, en el caso del Método de Monte Carlo, la complejidad sigue siendo cuadrática aplicando esta metodología, así que la única ventaja que obtenemos al usar esta idea con el Método de Monte Carlo es que podemos trabajar con mas partículas, y con el algoritmo propuesto en esta tesis podemos trabajar con mas partículas en un menor tiempo de ejecución y con complejidad lineal.

4.4. Descomposición en celdas

El método de Descomposición en celdas, propone utilizar un mallado para localizar las partículas vecinas, esto es similar a lo que trabajamos en esta tesis, pero nosotros hemos agregado un cálculo optimizado y directo de los índices en rejilla para aprovechar las ventajas del computo paralelo, además el ordenamiento de los datos en memoria y el uso del búffer nos brindan una complejidad lineal en comparación con los métodos reportados en la literatura que utilizan descomposición en celdas. Estos métodos son ampliamente utilizados en Dinámica Molecular pero aprovechamos las características del Método de Monte Carlo para tener un mayor beneficio.

Capítulo 5

Implementación

En este capítulo describiremos en detalle las ideas planteadas que permitan que el Método de Monte Carlo sea implementado con un algoritmo de complejidad lineal y también explicaremos como sacar mayor provecho de estas ideas al utilizar computo paralelo.

La mayor complicación a la que nos enfrentamos cuando deseamos realizar una implementación del método de Monte Carlo es el cálculo de la energía potencial inicial del sistema y el cálculo de la energía potencial del sistema cuando perturbamos una partícula. Como hemos mencionado, esta tarea involucra el cálculo de la interacción entre todos los pares de partículas. Al utilizar un radio de corte para truncar el potencial de interacción interatómica hemos visto que no es necesario calcular las interacciones entre todos los pares de partículas si no que es suficiente con calcular las interacciones entre los pares de partículas que se encuentran a una distancia menor que el radio de corte.

Podemos sacar provecho del hecho de que para las distancias mayores al radio de corte el valor del potencial es prácticamente 0*. La idea principal que nos ayudara a generar una implementación más eficiente del método de Monte Carlo es que para calcular la energía potencial del sistema solo necesitaremos considerar las interacciones entre las partículas que están a una distancia menor que el radio de corte.*

Para realizar el cálculo de la energía potencial del sistema necesitamos calcular la aportación de cada partícula i al potencial total, para calcular la aportación de cada partícula i necesitamos saber que partículas $j \neq i$ cumplen con la condición de que la distancia $d(i,j) < r_c$. La primera idea que se nos puede venir a la mente es calcular la distancia entre la partícula i y cada una de las $N-1$ partículas restantes del sistema y verificar que partículas cumplen con la condición deseada, si procediéramos con esta idea la complejidad del algoritmo seria cuadrática ya que sería necesario calcular las distancias entre todos los pares de las partículas pero ya hemos mencionado que un algoritmo de complejidad cuadrática dificultaría poder trabajar con valores de N muy grandes.

En esta sección vamos a describir un método que nos permitirá saber de forma rápida cuales son las partículas j tales que $d(i,j) < r_c$, sin la necesidad de tener que calcular las distancias entre la partícula i y las $N-1$ partículas restantes. Además veremos que este método hace que el Método de Monte Carlo sea de complejidad lineal.

5.1. Particionamiento del dominio

Podríamos decir que el problema al que nos enfrentamos cuando deseamos realizar una implementación del Método de Monte Carlo por medio de un algoritmo que será usado en una simulación por computadora, es un problema de gran escala, ya que mientras más grande sea N necesitaremos más capacidad de procesamiento y más tiempo para realizar el procesamiento de los datos. En computación existen diversas metodologías para atacar los problemas de gran escala, en particular, la metodología de divide y vencerás es muy utilizada en este tipo de problemas, esta idea también es ampliamente aplicada en el

computo paralelo, esto nos ayudara a hacer más eficientes nuestras implementaciones.

La metodología de divide y vencerás es utilizada en elemento finito donde para resolver una ecuación diferencial en un dominio, se particiona el dominio en subdominios llamados elementos finitos, la ecuación se resuelve en cada uno de estos elementos finitos de forma independiente sin perder detalle en las interacciones en las fronteras de los elementos, una vez que se ha dado solución a la ecuación diferencial en cada elemento finito, estas soluciones se integran en la solución global de la ecuación diferencial. El mallado que se utiliza para particionar el dominio puede tener múltiples naturalezas, desde mallados regulares con polígonos regulares, hasta mallados irregulares con distintos tipos de elementos finitos y de distintos tamaños.

Simon Green [3], utiliza un mallado regular para detectar colisiones entre partículas en su implementación de materiales granulares, esta idea puede ser utilizada para saber que partículas interactúan dentro de la influencia del radio de corte en una simulación de Monte Carlo.

Las simulaciones de Monte Carlo en la realidad se aplican en 3 dimensiones pero para ejemplificar nuestras ideas mostraremos el caso en 2 dimensiones, todas las ideas son fácilmente extensibles al caso 3D.

Supongamos que queremos hacer una simulación de Monte Carlo, para ello necesitamos definir el tamaño de la caja de simulación, la cantidad de partículas y cada una de las posiciones de las partículas. En el caso $2D$ la caja de simulación queda determinada si conocemos el punto inferior izquierdo (min_x, min_y) y el punto superior derecho (max_x, max_y). Denotamos con N la cantidad de partículas, las posiciones de cada una de las partículas se generan de forma aleatoria dentro de la caja de simulación.

Para poder saber que partículas interactúan dentro de la influencia del radio de corte r_c vamos a realizar un mallado regular dentro de la caja de simulación, el mallado se hace con cuadrados de lado r_c como se muestra en la imagen [5.1]. Llamaremos rejilla a cada cuadrado de la malla regular. Al realizar el mallado de esta forma, podemos garantizar fácilmente que para cada partícula dentro de la caja, las partículas que se encuentren en la vecindad de interacción estarán dentro de las 8 rejillas al rededor de la rejilla de la partícula y dentro de la rejilla a la que pertenece la partícula. Por ejemplo, en la imagen [5.1], todas las partículas que se encuentran a una distancia menor que r_c de la partícula de color rojo, se encontraran dentro de las rejillas 23, 24, 25, 33, 34, 35, 43, 44 y 45. Para poder afirmar esto es muy importante que el ancho de cada rejilla sea r_c.

El mallado de la caja de simulación es el primer paso para poder calcular la energía potencial del sistema de forma más eficiente y sin cometer errores en el cálculo al perturbar partículas como veremos más adelante. Sin pérdida de generalidad podemos suponer que el tamaño de la caja en cada una de las dimensiones del espacio es un múltiplo entero del radio de corte. Esto no es necesario para el buen funcionamiento pero ayuda a que las ideas sean más claras.

5.1.1. Hashing

Al realizar el mallado de la caja de simulación con las especificaciones mencionadas en la sección anterior, sabemos que las partículas que interactúan se encuentran en rejillas vecinas, considerando condiciones de frontera periódicas. Con la numeración de las rejillas que se muestra en la imagen [5.2] podemos asociar a cada partícula el número de rejilla en la que se encuentra, en términos de programación esto se conoce como hashing y a continuación vamos a describir como hacer este hashing.

Lo primero que necesitamos para poder hacer el hashing es numerar las rejillas, podríamos numerarlas de la manera que más nos guste, sin embargo, la numeración que se muestra en la figura [5.2] es más natural para hacer el hashing de forma sencilla.

Después de numerar las rejillas, vamos a numerar las columnas y los renglones del mallado comenzando por 0, los índices de la columnas corresponden a $grid\_pos\_x$ y los índices de los renglones corresponden

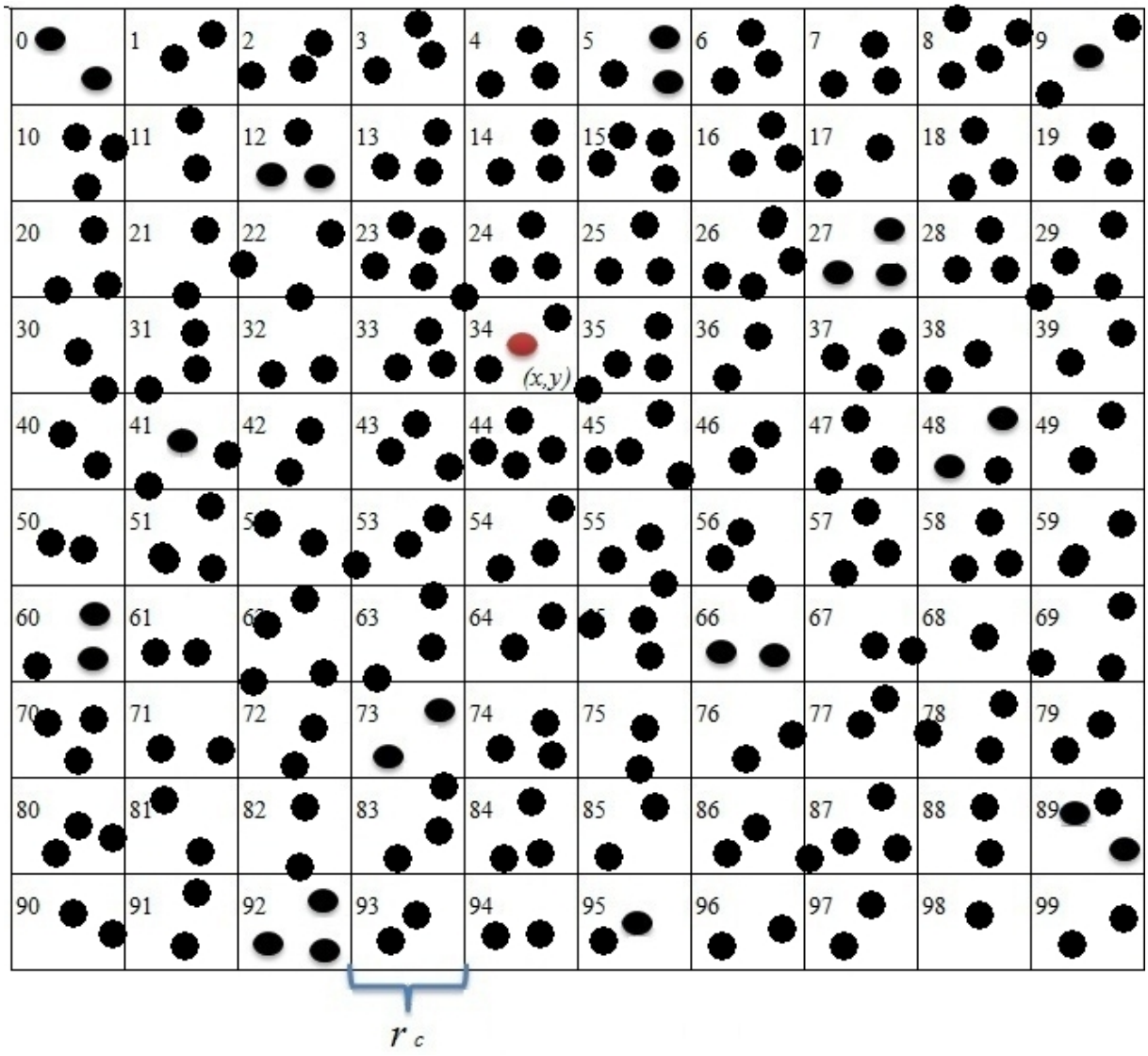

Figura 5.1: Particionamiento del dominio con malla regular cuadrada de ancho r_c

a grid_pos_y. Cada partícula tendrá asociado el valor de grid_pos_x y de grid_pos_y correspondiente a la rejilla en la que esta. Para la cantidad de columnas y de renglones del mallado usaremos los nombres tam_x y tam_y respectivamente. En el caso del mallado de la imagen [5.2], tam_x = 10 y tam_y = 10.

Observar que si conocemos los valores correspondientes de grid_pos_x y grid_pos_y asociados a una partícula, digamos la partícula a, podemos hacer fácilmente el hashing como se muestra a continuación:

$$a.indice\_rejilla = a.grid\_pos\_y * tam\_x + a.grid\_pos\_x \tag{5.1}$$

Para el caso de la partícula de color rojo en la imagen [5.2], grid_pos_x = 4 y grid_pos_y = 3, por lo que:

$$indice\_rejilla = grid\_pos\_y * tam\_x + a.grid\_pos\_x = 3 * 10 + 4 = 34 \tag{5.2}$$

Y como sabemos la partícula de color rojo se encuentra en la rejilla 34. El problema se reduce entonces a saber cuáles son los valores de grid_pos_x y grid_pos_y asociados a cada partícula, ya que conociendo estos valores podemos hacer el hashing de forma simple.

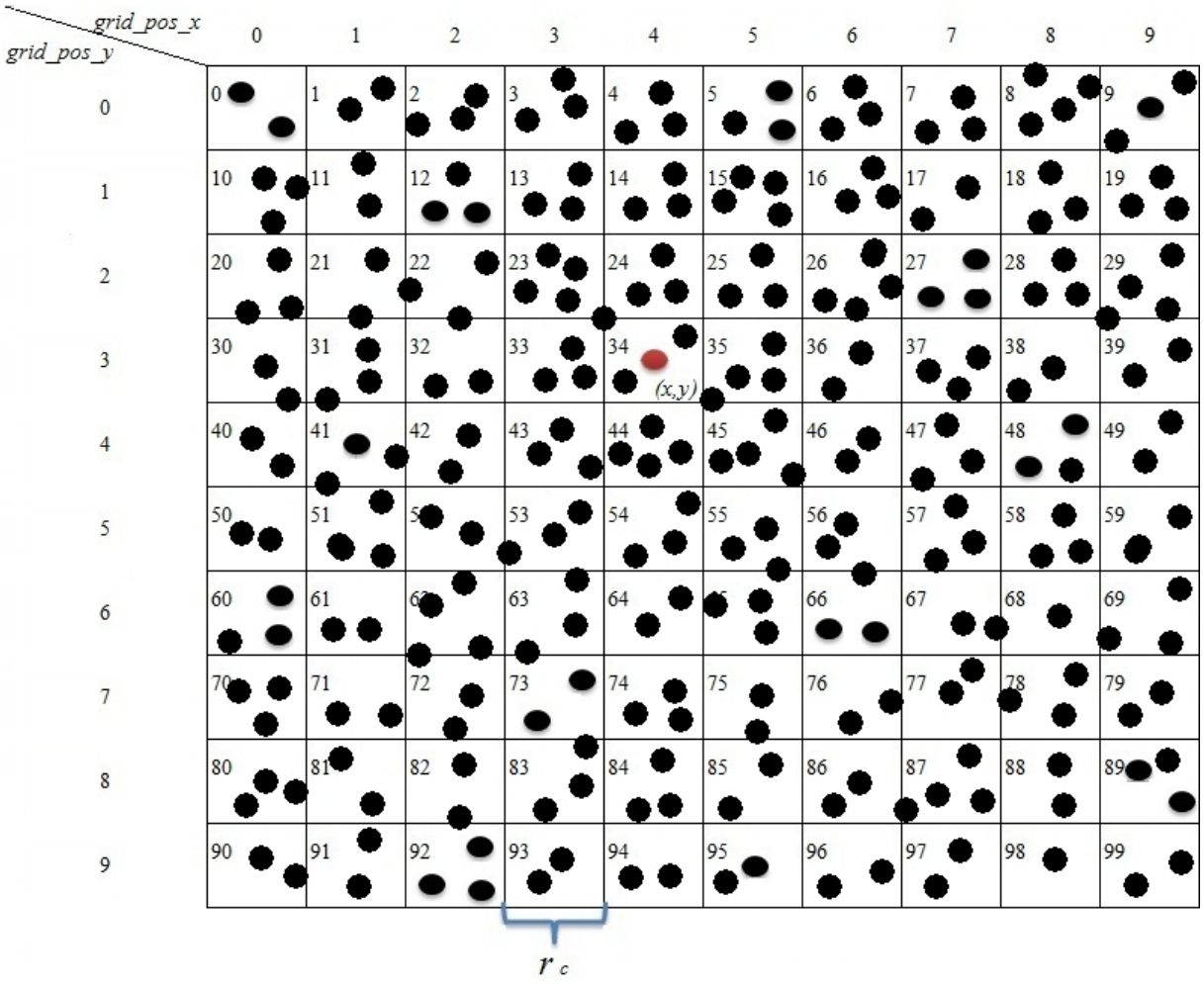

Figura 5.2: Hashing utilizando el mallado del dominio

Para calcular los valores de $grid\_pos\_x$ *y* $grid\_pos\_y$ *asociados a cada partícula utilizaremos las coordenadas de la partícula* (x, y)*, los dos puntos que determinan las dimensiones de la caja de simulación* (min_x, min_y) *,* (max_x, max_y) *y el radio de corte* r_c*. Con estos datos podemos hacer los siguientes cálculos:*

$$grid\_pos\_x = \left\lfloor \frac{x - min\_x}{r_c} \right\rfloor \tag{5.3}$$

$$grid\_pos\_y = \left\lfloor \frac{y - min\_y}{r_c} \right\rfloor \tag{5.4}$$

Donde el operador $\lfloor z \rfloor$ *es el mayor entero que es menor o igual que* z*, mejor conocido como el operador piso. Utilizando las ecuaciones [5.2],[5.3] y [5.4] podemos realizar el hashing que hace corresponder a cada partícula con el índice de la rejilla en la que se encuentra.*

En este punto de la implementación es conveniente hacer notar que cada partícula tiene asociados varios datos, además de sus coordenadas cada partícula tiene asociadas las siguientes variables:

- x *, la coordenada* x *de la partícula.*
- y *, la coordenada* y *de la partícula.*
- i_r *, el índice de la rejilla en la que se encuentra la partícula.*
- i_g *, el índice global de la partícula.*

- *grid_pos_x , el numero de columna correspondiente a la rejilla de la partícula.*

- *grid_pos_y , el numero de renglón correspondiente a la rejilla de la partícula.*

En términos de programación (C , C++), podríamos utilizar una estructura en el caso de C o una clase en el caso de C++ donde se engloban todos estos atributos, esto haría una programación más sencilla y limpia, sin embargo cuando inicialicemos nuestra simulación y asignemos valores a estos atributos, los datos de cada partícula se encontraran en espacios distantes de la memoria RAM lo que hace menos eficiente el procesamiento de los datos ya que no aprovechamos las ventajas de la memoria cache, aun en una implementación en serial es conveniente tener los datos de las partículas en localidades adyacentes de la memoria RAM, por esta razón es más conveniente utilizar vectores para cada una de las variables de las partículas, en la imagen [5.3] podemos ver una representación gráfica de esta idea.

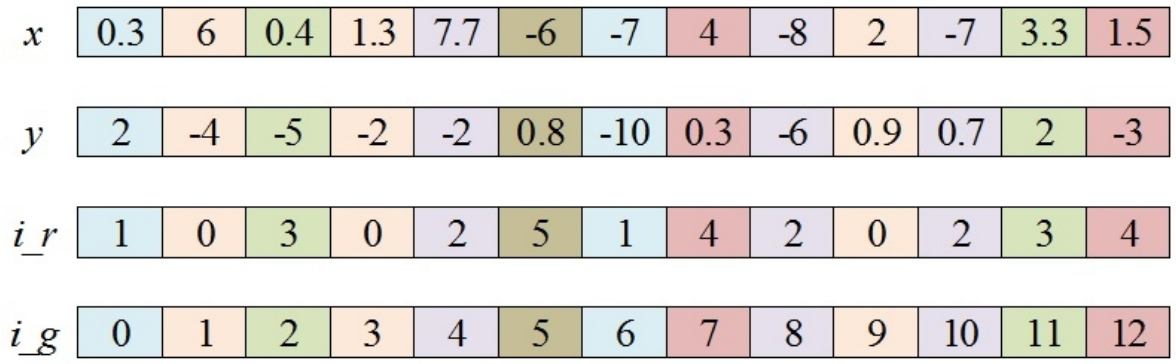

Figura 5.3: Datos de las partículas almacenados en vectores

Al realizar el almacenamiento de cada uno de los atributos de las partículas en vectores aprovechamos las ventajas que nos ofrece la memoria cache, ya que como es sabido, cuando el procesador necesita información que se encuentra almacenada en la memoria RAM, la lectura a la memoria RAM no se hace solo sobre la información que necesita el procesador si no que se copia una sección de memoria alrededor del dato solicitado y se almacena en la memoria cache la cual es de mas rápido acceso que la memoria RAM, la ventaja es que si en la siguiente instrucción se solicita un dato que se encuentra en una localidad cercada de la memoria RAM a la localidad del dato solicitado en la instrucción anterior, entonces el dato ya se encontrara en la memoria cache y no será necesario acceder hasta la memoria RAM lo cual nos hace ahorrar tiempo de ejecución.

Si los datos estuvieran en localidades separadas de memoria, como en el caso en que los datos de las partículas se encapsularan en una clase, entonces cada vez que solicitamos datos de partículas diferentes, es mucho más probable que en cada ocasión accediéramos a localidades de la RAM lo cual significa un proceso mucho más tardado.

Hasta este punto tenemos una manera rápida y sencilla de saber cuál es la rejilla del mallado correspondiente a cada partícula y sabemos que las partículas que interaccionas dentro de la influencia del radio de corte se encuentran en rejillas vecinas (considerando condiciones de frontera periódicas), pero aun nos falta saber cómo recorrer las casillas vecinas para una partícula dada y verificar que partículas interactúan con esta partícula dada. Para ello será importante analizar cómo funcionan las condiciones de frontera periódicas.

5.2. Condiciones de frontera periódicas

Como se ha mencionado en los capítulos anteriores, es necesario implementar condiciones de frontera periódicas en la caja de simulación para evitar los problemas de borde y además tener una simulación más acorde con la realidad. En esta sección veremos cómo trabajar con las condiciones de frontera periódicas, tanto en las actualizaciones de las coordenadas de una partícula en movimiento como para recorrer las rejillas vecinas en el mallado del dominio y también al calcular las distancias entre partículas.

5.2.1. Actualización de coordenadas con fronteras periódicas

Para que las ideas de cómo trabajar con las fronteras periódicas sean más claras analizaremos primero el caso en 1-D, posteriormente analizaremos el caso 2-D y el caso 3-D es análogo.

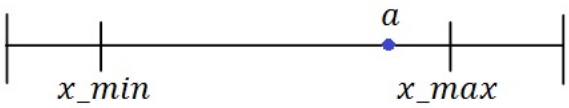

Figura 5.4: Caja de simulación 1D

El equivalente a la caja de simulación en 1-D es un intervalo en el eje real, este intervalo puede estar caracterizado por las cotas del intervalo x_min y x_max como se muestra en la imagen [5.4]. La idea gráfica de las condiciones de frontera periódicas es que si una partícula que se encuentra cerca de alguna de las fronteras, como la partícula a en la figura [5.4] tiene una perturbación p de tal forma que $x + p < x\_min$ o $x + p > x\_max$ según sea el caso de la frontera y donde x es la coordenada de la posición de la partícula a. Podríamos decir que la partícula sale del intervalo por una frontera y debe entrar por la otra frontera como si se tratara de un circulo formado por el intervalo (x_min, x_max) de tal forma que los extremos se unen como se muestra en la imagen[5.5].

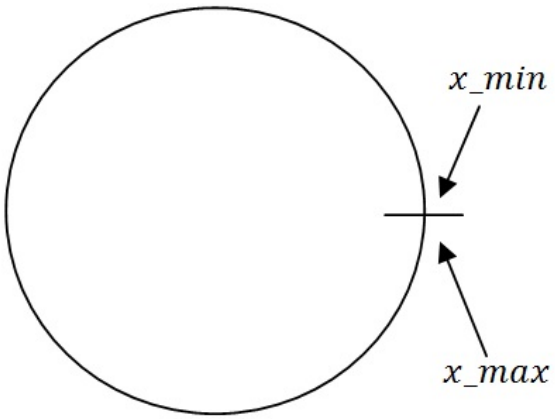

Figura 5.5: Analogía de las condiciones de frontera 1D

Si no consideráramos condiciones de frontera periódicas podríamos hacer que las partículas rebotaran en las fronteras o que las que se salen de las fronteras simplemente se pierden, en cualquiera de estos dos casos la actualización de las coordenadas de las partículas seria más simple pero como si consideramos condiciones de frontera periódicas necesitamos actualizar las coordenadas de las partículas de una forma especial como se muestra a continuación.

Sea a una partícula con posición de coordenada $x \in (x\_min, x\_max)$, es decir, $x\_min < x < x\_max$. Si damos una perturbación p a la partícula a pueden suceder tres casos posibles:

1. *$x\_min < x + p < x\_max$, lo que indica que la partícula no sale por ninguna de las fronteras.*
2. *$x + p < x\_min$, lo que significa que la partícula salió por la frontera izquierda.*
3. *$x\_max < x + p$, lo que significa que la partícula salió por la frontera derecha.*

En el caso 1 no habría que hacer nada extra y la actualización de la coordenada de la partícula seria $x' = x + p$ donde x' seria la nueva coordenada de la posición de la partícula a.

En el caso 2 hay que hacer la actualización de tal forma que la partícula entre por la frontera derecha, en este caso tendremos que hacer un paso previo para actualizar la coordenada, si $x + p < x\_min$ entonces $y = x\_min - (x + p)$ es el excedente que indica que tanto se salió la partícula a por la frontera izquierda, este excedente debe ser compensado en la frontera derecha de tal forma que la actualización de la coordenada de la partícula a es $x' = x_max - y$, de esta forma simulamos que la partícula que sale por la frontera izquierda entra por la frontera derecha.

El caso 3 es análogo al caso 2, si $x\_max < x+p$, entonces $y = (x+p) - x\_max$ representa el excedente que debe ser compensado en la frontera izquierda, de tal forma que la actualización es $x' = x\_min + y$, como se muestra en la imagen[5.6], y así simulamos que la partícula que sale por la frontera derecha entra por la frontera izquierda.

Figura 5.6: Partícula saliendo por la frontera derecha

Las condiciones de frontera periódicas en 1D son la base para las condiciones de frontera periódicas en 2D y 3D, el concepto puede extenderse a más dimensiones pero para nuestros fines solo necesitamos las condiciones de frontera en 3D.

Para implementar las condiciones de frontera en 2D la idea gráfica es similar que el caso 1D, solo que al tener 2 dimensiones los casos en los que se sale por una frontera son 4, podemos caracterizarlos como sigue:

1. *Si una partícula sale por la frontera izquierda debe de entrar por la frontera derecha.*
2. *Si una partícula sale por la frontera derecha debe de entrar por la frontera izquierda.*
3. *Si una partícula sale por la frontera superior debe de entrar por la frontera inferior.*
4. *Si una partícula sale por la frontera inferior debe de entrar por la frontera superior.*

Notar que las situaciones 1 y 2 solo involucran a la coordenada x y las situaciones 3 y 4 solo involucran a la coordenada y. Esto nos da la pauta para reducir el caso 2D a dos casos 1D en cada una de las dimensiones.

Recordar que en este caso la caja de simulación está determinada por el punto inferior izquierdo (x_min, y_min) y el punto superior derecho (x_max, y_max) como se muestra en la imagen [5.7]. Dadas las coordenadas (x, y) de una partícula y una perturbación (p_x, p_y), la perturbación a la partícula se hace con una suma de vectores de las coordenadas de la partícula y de las coordenadas de la perturbación, es decir:

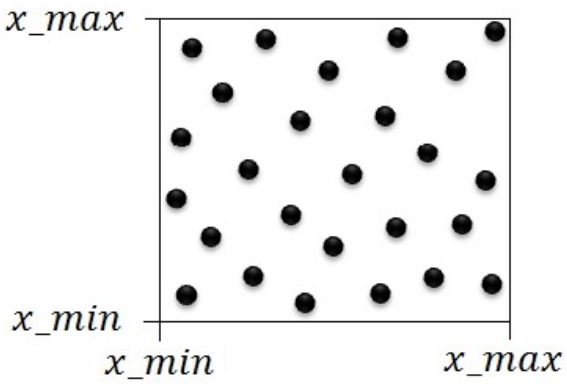

Figura 5.7: Caja de simulación 2D.

$$(x', y') = (x, y) + (p\_x, p\_y) = (x + p\_x, y + p\_y) \tag{5.5}$$

Lo que tenemos que hacer es observar que sucede con las coordenadas en cada una de las 4 situaciones en que una partícula sale por una frontera:

Caso 1: *La partícula sale por la frontera izquierda, en este caso tenemos que* $x + p\_x < x\_min$.

Caso 2: *La partícula sale por la frontera derecha, en esta caso tenemos que* $x\_max < x + p\_x$.

Caso 3: *La partícula sale por la frontera superior, en este caso tenemos que* $y\_max < y + p\_y$.

Caso 4: *La partícula sale por la frontera inferior, en este caso tenemos que* $y + p\_y < y\_min$.

Pero el caso 1 y 2 se reducen al caso de fronteras periódicas de una dimensión y el caso 3 y 4 también se reducen al caso de fronteras periódicas de una dimensión, por lo que tratando cada dimensión x *y* y *como un problema de fronteras periódicas de una dimensión podemos resolver el problema de fronteras periódicas en 2D.*

El caso de la implementación de las fronteras periódicas en 3D es análogo al caso 2D, con los mismos argumentos podemos llegar al resultado de que para implementar la actualización de las coordenadas de una partícula al ser perturbada, es suficiente con tratar el problema como un problema de fronteras periódicas en cada una de las dimensiones x, y *y* z.

Es importante mencionar que la separación de los problemas de fronteras periódicas en 2D y 3D en casos de 1D se basa en el hecho de que los casos en los que la partícula atraviesa una frontera son independientes en cada dimensión, es decir, el hecho de que una partícula atraviese una frontera en una de las dimensiones x, y *y* z *no implica que la partícula atraviese una frontera en alguna de las otras dos dimensiones restantes, dicho de otra manera, una partícula solo puede atravesar una frontera a la vez, tal vez esto cause algún conflicto si la partícula atraviesa exactamente por una esquina, pero como veremos a continuación una perturbación bidimensional puede ser descompuesta en dos perturbación unidimensionales.*

Retomemos la siguiente ecuación de actualización de las coordenadas en una perturbación.

$$(x', y') = (x, y) + (p\_x, p\_y) = (x + p\_x, y + p\_y) \tag{5.6}$$

Gráficamente podemos verlo en la imagen [5.8].

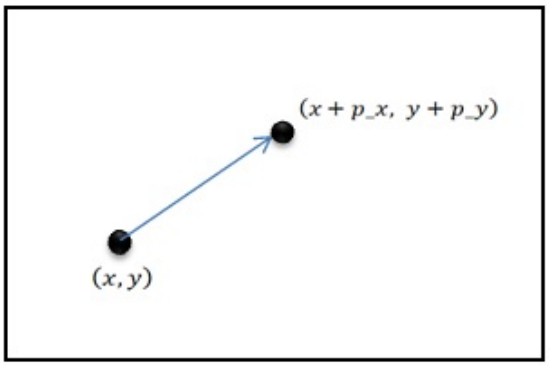

Figura 5.8: Actualización de las coordenadas en 2D

Con algunas manipulaciones algebraicas podemos reescribir esta ecuación como una doble suma unidimensional, primero veamos que:

$$(p\_x, p\_y) = (p\_x, 0) + (0, p\_y) \tag{5.7}$$

Por lo tanto:

$$(x', y') = (x, y) + (p\_x, 0) + (0, p\_y) = (x + p\_x, y) + (0, p\_y) = (x + p\_x, y + p\_y) \tag{5.8}$$

Lo cual es equivalente a actualizar primero una coordenada y después actualizar la siguiente coordenada, gráficamente podemos verlo en la imagen [5.9]. Esto se cumple también para el caso 3D.

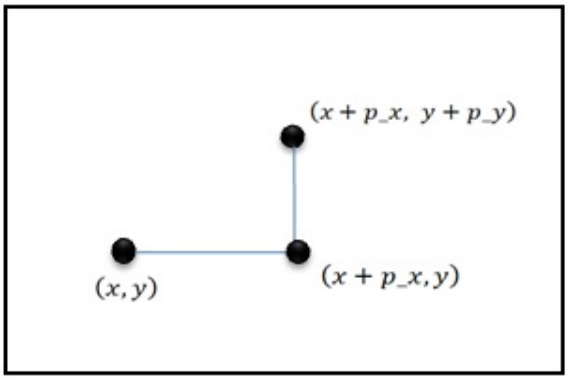

Figura 5.9: Actualización independiente de las coordenadas en 2D

Concluimos que es posible tratar las condiciones de frontera periódicas en dimensiones mayores como casos independientes de una dimensión en cada una de las dimensiones.

5.2.2. Distancia entre las partículas

Además de considerar las condiciones de frontera periódicas al realizar la actualización de las coordenadas de una partícula perturbada también necesitamos considerar estas condiciones cuando deseamos calcular la distancia entre dos partículas. Para este problema procederemos de igual forma que en el caso de la actualización de las coordenadas, primero analizaremos el caso 1D y posteriormente veremos que los casos de dimensiones mayores se reducen al caso 1-D.

En el caso 1D, comenzamos por determinar la caja de simulación como un intervalo (x_min, x_max) *en la recta real. Dadas dos partículas* a *y* b *con coordenadas* x_1 *y* x_2 *respectivamente, la distancia* $d(a,b)$ *entre las partículas* a *y* b *sin considerar condiciones de frontera periódicas está determinada por:*

$$d(a,b) = |x_2 - x_1| \tag{5.9}$$

Como se muestra en la imagen [5.10].

Figura 5.10: Distancia 1D sin condiciones de frontera periodicas

Pero si consideramos las condiciones de frontera periódicas hay dos posibilidades para la distancia entre las partículas como se muestra en la imagen [5.11], y la distancia entre las partículas será la de menor valor absoluto. Sin pérdida de generalidad podemos suponer que la partícula a *es la que se encuentra más cercana a la frontera izquierda, así las dos posibles distancias entre las partículas son:*

Figura 5.11: Distancia 1D con condiciones de frontera periodicas

- $d_1(a,b) = x_2 - x_1$
- $d_2(a,b) = (x_1 - x\_min) + (x\_max - x_2)$

por lo que la distancia entre las partículas a *y* b *considerando condiciones de frontera periódicas es:*

$$d(a,b) = \text{mín}(d_1(a,b), d_2(a,b)) \tag{5.10}$$

Si antes de calcular la distancia averiguamos que partícula está más cerca de la frontera izquierda podemos proceder calculando únicamente d_1 *y* d_2 *como se describió anteriormente pero si deseamos evitar averiguar que partícula se encuentra más cercana a la frontera izquierda podemos calcular las siguientes tres posibles distancias:*

- $dist_1(a,b) = |x_2 - x_1|$
- $dist_2(a,b) = (x_1 - x\_min) + (x\_max - x_2)$
- $dist_3(a,b) = (x_2 - x\_min) + (x\_max - x_1)$

y hacer:

$$d(a,b) = \text{mín}(dist_1(a,b), dist_2(a,b), dist_3(a,b)) \tag{5.11}$$

Cualquiera de las dos formas para calcular la distancia con condiciones de frontera periódicas da el mismo resultado, usar una o la otra dependerá de las ventajas y desventajas que cada una ofrezca dependiendo del lenguaje y la metodología que sean utilizadas para su implementación.

Para calcular la distancia entre las partículas en el caso 2D considerando condiciones de frontera periódicas primero analizaremos el caso sin fronteras periódicas.

Dadas dos partículas a y b con coordenadas (x_1, y_1) y (x_2, y_2) respectivamente, la distancia entre a y b sin considerar fronteras periódicas se calcula como sigue:

$$d(a,b) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.12}$$

Pero de la misma forma que en la sección anterior podemos reducir el cálculo de la distancia a dos problemas 1-D ya que $(x_2 - x_1)^2 + (y_2 - y_1)^2$ son igual al cuadrado de las distancias entre cada coordenada, por lo tanto podemos aplicar el cálculo de distancias 1-D.

5.2.3. Localizar las rejillas vecinas

Las condiciones de frontera periódicas deben de ser consideradas tanto en el cálculo de las distancias entre las partículas como en la localización de las rejillas vecinas, para hacer esta tarea recordemos que al hacer el hashing nos apoyamos de la identación de las columnas y renglones del mallado generado para particionar el dominio. Este indexado será el que nos permita realizar la búsqueda de las casillas vecinas considerando condiciones de frontera periódicas, recordemos que los índices $(grid\_pos\_y, grid\_pos\_x)$ de una casilla son tales que $0 \leq grid\_pos\_y < tam\_y$ y $0 \leq grid\_pos\_x < tam\_x$ para la fila y la columna respectivamente.

Primero vamos a analizar el caso de una casilla interna, llamaremos casilla interna a una casilla que no está en ninguna de las fronteras, para el caso ilustrativo de 2D cada casilla tendrá 8 casillas vecinas y para localizar estas casillas vecinas observemos que si la casilla actual tiene indexados $grid\_pos\_x = i$ y $grid\_pos\_y = j$ respectivamente entonces sus casillas vecinas son tales que los índices de indexado cumplen con las siguientes ecuaciones:

$$|grid\_pos\_x - i| \leq 1 \tag{5.13}$$

$$|grid\_pos\_x - j| \leq 1 \tag{5.14}$$

Para tener una idea más clara ver la imagen[5.12] donde se muestran explícitamente los valores de $grid\_pos\_x$ y $grid\_pos\_y$ para las casillas vecinas a la casilla con valores $grid\_pos\_x = i$ y $grid\_pos\_y = j$ respectivamente.

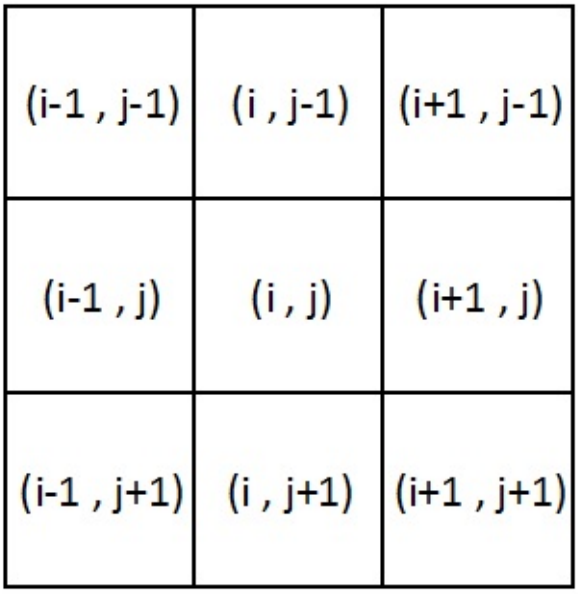

Figura 5.12: Casillas vecinas de la casilla con valores $grid\_pos\_x = i$ y $grid\_pos\_y = j$

De esta forma es muy sencillo saber cuáles son las casillas vecinas a una casilla interna con valores $grid\_pos\_x = i$ *y* $grid\_pos\_y = j$*, basta con recorrer las columnas con valores* $grid\_pos\_x = i - 1$ *,* $grid\_pos\_x = i$ *y* $grid\_pos\_x = i+1$ *y los renglones con valores* $grid\_pos\_y = j-1$ *,* $grid\_pos\_y = j$ *y* $grid\_pos\_y = j + 1$*, y así localizamos las 8 rejillas vecinas.*

Ahora que tenemos claro como localizar las casillas vecinas de una casilla interna generalizaremos esta idea para poder localizar las casillas vecinas de una casilla que se encuentre en alguna de las fronteras, es decir, para aquellas casillas tales que $grid\_pos\_x = 0$ *o* $grid\_pos\_x = tam\_x - 1$ *o* $grid\_pos\_y = 0$ *o* $grid\_pos\_y = tam\_y - 1$*. Para este tipo de casillas tenemos que considerar que:*

- *Si* $grid\_pos\_x = 0$ *entonces la casilla vecina del lado izquierdo tiene el valor* $grid\_pos\_x = tam_x - 1$*.*
- *Si* $grid\_pos\_x = tam_x - 1$ *entonces la casilla vecina del lado derecho tiene el valor* $grid\_pos\_x = 0$*.*
- *Si* $grid\_pos\_y = 0$ *entonces la casilla vecina del lado de arriba tiene el valor* $grid\_pos\_y = tam_y - 1$*.*
- *Si* $grid\_pos\_y = tam\_y - 1$ *entonces la casilla vecina del lado de abajo tiene el valor* $grid\_pos\_y = 0$*.*

Los casos de las cuatro esquinas vecinas se derivan fácilmente de estos cuatro casos considerando cada dimensión por separado como en el caso del cálculo de las distancias con condiciones de frontera periódicas.

Entonces lo único que necesitamos es poder actualizar el valor de $grid\_pos\_x$ *y* $grid\_pos\_y$ *respetando las cuatro condiciones de fronteras, notemos que movernos una casilla a la derecha o una casilla la izquierda se puede representar como sumar* 1 *o restar* 1 *al valor actual de* $grid\_pos\_x$ *y moverse una casilla arriba o una casilla abajo se puede representar como sumar* 1 *o restar* 1 *al valor actual de* $grid\_pos\_y$ *pero respetando las condiciones de frontera periódicas. Esta actualización podemos hacerla de la siguiente manera:*

1. *Llamaremos actual al valor* i *de* $grid\_pos\_x$*, y sumando al valor* $+1$ *o* -1 *según sea el caso.*
2. *Hacemos* $grid\_pos\_x = (actual + sumando + tam\_x) \,\%\, tam\_x$

de esta forma tenemos una actualización cíclica que representa las condiciones de frontera periódicas. El caso para actualizar grid_pos_y es análogo.

Esta idea podemos empaquetarla en una función de la siguiente manera:

```
//--------------------------------------------------------------------------
  /*@param actual El valor actual de grid_pos_x o grid_pos_y según sea el caso
  @param tam El valor de tam_x o tam_y según sea el caso
  @param sumando +1 para moverse a la derecha o para abajo y -1 para moverse a
  la izquierda o hacia arriba*/

  unsigned int actualizar_gridPos(unsigned int actual, unsigned int tam, int sumando){
    int aux = (int) actual + sumando + (int)tam;
    return aux % tam;
  }
//--------------------------------------------------------------------------
```

Para el caso 3D se procede de la misma forma considerando que tenemos 3 direcciones y 26 casillas vecinas.

5.3. Implementación Del Algoritmo 1

Como hemos mencionado en los capítulos anteriores las mayores complicaciones a las que nos enfrentamos al desarrollar un programa que implemente el Método de Monte Carlo, son el cálculo de la energía potencial total del sistema y el cálculo de la diferencia de energía potencial que se genera al perturbar una partícula. La dificultad de estos cálculos radica en el hecho de tener que considerar la interacción entre todos los pares de partículas pero ya hemos visto que podemos evitar tener que calcular todas estas interacciones si utilizamos un potencial de corto alcance, ya que esto nos permite introducir el uso de un radio de corte r_c y así consideramos únicamente las interacciones entre las partículas que tengan una distancia menor que r_c.

Algoritmo 1 Método de Monte Carlo con Vecindades de Verlet

for k = 1,2,..,iter_global **do**
  -Construir la vecindad de interacción de cada partícula.
  -Calcular la energía potencial del sistema $\mathcal{U}$.
  for j = 1,2,3,...,iter_local **do**
    -Elegir al azar una partícula n_i.
    -Dar un desplazamiento aleatorio a la partícula elegida.
    -Calcular la nueva energía potencial del sistema $\mathcal{U}_i$
    -Calcular $\Delta\mathcal{U} = \mathcal{U}_i - \mathcal{U}$,
    if $\Delta\mathcal{U} < 0$ **then**
      -Aceptar la transición
    else
      -Tomar aleatoriamente $\alpha \in [0, 1]$
      if $\alpha < exp[-\beta\Delta\mathcal{U}]$ **then**
        -Aceptar la transición
      else
        -Rechazar la transición
      end if
    end if
  end for
end for

El algoritmo [1] que utiliza las ideas de Verlet de realizar una lista de los vecinos de interacción para cada una de las partículas, como se muestra en la figura [5.13], puede ser implementado de múltiples

formas, hay quienes deciden calcular la distancia entre todos los pares de partículas para generar las vecindades de interacción, sin embargo hacerlo de esta forma sigue siendo muy costoso ya que solo reducimos proporcionalmente el tiempo dependiendo de la cantidad de pasos de Monte Carlo que decidamos hacer antes de actualizar las vecindades de interacción.

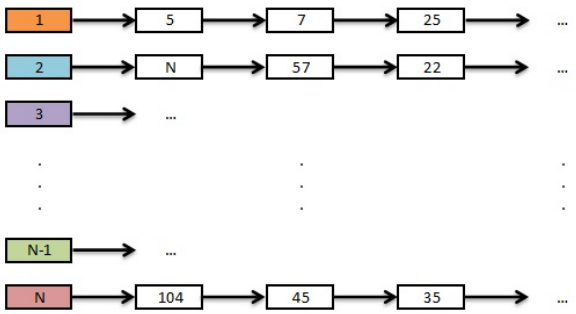

Figura 5.13: Listas de vecinos para cada partícula

Para evitar calcular la distancia entre todos los pares de partículas, nos apoyamos en el mallado del dominio descrito en la sección anterior, y en ningún momento generamos listas, ya que como hemos mencionado antes, las partículas que interactúan se encuentran en rejillas vecinas del mallado, así que recorriendo las casillas vecinas de una determinada partícula encontraremos las partículas con las que interacciona esta partícula. De esta forma reducimos el costo computacional considerablemente y la actualización de las vecindades de interacción se realiza actualizando el hashing de las partículas.

Recordar que en él algoritmo [1] debemos realizar el cálculo del potencial total del sistema cada vez que actualizamos el hashing ya que es necesario realizar un ajuste al potencial del sistema para corregir los posibles errores que cometemos al realizar cálculos intermedios sin actualizar el hashing.

5.3.1. Cálculo de la energía potencial total del sistema

La energía potencial del sistema está relacionada con el potencial de interacción interatómica entre las partículas, nos otros hemos elegido trabajar con el potencial de Lennard-Jones que describe la interacción entre pares de partículas, así, la energía potencial total del sistema se calcula como sigue:

$$\mathcal{U}(r^N) = \sum_{i=1}^{N} \sum_{j>i}^{N} LJ(dist(r_i, r_j)) \tag{5.15}$$

Donde r^N es el sistema de partículas, r_i y r_j son partículas y dist es la distancia entre las partículas.

Pero como ya sabemos que no es necesario calcular la interacción entre todos los pares de partículas podemos realizar el cálculo como sigue:

$$\mathcal{U}(r^N) = \sum_{i=1}^{N} \sum_{j \in V_i} LJ(dist(r_i, r_j)) \tag{5.16}$$

Donde V_i es la vecindad de interacción de la partícula i y como esta vecindad queda definida por el radio de corte r_c podemos realizar el cálculo como sigue:

$$\mathcal{U}(r^N) = \sum_{i=1}^{N} \sum_{j:i<j, dist(r_i,r_j)<r_c} LJ(dist(r_i, r_j)) \tag{5.17}$$

De esta forma el cálculo será más rápido, cabe mencionar que para implementaciones en paralelo puede resultar más conveniente calcular dos veces la interacción entre cada par de partículas y dividir entre 2 en lugar de realizar la comparación ($i < j$), si fuera así tendríamos que:

$$2\mathcal{U}(r^N) = \sum_{i=1}^{N} \sum_{j:dist(r_i,r_j)<r_c} LJ(dist(r_i, r_j)) \tag{5.18}$$

Ya que dependiendo de que método de cómputo paralelo se utilice las condicionales suelen provocar problemas de tener procesos detenidos.

El simple hecho de calcular la energía potencial del sistema utilizando el concepto de radio de corte es en sí mismo una mejora en procesamiento y tiempo para el método de Monte Carlo, pero podemos hacer aun más mejoras al cálculo de la energía potencial total del sistema y obtener un mejor rendimiento.

En este punto sabemos que dada una partícula r_i, para saber que partículas r_j están en su vecindad de interacción, utilizamos el mallado y el hashing para solo tener que buscar en las rejillas vecinas a la rejilla correspondiente a la partícula r_i, claro que también tenemos que buscar en la misma rejilla de la partícula r_i, teniendo un total de 9 rejillas para el caso 2D y 27 rejillas para el caso 3D. Pero si consideramos como se encuentra este mallado y los datos de cada una de las partículas en la memoria veremos que podemos mejorar el rendimiento ordenando de una manera conveniente los datos de las partículas en la memoria RAM.

Recordar que para guardar los datos de las partículas utilizamos un vector para cada dato y no un objeto o estructura para cada partícula. Para el caso 2D estos vectores son los siguientes:

- *x , la coordenada x de la partícula.*
- *y , la coordenada y de la partícula.*
- *i_r , el índice de la rejilla en la que se encuentra la partícula.*
- *i_g , el índice global de la partícula.*
- *$grid\_pos\_x$, el numero de columna correspondiente a la rejilla de la partícula.*
- *$grid\_pos\_y$, el numero de renglón correspondiente a la rejilla de la partícula.*

Para entender mejor la idea del ordenamiento mostraremos el almacenamiento en memoria de las primeras cuatro características. Como la configuración inicial del sistema es generada de forma aleatoria estos vectores estarían almacenados en memoria como se muestra en la imagen [5.14].

Observamos que los datos de las partículas que pertenecen a una misma rejilla se encuentran en localidades de memoria separadas, a primera vista esto no parece malo, pues al fin y al cabo lo que nos interesa es conocer la rejilla a la que pertenece cada partícula, pero, en el momento en el que busquemos todas las partículas que perteneces a una determinada rejilla tendríamos que recorrer todo el arreglo de particular y ver si pertenece o no a la rejilla de nuestro interés. Podría ser que para cada rejilla hiciéramos una lista de las partículas que se encuentran en esa rejilla y así ya sabríamos a que lugares de los arreglos dirigirnos, sin embargo para poder hacer estas listas sería necesario recorrer los arreglos de las partículas cada vez que actualicemos el hashing, para evitar esto ordenaremos las partículas en memoria de acuerdo al índice en rejilla, tener cuidado de modificar las posiciones en todos los arreglos correspondientes a cada característica de las partículas.

Para realizar el ordenamiento utilizamos el algoritmo quick-sort, pero para evitar la copia masiva de datos, en un primer paso solo vamos a realizar los cambios en el arreglo del índice global i_g respecto a los cambios que suceden en el arreglo de índice en rejilla i_r mientras este último es ordenado. En un segundo paso ordenamos todos los datos restantes utilizando las nuevas posiciones del índice global, es decir, si después de ordenar los arreglos i_g e i_r respecto a i_r, en la posición 99 del arreglo i_g esta

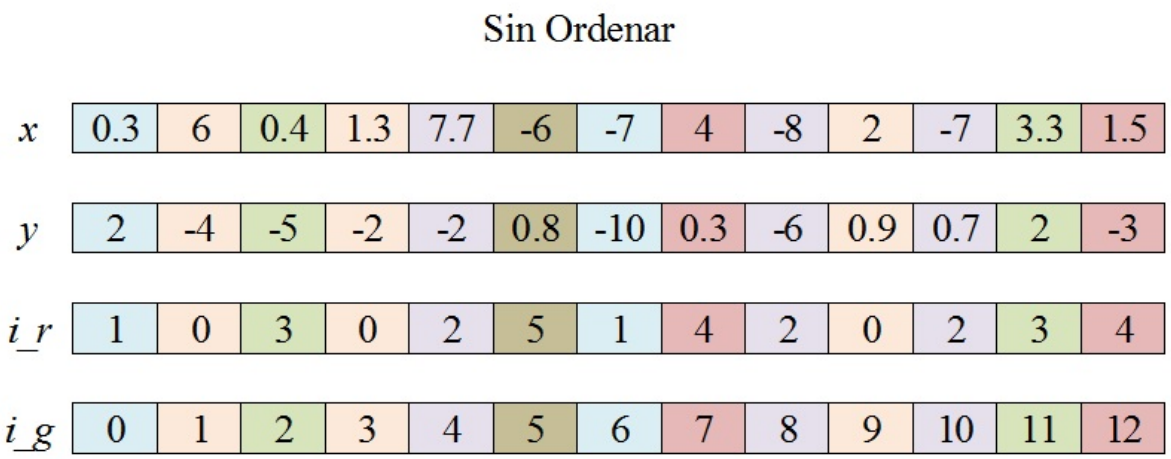

Figura 5.14: Datos de las partículas almacenados en vectores

el dato 1000 *eso significa que en los arreglos ordenados todos los datos correspondientes a la partícula con índice global* 1000 *deben estar en la posición* 99 *de los arreglos.*

Después de realizar el ordenamiento respecto al índice en rejilla i_r *los datos de las partículas se verán como se muestra en la imagen [5.15], de esta manera lo único que nos resta por hacer en generar un arreglo que nos indique los índices donde comienza cada rejilla.*

Figura 5.15: Datos de las partículas almacenados en vectores, ordenados respecto al índice en rejilla

Para generar el arreglo que nos indique en qué posición de los datos de las partículas comienza cada rejilla, solo necesitamos recorrer el arreglo de índice en rejilla i_r *y buscar los cambios de datos en posiciones adyacentes. Pero será muy importante que este arreglo tenga una dimensión* $n\_rejillas + 1$*, donde* $n\_rejillas$ *es la cantidad de rejillas en el mallado, esto con la idea de que en la última posición del arreglo de inicios pongamos el número total de partículas* N*, de esta forma podremos calcular de forma rápida la cantidad de partículas en cada rejilla, esta idea se muestra en la imagen [5.16] considerando los datos de ejemplo de la imagen [5.15]. Observar que por ejemplo para el caso de la rejilla con índice* 2*, la posición donde inician los datos de las partículas con índice en rejilla* 2 *se encuentra en la posición* 2 *del arreglo de inicios y más aun, si a la cantidad en la posición* 3 *del arreglo de 'indices le restamos el valor en la posición* 2 *del arreglo de índices, entonces obtenemos* $8 - 5 = 3$ *que es la cantidad de partículas con índice en rejilla* 2*.*

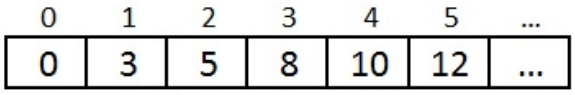

Figura 5.16: Arreglo de los inicios es memoria de cada rejilla

Con los datos de las partículas ordenados en memoria respecto al índice en rejilla, entonces cuando necesitemos los datos de las partículas con un i_r determinado, digamos j, bastara con recorrer los arreglos de datos desde la posición inicios$[j]$ hasta la posición inicios$[j+1]$, para ser más claros con esta idea mostramos el siguiente fragmento de código para recorrer los arreglos de los datos de las partículas correspondientes a la rejilla j:

```
int principio = inicios[j];
int tam = inicios[j+1] - principio;
for(int i = principio ; i < tam; i++){
  ...
  ...
  ...
}
```

La idea de ordenar los datos de las partículas en memoria, además de darnos la ventaja de saber de forma rápida que partículas se encuentran en cada rejilla, también nos ofrece ventajas computacionales ya que al recorrer cada rejilla aprovechamos la memoria cache almacenando en ella datos de partículas en la misma rejilla, los cuales serán solicitados por el procesador para ser procesados, de esta manera aceleramos la lectura de los datos de las partículas que pertenecen a la misma rejilla, y para las implementaciones en paralelo el aprovechamiento de la memoria caché es de una importancia relevante para realizar programas optimizados y rápidos en procesamiento.

La gran ventaja de utilizar la memoria cache es que cuando se necesiten datos de partículas en la misma rejilla no será necesario ir hasta la memora RAM por los datos de cada una de las partículas si no que cuando se solicite la información de la primer partícula en la rejilla, el sistema enviara a la memoria cache un bloque de memoria en el que se encontraran los datos de las partículas que se encuentran en la misma rejilla, de tal forma que cuando se necesiten los datos de las otras partículas que pertenecen a la misma rejilla ya no será necesario ir hasta la memoria RAM por la información, ya que los datos se encontraran en la memoria cache. Para convencerse de esta idea el lector puede realizar la siguiente prueba codificada en C/C++.

El primer código calcula la suma de los elementos de una matriz de n x m, recorriendo por columnas:

```
double Suma(double ** x, double * n, double *m){
  double sum = 0.0;

  for(int j = 0; j < m ; j++)
    for(int i = 0; i < n; i++)
       sum += x[i][j];

  return sum;
}
```

El segundo código calcula la suma de los elementos de una matriz de n x m, recorriendo por filas:

```
double Suma(double ** x, double * n, double *m){
  double sum = 0.0;

  for(int i = 0; i < n ; i++)
    for(int j = 0; j < m; j++)
       sum += x[i][j];
```

```
    return sum;
}
```

El lector puede ejecutar estos dos códigos y medir el tiempo para convencerse de que el segundo código es mucho más rápido ya que los datos de los renglones se encuentran en localidades adyacentes de la memoria RAM, mientras que los datos de las columnas se encuentran en localidades separadas de la memoria RAM.

Con las ideas y técnicas de programación indicadas anteriormente obtenemos una manera de calcular la energía potencial total del sistema de una manera, rápida y eficiente computacionalmente. Esto es muy importante ya que en el algoritmo [1] en el que aplicamos las idea de Verlet será necesario calcular la energía total del sistema en cada iteración global.

5.3.2. Cálculo del cambio en la energía potencial al perturbar una partícula

Ya que el Método de Monte Carlo es un método de carácter estocástico en el que en cada paso se perturba aleatoriamente una partícula elegida igualmente aleatoriamente, necesitamos calcular el cambio que sufre la energía potencial total del sistema al perturbar una partícula para posteriormente aplicar el algoritmo de Metrópolis para aceptar o rechazar la transición en el espacio fase.

Para calcular el cambio de la energía potencial al perturbar una partícula, necesitamos observar la formula con la que calculamos la energía potencial:

$$\mathcal{U}(r^N) = \sum_{i=1}^{N} \sum_{j:i<j, dist(r_i,r_j)<r_c} LJ(dist(r_i, r_j)) \tag{5.19}$$

En la primer suma observamos que cada partícula tiene una aportación al cálculo de la energía potencial total, sin embargo cuando el valor del índice de la primer suma se fija a un cierto valor, digamos 20*, solo estaremos calculando las interacciones de la partícula con índice global* 20 *con las partículas que tengan un índice global mayor a* 20 *ya que las interacciones con las partículas con un índice menor ya fueron calculadas en las sumas anteriores. Este comentario es importante porque al calcular el cambio de la energía potencial ocasionado por la perturbación de una partícula necesitaremos considerar las interacciones entre la partícula perturbada y todas las partículas dentro de su vecindad de interacción.*

Lo que necesitamos hacer para calcular el cambio de la energía potencial total del sistema, se divide en los siguientes pasos, suponiendo que la partícula perturbada tiene índice global i:

1. *Calcular la aportación de la partícula que será perturbada $\mathcal{U}_i^-$, considerando la interacción con todas las partículas dentro de su vecindad de interacción, determinada por el radio de corte.*

2. *Perturbar aleatoriamente la partícula elegida.*

3. *Calcular la aportación de la partícula ya perturbada $\mathcal{U}_i^+$, considerando la interacción con todas las partículas dentro de su vecindad de interacción, determinada por el radio de corte.*

4. *Hacer $\Delta\mathcal{U}(r^N) = \mathcal{U}_i^+ - \mathcal{U}_i^-$:*

Donde:

$$\mathcal{U}_i = \sum_{j \in V_i} LJ(dist(r_i, r_j)) \tag{5.20}$$

El súper índice $+$ o $-$ corresponde a si es antes o después de ser perturbada respectivamente.

Es importante mencionar que la vecindad de interacción de la partícula perturbada i no es la misma antes de ser perturbada que después de ser perturbada, algunas partículas entran y otras salen de esta vecindad de interacción.

Claro que para realizar estos cálculos nos apoyamos del particionamiento en rejilla para facilitar los cálculos, no sin antes mencionar que puede ser posible que la partícula perturbada cambie de rejilla, situación que nos generara errores en el cálculo, ya que a pesar de que consideremos el cambio de la rejilla al momento de calcular $\mathcal{U}_i^+$ los datos de la partícula perturbada seguirán en la misma posición de memoria durante esa iteración global y serán movidos a las posiciones de memoria correspondiente a la nueva rejilla hasta la siguiente iteración global, provocando que si en la misma iteración global se elige perturbar una partícula en una rejilla adyacente a la que será la nueva rejilla de la partícula i entonces no consideraremos la interacción de la nueva partícula a perturbar y la partícula i ya que en esa misma iteración global los datos de la partícula i estarán en las localidades de memoria correspondientes a la antigua rejilla de la partícula i.

Al cometer los errores en el cálculo de la energía potencial del sistema es necesario realizar un ajuste en cada iteración global, este hecho es lo que hace que el algoritmo [1] sea de complejidad cuadrática ya que el cálculo de la energía potencial es de orden cuadrático por la doble suma.

5.4. Implementación Del Algoritmo 2

La diferencia principal entre el algoritmo [1] y el algoritmo [2] es que en el algoritmo [2] calculamos el hashing de las partículas y la energía potencial total del sistema una sola vez, después de esto solo calculamos los cambios en la energía potencial al perturbar las partículas. Para poder hacer esto necesitamos que en el caso de que la partícula perturbada cambie de rejilla debemos de mover los datos de la partícula perturbada a localidades de memoria correspondientes a la nueva rejilla.

La pregunta natural es:

¿Cómo mover los datos de la partícula perturbada a localidades de memoria correspondientes a la nueva rejilla?

El problema al que nos enfrentamos es que las localidades de memoria correspondientes a cada una de las rejillas son espacios contiguos como podemos ver en la imagen [5.17], sin espacios vacíos para poder poner más datos.

Figura 5.17: Datos de las partículas almacenados en vectores, ordenados respecto al índice en rejilla

La respuesta es simple, asignemos a cada rejilla un búffer de tamaño suficiente para poder almacenar los datos de las partículas que se encuentran en esa rejilla y además espacio disponible para poder alma-

cenar los datos de las partículas que ingresen a la rejillas durante la simulación de Monte Carlo.

Para poder llenar el búffer primero necesitamos realizar el ordenamiento respecto a i_r de la misma forma como se realizo en el algoritmo [1], después debemos encontrar los inicios de cada rejilla en el arreglo ordenado y posteriormente llenar el búffer, considerando que cada rejilla tiene asignadas la misma cantidad de localidades de memoria en el búffer por lo que el tamaño del búffer tam_buffer para cada partícula será el mismo y puede ser determinado con la densidad del sistema.

Al principio de la simulación los datos de las partículas estarán en memoria como se muestra en la imagen [5.18] y después de algunos pasos de simulación podría verse como se muestra en la imagen [5.19].

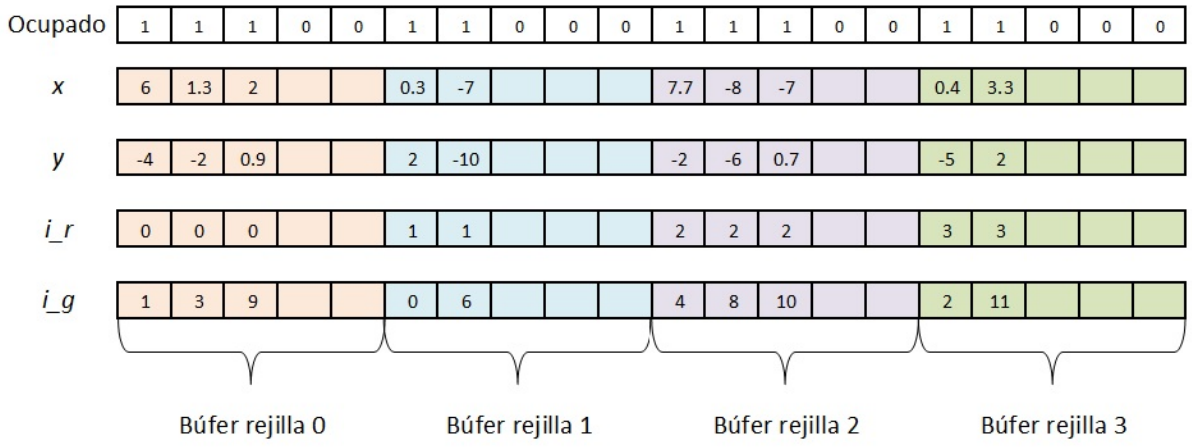

Figura 5.18: Buffer inicial

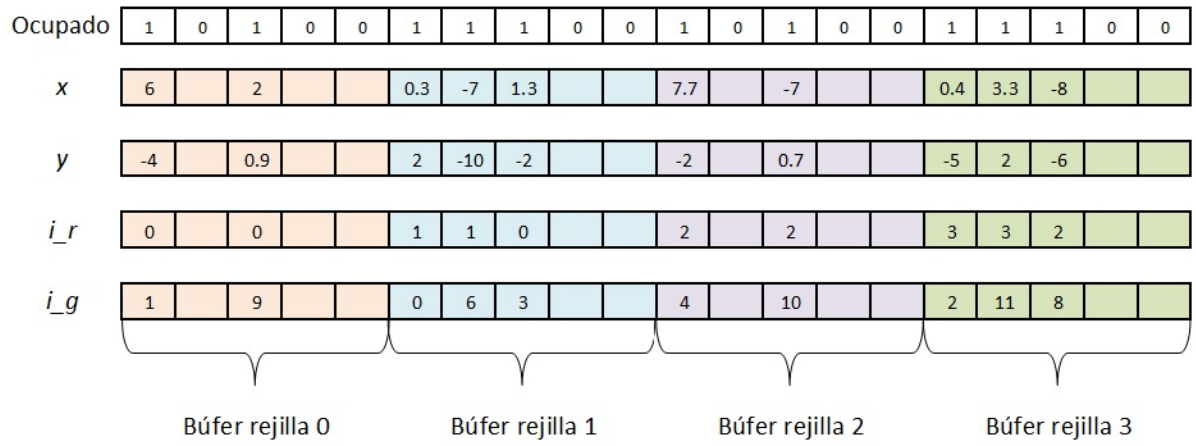

Figura 5.19: Buffer posterior

Notar que en la imagen [5.19] hay lugares vacíos donde inicialmente había datos de partículas, esto es porque dentro del búffer asignado a cada rejilla en lugar de ir recorriendo los datos hacia la izquierda cuando una partícula sale de la rejilla, solo se deja vacío ese lugar ya que el recorrer los datos sería muy costoso y además innecesario ya que dentro de la dinámica habrá otra partícula que entrara a la rejilla y esta posición vacía puede ser asignada a los datos de la nueva partícula que entra a la rejilla. Para saber qué lugares están ocupados y cuales están vacíos basta con tener un arreglo de booleanos e ir actualizando al ocupar o desocupar rejillas. Así cuando una nueva partícula entra a la rejilla se le asigna el primer lugar vacío dentro de la rejilla recorriendo de izquierda a derecha.

Con la idea del búffer el cálculo de la energía potencial total del sistema sigue teniendo una complejidad cuadrática, sin embargo solo lo calculamos una vez. Para el cálculo de la diferencia en la energía potencial del sistema al perturbar una partícula recordemos que necesitamos recorrer las casillas adyacentes a la rejilla de la partícula perturbada y la misma rejilla de la partícula perturbada, teniendo un total de 9 rejillas para el caso 2D y 27 rejillas para el caso 3D, pero como el búffer de cada rejilla tiene el mismo tamaño tam_buffer, entonces cada vez que perturbamos una partícula realizamos a lo más el cálculo de $2 * 9(tam\_buffer)$ o $2 * 27(tam\_buffer)$ interacciones según sea el caso, el multiplicar por dos es porque calculamos las interacciones antes y después de perturbar la partícula.

Es claro entonces que la complejidad del algoritmo [2] es lineal, pues en cada iteración realizamos la misma cantidad de operaciones que no depende del valor de N.

5.5. Implementación en Paralelo

Las implementaciones descritas en las secciones anteriores han sido diseñadas de tal manera que se aprovechen las ventajas del cómputo en paralelo, hemos mencionado que la mayor ventaja del cómputo paralelo es realizar operaciones independientes, así que para realizar una implementación en paralelo, ya sea con OpenMP o CUDA, hay que identificar que operaciones son independientes.

Las operaciones independientes mas claras en las implementaciones descritas son:

- *El hashing de las coordenadas de la partícula al índice en rejilla.*
- *Cálculo de la distancia entre partículas.*

Estas dos operaciones son las que podemos paralelizar de forma mas simple pues los cálculos son completamente independientes.

En el caso del cálculo de la energía potencial total del sistema:

$$\mathcal{U}(r^N) = \sum_{i=1}^{N} \sum_{j>i}^{N} LJ(dist(r_i, r_j)) \tag{5.21}$$

Donde r^N es el sistema de partículas, r_i y r_j son partículas y dist es la distancia entre las partículas.

Podemos utilizar cada core o cada thread de procesamiento para calcular la aportación de cada partícula a la energía potencial total del sistema. Esta operación aunque globalmente no es independiente si es independiente una vez que se fija el índice de la partícula, posteriormente hay que manejar la concurrencia entre los cores o los threads para realizar la suma total.

Cuando necesitamos calcular la diferencia en la energía potencial del sistema ocasionada por la perturbación de una partícula, necesitamos calcular la siguiente suma:

$$\mathcal{U}_i(r^N) = \sum_{j \in v_i}^{N} LJ(dist(r_i, r_j)) \tag{5.22}$$

Donde v_i es la vecindad de interacción de la partícula i.

En este caso podemos utilizar cada core o cada thread para calcular cada sumando, pues en cada sumando intervienen partículas diferentes.

Como observaremos en el capítulo de resultados, la complejidad lineal nos brinda mayores beneficios que la implementación en paralelo.

Capítulo 6

Resultados

Las soluciones numéricas de un sistema físico se basan en un modelo matemático a partir del cual se obtienen y resuelven las ecuaciones integro-diferenciales que describen un estado de dicho sistema. No obstante, existen problemas muy complejos, como las interacciones nucleares, que no pueden ser resueltos empleando modelos determinísticos. Con Monte Carlo, los procesos físicos son simulados teóricamente sin necesidad de resolver completamente las ecuaciones del sistema. Sin embargo es necesario conocer las funciones de densidad de probabilidad (pdf por sus siglas en inglés) que describen el comportamiento del sistema. Se puede estimar resultados incluso si el problema no tiene un contexto probabilístico.

Hasta hace pocos años la manera en la que se realizaban las simulaciones aplicando el método de Monte Carlo era de forma secuencial, y como la parte primordial para el sistema son las distancias entre cada par de partículas lo primero que se hacía era calcular todas las distancias y almacenarlas en memoria, sin embargo esto consume demasiados recursos en memoria ya que el numero de distancias para un sistema de N partículas es $(N-1)(N)/2$ lo cual hace que la cantidad de memoria necesaria para almacenar las distancias crezca de forma cuadrática con respecto aumentamos N. Otra limitante era que como el programa se realizaba secuencialmente el tiempo necesario para procesar N partículas también crece exponencialmente. Con el computo paralelo podemos utilizar distintas técnicas para obtener los mismos resultados reduciendo considerablemente el uso de los recursos computacionales y el tiempo necesario de procesamiento.

El método de Monte Carlo (MC) es quizá el método más usado en mecánica estadística computacional. En particular la técnica de Metrópolis-Monte Carlo ha sido muy usada en el estudio de líquidos, esto nos motivo a realizar la implementación del método de Monte Carlo en paralelo utilizando dos paradigmas de programación, OpenMP y CUDA, esta idea surgió por el gran desarrollo en las técnicas del computo en paralelo, en la actualidad muchos problemas que requieren de un alto poder de computo han sido resueltos utilizando computo paralelo.

Por la naturaleza del método de Monte Carlo, existen varias partes del método que se pueden dividir en secciones independientes para ser procesadas, por ejemplo en el cálculo de la energía potencial del sistema:

$$\mathcal{U}(r^N) = \sum_{i=1}^{N} \sum_{j>i}^{N} LJ(dist(r_i, r_j)) \tag{6.1}$$

Ya que la energía es expresada como una suma, podemos realizar el cálculo de cada suma parcial (fijando el valor de i de la primer suma) de forma independiente, pues ninguna suma parcial depende de la otra. De igual forma el cálculo del hashing para saber en qué rejilla se encuentra cada una de las partículas, se puede realizar de forma independiente para cada partícula.

Como hemos mencionado, la parte más costosa en el Método de Monte Carlo es el cálculo de la energía potencial total del sistema, así que si podemos realizar este cálculo de forma paralela entonces obtendremos una gran ganancia en tiempo de ejecución del algoritmo, sin embargo al realizar el estudio

detallado del método observamos que se podían realizar mejoras a la implementación del algoritmo [1] antes de optar por programar un algoritmo en paralelo. Estas mejoras dan lugar al algoritmo [2] el cual como veremos en esta sección, además de ser de complejidad lineal tiene óptimos resultados aun en la versión serial.

En esta sección presentaremos los resultados obtenidos a partir de la implementación de los algoritmos [1] y [2] en sus versiones seriales, OpenMP y CUDA.

Dada la naturaleza del Método de Monte Carlo y las características de implementación descritas en los algoritmos [1] y [2] se puede pensar que las diferencias mínimas en los algoritmos no pueden generar grandes diferencias en el tiempo de ejecución de cada uno de ellos, sin embargo como veremos las versiones de cada uno de estos algoritmos nos presentara diferentes resultados en términos del tiempo de ejecución, dicho de otra manera, algunas implementaciones tardaran más que otras para obtener los mismos resultados numéricos.

6.1. Algoritmo 1

A simple vista el método de Monte Carlo es muy simple de implementar, sin embargo como hemos mencionado anteriormente, el cálculo de la energía potencial es el paso más caro en cuanto a poder de cómputo y tiempo se refiere, a pesar de la simplicidad del potencial de interacción entre las partículas, la dificultad radica en el hecho de tener que calcular las distancias entre todos los pares de partículas, y es que, la cantidad de pares crece de forma exponencial respecto al número de partículas. Existen distintas metodologías para reducir el costo del cálculo del potencial, nosotros implementamos un particionamiento del dominio de tal forma que las partículas vecinas se encuentren en localidades cercanas de la memoria (RAM en el caso de realizar un proceso serial o utilizar OpenMP), de igual forma aplicamos cómputo paralelo para calcular de forma independiente el aporte que cada partícula hace a la energía potencial total.

En la literatura podemos encontrar que para evitar calcular las distancias entre los pares de partículas en cada paso del método de Monte Carlo, podemos hacer uso de un sistema de vecindario en el que cada partícula contiene la lista de sus partículas vecinas, esto considerando que se trabaja con un potencial de corto alcance en el que existe un radio de corte r_c a partir del cual el potencial es prácticamente cero, o en otras palabras, cada partícula interactúa únicamente con sus vecinos más próximos (por lo general no se excede de quintos vecinos). Se propone calcular el sistema de vecindario cada cierto número de pasos con la suposición de que en pocos pasos el sistema de vecindario se altera muy poco ya que en cada paso se perturba una sola partícula.

Las ideas anteriores dan lugar al algoritmo 1 el cual hemos mencionado anteriormente es de orden cuadrático.

El cálculo de la energía potencial en cada iteración global es necesaria para corregir circunstancialmente los errores cometidos con el método de metrópolis, ya que como suponemos que en cada perturbación la probabilidad de que la partícula perturbada cambie de vecindad es relativamente baja, cuando calculamos el cambio en la energía al perturbar una partícula, suponemos que la partícula perturbada se encuentra dentro de la misma vecindad que la partícula sin perturbación, sin embargo aunque la probabilidad sea mínima puede ocurrir que realmente la partícula cambie de vecindad y en este caso estaríamos cometiendo un error en el cálculo de la energía potencial, este error se puede eliminar si consideramos el cambio de vecindad de la partícula perturbada al momento de calcular el cambio en la energía, sin embargo, se siguen cometiendo errores en el cálculo ya que en caso de aceptar una transición al perturbar la partícula i, modificaremos las coordenadas de la partícula i que haya sido perturbada, y se actualizara su vecindad pero en memoria la partícula i seguirá perteneciendo a la vecindad a la que pertenecía antes de ser perturbada por lo que sí es el caso de que en la misma iteración global se elija otra partícula j que se encuentre en interacción con la partícula i antes elegida se cometerá un error en el cálculo del cambio en la energía potencial ya que la partícula i no estará en la memoria correspondiente a la vecindad actual a la que pertenece la partícula i si no que estará en la memoria correspondiente a la vecindad a la que pertenecía

Algoritmo 1 Método de Monte Carlo con Vecindades de Verlet

```
for k = 1,2,..,iter_global do
  -Construir la vecindad de interacción de cada partícula.
  -Calcular la energía potencial del sistema U.
  for j = 1,2,3,...,iter_local do
    -Elegir al azar una partícula n_i.
    -Dar un desplazamiento aleatorio a la partícula elegida.
    -Calcular la nueva energía potencial del sistema U_i
    -Calcular ΔU = U_i − U,
    if ΔU < 0 then
      -Aceptar la transición
    else
      -Tomar aleatoriamente α ∈ [0, 1]
      if α < exp[−βΔU] then
        -Aceptar la transición
      else
        -Rechazar la transición
      end if
    end if
  end for
end for
```

antes de ser perturbada, por lo que la partícula i se encontrara en la localidad de memoria adecuada hasta la siguiente iteración global cuando se actualice el sistema de vecindario, es por esta razón, que sigue siendo necesario realizar el cálculo de la energía potencial en cada iteración global para corregir estos errores.

Para este algoritmo se implementaron tres versiones, serial, OpenMP y CUDA, en cada implementación se tuvo cuidado de explotar al máximo las características del algoritmo y de cada metodología de programación para obtener los mejores resultados en tiempo de ejecución.

En la imagen [6.1] podemos observar los tiempos de ejecución en segundos para el algoritmo [1], en las columnas de la izquierda se muestran los tiempos para un sistema con 804335 *partículas, en las columnas de la derecha se muestran los tiempos para un sistema con* 1709505 *partículas, en cada grupo de columnas se representan los tiempos para los programas en serial, OpenMP y CUDA en azul, rojo y verde respectivamente.*

En la imagen [6.1] observamos claramente que las implementaciones en paralelo son más eficientes que las implementaciones en serial, en particular observamos que la implementación en CUDA es más eficiente que la implementación con OpenMP.

Las diferencias que observamos en los tiempos de ejecución para cada una de la implementación se deben en gran parte al tiempo que se tarda en calcular la energía potencial total del sistema en cada iteración global. En el caso de OpenMP y CUDA la diferencia en tiempos se debe a que CUDA puede realizar más procesos independientes que OpenMP. cabe mencionar que estos programas se ejecutaron en la misma estación de trabajo tonatiuh que cuenta con 12 *núcleos de procesamiento para OpenMP y la tarjeta gráfica NVIDIA GTX-480.*

Si tuviéramos que hacer la elección de un programa en este momento, es claro que elegiríamos el programa hecho en CUDA, pero antes de tomar esa decisión veremos los resultados obtenidos con el algoritmo [2].

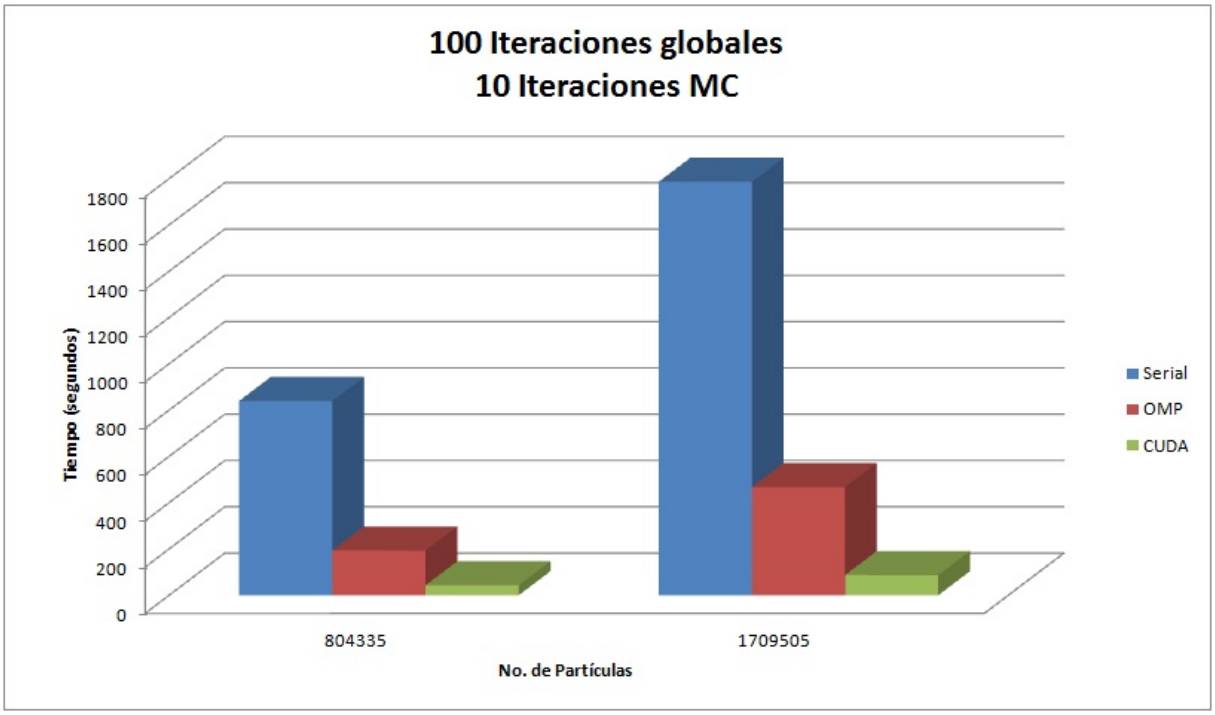

Figura 6.1: Tiempos de ejecución para el algoritmo 1

6.2. Algoritmo 2

Considerando que al utilizar el algoritmo [1] se cometen ciertos errores numéricos y además es muy costoso calcular el sistema de vecindario y la energía potencial total del sistema, hemos propuesto una modificación al algoritmo [1], actualizando en memoria los datos de la partícula perturbada al mismo momento en que el movimiento es aceptado. Entre las ventajas que ofrece esta propuesta se encuentra el hecho de que solo tenemos que calcular una vez el sistema de vecindario y la energía potencial total del sistema.

Para actualizar al vuelo los datos correspondientes a la partícula perturbada en caso de ser aceptado el movimiento necesitamos un búffer en memoria para cada vecindad con el tamaño suficiente para alocar la información de todas las partículas del vecindario en cada iteración.

Al realizar las modificaciones mencionadas al algoritmo [1], obtenemos el algoritmo [2].
Con este nuevo algoritmo evitamos calcular la energía potencial del sistema cada determinado número de iteraciones con lo que se reduce de forma significativa el tiempo de procesamiento del método de Monte Carlo.

En la imagen [6.2] podemos observar los tiempos de ejecución en segundos para el algoritmo [2], en las columnas de la izquierda se muestran los tiempos para un sistema con 804335 *partículas, en las columnas del centro se muestran los tiempos para un sistema con* 1709505 *partículas, en las columnas de la derecha se muestran los tiempos para un sistema con* 11520985 *partículas, en cada grupo de columnas se representan los tiempos para los programas en serial, OpenMP y CUDA en rojo, verde y morado respectivamente.*

Al igual que en el caso de la imagen [6.1] en la imagen [6.2] observamos claramente que las implementaciones en paralelo son más eficientes que las implementaciones en serial, en particular observamos que la implementación en CUDA es más eficiente que la implementación con OpenMP, sin embargo en este caso, la diferencia entre OpenMP y CUDA no es tan considerable como en el caso de la imagen [6.1], esta diferencia se debe a que en el algoritmo [2] calculamos la energía potencial del sistema una sola vez a diferencia del algoritmo [1] en el que calculamos la energía potencial del sistema en cada iteración global.

Algoritmo 2 Método de Monte Carlo con Rejilla y Búfer

-Hacer el particionamiento del dominio.
-Construir las vecindades de interacción
-Calcular la energía potencial total del sistema $\mathcal{U}$.
for j = 1,2,3,...,iter_global **do**
  -Elegir aleatoriamente una partícula n_i
  -Dar un desplazamiento aleatorio a la partícula elegida.
  -Calcular la nueva energía potencial del sistema$\mathcal{U}_i$
  -Calcular $\Delta\mathcal{U} = \mathcal{U}_i - \mathcal{U}$,
  if $\Delta\mathcal{U} < 0$ **then**
    -Aceptar la transición
  else
    -Elegir aleatoriamente un numero $\alpha \in [0, 1]$
    if $\alpha < exp[-\beta\Delta\mathcal{U}]$ **then**
      -Aceptar la transición
    else
      -Rechazar la transición
    end if
  end if
end for

Es importante mencionar que a pesar de la implementación del algoritmo [2] utilizando CUDA es mas rápida que la implementación echa en OpenMP, es notable que la diferencia no es tan considerable, además de las modificaciones que se utilizan en el algoritmo [2] hay grandes diferencias conceptuales entre OpenMP y CUDA, estas diferencias residen en el hecho de que OpenMP utiliza los recursos directos del sistema (procesador y memoria RAM), mientras que CUDA utiliza recursos de la tarjeta grafica, por lo que al utilizar la versión con CUDA estamos limitados a los recursos de la tarjeta gráfica, cosa que no sucede con OpenMP

La única ventaja que obtenemos al utilizar CUDA es el cálculo de la energía potencial total del sistema, ya que en las iteraciones posteriores al cálculo del potencial, solo utilizamos una cantidad reducida de las capacidades de CUDA, pues en cada iteración calculamos la interacción de la partícula perturbada con las 27 rejillas a su alrededor, y la cantidad de operaciones necesarias para realizar esta tarea es considerablemente bajo en comparación con las capacidades de CUDA, esto nos ocasiona problemas de desperdicio de recursos por lo que quizá sea más conveniente utilizar únicamente OpenMP o un híbrido entre OpenMP y CUDA, podemos utilizar CUDA para calcular la energía potencial inicial y OpenMP para realizar las iteraciones de Monte Carlo.

6.3. Comparación de metodologías

En la sección anterior hemos mostrado evidencias claras de que las implementaciones en paralelo son más eficientes que las implementaciones en serial pero no hemos visto de forma clara la comparación entre el algoritmo[1] y el algoritmo[2].

En la imagen [6.3] podemos observar los tiempos de ejecución en segundos para los algoritmos [1] y [2] en color azul y rojo respectivamente, en las columnas de la izquierda se muestran los tiempos para la implementación en serial, al centro la implementación con OpenMP y a la derecha la implementación en CUDA. Observamos claramente la ventaja que se obtiene con la modificación que se presenta en el algoritmo [1].

Con base en los resultados mostrados en esta sección podemos asegurar que las modificaciones propuestas para las implementaciones tradicionales del método de Monte Carlo brindan una gran ventaja y que además si realizamos la programación del nuevo algoritmo aplicando metodologías de computo paralelo

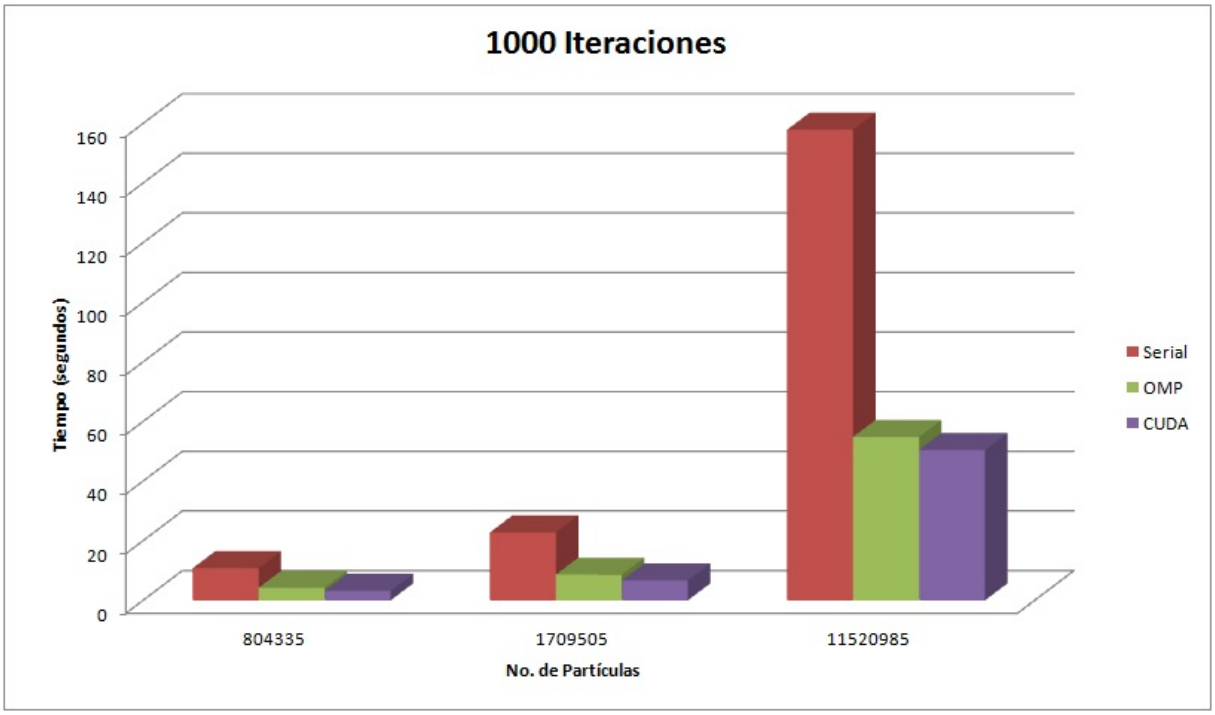

Figura 6.2: Tiempos de ejecución para el algoritmo 2

podemos tener una mejora considerable en el rendimiento de los algoritmos, este resultado es muy importante ya que una de las limitaciones actuales al utilizar el método de Monte Carlo en una simulación es que no se puede trabajar con gran numero de partículas, pero con este nuevo algoritmo podemos trabajar con valores de N en el rango de decenas de millones.

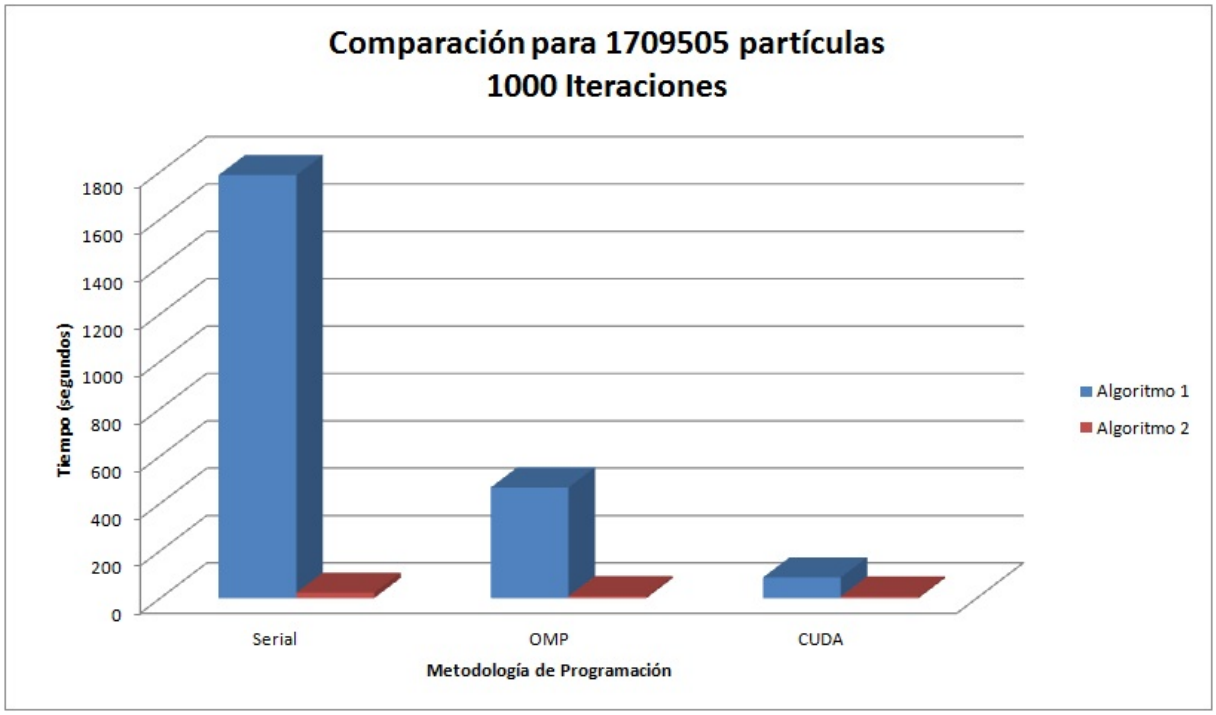

Figura 6.3: Comparación de tiempos de ejecución entre los algoritmos 1 y 2

Capítulo 7

Verificación y validación

En los capítulos anteriores hemos realizado un estudio detallado del Método de Monte Carlo y su implementación con dos algoritmos de los cuales mostramos resultados respecto a los tiempos de ejecución, en esta sección realizaremos la verificación y validación de estos algoritmos comparando nuestros resultados con datos teóricos y experimentales.

7.1. Introducción

Una de las tareas más importantes y difíciles a las que se debe enfrentar los programas que realizan la simulación de un fenómeno real es la validación y verificación de los mismos. Se debe trabajar lo más próximo posible a la realidad en la etapa de validación para reducir (o eliminar, si es posible) el escepticismo respecto a los resultados obtenibles mediante el proceso de modelado y simulación. Los objetivos fundamentales del proceso de validación son:

- *Producir un modelo que represente el comportamiento del sistema real lo suficientemente próximo como para que el modelo pueda sustituir al sistema con el objetivo de experimentar determinados aspectos del mismo.*
- *Programar los simuladores respetando las características reales del sistema a simular y teniendo riguroso cuidado en la asignación de las unidades correctas para las variables involucradas en el sistema a simular, posteriormente realizar la verificación y validación de los simuladores comparando con resultados reales y experimentales.*

En general, la verificación enfoca el tema de la consistencia interna de un modelo, mientras que la validación está relacionada con la correspondencia entre el modelo y la realidad. El termino validación se aplica a aquellos procesos que buscan determinar si una simulación es correcta o no respecto al sistema real. De forma más sencilla, la validación trata sobre la cuestión ¿Se está construyendo el sistema correcto?, mientras que la verificación responde a ¿Se está construyendo correctamente el sistema?. La verificación comprueba que la implementación del modelo de simulación (programa) corresponde al modelo, mientras que la validación comprueba que el modelo corresponde con la realidad. La calibración comprueba que los datos generados por la simulación coinciden con los datos reales observados.

Para ser más concisos consideramos las siguientes definiciones:

- *Validación: El proceso de comparar la salida del modelo con el comportamiento del fenómeno. En otras palabras: Comparar la ejecución del modelo con la realidad (física u otra cualquiera). Schlesinger [6] lo define como "sustanciación de que un modelo para computadora, con su dominio de aplicación, posee un rango satisfactorio de precisión consistente con la aplicación para la que se desea el modelo".*

- *Verificación: El proceso de comparar el código del programa con el modelo para garantizar que el código es una implementación correcta del modelo.*

- *Calibración: El proceso de estimar los parámetros de un modelo. La calibración es una forma de prueba y ajuste de los parámetros existentes y generalmente no incluye la introducción de otros nuevos, cambiando la estructura del modelo. En el contexto de optimización, calibración es un procedimiento de optimización implicado en la identificación del sistema durante el diseño experimental.*

7.1.1. Complicaciones de la validación

Está fuera de toda duda que existe un sin número de problemas cuando se intenta llevar a cabo el proceso de validación, pero los más importantes a considerar son los siguientes:

1. *No existe la Validación General. Cada modelo se valida con respecto a sus objetivos. No se puede decir que un modelo válido para un propósito lo tenga que ser necesariamente para otros. Por otro lado, la idea de la simulación es construir modelos sencillos para objetivos concretos, lo que aleja de la idea de validez general. La realidad es el único modelo válido de forma general.*

2. *Puede no existir mundo real con el que comparar. En muchos casos se desarrollan, a partir de sistemas del mundo real, nuevas funcionalidades o servicios. Por lo tanto, en estos casos no se tiene la referencia del mundo real para comparar. Se puede producir el caso de un modelo válido para funcionalidades conocidas del sistema, pero esto no garantiza la validez del mismo para representar cambios sobre el sistema inicial.*

3. *¿Cuál es el mundo real?. Diferentes personas pueden tener apreciaciones diferentes del mundo real, conocidas como visiones del mundo. Esto es un problema cuando se intenta validar modelos. Si la gente tiene diferentes interpretaciones del mundo real, ¿qué interpretación es la adecuada para desarrollar y validar el modelo?. Un modelo para una persona puede no serlo para otra.*

En nuestro caso el modelo que utilizamos es Lenard-Jones con el método numérico de Monte Carlo, el cual ha sido validado y probado ampliamente por la comunidad científica a tal grado que se considera un procedimiento estándar en la simulación. Solo nos resta verificar y validar lo algoritmos que utilizan este modelo, para hacer esto compararemos nuestros resultados con los resultados reales de laboratorio.

Para validar los resultados obtenidos con nuestras simulaciones compararemos con los resultados de laboratorio obtenidos a partir de un sistema de argón líquido con las siguientes características.

- *Densidad:* $1.396\ g/cm^3$

- *Temperatura:* $100.0K$

Es importante conocer los valores de los parámetros para el potencial de interacción interatómica, en este caso los valores para los parámetros del potencial de Lennard-Jones para el argón líquido son:

- $\varepsilon/K_B = 125.7K$

- $\sigma = 0.3345nm$

7.2. Comportamiento de la energía potencial del sistema

Tanto el método de Dinámica Molecular como el Método de Monte Carlo buscan el estado de equilibrio de un sistema de partículas, en el caso de Monte Carlo el equilibrio se logra cuando la energía potencial del sistema oscila alrededor de un valor, las oscilaciones quedan determinadas por las condiciones del sistema, densidad, temperatura, presión, etc.

Al principio de una simulación de Monte Carlo necesitamos conocer las posiciones de cada una de las partículas así como las dimensiones de la caja de simulación, la cantidad de partículas y la temperatura del sistema, en la práctica las posiciones de las partículas se inicializan en un estado cercano a una posición de equilibrio, dicho estado suele ser resultado de experimentos previos. Nosotros hemos inicializado las posiciones de las partículas de forma aleatoria para hacer una validación completa de nuestros algoritmos.

Las características que debemos verificar en el comportamiento de la energía potencial del sistema son:

1. *En la etapa de equilibrio (minimización de energía potencial), la energía potencial debe tener una tendencia decreciente.*

2. *Para valores de la temperatura $T \neq 0$ la energía potencial debe tener oscilaciones.*

3. *La energía potencial se debe estabilizar al rededor de un valor oscilando alrededor del mismo.*

En la imagen [7.1] podemos observar el comportamiento de la energía potencial para $T = 0$, el comportamiento que observamos es que efectivamente la energía potencial tiene una tendencia decreciente y se estabiliza en un valor, lo cual prueba que el funcionamiento esperado para $T = 0$ es el correcto.

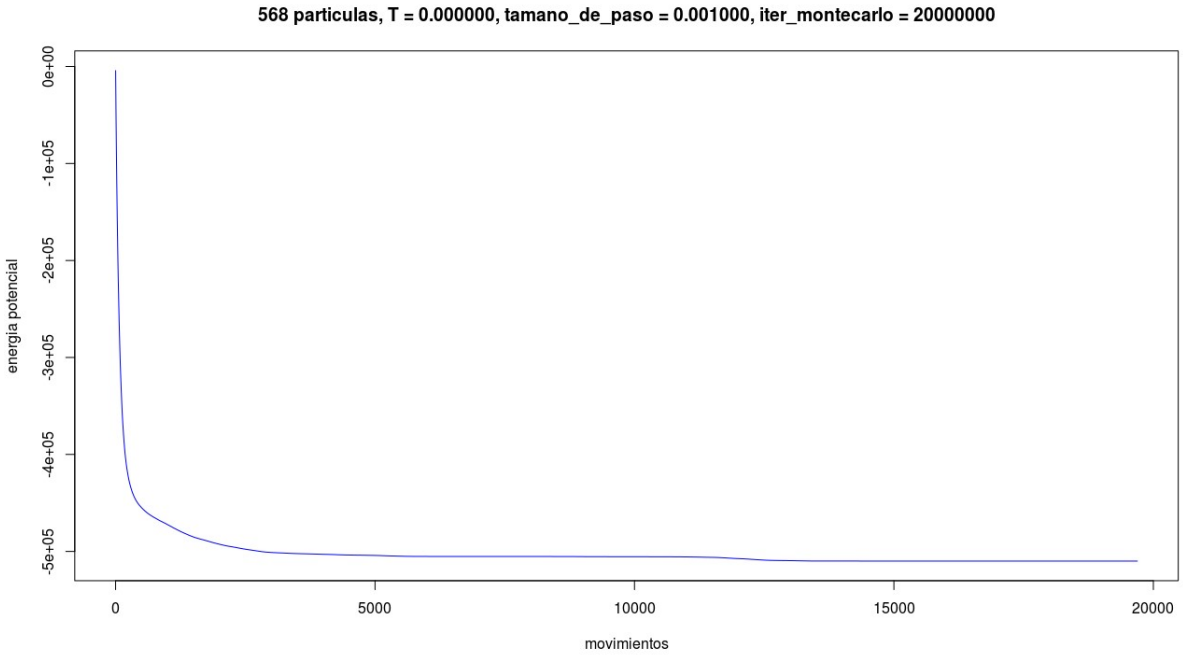

Figura 7.1: Energía potencial para $T = 0$

En la imagen [7.2] podemos observar el comportamiento de la energía potencial para $T = 100$, el comportamiento que observamos es que efectivamente la energía potencial tiene una tendencia decreciente oscilatorio y se estabiliza en un valor oscilando alrededor de él, esto prueba que el funcionamiento esperado para $T \neq 0$ es el correcto.

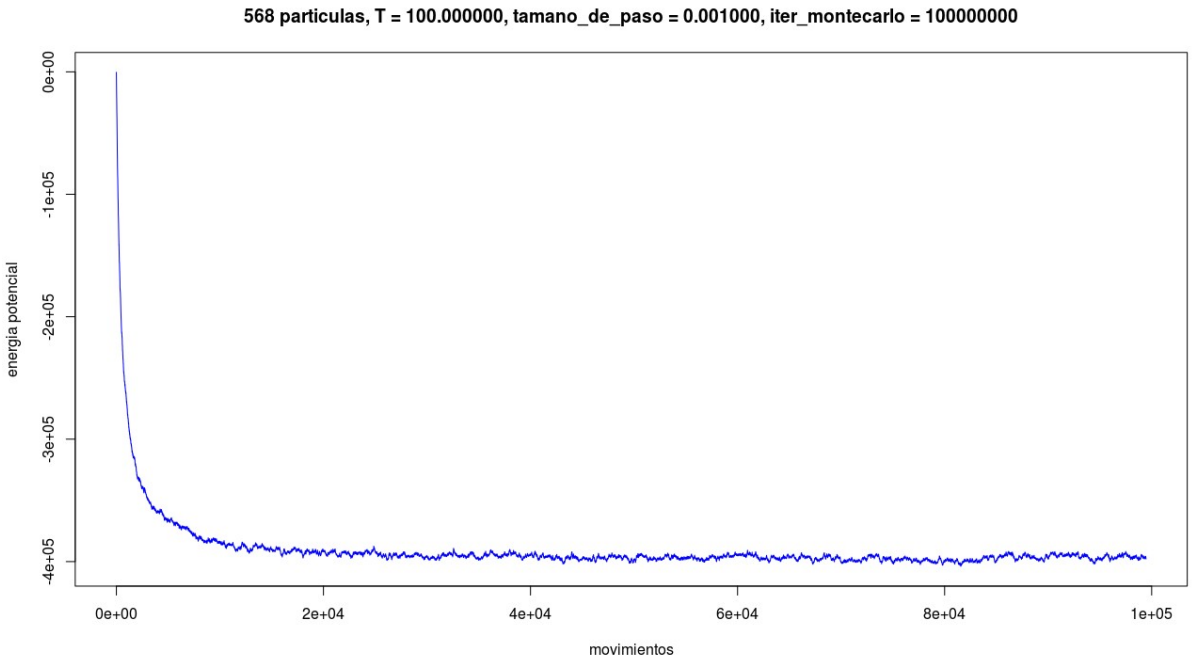

Figura 7.2: Energía potencial para $T = 100$

7.3. Metodologías de programación

Ahora que sabemos cuál debe ser el comportamiento de la energía potencial del sistema, necesitamos validar que las implementaciones de los algoritmos [1] y [2] dan los mismos resultados numéricos, en principio si partimos de la misma configuración inicial y ponemos solo una iteración local en el algoritmo [1] debemos obtener la misma trayectoria independientemente de que algoritmo estemos utilizando. Notar que al poner solo una iteración local en el algoritmo [1], estamos calculando el potencial total del sistema en cada iteración lo cual elimina los errores numéricos cometidos por este algoritmo.

En la imagen [7.3] podemos verificar que los dos algoritmos siguen la misma trayectoria, en la imagen se observa una trayectoria encima de la otra, esto verifica que las diferentes implementaciones arrojan los mismos resultados numéricos.

Para comparar los resultados con las diferentes versiones de cada algoritmo, serial, OpenMP y CUDA, podríamos hacer el mismo experimento que en la comparación de los algoritmos pero observaríamos gráficas encimadas así que optamos por utilizar la misma configuración inicial pero generar distintas trayectorias esperando que las 6 implementación se estabilicen alrededor del mismo valor. Para generar las diferentes trayectorias inicializamos la semilla del generador de números aleatorios con diferentes valores.

En la imagen [7.4] se muestra cómo es que cada implementación se estabiliza alrededor del mismo valor oscilando alrededor de él, la gran diferencia es el tiempo que cada uno de ellos tarda en lograr estabilizarse. Recordar la comparación de tiempos en la sección de resultados.

Ahora tenemos completa certeza en que no importa que algoritmo utilicemos ni que implementación utilicemos para realizar las simulaciones ya que obtenemos los mismos resultados, nosotros optamos como mejor opción utilizar la versión del algoritmo [1] en su versión OpenMP para continuar con la validación de los resultados.

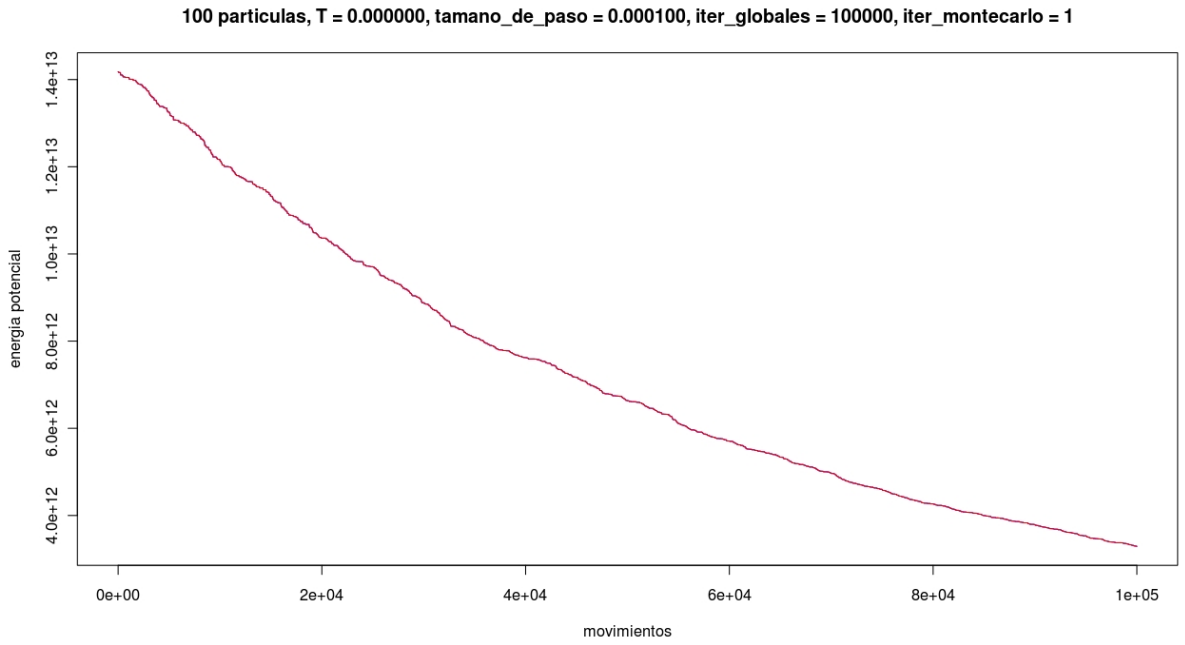

Figura 7.3: Comparación de algoritmos, en azul el algoritmo 1 y en rojo el algoritmo 2

7.4. Sistema de Argón líquido

Al comprobar el buen funcionamiento de los algoritmos y sus diferentes versiones, ahora calcularemos el valor de una variable macroscópica, el índice de solvatación de un sistema de argón liquido con cerca de $10,000,000$ *de átomos de argón. Este índice nos indica en promedio la cantidad de partículas que se encuentran como primeros vecinos al rededor de cada partícula. Para abordar el cálculo del índice de solvatación primero abordaremos la definición de la distribución radial.*

7.4.1. Función de Distribución Radial y el índice de solvatación

La estructura de los fluidos simples se puede caracterizar por un conjunto de funciones de distribución para las posiciones atómicas, de los cuales el más simple es la función distribución radial $g(r)$*. Esta función proporciona la probabilidad de encontrar un par de átomos a una distancia* r*, con respecto a la probabilidad esperada para una distribución completamente al azar a la misma densidad. Se puede definir como:*

$$g(r) = \frac{dN}{dV}\frac{V}{N} \tag{7.1}$$

Donde dN *y* dV *son la cantidad de partículas y el volumen dentro de una cáscara esférica de espesor* $dr = (r + dr) - (r)$*.*

En las imágenes 7.5 y 7.6 podemos observar la función de distribución radial del argón la cual ha sido cotejada con experimentos documentados de rayos X*, obteniendo las mismas características estructurales en la simulación [4].*

El índice de solvatación es el número promedio de primeros vecinos en torno a cualquier átomo de argón dado en el sistema. Con el fin de estimar su valor, hay que calcular la integral de la función de distribución radial:

$$\int_{r=0}^{r=r_{min}} g(r)dr \tag{7.2}$$

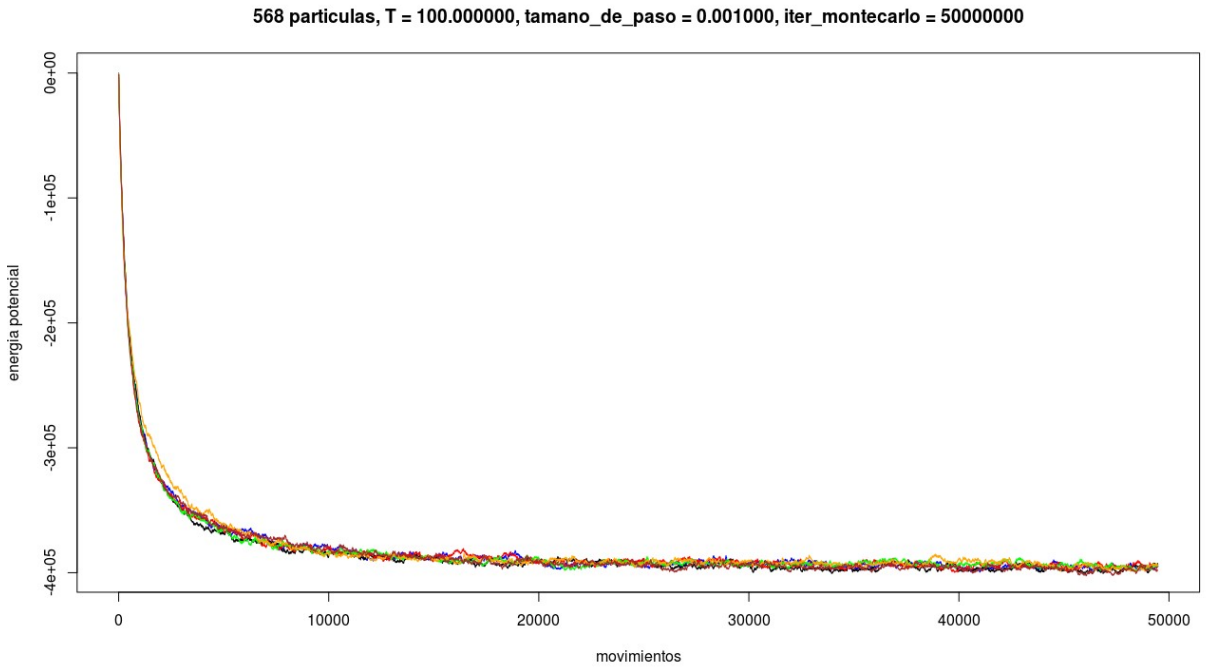

Figura 7.4: Algoritmo 1: (serial-azul), (OpenMP-negro), (CUDA-rojo), Algoritmo 2: (serial-naranja), (OpenMP-verde), (CUDA-cafe)

Donde r_{min}, es el primer mínimo de la función de distribución radial. En este caso, el valor estimado fue de 5.998 *átomos de argón en todos los casos, que se correlaciona bastante bien con las mediciones experimentales, [4].*

7.5. Complejidad Lineal

Una de las contribuciones de este trabajo, es el hecho de lograr una implementación del Método de Monte Carlo con un complejidad lineal, para los expertos quizá sea claro que el sistema de rejilla búffer representar la complejidad lineal, sin embargo para constatar este hecho, hemos realizado el estudio del tiempo que se tarda en ejecutarse una simulación de Monte Carlo en serial y en paralelo.

Hemos considerado seis diferentes valores de N que representa el tamaño de los sistemas con el fin de hacer comparaciones entre los tiempos de ejecución para valores crecientes de N, manteniendo la misma densidad y temperatura. Se utilizó el algoritmo con rejilla y búffer en su versión serial y OpenMP, corriendo ambos en una estación de trabajo con 24 núcleos de procesamiento.

Como se puede ver en la imagen [7.7], los tiempos de ejecución tienen un comportamiento lineal con respecto al número de partículas en el sistema, tanto para la versión serial como para la versión paralela con OpenMP. Este comportamiento se consigue mediante el uso de mallado y el búffer que elimina el cálculo de las distancias entre todos los pares de partículas.

El caso atípico del sistema con 568 *partículas, en el que la versión serial es más eficiente que la aplicación paralela con OpenMP, se debe a que la caja de simulación utilizada para mantener una densidad constante es demasiado pequeña para tomar ventaja de la aplicación con el mallado, y el costo para administrar concurrencias es más caro que para calcular la distancia entre todos los pares de partículas.*

A diferencia de la imagen [6.2] en la que solo se realizan $1,000$ *pasos, en la imagen [7.7] se realizan* 500 *millones de pasos, en el caso de la imagen [6.2] no se alcanza a observar el comportamiento lineal*

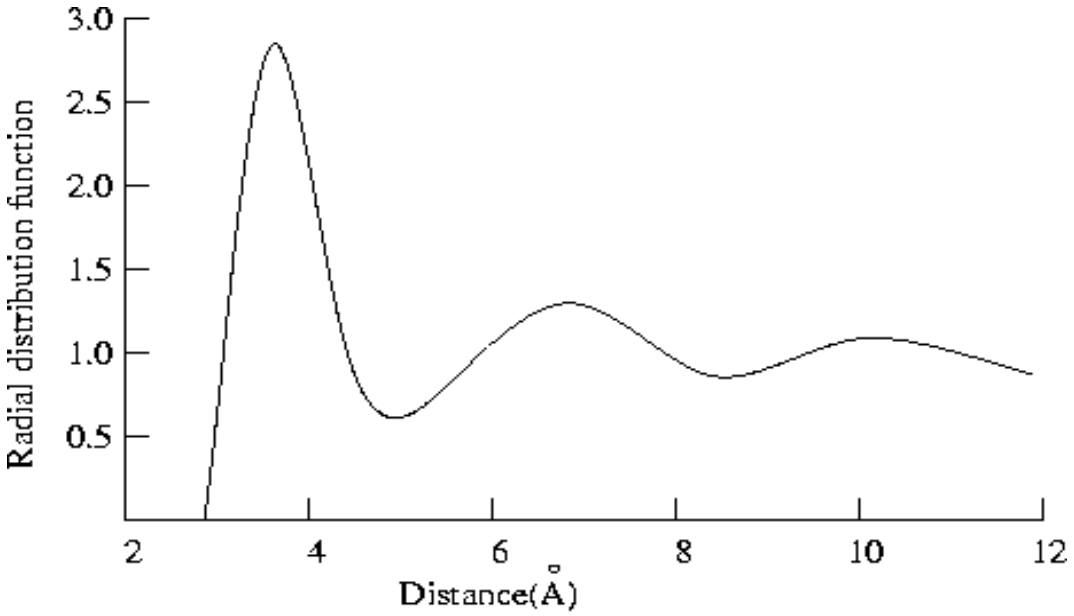

Figura 7.5: Resultados documentados

por que al ser pocos pasos aun pesa mas el tiempo que se tardo calcular la energía potencial total inicial, es por eso que incluso se observa una tendencia cuadrática, pero mientras mas pasos hacemos ya no pesa tanto este primer cálculo como es el caso de la imagen [7.7].

La razón por la que no hay dependencia en el tamaño del sistema (entiéndase tiempo de ejecución respecto al número de partículas), es por que en cada paso de Monte Carlo, al perturbar una partícula, para calcular la diferencia de la energía potencial, solo se analizaran las interacciones entre la partícula perturbada y las partículas en las rejillas vecinas. Como estamos trabajando con la misma densidad, cada rejilla tendrá en promedio el mismo número de partículas, independientemente de cuantas partículas haya en el sistema (ya sean $2,000$ *o* $10,000,000$ *de partículas). Por esta razón hacer un paso de Monte Carlo en un sistema con* $2,000$ *partículas requiere el mismo tiempo y el mismo numero de operaciones que hacer un paso de Monte Carlo en un sistema con* $10,000,000$ *de partículas.*

Por ejemplo si la densidad es tal que en cada rejilla hay en promedio 50 *partículas, entonces para hacer un paso de Monte Carlo se tendrían que revisar* $50 * 27$ *interacciones (en el caso* $3D$ *son* 27 *rejillas las que se revisan), independientemente de cuantas partículas haya en el sistema.*

En resumen, para hacer 1 millón de pasos de Monte Carlo en un sistema de $2,000$ *partículas requiere el mismo tiempo y las mismas operaciones que hacer 1 millón de pasos de Monte Carlo en un sistema con 10 millones de partículas. Sin embargo, mientras mas partículas hay en el sistema serán necesarios mas pasos para estabilizar el sistema.*

De echo la complejidad de las operaciones en cada paso es constante pero lo que hace que el algoritmo sea de complejidad lineal es que para calcular la energía potencial total del sistema al inicio de la simulación se realizan $N * 50 * 27$ *interacciones (considerando una densidad de 50 partículas por rejilla en promedio).*

Con las evidencias mostradas en esta sección hemos realizado la validación de los resultados obtenidos con nuestras propuestas de implementación.

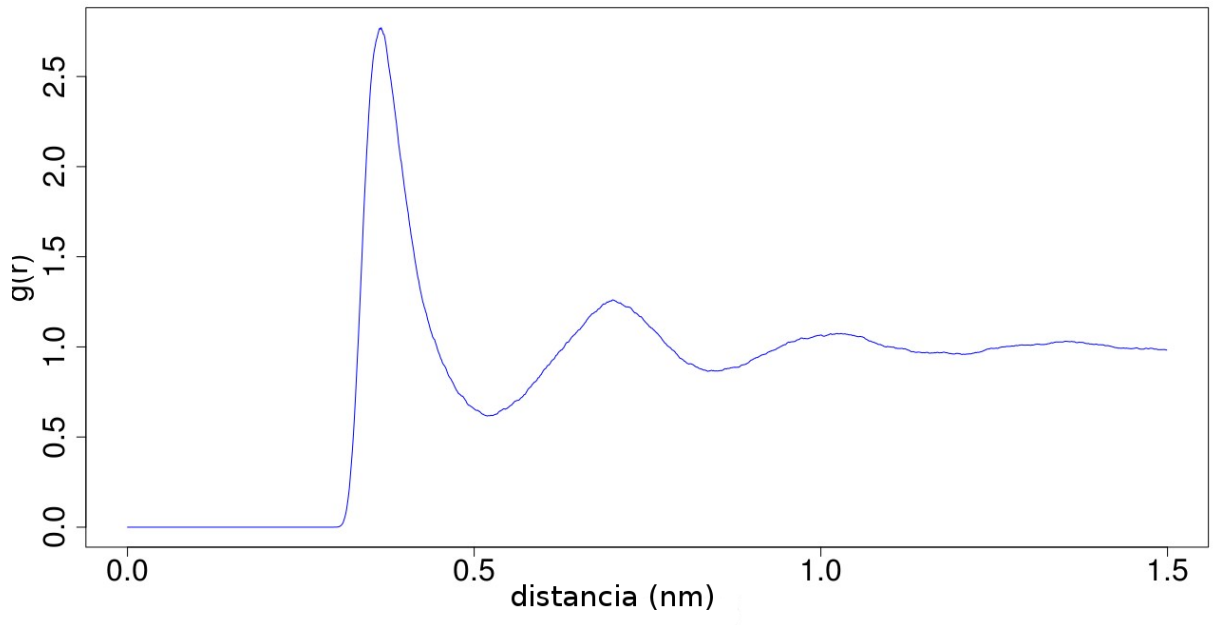

Figura 7.6: Función de distribución radial para el argón liquido

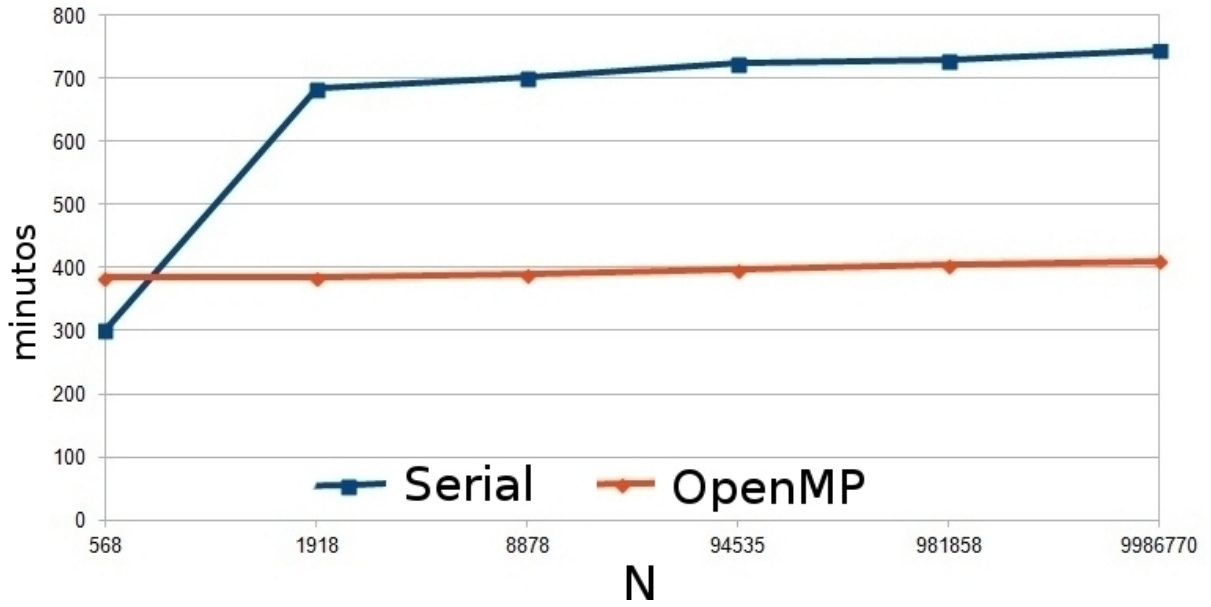

Figura 7.7: Comparación del tiempo de ejecución (min) como una función del tamaño del sistema N (átomos) utilizando la aplicación serial y OpenMP, simulación para 500 millones de pasos en 24 núcleos.

Capítulo 8

Monte Carlo con múltiples perturbaciones

Como se ha mencionado en el trabajo previo, con la implementación del algoritmo [2] no aprovechamos al máximo las capacidades de procesamiento de CUDA ya que los cálculos que realizamos en cada iteración del método de Monte Carlo son muy pocos en relación a la cantidad de cálculos que podemos hacer con CUDA, tomando en cuenta esta información y pensando en aprovechar al máximo el poder de CUDA abordamos el estudio de un sistema de N partículas realizando una modificación al método de Monte Carlo la cual describimos a continuación.

Si tenemos un sistema de N partículas sometido a una temperatura T , el método de Monte Carlo que implementamos es el siguiente:

1. *Calcular la energía potencial inicial* $\mathcal{U}(r)$
2. *Seleccionar una partícula* n_i.
3. *Dar a la partícula* n_i *un desplazamiento aleatorio y calcular la nueva energía* $\mathcal{U}(r')$
4. *Si* $\Delta\mathcal{U} < 0$ *aceptamos la transición y regresamos a 2. Si* $\Delta\mathcal{U} > 0$, *elegimos un número* $\alpha \in [0, 1]$ *al azar.*
   - *Si* $\alpha < exp[-\beta\Delta\mathcal{U}]$, *aceptamos la transición y regresamos a 2.*
   - *Si* $\alpha > exp[-\beta\Delta\mathcal{U}]$, *rechazamos la transición y regresamos a 2.*

Pero el punto débil de la implementación en CUDA es que en el paso 2, cuando calculamos la aportación de la partícula elegida al potencial antes y después de ser perturbada, los cálculos realizados son muy pocos en comparación con el poder de computo de CUDA, es por esto que proponemos mover más de una partícula en cada paso, para que de esta forma aprovechemos más el poder de CUDA.

La idea de perturbar varias partículas es válida para el método de metrópolis, ya que la importancia radica en que una vez que el sistema se encuentra en un estado, la probabilidad de pasar a cualquier otro estado sea la misma, esto se cumple tanto al perturbar una partícula como al perturbar varias partículas y hay que recordar que lo que hacemos al aplicar el método de metrópolis es un muestreo del espacio fase o lo que es lo mismo hacemos un muestreo de la distribución de Boltzmann, lo cual se sigue cumpliendo al perturbar varias partículas porque es una simulación de Monte Carlo-Cadenas de Markov.

Al perturbar varias partículas surgen las siguientes interrogantes:

1. *¿El sistema se estabiliza en menos pasos mientras más partículas perturbamos en cada paso?*
2. *¿Se llega al mismo punto de equilibrio al mover una o varias partículas?*
3. *¿Los observables macroscópicos tienen el mismo comportamiento?*

Para responder a estas preguntas haremos simulaciones para observar el comportamiento del sistema de N partículas, intuitivamente creemos que la respuesta a estas preguntas es afirmativa.

Si perturbamos varias partículas en cada paso del método de metrópolis el método es el siguiente:

1. *Calcular la energía potencial inicial* $\mathcal{U}(r)$

2. *Seleccionar aleatoriamente* m *partículas* $(n_1, n_2, n_3, ..., n_{m-1}, n_m)$.

3. *Dar a las* m *partículas elegidas* $(n_1, n_2, n_3, ..., n_{m-1}, n_m)$ *un desplazamiento aleatorio y calcular la nueva energía* $\mathcal{U}(r')$

4. *Si* $\Delta\mathcal{U} < 0$ *aceptamos la transición y regresamos a 2. Si* $\Delta\mathcal{U} > 0$*, elegimos un número* $\alpha \in [0, 1]$ *al azar.*

   - *Si* $\alpha < exp[-\beta\Delta\mathcal{U}]$*, aceptamos la transición y regresamos a 2.*
   - *Si* $\alpha > exp[-\beta\Delta\mathcal{U}]$*, rechazamos la transición y regresamos a 2.*

Esta modificación al método de metrópolis pueden realizarse directamente en los algoritmos [1] y [2] utilizando m *veces la rutina del cálculo del potencial perturbado para cuando solo perturbamos una partícula considerando en cada paso que los* m *movimientos suceden al mismo tiempo por lo que hay que verificar las interacciones entre las* m *partículas elegidas para ser perturbadas y en términos de programación hay que verificar la concurrencia de los datos.*

El cálculo en la energía potencial total del sistema se calcula de forma independiente para cada una de las partículas perturbadas y posteriormente se sincronizan los resultados sumando las aportaciones de cada una de las partículas perturbadas, esto es lo que nos permite aprovechar mejor CUDA.

Es importante considerar si las partículas elegidas para ser perturbadas se encuentran interactuando antes y después de ser perturbadas para poder hacer los cálculos correctos y no cometer errores numéricos en el cálculo, afortunadamente estas verificaciones las podemos hacer rápidamente si nos apoyamos en el particionamiento que hacemos del dominio ya que basta con verificar para cada par de partículas perturbadas si las rejillas correspondientes son iguales o son vecinas, sin olvidar las condiciones de frontera periódicas.

8.1. 2345 Partículas $T = 0$

Para observar el comportamiento del método de Monte Carlo perturbando varias partículas en cada paso observaremos el caso $T = 0$*. Elegimos perturbar* $2, 5$ *y* 10 *partículas en cada paso.*

En la figura (8.1) podemos observar el comportamiento del potencial respecto al movimiento de varias partículas en cada paso, como podemos ver parece ser que nuestra intuición es correcta ya que el potencial se estabiliza en menos pasos mientras aumentamos el numero de partículas que perturbamos en cada paso.

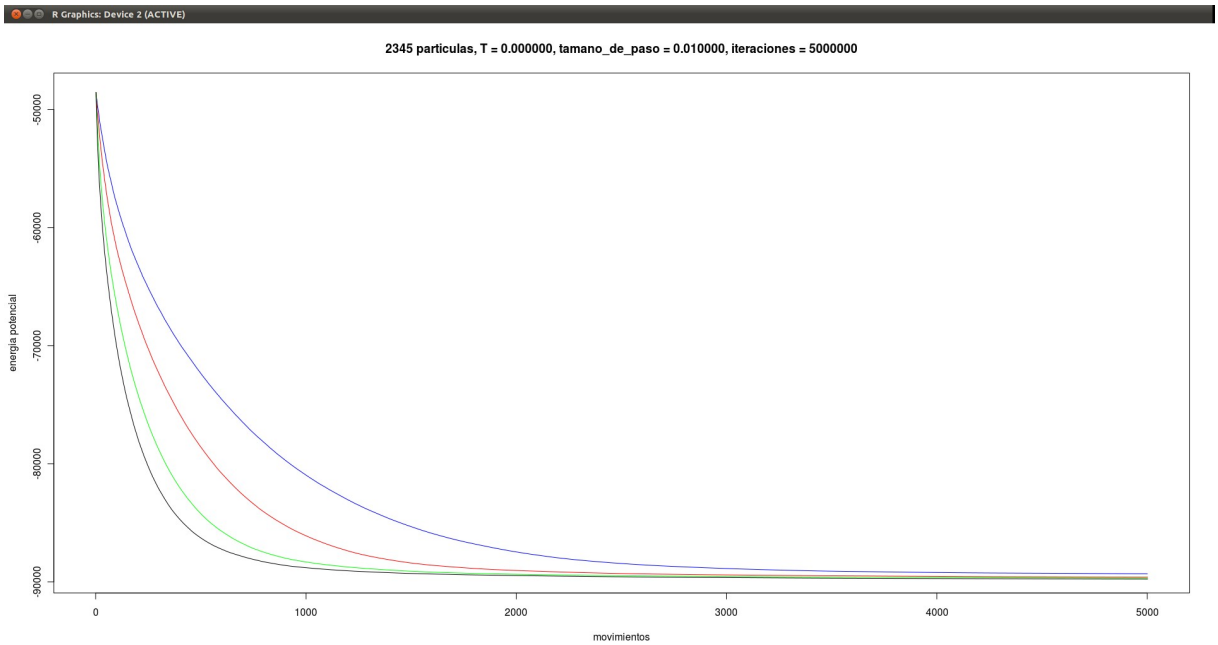

Figura 8.1: Potencial perturbando $1, 2, 5$ y 10 partículas (azul,rojo,verde y negro) respectivamente

Procedemos a perturbar $20, 50, 100$ *y* 500 *partículas, en la figura (8.2) podemos observar el comportamiento del potencial respecto a la perturbación de las partículas. En este caso observamos que al perturbar* $20, 50$ *y* 100 *partículas observamos un comportamiento similar a cuando solo perturbamos* 10 *partículas, sin embargo no sucede lo mismo cuando perturbamos* 500 *partículas, esto nos indica que quizá hay un numero óptimo de partículas que se pueden perturbar para obtener los resultados deseados, esta cantidad óptima claramente debe cambiar respecto al número total de partículas en el sistema, pero será necesario determinar de qué forma cambia este óptimo.*

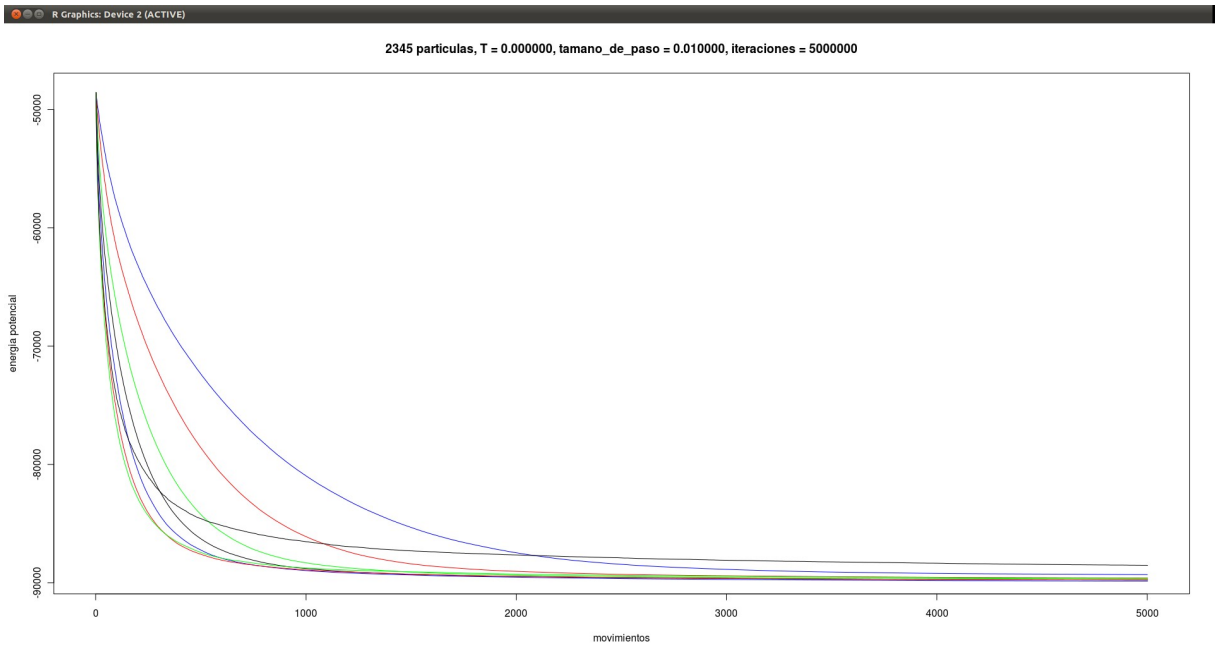

Figura 8.2: Potencial perturbando $1, 2, 5$ y 10 partículas (azul,rojo,verde y negro) respectivamente, y perturbando $20, 50, 100$ y 500 partículas (azul,rojo,verde y negro) respectivamente

El análisis del comportamiento del sistema de partículas al aplicar movimientos múltiples en el método de Monte Carlo nos brinda un nuevo tema de estudio que puede ser explorando realizando las modificaciones adecuadas al algoritmo propuesto en esta tesis. Este nuevo tema de estudio es prometedor ya que como hemos visto podemos aprovechar las nuevas tecnologías de computo paralelo.

Capítulo 9

Conclusiones

En este trabajo se muestra que en una simulación de Monte Carlo se obtiene una ventaja considerable haciendo un particionamiento del dominio con una cuadrícula regular y utilizando un sistema de buffer para almacenar los datos de cada una de las partículas del sistema, ordenando en memoria adecuadamente por cada rejilla del mallado. El resultado es que podemos hacer simulaciones en tiempos que reflejan una complejidad lineal, en contraste con las implementaciones tradicionales en donde se realiza el cálculo de la distancia entre todos los pares de átomos en el sistema, que tiene una complejidad cuadrática .

Además, se muestra la ventaja del uso de la computación en paralelo para realizar el cálculo de la energía potencial total de un sistema de partículas y el cálculo de la variación de energía potencial cuando perturbamos la posición de una partícula durante la simulación de Monte Carlo, produciendo gran aceleración y una alta escalabilidad.

Los resultados obtenidos con las simulaciones han sido cotejados con experimentos realizados en laboratorio, y hemos obtenido resultados numéricos con un alto grado de correlación respecto a los datos de laboratorio, lo que significa que las simulaciones arrojan resultados correctos.

Apéndice A

Computo en Paralelo

En los últimos años el desarrollo de la computación ha ido de la mano de los avances en la ciencia y la tecnología, este desarrollo ha permitido realizar tareas que hace apenas unas décadas eran impensables, como por ejemplo en las áreas de medicina, biología y videojuegos. Particularmente se ha desarrollado el paradigma de programación en paralelo cuya idea básica es poder dividir un problema complejo en pequeñas tareas que se puedan resolver de manera independiente y así poder tratarlas por separado, ya sea en un clúster de computadoras con un esquema de memoria distribuida, o en una computadora con múltiples núcleos de procesamiento con un esquema de memoria compartida .

En la actualidad existen distintos lenguajes de programación y distintas metodologías para realizar cómputo en paralelo, una de ellas es la tecnología CUDA (Compute Unified Device Architecture) que se refiere a un conjunto de herramientas y a un compilador desarrollado por NVIDIA. CUDA hace uso de las tarjetas gráficas que en un principio solo se enfocaban a realizar múltiples operaciones para el despliegue de gráficos en los monitores de las computadoras de forma rápida, es decir que las tarjetas gráficas están hechas para un propósito bien definido a diferencia de los CPU convencionales que son de propósito general.

A.1. OpenMP

A.1.1. ¿Que es OpenMP?

OpenMP es una API que nos permite añadir concurrencia a las aplicaciones mediante paralelismo con memoria compartida [5]. Se basa en la creación de threads de ejecución paralelos compartiendo las variables del proceso padre que los crea. OpenMP comprende de tres componentes complementarios:

- *Un conjunto de directivas de compilador usado por el programador para comunicarse con el compilador en paralelismo.*
- *Una librería de funciones en tiempo de ejecución que habilita la colocación e interroga sobre los parámetros paralelos que se van a usar, tal como número de los threads que van a participar y el número de cada hilo.*
- *Un número limitado de las variables de entorno que pueden ser usadas para definir en tiempo de ejecución parámetros del sistema en paralelo tales como el número de threads.*

OpenMP ha sido desarrollado específicamente para procesamiento de memoria compartida en paralelo. De hecho se ha vuelto el estándar de paralelización en este tipo de arquitecturas. En esta sección se describe el uso de OpenMP para explotar el paralelismo en arquitecturas de memoria compartida.

Un proceso puede consistir de múltiples threads, cada uno con su propio flujo de control pero compartiendo el mismo espacio de direcciones, un programa comienza con un solo hilo Master que puede iniciar otros threads constituyendo un equipo de hilos en una sección paralela.

Al ser OpenMP un modelo de memoria compartida, los threads se comunican utilizando variables compartidas. El uso inadecuado de variables compartidas origina secciones críticas, es por esto que para controlar las variables compartidas se requiere el uso de sincronización para protegerse de los conflictos de datos. La sincronización es costosa, entonces es útil modificar cómo se almacenan los datos para minimizar la necesidad de sincronización.

En OpenMP, estas son las dos principales formas de aprovechamiento para asignación de trabajo en threads:

- *Niveles de Loop Paralelos*
- *Regiones Paralelas*

En el primer caso, loops individuales son paralelizados con cada uno de los hilos empezando una única asignación para el índice del loop. A esto se le llama paralelismo de fina granulación. En el aprovechamiento de regiones paralelas, cualquier sección del código puede ser paralelizado no precisamente por loops. Esto es algunas veces llamado paralelismo de gruesa granulación. El trabajo es distribuido explícitamente en cada uno de los hilos usando el único identificador para cada hilo.

OpenMP entra en un programa nuevo o existente a través de directivas, con funciones propias (que pueden ser condicionalmente compiladas) y variables de entorno, que pueden modificar el comportamiento en tiempo de ejecución.

En el caso de la paralelización por loops, la paralelización con OpenMP se traduce a paralelizar un for en el que cada thread trabaja sobre un rango determinado de los valores de la variable de control del for, podría decirse que solo podemos paralelizar los for.

Operaciones matemáticas fácilmente paralelizables

Decir que una operación es paralelizable significa que pueden separarse en varias sub-operaciones que puede realizarse de forma Independiente. Por ejemplo, en la suma de dos vectores x, y para producir otro vector z:

$$z = x + y \tag{A.1}$$

$$z_i = x_i + y_i \ , \ i = 1, ..., N \tag{A.2}$$

En este caso las N sumas pueden realizarse simultáneamente, asignando una a cada procesador. Lo que hay que resaltar es que no hay dependencia entre los diferentes pares de datos, tenemos entonces el paralelismo más eficiente.

No tan fáciles de paralelizar

Por ejemplo el producto punto de dos vectores x , y

$$a = \sum_{i=1}^{N} x_i y_i \tag{A.3}$$

Donde a es un escalar, una primera aproximación sería verlo como una secuencia de sumas de productos que requieren irse acumulando, al verlo así no es una operación paralelizable.

El problema en este tipo de operaciones es que el resultado parcial de cada uno de los hilos se tiene que almacenar en una sola variable compartida lo que ocasiona problemas de sincronización. Para este tipo de problemas OpenMP cuenta con directivas que manejan la concurrencia de datos, sin embargo la sincronización es costosa por lo que se evita utilizar este tipo de herramientas y se opta por utilizar técnicas de reducción que consisten en hacerla suma por bloques.

A.2. Ejemplos

Para tener una mejor idea de cómo se programa en paralelo utilizando OpenMP, en esta sección mostraremos algunos ejemplos.

Comencemos con la paralelización de la suma de dos vectores, usualmente una función en C/C++ se escribiría así:

```
void Suma(double* a, double* b, double* c, int size)
  {
    for (int i = 0; i < size; ++i)
    {
     c[i] = a[i] + b[i];
    }
  }
```

La paralelización con OpenMP seria:

```
void Suma(double* a, double* b, double* c, int size)
{
  #pragma omp parallel for
  for (int i = 0; i < size; ++i)
  {
   c[i] = a[i] + b[i];
  }
}
```

Para compilar con GCC es necesario agregar:

```
gcc -o programa -fopenmp main.c
```

¿Cómo funciona?

Supongamos que size $= 30$*, si la computadora tiene* 3 *procesadores, el código se ejecutaría como:*

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i)
{
  c[i] = a[i] + b[i];
}

for (int i = 10; i < 20; ++i)
{
  c[i] = a[i] + b[i];
}

for (int i = 20; i < 30; ++i)
{
  c[i] = a[i] + b[i];
}
```

¿Cómo se define el número de procesadores/threads a utilizar?

- *Por default el número de threads es igual al número de cores en la computadora.*
- *Se puede establecer por medio de variables de ambiente.*

Para poder definir el numero de threads es necesario incluir el header .ºmp.h", el cual incluye varias funciones para el control de threads.

```
#include <omp.h>
int threads = 3;

void Suma(double* a, double* b, double* c, int size)
{
  omp_set_num_threads(threads);

  #pragma omp parallel for
  for (int i = 0; i < size; ++i)
  {
    c[i] = a[i] + b[i];
  }
}
```

omp_set_num_threads() permite establecer el número de threads a utilizar en las siguientes partes del programa.

A.3. Paralelizar bloques de código

Además de paralelizar ciclos, es posible paralelizar bloques, para poder hacer esto necesitamos identificar el hilo de ejecución que está realizando la parte del código paralelizada, por ejemplo:

```
#pragma omp parallel
{
  // ... c\'odigo a ejecutar en paralelo...
  int thread = omp_get_thread_num();
}
```

omp_get_thread_num regresa el número de thread actual.

Se puede hacer que varias secciones de código se ejecuten de forma simultánea:

```
int threads = 2;
#pragma omp parallel sections num_threads(threads)
{
  #pragma omp section
  {
    // ... algo de c\'odigo
  }

  #pragma omp section
  {
  // ... otro c\'odigo
  }
}
```

Con num_threads() podemos especificar cuantos threads/cores se utilizarán. En este caso uno por cada sección.

A grandes rasgos estos son los principios básicos de la programación en paralelo utilizando OpenMP, para mayores especificaciones el lector puede consultar la página oficial: http://openmp.org/wp/.

A.4. CUDA

A.4.1. ¿Que es CUDA?

Hasta hace poco lo habitual para la computación en paralelo era unir varios procesadores en una misma computadora o montar clústeres, pero el procesador no es el único elemento de la computadora que realiza cálculos. Las tarjetas gráficas realizan gran cantidad de cálculos al trabajar con las imágenes. En particular las tarjetas gráficas más recientes de NVIDIA nos ofrecen la tecnología CUDA (Compute Unified Device Architecture), que nos permite realizar computo en paralelo utilizando el poder de procesamiento de estas tarjetas gráficas. Una de las grandes ventajas de CUDA es que permite a los programadores usar una variación del lenguaje de programación C para codificar algoritmos que exploten el paralelismo masivo ofrecido por las GPUs.

CUDA intenta explotar las ventajas de las GPU frente a las CPU de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneamente. Por ello, si una aplicación está diseñada utilizando gran cantidad de hilos que realizan tareas independientes (que es lo que hacen las GPU al procesar gráficos, su tarea natural), una GPU podrá ofrecer un gran rendimiento en campos que podrían ir desde la biología computacional a la criptografía por ejemplo.

Una de las grandes diferencias entre los CPU de propósito general y las tarjetas gráficas es que a diferencia de los primeros, las tarjetas gráficas cuentan con múltiples unidades de procesamiento, en algunos casos hasta 500 procesadores integrados dentro de una misma tarjeta gráfica, esto es lo que nos permite desarrollar computo paralelo en estas tarjetas, para poder hacerlo se necesita un lenguaje de programación y un compilador es por eso que envidia se apoyo del lenguaje de programación C (y posteriormente C++) al que agrego algunas funciones para poder trabajar en las tarjetas gráficas, para compilar se utiliza el compilador nvcc que está basado en los compiladores gcc y g++ respectivamente para cuando utilizamos C y C++. También es posible utilizar Fortran aunque hay mayor desarrollo y soporte para C y C ++.

A.4.2. Arquitectura CUDA

Para estar en contexto con la literatura oficial de CUDA utilizaremos la palabra device para referirnos a la tarjeta gráfica y host para referirnos al sistema en el que se encuentra la tarjeta, es decir el CPU principal de la computadora y la memoria RAM del sistema. Una de las ideas básicas de CUDA es ejecutar funciones, que llamaremos kernel, en el device realizando la llamada a la función desde el programa principal en el host, estas funciones deben realizar la misma operación para distintos datos de entrada, en teoría, cada operación se realizara en cada uno de los procesadores de la tarjeta de forma independiente, para lograr esto necesitamos saber que datos operara cada procesador y para ello utilizaremos threads, cada thread estará asociado a un procesador y cuando llamemos a la función en el device debemos indicar la cantidad de threads que se ejecutaran, esto es equivalente a decir que ejecutaremos una función en el device la misma cantidad de veces que se la cantidad de threads que indiquemos, con el entendido de que cada thread manejara distintos datos.

Para saber cómo se manejan los threads es necesario conocer un poco sobre la arquitectura del device, y como se organizan los threads en el device, esto será mas claro con un ejemplo, el ejemplo que abordamos es la de sumar un entero y a un arreglo de enteros x de dimensión n. El código para hacer esta tarea en C es el siguiente:

```
//----------------------------------------
#include <stdio.h>
#include <stdlib.h>

//----------------------------------------
void sumar_y(int * x, int y, int n){
   for(int i = 0 ; i < n ; i++)
```

```
        x[i] += y;
}

//----------------------------------------
int main(void){
   int n = 100000;
   int y = 5;
   int * x = (int *)malloc(n*sizeof(int));

   for(int i = 0 ; i < n ; i++)
      x[i] = 10;

   sumar_y(x,y,n);
   free(x);
   return 0;
}
//----------------------------------------
```

Es importante mencionar que la memoria solicitada con la función malloc es memoria del host, es decir, localidades de memoria en la memoria RAM del sistema, además, las instrucciones en los for se llevan a cabo por el CPU del sistema en forma secuencial.

Para hacer la misma tarea en CUDA primero explicaremos como están organizados los thread, podemos decir que la unidad básica para manejar información en CUDA son los thread, cada thread pertenece a un conjunto de thread llamado bloque, los bloques a su vez pertenecen a un conjunto de bloques llamada grid, es importante saber que cuando realicemos la llamada a una función kernel que se ejecutara en el device necesitamos indicar las dimensiones del grid y las dimensiones de los bloques, todos los bloques tienen las mismas dimensiones y pueden ser 1D, 2D, o 3D, el grid puede tener 1 o 2 dimensiones dependiendo de la naturaleza del kernel. Para identificar los thread dentro de un bloque cada thread tiene asociado 3 variables para identificarlo:

$threadIdx.x$, $threadIdx.y$ *y* $threadIdx.z$

Estas variables no son más que los índices del thread respecto a cada dimensión del bloque al que pertenece. De la misma forma cada bloque tiene asociado 2 variables para identificarlo:

$blockIdx.x$ *y* $blockIdx.y$

Que de la misma manera que sucede con los thread, estas variables son los índices del bloque respecto a cada dimensión del grid. Para saber las dimensiones del bloque contamos con las variables:

$blockDim.x$, $blockDim.y$ *y* $blockDim.z$

Y para saber las dimensiones del grid tenemos las variables:

$gridDim.x$ *y* $gridDim.y$

Ahora que conocemos como están organizados los thread debemos saber que los datos que operara la función kernel deben estar almacenados en la memoria del device, para hacer esto necesitamos reservar memoria en el device, la función que utilizaremos es:

$cudaMalloc(void**devPtr, size_t size)$;

Esta función se utiliza de forma similar que la función malloc de C para solicitar memoria del host solo que cudaMalloc reserva memoria en el device, también existe el equivalente a free, la función para liberar memoria del device es: cudaFree. Una vez que tenemos reservada la memoria en el device necesitamos copiar los datos del host al device para poder realizar los cálculos necesarios, para copiar los datos

utilizamos la función:

$$cudaMemcpy(void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)$$

Esta función se utiliza tanto para copiar del host al device como del device al host una vez que los cálculos se realizaron, para indicar la dirección en la que se realiza la copia se utiliza el ultimo parámetro.

Ahora que conocemos las funciones más importantes que se utilizan en CUDA vamos a desarrollar el código para realizar la misma tarea de sumar un entero y a un arreglo de enteros x de dimensión n, y realizar la comparación con el código hecho en C. A continuación se muestra el código en CUDA:

```
//----------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

//utilizamos bloques de una dimensi\'on
const int hilosPorBloque = 512;

//----------------------------------------
__global__ void kernel(int * d_x, int y , int n){
   int tid = blockDim.x * blockIdx.x + threadIdx.x;

   if(tid < n){
      d_x[i] += y;
   }
}

//----------------------------------------
int main(void){
   int n = 100000;
   int y = 5;
   //apuntador a memoria en el host
   int * h_x;

   //memoria en el host
   h_x = (int *)malloc(n*sizeof(int));

   //apuntador a memoria del device
   int * d_x;

   //memoria en el device
   cudaMalloc((void **) &d_x ,n*sizeof(int));

   //llenamos los datos en el device
   for(int i = 0; i < n ; i++)
      h_x[i] = 10;

   //copiamos del host al device
   cudaMemcpy(d_x, h_x, n * sizeof(int), cudaMemcpyHostToDevice);

   //utilizamos un grid de una dimensi\'on
   const int bloquesPorGrid = (n + hilosPorBloque -1)/hilosPorBloque;

   //lanzamos el kernel
   kernel<<<bloquesPorGrid,hilosPorBloque>>>(d_x,y,n);
```

```
    //copiamos del device al host
    cudaMemcpy(h_x, d_x, n * sizeof(int), cudaMemcpyDeviceToHost);

    //liberamos memoria en el host
    free(h_x);

    //liberamos memoria en el device

    cudaFree(d_x);
    return 0;
}
```

Con los comentarios en el código es claro lo que hace cada instrucción pero daremos una explicación detallada de los puntos importantes. Para indicar que la función kernel se ejecutara en el device existe el modificador de función $__global__$*, estas funciones deben ser de tipo void y la sintaxis es como se realizo en el código anterior:*

$$__global__ voidkernel(int * d_x, inty, intn);$$

El nombre de la función no tiene que ser necesariamente kernel lo importante es el modificador $__global__$ *que nos indicara que la función se ejecutara en el device pero se llamara desde el host, existen otros modificadores de función,* $__host__$ *son funciones que se ejecutan en host y solo pueden ser llamadas desde el host, las funciones* $__device__$ *son funciones que se ejecutan en el device pero solo pueden ser llamadas desde funciones tipo* $__global__$ *o funciones tipo* $__device__$*, estas funciones no tienen que ser necesariamente void pero solo manipularan información que se encuentre en el device.*

Como se menciono anteriormente para ejecutar una función $__global__$ *en el device es necesario indicar las dimensiones del grid a utilizar y las dimensiones de los bloques, es este caso utilizamos un grid y bloques de una dimensión. Es necesario indicar que el número de hilos por bloque tiene una cota superior, en algunas tarjetas el máximo es* 512 *y en otras más recientes es* 1024*. La sintaxis para ejecutar una función* $__global__$ *es como se utilizo en el código:*

$$kernel <<< bloquesPorGrid, hilosPorBloque >>> (d\_x, y, n);$$

A diferencia de las llamadas a funciones en C debemos agregar $<<< bloquesPorGrid, hilosPorBloque >>>$ *para indicar las dimensiones del grid y de los bloques.*

Los datos que se envían a las funciones $__global__$ *por lo general son apuntadores a memoria en el device, para tener un mejor control de estos apuntadores se utilizan los indicadores* $d\_$ *y* $h\_$ *para device y host respectivamente, estos indicadores forman parte del nombre de las variables por lo que se pueden utilizar los indicadores que el desarrollador desee pero deben ser claros para saber a qué tipo de memoria apuntan estas variables.*

En el cuerpo de la función $__global__$ *kernel se encuentra la siguiente línea de código:*

$$inttid = blockDim.x * blockIdx.x + threadIdx.x;$$

Es fácil ver que esta línea lo que hace es dar un índice global al thread para identificar los datos con los que trabajara cada thread, pero necesitamos asegurarnos que en el caso en el que se hayan lanzado mas thread de los necesarios (por la cantidad de hilosPorBloque) hay que verificar que $tid < n$*, así nos aseguramos de no acceder a memoria que no tenemos asignada.*

A.5. Computo Paralelo Para el Método de Monte Carlo

Existen diversas implementaciones para realizar simulaciones de sistemas de partículas aplicando el método de Monte Carlo, sin embargo, la mayoría de ellas solo se pueden utilizar de forma serial, aun cuando son ejecutadas en computadoras con más de un procesador.

La limitación mas importante de ejecutar los programas de forma serial es que el número de partículas con las que se puede realizar la simulación no excede los cientos de miles de partículas, esta limitación se debe a que el cálculo de la energía potencial del sistema es de orden N^2, por lo que, mientras más aumentamos el número de partículas, el tiempo necesario para realizar la simulación crece de forma cuadrática. El tiempo necesario para realizar una simulación de Monte Carlo puede reducirse de forma considerable si aplicamos adecuadamente el computo en paralelo.

Si deseamos realizar una implementación en paralelo del método de Monte Carlo es necesario identificar los procesos y cálculos independientes que se pueden realizar de forma simultánea para aprovechar las ventajas del computo en paralelo. Resulta que la parte más costosa en una simulación de Monte Carlo es el cálculo de la energía potencial del sistema y el aporte de cada una de las partículas al potencial total se puede calcular de forma independiente, esto nos permite aplicar el cómputo en paralelo de forma satisfactoria y eficiente.

El objetivo de aplicar el computo en paralelo es aumentar el número de partículas en la simulación y realizar las simulaciones en un tiempo considerable, de esta forma las simulaciones serán mas consistentes con la realidad.

Apéndice B

Unidades de las Variables

Cuando realizamos una simulación computacional de un fenómeno físico es de vital importancia considerar las unidades adecuadas para las variables involucradas, ya que de no hacerlo corremos el riego de obtener resultado erróneos, pues no es lo mismo que trabajemos en un sistema con carácter nanométrico a que trabajemos con un sistema de carácter galáctico.

Si no se consideran las unidades adecuadas, podemos perder mucho tiempo pensando que nuestras implementaciones son inadecuadas, un error muy común es considerar que las constantes físicas pueden tener valor 1*, aunque matemáticamente hacer esta consideración puede ser consistente con los modelos matemáticos, la realidad es que para cada sistema las constantes físicas juegan papeles importantes y en cada caso se pueden tener diferentes valores dependiendo de las unidades en que se esté trabajando.*

En el caso de las simulaciones numéricas aquí presentadas, estamos trabajando en un sistema de partículas a nivel atómico por lo que tenemos que considerar las unidades de distancia adecuadas.

Para lograr la consistencia con los resultados de laboratorio, en esta tesis hemos utilizado el siguiente sistema de unidades:

- *Temperatura* K *kelvins.*
- *Distancia:* nm *nanómetros.*
- *Densidad:* g/cm^3 *gramos por centímetro cubico.*
- *Energía:* J *Joules.*

Con este sistema de unidades, los valores para las constantes involucradas en el Método de Monte Carlo son:

- $\beta = \frac{1}{T}$ *con unidades* $\frac{1}{K}$

Es importante notar que la constante de Boltzmann se incluye en el valor para la variable ε *del potencial de Lennard-Jones, en el caso del argón líquido el valor es:*

$$\frac{\varepsilon}{K_B} = 125.7\ K \tag{B.1}$$

Con estas unidades y valores para las constantes estamos en condiciones de realizar simulaciones de Monte Carlo a nivel atómico.

Bibliografía

[1] Loup Verlet, Computer Experiments on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review 1967.*

[2] Allen & Tildesley, Computer simulation of liquids. *1991.*

[3] Simon Green, Particle Simulation using CUDA. *2010.*

[4] A. Rahman, Correlation in the motion of atoms in liquid argon. *Physical Review 136(2A):405(1964).*

[5] M. Vargas Felix, Curso de computo en paralelo. *CIMAT (2011).*

[6] Schlesinger, Verificaction, Validation and Testing.. *1979.*

[7] Guillermo Amaro, Carrillo Tripp, Salvador Botello. Linear Complexity and High Scalability of a Parallel Monte Carlo Simulation Method. *3PGCIC 2013 : International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Octubre 2013.*