UNIVERSIDAD AUTÓNOMA DE SAN LUIS POTOSÍ

FACULTAD DE CIENCIAS

Simulación de dinámica browniana de un electrolito confinado alrededor de un macroion con carga discreta

TESIS

Para obtener el grado de:

Licenciado en Física

Presenta:

Eduardo Rodolfo Rodríguez Gallegos

Asesor:

Dr. Guillermo Iván Guerrero García

San Luis Potosí, S.L.P. Junio 2023

Agradecimientos

A mis padres, quienes han sido mi mayor fuente de apoyo y motivación a lo largo de toda mi vida. Gracias por su amor incondicional, su constante aliento y por creer en mí en cada paso del camino. Su apoyo inquebrantable ha sido el motor que me impulsa a seguir buscando mis sueños.

A mis amigos, quienes han sido un apoyo constante en mi vida. Gracias por compartir risas, experiencias y momentos inolvidables. En particular, a mi amigo Jaime Stack, por ser ese empuje constante que necesitaba para lograr mis metas.

Al Dr. Guillermo, mi asesor de tesis, por brindarme la oportunidad de trabajar con él y demostrar confianza en mis aptitudes. Aprecio sinceramente su constante disposición y valiosa retroalimentación.

Se agradece el apoyo financiero del Fondo Sectorial de Investigación para la Educación SEP-CONACYT proveniente del proyecto CB 2016-286105, y al CNS-IPICYT por los recursos computacionales otorgados mediante el proyecto TKII-IVGU001.

Resumen

En este trabajo se presenta el desarrollo de dos programas de simulación de dinámica browniana. El primero consiste en resolver el problema de Thomson, para cualquier número de partículas. Se empleó el método de descenso de gradiente para conseguir una configuración con menor energía potencial.

El segundo programa desarrollado es la simulación de la dinámica browniana de un electrolito confinado alrededor de un macroion con carga discreta, siendo la configuración de este último un resultado de la primera simulación. Para el confinamiento del electrolito se utilizó un potencial de núcleo repulsivo. Se analizaron dos casos en concreto, cuando el macroion de carga discreta tiene un solo tipo de carga repartida de manera uniforme en los sitios de carga, y cuando el macroion de carga discreta esta compuesto por cargas negativas y positivas, variando el número de sitios de carga que componen al macroion en ambos casos.

Los programas fueron optimizados y paralelizados con directivas de OpenMP y OpenACC, permitiendo un análisis detallado de la distribución de iones y su comportamiento en diferentes escenarios.

En el capítulo cuatro se presentan los algoritmos de las implementaciones desarrolladas, para las cuales se realizaron pruebas de eficiencia y eficacia. Estos algoritmos brindan la oportunidad de analizar otros sistemas complejos que requieren un cálculo y análisis intensivo en un tiempo conveniente.

Índice

CAPÍTULO 1

Introducción

1.1 Motivación

La simulación de sistemas físicos mediante métodos computacionales ha sido una herramienta invaluable para entender el comportamiento de sistemas complejos. La manera de actuar los electrolitos en solución es un área de interés en la investigación, en particular, la forma de comportarse de los electrolitos alrededor de partículas cargadas es esencial para comprender aplicaciones prácticas en la biología molecular [1], en la nanotecnología [2] y en la química de coloides [3]. El estudio de estos sistemas mediante simulaciones computacionales es una herramienta útil para entender su modo de actuar y predecir sus propiedades.

Estas simulaciones pueden demandar una gran capacidad de procesamiento computacional y requerir largos tiempos de cálculo, debido a la gran cantidad de partículas y la interacción no lineal entre ellas. Por lo tanto, la paralelización de estas simulaciones mediante múltiples procesadores y/o tarjetas gráficas puede mejorar significativamente el tiempo de simulación y permitir la realización de análisis más detallados de estos sistemas.

El desarrollo de algoritmos de paralelización para simulaciones en física es un tema de investigación muy activo en la actualidad. La implementación de estos algoritmos puede resultar en una aceleración significativa en el tiempo de simulación, lo que permite la ejecución de simulaciones más grandes y complejas, y por lo tanto, una mejor comprensión del comportamiento de los sistemas físicos. En este sentido, la implementación de la paralelización en simulaciones de electrolitos alrededor de partículas grandes con carga discreta puede ser de gran importancia para el avance en la comprensión de estos sistemas.

1.2 Problemática

El estudio de los sistemas físicos en presencia de iones es de gran interés en la investigación, ya que estos sistemas son relevantes en muchas aplicaciones prácticas. Entre estos sistemas, el de un electrolito alrededor de una partícula grande con carga discreta es de particular interés debido a su complejidad y al papel crucial que juega en muchos procesos físicos y químicos en medios acuosos.

Sin embargo, la simulación de este tipo de sistema es un desafío debido a que las interacciones se vuelvan más complejas a medida que el número de partículas aumenta, además de la falta de herramientas disponibles para su análisis. Como resultado, este sistema ha sido menos estudiado que otros sistemas más simples, limitando la capacidad de los investigadores para comprender completamente la manera de actuar de este.

1.3 Objetivos

- Desarrollar un programa que simule la dinámica browniana de un electrolito confinado alrededor de una partícula grande con carga discreta.
- Analizar el comportamiento del electrolito en torno a una partícula grande con carga discreta, en particular, cómo el número de cargas que componen la partícula afecta la distribución de iones del electrolito circundante.
- Mejorar la eficiencia y la escalabilidad del programa mediante su paralelización con varios procesadores y tarjetas gráficas, permitiendo la ejecución de simulaciones más grandes y complejas, además de una mejor comprensión de la manera de actuar de estos sistemas.

1.4 Justificación

La comprensión de la conducta de los electrolitos en solución en torno a una partícula grande con carga discreta es esencial para muchos campos de la ciencia y la tecnología, sin embargo, el estudio de este tipo de sistemas mediante simulaciones computacionales es muy demandante en términos de capacidad de procesamiento y tiempo de cálculo, lo que puede limitar el desarrollo de estudios detallados y la comprensión completa del comportamiento de los sistemas.

La implementación de un algoritmo de paralelización puede acelerar significativa-

mente el tiempo de simulación y permitir realizar simulaciones más grandes y complejas, lo que a su vez puede llevar a una mejor comprensión del modo de actuar del sistema mencionado. Además, la implementación de este algoritmo puede tener implicaciones en el desarrollo de herramientas computacionales para otros sistemas físicos complejos.

1.5 Contenido

El trabajo presente se divide en seis capítulos, el primero es la *introducción*, donde se presentan la motivación, la problemática, los objetivos y la justificación de este trabajo.

El segundo capítulo es el *marco teórico*, que abarca los temas necesarios para entender la simulación que se va a desarrollar, como la dinámica browniana, la energía potencial electrostática, el potencial de núcleo repulsivo, entre otros. También incluye una introducción al cómputo paralelo, donde se habla de las herramientas que se van a utilizar para la implementación de la paralelización de la simulación.

En el tercer capítulo se presentan las *herramientas y recursos* utilizados para la implementación del programa, como el lenguaje de programación, las bibliotecas, los compiladores y más.

El cuarto capítulo presenta los algoritmos utilizados para implementar la simulación. En la primera parte de este capítulo se encuentran las implementaciones del programa que construye la partícula con carga discreta y en la segunda parte las implementaciones que simulan el electrolito alrededor de esta partícula con carga discreta.

El capítulo cinco tiene las pruebas que se llevaron a cabo para validar los programas y el análisis de los resultados de dos de los casos estudiados con la simulación.

El último capítulo es el de *conclusiones*, en este se resume el trabajo realizado, los resultados, los aportes y recomendaciones para mejorar aún más la eficiencia de la simulación.

Nota: En la versión pdf de este texto, las palabras y los números resaltados en azul están asociados mediante un hipervínculo a una cita, sección, ecuación, tabla o figura presente en este documento. Por lo anterior, se sugiere hacer clic sobre estos para acceder a dichas secciones o pasar el puntero sobre ellas para ver su contenido si se utiliza una aplicación para leer archivos pdf.

CAPÍTULO 2

Marco teórico

En este capítulo se introducirán los conceptos clave para entender la simulación desarrollada.

2.1 Dinámica browniana

La dinámica browniana es un fenómeno físico que describe el movimiento aleatorio e impredecible de partículas suspendidas en un fluido o gas. Este movimiento se debe a la colisión de las moléculas del fluido o gas con las partículas, es un movimiento estocástico. Se puede observar en diversos sistemas, desde partículas suspendidas en líquidos hasta en la trayectoria de partículas en el aire.

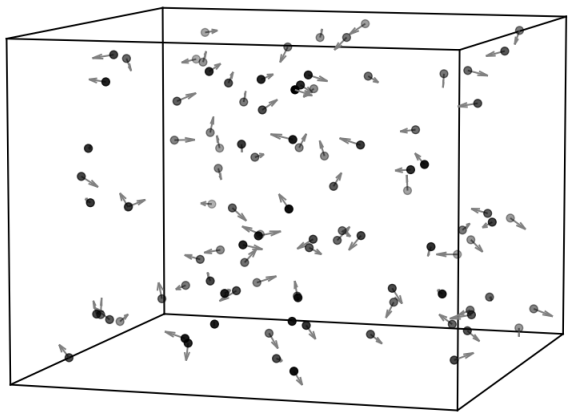

Figura 2.1: Representación de partículas con dirección aleatoria suspendidas en un fluido o gas.

La comprensión de la dinámica browniana es fundamental para entender el com-

portamiento de partículas en sistemas físicos, biológicos y químicos. Por ello, se ha modelado matemáticamente mediante la ecuación de Langevin y la ecuación de Smoluchowski [4]. La ecuación de Smoluchowski se utiliza para describir el movimiento de partículas de tamaño nanométrico en un medio fluido o gaseoso, tiene en cuenta la interacción entre partículas, el medio circundante y la energía térmica del medio que impulsa el movimiento aleatorio de las partículas, por lo que es ideal para las simulación de este trabajo.

La ecuación de Smoluchoski que describe el movimiento de una partícula cargada en solución, propuesta por Ermak [5], tiene la siguiente forma:

$$\frac{\partial P}{\partial t} = D\nabla^2 P - \frac{D}{kT}\nabla \cdot (\mathbf{F}(\mathbf{r}, t)P)) \tag{2.1}$$

donde P es la función de distribución de probabilidad, que da la probabilidad de que la partícula cargada realice un desplazamiento Δr en un intervalo de tiempo Δt. D es el coeficiente de difusión para la partícula, k es la constante de Boltzmann, T la temperatura absoluta y $\mathbf{F}(\mathbf{r}, t)$ es la fuerza que experimenta la partícula debido a las demás partículas en el sistema. En las siguientes secciones se presentan las fuerzas que actúan en la simulación.

Para un Δt suficientemente pequeño, las partículas cargadas más rápidas se mueven una distancia corta, por lo que se toma $\mathbf{F}(\mathbf{r}, t)$ como constante durante Δt y se puede dar la solución siguiente:

$$P\left[\mathbf{r}(\Delta t), \mathbf{F}\right] = \left(\frac{1}{4\pi D\Delta t}\right)^{3/2} \cdot exp\left(\frac{-\left|\mathbf{r}(t+\Delta t) - \mathbf{r}(t) - \frac{D}{kT}\mathbf{F}(t)\Delta t\right|^2}{4D\Delta t}\right) \tag{2.2}$$

Quedando el desplazamiento medio como:

$$\langle \Delta \mathbf{r} \rangle = \frac{D}{kT} \cdot \mathbf{F}(t) \cdot \Delta t \tag{2.3}$$

Y la ecuación que da la posición de una partícula en el tiempo $t + \Delta t$, es la siguiente:

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \sqrt{2D\Delta t} \cdot \mathbf{G}(0, 1) + \frac{D}{kT} \cdot \mathbf{F}(t) \cdot \Delta t \tag{2.4}$$

donde el termino $\sqrt{2D\Delta t} \cdot \mathbf{G}(0, 1)$ es el desplazamiento aleatorio que realiza la partícula debido al movimiento estocástico, siendo $G(0, 1)$ una función de distribución de probabilidad con media cero y varianza uno.

2.2 Energía potencial electrostática

La ley de Coulomb establece que la fuerza electrostática entre dos cargas eléctricas es proporcional al producto de sus cargas e inversamente proporcional al cuadrado de la distancia entre ellas. Matemáticamente, la ley de Coulomb se puede expresar en forma vectorial como:

$$\mathbf{F}^{\mathrm{el}}(\mathbf{r_{ij}}) = \frac{1}{4\pi\epsilon_0\epsilon_r}\frac{q_i q_j}{r_{ij}^2}\hat{\mathbf{r}}_{\mathbf{ij}} \tag{2.5}$$

donde $\mathbf{F}^{\mathrm{el}}(\mathbf{r_{ij}})$ es la fuerza electrostática sobre la carga q_i debido a la carga q_j, separadas una distancia r_{ij}, ϵ_0 es la constante dieléctrica del vacío y ϵ_r es la constante dieléctrica relativa del medio. $\hat{\mathbf{r}}_{\mathbf{ij}}$ es el vector unitario que apunta de la carga j a la carga i [6], es lo mismo que:

$$\hat{\mathbf{r}}_{\mathbf{ij}} = \frac{\mathbf{r_{ij}}}{r_{ij}} = \frac{\mathbf{r_i} - \mathbf{r_j}}{|\mathbf{r_i} - \mathbf{r_j}|} \tag{2.6}$$

La fuerza electrostática que actúa sobre una partícula es conservativa, lo que significa que el trabajo realizado por esta sobre una partícula en un camino cerrado es cero [7], por lo tanto existe un potencial escalar tal que:

$$\mathbf{F}^{\mathrm{el}}(\mathbf{r_{ij}}) = -\vec{\nabla}U^{\mathrm{el}}(\mathbf{r_{ij}}) \tag{2.7}$$

donde $\vec{\nabla}$ es el operador gradiente y $U^{\mathrm{el}}(\mathbf{r_{ij}})$ es el potencial escalar llamado energía potencial electrostática, que actúa entre las cargas q_i y q_j.

Entonces, partiendo de la ecuación de la ley de Coulomb (ec. 2.5) y despejando $U^{\mathrm{el}}(\mathbf{r_{ij}})$ de la ecuación que relaciona la fuerza electrostática con la energía potencial electrostática:

$$U^{\mathrm{el}}(\mathbf{r_{ij}}) = \frac{1}{4\pi\epsilon_0\epsilon_r}\frac{q_i q_j}{r_{ij}} \tag{2.8}$$

por lo que la energía potencial electrostática entre dos cargas puntuales es proporcional al producto de las cargas eléctricas e inversamente proporcional a la distancia entre ellas. La energía potencial electrostática se utiliza para describir la energía almacenada en un sistema de cargas eléctricas y tiene numerosas aplicaciones en la física y la ingeniería, como la física de plasmas y la física de semiconductores.

En el caso de un sistema de múltiples cargas, la fuerza total sobre la carga i es la suma de las fuerzas que actúan sobre ella debido a las demás cargas j:

$$\mathbf{F}_i^{\mathrm{el}} = \sum_{j=1, j\neq i}^{N} \mathbf{F}^{\mathrm{el}}(\mathbf{r_{ij}}) = \frac{1}{4\pi\epsilon_0\epsilon_r} \sum_{j=1, j\neq i}^{N} \frac{q_i q_j}{r_{ij}^2} \hat{\mathbf{r}}_{\mathbf{ij}} \tag{2.9}$$

Y la energía potencial electrostática por partícula de un sistema con N cargas puntuales es:

$$\bar{U}^{\mathrm{el}} = \frac{1}{2N} \sum_{i=1, i\neq j}^{N} \sum_{j=1}^{N} U^{\mathrm{el}}(\mathbf{r_{ij}}) = \frac{1}{2N} \cdot \frac{1}{4\pi\epsilon_0\epsilon_r} \sum_{i=1, i\neq j}^{N} \sum_{j=1}^{N} \frac{q_i q_j}{r_{ij}} \tag{2.10}$$

donde N es el número de cargas o partículas cargadas y el factor $1/2$ es porque la energía potencial entre la carga i y la carga j se cuenta dos veces en la suma.

En la simulación, se toman las valencias de las partículas cargadas para calcular la fuerza electrostática y la energía potencial electrostática por partícula entre estas. En la práctica, el cálculo de la fuerza y la energía potencial electrostática puede ser computacionalmente costoso, especialmente en sistemas con muchas cargas.

2.3 Potencial de núcleo repulsivo

En la naturaleza las partículas cargadas se pueden juntar, pero no superponer, en cambio, en una simulación esto puede pasar, causando resultados no realistas, por lo que es necesario una fuerza de repulsión para mantener una distancia mínima entre las partículas.

El potencial de Lennard-Jones se utiliza para evitar el solapamiento entre partículas en simulaciones de dinámica molecular, para las simulaciones que se llevarán a cabo se usara el potencial modificado de Lennard-Jones siguiente:

$$U^{\mathrm{rc}}(r_{ij}) = \begin{cases} 4\left[\left(\frac{\sigma}{r_{ij}-\Delta_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}-\Delta_{ij}}\right)^{6} + \frac{1}{4}\right] & r_{ij} \leq \Delta_{ij} + 2^{1/6}\sigma \\ 0 & r_{ij} > \Delta_{ij} + 2^{1/6}\sigma \end{cases} \tag{2.11}$$

$$\Delta_{ij} = \frac{d_i + d_j}{2} - \sigma \tag{2.12}$$

donde r_{ij} es la distancia entre las partículas i y j, d_i y d_j son los diámetros de las partículas i y j, σ es el parámetro de distancia y $\Delta_{ij} + 2^{1/6}\sigma$ es la distancia a la cual el potencial entre las partículas es mínimo [8]. El comportamiento de la ecuación 2.11 se puede ver en la figura 2.2.

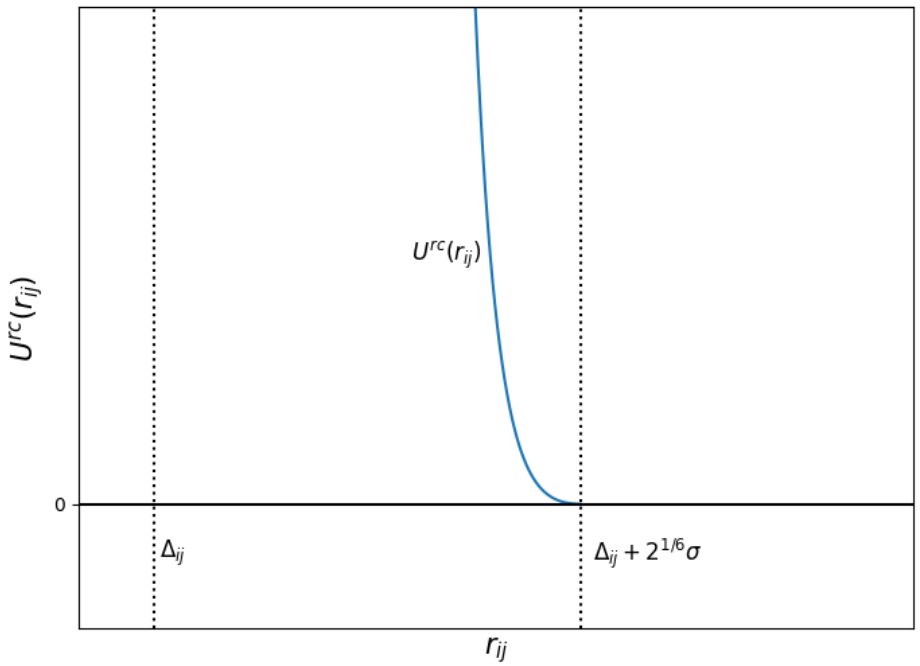

Figura 2.2: Potencial de núcleo repulsivo. Cuando la distancia entre dos partículas es menor o igual que $\Delta_{ij} + 2^{1/6}\sigma$, entra en acción el potencial repulsivo de Lennard-Jones modificado, si la distancia es mayor que $\Delta_{ij} + 2^{1/6}\sigma$ el potencial de núcleo repulsivo se anula.

El parámetro σ es de vital importancia, este es mayor o igual a cero y menor o igual que $\frac{d_i+d_j}{2}$. Con $\sigma = 0$ las partículas simulan ser esferas duras, esto quiere decir que en la simulación las partículas no pueden ser deformadas ni atravesadas, en cambio, para $\sigma = \frac{d_i+d_j}{2}$, estas podrían acercarse entre sí una distancia menor a la suma de sus radios, pero no se sobrepondrían en ningún momento debido a este potencial de repulsión.

El potencial de núcleo repulsivo se utiliza comúnmente en simulaciones de sistemas de partículas blandas, como polímeros, coloides y proteínas [9]. La función de potencial suave permite que las partículas se aproximen sin solaparse, lo que permite simular la dinámica de sistemas a escalas macroscópicas.

El potencial de núcleo repulsivo U^{rc} también es una función escalar y diferenciable que cumple con la regla de la conservación del momento [7], por lo que se cumple:

$$\mathbf{F}^{\text{rc}}(\mathbf{r_{ij}}) = -\vec{\nabla} U^{\text{rc}}(\mathbf{r_{ij}}) \tag{2.13}$$

Entonces, la fuerza de interacción entre dos partículas i y j, asociada al potencial de núcleo repulsivo U^{rc}, de acuerdo a la ecuación 2.11 es:

$$\mathbf{F}^{\text{rc}}(\mathbf{r_{ij}}) = \begin{cases} \frac{24}{\sigma}\left[2\left(\frac{\sigma}{r_{ij}-\Delta_{ij}}\right)^{13} - \left(\frac{\sigma}{r_{ij}-\Delta_{ij}}\right)^{7}\right]\hat{\mathbf{r}}_{\mathbf{ij}}, & r_{ij} < \Delta_{ij} + 2^{1/6}\sigma \\ 0, & r_{ij} \geq \Delta_{ij} + 2^{1/6}\sigma \end{cases} \tag{2.14}$$

de forma análoga que para la fuerza de Coulomb, $\hat{\mathbf{r}}_{\mathbf{ij}}$ es el vector unitario que apunta desde la partícula j hacia la partícula i [6].

La fuerza de núcleo repulsivo es utilizada en la simulación para evitar la superposición de las partículas cargadas, así como para su confinamiento, evitando que se salgan del sistema.

2.4 Método de descenso de gradiente

El método de descenso de gradiente es una técnica de optimización que se utiliza para encontrar el mínimo de una función escalar $f(\mathbf{x})$ mediante la iteración de la siguiente ecuación:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \vec{\nabla} f(\mathbf{x}_n) \tag{2.15}$$

donde $\mathbf{x}_n$ es el valor de la variable de optimización en la iteración n, α es la longitud del paso o tamaño de paso, y $\vec{\nabla} f(\mathbf{x}_n)$ es el gradiente de $f(\mathbf{x}_n)$ evaluado en $\mathbf{x}_n$ [10].

El método de descenso de gradiente se utiliza a menudo en la optimización de funciones en el campo de la simulación de sistemas físicos, ya que muchas veces se busca minimizar la energía potencial electrostática del sistema. En este caso, la función $f(\mathbf{x})$ representa la energía potencial del sistema, y $\mathbf{x}$ representa el conjunto de coordenadas de todas las partículas en el sistema.

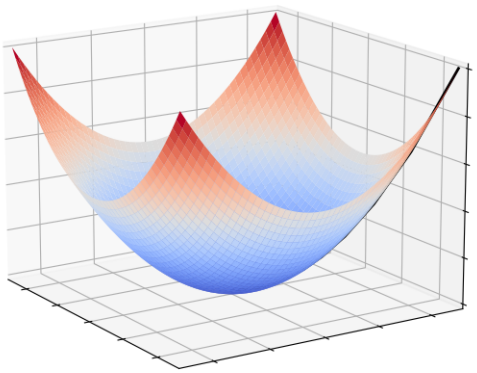

Figura 2.3: Representación 3D del método de descenso de gradiente en una función de dos variables. La tonalidad azul representa la zona donde la función se minimiza.

El tamaño de paso α es un parámetro crítico en el método de descenso de gradiente, ya que si es demasiado grande, el método puede diverger, mientras que si es demasiado pequeño, puede requerir un número excesivo de iteraciones para alcanzar la convergencia. Por lo tanto, la elección de α debe ser cuidadosamente seleccionada para garantizar la convergencia y la eficiencia del método.

Este método se utiliza para encontrar los sitios de carga del macroion, de la sección 4.1.

2.5 Densidad de carga por unidad de volumen $\rho(r)$

La densidad de carga por unidad de volumen $\rho(r)$ es una magnitud física que se utiliza para describir la cantidad de carga eléctrica presente en una región determinada del espacio. Para distribuciones de carga continua, se define como:

$$\rho(r) = \frac{dq}{dV} \tag{2.16}$$

donde dq es la carga contenida en el volumen infinitesimal dV que se encuentra en r [11].

Para la simulación, nos interesa saber la cantidad de cargas positivas y negativas que se encuentran entre cascarones esféricos concéntricos separados por una distancia pequeña llamada dr. Esto se ve más claro con la figura 2.4, donde se muestran los cascarones separados por una distancia dr y entre estos se encuentran cargas puntuales.

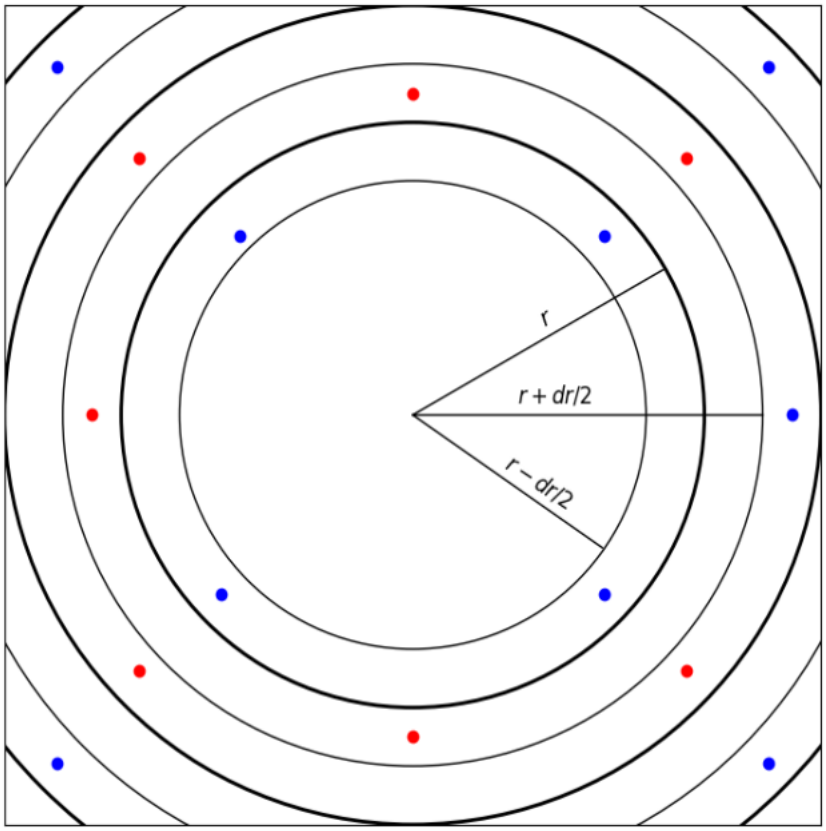

Figura 2.4: Las circunferencias de línea gruesa representan los cascarones esféricos en una solución con partículas cargadas, están separados entre si por una distancia dr. Los puntos azules representan cargas positivas y los rojos cargas negativas.

Tomando un dr muy pequeño se puede calcular la densidad de carga por unidad de volumen de forma precisa para cualquier distancia. El volumen entre dos cascarones que se encuentran en r y $r + dr$ es:

$$dV = \frac{4\pi}{3}\left[(r + dr)^3 - r^3\right] \tag{2.17}$$

Para una distancia d tal que $r < d \leq r + dr$, la densidad de carga por unidad de volumen en d es:

$$\rho(d) = \frac{\text{Número total de cargas en } (r, r + dr]}{\text{Volumen en } [r, r + dr]} \tag{2.18}$$

La densidad por unidad de volumen de cargas positivas para la distancia d es:

$$\rho_+(d) = \frac{\text{Número de cargas positivas en } (r, r+dr]}{\text{Volumen en } [r, r+dr]} \tag{2.19}$$

Y la densidad por unidad de volumen de cargas negativas para la distancia d es:

$$\rho_-(d) = \frac{\text{Número de cargas negativas en } (r, r+dr]}{\text{Volumen en } [r, r+dr]} \tag{2.20}$$

Con $\rho_+(r)$ y $\rho_-(r)$ se podrá analizar de forma precisa el comportamiento de las cargas en los sistemas que se van a simular.

2.6 Carga integrada $P(r)$

La carga integrada $P(r)$ es una propiedad importante en sistemas de partículas cargadas, esta nos indica la suma de las cargas que se encuentran dentro de una esfera de radio r, incluyendo la carga en la superficie de la esfera. La carga integrada se define como:

$$P(r) = 4\pi \int_0^r \rho(r')r'^2 dr' \tag{2.21}$$

donde $\rho(r')$ es la densidad de carga por unidad de volumen en la posición r'. En sistemas electroneutros, se cumple que $\lim_{r\to\infty} P(r) = 0$ [12].

La carga integrada es útil para caracterizar la distribución de carga en una muestra y puede ser calculada a partir de la distribución de densidad de carga por unidad de volumen. El valor de la carga integrada a diferentes distancias r puede ser utilizado para entender la interacción electrostática entre las partículas.

Por ejemplo, de la figura 2.4, $P(r-\frac{dr}{2}) = 0$, $P(r) = 4$, $P(r+\frac{dr}{2}) = 1$ y $P(r+dr) = -2$. $P(r)$ varía según la distribución de las cargas.

2.7 Potencial eléctrico $V(r)$

El potencial eléctrico $V(r)$ es una medida de la energía potencial eléctrica que una carga q experimenta debido a la presencia de una carga puntual Q en un punto r. En el

caso de una distribución de cargas, el potencial eléctrico se define como la suma de las contribuciones individuales de cada carga. El potencial eléctrico $V(r)$ a una distancia r de una carga Q se define como:

$$V(r) = \frac{1}{4\pi\epsilon_0\epsilon}\frac{Q}{r} \tag{2.22}$$

donde ϵ es la permitividad eléctrica del medio en el que se encuentra la carga y ϵ_0 es la permitividad eléctrica del vacío.

Para una distribución de cargas, el potencial eléctrico se puede calcular como:

$$V(r) = \frac{1}{4\pi\epsilon_0\epsilon}\int \frac{\rho(r')}{r}dV' \tag{2.23}$$

donde $\rho(r')$ es la densidad de carga en el punto r' y dV' es un elemento de volumen en torno a r'. Para una distribución de cargas esféricamente simétrica, el potencial eléctrico $V(r)$ puede expresarse en términos de la carga integrada $P(r)$ como:

$$V(r) = \frac{1}{4\pi\epsilon_0\epsilon}\int_0^r \frac{P(r')e_0}{r'^2}dr' \tag{2.24}$$

donde ϵ_0 es la carga elemental. En la práctica, el potencial eléctrico se utiliza para describir fenómenos electrostáticos, tales como la interacción entre cargas o la forma de comportarse de conductores y dieléctricos en campos eléctricos. La capacidad de calcular el potencial eléctrico en sistemas complejos es fundamental para la comprensión y el diseño de dispositivos y materiales en el campo de la electrónica y la electroquímica [6].

2.8 Unidades reducidas

Las unidades reducidas son un sistema de unidades adimensional utilizado en física y química para simplificar el análisis de problemas y ecuaciones sin la necesidad de trabajar con unidades físicas específicas. En este sistema, todas las cantidades se expresan en términos de unidades básicas y se normalizan de acuerdo a una escala común [8].

En el sistema de esta simulación las distancias son del tamaño de ángstroms. Para simplificar el análisis del programa, mejorar la estabilidad numérica, reducir el efecto de errores de redondeo y otros problemas que surgen al trabajar con números muy

pequeños o muy grandes, se usarán unidades reducidas para todas las distancias, como la separación entre partículas, la σ del potencial de núcleo repulsivo, el radio de las partículas cargadas, entre otras.

Entonces, la distancia d reducida es:

$$d^* = \frac{d}{1\ \text{Å}} \tag{2.25}$$

siendo d^* la distancia d reducida y $1\ \text{Å} = 10^{-10}$ m. Esto también aplicará para los vectores posición de todas las partículas en las simulaciones y en los diferenciales que se usarán en las integrales.

También se llevará a cabo un escalamiento de energía, dividiendo la fuerza y el potencial electrostático entre k_BT con el fin de tener en cuenta el efecto de la temperatura en el sistema, haciendo un poco más realista la simulación. Para esto se usará la longitud de Bjerrum, que se define como la distancia a la cual la energía electrostática entre dos cargas opuestas en un medio dieléctrico es igual a la energía térmica del medio (k_BT)[13], esto es:

$$l_B = \frac{e^2}{4\pi\epsilon_0\epsilon_r k_B T} \tag{2.26}$$

donde l_B es la longitud de Bjerrum, e es la carga elemental, ϵ_0 es la permitividad del vacío, ϵ_r es la constante dieléctrica relativa del medio, k_B es la constante de Boltzmann y T es la temperatura en kelvin. La longitud de Bjerrum reducida es:

$$l_B^* = \frac{e^2}{4\pi\epsilon_0\epsilon_r k_B T \cdot 10^{-10}\ \text{m}} \tag{2.27}$$

Entonces, la fuerza electrostática reducida es:

$$\mathbf{F}^{\text{el}*}(\mathbf{r}_{\mathbf{ij}}^*) = \frac{\mathbf{F}^{\text{el}}(\mathbf{r}_{\mathbf{ij}}^*)}{k_BT} = \frac{l_B^*}{10^{-10}\ \text{m}} \cdot \frac{z_i z_j}{r_{ij}^{*2}} \hat{\mathbf{r}}_{\mathbf{ij}}^* \tag{2.28}$$

donde r_{ij}^* es la distancia reducida entre las partículas i y j, z_i y z_j son las valencias de dichas partículas y $\hat{\mathbf{r}}_{\mathbf{ij}}^*$ es el vector unitario reducido, que de acuerdo a su definición en la sección 2.2, es igual que $\hat{\mathbf{r}}_{\mathbf{ij}}$.

La energía potencial electrostática reducida es:

$$U^{\mathrm{el}*}(\mathbf{r}^*_{\mathbf{ij}}) = \frac{U^{\mathrm{el}}(\mathbf{r}^*_{\mathbf{ij}})}{k_B T} = l^*_B \cdot \frac{z_i z_j}{r^{*2}_{ij}} \tag{2.29}$$

En el capítulo cuatro se explicará la implementación en la simulación de estás ecuaciones con unidades reducidas.

2.9 Macroion con carga discreta

En la física y la química, el término macroion se refiere a partículas grandes que tienen una carga eléctrica neta significativa debido a la presencia de grupos cargados en su superficie o en su interior. Un macroion con carga discreta es una partícula grande que tiene una carga eléctrica distribuida en un número finito de sitios a lo largo de su superficie.

Al simular un macroion con carga discreta, es necesario distribuir la carga en sitios de manera que el sistema se comporte como una sola partícula, similar a una esfera. Esta distribución de carga permite que las fuerzas electrostáticas internas se equilibren y que el macroion se comporte como una entidad coherente. La elección de una distribución de carga esférica ayuda a lograr esta conducta deseada, ya que la esfera es una forma que minimiza la energía y proporciona una simetría adecuada para el sistema. En las siguientes figuras se puede ver la configuración con la energía potencial electrostática mínima para 120 y 1000 partículas, en esferas de radio igual a 10 y 25 ángstroms respectivamente.

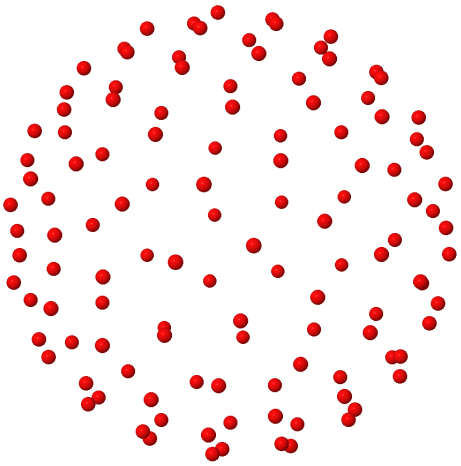

Figura 2.5: Configuración con energía potencial mínima para 120 partículas.

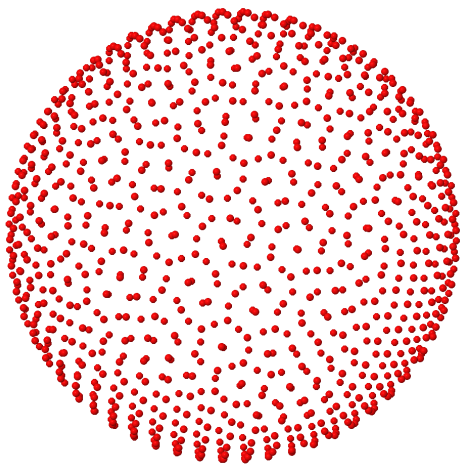

Figura 2.6: Configuración con energía potencial mínima para 1000 partículas.

2.10 Electrolito confinado

Un electrolito es una solución que contiene iones libres y es capaz de conducir electricidad debido a la presencia de estos iones cargados. En este trabajo se analizará la forma de comportarse de un electrolito confinado alrededor de un macroion con carga discreta, como el de la figura 2.7. En este caso, el electrolito es un sistema de partículas cargadas en solución que interactúan entre sí y con el macroion cargado, cuya dinámica es afectada por factores como la temperatura, el coeficiente de difusión, la fuerza electrostática y el potencial de núcleo repulsivo.

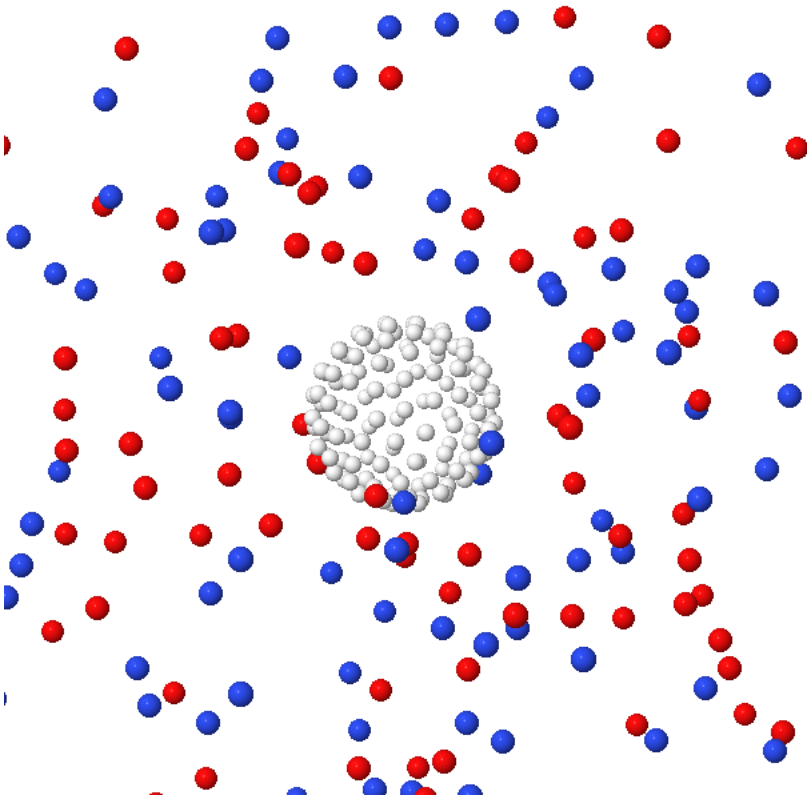

Figura 2.7: Representación de un electrolito alrededor de un macroion con carga discreta. Los puntos azules representan iones positivos, los rojos iones negativos y los blancos los sitios de carga del macroion.

El potencial de nucleo repulsivo se utiliza para confinar el electrolito en una esfera y para evitar la superposición de los iones en la simulación, permitiendo un estudio más preciso y realista de la dinámica del electrolito. La fuerza electrostática es la interacción entre las cargas eléctricas de los iones, responsable de la formación de estructuras complejas y del movimiento del electrolito.

La simulación de la dinámica browniana de este sistema permite estudiar el comportamiento del electrolito, como la densidad de carga por unidad de volumen, la carga integrada y el potencial eléctrico, lo que tiene implicaciones importantes en la comprensión de procesos electroquímicos y la síntesis de materiales [14].

2.11 Cómputo paralelo

El cómputo paralelo es una técnica utilizada para mejorar la eficiencia y el rendimiento de los programas computacionales, en particular aquellos que realizan tareas intensivas en cálculo y procesamiento. En esta sección se presentarán diferentes formas de hacer cómputo paralelo, sus características, diferencias y los paradigmas de programación paralela seleccionados para implementar en la simulación.

2.11.1 Paradigmas de programación paralela

Existen varias formas de hacer cómputo paralelo, entre las cuales se encuentran:

OpenMP: es un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno que se utilizan para especificar el paralelismo de alto nivel en programas escritos en Fortran y C/C++. OpenMP se utiliza para dividir una tarea en subtareas que se pueden ejecutar en paralelo en múltiples núcleos de CPU (unidad de procesamiento central, por sus sigla en inglés) [15].

MPI: es una interfaz de paso de mensajes estándar para la computación paralela en sistemas de memoria distribuida. MPI no es un lenguaje de programación en sí mismo, sino una biblioteca de funciones que se pueden llamar desde código escrito en C, C++ o Fortran para escribir programas paralelos. Con MPI, un comunicador MPI puede ser creado dinámicamente y tener múltiples procesos ejecutándose simultáneamente en nodos separados de clusters. En este contexto, se considera un paradigma de programación de bajo o medio nivel [16].

OpenACC: es un modelo de programación basado en directivas que permiten que regiones de código se trasladen desde una CPU a una unidad de procesamiento gráfico (GPU) para su cómputo. La programación con OpenACC es similar a la programación con OpenMP, pero se enfoca en la programación de alto nivel para la GPU. OpenACC es compatible con C/C++ y Fortran [17].

CUDA: es una plataforma de computación paralela y un modelo de programación de bajo nivel desarrollado por NVIDIA para la computación en GPU. CUDA proporciona un acceso completo a las opciones de la GPU y se utiliza principalmente para

aplicaciones de alta computación y aprendizaje profundo [18].

Cada forma de cómputo paralelo tiene sus propias características, por lo que se deben seleccionar de acuerdo con las necesidades y requisitos específicos del proyecto. OpenMP y MPI son enfoques de cómputo paralelo que se utilizan principalmente para computadoras de múltiples núcleos y sistemas de memoria distribuida, respectivamente. Por otro lado, OpenACC y CUDA se enfocan en la programación de la GPU para aplicaciones de alta computación y aprendizaje profundo. Mientras que OpenACC tiene una gran similitud con OpenMP, proporcionando una interfaz de alto nivel para la programación de la GPU, CUDA proporciona un acceso completo al hardware de la GPU.

2.11.2 OpenMP y OpenACC

En este trabajo se busca mejorar el rendimiento de un programa intensivo en cálculo y procesamiento, por lo que se consideró utilizar diferentes formas de cómputo paralelo. Después de evaluar las características y las necesidades del proyecto, se decidió utilizar OpenMP y OpenACC como enfoques de cómputo paralelo.

La elección de OpenMP se basó en su capacidad para paralelizar eficientemente tareas en múltiples núcleos de CPU, lo que es adecuado para la simulación que se va a llevar a cabo en este trabajo.

Por otro lado, la elección de OpenACC se debe a su similitud con OpenMP y su capacidad para utilizar la GPU para acelerar las operaciones de cálculo y procesamiento. Además, OpenACC es compatible con C/C++ y Fortran, que son los lenguajes utilizados en la computación científica y de alto rendimiento. *"More Science, Less Programming"* es como NVIDIA invita a usar OpenACC con sus compiladores de alto rendimiento [19].

Se evaluó la opción de también utilizar CUDA, pero debido a que esta plataforma implica una programación más cercana al nivel de hardware (por tanto más compleja), se estimó que su implementación requeriría más tiempo del previsto.

2.11.3 Race condition

En programación paralela, *race condition* (condición de carrera) es una situación en la que múltiples hilos o procesos acceden simultáneamente a una variable compartida y modifican su valor, lo que puede dar lugar a resultados no deterministas.

Cuando en una paralelización hay *race condition*, el resultado final depende del

orden en que se ejecuten las operaciones. Si dos o más hilos intentan actualizar una variable al mismo tiempo, puede ocurrir que las actualizaciones se superpongan y se pierda información. Por ejemplo, si varios hilos intentan incrementar una variable en una unidad al mismo tiempo, el valor final de la variable puede ser menor de lo que debería ser, ya que las actualizaciones se superpondrán y algunas de ellas se perderán.

El *race condition* es un problema común en la programación paralela y puede ser difícil de detectar y depurar, ya que los resultados pueden ser impredecibles y no reproducibles. Para evitar este, se utilizan diversas técnicas de sincronización que permiten a los hilos o procesos coordinar su acceso a las variables compartidas y garantizar la coherencia de los datos.

En OpenMP y OpenACC, se pueden utilizar las cláusulas de `atomic` y `reduction` para garantizar la coherencia de los datos y evitar *race condition* en las operaciones de lectura y escritura en la memoria compartida. En el siguiente capítulo se verán en detalle estas cláusulas.

CAPÍTULO 3

Herramientas y recursos

3.1 C++

C++ es un lenguaje de programación ampliamente utilizado en la industria y en la academia debido a su gran eficiencia, flexibilidad y capacidad para trabajar con sistemas de bajo nivel y alto rendimiento. Por esto se decidió usar este lenguaje de programación para realizar la simulación.

Las librerías y herramientas de programación disponibles en C++ pueden ayudar a acelerar el proceso de desarrollo y aumentar la eficiencia del programa. Otra ventaja de C++ es su capacidad de acceso a memoria directa, lo que permite trabajar directamente con bloques de memoria y acceder a ellos de manera eficiente [20]. Esto es particularmente útil para la simulación de sistemas físicos, ya que permite una representación detallada de los datos y un control preciso sobre los cálculos.

3.1.1 IDE

Un entorno de desarrollo integrado (IDE, por sus siglas en inglés) es una herramienta que proporciona a los programadores un conjunto de herramientas integradas para facilitar el desarrollo de software. Esto puede incluir un editor de código, un depurador, un compilador y herramientas de automatización de tareas.

Visual Studio Code es un IDE de código abierto y multiplataforma que ofrece una amplia gama de herramientas para la programación de software. Es compatible con lenguajes de computación de alto rendimiento como C, C++ y Fortran, por esto es el IDE en el que se desarrolla el programa de simulación de este trabajo.

Además, Visual Studio Code ofrece una gran cantidad de extensiones y complementos que pueden mejorar la productividad del programador, como la integración con sistemas de control de versiones, la depuración de código en tiempo real y la gestión de

paquetes y bibliotecas de terceros [21].

3.1.2 Bibliotecas

Durante el desarrollo del programa de simulación se utilizaron diversas bibliotecas para mejorar su eficiencia y funcionalidad. A continuación, se listan y describen las bibliotecas utilizadas en el programa:

- **cmath:** Ofrece funciones matemáticas, como sin(), cos(), sqrt(), etc.
- **curand.h:** Contiene funciones para generar números aleatorios en la GPU.
- **fenv.h:** Detecta errores aritméticos como divisiones por cero y operaciones inválidas en números de coma flotante.
- **fstream:** Permite crear, leer y escribir archivos de texto.
- **iomanip:** Controla la precisión de los valores en la salida estándar.
- **iostream:** Facilita las funciones de entrada y salida estándar.
- **omp.h:** Controla hilos y procesos en paralelo utilizando OpenMP.
- **openacc.h:** Brinda funciones y directivas para programación en paralelo con OpenACC.
- **random:** Genera números pseudoaleatorios.
- **sstream:** Realiza operaciones de entrada/salida en cadenas.
- **stdlib.h:** Gestiona la memoria dinámica.
- **string:** Manipula cadenas de caracteres.
- **thread:** Obtiene información sobre los procesadores disponibles y controla los hilos de ejecución.
- **time.h:** Provee funciones para medir el tiempo transcurrido en programas.
- **unordered_set:** Maneja conjuntos desordenados.

Cada una de estas bibliotecas fue seleccionada por su capacidad para mejorar la eficiencia y funcionalidad del programa de simulación. El uso de estas bibliotecas permitió reducir el tiempo de ejecución del programa, mejorar la gestión de memoria y aumentar la precisión de los cálculos realizados [22].

3.1.3 Compiladores

Un compilador es un programa informático que se encarga de traducir el código fuente escrito en un lenguaje de programación a un código objeto de lenguaje de bajo nivel que puede ser ejecutado por una computadora. Existen diferentes compiladores para diferentes lenguajes de programación y arquitecturas de computadoras.

En la programación de alto rendimiento (HPC, por sus siglas en inglés), se utilizan compiladores específicos para aprovechar al máximo el hardware y obtener el mejor rendimiento posible en las aplicaciones. Estos compiladores están optimizados para la arquitectura de las unidades de procesamiento y las jerarquías de memoria, además de contar con características como la vectorización automática, la optimización de bucles, entre otras.

Entre los compiladores disponibles para HPC, se encuentran el Intel C++ Compiler (icpc) y el NVIDIA HPC Compiler (nvc++). Ambos compiladores soportan los estándares de C++ y cuentan con características avanzadas de optimización.

El compilador icpc es desarrollado por Intel y está diseñado para optimizar aplicaciones en arquitecturas Intel. Soporta características como OpenMP, que permite paralelizar secciones de código en diferentes hilos, y vectorización automática, que utiliza las instrucciones de vectorización de Intel para procesar múltiples elementos de datos al mismo tiempo [25].

Por su parte, nvc++ es un compilador desarrollado por NVIDIA para optimizar aplicaciones en GPUs NVIDIA. Ofrece características avanzadas de optimización de código y paralelización en hilos, soporta los estándares de programación en paralelo OpenACC, OpenMP y CUDA. Es una herramienta clave para el desarrollo de aplicaciones de cómputo de alto rendimiento en GPUs. [26].

3.1.4 DOD

En el programa desarrollado en este trabajo se utilizaron arreglos y no estructuras de datos, ya que en la computación de alto rendimiento (HPC), el *diseño orientado a datos* (DOD, por sus sigla en inglés) es imprescindible. Esta es una técnica que se enfoca en la forma en que los datos son almacenados en la memoria y accedidos por el programa. El diseño orientado a datos se refiere a:

- Operar en arreglos en lugar de en datos individuales, evitando la sobrecarga de llamadas y los fallos en la caché de instrucciones y datos.

- Preferir los arreglos en lugar de las estructuras de datos para un mejor uso de la memoria caché.
- Incrustar subrutinas en lugar de hacer llamadas a profundas rutinas.
- Controlar la asignación de memoria para evitar la reasignación de memoria en lugares que no se ven.
- Utilizar listas enlazadas basadas en arreglos contiguos para evitar las implementaciones de listas enlazadas estándar utilizadas en C y C++.

La técnica DOD se utiliza principalmente en el desarrollo de aplicaciones que requieren un alto rendimiento, como videojuegos, motores de renderizado, simulaciones y aplicaciones científicas. Al organizar los datos de manera eficiente, se pueden aprovechar mejor las capacidades del hardware, como la caché y los procesadores múltiples, para mejorar el rendimiento y la escalabilidad del sistema [23].

3.1.5 Vectorización

La vectorización es una técnica de optimización que consiste en procesar varias operaciones en paralelo utilizando instrucciones SIMD (Single Instruction, Multiple Data). En otras palabras, la CPU o la GPU ejecutan la misma operación en varios datos simultáneamente, en lugar de procesarlos uno por uno.

En la programación HPC, la vectorización es especialmente importante, ya que permite realizar cálculos complejos en grandes cantidades de datos de manera más rápida y eficiente. Los compiladores modernos suelen incluir técnicas de vectorización automática, que transforman el código fuente para aprovechar al máximo las capacidades SIMD de la CPU o la GPU.

Es importante destacar que no todos los algoritmos son fácilmente vectorizables, y que la vectorización no siempre garantiza una mejora significativa en el rendimiento del programa. En algunos casos, la sobrecarga generada por la vectorización puede incluso ralentizar el código.

En general, la vectorización es una técnica muy útil para acelerar el procesamiento de grandes cantidades de datos en paralelo, pero es importante evaluar su efectividad en cada caso particular y considerar otras técnicas de optimización en conjunto [23].

3.1.6 Punteros, asignación de memoria

En el lenguaje de programación C++, los punteros son variables que contienen direcciones de memoria, es decir, la ubicación en la memoria RAM donde se encuentra almacenado un valor o una variable. Los punteros son una herramienta muy útil para manipular estructuras de datos complejas y para realizar operaciones de asignación de memoria dinámica [22].

En el contexto de una simulación, donde es común trabajar con grandes cantidades de datos y estructuras de datos complejas, el uso de punteros y asignación de memoria dinámica es fundamental para poder crear y manipular los objetos necesarios para llevar a cabo la simulación.

Un ejemplo de asignación de memoria dinámica usando punteros en C++ es la siguiente línea de código:

```
double *restrict x = new double[N];
```

En esta línea, se declara un puntero llamado `x` que apunta a un arreglo de tipo `double` de tamaño `N`, el cual se reserva en la memoria RAM usando el operador `new`. El operador `new` es el encargado de asignar dinámicamente la memoria necesaria para el arreglo y devuelve la dirección de memoria donde empieza la zona reservada.

La palabra clave `restrict` se usa para indicarle al compilador que la memoria a la que apunta el puntero `x` no será accesible mediante otros punteros. Esta optimización puede mejorar el rendimiento del programa en algunos casos, pero es necesario asegurarse de que se utiliza correctamente para evitar errores en tiempo de ejecución [23].

Es necesario liberar la memoria reservada por la asignación dinámica al finalizar su uso, para ello se utiliza el operador `delete`:

```
delete[] x;
```

Este operador libera la memoria reservada por el arreglo `x` y permite que la memoria sea reutilizada por otros programas o procesos en el sistema.

Se debe tener en cuenta que el uso incorrecto de punteros y la asignación dinámica de memoria puede llevar a errores en tiempo de ejecución y fugas de memoria, lo cual puede causar que el programa falle o consuma más memoria de la necesaria. Por lo tanto, es recomendable tener un cuidado especial al trabajar con estas herramientas y asegurarse de que se están utilizando correctamente.

3.1.7 Números pseudoaleatorios

Los números pseudoaleatorios son una herramienta esencial en la simulación de sistemas complejos. A diferencia de los números verdaderamente aleatorios, que se generan a partir de procesos físicos impredecibles, los números pseudoaleatorios se generan mediante algoritmos deterministas. A pesar de esta aparente limitación, los números pseudoaleatorios son lo suficientemente impredecibles como para ser útiles en la mayoría de las aplicaciones de simulación.

Para este trabajo, se utilizan varias clases y funciones de la biblioteca `random` de C++ para generar números pseudoaleatorios. En particular, se utilizan las siguientes:

- `random_device`: Clase que se utiliza para generar números aleatorios a partir de fuentes de entropía del sistema, como la hora actual del reloj o el ruido del hardware.
- `mt19937`: Clase generadora de números pseudoaleatorios basado en el algoritmo Mersenne Twister.
- `uniform_real_distribution`: Clase que se utiliza para generar números pseudoaleatorios distribuidos uniformemente en un rango determinado.
- `uniform_int_distribution`: Clase que se utiliza para generar números pseudoaleatorios enteros distribuidos uniformemente en un rango determinado.
- `normal_distribution`: Clase que se utiliza para generar números pseudoaleatorios distribuidos normalmente con una media y una desviación estándar especificadas.

En el programa, se utilizan estas clases y funciones para generar posiciones aleatorias, asignar las valencias a los sitios del macroion con carga discreta y simular una distribución de probabilidad con media cero y varianza uno para la dinámica browniana.

Por ejemplo, para generar posiciones aleatorias se utilizan las siguientes líneas de código:

```
random_device rd{};
mt19937 gen{rd()};
uniform_real_distribution<> pos{-1.0, 1.0};
x[i] = pos(gen); y[i] = pos(gen); z[i] = pos(gen);
```

donde `rd{}` es un objeto de la clase `random_device` que genera una semilla aleatoria que se utiliza como entrada para el generador de números pseudoaleatorios `gen{}` de la

clase `mt19937`. `pos{-1.0, 1.0}` es un objeto de la clase `uniform_real_distribution` que genera números pseudoaleatorio distribuidos uniformemente en el rango $[-1, 1)$, y finalmente, usando el objeto `pos` y el generados de números pseudoaleatorios de `mt19937` inicializado con `rd{}`, en la última línea se asignan posiciones pseudoaleatorias diferentes a los arreglos `x`, `y` y `z` [24].

3.2 OpenMP

OpenMP (Open Multi-Processing) es una API para la programación en paralelo de aplicaciones que corren en CPUs y GPUs, especialmente diseñada para sistemas de memoria compartida. Esta herramienta permite a los desarrolladores escribir programas paralelos en lenguajes como C/C++, y Fortran, utilizando directivas de compilación conocidas como pragmas.

Un pragma es una directiva específica del compilador que permite a los programadores agregar información adicional al código fuente para controlar cómo se compila o se ejecuta el programa. Los pragmas son en general específicos de un lenguaje de programación y son ignorados por los compiladores que no los reconocen. En C y C++, los pragmas comienzan con `#pragma` y se utilizan, por ejemplo, para indicar que una sección de código debe ser paralelizada con OpenMP.

Una de las ventajas principales de OpenMP es su facilidad de uso y su integración con compiladores existentes, lo que la convierte en una opción atractiva para aquellos que buscan aprovechar la paralelización de sus aplicaciones sin tener que aprender nuevas herramientas o lenguajes de programación.

Al utilizar OpenMP, los desarrolladores pueden aprovechar los núcleos de las CPUs y GPUs para ejecutar tareas simultáneamente, lo que acelera el procesamiento y mejora el rendimiento del programa. Los aspectos más relevantes al utilizar OpenMP incluyen la gestión de la concurrencia, la asignación de tareas a hilos, y la sincronización de la ejecución.

Es fundamental tener en cuenta que el uso de OpenMP no garantiza necesariamente una mejora en el rendimiento del programa y es necesario evaluar su efectividad en cada caso particular. También es importante considerar otras herramientas y técnicas de programación paralela en conjunto con OpenMP para maximizar el rendimiento y la eficiencia de la simulación [23].

3.2.1 Hilos

En OpenMP, los hilos son una forma de paralelizar la ejecución de un programa en una CPU. Cada hilo representa una unidad independiente de ejecución dentro del programa, que puede ser ejecutada simultáneamente en la CPU. Utilizar hilos en OpenMP permite acelerar la simulación, al dividir la carga de trabajo entre varios núcleos de la CPU. En la actualidad un núcleo de CPU tiene al menos dos hilos, en la figura 3.1 se pueden ver la representación de un CPU con cincuenta núcleos, cada uno con un par de hilos, dando un total de cien hilos en ese CPU.

La cantidad de hilos utilizados en OpenMP se puede ajustar para optimizar el rendimiento del programa. Por defecto, OpenMP utiliza el número de núcleos disponibles en la CPU, pero se puede especificar manualmente la cantidad de hilos a utilizar en función de las necesidades del programa y del hardware disponible. Hay varias formas de elegir la cantidad de hilos para ejecutar un programa, en las simulaciones desarrolladas en este trabajo se utiliza la función `omp_set_num_threads(n)`, siendo `n` el número deseado de hilos, esta función se coloca en el código antes de la sección paralelizada [23].

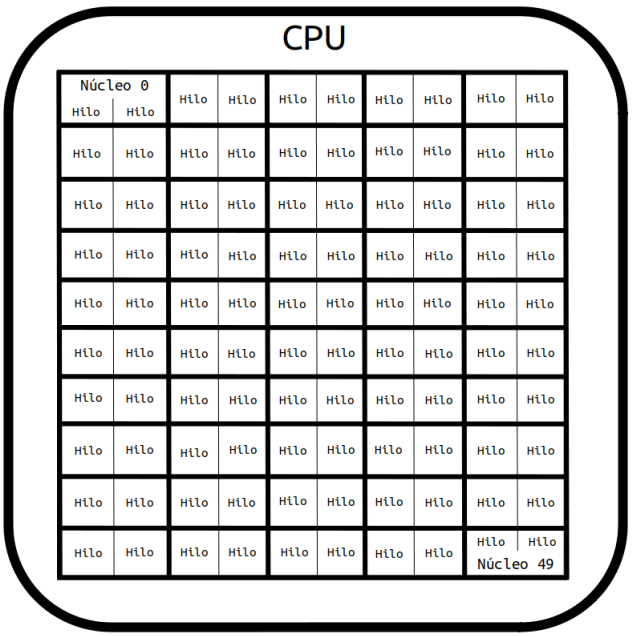

Figura 3.1: Representación de los hilos y núcleos en un CPU. Un CPU puede tener N cantidad de núcleos, y estos a su vez H cantidad de hilos, por lo que en total hay $N * H$ hilos que pueden ser utilizados para paralelizar en un CPU.

Cabe mencionar que los CPUs de la actualidad con más núcleos rondan los sesenta

y cuatro núcleos, con un par de hilos en cada uno dan un total de ciento veintiocho hilos [27]. Sin embargo, con sistemas de múltiples CPUs se pueden llegar a tener cientos de hilos sin necesidad de tener el CPU con más núcleos del mercado.

Para seleccionar la cantidad óptima de hilos a utilizar, es necesario realizar pruebas empíricas y ajustar el número de hilos en función del tamaño del problema, el hardware utilizado y otros factores. El uso de demasiados hilos puede ralentizar el programa debido al trabajo adicional de la gestión de hilos, mientras que el uso de muy pocos hilos puede no aprovechar completamente la capacidad de la CPU.

3.2.2 Directivas de OpenMP

En este trabajo se implementó OpenMP para la paralelización con núcleos de CPUs y se utilizaron los siguientes pragmas:

```
/* Sección de código con OpenMP */
#pragma omp parallel for
for(int i = 0; i < N; i++){
    // Calculo a realizar en paralelo
}
```

donde `N` es el número total de iteraciones que se deben realizar. La directiva `#pragma omp parallel for` crea un equipo de hilos que se distribuyen las iteraciones del bucle de forma equitativa, entonces cada hilo procesa una parte del bucle en paralelo con los demás hilos, lo que aumenta el rendimiento del programa.

```
/* Sección de código con OpenMP para evitar race condition*/
double U = 0.0;

#pragma omp parallel for reduction(+:U)
for(int i = 0; i < N; i++){
    // Calculo a realizar en paralelo
    U += calculaU(i);
}

#pragma omp parallel for
for(int i = 0; i < N; i++){
    // Calculo a realizar en paralelo
    #pragma omp atomic update
    U += calculaU(i);
}
```

La directiva `#pragma omp parallel for reduction(+:U)` es útil para realizar operaciones de reducción, es decir, sumar los valores de una variable `U` que se están calculando en paralelo y almacenando en cada hilo, evitando el race condition [23]. En cambio, la clausula `atomic update` hace que la operación de la siguiente línea se realice de manera segura, es decir, sin perdida de información.

Utilizando los pragmas mencionados fue posible paralelizar con múltiples hilos toda la simulación desarrollada, por lo que implementar los pragmas OpenMP es una de las formas más sencillas de paralelizar un programa.

3.3 OpenACC

OpenACC (Open Accelerator) es un modelo de programación paralela diseñado para aprovechar las capacidades de procesamiento de las unidades de procesamiento gráfico (GPU). Permite a los programadores escribir código en C, C++ o Fortran y añadir directivas de compilador (pragmas) para acelerar las operaciones en la GPU.

Entre las directivas más utilizadas en OpenACC se encuentran `kernels`, `loop` y `data`. La directiva `kernels` se usa para indicar que una sección de código debe ejecutarse en paralelo en la GPU. La directiva `loop` se emplea para paralelizar un bucle y la directiva `data` se maneja para indicar los datos que deben ser transferidos entre la memoria del host y la GPU [17].

Es importante tener en cuenta que como con OpenMP, la aceleración con OpenACC, no siempre conlleva una mejora en el rendimiento, ya que depende de varios factores como el tipo de operación, el tamaño del conjunto de datos y el tipo de hardware utilizado. Por lo tanto, es necesario realizar pruebas de rendimiento para determinar si OpenACC es la opción adecuada para optimizar un determinado código.

3.3.1 GPUs

Las GPUs (unidades de procesamiento gráfico, por sus siglas en inglés) son dispositivos especializados en el procesamiento paralelo de grandes cantidades de datos. A diferencia de las CPUs convencionales, las GPUs cuentan con cientos o incluso miles de núcleos que pueden procesar tareas en paralelo. Esto las convierte en herramientas muy eficientes para el procesamiento de datos masivos en aplicaciones de HPC y gráficos 3D.

OpenACC permite aprovechar la potencia de las GPUs en la programación de aplicaciones paralelas, al utilizar sus directivas los programadores pueden diseñar códigos que aprovechen la arquitectura de la GPU de forma transparente, sin tener que preocu-

parse por los detalles de la programación de bajo nivel. OpenACC también proporciona mecanismos de transferencia de datos entre la memoria principal y la memoria de la GPU, lo que simplifica aún más el desarrollo de aplicaciones para GPU.

Una de las ventajas de OpenACC es que puede utilizarse con diferentes plataformas de hardware, incluyendo GPUs de diferentes fabricantes como NVIDIA, AMD o Intel. Esto significa que los desarrolladores pueden escribir código en OpenACC y luego compilarlo para diferentes plataformas, sin tener que preocuparse por los detalles de la arquitectura subyacente [17].

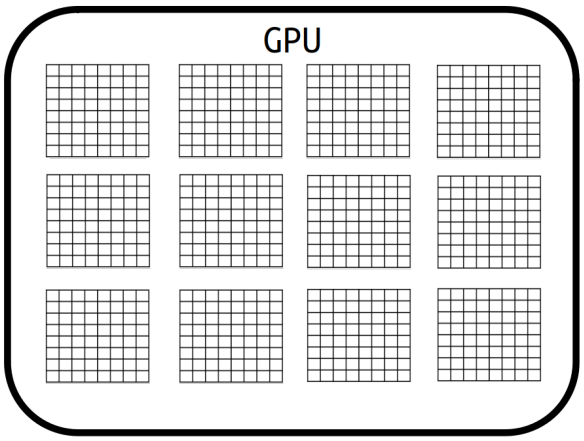

Figura 3.2: Representación de los núcleos en una GPU. Una GPU puede tener miles de núcleos, distribuidos en bloques como los de la figura.

3.3.2 Directivas de OpenACC

Las directivas de OpenACC son muy parecidas a las de OpenMP. En la paralelización con GPUs se le llama *host* al CPU y *device* a la GPU [23]. Las directivas se especifican mediante la sintaxis de preprocesador `#pragma acc` y a cada una se le pueden añadir distintas cláusulas que controlan el modo de actuar de la ejecución paralela. A continuación se presentan las cláusulas que se utilizaron en las diferentes directivas para paralelizar la simulación:

- `copyin(...)`: Copia los datos desde el host al device.
- `copyout(...)`: Copia los datos desde el device al host.
- `present(...)`: Indica que los datos ya se encuentran en el device.
- `delete(...)`: Elimina un objeto del device. El objeto puede ser una variable o un arreglo que se haya pasado previamente con las cláusulas anteriores.

- `async`: Indica que la región paralela se ejecutará de forma asíncrona al host y a otras regiones que tengan la misma cláusula.
- `gang`: Distribuye los bloques de núcleos en grupos (gangs).
- `worker`: Distribuye los núcleos dentro de cada gang en grupos (workers).
- `vector`: Distribuye el trabajo entre los núcleos de un mismo worker.
- `independent`: Indica que las iteraciones del bucle no tienen dependencias entre sí.
- `reduction(...)`: Realiza una operación de reducción en la variable especificada.
- `collapse(...)`: Combina múltiples bucles anidados en uno solo. Recibe como argumento el número de ciclos que se van a combinar, tienen que estar perfectamente anidados (sin operaciones/declaraciones entre ellos).
- `wait`: Sincroniza la ejecución del host y el device, esperando a que acaben los cálculos asíncronos.
- `update`: Actualiza los datos en el host o el device.
- `host_data`: Indica que los datos serán usados en el host y en el device.

Las directivas de OpenACC empleadas en la paralelización de la simulación, mediante GPU, son las siguientes:

- `#pragma acc set device_num(0)`: Selecciona el número del device (GPU) que se va a utilizar para ejecutar el código en una región paralela.
- `#pragma acc enter data copyin(...)`: Transfiere datos desde la memoria del host a la memoria del dispositivo. Los argumentos en paréntesis indican los datos que se van a transferir.
- `#pragma acc parallel loop gang present(...) independent async`: Define que un bucle se debe ejecutar en paralelo en la GPU utilizando múltiples bloques de núcleos.
- `#pragma acc loop vector reduction(...)`: Especifica que un bucle debe ejecutarse en paralelo utilizando vectores en la GPU. Quiere decir que el trabajo se va a distribuir entre los diferentes núcleos de un bloque.
- `#pragma acc host_data use_device(...)`: Aclara que los datos deben estar disponibles tanto en el host como en el dispositivo, lo que permite que la CPU y la GPU compartan los datos.

- `#pragma acc parallel loop collapse(2) reduction(+:...)`: Paraleliza un bucle anidado de `for` con un tamaño de iteración grande.
- `#pragma acc parallel loop reduction(min:m) reduction(max:M)`: Paraleliza un bucle con dos operaciones de reducción en las variables `m` y `M`, guardando el valor mínimo y máximo en estas respectivamente.
- `#pragma acc wait`: Espera a que todas las operaciones en la GPU se completen antes de continuar con la ejecución en la CPU.
- `#pragma acc exit data copyout(...)`: Transfiere datos desde la memoria del device a la memoria del host.
- `#pragma acc atomic update`: Garantiza que las operaciones de actualización en un bucle se realicen de forma atómica, lo que evita conflictos de memoria y garantiza la integridad de los datos. Es una alternativa costosa en tiempo a la cláusula `reduction()`.
- `#pragma acc update host(...)`: Transfiere datos desde la memoria del device a la memoria del host.

3.3.3 Números pseudoaleatorios en OpenACC

Los números aleatorios de la simulación se requieren para asignar las posiciones, las valencias y el desplazamiento aleatorio de los iones debido a la dinámica browniana. Para este último, se necesitan generar millones de números aleatorios.

La generación de números pseudoaleatorios de la biblioteca `<random>` se ejecuta en la CPU. La memoria de la CPU y de la GPU no son la misma, entonces si se usa la biblioteca `<random>` para generar un número aleatorio que se va a operar en la GPU, cada número aleatorio debe transferirse a la GPU, creando un cuello de botella.

OpenACC no tiene una forma de generar números aleatorios en GPU, pero CUDA sí utilizando la biblioteca `<curand.h>`. Con OpenACC se pueden utilizar las funciones de CUDA con el pragma `#pragma acc host_data use_device(...)` [28].

Un ejemplo para generar números pseudoaleatorios, de distribución normal con media cero y desviación estándar uno, en GPU con OpenACC, es el siguiente:

```
/* Sección de código con OpenACC */
int tam = N * 3;
double *restrict G = new double[tam];
```

```
#pragma acc enter data copyin(G[:tam], x[:N], y[:N], z[:N])

random_device rd{};
cudaStream_t stream;
curandGenerator_t gen;
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);

#pragma acc host_data use_device(G)
{
   stream = (cudaStream_t) acc_get_cuda_stream(acc_async_sync);
}

curandSetStream(gen, stream);
curandSetPseudoRandomGeneratorSeed(gen, rd());

#pragma acc host_data use_device(G)
{
   curandGenerateNormalDouble(gen, G, tam, 0.0, 1.0);
}

#pragma acc parallel loop present(G[:tam], x[:N], y[:N], z[:N])
for(int i = 0; i < N; i++){
    x[i] += G[3*i];
    y[i] += G[3*i+1];
    z[i] += G[3*i+2];
}
```

La primera línea de código define el tamaño del arreglo `G` como `N*3`. Luego, se utiliza la función `new` para asignar memoria dinámica al arreglo `G` y se utiliza la palabra clave `restrict` para indicar al compilador que el arreglo `G` es el único puntero que puede apuntar a los datos en esa memoria.

A continuación, se utiliza la directiva `#pragma acc enter data` para copiar los datos de `G`, `x`, `y` y `z` desde el host (CPU) al device (GPU). La sintaxis `G[:tam]` se utiliza para indicar que se copiará el arreglo `G` de longitud `tam`.

Enseguida, se crea un objeto `random_device` de la biblioteca estándar de C++ para generar una semilla aleatoria para el generador de números aleatorios `curand`. Se utiliza la función `cudaStream_t` para crear un objeto de flujo CUDA (CUDA stream object) para la ejecución asincrónica, y se utiliza la función `curandCreateGenerator` para crear un objeto generador de números aleatorios `curand` con el generador pseudoaleatorio por defecto.

Luego, se utiliza la directiva `#pragma acc host_data` para asegurarse de que `G` se encuentre en el device, y se obtiene el flujo CUDA de la directiva `acc_get_cuda_stream`. Este flujo se utiliza para configurar el generador de números aleatorios de la biblioteca `<curand.h>` utilizando las funciones `curandSetStream` y `curandSetPseudoRandomGeneratorSeed`.

Posteriormente, se utiliza la directiva `#pragma acc host_data` para señalar que se va a usar el arreglo `G` que se encuentra en el device, y se utiliza la función `curandGenerateNormalDouble` para generar números aleatorios de distribución normal con media cero y desviación estándar uno. Los números aleatorios se almacenan en el arreglo `G` del device.

Finalmente, se utiliza la directiva `#pragma acc parallel loop` para iterar sobre los arreglos `x`, `y` y `z`, para sumar a cada uno un número pseudoaleatorio del arreglo `G`. La cláusula `present(G[:tam], x[:N], y[:N], z[:N])` se utiliza para indicar que los arreglos `G`, `x`, `y` y `z` se encuentran en el dispositivo y están disponibles para su ejecución en paralelo [17].

3.4 Shell de Unix

El `shell` de Unix es una interfaz de línea de comandos que permite a los usuarios interactuar con un sistema operativo Unix o Unix-like. A través del shell, los usuarios pueden ejecutar comandos y programas, manipular archivos y directorios, configurar el sistema y automatizar tareas.

El shell más común en los sistemas Unix es Bash (Bourne-again shell), que es una versión mejorada del shell original escrito por Steve Bourne. Bash es una herramienta poderosa y versátil que se utiliza en una gran variedad de sistemas Unix, incluyendo Linux y macOS [29].

Además de Bash, existen otros shells de Unix disponibles, cada uno con sus propias características y funcionalidades únicas. Algunos ejemplos incluyen Korn shell (ksh), C shell (csh) y Z shell (zsh).

En general, el uso del shell de Unix puede ser intimidante para los nuevos usuarios debido a la necesidad de memorizar comandos y opciones. Sin embargo, una vez que se adquiere familiaridad con el shell, se puede convertir en una herramienta muy eficiente y poderosa para la administración del sistema y la automatización de tareas repetitivas.

Las simulaciones desarrolladas en este trabajo se corrieron (ejecutaron) en servidores con distribuciones Linux, por lo que el shell utilizado es Bash.

3.4.1 SSH

SSH (Secure Shell) es un protocolo de red que permite a los usuarios conectarse de forma segura a un servidor remoto a través de una conexión cifrada. La conexión SSH permite al usuario acceder al shell de Unix del servidor remoto y ejecutar comandos como si estuviera físicamente conectado al servidor.

Además de proporcionar una conexión segura, SSH también permite la transferencia de archivos y el uso de aplicaciones gráficas remotas. SSH se utiliza comúnmente para administrar servidores remotos, ya que proporciona una forma segura de conectarse y ejecutar comandos sin necesidad de estar físicamente presente en el servidor [30].

La conexión SSH es la forma en la que se accedió a los servidores de la universidad para ejecutar los programas en CPUs de múltiples hilos y en diferentes GPUs.

3.4.2 Banderas

Las banderas, también conocidas como opciones o argumentos de línea de comando, son elementos que se utilizan junto con los comandos del shell para modificar su comportamiento. En la compilación de un programa, las banderas se utilizan para indicarle al compilador cómo debe tratar el código fuente y generar el ejecutable.

Las banderas se pueden utilizar para especificar la arquitectura de la máquina en la que se compila el programa, el nivel de optimización deseado, el tipo de salida deseado, el conjunto de instrucciones a utilizar, entre otros.

Es importante tener en cuenta que las banderas pueden variar entre los diferentes compiladores y sistemas operativos. Algunas banderas también pueden tener efectos no deseados en el rendimiento o la estabilidad del programa, por lo que se recomienda utilizarlas con precaución y solo cuando sea necesario.

Con el compilador de Intel icpc, se utilizaron las siguientes banderas [31]:

- **-g**: Genera información de depuración para el código compilado. Esto permite a los programadores depurar el programa de forma más fácil.
- **-O3**: Activa la optimización de nivel 3. Esta bandera optimiza el código para que se ejecute más rápido, aunque puede hacer que el tiempo de compilación sea más largo.
- **-restrict**: Indica al compilador que ciertos punteros no se superponen. Esto puede permitir al compilador generar código más eficiente.

- **-qopenmp**: Activa la compatibilidad con OpenMP (Open Multi-Processing).
- **-qopenmp-simd**: Habilita el soporte para SIMD (Single Instruction, Multiple Data). SIMD permite a los procesadores ejecutar una misma operación en múltiples datos de manera simultánea.
- **-qopt-zmm-usage=high**: Esta bandera habilita el uso máximo de los registros ZMM, que son un tipo especial de registro de procesador que permite procesar grandes cantidades de datos en paralelo. El uso de estos registros puede mejorar significativamente el rendimiento de las aplicaciones que involucran operaciones de punto flotante.
- **-qopt-subscript-in-range**: Habilita la verificación en tiempo de compilación de que los índices de los arreglos están dentro de sus límites.

Y para el compilador de NVIDIA nvc++ se utilizaron la siguientes banderas [26]:

- **-g**: Genera información de depuración para el código compilado.
- **-O3**: Habilita las optimizaciones de nivel 3, lo que puede mejorar significativamente el rendimiento del programa.
- **-acc=gpu**: Habilita el uso de OpenACC para acelerar el procesamiento en la GPU.
- **-Mpreprocess**: Indica al compilador que debe realizar la preprocesamiento del código fuente antes de la compilación.
- **-Mcuda**: Habilita la generación de código CUDA para la ejecución en GPU NVIDIA.
- **-lcurand**: Enlaza el programa con la biblioteca cuRAND de NVIDIA, que proporciona funciones para la generación de números aleatorios en la GPU.
- **-alias=ansi**: Establece que los punteros no pueden apuntar a la misma ubicación de memoria en el código. Al seguir esta regla, el compilador puede generar un código más eficiente y seguro.
- **-Minfo=accel**: Habilita la generación de mensajes informativos relacionados con la aceleración en la GPU.

Estas banderas se escriben cada que se va a compilar el programa, si el programa tiene un error o se modifica algo, se tiene que volver a compilar. El volver a compilar el programa cuando se están haciendo las pruebas es una tarea repetitiva, la cual se puede automatizar con un *script*.

3.4.3 Scripts

Un *script* es un archivo de texto que contiene un conjunto de comandos que se ejecutan en el orden en que aparecen. Los *scripts* se utilizan comúnmente en sistemas Unix para automatizar tareas o para realizar una serie de acciones repetitivas [29].

Para los programas desarrollados se utilizó un *script* para compilar el programa, borrar los archivos temporales generados durante la compilación y ejecutar el programa en primer o segundo plano.

La ejecución de un *script* se realiza mediante un intérprete de comandos, como Bash. Una vez que se ha creado el *script*, se puede ejecutar simplemente escribiendo `./nombre_script` en la línea de comandos.

El uso de un *script* puede ahorrar tiempo y reducir errores al automatizar el proceso, evitando la necesidad de repetir una serie de comandos manualmente cada vez que se quiere compilar y ejecutar un programa.

3.4.4 Nvproof

Nvproof es una herramienta de perfilado de rendimiento que permite analizar el comportamiento de una aplicación en tiempo de ejecución. La herramienta se ejecuta en segundo plano junto con la aplicación para registrar los tiempos de ejecución y los eventos del sistema, como los accesos a la memoria y la utilización de la GPU.

Una vez que se ha completado la ejecución del programa, los datos recopilados por Nvproof se pueden analizar para identificar cuellos de botella y áreas de mejora en la implementación del programa. Esto puede ayudar a los desarrolladores a optimizar su código y mejorar el rendimiento de la aplicación.

Nvproof ofrece varias opciones de visualización y análisis de los datos, como gráficos de línea y tablas de resumen. También se pueden generar informes detallados que incluyen estadísticas como el tiempo de ejecución total, el número de accesos a la memoria y la utilización de la GPU [32].

3.5 Servidores

En esta sección se encuentran las especificaciones de los CPUs y GPUs de los servidores utilizados para correr los programas desarrollados en este trabajo.

La primera tabla presenta una comparación de los CPUs instalados en los servidores. En la tabla se muestra la frecuencia base, la frecuencia turbo y el número de núcleos de cada CPU. La frecuencia base indica la velocidad de reloj nominal del CPU, mientras que la frecuencia turbo es la velocidad máxima a la que puede operar el CPU cuando se requiere una mayor potencia de procesamiento. El número de núcleos indica la cantidad de unidades de procesamiento independientes que tiene el CPU, hay que recordar que cada núcleo de CPU contiene dos hilos, que son la cantidad de procesos que pueden ser manejados simultáneamente por el CPU. Con esta información, se puede determinar el CPU más adecuado para cada programa.

Servidor	**CPU**	**Frecuencia base/turbo**	**Núcleos**
Maxwell	AMD Ryzen™ 5 2400G	3.60 GHz / 3.90 GHz	4
Boltzmann	Intel® Core™ i5-10400F	2.90 GHz / 4.30 GHz	6
Mangore	Intel® Core™ i7-8700	3.20 GHz / 4.60 GHz	6
FJEstrada	Intel® Xeon® Gold 5220R	2.20 GHz / 4.0 GHz	24
Ampere	Intel® Xeon® Gold 5320 (x2)	2.20 GHz / 3.40 GHz	26 (x2)

Tabla 3.1: Comparación de los CPUs de los servidores utilizados.

Los CPUs con seis núcleos o menos son procesadores de escritorio, están diseñados para tareas como juegos y edición de video. Por otro lado, los CPUs Xeon® son procesadores de servidor, diseñados para una alta capacidad de procesamiento. El servidor Ampere tiene instalados dos CPUs de veintiséis núcleos, por lo que este es el servidor utilizado con más hilos.

La siguiente tabla muestra las características de las GPUs instaladas en los servidores. Todas son de la marca NVIDIA, conocidas por su potencia y capacidad de procesamiento. El rendimiento se mide en teraflops (TFLOPs), que representa la cantidad de operaciones de punto flotante por segundo que una GPU puede realizar, en la tabla se encuentra el rendimiento para operaciones de precisión doble. Los núcleos CUDA, son los núcleos de las GPUs de NVIDIA, los mismos de que se vieron en la subsección 3.3.1.

Servidor	**GPU**	**Rendimiento de doble precisión**	**Núcleos CUDA**
Maxwell	GeForce GTX 1050	0.058 TFLOPs	768
Boltzmann	GeForce GTX 1650	0.093 TFLOPs	1024
Mangore	Titan Xp	0.379 TFLOPs	3840
	Tesla K40c	1.682 TFLOPs	2880
FJEstrada	Tesla K80 (x6)	1.371 TFLOPs (x6)	2496 (x6)
	Tesla K40m	1.682 TFLOPs	2880
Ampere	A100	9.742 TFLOPs	6912

Tabla 3.2: Comparación de las GPUs de los servidores utilizados [33].

Los servidores Mangore y FJEstrada tienen múltiples GPUs instaladas que pueden trabajar de manera independiente y de forma paralela, lo que permite correr múltiples programas a la vez. Entonces, se pueden utilizar múltiples GPUs de gama alta de tecnologías anteriores para realizar un trabajo, en el mismo tiempo o mejor que con una GPU de nueva generación.

Las GPUs de los servidores se utilizan en diversas áreas y tienen aplicaciones específicas. La GeForce GTX 1050 y la GeForce GTX 1650 son GPUs orientadas al rendimiento en videojuegos, al igual que la Titan Xp, pero esta es una GPU de gama alta, que brinda un rendimiento excepcional en juegos y tareas de computación intensiva. En cambio, las GPUs Tesla K40c, Tesla K80 y Tesla K40m se utilizan en aplicaciones de alto rendimiento y cálculos científicos. La Ampere A100 es una GPU de última generación especialmente diseñada para inteligencia artificial, aprendizaje automático y análisis de datos a gran escala.

CAPÍTULO 4

Desarrollo de la simulación

4.1 Macroion con carga discreta

Los sistemas en la naturaleza tienden a evolucionar hacia un estado de energía mínima, buscando un equilibrio en el cual las fuerzas internas y externas se compensan. Esta búsqueda de equilibrio se refleja en la formación de estructuras estables, como una gota de agua adoptando una forma esférica. La esfericidad de la gota minimiza la energía superficial al tener la menor área posible para un volumen dado. Por esta razón y por la simetría del sistema, el macroion con carga discreta de la simulación tiene forma esférica.

Para el macroion con carga discreta hay que determinar los sitios de carga en la superficie de una esfera con energía potencial mínima, una forma de hacerlo es mediante el problema de Thomson [34]. El objetivo de esta simulación es encontrar la configuración con energía potencial mínima para N^T cargas en la superficie de una esfera, configuración que se utiliza en las simulaciones de la siguiente sección.

El tiempo de cálculo computacional para encontrar la configuración con energía mínima aumenta conforme N^T crece. Para hacer las simulaciones con un macroion de carga discreta, tal que el número de cargas que componen el macroion varía entre una simulación y otra, es necesario encontrar una configuración mínima para cada simulación, lo que puede llevar un gran tiempo de espera si N^T es muy grande.

Una computadora siempre puede tener más de un hilo que puede ser aprovechado en los cálculos de un programa, pero no siempre se tiene una GPU disponible que puede usar miles de núcleos a la vez, por esto se decidió paralelizar la simulación haciendo otras dos versiones de este programa, una utilizando los hilos de un CPU y otra los núcleos de una GPU.

4.1.1 Descripción de la simulación

El programa consiste en simular la dinámica browniana para N^T cargas distribuidas en la superficie de una esfera, con el fin de encontrar la configuración con la mínima energía potencial. Ya que el movimiento browniano es de naturaleza aleatoria, la energía de la configuración con dinámica browniana no es mínima, por lo que es necesario aplicar un método adicional para encontrar la configuración con energía mínima. El método aplicado a la configuración dada por la dinámica browniana es el de descenso de gradiente, visto en la sección 2.4.

De la ecuación de movimiento browniano (ec. 2.4), utilizando unidades reducidas, se tiene que:

$$\mathbf{r_i^*}(t+\Delta t) = \frac{\mathbf{r_i}(t)}{10^{-10}\text{ m}} + \frac{\sqrt{2D\Delta t}\cdot\mathbf{G}(0,1)}{10^{-10}\text{ m}} + \frac{D}{kT\cdot 10^{-10}\text{ m}}\cdot\mathbf{F_i^*}(t)\cdot\Delta t \tag{4.1}$$

donde:

- $\mathbf{r_i}(t)$ es la posición de la partícula i en el tiempo t.
- D es el coeficiente de difusión del medio.
- Δt es el tamaño de paso.
- $\mathbf{G}(0,1)$ es un número aleatorio de una distribución normal con media cero y desviación estándar uno.
- k es la constante de Boltzmann.
- T K es la temperatura absoluta del sistema.
- $\mathbf{F_i^*}(t)$ es la fuerza electrostática total que siente la partícula i en el tiempo t en unidades reducidas.

De la sección 2.8, la fuerza electrostática que siente una partícula i debido a las demás partículas, en unidades reducidas es:

$$\mathbf{F}_i^{\text{el}*} = \frac{l_B^*}{10^{-10}\text{ m}}\sum_{j=1,j\neq i}^{N^T}\frac{z_i z_j}{r_{ij}^{*2}}\mathbf{\hat{r}_{ij}^*} \tag{4.2}$$

En el problema de Thomson, $z_i = z_j = 1$, por lo que todas las fuerzas son repulsivas a la hora de buscar la configuración con energía potencial mínima.

Para medir la energía potencial de las configuración se utiliza el potencial simple:

$$U^{\mathrm{T}}(r_{ij}) = \frac{1}{2} \sum_{i=1, i\neq j}^{N^T} \sum_{j=1}^{N^T} \frac{1}{r_{ij}} \tag{4.3}$$

Con este potencial y usando una esfera de radio uno, se pueden comparar los resultados con los de otros autores que han trabajado en el problema de Thomson, como los de David J. Wales y Sidika Ulker [35].

Con la dinámica browniana se logra un error porcentual menor a 1% con respecto a estos resultados, con un número de pasos menor a 10^4 para las N^T utilizadas en las simulaciones.

Para aplicar el método de descenso de gradiente, se utiliza la ecuación 2.15, quedando de la siguiente forma:

$$\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha \vec{\nabla} U^{\mathrm{T}}(r_n) \tag{4.4}$$

donde $\mathbf{r}_{n+1}$ son las nuevas posiciones de los sitios de carga en la superficie de la esfera, $\mathbf{r}_n$ son las posiciones anteriores, $\alpha = 0.5$ es el tamaño de paso y, de acuerdo a la ecuación 4.3:

$$\vec{\nabla} U^{\mathrm{T}}(r_{ij}) = -\frac{1}{2} \sum_{i=1, i\neq j}^{N^T} \sum_{j=1}^{N^T} \frac{1}{r_{ij}^2} \tag{4.5}$$

Con este método, se logra un error porcentual que ronda 10^{-3}% con 3×10^4 iteraciones para cualquier N^T. Para $N^T < 100$, este error puede ser menor a 10^{-10}% o nulo con solo cientos de iteraciones.

El programa se divide en tres archivos, el *main*, el *header* (cabecera) y el programa que calcula las posiciones, llamado *Macroion*. El *main* contiene las constantes y variables del programa, además de llamar a la función principal de *Macroion*. En el *header* se encuentran las bibliotecas y la declaración de las funciones que se implementan y llaman en los otros archivos. El archivo *Macroion* contiene una función principal que llama a las otras funciones que se encuentran en este, para calcular la configuración de partículas con energía potencial mínima en la superficie de una esfera de radio R_m, mediante dinámica browniana y el método de descenso de gradiente.

4.1.2 Implementación serial

Si se quiere paralelizar un programa es porque se necesitan tiempos de ejecución más rápidos, por lo que se puede empezar a optimizar desde la versión serial, utilizando algoritmos eficientes, evitando cuellos de botella, minimizando operaciones innecesarias y garantizando una buena gestión de memoria.

Además, se puede dividir el programa en funciones o subrutinas coherentes y bien definidas para que a la hora de paralelizar le programa sea más sencillo identificar los bloques de código a paralelizar.

El archivo *Macroion* está dividido en tres partes principales que son: *Funciones iniciales*, *Dinámica browniana* y *Método de descenso de gradiente*.

En *Funciones iniciales* se asigna la memoria para los arreglos de las posiciones y de las fuerzas de tamaño N^T; se formatean los arreglos de fuerzas; se asignan las posiciones iniciales de forma aleatoria en la superficie de la esfera de tamaño R_m; se imprime una cabecera para que el usuario sepa la configuración que se está ejecutando; y, por último, se inicializan los números pseudoaleatorios.

El bloque *Dinámica browniana* calcula la fuerza electrostática neta que actúa en cada partícula y utiliza la ecuación de movimiento de la dinámica browniana (ec. 4.1) para obtener las nuevas posiciones. Además calcula e imprime la energía potencial U^T (ec. 4.3) cada mil iteraciones.

En *Método de descenso de gradiente* primero se asigna la memoria para los arreglos que guardan las posiciones anteriores y los gradientes de potencial; se formatean los arreglos de los gradientes; se calcula el gradiente de potencial de las posiciones que salieron de la dinámica browniana; y después empieza el método.

Algoritmo: Problema de Thomson Serial — Parte 1

Resultado: Configuración de energía potencial mínima para N^T partículas en la superficie de una esfera de radio R_m.

Entrada: N^T, l_B, Δt, D

```
// Funciones iniciales //
Asigna_memoria(x, y, z, Fx, Fy, Fz) ;        // Posiciones y fuerzas
Inicializa_arreglos(N^T, Fx, Fy, Fz) ;       // Formatea los arreglos
Posiciones_iniciales(N^T, x, y, z) ;         // Posiciones aleatorias
Proyecta_a_esfera(N^T, x, y, z, R_m);
Imprime_cabecera(N^T, R_m, Δt);
Inicializa_números_pseudoaleatorios();
```

Algoritmo: Problema de Thomson Serial — Parte 2

```
// Dinámica browniana //
for iteracion ← 1 to 10^4 do
    for i ← 1 to N^T do
        for j ← 1 to N^T do
            if i ≠ j then
                Calcula la fuerza electrostática (Fx, Fy, Fz) que actúa sobre
                 cada partícula (i) debido a las demás partículas (j) (ec. 2.28).
            end
        end
    end
    for i ← 1 to N^T do
        Calcula las nuevas posiciones (x, y, z) para cada partícula (i) con la
         ecuación de movimiento de la dinámica browniana (ec. 4.1).
    end
    Proyecta_a_esfera(N^T, x, y, z, R_m);
    if iteracion % 1000 == 0 then
        U^T ← Calcula_energía_potencial(N^T, x, y, z);
        Imprime_energía_potencial(U^T);
    end
end
Libera_memoria(Fx, Fy, Fz)
```

La subrutina *Proyecta_a_esfera(N^T, x, y, z, R_m)* coloca nuevamente las partículas en la superficie de la esfera de radio R_m, ya que no hay forma de que al moverse estas se mantengan en la superficie de la esfera.

El siguiente algoritmo es de la función *Calcula_energía_potencial(N^T, x, y, z)*:

Algoritmo: Calculo de la energía potencial (U^T) de forma serial

```
for i ← 1 to N^T do
    for j ← 1 to N^T do
        if i ≠ j then
            Calcula la energía potencial (U^T) que hay entre cada partícula (i)
             con las demás partículas (j) (ec. 4.3).
        end
    end
end
return U^T
```

Algoritmo: Problema de Thomson Serial — Parte 3

```
// Método de descenso de gradiente //
Asigna_memoria(xa, ya, za, Gx, Gy, Gz);
Inicializa_arreglos(N^T, Gx, Gy, Gz);
Calcula_gradiente_potencial(N^T, x, y, z, Gx, Gy, Gz);
α ← 0.5 ;                                              // Tamaño de paso
Ua ← 0.0 ;                                             // Potencial anterior
contador ← 0 ;                                         // Contador de iteraciones
U^T ← Calcula_energía_potencial(N^T, x, y, z) ;        // Potencial inicial
while (U^T − Ua) > 10^−10 & contador < 3 × 10^4 do
    for i ← 1 to N^T do
        Guarda_posiciones_anteriores(N^T, x, y, z, xa, ya, za);
        Calcula las nuevas posiciones (x, y, z) para cada partícula (i) con la
         ecuación de descenso de gradiente (ec. 4.4).
    end
    Proyecta_a_esfera(N^T, x, y, z, R_m);
    Ua ← U^T ;                                         // Guarda el potencial
    U^T ← Calcula_energía_potencial(N^T, x, y, z) ;    // Potencial nuevo
    if U^T > Ua then
        α ← α × 0.5 ;                                  // Disminuye el tamaño de paso
        for i ← 1 to N^T do
            Regresa a las posiciones anteriores.
        end
    end
    Calcula_gradiente_potencial(N^T, x, y, z, Gx, Gy, Gz);
    if contador % 1000 == 0 then
        U^T ← Calcula_energía_potencial(N^T, x, y, z);
        Imprime_energía_potencial(U^T);
    end
    contador ← contador + 1;
end
Libera_memoria(N^T, xa, ya, za, Gx, Gy, Gz);
Guarda_posiciones(N^T, x, y, z, contador, U^T);
Libera_memoria(x, y, z);
```

Al llamar a la función *Calcula_gradiente_potencial*(N^T, x, y, z, Gx, Gy, Gz), esta calcula el gradiente del potencial de cada partícula y los guarda en los arreglos Gx, Gy,

Gz, utilizando la ec. 4.5.

El método de descenso de gradiente se detiene cuando la diferencia entre el potencial nuevo y el potencial anterior es menor que 10^{-10} o si el contador llega a las 3×10^4 iteraciones.

La subrutina del final llamada *Guarda_posiciones(N^T, x, y, z, contador, U^T)* crea un archivo de texto que contiene las posiciones de las partículas, la energía potencial y el número de iteraciones del método de descenso de gradiente.

4.1.3 Implementación con OpenMP

Con la estructura de la versión serial, es bastante simple implementar OpenMP, basta con agregar la directiva `#pragma omp parallel for` antes de cada ciclo `for` externo que va de 1 hasta N^T, por ejemplo:

Algoritmo: Ciclos `for` paralelizados con OpenMP

```
#pragma omp parallel for
for i ← 1 to N^T do
    for j ← 1 to N^T do
        if i ≠ j then
            Calcula la fuerza electrostática neta que siente la partícula i debido
              a las partículas j, de forma paralela.
        end
    end
end
#pragma omp parallel for
for i ← 1 to N^T do
    Calcula las nuevas posiciones para cada partícula (i), de forma paralela.
end
```

El número de hilos que se van a repartir los cálculos de los ciclos paralelizados, se especifica en el archivo *main*, con la línea "`omp_set_num_threads(n);`", donde n es el número de hilos.

Cuando se calcula U^T de forma paralela, varios hilos intentan actualizar la variable (U^T) al mismo tiempo, esto es *race condition*, por lo que es necesario utilizar una directiva o clausula para evitarlo. En este caso se decidió utilizar la clausula `reduction(+:`U^T`)`, ya que la directiva `atomic` puede llegar a ser un cuello de botella.

El algoritmo para calcular U^T de forma paralela con OpenMP y sin *race condition*,

es el siguiente:

Algoritmo: Calculo de la energía potencial (U^T) en paralelo con OpenMP

#pragma omp parallel for reduction(+:U^T)
for $i \leftarrow 1$ **to** N^T **do**
  for $j \leftarrow 1$ **to** N^T **do**
    if $i \neq j$ **then**
      Calcula la energía potencial (U^T) que hay entre cada partícula (i) con las demás partículas (j), de forma paralela con múltiples hilos.
    end
  end
end
return U^T

4.1.4 Implementación con OpenACC

A diferencia de OpenMP, con OpenACC hay que usar más directivas para evitar el *race condition* en diferentes partes del código y no solo en el calculo de U^T.

Algoritmo: Problema de Thomson paralelizado con OpenACC — Parte 1

Resultado: Configuración de energía potencial mínima para N^T partículas en la superficie de una esfera de radio R_m.

Entrada: N^T, l_B, Δt, D

```
// Funciones iniciales //
```

Asigna_memoria(x, y, z, Fx, Fy, Fz) ; `// Posiciones y fuerzas`
Inicializa_arreglos(N^T, Fx, Fy, Fz) ; `// Formatea los arreglos`
Posiciones_iniciales(N^T, x, y, z) ; `// Posiciones aleatorias`
Proyecta_a_esfera(N^T, x, y, z, R_m);
Imprime_cabecera(N^T, R_m, Δt);

Inicializa los números pseudoaleatorios en GPU para OpenACC (sec. 3.3.3).

#pragma acc enter data copyin(x[:N^T], y[:N^T], z[:N^T], Fx[:N^T], Fy[:N^T], N^T, R_m, Δt, ...)

Los tres puntos representan arreglos y variables presentes en la simulación.

De la primera parte, lo que cambia con respecto a la versión serial es la inicialización de los números pseudoaleatorios y la copia de arreglos y variables a la GPU.

Algoritmo: Problema de Thomson paralelizado con OpenACC — Parte 2

```
// Dinámica browniana //
for iteracion ← 1 to 10^4 do
    #pragma acc parallel loop gang present(...) independent
    for i ← 1 to N^T do
        // Variables auxiliares //
        Fxx ← 0.0;
        Fyy ← 0.0;
        Fzz ← 0.0;
        #pragma acc loop vector reduction(+:Fxx, Fyy, Fzz)
        for j ← 1 to N^T do
            if i ≠ j then
                Calcula la fuerza electrostática (Fxx, Fyy, Fzz) que actúa sobre
                 cada partícula (i) debido a las demás partículas (j) (ec. 2.28).
            end
        end
        Fx[i] ← Fxx;
        Fy[i] ← Fyy;
        Fz[i] ← Fzz;
    end

    #pragma acc host_data use_device(...)
    {
        Genera un conjunto de números aleatorios de distribución normal con
         media cero y desviación estándar uno, utilizando la GPU.
    }

    #pragma acc parallel loop present(...) independent
    for i ← 1 to N^T do
        Calcula las nuevas posiciones (x, y, z) para cada partícula (i) con la
         ecuación de movimiento de la dinámica browniana (ec. 4.1).
        Proyecta_a_esfera(N^T, x, y, z, R_m);
    end
    if iteracion % 1000 == 0 then
        Calcula U^T de forma paralela con OpenACC.
        Imprime_energía_potencial(U^T);
    end
end
#pragma acc exit data delete(Fx[:N^T], Fy[:N^T], Fz[:N^T], ...)
Libera_memoria(Fx, Fy, Fz)
```

Cada sección de la segunda parte está paralelizada, evitando la transferencia de datos entre la CPU y la GPU cuando se están haciendo los cálculos.

Las variables auxiliares se utilizan para guardar el valor de la fuerza neta en la partícula i, valor que es calculado en paralelo. Para evitar *race condition* se utiliza la clausula `reduction(+:...)`, y al final del ciclo se pasan a los arreglos de fuerzas.

A continuación, se presenta la paralelización con OpenACC del calculo de la energía potencial (U^T) y del gradiente del potencial ($\vec{\nabla}U^T$):

Algoritmo: Calculo de la energía potencial en paralelo con OpenACC

#pragma acc parallel loop collapse(2) reduction(+:U^T) present(...)
 independent
for $i \leftarrow 1$ **to** N^T **do**
  for $j \leftarrow 1$ **to** N^T **do**
    if $i \neq j$ **then**
      Calcula la energía potencial (U^T) que hay entre cada partícula (i)
       con las demás partículas (j), de forma paralela en GPU.
    end
  end
end

Algoritmo: Calculo del gradiente del potencial en paralelo con OpenACC

#pragma acc parallel loop gang present(...) independent
for $i \leftarrow 1$ **to** N^T **do**
  `// Variables auxiliares //`
  $Gxx \leftarrow 0.0$;
  $Gyy \leftarrow 0.0$;
  $Gzz \leftarrow 0.0$;
  #pragma acc loop vector reduction(+:Gxx, Gyy, Gzz)
  for $j \leftarrow 1$ **to** N^T **do**
    if $i \neq j$ **then**
      Calcula el gradiente del potencial (Gxx, Gyy, Gzz) de cada
       partícula (i) con las demás partículas (j) (ec. 4.5).
    end
  end
  $Gx[i] \leftarrow Gxx$;
  $Gy[i] \leftarrow Gyy$;
  $Gz[i] \leftarrow Gzz$;
end

Al igual que en la parte dos, en el ciclo principal de la siguiente sección todo está paralelizado con OpenACC, evitando cualquier transferencia entre la GPU y la CPU.

Algoritmo: Problema de Thomson paralelizado con OpenACC — Parte 3

```
// Método de descenso de gradiente //
Inicializa_método() ;                          // Igual que la versión serial
for contador ← 1 to contador < 3 × 10^4 do
    #pragma acc parallel loop present(...) independent
    for i ← 1 to N^T do
        Guarda_posiciones_anteriores(N^T, x, y, z, xa, ya, za);
        Calcula las nuevas posiciones (x, y, z) para cada partícula (i) con la
         ecuación de descenso de gradiente (ec. 4.4).
        Proyecta_a_esfera(N^T, x, y, z, R_m);
        Ua ← U^T ;                                  // Guarda el potencial
        U^T ← 0.0 ;                                // Reinicia la variable
    end
    Calcula_energía_potencial(N^T, x, y, z);
    #pragma acc parallel present(...) independent
    {
        if U^T > Ua then
            α ← α × 0.5 ;                  // Disminuye el tamaño de paso
            for i ← 1 to N^T do
                Regresa a las posiciones anteriores.
            end
        end
    }
    Calcula_gradiente_potencial(N^T, x, y, z, Gx, Gy, Gz);
    if contador % 1000 == 0 then
        U^T ← Calcula_energía_potencial(N^T, x, y, z);
        Imprime_energía_potencial(U^T);
    end
end
#pragma acc data copyout(x[:N^T], y[:N^T], z[:N^T])
#pragma acc exit data delete(x[:N^T], y[:N^T], z[:N^T], ...)
Guarda_posiciones(N^T, x, y, z, contador, U^T);
Libera_memoria(x, y, z, ...);
```

4.2 Electrolito alrededor de un macroion con carga discreta

Esta sección presenta la implementación de la simulación de dinámica browniana de un electrolito confinado alrededor de un macroion con carga discreta.

Se consideran dos casos en particular, el primero es cuando la carga del macroion se divide uniformemente en cada sitio de este y el segundo es cuando el macroion tiene una carga neta compuesta por sitios de carga positiva y negativa. En cada caso, se varía el número de sitios de carga (N^T) que componen el macroion.

El objetivo es analizar el comportamiento del electrolito cuando N^T varía para cada caso, ya que no es una reacción trivial. Para analizar el comportamiento del electrolito se utiliza la densidad de carga por unidad de volumen de iones positivos y negativos, la carga integrada y el potencial eléctrico. Estos se calculan cuando el sistema se encuentra en equilibrio, es decir, cuando la energía potencial electrostática por partícula deja de bajar (ec. 2.10).

4.2.1 Descripción de la simulación

El programa consiste en simular la dinámica browniana de un electrolito confinado alrededor de una macroion con carga discreta, las fuerzas que siente el electrolito son la electrostática y la de núcleo repulsivo.

La ecuación que rige el movimiento browniano de la simulación es la misma que la de el programa *Macroion* de la sección anterior (ec. 4.1). El único parámetro que cambia con respecto a la simulación del problema de Thomson es el tamaño de paso Δt, que para las simulaciones que se presentan se pudo aumentar a $\Delta t = 10^{-11}$ s, logrando una estabilización más rápida en un menor número de iteraciones.

La fuerza electrostática que sienten los iones del electrolito debido a los otros iones y a los sitios de carga del macroion también esta dada por la ecuación 4.2. En esta simulación la configuración de los sitios de carga del macroion está fija, por lo que no se calculan las interacciones entre estos.

La valencia (z_{ion}) para los iones es ± 1. Para el caso en que la carga del macroion es uniforme en todos los sitios, la valencia de cada sitio es igual a la carga del macroion entre el número de sitios ($z_s = Q_m/N^T$). En el segundo caso, la carga de los sitios también es ± 1, la suma de estos es igual a la carga neta del macroion.

Además de la fuerza electrostática entre los iones, también está presente la fuerza

de núcleo repulsivo entre iones, entre un ion y el macroion, y entre un ion con el cascarón exterior (R_c). La primera evita que los iones se superpongan; la segunda y tercera confinan al electrolito entre el radio del macroion (R_m) y el radio del cascarón exterior(R_c).

La fuerza de núcleo repulsivo con unidades reducidas es la siguiente:

$$\mathbf{F}^{\mathrm{rc}*}(\mathbf{r}^*_{\mathbf{ij}}) = \begin{cases} \frac{24}{\sigma^* 10^{-10}\ \mathrm{m}} \left[2\left(\frac{\sigma}{r_{ij}-\Delta_{ij}}\right)^{13} - \left(\frac{\sigma}{r_{ij}-\Delta_{ij}}\right)^{7} \right] \hat{\mathbf{r}}_{\mathbf{ij}}, & r^*_{ij} < \Delta^*_{ij} + 2^{1/6}\sigma^* \\ 0, & r^*_{ij} \geq \Delta^*_{ij} + 2^{1/6}\sigma^* \end{cases} \tag{4.6}$$

$$\Delta^*_{ij} = \frac{d^*_i + d^*_j}{2} - \sigma^* \tag{4.7}$$

donde d_i y d_j son los diámetros de los iones, del macroion o del cascarón. Esta fuerza cambia de dirección en el caso de la interacción ión-cascarón, ya que evita que salga del sistema. Los detalles de la ecuación se pueden ver en la sección 2.3.

La distancias límites de los iones son las siguientes:

- ion-ion: $\Delta_{ij} = 2 \times r_{ion} - \sigma$.
- ion-macroion: $\Delta^m_{ij} = R_m + r_{ion} - \sigma$.
- ion-capa exterior: $\Delta^c_{ij} = R_c - r_{ion} + \sigma$.

Y la fuerza de núcleo repulsivo empieza a actuar cuando:

- $r_{ij} \leq \Delta_{ij} + 2^{1/6}\sigma$.
- $r_{ij} \leq \Delta^m_{ij} + 2^{1/6}\sigma$.
- $r_{ij} \leq \Delta^c_{ij} - 2^{1/6}\sigma$.

El programa se divide en cuatro archivos:

1. ***main***: Declara las constantes y variables del programa, como: la longitud de Bjerrum (l_B); el radio del macroion (R_m); el límite exterior, también llamado radio del cascarón (R_c); el radio de los iones que componen el electrolito (r_{ion}); la σ del potencial de núcleo repulsivo; el número de sitios de carga que compone el macroion (N^T); el número de iones del electrolito (N), la carga neta del electrolito (Q_m), el número de iteraciones de la simulación; el número de CPUs y el número de la GPU que se va a utilizar. Además llama a las funciones principales de los otros archivos, conectándolos.

2. ***header***: Contiene las bibliotecas y declara las funciones que son implementadas y llamadas en los otros archivos.

3. ***Thomson***: Calcula la configuración con energía potencial mínima para N^T partículas en la superficie de una esfera de radio 1 Å.

4. ***Electrolito***: Simula las interacciones de un electrolito confinado compuesto por N iones, con un macroion central de carga discreta compuesto por N^T sitios de carga en su superficie. Se divide en cinco partes: *Inicialización*, *Dinámica browniana*, *Chequeo*, *Toma de imágenes* y *Crea archivos*.

En esta sección se presentan los algoritmos utilizados en el archivo llamado *Electrolito* en su versión serial y paralela.

A continuación, se presentan los algoritmos utilizados para calcular $\rho(r)$, $P(r)$ y $V(r)$, que corresponden a las partes *Toma de imágenes* y *Crea archivos*.

4.2.2 Algoritmos para calcular $\rho(r)$, $P(r)$ y $V(r)$

Para poder calcular estas propiedades se necesita de un dr para realizar las integrales numéricas, este dr define el espesor de una capa o cascarón esférico. Para las simulaciones llevadas a cabo se seleccionó $dr = 1/16$, entonces, el volumen esférico del sistema de radio R_c está conformado por $R_c \times 16$ capas esféricas de espesor dr. La primera capa tiene un radio igual a dr, la segunda tiene un radio $2dr$ y así sucesivamente hasta llegar a una capa con radio igual a R_c de espesor dr.

Con estas capas y conociendo la distancia de cada ion respecto al origen, se puede saber la cantidad de iones positivos y negativos que viven en cada región del sistema. Con esto se puede calcular $\rho(r)$, $P(r)$ y $V(r)$.

Durante la ejecución en paralelo del programa solo se calcula el número de iones positivos y negativos que viven en cada capa así como la suma de las cargas integradas por capa que van del origen al cascarón exterior. Y en la creación de los archivos de salida es cuando se calculan $\rho(r)$, $P(r)$ y $V(r)$, disminuyendo la cantidad de trabajo para su calculo.

La creación de archivos de salida no está paralelizada, ya que la impresión de los archivos tiene que ser serial. El hecho de que no esté paralelizada no afecta en nada el rendimiento del programa, ya que se crean archivos cada tercio del número total de iteraciones.

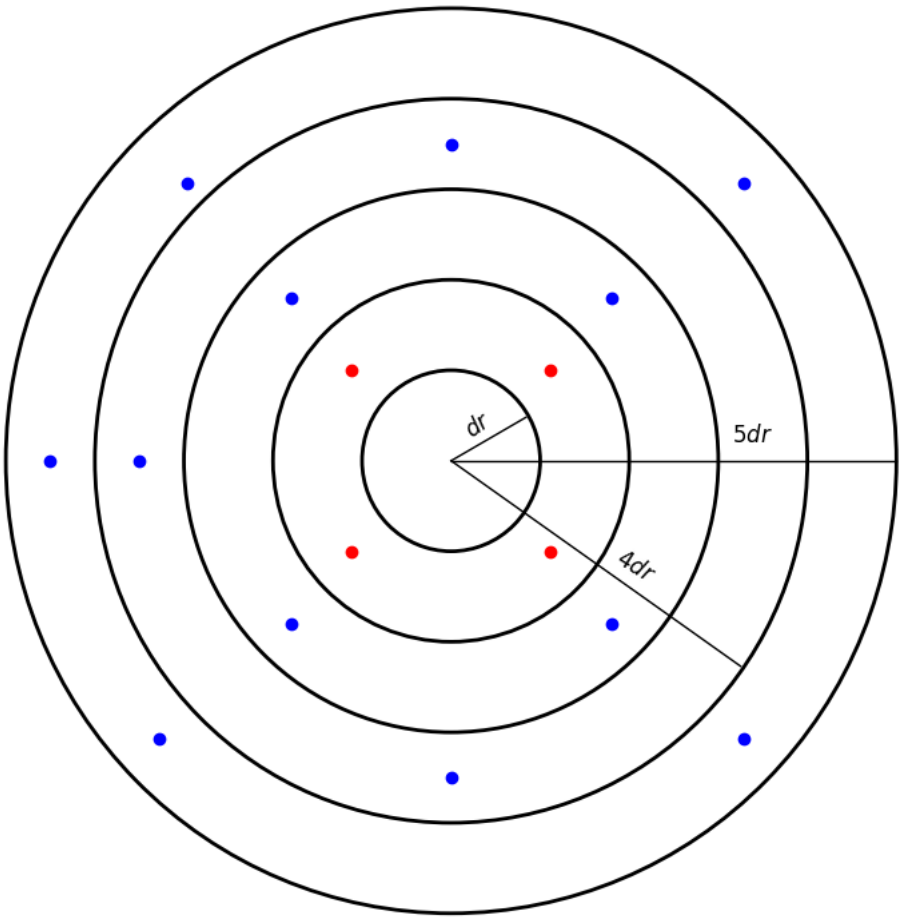

Figura 4.1: Representación de capas o cascarones esféricas con ancho dr. Los puntos azules representan iones con carga positiva y los rojos iones con carga negativa.

El calculo de iones positivos y negativos, así como de la carga integrada por capa, se realiza una vez que el sistema está en estado de equilibrio, a partir de ahí se toma el muestreo cada 10 pasos de la dinámica browniana.

Para calcular esto, primero se tiene que calcular la distancia de cada ion al origen, así como el número de capa al que pertenece, una vez hecho esto, dependiendo el signo de la valencia, el ion se añade a la capa correspondiente.

Con la distancia ion-origen calculada, se procede a iterar sobre cada capa desde el origen hasta R_c para añadir el número de iones que hay hasta la capa iterada, esto es la carga integrada.

Como las muestras se calculan cada 10 pasos, el resultado se debe dividir entre el número de muestras, obteniendo el promedio de la densidad por unidad de volumen de iones positivos y negativos, y la carga integrada por capa.

El algoritmo que toma estas muestras es el siguiente:

Algoritmo: Calculo de iones positivos, negativos y carga integrada por capa.

```
if iteracion > 10^6 & iteracion % 10 == 0 then
    for i ← 1 to N do
        Calcula_distancias_ion_origen(x, y, z);
        Calcula_número_de_capa(distancias, dr);
        if z_i > 0 then
            Suma la valencia del ion i al número de capa correspondiente del
             arreglo de iones positivos.
        else
            Resta la valencia del ion i al número de capa correspondiente del
             arreglo de iones negativos.
        end
    end
    for capa ← 1 to R_c * 16 do
        for i ← 1 to N do
            if distancia_i < capa × dr then
                Suma la valencia del ion i al número de capa correspondiente del
                 arreglo de la carga integrada.
            end
        end
    end
end
```

La primera parte, de la función *Crea_archivos*, calcula $\rho_+(r)$ y $\rho_-(r)$ de acuerdo a la cantidad de iones por capa en el muestreo. El radio de la capa (r), $\rho_+(r)$ y $\rho_-(r)$, se guardan en un archivo con un nombre distintivo, que contiene el número de iones, el número de sitios de carga, la iteración en la que se creó el archivo y el parámetro σ.

Para calcular $\rho_+(r)$ y $\rho_-(r)$, primero se calcula el volumen de la capa con radio $r + dr$ y después se promedia el número de iones positivos y negativos de esta capa, el resultado se pasa a unidades de mol sobre litro.

La parte dos, de la función *Crea_archivos*, promedia $P(r)$ para cada capa, guardándolo en un archivo cuyo nombre contiene los mismos datos que el archivo de $\rho(r)$. El archivo contiene el radio de la capa (r) y $P(r)$.

En la tercera parte se crea el archivo para $V(r)$, con el mismo formato que para $P(r)$, y se calcula $V(r)$ utilizando la carga integrada por capa de las muestras (sin promediar), con la integral de la ecuación 2.24.

La cuarta y última parte, crea un archivo que contiene las posiciones de cada ion del electrolito y sitio de carga del macroion. El formato del archivo está diseñado para visualizarse con el programa *Jmol*, por lo que, para identificar las partículas, se asigna un color de acuerdo al signo de su carga.

A continuación, se muestran los algoritmos de las cuatro partes de la función *Crea_archivos*:

Algoritmo: Función *Crea_archivos* - Parte 1

Resultado: Crea los archivos de $\rho(r)$, $P(r)$, $V(r)$ y de posiciones (x, y, z)

Entrada: Arreglos de posiciones; arreglo de iones positivos; arreglo de iones negativos; y arreglo de la carga integrada.

// $\rho_+(r)$ y $\rho_-(r)$ //

Crea y abre el archivo Rho(r).

Escribe la cabecera en el archivo Rho(r).

for $r \leftarrow 0$ **to** R_c **increases** $r \leftarrow r + dr$ **do**

- Calcula el volumen de la capa esférica de grosor dr con radio $r + dr$.
- Promedia el número de iones positivos de la capa en r/dr y calcula $\rho_+(r)$ en unidades [mol/litro].
- Promedia el número de iones negativos de la capa en r/dr y calcula $\rho_-(r)$ en unidades [mol/litro].
- Imprime en el archivo Rho(r) el radio de la capa (r), $\rho_+(r)$ y $\rho_-(r)$.

end

Cierra el archivo Rho(r).

Algoritmo: Función *Crea_archivos* - Parte 2

// $P(r)$ //

Crea y abre el archivo P(r).

Escribe la cabecera en el archivo P(r).

for $i \leftarrow 1$ **to** $R_c \times 16$ **do**

- Imprime en el archivo P(r) el radio de la capa i y el promedio de las cargas integradas en esa capa.

end

Cierra el archivo P(r).

Algoritmo: Función *Crea_archivos* - Parte 3

`// V(r) //`

Crea y abre el archivo V(r).

Escribe la cabecera en el archivo V(r).

for $capa \leftarrow R_c * 16$ **to** 1 **decreases** $capa \leftarrow capa - 1$ **do**

  Calcula $V(r)$ en milivoltios [mV].

end

for $i \leftarrow 1$ **to** $R_c \times 16$ **do**

  Imprime en el archivo V(r) el radio de la capa i y el promedio de los potenciales eléctricos en esa capa.

end

Cierra el archivo V(r).

Algoritmo: Función *Crea_archivos* - Parte 4

`// Posiciones //`

Crea y abre el archivo Posiciones.

Escribe la cabecera en el archivo Posiciones.

for $i \leftarrow 1$ **to** N^T **do**

  Imprime en el archivo Posiciones los sitios de carga del macroion. Dependiendo el signo de la carga se asigna un tipo de átomo para visualizar con Jmol.

end

for $i \leftarrow 1$ **to** N **do**

  Imprime en el archivo Posiciones los sitios de los iones del electrolito. Dependiendo el signo de la valencia se asigna un tipo de átomo para visualizar con Jmol.

end

Cierra el archivo Posiciones.

4.2.3 Implementación serial

En esta sección se presentan los algoritmos de *Inicialización* y *Dinámica browniana*, correspondientes al archivo *Electrolito*.

En la primera parte, *Inicialización*, se llaman todas las funciones iniciales y se declaran las variables para la simulación.

Entre las funciones iniciales se encuentran: asignación de memoria; inicialización de arreglos; posiciones iniciales; asignación de valencias; inicialización de números aleatorios; y reposición de los sitios de carga del macroion. Las posiciones también son aleatorias, pero la inicialización de estos números pseudoaleatorios se encuentra dentro de la función.

Las constantes declaradas pertenecen al cálculo del potencial de núcleo repulsivo, y las variables auxiliares controlan cuándo empieza el muestreo, cuándo se crean los archivos, cada cuántos pasos se toma la muestra y cada cuántos pasos se revisa que ningún ion haya salido del sistema, así como el cálculo de la energía potencial electrostática por partícula en ese momento.

La impresión de la cabecera es importante para saber qué sistema se está corriendo, ya que en las pruebas se utilizaron distintos parámetros.

Algoritmo: Electrolito Serial — Parte 1

Resultado: Simulación de dinámica browniana de un electrolito confinado alrededor de un macroion con carga discreta.

Entrada: Posiciones del macroion (x^T, y^T, z^T), constantes de la simulación (N, N^T, R_m, R_c, $iteraciones$, r_{ion}, σ).

`// Funciones iniciales //`

Asigna_memoria(x, y, z, Fx, Fy, Fz, v, v^T, ...);

Inicializa_arreglos(N, Fx, Fy, Fz, ...) ; `// Formatea los arreglos`

Posiciones_iniciales(N, x, y, z) ; `// Posiciones aleatorias`

Valencias(N, N^T, v, v^T, Q_m) ; `// Asigna las valencias y cargas`

Declara las variables auxiliares de la simulación.

Constantes y límites para los cálculos de las fuerzas de núcleo repulsivo.

Coloca los sitios del macroion en el límite donde los iones no llegan debido al potencial de núcleo repulsivo ($\Delta_{ij}^m = R_m + r_{ion} - \sigma$).

Inicializa_números_pseudoaleatorios();

Imprime_cabecera(R_m, R_c, Δt, σ, $iteraciones$);

El cambio entre los dos casos que se estudian yace en la función que asigna las valencias. Para el primer caso, la carga del macroion se reparte en partes iguales en los N^T sitios de carga de este. En el segundo caso, la carga de cada sitio es $+1$ o -1 y la

suma de estas cargas es igual a la carga neta del macroion (Q_m).

Los algoritmos de la función *Valencias* para ambos casos se presentan a continuación. Cabe mencionar que estos algoritmos solo se pueden aplicar para N, N^T y Q_m pares, si se desea utilizar un número impar hay que editar la función *Valencias*.

Algoritmo: Asignación de valencias - Caso 1

Resultado: Reparte la carga del macroion en los N^T sitios que lo componen.

Entrada: Arreglos de valencias de los iones, arreglo de las cargas del macroion, N, N^T y la carga del macroion (Q_m).

```
// Cargas del macroion //
num_sitios ← N^T;
carga_macroion ← Q_m;
for i ← 1 to N^T do
    carga_sitio[i] ← carga_macroion/num_sitios;
end
// Valencias de los iones //
for i ← 1 to N do
    if i ≤ N/2 − Q_m/2 then
        valencia[i] ← 1;
    else
        valencia[i] ← −1;
    end
end
```

El algoritmo de la función *Valencias* del primer caso reparte la carga neta del macroion, en los N^T sitios de este, en partes iguales. Después asignan las valencias de los iones, con valor ± 1, de tal forma que la suma de estas sea igual al negativo de la carga neta del macroion, para obtener un sistema electroneutro.

En cambio, el algoritmo de la función *Valencias* para el segundo caso, asigna las cargas de los N^T sitios con valencia ± 1, de tal manera que la suma de estos sea igual a la carga neta del macroion, sin importar el tamaño de N^T. Después se asigna la valencia de los iones para que el sistema sea electroneutro, igual que en el primer caso.

El siguiente algoritmo corresponde a la función *Valencias* para el segundo caso:

Algoritmo: Asignación de valencias - Caso 2

Resultado: Reparte la carga del macroion en N^T sitios que lo componen con cargas de valor ± 1.

Entrada: Arreglos de valencias de los iones, arreglo de las cargas del macroion, N, N^T y la carga del macroion (Q_m).

```
// Cargas del macroion //
```

for $i \leftarrow 1$ **to** N^T **do**
  if $i \leq N^T/2 + Q_m/2$ **then**
    $carga\_sitio[i] \leftarrow 1$;
  else
    $carga\_sitio[i] \leftarrow -1$;
  end
end

```
// Valencias de los iones //
```

for $i \leftarrow 1$ **to** N **do**
  if $i \leq N/2 - Q_m/2$ **then**
    $valencia[i] \leftarrow 1$;
  else
    $valencia[i] \leftarrow -1$;
  end
end

Ahora viene la parte bonita del programa, la segunda parte del programa *Electrolito*, donde se calculan todas las interacciones del sistema.

Primero se calculan las fuerzas electrostáticas que sienten los iones debido a los otros iones y a los sitios de carga del macroion; si la distancia entre un ion y otro es menor que $\Delta_{ij} + 2^{1/6}\sigma$, se calcula la fuerza de núcleo repulsivo entre estos. Después, si un ion se encuentra cerca del macroion, se calcula la fuerza de núcleo repulsivo para alejar al ion del macroion. Y si un ion está cerca del cascarón exterior, se calcula la fuerza de núcleo repulsivo para evitar que el ion salga del sistema. Con las fuerzas que siente cada ion calculadas, lo siguiente es calcular la nueva posición de este, utilizando la ecuación de movimiento de la dinámica browniana (ec. 4.1).

Algoritmo: Electrolito Serial — Parte 2

```
// Dinámica browniana //
for iteracion ← 1 to iteraciones do
    for i ← 1 to N do
        for j ← 1 to N do
            if i ≠ j then
                Calcula la fuerza electrostática (Fx, Fy, Fz) que actúa sobre
                 cada ion (i) debido a los demás iones (j) (ec. 2.28).
                if r_ij < Δ_ij + 2^(1/6)σ then
                    Calcula la fuerza de núcleo repulsivo (ec. 4.6) entre el ion i y
                     el ion j.
                end
            end
        end
        for j ← 1 to N^T do
            Calcula la fuerza electrostática (Fx, Fy, Fz) que actúa sobre cada
             ion (i) debido a los sitios de carga del macroion (j) (ec. 2.28).
        end
    end
    for i ← 1 to N^T do
        Calcula la distancia del ion i al origen.
        if distancia_i < Δ^m_ij + 2^(1/6)σ then
            Calcula la fuerza de núcleo repulsivo (ec. 4.6) que evita que el ion i
             entre al macroion.
        end
        if distancia_i > Δ^c_ij − 2^(1/6)σ then
            Calcula la fuerza de núcleo repulsivo (ec. 4.6) que evita que el ion i
             vaya más allá del cascarón exterior (R_c).
        end
    end
    for i ← 1 to N^T do
        Calcula las nuevas posiciones (x, y, z) para cada partícula (i) con la
         ecuación de movimiento de la dinámica browniana (ec. 4.1).
    end
    // Chequeo, impresiones, toma de imágenes y crea archivos //
end
```

4.2.4 Implementación con OpenMP

Al igual que con el programa anterior, es bastante simple implementar OpenMP con la estructura de la versión serial, solo se necesita con agregar la directiva `#pragma omp parallel for` antes de cada ciclo `for` externo que va de 1 hasta N en este caso, por ejemplo:

Algoritmo: Ciclos `for` paralelizados con OpenMP

```
#pragma omp parallel for
for i ← 1 to N do
    for j ← 1 to N do
        if i ≠ j then
            Calcula la fuerza electrostática neta que siente la partícula i debido
              a las partículas j, de forma paralela.
        end
    end
    for j ← 1 to N^T do
        Calcula la fuerza electrostática (Fx, Fy, Fz) que actúa sobre cada ion
          (i) debido a los sitios de carga del macroion (j) (ec. 2.28).
    end
end

#pragma omp parallel for
for i ← 1 to N do
    Calcula las nuevas posiciones para cada partícula (i), de forma paralela.
end
```

El número de hilos que se van a repartir los cálculos de los ciclos paralelizados, se especifica en el archivo *main*, con la línea "`omp_set_num_threads(n);`", donde n es el número de hilos.

En este programa se detectaron dos problemas de *race condition* que fueron solucionados utilizando la clausulas `reduction` y `atomic`.

El primer problema de *race condition* se encontró en el calculo de la energía potencial electrostática por partícula, al igual que en el programa anterior. La clausula `reduction(+:`$\bar{U}^{\mathrm{el}}$`)` fue suficiente para resolverlo.

El segundo problema de *race condition* se encontraba en las muestras de $\rho(r)$, este se resolvió utilizando la directiva `#pragma omp atomic update`.

Los algoritmos paralelizados con OpenMP y corregidos son los siguientes:

Algoritmo: Calculo de la energía potencial electrostática ($\bar{U}^{\text{el}}$) en paralelo con OpenMP

```
#pragma omp parallel for reduction(+:Ū^el)
for i ← 1 to N do
    for j ← 1 to N do
        if i ≠ j then
            Calcula la energía potencial electrostática por partícula (Ū^el) que
             hay entre cada partícula (i) con las demás partículas (j), de forma
             paralela con múltiples hilos.
        end
    end
end
return Ū^el
```

Algoritmo: Calculo de muestras en paralelo con OpenMP

```
if iteracion > 10^6 & iteracion % 10 == 0 then
    for i ← 1 to N do
        Calcula_distancias_ion_origen(x, y, z);
        Calcula_número_de_capa(distancias, dr);
        if z_i > 0 then
            #pragma omp atomic update
            Suma la valencia del ion i al número de capa correspondiente del
             arreglo de iones positivos.
        else
            #pragma omp atomic update
            Resta la valencia del ion i al número de capa correspondiente del
             arreglo de iones negativos.
        end
    end
    for capa ← 1 to R_c * 16 do
        for i ← 1 to N do
            if distancia_i < R_m + capa × dr then
                Suma la valencia del ion i al número de capa correspondiente del
                 arreglo de la carga integrada.
            end
        end
    end
end
```

4.2.5 Implementación con OpenACC

La primer directiva para esta versión se encuentra en el archivo *main*, esta es `#pragma acc set device_num(n)`, donde `n` es el número de la GPU en la que se van a correr las secciones paralelizadas.

La siguiente directiva es `#pragma acc enter data copyin(...)`, está al final de la inicialización en el archivo *Electrolito*, y es la encargada de copiar los arreglos y variables a la GPU antes de empezar la dinámica browniana.

Algoritmo: Electrolito paralelizado con OpenACC — Inicialización

Resultado: Simulación de dinámica browniana de un electrolito confinado alrededor de un macroion con carga discreta.

Entrada: Posiciones del macroion (x^T, y^T, z^T), constantes de la simulación (N, N^T, R_m, R_c, $iteraciones$, r_{ion}, σ).

`// Funciones iniciales //`

Asigna_memoria(x, y, z, Fx, Fy, Fz, v, v^T, ...);

Inicializa_arreglos(N, Fx, Fy, Fz, ...) ; `// Formatea los arreglos`

Posiciones_iniciales(N, x, y, z) ; `// Posiciones aleatorias`

Valencias(N, N^T, v, v^T, Q_m) ; `// Asigna las valencias y cargas`

Declara las variables auxiliares de la simulación.

Constantes y límites para los cálculos de las fuerzas de núcleo repulsivo.

Coloca los sitios del macroion en el límite donde los iones no llegan debido al potencial de núcleo repulsivo ($\Delta_{ij}^m = R_m + r_{ion} - \sigma$).

Inicializa los números pseudoaleatorios en GPU para OpenACC (sec. 3.3.3).

Imprime_cabecera(R_m, R_c, Δt, σ, $iteraciones$);

#pragma acc enter data copyin(x[:N], y[:N], z[:N], Fx[:N], Fy[:N], x^T[:N^T], y^T[:N^T], z^T[:N^T], N, N^T, R_m, Δt, ...)

Para paralelizar el calculo de las fuerzas electrostáticas y de núcleo repulsivo entre iones se utilizaron las directivas `#pragma acc parallel loop gang present(...) independent async` y `#pragma acc loop vector reduction(+:`Fxx`,` Fyy`,` Fzz`)`. La primera distribuye las iteraciones entre bloques de núcleos de la GPU y la segunda distribuye el trabajo en esos núcleos.

Las variables Fxx, Fyy, Fzz se utilizan para evitar el *race condition*, guardando temporalmente el valor de las fuerzas, ya que no se puede utilizar la directiva `reduction`

para arreglos. La sección paralelizada de la dinámica browniana se presenta en dos partes, esta es la primera:

Algoritmo: Electrolito paralelizado con OpenACC — Dinámica browniana - Parte 1

```
// Dinámica browniana - Parte 1 //
for iteracion ← 1 to iteraciones do
    #pragma acc parallel loop gang present(Fx[:N], Fy[:N], Fz[:N], ...)
     independent async
    for i ← 1 to N do
        // Variables auxiliares //
        Fxx ← 0.0;
        Fyy ← 0.0;
        Fzz ← 0.0;
        #pragma acc loop vector reduction(+:Fxx, Fyy, Fzz)
        for j ← 1 to N do
            if i ≠ j then
                Calcula la fuerza electrostática (Fxx, Fyy, Fzz) que actúa sobre
                 cada ion (i) debido a los demás iones (j) (ec. 2.28).
                if r_ij < Δ_ij + 2^(1/6)σ then
                    Calcula la fuerza de núcleo repulsivo (ec. 4.6) entre el ion i y
                     el ion j.
                end
            end
        end
        #pragma acc loop vector reduction(+:Fxx, Fyy, Fzz)
        for j ← 1 to N^T do
            Calcula la fuerza electrostática (Fxx, Fyy, Fzz) que actúa sobre
             cada ion (i) debido a los sitios de carga del macroion (j) (ec. 2.28).
        end
        Fx[i] ← Fxx;
        Fy[i] ← Fyy;
        Fz[i] ← Fzz;
    end
    // Continua en el siguiente algoritmo //
end
```

Algoritmo: Electrolito paralelizado con OpenACC — Dinámica browniana - Parte 2

```
// Dinámica browniana - Parte 2 //
for iteracion ← 1 to iteraciones do
    // Continuación del algoritmo anterior //
    #pragma acc parallel loop present(...) independent async
    for i ← 1 to N^T do
        Calcula la distancia del ion i al origen.
        if distancia_i < Δ^m_ij + 2^(1/6)σ then
            Calcula la fuerza de núcleo repulsivo (ec. 4.6) que evita que el ion i
             entre al macroion.
        end
        if distancia_i > Δ^c_ij − 2^(1/6)σ then
            Calcula la fuerza de núcleo repulsivo (ec. 4.6) que evita que el ion i
             vaya más allá del cascarón exterior (R_c).
        end
    end
    #pragma acc wait
    #pragma acc host_data use_device(...)
    {
        Genera un conjunto de números aleatorios de distribución normal con
         media cero y desviación estándar uno, utilizando la GPU.
    }
    #pragma acc parallel loop present(...) independent
    for i ← 1 to N^T do
        Calcula las nuevas posiciones (x, y, z) para cada partícula (i) con la
         ecuación de movimiento de la dinámica browniana (ec. 4.1).
    end
    // Chequeo, impresiones, toma de imágenes y crea archivos //
end
```

La segunda parte, utiliza las directivas `#pragma acc parallel loop`, `#pragma acc host_data use_device(...)` y `#pragma acc wait`. Y las clausulas `present(...)` `independent` `async`. La primera clausula indica las variables y arreglos que se encuentran en la región paralelizada; `independent` le dice al compilador que no hay una dependencia entre iteraciones; y la clausula `async` indica que el ciclo se va a correr en paralelo con los demás ciclos que contengan esta clausula.

La directiva `#pragma acc wait` se utiliza para esperar a que terminen los ciclos con

la clausula `async`. Todas estas directivas y clausulas se pueden ver con detalle en la sección 3.3.2.

Las secciones de chequeo y toma de imágenes o muestras también están paralelizadas, ya que se tiene que evitar a toda costa la transferencia de datos entre la GPU y la CPU.

Algoritmo: Calculo de iones positivos, negativos y carga integrada por capa - Paralelizado con OpenACC

```
if iteracion > 10^6 & iteracion % 10 == 0 then
    #pragma acc parallel loop present(x[:N], y[:N], z[:N], ...) independent
    for i ← 1 to N do
        Calcula_distancias_ion_origen(x, y, z);
        Calcula_número_de_capa(distancias, dr);
        if z_i > 0 then
            #pragma acc atomic update
            Suma la valencia del ion i al número de capa correspondiente del
             arreglo de iones positivos.
        else
            #pragma acc atomic update
            Resta la valencia del ion i al número de capa correspondiente del
             arreglo de iones negativos.
        end
    end
    #pragma acc parallel loop gang present(P[:R_c × 16]) independent
    for capa ← 1 to R_c * 16 do
        // Variable auxiliar //
        Prr ← 0.0;
        for i ← 1 to N do
            if distancia_i < capa × dr then
                Suma la valencia del ion i al número de capa correspondiente de
                 la carga integrada.
            end
        end
        P[r] ← Prr;
    end
end
```

El chequeo consiste en calcular la energía potencial electrostática por partícula (ec. 2.10), para saber si el sistema ya se encuentra en equilibro, y checa que los iones sigan entre el macroion y el cascarón exterior. Este es el algoritmo del chequeo paralelizado:

Algoritmo: Chequeo paralelizado con OpenACC

```
for iteracion ← 1 to iteraciones do
    // Dinámica Brwoniana //
    // Chequeo //
    if iteracion%chequeo == 0 then
        #pragma acc enter data copyin(Ū^el, dm, dM)
        #pragma acc parallel loop collapse(2) reduction(+:Ū^el) present(...)
         independent async
        for i ← 1 to N do
            for j ← 1 to N do
                if i ≠ j then
                    Calcula la energía potencial electrostática por partícula (ec.
                     2.10) que actúa sobre cada ion (i) debido a los demás iones
                     (j).
                end
            end
        end
        #pragma acc parallel loop collapse(2) reduction(+:Ū^el) present(...)
         independent async
        for i ← 1 to N do
            for j ← 1 to N^T do
                if i ≠ j then
                    Calcula la energía potencial electrostática por partícula (ec.
                     2.10) que actúa sobre cada ion (i) debido a los sitios de
                     carga (j) del macroion.
                end
            end
        end
        #pragma acc parallel loop reduction(min:dm) reduction(max:dM)
         present(...) independent
        for i ← 1 to N do
            Calcula la distancia del origen al ion i y guarda la distancia máxima
             y mínima en las variables dm y dM respectivamente.
        end
        #pragma acc wait
        #pragma acc exit data copyout(Ū^el, dm, dM)
    end
end
```

CAPÍTULO 5

Pruebas y resultados

Con los archivos de los programas listos, lo siguiente es subirlos o copiarlos al servidor en el que se van a realizar las pruebas. Para correr el programa se utiliza un *script* que contiene todas las banderas e instrucciones para la compilación y ejecución del programa en cualquier versión (Serial, OpenMP y OpenACC). Una vez termina la ejecución de los programas, los resultados crudos se encuentran en una carpeta llamada "Salidas" en el mismo directorio del programa, y se pueden ver desde la consola. Para graficarlos se tienen que descargar y utilizar un programa para su visualización. En las pruebas se utilizó *Gnuplot* y para los resultados finales se empleó la biblioteca *matplotlib*.

Las pruebas siguientes fueron ejecutadas en los servidores *Ampere* y *FJEstrada*. Los demás servidores se utilizaron para comprobar el funcionamiento de los programas, pero los servidores con mejor *hardware* fueron los utilizados para las pruebas finales. Las especificaciones de los servidores se pueden ver en la sección 3.5.

5.1 Macroion con carga discreta

Este programa crea un archivo con la configuración de energía potencial mínima para N^T partículas en la superficie de una esfera de radio 1 Å. Además, se guarda la energía potencial de esta configuración, la cual se puede comparar con los resultados de David J. Wales y Sidika Ulker [35].

Los parámetros utilizados para este programa son:

- Coeficiente de difusión del medio: $D = 10^{-12}\,\mathrm{m}^2/\mathrm{s}$.
- Temperatura absoluta del sistema: $T = 298.0$ K.
- Tamaño de paso de la dinámica browniana: $\Delta t = 10^{-12}$ s.
- Número de pasos de la dinámica browniana: 10^4.

- Máximo número de pasos del método de descenso de gradiente: 3×10^4.

5.1.1 Validación del modelo

La siguiente tabla muestra una comparación de los resultados de *The Cambridge Cluster Database* [36] con los de la simulación desarrollada para encontrar la configuración de energía potencial mínima, en la superficie de una esfera de radio 1 Å, para diferente número de partículas (N^T):

$\mathbf{N^T}$	**Energía potencial ($\mathbf{U^T}$)**		**Error**
	Wales (2009)	Simulación	
10	32.7169494	32.7169494	$< 10^{-6}$ %
80	2805.355876	2805.355876	$< 10^{-6}$ %
972	455651.0809	455654.2704	$< 10^{-3}$ %
4352	9311276.276	9311313.465	$< 10^{-3}$ %

Tabla 5.1: Energías potenciales mínimas para una configuración de N^T partículas en la superficie de una esfera de radio 1 Å.

Estos resultados aplican para las tres versiones desarrolladas (Serial, con OpenMP y con OpenACC), así fue como se validó el funcionamiento de estas.

Para $N^T < 100$, el método de descenso de gradiente converge en cientos de iteraciones, mientras que para un número de partículas más grande el método se limita a 3×10^4 pasos, asegurando un error menor a 10^{-3}% con respecto a los resultados de Wales (2009) [36].

Ya que la diferencia de potenciales es mínima, se pueden utilizar las configuraciones para simular los sitios de carga del macroion con carga discreta, simulando ser un solo ente de manera precisa.

El archivo que crea el programa, con la configuración de energía potencial mínima, está diseñado para leerse con *Jmol*, por lo que se puede ver la configuración para N^T partículas de forma rápida y sencilla.

A continuación, se muestran las configuraciones para $N^T = 80$ y para $N^T = 972$, creadas con la simulación del macroion con carga discreta.

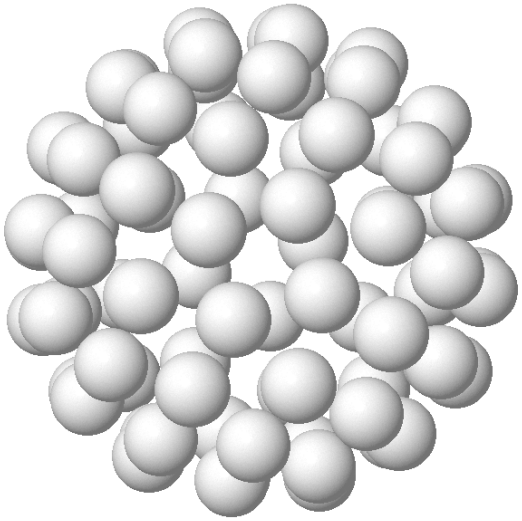

Figura 5.1: Configuración con energía potencial mínima para 80 partículas en una esfera de radio igual 1 Å.

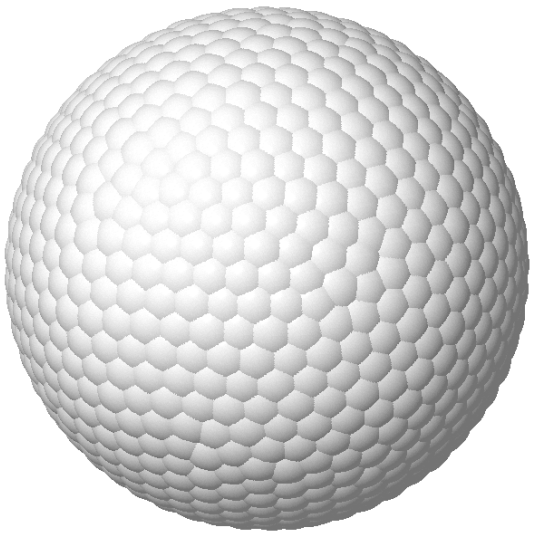

Figura 5.2: Configuración con energía potencial mínima para 972 partículas en una esfera de radio igual 1 Å.

5.1.2 Tiempos: Serial vs OpenMP vs OpenACC

Con las distintas versiones del programa validadas, se procedió a comparar los tiempos de ejecución, tomando los tiempos que conlleva hacer el número máximo de pasos del método de descenso de gradiente para $N^T > 100$, ya que para un número de partículas menor el método converge con solo cientos de iteraciones.

$\mathbf{N^T}$	**Tiempo de ejecución [s]**		
	Serial	**OpenMP**	**OpenACC**
8	0.05	1.79	0.38
80	0.54	2.61	0.66
160	5.09	3.25	0.99
1000	189.15	10.24	2.92
2000	757.85	29.87	5.47
4000	3029.04	90.74	12.7
8000	12,049.00	352.2	37.2

Tabla 5.2: Tiempos de ejecución en segundos del programa *Macroion con carga discreta*. Estos programas se ejecutaron en el servidor *Ampere* y se utilizaron 96 hilos para la versión con OpenMP. Las especificaciones de este servidor se encuentran en la sección 3.5.

La versión serial parece tener una complejidad $O(n^2)$ en comparación con las otras versiones. El programa con OpenMP tiene un tiempo de ejecución correspondiente a $O(n \log n)$ y la implementación con OpenACC $O(n)$. La versión serial es la que tiene la pendiente más pronunciada, aunque para un N^T pequeño, esta es la mejor opción, debido a la convergencia rápida.

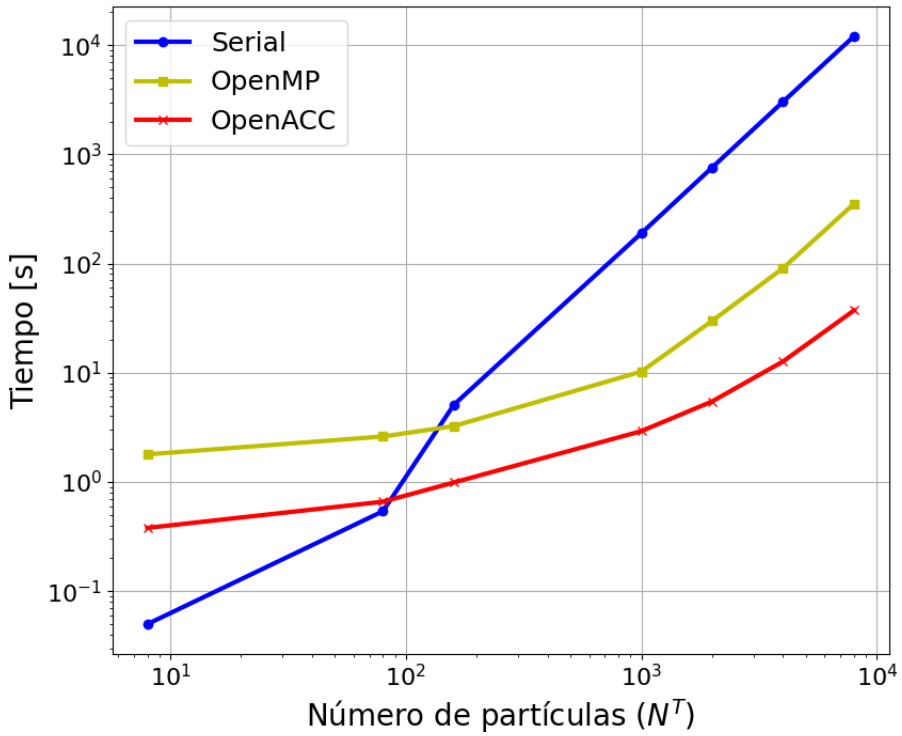

Figura 5.3: Tiempos de ejecución en segundos del programa *Macroion con carga discreta*. Estos programas se ejecutaron en el servidor *Ampere* y se utilizaron 96 hilos para la versión con OpenMP. Las especificaciones de este servidor se encuentran en la sección 3.5.

Las versiones paralelizadas con OpenMP y con OpenACC son más lentas que la versión serial para $N^T < 80$. En el caso de la versión con OpenMP es porque los datos se tienen que repartir entre 96 hilos, y para un número de partículas pequeño esto tiene un impacto negativo en el tiempo. Con respecto a OpenACC, la copia de datos a la GPU y desde la GPU, es más tardada que la ejecución serial para pocas partículas, además, este programa está diseñado para comprobar la convergencia cada 1000 iteraciones y no en cada iteración del descenso de gradiente, esto para evitar la transferencia de datos en cada iteración entre la GPU y el CPU.

5.2 Electrolito alrededor de un macroion con carga discreta

Las salidas de este programa son los archivos correspondientes a la densidad de carga por unidad de volumen de iones positivos y negativos, la carga integrada, el potencial eléctrico, la energía potencial electrostática, y las posiciones de cada ion del electrolito y sitio de carga del macroion con su signo correspondiente.

Para este programa se analizaron dos casos en particular:

- **Caso 1:** *Electrolito confinado alrededor de un macroion con carga discreta uniforme.* La carga del macroion se distribuye en los N^T sitios de carga de manera uniforme ($z_s = Q_m/N^T$).
- **Caso 2:** *Electrolito confinado alrededor de un macroion con carga discreta, compuesta por cargas positivas y negativas* Los N^T sitios de carga tienen valencia $+1$ y -1 ($z_s = \pm 1$), la suma de estos es igual a la carga neta del macroion (Q_m).

Antes de empezar a obtener los resultados para estos casos, hay que validar las interacciones del electrolito para cada implementación (serial, con OpenMP o con OpenACC), y revisar cuál de estas tiene los menores tiempos de ejecución.

5.2.1 Validación del modelo

Para validar el comportamiento del electrolito se utilizó un sistema con $N^T = 1$ (en el centro) y se comparó con corridas de Monte Carlo (MC) para esferas duras, con los siguientes parámetros:

- $\sigma = 5.0$ Å, $\sigma_{MC} = 0.0$ Å.
- Carga del macroion: $Q_m = -72$.
- Valencia de sitio del macroion: $z_s = Q_m$.
- Sitios de carga del macroion: $N^T = 1$.
- Valencia de los iones: $z_{ion} = \pm 1$.
- Constante dieléctrica: $\epsilon_r = 78.5$.

- Radio del macroion: $R_m = 25.0$ Å.
- Radio de los iones: $r_{ion} = 2.5$ Å.
- Radio de la última capa: $R_c = 106.0$ Å.
- Número de iones del electrolito: $N = 1272$.
- Coeficiente de difusión del medio: $D = 10^{-12}\mathrm{m}^2/\mathrm{s}$.
- Temperatura absoluta del sistema: $T = 298.0$ K.
- Tamaño de paso de la dinámica browniana: $\Delta t = 10^{-11}$ s.
- Número de pasos de la dinámica browniana: 3×10^8.

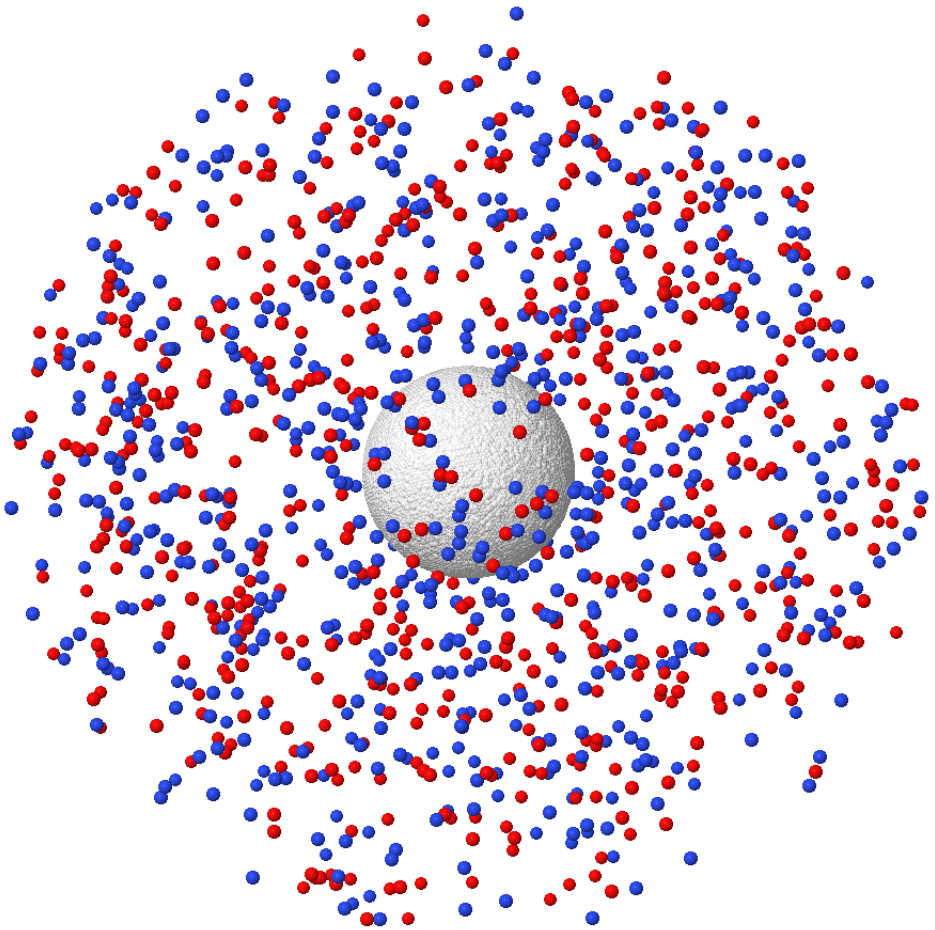

Figura 5.4: Sistema para validar el comportamiento del electrolito alrededor de un macroion de carga uniforme ($N^T = 1$). La carga del macroion ($Q_m = -72$) está concentrada en su centro. Las esferas azules y rojas representan los iones ($N = 1272$) con valencias positivas ($z_{ion} = +1$) y negativas ($z_{ion} = -1$) respectivamente.

La figura 5.17 es un sistema en equilibrio y no se alcanza a apreciar la variación en la distribución de los iones, por esto se calculan $\rho(r)$, $P(r)$ y $V(r)$.

5.2.1.1 Versión Serial vs OpenMP vs OpenACC vs Monte Carlo

El comportamiento del sistema mostrado en la figura 5.17 se comparó con corridas de Monte Carlo para un sistema con los mismos parámetros, pero para esferas duras, es decir, los iones del electrolito no pueden estar en contacto entre ellos ni con el macroion ($\sigma_{MC} = 0$), por lo que el modo de actuar en los límites de r debe ser diferente.

A continuación, se presentan las comparaciones del sistema de validación, descrito en la sección 5.2.1), para las diferentes versiones del programa desarrolladas (serial, con OpenMP o con OpenACC), junto con las corridas de Monte Carlo.

Densidad de carga por unidad de volumen de iones positivos y negativos [$\rho(r)$]:

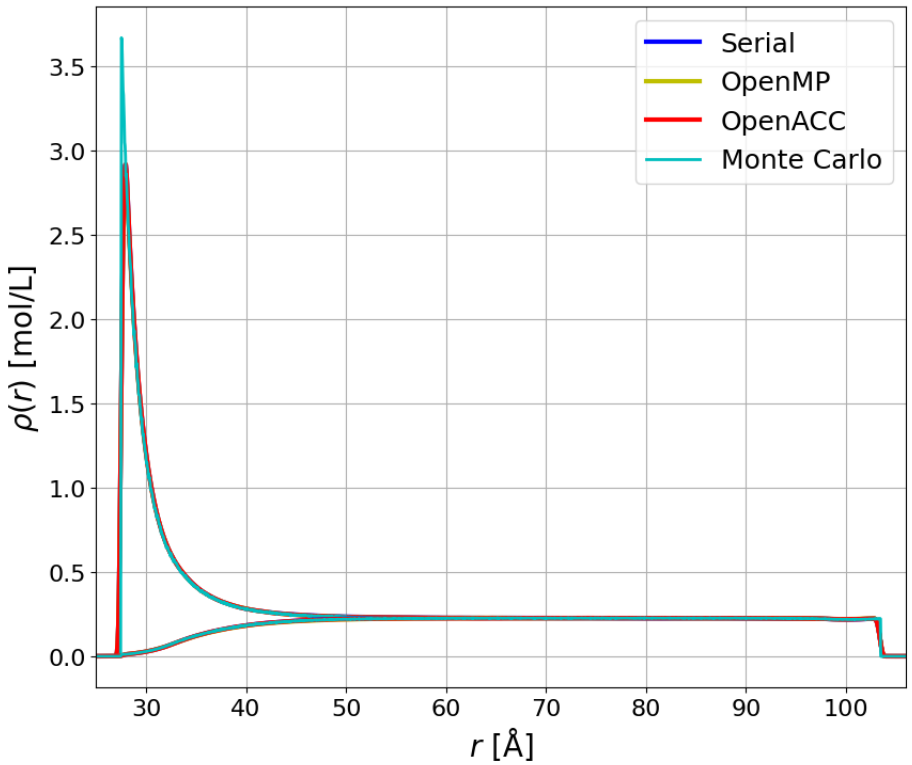

Figura 5.5: Densidad de carga por unidad de volumen de iones positivos (mayor concentración en el inicio) y iones negativos (menor concentración en el inicio) de las distintas versiones del sistema de validación, descrito en la sección 5.2.1. Los datos están superpuestos, por esto solo se aprecia un color en gran parte de la gráfica.

En la vista general de $\rho(r)$ se aprecia una variación en la concentración de iones positivos al inicio ($r < 50$ Å), después se ve la misma manera de actuar para los iones positivos y negativos para todas las versiones, hasta llegar al final ($R_c = 106$ Å).

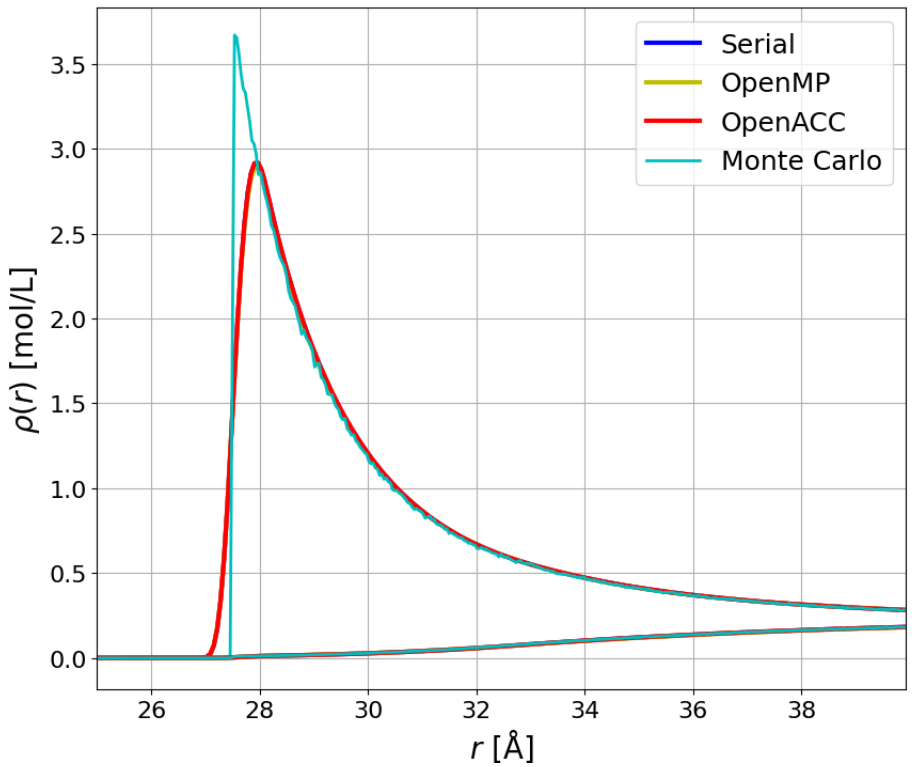

Figura 5.6: Densidad de carga por unidad de volumen de iones positivos (mayor concentración) y iones negativos (menor concentración) cercanos al macroion, de las distintas versiones del sistema de validación, descrito en la sección 5.2.1. Los datos para las versiones desarrolladas están superpuestos, por esto solo se aprecia el color correspondiente a la versión con OpenACC.

En la figura 5.6 se puede ver la distinción del parámetro σ entre las versiones desarrolladas y la corrida de Monte Carlo. No se aprecia un cambio entre las corridas de las versiones implementadas.

La siguiente figura muestra la parte final de $\rho(r)$, en esta no hay una varianza entre las densidades por unidad de volumens de iones positivos y negativos. Se puede ver la diferencia entre los parámetros $\sigma = 5.0$ y $\sigma_{MC} = 0.0$.

La diferencia entre el parámetro σ se ve con más detalle en la siguiente sección, por ahora solo se muestra la conducta del electrolito para el sistema de validación corrido de forma serial, con OpenMP y con OpenACC, comparado con la corrida de Monte Carlo.

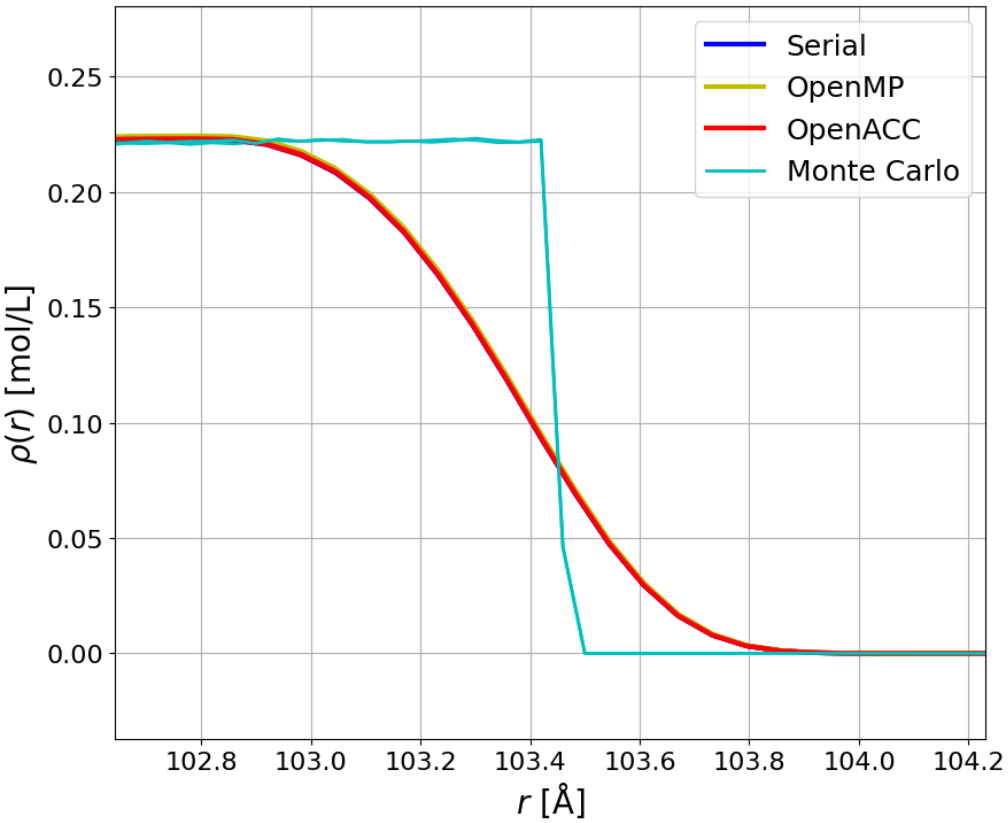

Figura 5.7: Densidad de carga por unidad de volumen de iones positivos y iones negativos cercanos al cascarón exterior, de las distintas versiones del sistema de validación, descrito en la sección 5.2.1. Los datos para las versiones desarrolladas están superpuestos, por esto solo se aprecia el color correspondiente a la versión con OpenACC.

Carga integrada $[P(r)]$**:**

En la vista general de la carga integrada (figura 5.8) no se aprecia un cambio notorio entre las corridas del sistema de validación.

Al acercarse a la curva del inicio ($27 < r < 28$) de la carga integrada (figura 5.9), se puede ver la distinción del parámetro σ de la corrida con Monte Carlo. No se nota una divergencia entre las implementaciones creadas.

Con esta conducta tan similar que se presenta, se puede decir que la simulación del electrolito es válida. De existir alguna discrepancia entre las versiones desarrolladas, entonces se tendría que revisar el programa, tanto la dinámica browniana como la creación de los archivos, ya que podría existir un problema de *race condition* en tal caso.

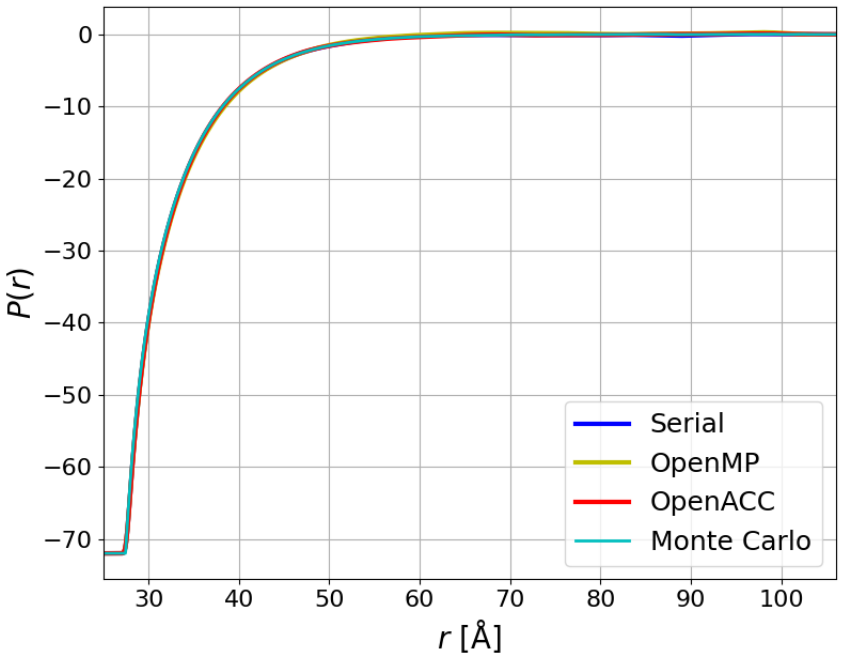

Figura 5.8: Vista general de la carga integrada de las distintas versiones del sistema de validación, descrito en la sección 5.2.1. Los datos están superpuestos, por esto solo se aprecia un color en gran parte de la gráfica.

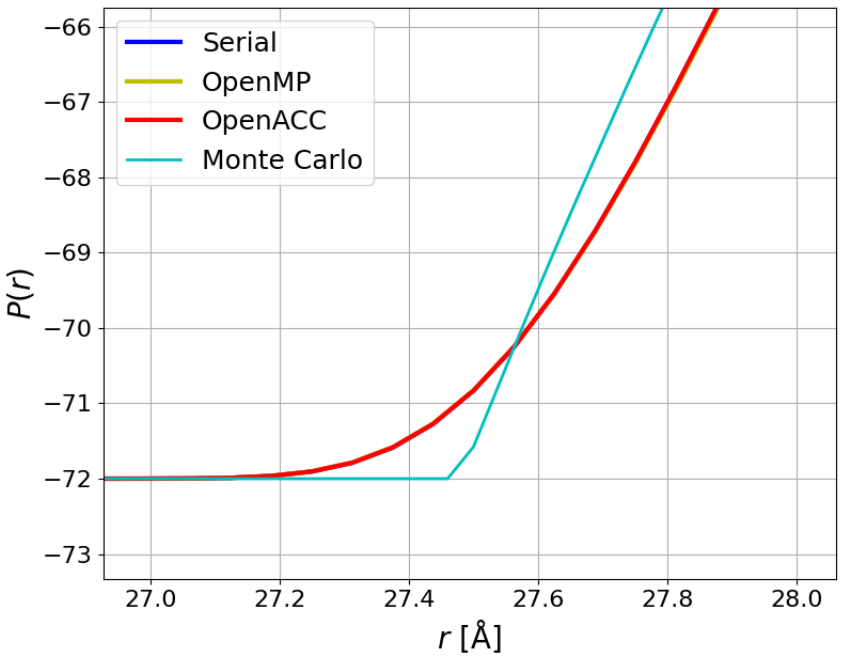

Figura 5.9: Carga integrada cerca del macroion de las distintas versiones del sistema de validación. Los datos para las versiones desarrolladas están superpuestos, por esto solo se aprecia el color correspondiente a la versión con OpenACC.

Potencial eléctrico $[V(r)]$**:**

Ya que el potencial eléctrico depende de la carga integrada, su modo de actuar es muy similar, pero no es el mismo.

En la vista general del potencial eléctrico (figura 5.10) no hay un contraste entre las corridas presentadas, parece que es el mismo comportamiento.

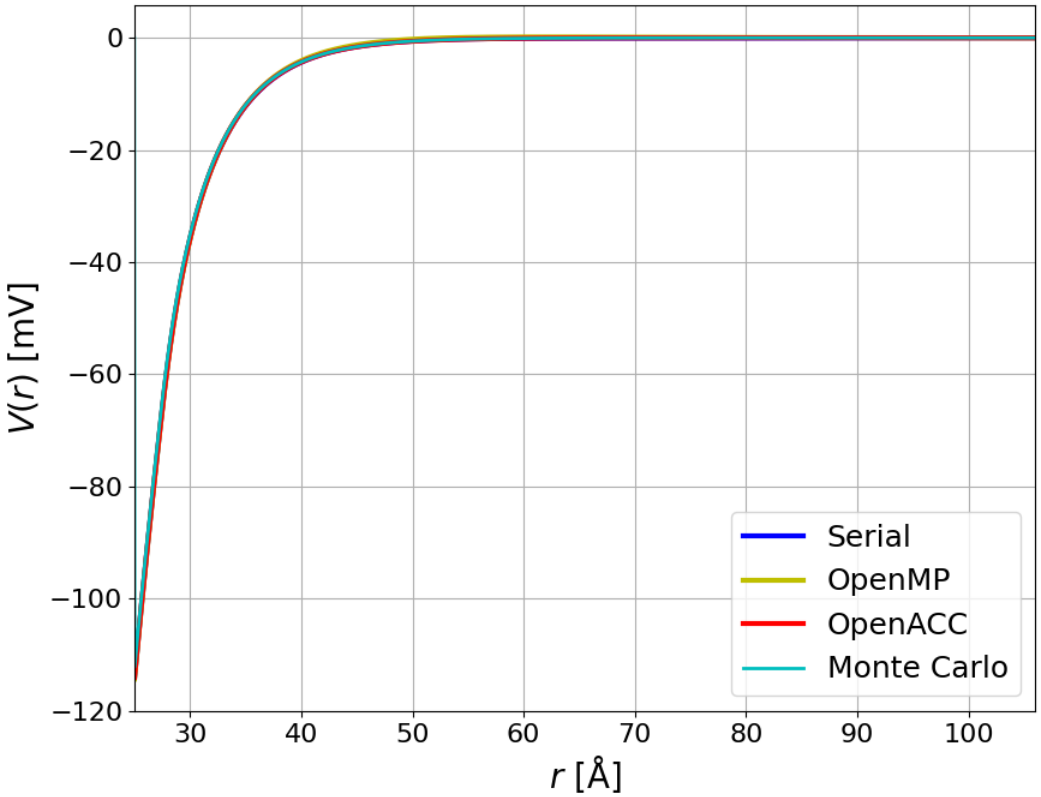

Figura 5.10: Vista general del potencial eléctrico de las distintas versiones del sistema de validación, descrito en la sección 5.2.1. Los datos están superpuestos, por esto solo se aprecia un color en gran parte de la gráfica.

Para poder ver un cambio entre las corridas, se tiene que hacer un gran acercamiento. Este cambio tiene que aparecer ya que las corridas son individuales, pero con los mismos parámetros. En la figura 5.11 se observa esta pequeña variación entre las corridas de las versiones desarrolladas.

También se puede notar nuevamente el cambio con respecto a la corrida de Monte Carlo, el parámetro σ, haciendo que las curvas se encuentren separadas al inicio ($r < 50$ Å).

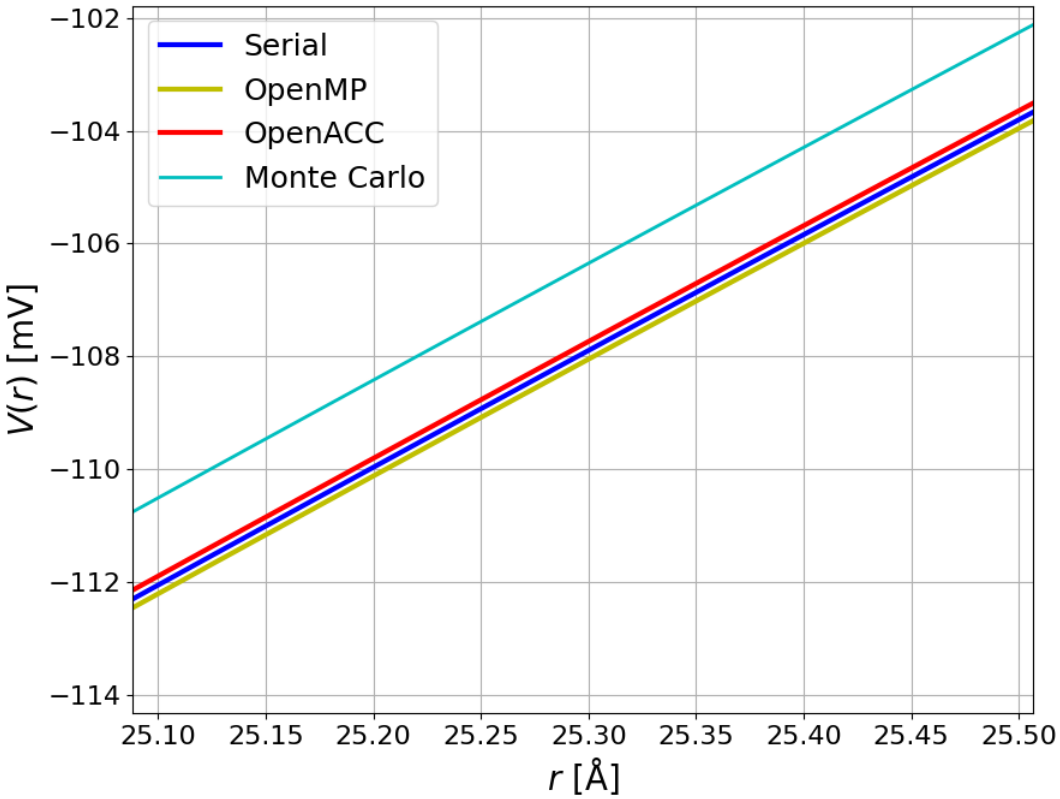

Figura 5.11: Potencial eléctrico cercano al macroion de las distintas versiones del sistema de validación, descrito en la sección 5.2.1.

5.2.1.2 Parámetro σ

Este parámetro define qué tanto se puede traslapar un ion con el macroion, con el cascarón exterior y con otro ion. Se corrió el sistema de validación, descrito en la sección 5.2.1, con OpenACC para distintas σ, específicamente con $\sigma = 1.0$, 3.0 y 5.0 Å. La corrida de Monte Carlo tiene $\sigma_{MC} = 0.0$ Å, ya que simula esferas duras.

Los iones del sistema de validación, descrito en la sección 5.2.1 tienen un radio $r_{ion} = 2.5$ Å, el macroion $R_m = 25.0$ Å, y el cascarón exterior $R_c = 106.0$ Å.

La distancias límites de los iones son las siguientes:

- ion-ion: $\Delta_{ij} = 2 \times r_{ion} - \sigma$.
- ion-macroion: $\Delta_{ij}^{m} = R_m + r_{ion} - \sigma$.
- ion-capa exterior: $\Delta_{ij}^{c} = R_c - r_{ion} + \sigma$.

Y la fuerza de núcleo repulsivo empieza a actuar cuando:

- $r_{ij} \leq \Delta_{ij} + 2^{1/6}\sigma$.
- $r_{ij} \leq \Delta_{ij}^{m} + 2^{1/6}\sigma$.
- $r_{ij} \leq \Delta_{ij}^{c} - 2^{1/6}\sigma$.

Para utilizar $\sigma = 1.0$ Å, el tamaño de paso de la dinámica browniana se tiene que reducir un orden, sino los iones se pueden sobreponer ($r_{ij} = 0$ Å), debido a que la fuerza repulsiva no alcanza a entrar en acción, produciendo una conducta irreal y fallas en el programa.

Para $\sigma = 3.0$ Å y $\sigma = 5.0$ Å, se utiliza el mismo tamaño del sistema de validación.

A continuación, se presenta la forma de comportarse del sistema de validación, descrito en la sección 5.2.1, para las diferentes σ, junto con las corridas de Monte Carlo.

Densidad de carga por unidad de volumen de iones positivos y negativos [$\rho(r)$]:

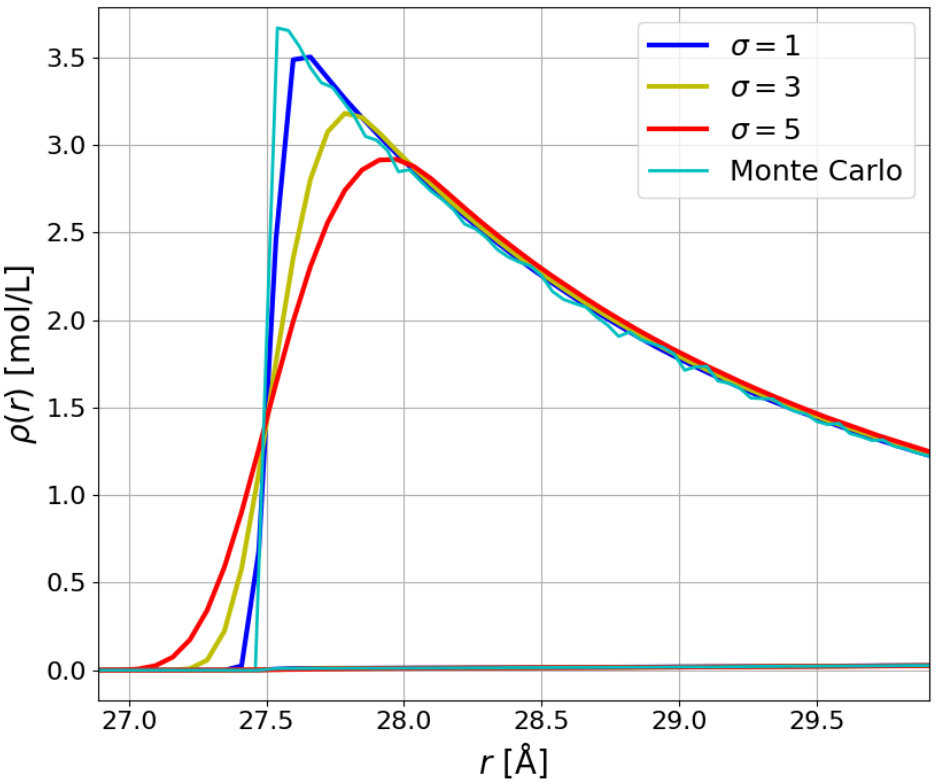

Figura 5.12: Densidad de carga por unidad de volumen de iones positivos (mayor concentración) y iones negativos (menor concentración), cercanos al macroion, de las corridas con diferente σ [Å] del sistema de validación, descrito en la sección 5.2.1. Monte Carlo tiene $\sigma_{MC} = 0.0$ Å.

Al calcular $\rho(r)$, la distinción de las corridas con diferente parámetro σ se encuentra en los límites del sistema, esto es, cerca del macroion y del cascarón exterior.

En la figura 5.12 se muestra la desigualdad de la densidad de carga por unidad de volumen de iones positivos para las distintas σ en la parte inicial ($r < 30$ Å). Se aprecia el recorrido de la concentración debido al parámetro σ, los picos máximos se encuentran alrededor de la distancia límite ion-macroion (Δ_{ij}^{m}).

La siguiente figura muestra el contraste de la densidad de carga por unidad de volumen de iones negativos para las diferentes σ en la parte inicial ($r < 30$ Å).

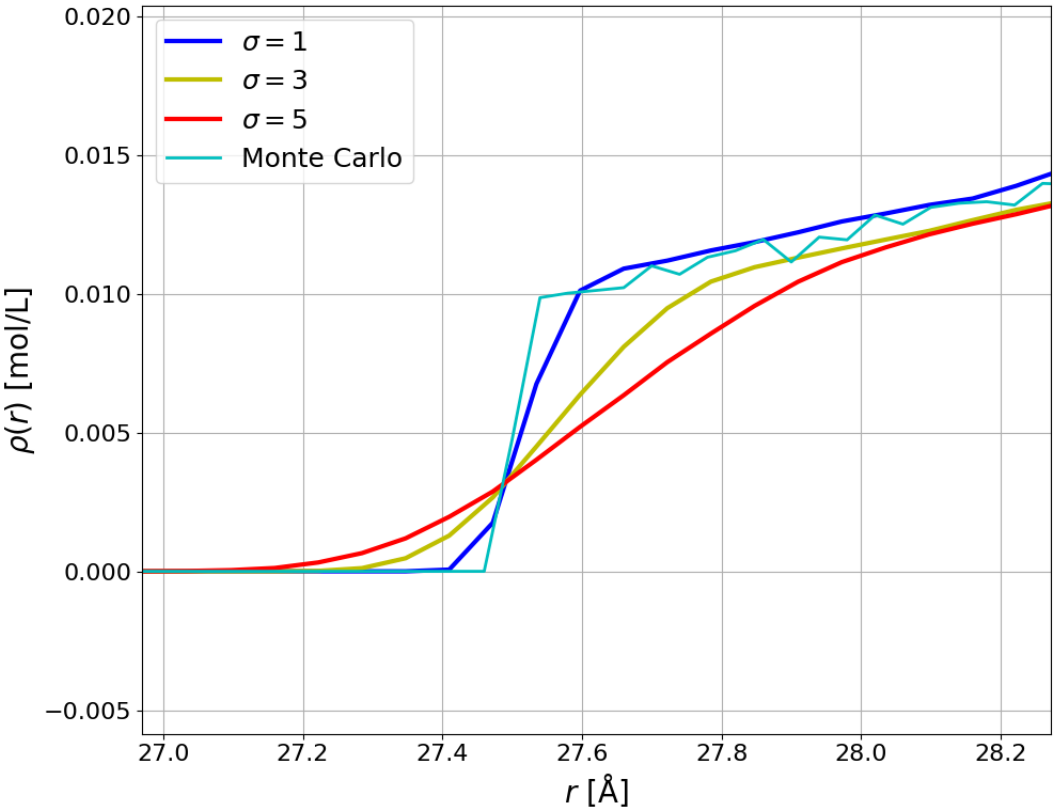

Figura 5.13: Densidad de carga por unidad de volumen de iones negativos cercanos al macroion, de las corridas con diferente σ [Å] del sistema de validación, descrito en la sección 5.2.1. Monte Carlo tiene $\sigma_{MC} = 0.0$ Å.

De la figura 5.13 se puede ver la pequeña concentración de iones negativos alrededor del macroion, debido a que este último también tiene carga negativa. Aún así existe una distinción entre las corridas con diferente σ.

La figura siguiente muestra la densidad de carga por unidad de volumen de iones positivos y negativos de la parte final ($102 < r < R_c$). Se puede ver la misma concen-

tración de iones negativos y negativos para cada σ, así como el desplazamiento de cada corrida.

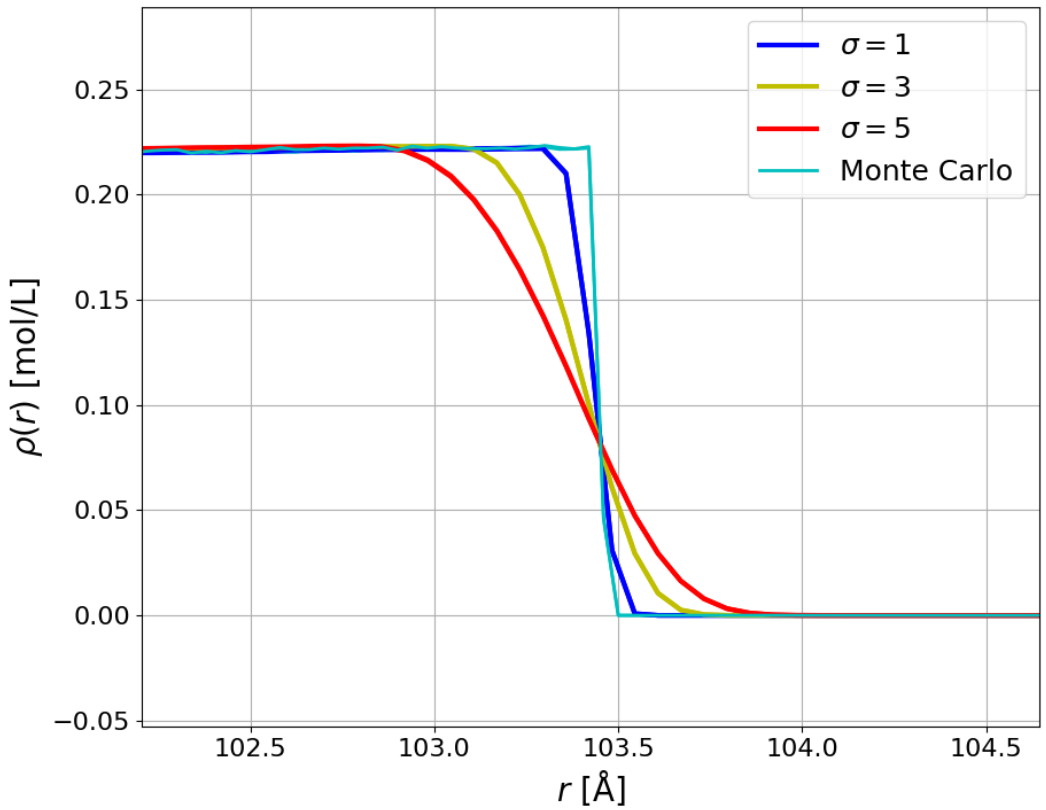

Figura 5.14: Densidad de carga por unidad de volumen de iones positivos y negativos cercanos al cascarón exterior, de las corridas con diferente σ [Å] del sistema de validación, descrito en la sección 5.2.1. Monte Carlo tiene $\sigma_{MC} = 0.0$ Å.

Carga integrada $[P(r)]$ y potencial eléctrico $[V(r)]$:

La carga integrada $[P(r)]$ y el potencial eléctrico $[V(r)]$ no miden lo mismo, la primera mide la cantidad de carga encerrada en una esfera de radio r, y el potencial eléctrico mide la energía potencial eléctrica, en milivoltios, de esa carga encerrada.

La forma de las curvas de $P(r)$ y $V(r)$ son muy parecidas entre las corridas con distinta σ, si se ven de forma general no se alcanza a distinguir un cambio entre estas.

Para ver la distinción de las corridas con diferente σ se tiene que hacer un gran acercamiento a las curvas de $P(r)$ y $V(r)$. A continuación, se muestran las figuras con estos acercamientos:

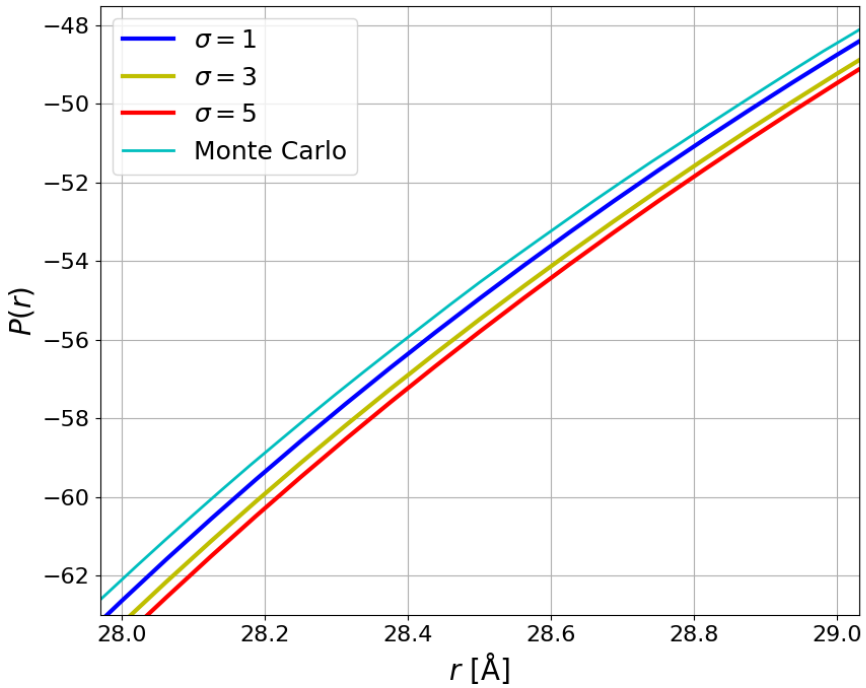

Figura 5.15: Acercamiento a la curva de la carga integrada $[P(r)]$, de las corridas con diferente σ [Å] del sistema de validación, descrito en la sección 5.2.1.

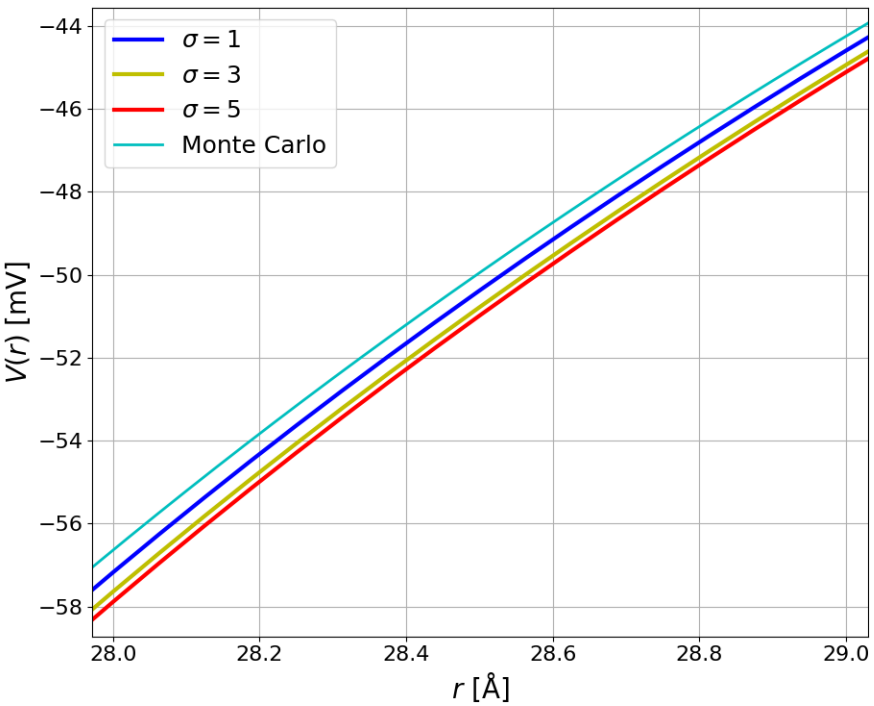

Figura 5.16: Acercamiento a la curva del potencial eléctrico $[V(r)]$, de las corridas con diferente σ [Å], del sistema de validación, descrito en la sección 5.2.1.

Con el comportamiento del electrolito validado, lo siguiente es juntar ambos programas, para obtener la simulación de un electrolito confinado alrededor de un macroion de carga discreta.

5.2.2 Tiempos: Serial vs OpenMP vs OpenACC

Para comparar los tiempos de las versiones, se utilizó como base el primer caso, el electrolito confinado alrededor de un macroion con carga discreta uniforme. Los tiempos deben ser iguales para el segundo caso, ya que solo cambia la distribución de la carga del macroion.

Los parámetros para el primer caso son los siguientes:

- $\sigma = 5.0$ Å.
- Carga del macroion: $Q_m = -72$.
- Valencia de los iones: $z_{ion} = \pm 1$.
- Valencia de sitio del macroion: $z_s = Q_m/N^T$.
- Constante dieléctrica: $\epsilon_r = 78.5$.
- Radio del macroion: $R_m = 25.0$ Å.
- Radio de los iones: $r_{ion} = 2.5$ Å.
- Radio de la última capa: $R_c = 106.0$ Å.
- Número de iones del electrolito: $N = 1272$.
- Temperatura absoluta del sistema: $T = 298.0$ K.
- Coeficiente de difusión del medio: $D = 10^{-12}\,\mathrm{m}^2/\mathrm{s}$.
- Tamaño de paso de la dinámica browniana: $\Delta t = 10^{-11}$ s.
- Número de pasos de la dinámica browniana: 3×10^8.

El único cambio con respecto al sistema de validación es el número de sitios de carga que componen al macroion, en este caso no hay un número fijo y en el de validación $N^T = 1$. Por esto, el primer caso se puede comparar con con el sistema de validación.

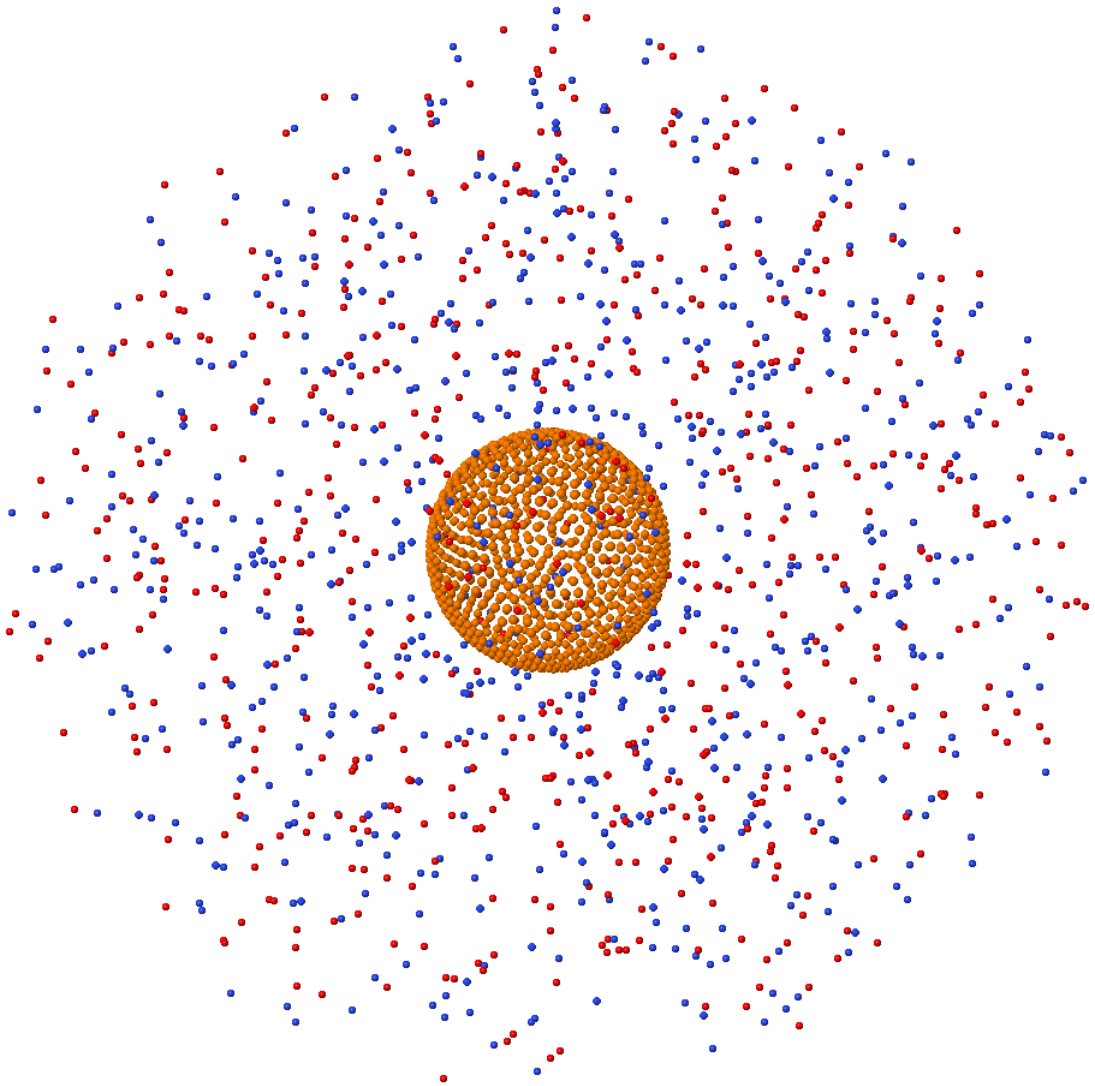

Figura 5.17: Electrolito alrededor de un macroion con carga discreta uniforme. La carga del macroion ($Q_m = -72$) está repartida en los N^T sitios de carga de este, de manera uniforme ($z_s = Q_m/N^T$). Las esferas naranjas representan los sitios de carga de macroion ($N^T = 1000$), mientras que las esferas azules y rojas simbolizan los iones del electrolito ($N = 1272$), con valencias positivas ($z_{ion} = +1$) y negativas ($z_{ion} = -1$) respectivamente.

La figura anterior muestra el resultado de las posiciones de los sitios de carga del macroion (esferas naranjas) y de los iones (esferas azules y rojas) para un sistema con $N^T = 1000$, correspondiente al primer caso.

A continuación, se presentan la tabla y la gráfica de tiempos de ejecución del programa para 10^4 pasos de la dinámica browniana, de los 3×10^8 pasos totales.

N^T	Tiempo de 10^4 pasos [s]		
	Serial	**OpenMP**	**OpenACC**
8	69.20	4.46	0.72
80	70.90	4.52	0.70
160	72.90	4.50	0.71
1000	92.50	5.10	0.78
2000	115.9	5.73	0.88
4000	162.7	7.31	1.09
8000	256.2	10.37	1.60

Tabla 5.3: Tiempos de ejecución de la versión serial, con OpenMP y con OpenACC del programa *Electrolito alrededor de un macroion con carga discreta*. Estos programas se ejecutaron en el servidor *Ampere* y se utilizaron 96 hilos para la versión con OpenMP. Las especificaciones de este servidor se encuentran en la sección 3.5.

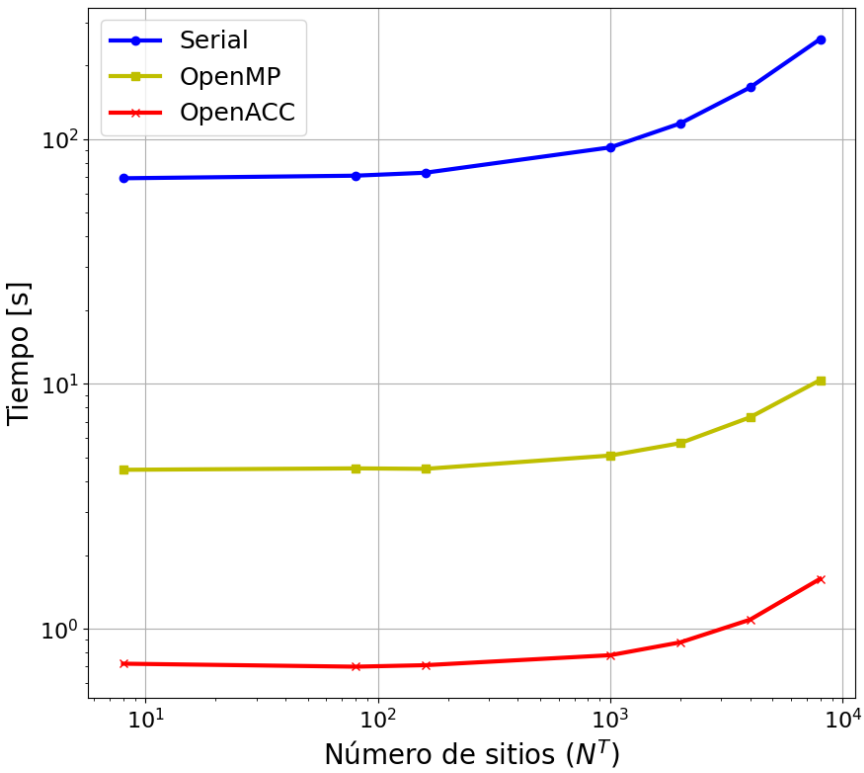

Figura 5.18: Tiempos de ejecución de la versión serial, con OpenMP y con OpenACC del programa *Electrolito alrededor de un macroion con carga discreta*. Estos programas se ejecutaron en el servidor *Ampere* y se utilizaron 96 hilos para la versión con OpenMP. Las especificaciones de este servidor se encuentran en la sección 3.5.

El programa que crea los sitios de carga del macroion mueve N^T partículas debido a las interacciones entre ellas, en cambio, este programa mueve N iones debido a las interacciones con los N^T sitios de carga del macroion más los N iones del electrolito, por lo que, al variar N^T, el número de partículas que se mueven es el mismo (N), causando un aumento en el tiempo menos acelerado que el del programa previo.

El tiempo de ejecución del programa con OpenACC es el mejor, siendo más de dos ordenes menor que el tiempo serial y poco menos de un orden respecto a la versión con OpenMP utilizando 96 hilos.

En otras palabras, mientras que el programa con OpenACC usando $N^T = 1000$ toma 6.5 horas en finalizar, el programa con OpenMP ocupa 42.5 horas, y la versión serial demanda poco más de 32 días.

5.2.3 Macroion con carga discreta uniforme

Este es el primer caso, el propósito de este es analizar la distribución de iones cuando el número de sitios de carga del macroion varía, siendo la valencia de cada sitio igual a la carga neta del macroion entre el número de sitios de carga ($z_s = Q_m/N^T$).

Los parámetros de este sistema, así como la representación visual de este (para $N^T = 1000$), fueron presentados en la sección anterior (sec. 5.2.2).

En particular, se van a revisar las configuraciones con $N^T = 1$, 2, 4, 8, 32, 80 y 10^4, donde $N^T = 1$ corresponde al sistema de validación del electrolito con $\sigma = 5.0$ Å, y las demás configuraciones pertenecen a este caso.

Densidad de carga por unidad de volumen de iones positivos y negativos [$\rho(r)$]:

En la siguiente figura (figura 5.19) se nota una clara diferencia en la concentración de iones positivos cercanos al macroion, entre las configuraciones con $N^T = 1$, 2, 4 y 8. Esto se debe a que la carga del macroion ($Q_m = -72$) está concentrada en una pequeña cantidad de sitios (N^T), haciéndolos más atractivos para los iones positivos del electrolito.

Las curvas de las configuraciones $N^T = 1$, 32, 80 y 10^4 parecen estar apiladas. Haciendo un acercamiento a la máxima concentración de iones positivos (figura 5.20), se observa que la curva para $N^T = 32$ sobresale de la curva, por lo que los sistemas con $N^T > 32$ se comportan igual que el sistema del macroion con carga uniforme no discreta ($N^T = 1$, figura 5.17).

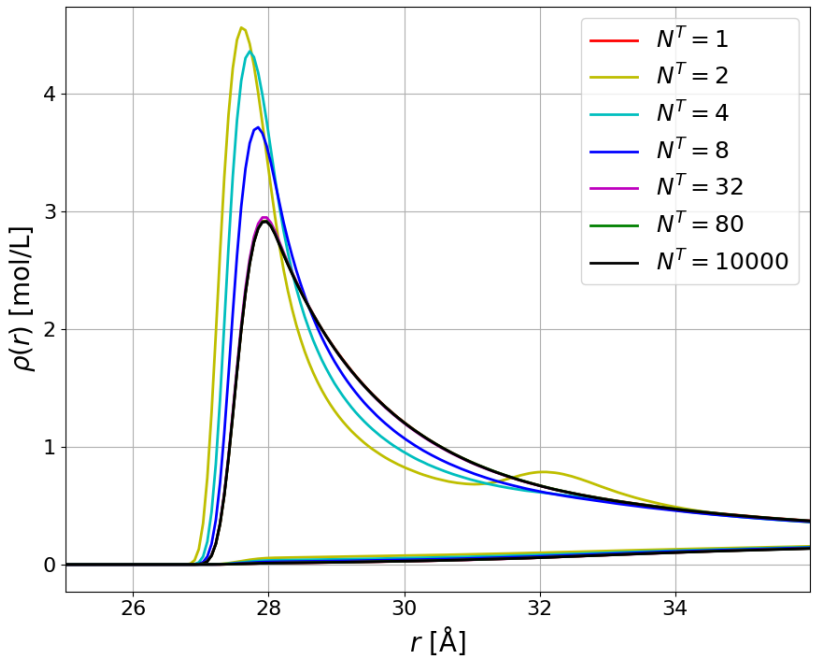

Figura 5.19: Densidad de carga por unidad de volumen de iones positivos (mayor concentración) y iones negativos (menor concentración) cercanos al macroion, para distintos N^T. Las curvas de los sistemas con $N^T = 1$, 32, 80 y 10^4 parecen estar superpuestas.

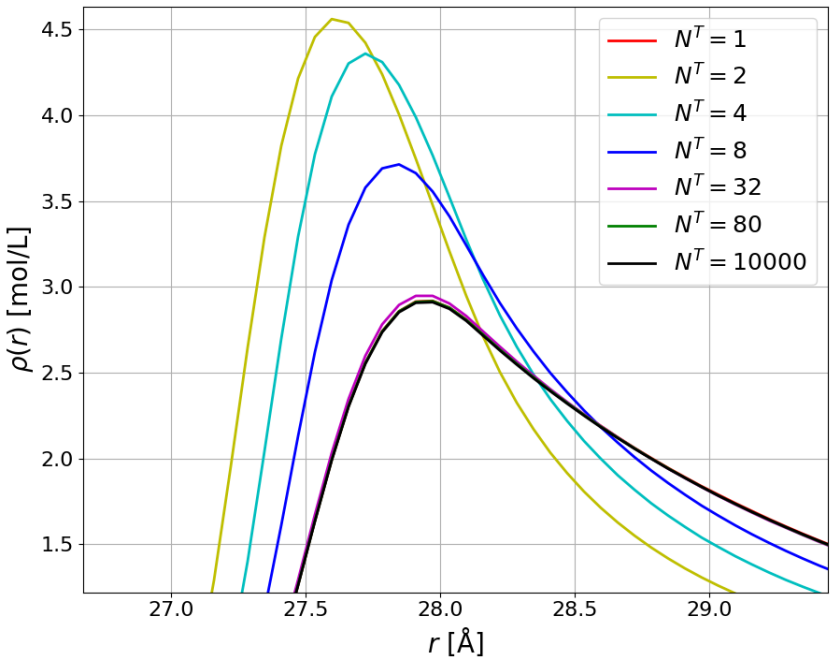

Figura 5.20: Acercamiento a las máximas densidades de carga por unidad de volumen de iones positivos, para distintos N^T. Las curvas de los sistemas con $N^T = 1$, 80 y 10^4 se encuentran apiladas.

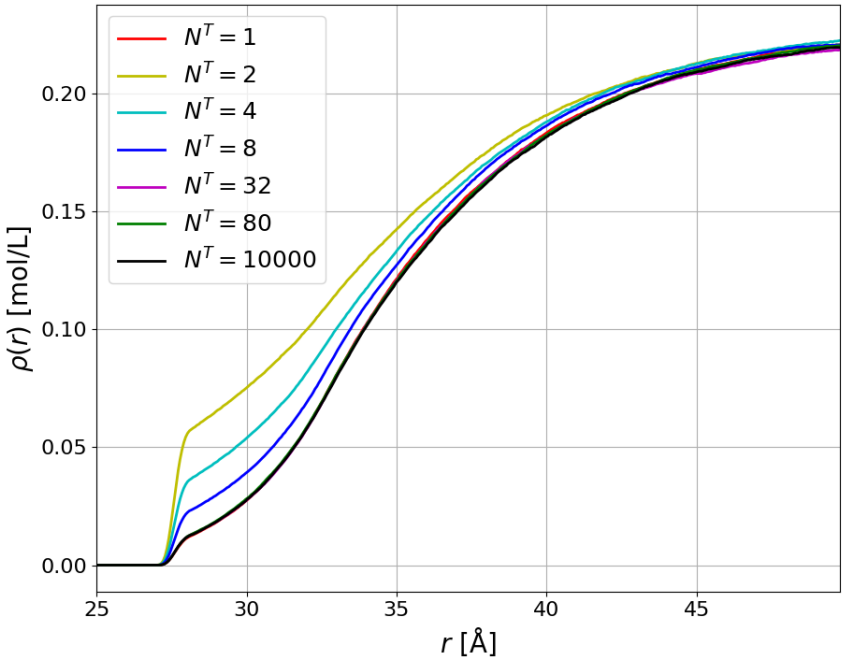

Figura 5.21: Densidad de carga por unidad de volumen de iones negativos cercanos al macroion, para distintos N^T. Las curvas de los sistemas con $N^T = 1$, 32, 80 y 10^4 parecen estar superpuestas.

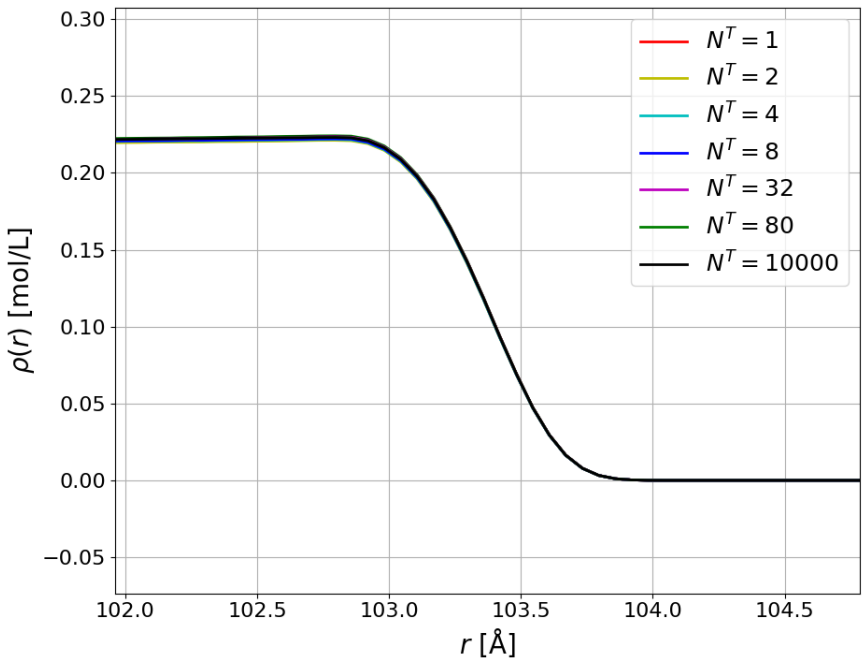

Figura 5.22: Densidad de carga por unidad de volumen de iones positivos y iones negativos cercanos al cascarón exterior, para distintos N^T. Las curvas de todos los sistemas parecen estar apiladas.

Si la densidad de iones positivos cerca del macroion cambia notoriamente entre los sistemas con $N^T < 32$, también lo hace la densidad de iones negativos, esto se puede ver en la figura 5.21.

De la figura 5.22, se puede ver que la densidad de carga por unidad de volumen de iones positivos y negativos, cercanos al cascarón exterior, no presenta un cambio notorio para las corridas con distinto N^T, por lo que la concentración de iones positivos y negativos lejanos del macroion es igual para cualquier N^T.

A diferencia de los sistemas con $N^T > 32$, la configuración con $N^T = 2$ no parece una esfera, la concentración de iones es diferente que la de los otros sistemas, creando un segundo máximo en la densidad por unidad de volumen de iones positivos, que se puede ver en la figura 5.19. Esta configuración se puede ver en la siguiente figura:

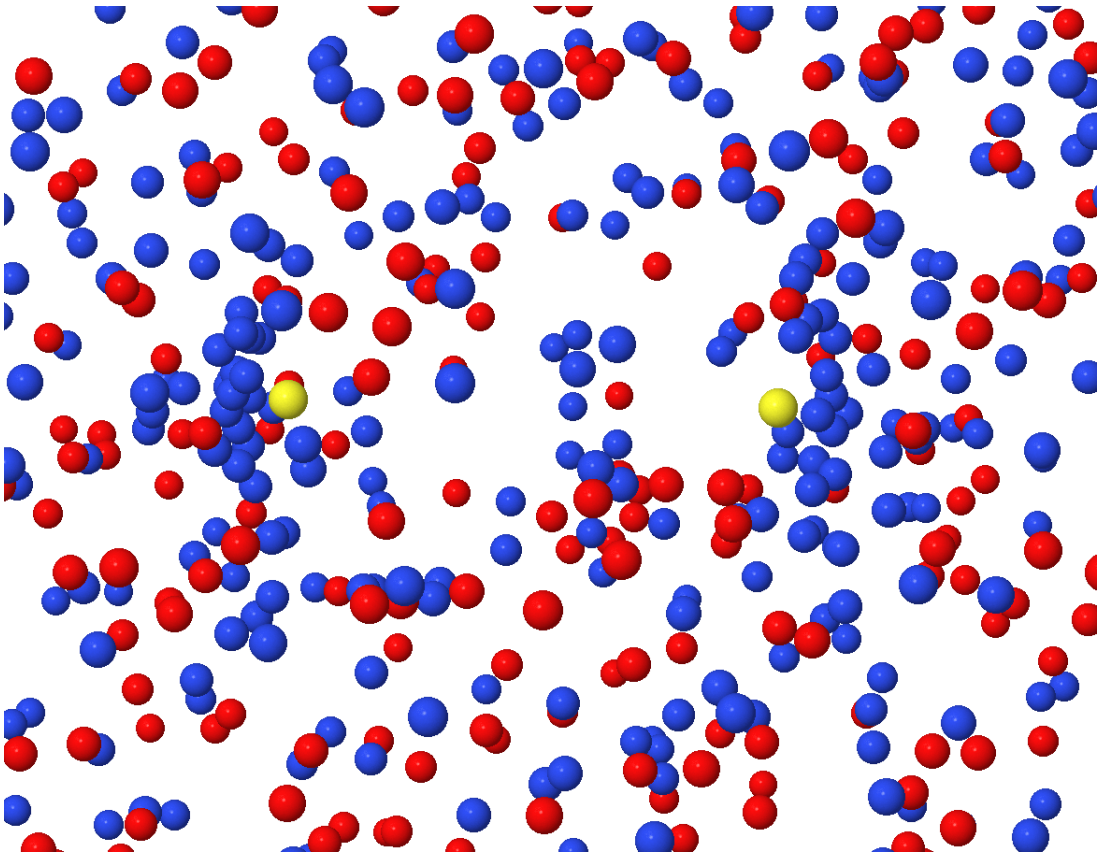

Figura 5.23: Electrolito alrededor de un macroion con carga discreta uniforme. La carga del macroion ($Q_m = -72$) está repartida en los N^T sitios de carga de este, de manera uniforme ($z_s = Q_m/N^T$). Las esferas amarillas representan los sitios de carga de macroion ($N^T = 2$), mientras que las esferas azules y rojas simbolizan los iones del electrolito ($N = 1272$), con valencias positivas ($z_{ion} = +1$) y negativas ($z_{ion} = -1$) respectivamente.

Carga integrada $[P(r)]$ y potencial eléctrico $[V(r)]$:

El segundo máximo de la densidad por unidad de volumen de iones positivos para $N^T = 2$, se aprecia también en la curva de la carga integrada de la figura 5.24. Este sistema se neutraliza a una distancia del macroion menor que las otras configuraciones.

El comportamiento de las demás configuraciones es el esperado de acuerdo a las figuras anteriores. La distinción entre las curvas de la carga integrada de las configuraciones con diferente N^T se encuentra cerca del macroion, para $r > 50$ Å esta no se nota.

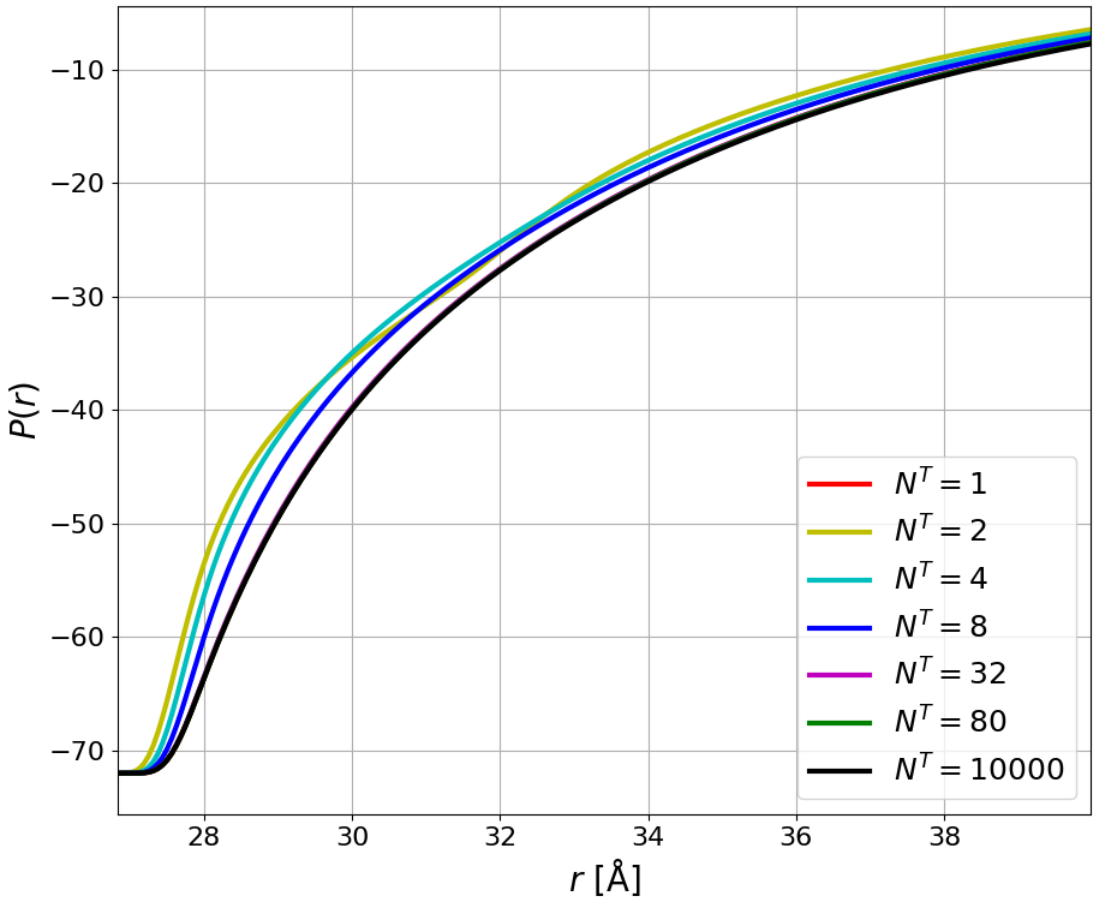

Figura 5.24: Carga integrada $[P(r)]$ cerca del macroion, para distintos N^T Las curvas de los sistemas con $N^T = 1$, 80 y 10^4 se encuentran apiladas.

En el potencial eléctrico, este segundo pico de la configuración $N^T = 2$ no se nota tanto, pero ahí está, cuando la curva de esta configuración se junta con la de $N^T = 4$ en un tramo pequeño (figura 5.25).

El comportamiento de los sistemas con $N^T > 32$ valida las interacciones del macroion

de carga discreta con el electrolito, ya que concuerda con el caso del macroion con carga uniforme no discreta (sec. 5.2.1).

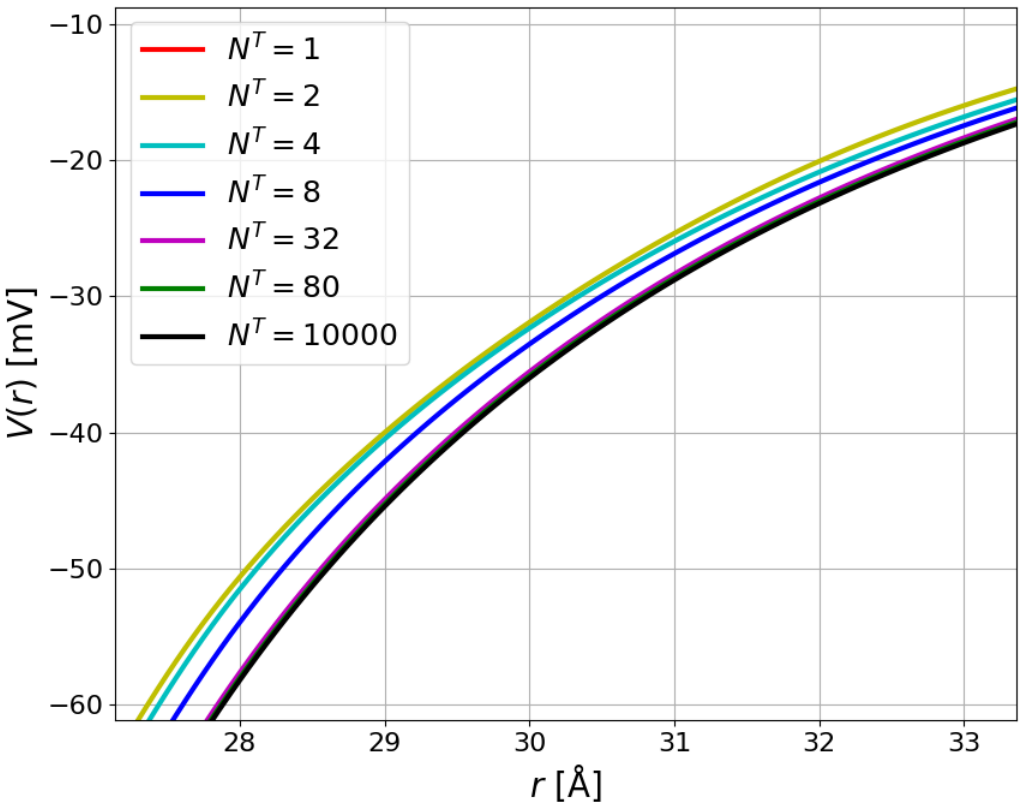

Figura 5.25: Potencial eléctrico $[V(r)]$ cerca del macroion, para distintos N^T Las curvas de los sistemas con $N^T = 1$, 80 y 10^4 se encuentran apiladas.

5.2.4 Macroion con carga discreta, compuesta por cargas positivas y negativas

Este es el segundo caso, el objetivo de este es ver cómo varía la distribución de los iones alrededor del macroion con carga discreta, compuesta por cargas positivas y negativas, utilizando distintos números de sitios de carga del macroion, con una misma carga neta ($Q_m = -100$). La representación de una configuración con este sistema se encuentra en la figura 5.26.

Los parámetros para este segundo caso son los siguientes:

- $\sigma = 5.0$ Å.
- Carga neta del macroion: $Q_m = -100$.

- Valencia de los iones: $z_{ion} = \pm 1$.
- Valencia de sitio del macroion: $z_s = \pm 1$.
- Constante dieléctrica: $\epsilon_r = 78.5$.
- Radio del macroion: $R_m = 25.0$ Å.
- Radio de los iones: $r_{ion} = 2.5$ Å.
- Radio de la última capa: $R_c = 106.0$ Å.
- Número de iones del electrolito: $N = 1272$.
- Temperatura absoluta del sistema: $T = 298.0$ K.
- Coeficiente de difusión del medio: $D = 10^{-12}\,\mathrm{m}^2/\mathrm{s}$.
- Tamaño de paso de la dinámica browniana: $\Delta t = 10^{-11}$ s.
- Número de pasos de la dinámica browniana: 3×10^8.

Las diferencia con el primer caso es la valencia de los sitios de carga del macroion y la carga neta de este. En este caso $z_s = \pm 1$ y $Q_m = -100$.

La distribución de las cargas positivas y negativas en los sitios del macroion es aleatoria, por lo que cada corrida de este sistema es única si $|Q_m|/N^T > 1$. Concretamente, se estudiaron las configuraciones con $N^T = 1$, 100, 1700, y 1800.

Para analizar estos sistemas se realizaron diez corridas para cada configuración y se promediaron los resultados. Las corridas tienen las mismas posiciones de los sitios de carga del macroion, pero diferente distribución de carga en estos.

El caso con $N^T = 1$ es igual que el sistema de validación, pero con diferente carga neta ($Q_m = -100$). Para $N^T = 100$, cada sitio de carga tiene valencia -1, por lo que es equivalente al sistema del macroion con carga discreta uniforme. Con $N^T = 1700$ hay 800 sitios positivos y 900 sitios negativos (figura 5.26), y la configuración con $N^T = 8100$ tiene 4000 sitios positivos y 4100 sitios negativos (figura 5.32).

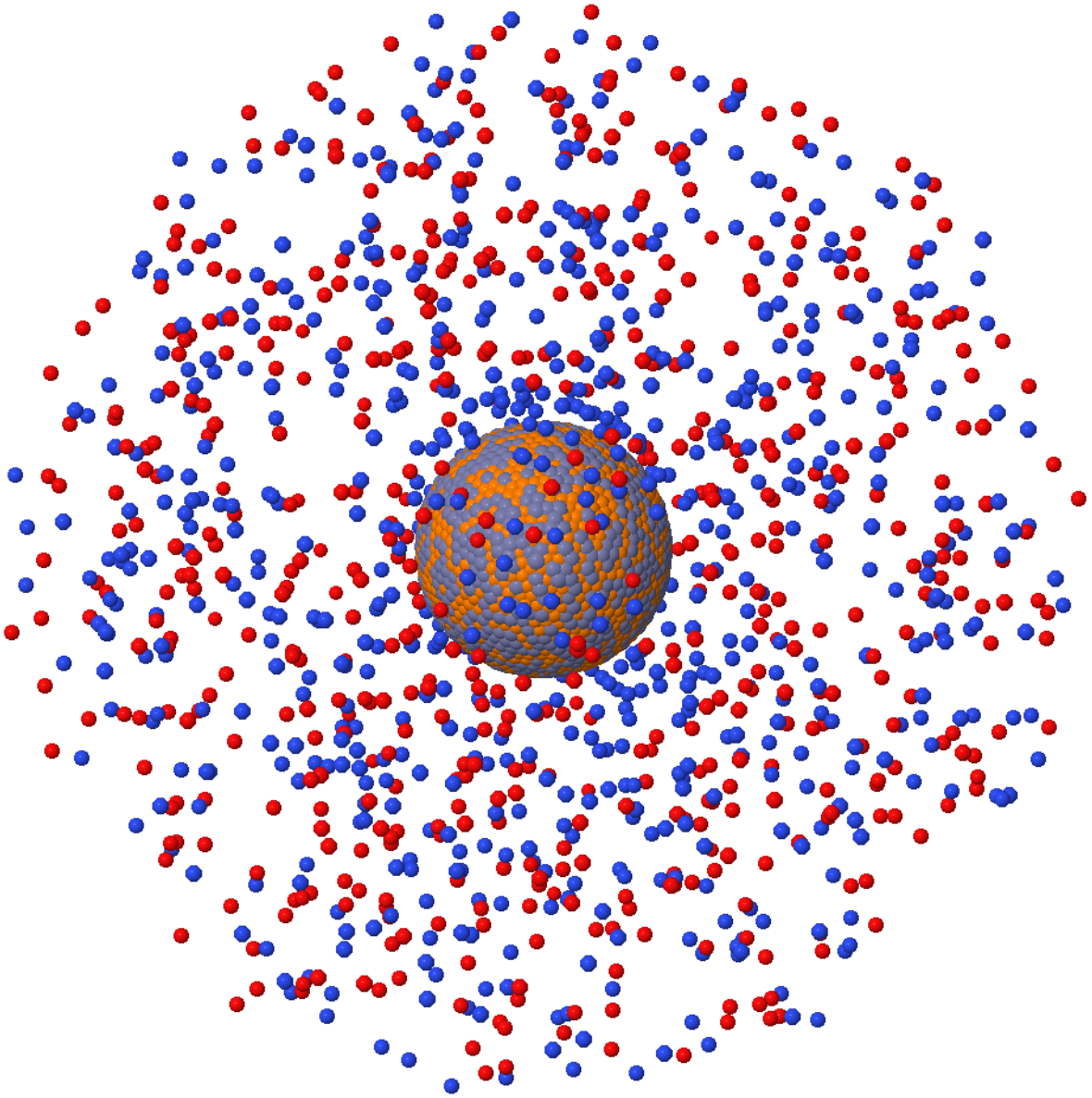

Figura 5.26: Electrolito alrededor de un macroion con carga discreta, compuesta por cargas positivas y negativas. La suma de las valencias ($z_s = \pm 1$) de los sitios de carga del macroion ($N^T = 1700$) es igual a la carga neta de este ($Q_m = -100$). Las esferas naranjas representan los sitios de carga de macroion con carga negativa ($N_-^T = 900$), y las celestes los positivos ($N_+^T = 800$). Mientras que las esferas azules y rojas simbolizan los iones del electrolito ($N = 1272$), con valencias positivas y negativas respectivamente.

Densidad de carga por unidad de volumen de iones positivos y negativos $[\rho(r)]$:

La siguiente figura muestra la densidad de carga por unidad de volumen de iones positivos cerca del macroion para las distintas configuraciones.

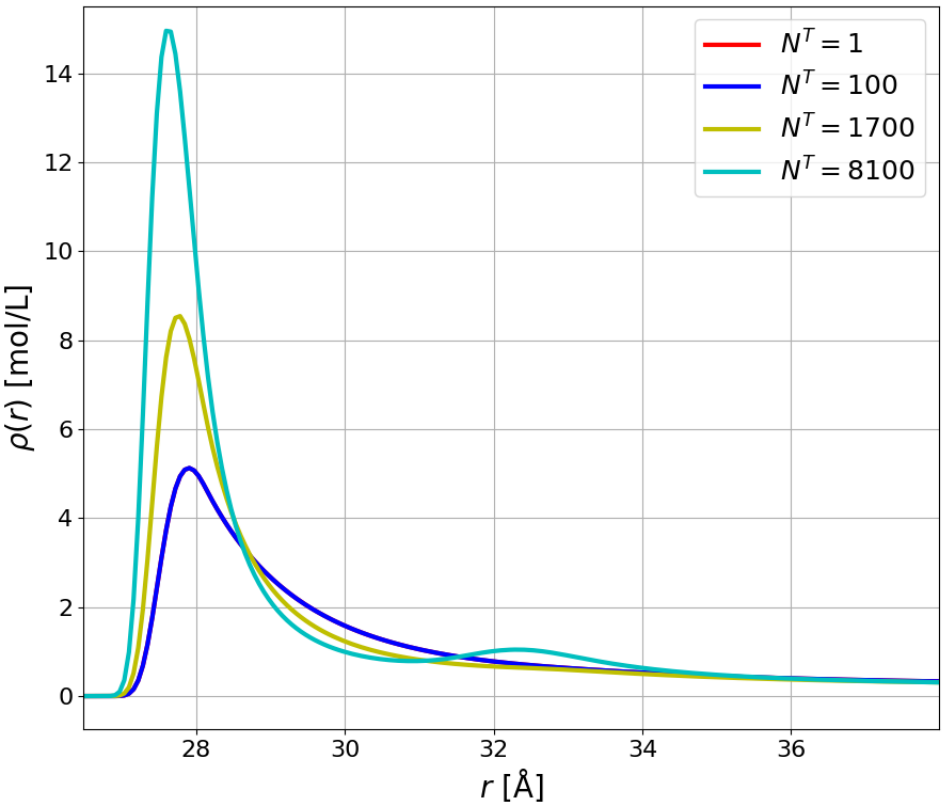

Figura 5.27: Densidad de carga por unidad de volumen de iones positivos cercanos al macroion, para distintos N^T. Las curvas de los sistemas con $N^T = 1$ y 100 se encuentran apiladas.

Se ve un cambio notable en la máxima densidad de iones positivos para las configuraciones con $N^T = 1700$ y 8100 respecto a los sistemas con $N^T = 1$ y 100. Las curvas de estos últimos están sobrepuestas, parecen tener el mismo comportamiento, tal como se esperaba de las pruebas anteriores.

Ahora viene un resultado relevante, la concentración de iones negativos cercanos al macroion (figura 5.28). A pesar de que la carga neta del macroion es $Q_m = -100$, la máxima concentración de iones negativos se encuentra cerca del macroion, para las configuraciones con $N^T = 1700$ y 8100. Esto se debe a que el macroion no está compuesto enteramente de cargas negativas, como en los sistemas previos.

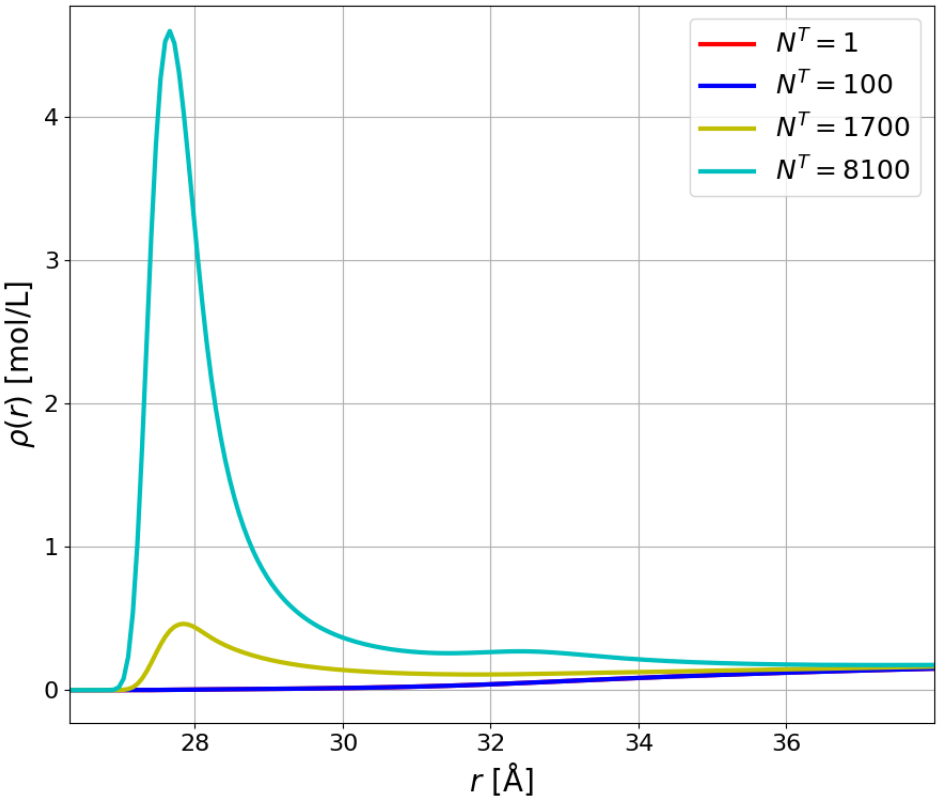

Figura 5.28: Densidad de carga por unidad de volumen de iones negativos cercanos al macroion, para distintos N^T. Las curvas de los sistemas con $N^T = 1$ y 100 se encuentran apiladas.

Por otro lado, las configuraciones con $N^T = 1$ y 100 tienen una densidad mínima de iones negativos cerca del macroion.

Las densidades de carga por unidad de volumen de iones positivos y negativos para la configuración con $N^T = 8100$, tienen un segundo máximo en $r = 32.35$ Å y $r = 32.41$ Å respectivamente. Pareciera que estos segundos máximos corresponden a los iones que son atraídos por los sitios de carga del macroion, pero repelidos por los iones alrededor de este.

La densidad de carga por unidad de volumen de iones positivos y negativos cercanos al cascarón exterior es la misma para cada configuración, y se ve claramente una diferencia entre las configuraciones corridas, ya que las máximas densidades para los sistemas con $N^T = 1700$ y 8100 se encuentran cerca del macroion. Por lo tanto, hay menos iones cercanos al cascarón exterior.

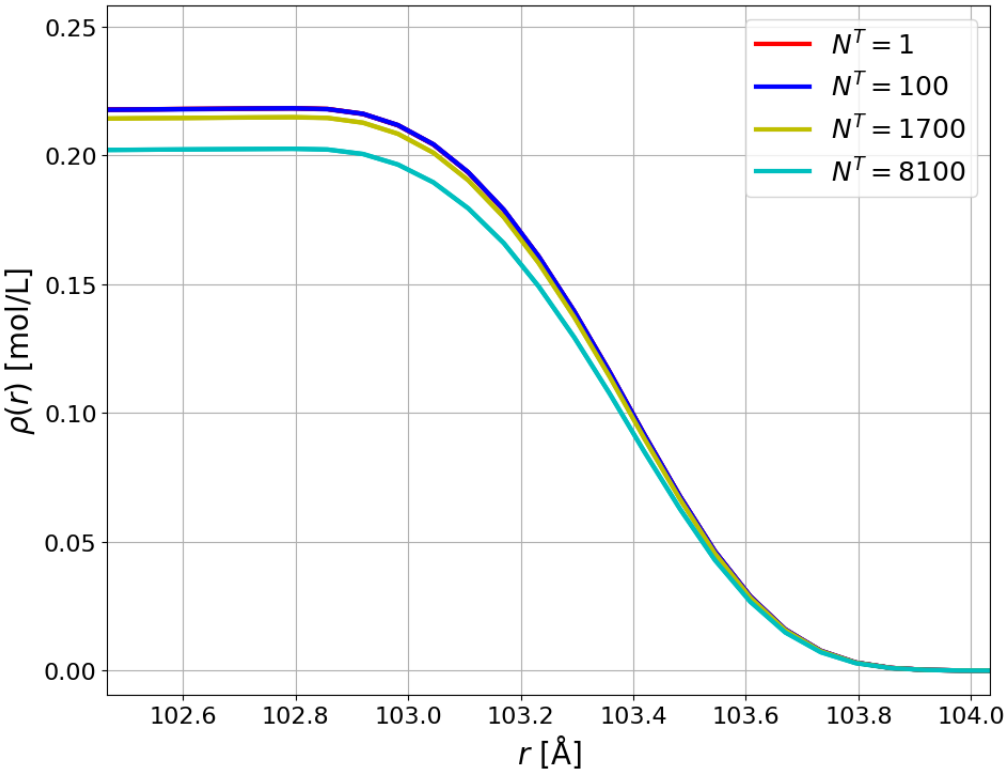

Figura 5.29: Densidad de carga por unidad de volumen de iones positivos y negativos cercanos al cascarón exterior, para distintos N^T. Las curvas de los sistemas con $N^T = 1$ y 100 se encuentran apiladas.

Carga integrada $[P(r)]$ y potencial eléctrico $[V(r)]$:

El cambio en las curvas de la carga integrada y del potencial eléctrico de las configuraciones con $N^T = 1700$ y 8100 respecto a los sistemas previos es sobresaliente (figuras 5.30 y 5.31).

Los segundos máximos de $\rho(r)$, de la configuración con $N^T = 8100$, se notan cuando la curva de la carga integrada atraviesa la curva del sistema con $N^T = 1700$. También cuando las curvas del potencial eléctrico, de las configuraciones mencionadas, se juntan y se separan antes de lo debido.

De cierta forma, el comportamiento del electrolito para la configuración con $N^T = 8100$ se asemeja al sistema con $N = 2$ del primer caso, teniendo un modo de actuar diferente al de los demás sistemas.

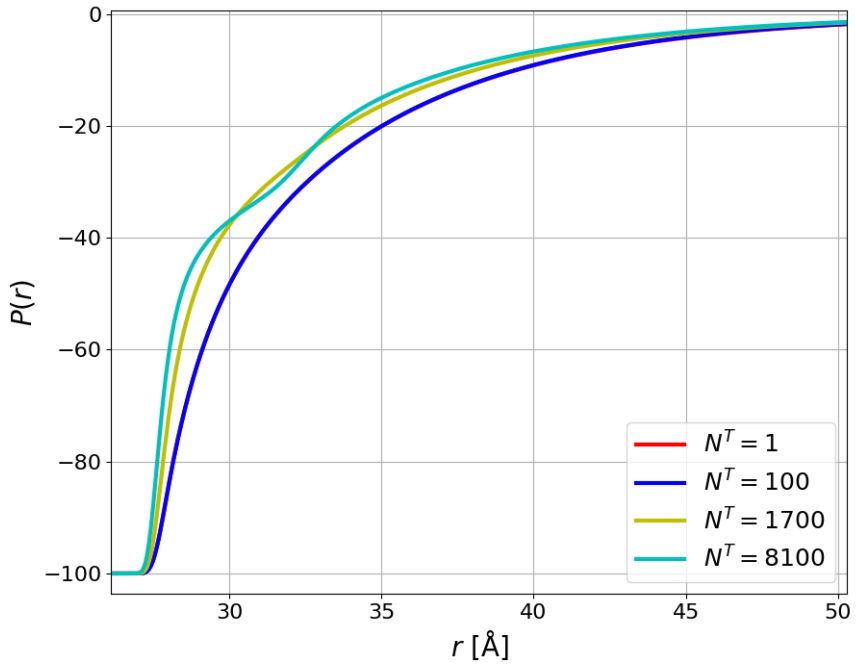

Figura 5.30: Carga integrada $[P(r)]$ cerca del macroion, para distintos N^T. Las curvas de los sistemas con $N^T = 1$ y 100 se encuentran apiladas.

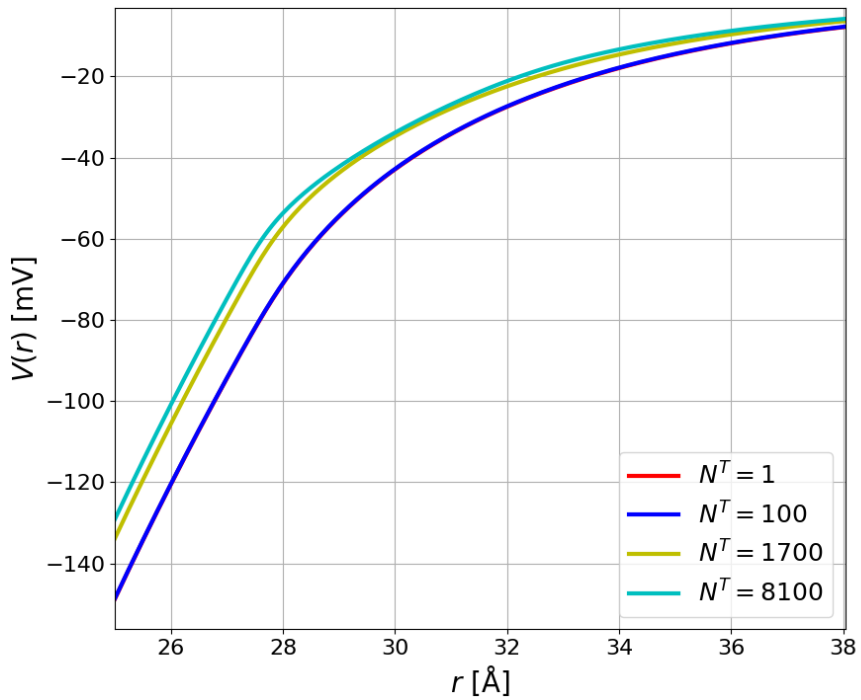

Figura 5.31: Potencial eléctrico $[V(r)]$ cerca del macroion, para distintos N^T. Las curvas de los sistemas con $N^T = 1$ y 100 se encuentran apiladas.

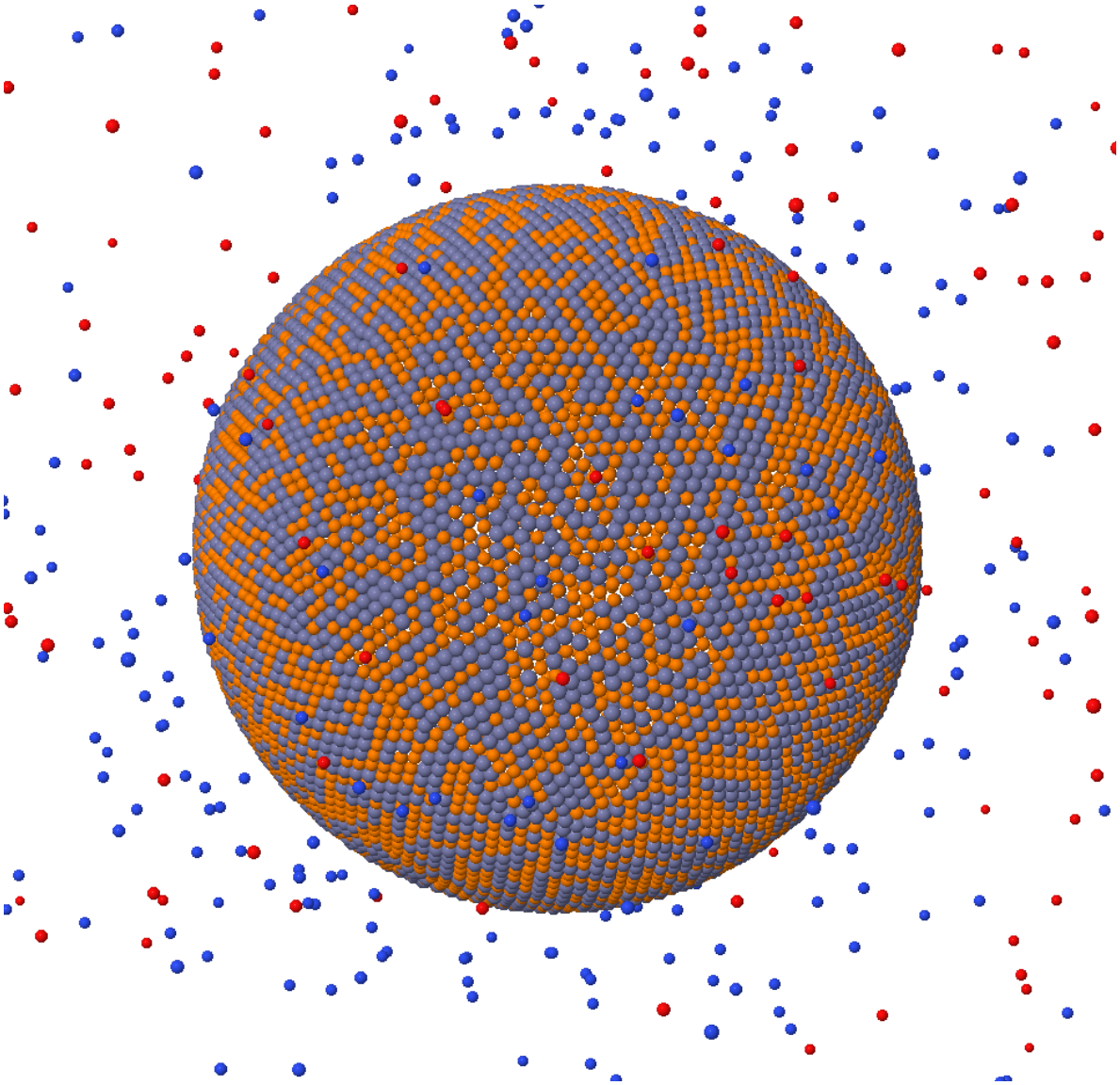

Figura 5.32: Electrolito alrededor de un macroion con carga discreta, compuesta por cargas positivas y negativas. La suma de las valencias ($z_s = \pm 1$) de los sitios de carga del macroion ($N^T = 8100$) es igual a la carga neta de este ($Q_m = -100$). Las esferas naranjas representan los sitios de carga de macroion con carga negativa ($N_-^T = 4100$), y las celestes los positivos ($N_+^T = 4000$). Mientras que las esferas azules y rojas simbolizan los iones del electrolito, con valencias positivas ($z_{ion} = +1$) y negativas ($z_{ion} = -1$) respectivamente.

CAPÍTULO 6

Conclusiones

En este trabajo se desarrollaron dos programas de simulación de dinámica browniana. El primero consiste en resolver el problema de Thomson, obteniendo un error porcentual menor a 1% para cualquier número de partículas, y se empleó el método de descenso de gradiente para conseguir un error menor a 10^{-3}% con respecto a los resultados de *The Cambridge Cluster Database* [36].

El segundo programa desarrollado es la simulación de la dinámica browniana de un electrolito confinado alrededor de un macroion con carga discreta, siendo la configuración de este último un resultado de la primera simulación. Para el confinamiento del electrolito se añadieron fuerzas de potencial de núcleo repulsivo en la dinámica browniana. Se analizaron dos casos en concreto, cuando el macroion de carga discreta tiene un solo tipo de carga repartida de manera uniforme en los sitios de carga, y cuando el macroion de carga discreta esta compuesto por cargas negativas y positivas, variando el número de sitios de carga que componen al macroion en ambos casos.

Los programas fueron optimizados y paralelizados con directivas de OpenMP y OpenACC, logrando simulaciones en tiempos de hasta dos ordenes de magnitud más bajos que la versión serial. Esto permitió un análisis detallado de la distribución de iones y su comportamiento en diferentes escenarios. Las diferentes versiones del segundo programa (serial, con OpenMP y con OpenACC) fueron validadas utilizando corridas de Monte Carlo, obteniendo errores casi imperceptibles.

Para el caso en el que el macroion con carga discreta tiene carga distribuida de manera uniforme en los sitios que lo componen, se llegó a la conclusión de que si se quiere simular el comportamiento de un macroion con carga uniforme, se debe utilizar un mínimo número de sitios de carga, que depende de la carga neta del macroion.

En el segundo caso, cuando el macroion de carga discreta esta compuesto por cargas negativas y positivas, la distribución de iones alrededor del macroion es diferente si el número de sitios varía entre simulaciones, aún teniendo la misma carga neta.

Los resultados conseguidos respaldan de manera sólida los objetivos planteados al

inicio de la tesis. La paralelización de los programas de simulación ha sido una inversión valiosa, permitiendo realizar análisis exhaustivos y obtener conclusiones significativas en un tiempo conveniente.

La principal aportación de este trabajo son los algoritmos de las implementaciones desarrolladas (seriales, con OpenMP y con OpenACC), para las cuales se realizaron pruebas de eficiencia y eficacia. Estos algoritmos brindan la oportunidad de analizar otros sistemas complejos que requieren un cálculo y análisis intensivo en un tiempo conveniente.

El macroion con carga discreta puede ser utilizado en la investigación de diversos sistemas coloidales y partículas cargadas, permitiendo realizar simulaciones más realistas.

Los resultados obtenidos para el electrolito confinado alrededor de un macroion con carga discreta, compuesta por cargas positivas y negativas, permitirán profundizar en el estudio de las interacciones con otros fluidos y sustancias.

El perfilado de la ejecución de la simulación paralelizada con OpenACC, utilizando la herramienta *nvproof*, fue fundamental para quitar los cuellos de botella del programa. Los tiempos de ejecución de las versiones paralelizadas podrían ser mejorados utilizando nuevas herramientas de perfilado.

Para simulaciones que requieran un cálculo más exhaustivo, se podría implementar la paralelización con múltiples GPUs, distribuyendo la carga de trabajo en más núcleos, obteniendo tiempos de ejecución aún más bajos.

El desarrollo de este trabajo ha sido una experiencia significativa en mi trayecto hacia la culminación de mis estudios en física. A lo largo de este proceso, he enfrentado desafíos emocionantes y he aprendido lecciones valiosas que me han permitido crecer tanto en conocimientos como en habilidades. Sumergirme en la simulación de partículas y la paralelización de programas ha ampliado mi comprensión de la física y del estudio de sistemas complejos, brindándome una perspectiva valiosa.

Cada obstáculo superado ha sido una oportunidad para crecer y demostrar mi perseverancia. Agradezco sinceramente el apoyo recibido, el cual ha sido invaluable en este proceso. La experiencia adquirida a través de esta trabajo se convierte en una base sólida para mi desarrollo profesional, y estoy emocionado por los desafíos futuros que me esperan en este apasionante campo científico.

Este logro representa un punto de partida hacia nuevas metas y contribuciones en el campo de la física. Estoy decidido a seguir explorando y participando en proyectos que impulsen el avance científico y el entendimiento de nuestro mundo.

Referencias bibliográficas

[1] Boda, D. (2014). *Monte Carlo Simulation of Electrolyte Solutions in Biology*. Annual Reports in Computational Chemistry, 127–163. doi:10.1016/b978-0-444-63378-1.00005-7.

[2] Corni, I., Ryan, M. P., & Boccaccini, A. R. (2008). *Electrophoretic deposition: From traditional ceramics to nanotechnology*. Journal of the European Ceramic Society, 28(7), 1353–1367. doi:10.1016/j.jeurceramsoc.2007.12.011.

[3] Yu, Y.-X., Wu, J., & Gao, G.-H. (2004). *Density-functional theory of spherical electric double layers and ζ potentials of colloidal particles in restricted-primitive-model electrolyte solutions*. The Journal of Chemical Physics, 120(15), 7223–7233. doi:10.1063/1.1676121.

[4] Liu, Z. y Jia, Y. (2012). *Two Simulation Methods of Brownian Motion*. Journal of Physics: Conference Series, 2012, 012015. https://doi.org/10.1088/1742-6596/2012/1/012015.

[5] Ermak, D. L. (1975). *A computer simulation of charged particles in solution. I. Technique and equilibrium properties*. The Journal of Chemical Physics, 62(10), 4189–4196. doi:10.1063/1.430300.

[6] Purcell, E. M. & Morin, D. J. (2013). *Electricity and magnetism*. Third edition. Cambridge University Press.

[7] Arfken, G. B. (1985). *Mathematical Methods for Physicists* (3rd ed.). Academic Press.

[8] Allen, M.P., & Tildesley, D.J. (1987). *Computer Simulation of Liquids*. Oxford University Press.

[9] Steinbrecher, T., Mobley, D. L., & Case, D. A. (2007). *Nonlinear scaling schemes for Lennard-Jones interactions in free energy calculations*. The Journal of Chemical Physics, 127(21), 214108. doi:10.1063/1.2799191.

[10] Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge University Press.

[11] Reitz, J. R. & Milford, F. J. (1960). *Foundations of Electromagnetic Theory*. Addison-Wesley. (pp. 24).

[12] Guerrero-García, G. I., González-Tovar, E., & Olvera de la Cruz, M. (2010). *Effects of the ionic size-asymmetry around a charged nanoparticle: unequal charge neutralization and electrostatic screening*. Soft Matter, 6(9), 2056. doi:10.1039/b924438g.

[13] Russel, W. B., Saville, D. A., & Schowalter, W. R. (1989). *Colloidal Dispersions*. Cambridge University Press.

[14] Doyle, P. S., & Underhill, P. T. (2005). *Brownian Dynamics Simulations of Polymers and Soft Matter*. Handbook of Materials Modeling, 2619–2630. doi:10.1007/978-1-4020-3286-8_140.

[15] OpenMP Architecture Review Board. (2018). *OpenMP Frequently Asked Questions*. Recuperado el 26 de abril de 2023, de https://www.openmp.org/about/openmp-faq/#WhatIs.

[16] New Mexico State University. (2021). *Introduction to MPI*. Recuperado el 26 de abril de 2023, de https://hpc.nmsu.edu/discovery/mpi/introduction/.

[17] OpenACC. (2019). *OpenACC specification*. Recuperado el 26 de abril de 2023, de https://www.openacc.org/specification/.

[18] NVIDIA. (2023). *CUDA Zone*. Recuperado el 26 de abril de 2023, de https://developer.nvidia.com/cuda-zone.

[19] NVIDIA. (2023). *OpenACC Overview*. Recuperado el 26 de abril de 2023, de https://developer.nvidia.com/openacc/overview.

[20] Barton, J. J., & Nackman, L. R. (1994). *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley Professional.

[21] Microsoft Corporation. (2023). *Visual Studio Code Documentation*. Recuperado el 27 de abril de 2023, de https://code.visualstudio.com/docs.

[22] Lippman, S. B., Lajoie, J., & Moo, B. (2012). *C++ primer* (5th ed.). Addison-Wesley Professional.

[23] Robey, R. & Zamora, Y. (2021). *Parallel and High Performance Computing*. Manning.

[24] cplusplus.com. (2023). *random - C++*. Recuperado el 27 de abril de 2023, de https://cplusplus.com/reference/random/.

[25] Intel Corporation. (2021). *Intel C++ Compiler Developer Guide and Reference*. Recuperado el 27 de abril de 2023, de https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/overview.html.

[26] NVIDIA Corporation. (2021). *NVIDIA HPC SDK Compilers User Guide*. Recuperado el 27 de abril de 2023, de https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html.

[27] AMD. (2021). *Ryzen Threadripper PRO*. Recuperado el 8 de mayo de 2023, de https://www.amd.com/es/processors/ryzen-threadripper-pro.

[28] Abramkina, O., Dubois, R., & Véry, T. (2022). *OpenACC for GPU: an introduction*. Recuperado en febrero de 2023, de http://www.idris.fr/media/formations/openacc/openacc_gpu_idris_fortran.pdf.

[29] Burtch, K. O. (2004). *Linux Shell Scripting with Bash*. Sams Publishing.

[30] SSH. (2023). *SSH Academy*. Recuperado el 9 de mayo de 2023, de https://www.ssh.com/academy/ssh.

[31] Intel Corporation. (2022). *Alphabetical option list*. Recuperado en diciembre de 2022, de https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/alphabetical-option-list.html.

[32] NVIDIA Corporation. (2021). *CUDA Profiler User's Guide*. Recuperado en marzo de 2022, de https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[33] TechPowerUp. (2023). *GPU Specs Database*. Recuperado el 9 de mayo de 2023, de https://www.techpowerup.com/gpu-specs/.

[34] Erber, T., & Hockney, G. M. (1991). *Equilibrium configurations of N equal charges on a sphere*. Journal of Physics A: Mathematical and General, 24(23), L1369–L1377. doi:10.1088/0305-4470/24/23/008.

[35] Wales, D. J., & Ulker, S. (2006). *Structure and Dynamics of Spherical Crystals Characterised for the Thomson Problem*. Physical Review B, 74(21), 212101.

[36] Wales, D. J., Mackay, H., & Altshuler, E. L. (2009). *Lowest minima located for the Thomson problem*. Physical Review B, 79(22), 224115. Recuperado de https://www-wales.ch.cam.ac.uk/~wales/CCD/Thomson2/table.html