UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PARALELIZACIÓN EN CUDA Y VALIDACIÓN DE CORRECCIÓN DE TRASLAPES EN SISTEMA DE PARTÍCULAS COLOIDALES

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FRANCISCO JAVIER CARTER ARAYA

PROFESOR GUÍA:
RODRIGO SOTO BERTRÁN

MIEMBROS DE LA COMISIÓN:
NANCY HITSCHFELD KAHLER
PATRICIO POBLETE OLIVARES

SANTIAGO DE CHILE
2016

Financiado parcialmente por el proyecto FONDECYT #1140778

Resumen

La simulación de cuerpos que interactúan entre sí por medio de fuerzas y la detección de colisiones entre cuerpos son problemas estudiados en distintas áreas, como astrofísica, fisicoquímica y videojuegos. Un campo en particular corresponde al estudio de los coloides, partículas microscópicas suspendidas sobre otra sustancia y que tienen aplicaciones en distintas industrias. El problema consiste en simular la evolución de un sistema con distintos tipos de partículas coloidales a través del tiempo, cumpliendo las propiedades de volumen excluido, movimiento aleatorio y condiciones de borde periódicas. Además, la interacción de largo alcance entre coloides presenta la particularidad de no cumplir con el principio de acción y reacción.

Se desarrolló un algoritmo de simulación completamente paralelo en GPU, implementado en la plataforma CUDA y C++. La solución utiliza una triangulación de Delaunay en memoria de la tarjeta gráfica para conocer eficientemente la vecindad de cada partícula, lo que permite resolver traslapes entre partículas sin tener que evaluar todo el sistema. Se utilizó una implementación reciente del algoritmo de *edge-flip* para mantener la triangulación actualizada en cada paso de tiempo, extendiendo además el algoritmo para corregir los triángulos invertidos. Para el caso de fuerzas de corto alcance, además se desarrolló un algoritmo paralelo que construye y utiliza listas de Verlet para manejar las vecindades entre partículas de forma más eficiente que la implementación anterior.

Los resultados obtenidos con la implementación paralela presentan una mejora de hasta dos órdenes de magnitud con respecto al tiempo de la solución secuencial existente. Por otro lado, el algoritmo para fuerza de corto alcance mejora de igual magnitud con respecto a la solución de largo alcance desarrollada. También se verificó que la corrección de traslapes con triangulación de Delaunay se hace de forma eficiente, y que esta estructura puede ser aplicada para otros problemas relacionados, como implementar el cálculo de fuerzas de corto alcance (y compararlo con la implementación ya existente) o realizar simulaciones aproximadas utilizando triangulaciones.

Dedicado a mi mamá, papá, hermana y hermano, quienes me apoyaron en todo momento de este largo -y en momentos difícil- recorrido.

Agradecimientos

Se agradece a la profesora Nancy Hitschfeld y al profesor Rodrigo Soto, quienes guiaron este trabajo desde su muy temprana concepción hasta su finalización. La experiencia enriquecedora adquirida durante todo este periodo se debe en gran parte a su apoyo.

También se agradece a Cristóbal Navarro por su orientación y consejos durante el desarrollo del trabajo, además de proveer las librerías con las que se crearon parte de las figuras presentadas. El algoritmo que desarrolló es una de las componentes fundamentales que hacen posible este trabajo.

Finalmente, se agradece al *NVIDIA GPU Research Center* del Departamento de Ciencias de la Computación (DCC) de la Universidad de Chile, por facilitar los equipos con los que se realizaron los experimentos presentados en este documento.

Francisco Javier Carter Araya

Tabla de contenido

Resumen I

Agradecimientos III

1. Introducción 1

1.1. Contexto 1

1.2. Resumen 2

1.3. Objetivos 3

1.4. Contenidos 4

2. Marco teórico 5

2.1. Conceptos 5

2.1.1. Simulación de n-body 5

2.1.2. Corrección de traslapes 6

2.2. Algoritmos existentes 7

2.2.1. N-body paralelo 7

2.2.2. Simulación de Barnes-Hut 7

2.2.3. Lista de Verlet 8

2.2.4. Dinámica guiada por eventos 8

2.2.5. Corrección de traslapes 9

2.3. Tecnologías 9

2.3.1. CUDA 9

3. Especificación del problema **13**

3.1. Descripción 13

3.1.1. Interacción entre partículas 13

3.1.2. Volúmen excluido 14

3.1.3. Condiciones de borde periódicas 15

3.1.4. Ruido aleatorio 15

3.2. Relevancia de la solución 17

3.3. Requisitos de la solución 17

3.4. Características deseadas 18

3.5. Trabajo relacionado 18

4. Especificación de la solución **20**

4.1. Elección de la solución 20

4.1.1. Triangulación de Delaunay 22

4.2. Diseño de clases 23

4.3. Estructuras de datos 25

4.3.1. Estructuras para ejecución paralela 25

4.3.2. Constantes de simulación 26

4.4. Algoritmos 26

4.4.1. Generación de la configuración inicial 27

4.4.2. Integración para fuerza de largo alcance 27

4.4.3. Actualización de la triangulación 29

4.4.4. Corrección de triángulos invertidos 30

4.4.5. Corrección de traslapes 33

4.4.6. Integración para fuerza de corto alcance 35

4.5. Interfaz de usuario 36

5. Validación de la solución **37**

5.1. Experimento 37

5.2. Resultados 39

5.2.1. Simulación de sistemas de partículas 39

5.2.2. Localidad de corrección de traslapes 41

5.2.3. Gráficos 41

5.3. Análisis 48

6. Conclusiones **50**

6.1. Resumen 50

6.2. Recuento de objetivos 51

6.3. Aprendizaje 52

6.4. Trabajo futuro 52

Apéndices **54**

A . Tablas 54

Bibliografía **57**

Índice de tablas

5.1. Configuraciones usadas en las pruebas, en donde cada configuración está identificada por un dígito y tiene dos tipos de partículas. ϕ_i corresponde a la fracción de partículas del tipo i, α_i, μ_i son valores utilizados en el cálculo de fuerzas y ρ es la fracción de volúmen del sistema ocupado por partículas. 38

6.1. Tiempos de ejecución promedio en milisegundos de un paso de tiempo para fuerza de largo alcance. N es la cantidad de partículas simuladas y $C_0 - C_4$ corresponden a las configuraciones de cada sistema, descritas en la tabla 5.1. 54

6.2. Cantidad de traslapes promedio por paso de tiempo para fuerza de largo alcance, con los mismos parámetros que en la tabla 6.1. 55

6.3. Cantidad de *edge-flips* promedio por paso de tiempo para fuerza de largo alcance, con los mismos parámetros que en la tabla 6.1. 55

6.4. Tiempos de ejecución promedio en milisegundos por paso de tiempo para fuerza de corto alcance, con los mismos parámetros que en la tabla 6.1. 55

6.5. Cantidad de traslapes promedio por paso de tiempo para fuerza de corto alcance, con los mismos parámetros que en la tabla 6.1. 56

6.6. Comparación de los tiempos de ejecución promedio en milisengundos entre los distintos métodos de n-body cuadrático. Los tiempos del algoritmo secuencial no se midieron para todos los valores de N para no detener por un tiempo prolongado el servidor remoto en el que se realizaron las pruebas. 56

Índice de ilustraciones

2.1. Esquema de memoria en CUDA. 11

2.2. Ejecución divergente de un *warp*. Los *threads* en azul que satisfacen una misma condición se ejecutan en paralelo, mientras que los *threads* en rojo que no la cumplen permanecen detenidos hasta terminar esa rama de la condición (las barras negras). Al final de la divergencia, el *warp* continúa la ejecución en paralelo. 12

3.1. Influencia de la distancia en el cálculo para la fuerza de corto alcance entre partículas. A partir del radio de corte r_{cutoff} marcado en magenta, la expresión para la fuerza no se evalúa y el valor de f_{ij} se considera 0. 14

3.2. Distribución normal. Las bandas indican las probabilidades respectivas de que un valor generado con la distribución normal esté comprendido en ese rango. 16

3.3. Formas de eliminar valores extremos generados aleatoriamente. 16

3.4. Lista de celdas, en donde cada celda tiene una lista de partículas que residen en ella. La circunferencia de radio r_s que rodea a p_3 corresponde a la piel de esa partícula. Luego, p_4 y p_8 forman parte de la lista de vecinos de p_3. 18

4.1. Distintas configuraciones de partículas distribuidas en el área de simulación, para $N = 896$ y $dt = 0.01$. Los números de cada configuración denotan a las descritas en la tabla 5.1. Los colores denotan distintos tipos de partículas. 21

4.2. Movimiento aleatorio de una partícula durante un paso de tiempo. 22

4.3. Triangulación de Delaunay en 2 dimensiones bajo condiciones de borde periódicas. Las líneas corresponden a arcos de la triangulación. 24

4.4. Posiciones iniciales de las partículas del sistema en $t = 0$ para $N = 896$. Ambas configuraciones corresponden a las descritas en la tabla 5.1. Los colores de las partículas indican su tipo. 27

4.5. Detección de movimiento inválido. El vector azul $\vec{r_{bc}}^{\,0}$ corresponde a la distancia entre partículas. Después del desplazamiento, el vector en rojo $\vec{r_{bc}}^{\,1}$ cambia su sentido general con respecto a la situación inicial, indicando que hubo un traslape más fuerte entre ambas partículas. 29

4.6. Resolución de triángulo invertido. El polígono que se invirtió producto del desplazamiento de la partícula d está sombreado en color cian. 31

4.7. Resolución de triángulo invertido con corrección de dos arcos. 31

4.8. Detección de tríangulo invertido utilizando coordenadas baricéntricas. 32

4.9. Corrección de traslapes. Al aplicar el desplazamiento neto sobre B producto de A y C, puede ocurrir que los traslapes no queden resueltos. 34

4.10. Corrección de traslapes en paralelo. El desplazamiento sobre C producto de los traslapes con A y B es mayor que el necesario, por lo que se trunca su magnitud. 35

5.1. Sistemas de partículas tras 10000 iteraciones con $N = 4096$ y $dt = 0.01$. Los colores de las partículas denotan el tipo al que pertenecen. Las configuraciones de cada sistema y los valores respectivos para cada tipo de partículas corresponden a los descritos en la tabla 5.1. 40

5.2. Gráficos de localidad para corrección de traslapes tras 10000 iteraciones con $dt = 0.01$. Las partículas verdes participaron en al menos una corrección de traslapes en un mismo paso de tiempo, mientras que las rojas no lo hicieron. Las configuraciones de cada sistema corresponden a las descritas en la tabla 5.1. 42

5.3. Gráficos de localidad de corrección de traslapes para $dt = 0.001$, en donde los demás parámetros son los mismos que en la figura 5.2. 43

5.4. Gráficos de localidad de corrección de traslapes para $dt = 0.0001$, en donde los demás parámetros son los mismos que en la figura 5.2. 44

5.5. Tiempos de ejecución en milisegundos para los algoritmos de largo y corto alcance. Las configuraciones de cada sistema y los valores respectivos para cada tipo de partículas corresponden a los descritos en la tabla 5.1. 44

5.6. Influencia de las distintas configuraciones sobre el tiempo de ejecución para simular un paso de tiempo físico, con los mismos parámetros que los utilizados en la figura 5.5. 45

5.7. Promedio de iteraciones por paso de tiempo para fuerzas de largo y corto alcance, con los mismos parámetros que en la figura 5.5. 45

5.8. Promedio de iteraciones de *edge-flips* por paso de tiempo para algoritmo de fuerza de largo alcance, con los mismos parámetros que en la figura 5.5. 46

5.9. Triangulación de Delaunay con $N = 4096$ para la configuración 4 descrita en la tabla 5.1. 47

5.10. Comparación entre los tiempos de ejecución en milisegundos para las implementaciónes del método *n-body* de $O(N^2)$, utilizando la configuración C_0 descrita en la tabla 5.1. 47

Capítulo 1

Introducción

Este capítulo describe el contexto en el que se origina el problema a resolver, además de los objetivos que se pretende cumplir con su resolución. Luego, se hace un resumen de los contenidos abordados en el trabajo, con sus respectivas secciones en donde se trata cada tema en detalle.

1.1. Contexto

La simulación de partículas es un problema activo en computación, ya que los experimentos científicos se están realizando sobre volúmenes de datos cada vez más grandes. Las partículas se pueden modelar de distintas formas dependiendo del área: en astronomía se simulan cúmulos de galaxias, mientras que en biotecnología se representan los enlaces entre distintos tipos de moléculas. Para esta situación en particular, se estudió la simulación de sistemas de partículas coloidales [18], caracterizadas por tener diámetros del orden de micrones ($1\mu m = 1 \times 10^{-6} m$) y estar suspendidas en fluidos. Algunas áreas de aplicación para estos sistemas son:

Cosméticos: Las cremas corresponden a un sistema coloidal de partículas de grasa o plástico suspendidas en agua, alcohol o aceite. En este caso, las partículas ayudan a aumentar la viscosidad o cremosidad de las emulsiones.

Pinturas: El color de una pintura es determinado por partículas suspendidas en agua si se trata de latex, o en aceite para el caso del óleo.

Alimentos: Al igual que en el caso de las cremas, a la mayonesa, leche y otras emulsiones se les puede agregar partículas coloidales para modificar su viscosidad o prevenir la solidificación,

entre otras propiedades.

Minería: Los relaves mineros se componen de partículas de distinto tamaño suspendidas en agua. La acidez del agua modifica las interacciones entre partículas, lo que puede hacer que el relave se vuelva muy viscoso, y por lo tanto, caro de transportar.

En todos estos casos, las propiedades del producto final dependen de las interacciones entre partículas coloidales, lo que crea la necesidad de simular dichas interacciones para encontrar configuraciones deseables que permitan generar nuevos avances. Típicamente, mediante físicoquímica se pueden conocer las fuerzas que ejercen los coloides entre sí, las cuales consisten de atracciones y repulsiones que pueden ser de corto o largo alcance. Luego, el objetivo final es saber cómo se comporta el sistema completo bajo las propiedades mencionadas y ciertas configuraciones, todo esto para cantidades de partículas cada vez mayores.

Existe una implementación secuencial que permite realizar simulaciones de interacciones entre partículas coloidales, pero no es eficiente para números grandes de partículas. Existe una variedad de algoritmos relacionados que abordan simulaciones de galaxias o moléculas y se ajustan parcialmente al problema estudiado, pero existen diferencias que no hacen deseable implementar directamente estas soluciones.

A diferencia de las simulaciones astrofísicas, los coloides presentan volúmenes excluidos que dan origen a colisiones y traslapes, algo que se tiene en común con los experimentos de fisicoqúimica. Por otro lado, la existencia de movimientos aleatorio en las partículas hace que las colisiones no puedan ser predichas como en el caso de las moléculas. Por esto, se propone un método novedoso para resolver el problema utilizando la estructura de triangulación de Delaunay, lo cual puede servir como punto de partida para desarrollar nuevas soluciones que se adapten completamente a este tipo de simulaciones.

1.2. Resumen

Para este trabajo se investigaron algoritmos que permitan realizar simulaciones de partículas, tales como la simulación de *n-body* y la detección de colisiones. En ambos casos se priorizó la investigación de soluciones paralelas en GPU, pues ya existe una implementación de ese tipo adaptadas a sistemas de partículas coloidales, pero que solo sirve para simular interacciones de corto alcance.

Por lo tanto, se necesita una solución que aborde el problema de simular cualquier tipo de fuerza, que puede ser combinación de interacciones de corto y largo alcance.

Posteriormente, se identificaron las propiedades que deben cumplir los sistemas a simular, tales como la ecuación de integración de las partículas y condiciones de borde periódicas del sistema. Estas características permitieron descartar varias de las soluciones estudiadas anteriormente, pues no eran adaptables a las necesidades específicas del problema. Por lo tanto, se decidió implementar una solución nueva utilizando la triangulación de Delaunay, que permite resolver traslapes entre partículas de forma eficiente. A partir del análisis de este algoritmo, se pudo detectar una heurística que permite simular correctamente pasos de tiempo más grandes, la cual también se aplicó en la implementación para la fuerza de corto alcance.

Para validar esta solución, se realizaron experimentos para medir el rendimiento de los distintos algoritmos desarrollados que componen la simulación. Además se generaron visualizaciones que permitieron comprobar o explicar los supuestos generados a partir de los resultados obtenidos en los experimentos. Considerando todo lo anterior, se pudo concluir que el uso la de triangulación de Delaunay permite resolver el problema de corrección de traslapes y es posible aplicar la estructura tanto al cálculo de fuerzas de corto alcance como para fuerzas de largo alcance aproximadas.

1.3. Objetivos

El objetivo general de este trabajo consiste en diseñar e implementar un algoritmo paralelo en GPU para simular interacciones (de corto y largo alcance) entre partículas y corregir traslapes entre ellas, visualizar y validar los resultados obtenidos, además de compararlos con las soluciones previas ya implementadas. Este objetivo se puede descomponer en los siguientes objetivos específicos:

1. Investigar la literatura actual para decidir una solución paralela en GPU que resuelva la corrección de traslapes más eficientemente que la implementación secuencial existente.
2. Diseñar e implementar un algoritmo en base a la solución elegida.
3. Verificar que la corrección de traslapes implementada resuelve todas las colisiones entre partículas (o la mayor cantidad posible, según alguna medida de tolerancia) en cada paso de tiempo.

4. Realizar pruebas con configuraciones reales y comparar los tiempos de ejecución con la versión secuencial y la solución paralela existente, para las fuerzas de largo y corto alcance.

5. Evaluar la escalabilidad del algoritmo sobre el número de partículas simuladas con respecto al tiempo de ejecución.

1.4. Contenidos

La sección 2 muestra los conceptos involucrados para entender el problema, las soluciones existentes y las tecnologías involucradas en su implementación. En particular, se describe la arquitectura del *hardware* para computación paralela, utilizada tanto en algunas alternativas revisadas como en la solución final implementada. La sección 3 describe las características particulares que distinguen a este tipo de simulaciones con respecto a otras alternativas abordadas por los algoritmos mencionados en la sección anterior. También se mencionan los requisitos que debe cumplir el programa para considerar que cumple con el problema presentado, así como los estándares de calidad deseados.

En la sección 4 se detallan las razones que llevaron a elegir la solución implementada, así como los detalles de la implementación. Se describen en detalle los algoritmos en los cuales se descompone la simulación, las estructuras de datos, clases y librerías utilizadas. La sección 5 corresponde a los experimentos realizados para validar que la solución cumple efectivamente con el problema presentado, incluyendo la metodología, hardware y métricas utilizados. A partir de los resultados obtenidos, se realiza el análisis para validar o refutar los supuestos que llevaron a la creación de esta solución.

La sección 6 hace un recuento de los objetivos enunciados anteriormente para verificar si fueron cumplidos o no, además de realizar las reflexiones sobre el trabajo realizado y plantear posibles trabajos futuros a partir de esta solución.

Capítulo 2

Marco teórico

Este capítulo corresponde a la descripción de los conceptos involucrados, algoritmos estudiados y tecnologías utilizadas que permiten entender la solución desarrollada y diferenciarla con respecto a las alternativas ya disponibles. En particular, se describe la plataforma de computación paralela utilizada para implementar la solución y varios de los algoritmos mostrados en este capítulo.

2.1. Conceptos

El problema abordado se puede separar en dos procesos principales ejecutados en secuencia uno después del otro. Estos subproblemas consisten en integrar el sistema de partículas y resolver las inconsistencias físicas producidas por el paso de integración. Para este último también se describe el problema de detección de colisiones, al ser suficientemente similar a la corrección de traslapes y aparecer en la mayoría de las alternativas estudiadas.

2.1.1. Simulación de n-body

Definición (Simulación de n-body): Dada una lista de cuerpos $P = p_1, p_2, ..., p_N$ en un espacio k-dimensional, obtener la interacción F_i sobre cada cuerpo por efecto de los $N - 1$ cuerpos en el sistema.

Este problema se expresa frecuentemente en términos de la atracción gravitacional entre galaxias [11], pero también se pueden simular fuerzas intermoleculares u otro tipo de interacciones microscópicas.

La fuerza F simulada típicamente depende de la distancia r_{ij} entre dos cuerpos. Si $F \sim r^{-n}$, se dice que la fuerza es de corto alcance si $n > d$, donde d es la dimensión del espacio simulado. Por el contrario, si $n \leq d$, se dice que F es de largo alcance.

Dependiendo del comportamiento específico de la fuerza simulada se pueden desarrollar optimizaciones, como por ejemplo particionar el espacio para agrupar cuerpos que estén cercanos entre sí.

2.1.2. Corrección de traslapes

Antes de describir la corrección de traslapes, se define primero el problema similar de detección de colisiones en un sistema de cuerpos:

Definición (Detección de colisiones): Dada una lista de cuerpos $P = p_1, p_2, ..., p_N$ y un paso de tiempo t, obtener la lista con los pares de cuerpos cuyos volúmenes se intersectan entre sí, y ejecutar una acción en el paso de tiempo $t + 1$ por cada cuerpo en colisión.

La acción ejecutada en $t + 1$ depende de la dinámica propia de cada simulación. Por ejemplo, se puede alejar los cuerpos entre sí, o actualizar la velocidad de los elementos involucrados por efecto de colisiones elásticas o inelásticas.

A partir de esto, se define el problema de corrección de traslapes:

Definición (Corrección de traslapes): Dada una lista de cuerpos $P = p_1, p_2, ..., p_N$ y un paso de tiempo t, obtener la lista con los pares de cuerpos cuyos volúmenes se intersectan entre sí, y separar cada par de cuerpos en colisión de manera que no existan pares $\in P$ que cumplan esta condición.

En la detección de colisiones, se ejecuta una serie de acciones que puede resolver los traslapes entre partículas en el siguiente paso de tiempo o evitarlos en pasos posteriores. Por otro lado, en la corrección de traslapes se busca que el sistema quede libre de intersecciones entre partículas en el mismo paso de tiempo, lo que implica calcular una lista de desplazamientos que debe aplicarse a cada partícula para corregir el sistema sin producir traslapes adicionales.

2.2. Algoritmos existentes

A continuación se muestran los algoritmos para resolver simulaciones de $n-body$ y corrección de traslapes, los cuales se analizaron antes y durante el desarrollo de la solución desarrollada. También se estudiaron soluciones para la detección de colisiones aún cuando no se utiliza directamente en el problema a resolver, con el objetivo de analizar la factibilidad de adaptarlas a la corrección de traslapes.

2.2.1. N-body paralelo

Una solución paralela en CUDA provista por NVIDIA [14] asigna una partícula por unidad de trabajo paralelo, y luego particiona la entrada en bloques de partículas que son cargados en un tipo de memoria de acceso más rápido que la memoria global en la GPU. Debido a que cada unidad de trabajo accede una vez a esta memoria compartida para computar su interacción con la partícula almacenada ahí, se logra reducir la latencia en los accesos de lectura, mejorando el rendimiento del algoritmo con respecto a la implementación de fuerza bruta.

Esta solución corresponde a una paralelización del algoritmo cuadrático para $n-body$, por lo que existen soluciones más eficientes como las mostradas a continuación. Sin embargo, ésta sirve como referencia sobre las formas de diseñar algoritmos paralelos que utilicen y exploten correctamente el potencial de las tarjetas gráficas.

2.2.2. Simulación de Barnes-Hut

Este algoritmo $O(nlogn)$ [1] aproxima la simulación de *n-body* a partir del hecho que las partículas suficientemente lejanas no necesitan ser evaluadas con tanta precisión, por lo que se puede aproximar un grupo de partículas lejanas como un solo objeto para efectos del cálculo de fuerzas. Para agrupar las partículas se particiona el espacio a través de un quad-tree (o *kd-tree* en k dimensiones), que es recorrido al momento de calcular las fuerzas.

Si una partícula interactúa con un nodo del árbol suficientemente alejado de ella, entonces se evalúa la fuerza considerando una partícula representante de todos los objetos contenidos en el nodo y sus niveles inferiores. En cambio, si la partícula y el nodo están cerca, se sigue hacia abajo en el árbol y se repite el procedimiento hasta llegar a las hojas, que contienen las partículas reales.

Existen implementaciones paralelas en GPU [4] y [2], las cuales se ejecutan completamente en la tarjeta gráfica), paralelizando tanto la construcción como el recorrido del árbol que se hace durante el cálculo de interacciones.

2.2.3. Lista de Verlet

Este algoritmo [21] se utiliza si la fuerza simulada tiene un radio de alcance r_c limitado, tras el cual su valor es despreciable y se considera como nulo. Consiste en crear listas de vecinos por cada cuerpo, considerando como vecino a cualquier otro elemento que esté a una distancia $r^* \leq r_c$, en donde $r*$ es la distancia entre partículas.

La ventaja de esta lista es que la lista de vecinos no necesita calcularse en cada paso de tiempo, ya que los vecinos de una partícula siguen siendo los mismos hasta que algún objeto en el sistema se haya desplazado suficientemente como para entrar o salir de alguna lista. En tal caso, se deben recalcular las vecindades de todas las partículas. Una estrategia conservadora consiste en evaluar los desplazamientos de cada partícula con respecto a su posición respectiva en el instante que la lista fue creada por última vez. Si existe un desplazamiento $\Delta r_{max} \geq \frac{r^* - r_c}{2}$, entonces se recalcula la lista de vecinos y se guardan las posiciones de cada partícula en este instante.

La implementación paralela para GPU en [10] y [15] considera una lista maestra con los $O(N^2)$ pares posibles de partículas, la cual se evalúa con el predicado que indica si las partículas son vecinas entre sí. La lista se ordena por llave-valor para dejar al comienzo todos los elementos cuyo predicado es verdadero, lo que permite copiar todos esos pares a la lista de vecinos que finalmente se utiliza en el cálculo de las fuerzas.

2.2.4. Dinámica guiada por eventos

La simulación de EDMD (*Event-Driven Molecular Dynamics*) es un algoritmo de detección de colisiones en el que no se actualizan todas las partículas en un paso de tiempo global, sino que el sistema va avanzando por sub-intervalos a medida que van ocurriendo eventos de colisión entre partículas. Antes de avanzar un paso de tiempo $t + \Delta t$, se computan los $0 \leq k \leq \binom{N}{2}$ pares de partículas que hayan participado en traslapes, lo cual se traduce en k colisiones que ocurrieron durante el paso de tiempo. Una vez obtenidos los tiempos $t_0, t_1, ..., t_k$, se ordenan de mayor a menor para que el sistema vaya avanzando en pasos de tiempo parciales, hasta llegar a $t + \Delta t$.

La implementación en GPU [7] (para una o múltiples tarjetas gráficas) ejecuta iterativamente la función que encuentra el mayor tiempo (contando desde $t + \Delta t$ hacia atrás) que el par de partículas involucradas en ese traslape debe retroceder hasta el instante t^* de la colisión. Una vez procesada la colisión, se integra ese par nuevamente hasta $t + \Delta t$ con sus velocidades actualizadas producto de la colisión, lo que puede producir nuevos traslapes que son agregados a la cola de eventos.

2.2.5. Corrección de traslapes

Debido a la dificultad de obtener la lista de desplazamientos que corrija todos los traslapes y no produzca colisiones entre pares que no estén en la lista inicial, la corrección se puede realizar en pasos iterativos hasta que no existan traslapes. Una solución secuencial consiste en hacer un recorrido $O(N^2)$ de todos los pares de partículas, corrigiendo los pares traslapados a medida que son detectados. El algoritmo termina cuando se recorren todos los pares de cuerpos y no se detectan traslapes en ninguno.

El recorrido de todos los pares de partículas es muy ineficiente, ya que la mayor parte corresponde a partículas lejanas que no tienen posibilidad de traslaparse entre sí. Al particionar el espacio en celdas, se puede reducir la verificación a comparar el cuerpo con los integrantes de su misma celda y los cuerpos de las celdas vecinas. Para fuerzas de corto alcance, se puede aprovechar la lista de vecinos definida en el algoritmo de Verlet (sección 2.2.3) para verificar traslapes de un cuerpo con los miembros de su lista de vecinos respectiva.

2.3. Tecnologías

Ya que tanto las alternativas estudiadas como la solución desarrollada están diseñadas para ser ejecutadas paralelamente en tarjetas gráficas (GPU), se analizó una de las plataformas que permite implementar estos algoritmos, y que fue la utilizada para la solución desarrollada.

2.3.1. CUDA

CUDA (*Compute Unified Device Architecture*) es al mismo tiempo una plataforma de computación y modelo de programación desarrollado por NVIDIA [6]. La plataforma permite implementar código en los lenguajes C y C++ a través de extensiones manejadas por el *driver* de compilador

`nvcc`, pero además existen extensiones para lenguajes como Python, Matlab y R, entre otros.

Modelo de programación

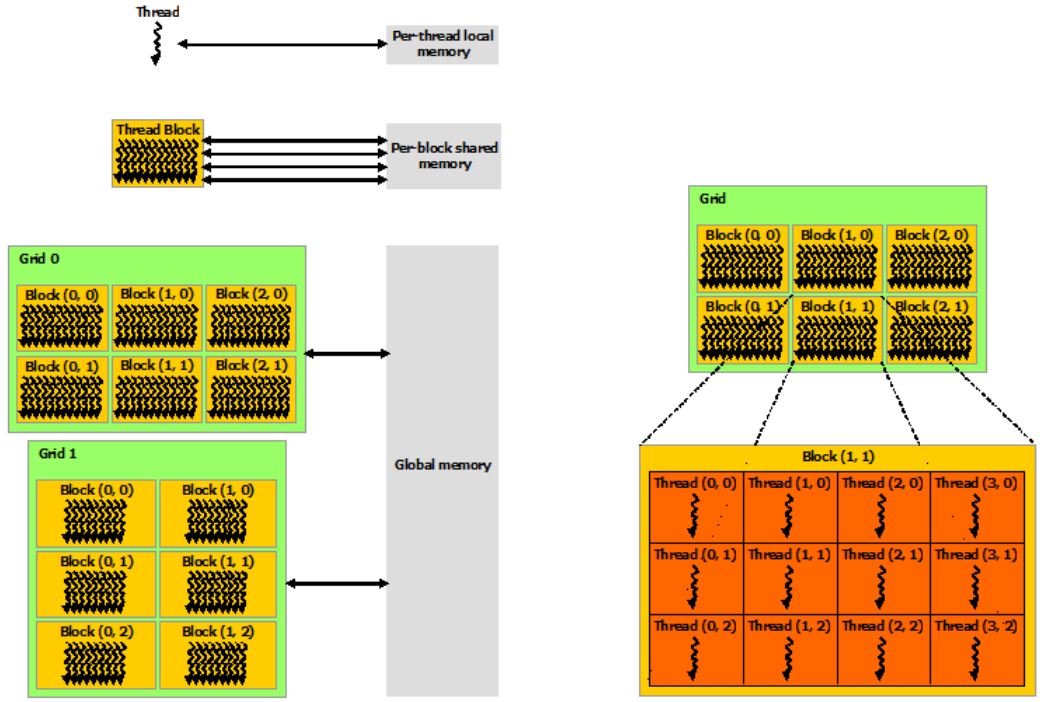
El modelo de programación en CUDA permite escribir código generable en paralelo a través de una jerarquía de estructuras, lo que permite generar código de manera independiente a las especificaciones de cada tarjeta gráfica. Las unidades de trabajo en paralelo se agrupan entre sí para formar bloques, los cuales a su vez forman una grilla que corresponde al parámetro de la función paralelizada a ejecutar.

Kernels: Corresponden a funciones C que se ejecuta múltiples veces en una misma llamada, al contrario de las funciones convencionales que se ejecutan una sola vez. La cantidad de veces que se ejecuta un *kernel* depende del tamaño de la grilla que se le entrega como parámetro, la cual está compuesta de bloques que a su vez están compuestos de *threads*. De este modo, la grilla puede ser lineal, cuadrática o cúbica con respecto a sus dimensiones, lo que permite adaptar el código a distintos esquemas de indexamiento según lo requiera el problema en cuestión.

Bloques: De manera similar a la grilla, los elementos de un bloque pueden ser indexados en 1, 2 o 3 dimensiones según el esquema de indexamiento adecuado al problema en cuestión. Ya que las unidades de trabajo que componen un bloque residen en el mismo multiprocesador, su tamaño queda restringido a 1024 elementos para las GPU actuales, pero no siempre se obtiene un mejor rendimiento con una cantidad mayor de *threads* por bloque. Una regla general es que la cantidad de bloques en el kernel debe ser a lo menos igual a la cantidad de multiprocesadores en la GPU, lo que también puede variar según el tamaño de cada bloque.

Threads: Son las unidades básicas de trabajo paralelo, las cuales se encargan de ejecutar el código implementado en los *kernels*. Cada *thread* tiene un espacio de memoria local aislado del resto de sus demás pares, por lo que la comunicación directa entre elementos (es decir, sin lectura/escritura en memoria de la GPU) no es posible en todos los casos. El esquema de asignación de *threads* en un problema es una decisión de diseño del algoritmo, pero generalmente se hace de manera que el número de threads sea grande y el trabajo realizado por cada uno sea pequeño. Esto permite aprovechar al máximo el paralelismo de la GPU, que está especializada en ejecutar un gran volúmen

de instrucciones.



(a) Jerarquía de memoria. (b) Grilla de bloques bidimensionales.

Figura 2.1: Esquema de memoria en CUDA.

Arquitectura:

La arquitectura de CUDA corresponde a SIMT (*Single Instruction, Multiple Threads*), similar a la arquitectura SIMD (*Single Thread, Multiple Data*). Esta última requiere combinar los accesos en una misma instrucción, mientras que SIMT permite escribir código a nivel de *threads*, los cuales pueden diverger independientemente unos de otros.

Warps: Un conjunto de 32 *threads* paralelos (para las tarjetas gráficas actuales) se agrupa como *warp*. Un *warp* ejecuta una instrucción común al mismo tiempo en paralelo, por lo que la eficiencia es óptima cuando todos los *threads* mantienen coherencia en el código que ejecutan y no existe divergencia durante el flujo de ejecución. Este es un factor importante al momento de implementar algoritmos en CUDA, ya que la presencia de bifurcaciones en el código (típicamente en forma de `if - else`) puede provocar que la ejecución se serialize entre los threads que tomaron una u otra rama respectivamente. Este fenómeno se llama **divergencia de warp**, y puede ralentizar un programa de manera importante según el tamaño de los *branches*, la cantidad de ramificaciones y la frecuencia con que ocurren.

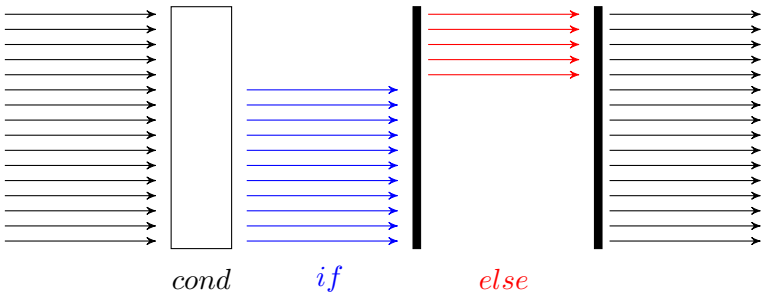


Figura 2.2: Ejecución divergente de un *warp*. Los *threads* en azul que satisfacen una misma condición se ejecutan en paralelo, mientras que los *threads* en rojo que no la cumplen permanecen detenidos hasta terminar esa rama de la condición (las barras negras). Al final de la divergencia, el *warp* continúa la ejecución en paralelo.

Modelo de memoria

Debido al nivel de especialización que han adquirido a través del tiempo, las tarjetas gráficas presentan distintos tipos de memorias con latencias específicas. Por ejemplo, cada multiprocesador de la GPU tiene su propio chip de memoria dedicada, mientras que la tarjeta como conjunto dispone de un espacio general accesible desde sus subunidades.

Memoria Global: Corresponde al espacio general de memoria en la GPU, el cual se inicializa y libera típicamente con versiones especiales de `malloc` y `free`, mientras que la transferencia de datos con la memoria principal se hace con la función `memcpy`. Si un *warp* accede a direcciones contiguas en memoria global, los accesos de cada *thread* se combinan en una o más transacciones equivalentes. Este factor es clave al intentar reducir el tiempo utilizado en accesos a memoria.

Memoria Constante: Corresponde a un caché especial de la GPU. Si los distintos *threads* solicitan la misma dirección en memoria constante, entonces se obtiene el resultado al costo de un éxito en caché. De lo contrario, el costo de acceso es igual al de memoria global.

Memoria Compartida: Es propia a un mismo bloque, y todos los *threads* que lo componen pueden acceder a ella. La cantidad máxima de memoria compartida está limitada por el *hardware*, debido a la misma razón que las dimensiones de un bloque lo están. El acceso a memoria compartida típicamente es más rápido que la memoria global en un factor de $\times 10 - 30$.

Capítulo 3

Especificación del problema

A continuación se describen las propiedades de este problema en particular, las cuales permiten caracterizarlo con respecto a otros tipos de simulaciones para sistemas de cuerpos. Además, se especifican las carácteristicas que debe cumplir la solución desarrollada para considerarla como respuesta válida al problema en cuestión, así como el trabajo relacionado ya existente que sirve como apoyo y comparación.

3.1. Descripción

La simulación de partículas coloidales presenta características adicionales o distintas con respecto a otras simulaciones de interacciones entre cuerpos, ya sea por propiedades de los coloides o por el tipo de experimento que se quiere realizar con ellos. En esta sección se describen cada una de estas propiedades.

3.1.1. Interacción entre partículas

La regla de integración para actualizar la posición de una partícula en el tiempo se define de la siguiente forma:

$$\vec{r_i}(t + \Delta t) = \vec{r_i}(t) + \vec{F_i}(t)\Delta t + \vec{\xi}\sqrt{T\Delta t},$$

en donde $\vec{F_i} = \dot{\vec{r_i}}$ es la velocidad determinista obtenida a partir de las interacciones entre la partícula i y el resto de ellas, T es el coeficiente de difusión o temperatura del sistema, Δt es el paso de tiempo y $\vec{\xi}$ es una variable aleatoria con distribución normal estándar, que corresponde al ruido que se agrega al movimiento como factor aleatorio [16].

La fuerza a simular no cumple con el principio de acción y reacción [17], y depende de los valores α_i, μ_i de las partículas involucradas en la interacción como sigue:

$$\vec{F_i} = \sum_{k \neq i} \mu_i \alpha_k \vec{f}(\vec{r_i} - \vec{r_k}),$$

en donde $\vec{f}(\vec{r}) = \frac{\vec{r}}{r^3}$ para la fuerza de largo alcance estudiada, mientras que $\vec{f}(\vec{r}) = \frac{\vec{r}}{r^7}$ para la fuerza de corto alcance. Es posible que en una simulación existan fuerzas distintas de largo y corto alcance actuando sobre las partículas.

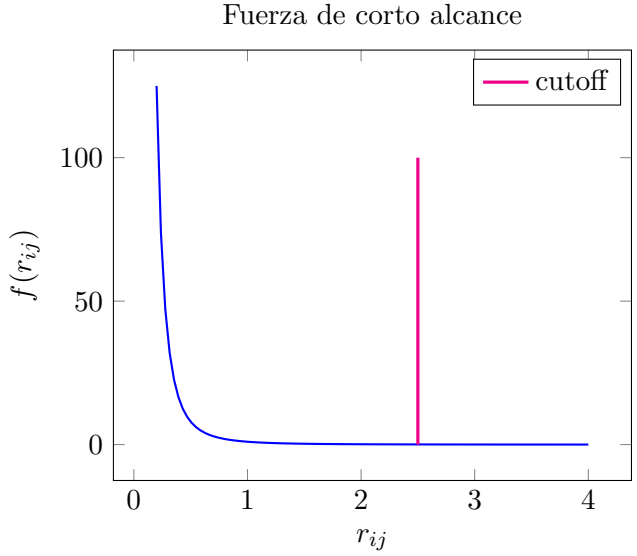


Figura 3.1: Influencia de la distancia en el cálculo para la fuerza de corto alcance entre partículas. A partir del radio de corte r_{cutoff} marcado en magenta, la expresión para la fuerza no se evalúa y el valor de f_{ij} se considera 0.

Debido a que la fuerza de corto alcance decae de forma mucho más rápida con respecto a la distancia en comparación a la fuerza de largo alcance, se considera una distancia de corte tras el cual la fuerza se evalúa como nula, de acuerdo a lo mostrado en la figura 3.1. Por lo tanto, el cálculo de la fuerza de corto alcance entre dos partículas se define como:

$$\vec{F_{ij}}(\vec{r_{ij}}) = \begin{cases} \mu_i \alpha_k \vec{f}(\vec{r_{ij}}), & \text{si } r_{ij} \leq r_{cutoff} \\ 0, & \text{si no} \end{cases}$$

3.1.2. Volúmen excluido

Las partículas coloidales a simular no son puntuales, sino que tienen un diámetro σ igual para todas (y por consecuencia, la misma área). Como la regla de integración definida anteriormente no

considera esta propiedad, puede ocurrir que las posiciones actualizadas produzcan partículas cuyas áreas (o volúmenes, en 3 dimensiones) se intersecten y exista un traslape entre ellas, lo cual no es físicamente válido. Para asegurar que esta propiedad se cumpla al final de cada paso de tiempo, a cada par de partículas (p_i, p_j) se les aplican desplazamientos que corrigen los traslapes:

$$\vec{r_1}\prime = \vec{r_1} - \delta^* \frac{(\vec{r_2} - \vec{r_1})}{|\vec{r_{12}}|} \qquad\qquad \vec{r_2}\prime = \vec{r_2} - \delta^* \frac{(\vec{r_1} - \vec{r_2})}{|\vec{r_{12}}|},$$

en donde $\vec{r_1}\prime, \vec{r_2}\prime$ son las posiciones actualizadas producto del traslape corregido entre $\vec{r_1}$ y $\vec{r_2}$. Si $\delta^* = \frac{\sigma - |\vec{r_{12}}|}{2}$, las partículas se desplazan en direcciones opuestas entre sí a lo largo de $\hat{n} = \frac{(\vec{r_2} - \vec{r_1})}{|\vec{r_{12}}|}$ hasta quedar en contacto tangencial. En cambio, si $\delta^* = \sigma - |\vec{r_{12}}|$, las partículas se separan en la misma dirección a una distancia proporcional a la magnitud del traslape que había entre ellas, lo que simula un efecto de rebote producto de la colisión en algún instante $t^* \leq t + \Delta t$. Este último valor es el que se usa para procesar los traslapes en la simulación.

3.1.3. Condiciones de borde periódicas

El área de simulación es periódica en las direcciones X e Y. Esto implica que cuando una partícula cruza alguno de los bordes del área, reaparece en el borde opuesto, por lo que sus coordenadas siguen estando restringidas dentro de los márgenes originales. Para obtener las distancias entre partículas se utiliza la convención de imagen mínima, en donde una partícula p_i interactúa con p_j ya sea directamente o con su imágen, la que esté más cerca.

3.1.4. Ruido aleatorio

El movimiento de las partículas en el paso de integración tiene una componente de ruido aleatorio, modelada como una variable aleatoria con distribución normál estándar o *gaussiana* (media $\mu = 1$ y desviación estándar $\sigma = 0$). La figura 3.2 muestra la distribución normal y distintas bandas de unidad σ y centradas en el origen, lo que corresponde a la probabilidad de obtener valores en el rango correspondiente bajo esta distribución.

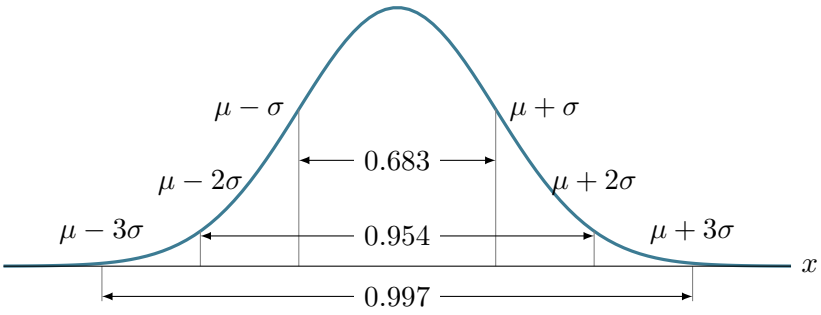
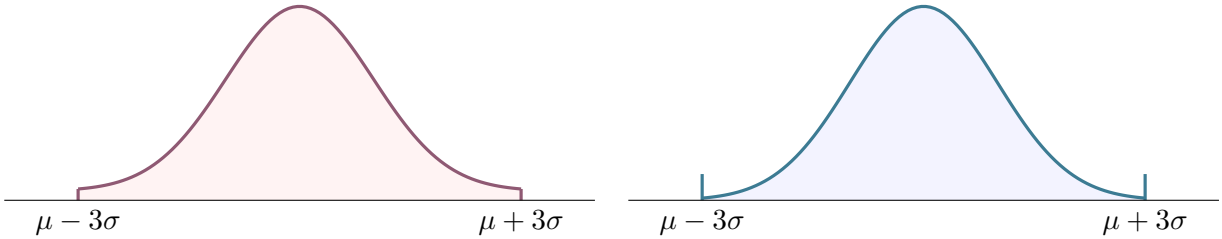


Figura 3.2: Distribución normal. Las bandas indican las probabilidades respectivas de que un valor generado con la distribución normal esté comprendido en ese rango.

Para evitar que se produzcan desplazamientos excesivos debido a valores demasiado grandes en el ruido (en los extremos de la campana *gaussiana*), se eliminó la posibilidad de que sean generados durante la simulación. Se consideraron dos métodos para lograr esto, los cuales se muestran en la figura 3.3:

(a) Re-generar los valores que estén fuera del rango $[\mu - 3\sigma, \mu + 3\sigma]$.

(b) Truncar los valores en el rango $[\mu - 3\sigma, \mu + 3\sigma]$.



(a) Regenerando valores fuera de rango.

(b) Truncando los valores generados.

Figura 3.3: Formas de eliminar valores extremos generados aleatoriamente.

Ambos métodos producen distribuciones de probabilidad distintas entre sí y con respecto a la original; mientras la alternativa 1 aumenta la probabilidad de todos los valores en el rango, la alternativa 2 aumenta la probabilidad de generar los valores extremos. Esto no produce una distorsión estadística demasiado grande, debido a que el rango considerado corresponde a la banda centrada en la media de ancho igual a 3 desviaciones estandar, lo que incluye al 99.7 % de los valores posibles.

La implementación de la solución se hizo utilizando la alternativa (b), debido a que la operación de truncado se puede realizar indistintamente sobre todos los valores generados, y solo afectar a

aquellos que están fuera de rango. En cambio, para la alternativa (a) un valor fuera de rango debe recrearse hasta satisfacer la condición, lo que produce los efectos de divergencia explicados en la figura 2.2 al realizarse en paralelo.

3.2. Relevancia de la solución

Actualmente existe una solución secuencial descrita en [19] e implementada en C que resuelve el problema planteado. Sin embargo, el tiempo de ejecución crece rápidamente con el número de partículas, por lo que no es posible completar simulaciones con más de 10^4 partículas en un tiempo razonable. Esto limita la complejidad de los experimentos que se pueden realizar con el programa existente.

3.3. Requisitos de la solución

El foco principal de la solución a desarrollar se encuentra en resolver la corrección de traslapes de manera eficiente, dejando como prioridad secundaria el desarrollo de un algoritmo *n-body* para fuerza de largo alcance más eficiente que el actual. La lista de requisitos consiste en:

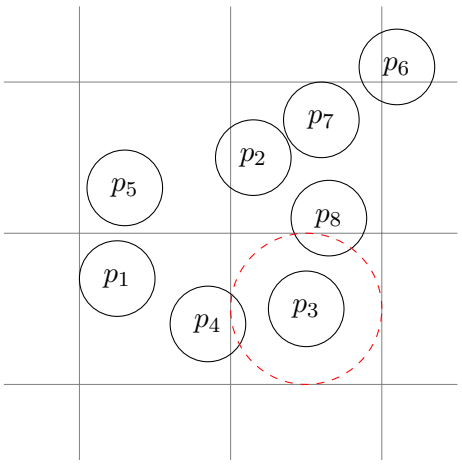
- La implementación debe poder simular interacciones de cuatro distintos tipos de partículas, con la posibilidad de llegar hasta 10 tipos.
- Al final de cada paso de tiempo, se debe mantener el invariante del volumen excluido para todas las partículas del sistema. Si debido a la configuración de las partículas no es posible conseguir lo anterior, la cantidad de traslapes no corregidos debe ser pequeña y no debe aumentar en pasos de tiempo posteriores.
- La corrección de traslapes debe mejorar a la solución secuencial, en donde se evalúan todos los pares de forma innecesaria.

3.4. Características deseadas

Idealmente, la solución implementada debería ser capaz de realizar simulaciones con hasta 10^6 partículas. Además, cada cierta cantidad de pasos de tiempo especificada por el usuario, la solución debe entregar la configuración del sistema a memoria principal en ese instante para realizar mediciones.

3.5. Trabajo relacionado

Además de la solución secuencial descrita anteriormente, existe una implementación paralela para la fuerza de largo alcance que utiliza el algoritmo *n-body* descrito en [14] y una lista de vecinos. Cada vez que el algoritmo anterior procesa un par de partículas, se verifica si están a distancia $r_{ij} \leq r_s$, en donde $r_s \geq \sigma/2$ es el valor de piel (o *skin*) igual para cada partícula. Si la condición se cumple, p_j se agrega a la lista de vecinos de p_i y viceversa, lo que permite hacer la verificación de traslapes entre ese par de partículas en la etapa de corrección. Esto permite considerar como vecinos a partículas que inicialmente están lejos, pero pueden acercarse lo suficiente como para producir traslapes debido a correcciones entre otras partículas.



(a) Espacio particionado.

id celda	id partícula
cell1	p_5
cell1	
cell1	
cell2	p_2
cell2	p_7
cell2	p_8
cell3	p_1
cell3	p_4
cell3	
cell4	p_3
cell4	
cell4	

(b) Lista de celdas.

Figura 3.4: Lista de celdas, en donde cada celda tiene una lista de partículas que residen en ella. La circunferencia de radio r_s que rodea a p_3 corresponde a la piel de esa partícula. Luego, p_4 y p_8 forman parte de la lista de vecinos de p_3.

También existe una implementación en CUDA del algoritmo de lista de Verlet para fuerzas

de corto alcance. Además de la piel de radio r_s, se particiona el espacio 2D en celdas cuadradas uniformes de largo $l \geq r_s$ como las mostradas en la figura 3.4, en donde cada celda puede almacenar una cantidad finita de partículas. Luego, para cada partícula solo se necesita recorrer la celda a la que pertenece y las 8 celdas vecinas para construir la lista de vecinos, la cual se usa en el cálculo de fuerza de corto alcance y la corrección de traslapes. A diferencia de la implementación anterior en donde la lista de vecinos se recalcula en cada paso de tiempo, la lista de vecinos y la lista de celdas se obtienen al momento de construir y actualizar la lista de Verlet.

Capítulo 4

Especificación de la solución

Este capítulo contiene las razones que llevaron a desarrollar la solución implementada, a partir de lo presentado en las secciones 2 y 3. Luego, se describen los algoritmos que componen la implementación así como el esquema de clases con el que se organizan y ejecutan. Además, se describen las estructuras adicionales utilizadas en la solución y que no fueron introducidas en la sección 2, al no ser utilizadas por ninguna de las soluciones estudiadas.

4.1. Elección de la solución

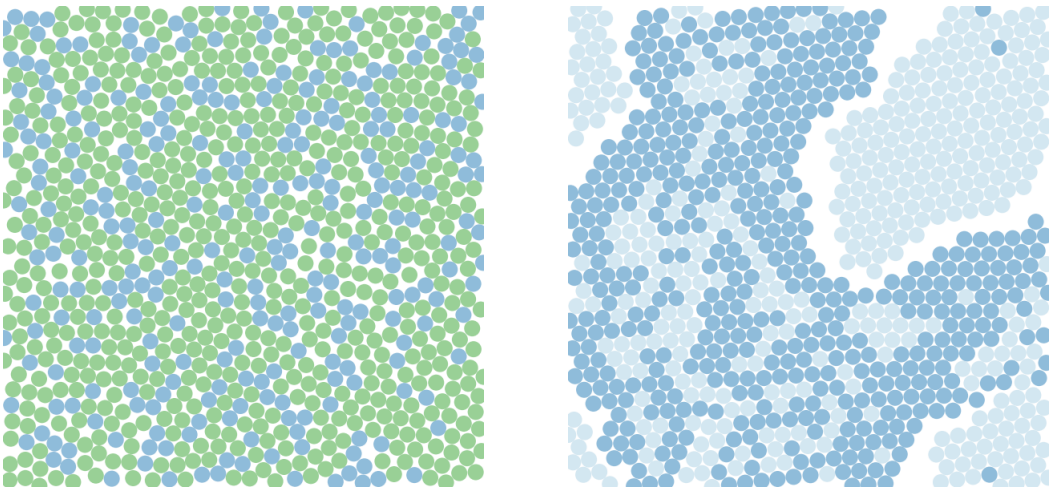
El problema a resolver es buen candidato a beneficiarse de la paralelización en GPU, debido a que se debe ejecutar una misma instrucción (el cálculo de fuerzas) sobre diferentes datos (las partículas). Esto se hace evidente al considerar la cantidad de trabajos existentes que intentan resolver problemas similares como simulaciones astrofísicas o de dinámica molecular. Luego, se elige implementar la solución escogida en CUDA, debido a que tanto el equipo con el que se desarrolló este trabajo como el servidor en el cual se realizaron las pruebas cuentan con tarjetas gráficas que soportan esta plataforma. La parte secuencial se desarrolló en C++, debido a que puede interoperar con CUDA sin necesidad de librerías externas; además, la implementación diseñada necesita de la librería CGAL [20] como componente.

Para aplicaciones que simulan interacciones como la fuerza de gravedad, la relación recíproca $\vec{f_{ji}} = -\vec{f_{ij}}$ permite optimizar la integración al reducir a la mitad el número de evaluaciones. Esta optimización no se cumple para las estrategias de evaluación en paralelo, en donde se desea que un *thread* se ocupe exclusivamente de su partícula respectiva para evitar accesos dispersos a memoria

de GPU [14]. Sin embargo, como en este problema no se cumple la simetría de acción y reacción, la relación anterior no es cierta y se deben evaluar las N^2 fuerzas entre partículas de todas maneras, por lo que la evaluación paralela es ideal para simular este tipo de interacciones.

En las simulaciones astrofísicas, las galaxias generalmente se consideran como objetos puntuales que pueden formar cúmulos considerablemente más densos que otras regiones, como por ejemplo el centro de una galaxia espiral con respecto a sus brazos. Esto hace que los esquemas de particionamiento no uniforme del espacio puedan optimizar efectivamente la computación de las fuerzas, debido a que las regiones menos interesantes queden representadas en celdas de mayor tamaño, al contrario de las zonas densas que son representadas con mayor detalle al usar más celdas.

En cambio, para el caso de las partículas coloidales, la propiedad de volumen excluido no permite que los objetos se compacten demasiado y dejen espacios vacantes considerablemente grandes, lo que produce particionamientos más uniformes con respecto al caso de las galaxias. Existen configuraciones estudiadas como las que aparecen en la figura 4.1 que se distribuyen uniformemente a través del área de simulación, casos en los cuales el particionamiento jerárquico del espacio sería muy similar a dividir en celdas de tamaño uniforme.



(a) Configuración 0. (b) Configuración 3.

Figura 4.1: Distintas configuraciones de partículas distribuidas en el área de simulación, para $N = 896$ y $dt = 0.01$. Los números de cada configuración denotan a las descritas en la tabla 5.1. Los colores denotan distintos tipos de partículas.

Para implementar una solución de dinámica guiada por eventos, se necesita conocer las posiciones y velocidades de ambas partículas involucradas en el traslape para evaluar el instante de colisión y prevenir el traslape. Sin embargo, la posición de una particula en este problema depende tanto de la

velocidad computada como del ruido generado en ese paso de tiempo. Esto dificulta la determinación del instante exacto de colisión, debido a que la partícula se movió aleatoriamente durante todo el paso de tiempo y no hay una forma de determinar cuál era su posición en el instante intermedio a partir del valor del ruido en Δt.

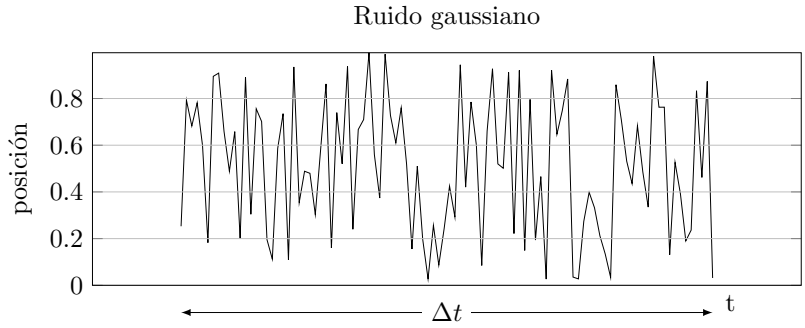


Figura 4.2: Movimiento aleatorio de una partícula durante un paso de tiempo.

Por otro lado, el ruido gaussiano se puede generar en paralelo de forma directa a través de la librería cuRAND. Su API provee una función que entrega en una sola llamada dos valores aleatorios con distribución normal, utilizando internamente la transformación de Box-Muller [13]. Esto es ideal para generar un vector de ruido en cada paso de tiempo y sumarlo con el desplazamiento determinista para integrar cada partícula del sistema.

4.1.1. Triangulación de Delaunay

Para realizar la corrección de traslapes eficientemente es necesario poder acceder a la vecindad de las partículas, información que debe ser válida en cada paso de tiempo. Para ello se eligió utilizar la triangulación de Delaunay, la cual se define junto a otros conceptos asociados a continuación.

Una subdivisión planar maximal de un conjunto de puntos $P = p_1, p_2, ..., p_n$ es una subdivisión tal que no se pueden agregar arcos al grafo correspondiente sin afectar la planaridad del mismo. Luego, una triangulación de P es una subdivisión planar maximal sobre el conjunto de vértices P. Esto permite definir formalmente la triangulación de Delaunay:

Definición: Una triangulación de Delaunay es una triangulación de un conjunto de puntos P en donde ninguno de los circuncírculos respectivos de cada tríangulo contienen a algún otro punto de P.

Una propiedad de esta estructura es que maximiza el ángulo mínimo de los triángulos que la componen [3], por lo que tienden a ser equiláteros y las longitudes de los arcos correspondientes no son demasiado grandes. Esto produce que las partículas cercanas entre sí estén unidas por arcos en la triangulación, lo que permite hacer verificaciones únicamente entre objetos que tienen la posibilidad de participar en un traslape.

La triangulación debe cumplir con las condiciones de borde periódicas para poder ser utilizada en el sistema, lo que puede ser representado como un toro de radios mayor y menor $R = r = L$ [8] según se muestra en la figura 4.3. No se encontró trabajo sobre la construcción de triangulaciones periódicas en paralelo, por lo que se utilizó la implementación provista por la librería CGAL . Esto no representa una desventaja mayor si es que la triangulación construida en memoria principal solo se envía una sola vez a GPU, lo cual se realiza en la etapa de inicialización.

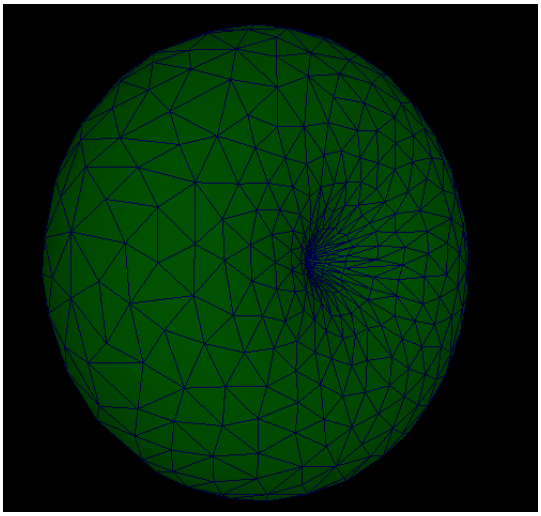
En la figura 4.3 se muestran representaciones planas y de la superficie triangulada, en donde los extremos del área de simulación pueden estar conectados por arcos.

4.2. Diseño de clases

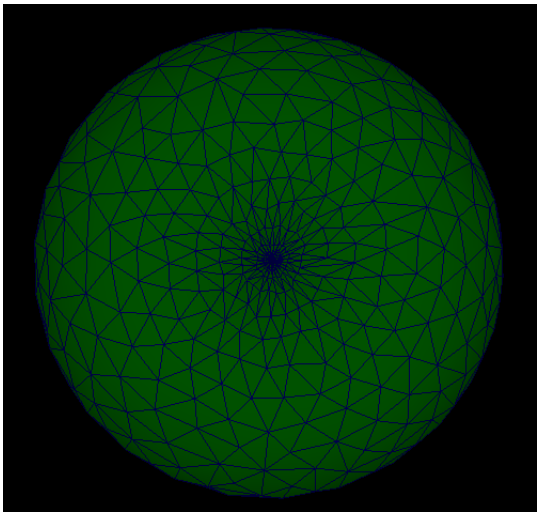
Se diseñó una única clase `Particle_System` la cual representa una instancia de simulación. Esta clase se encarga de inicializar, operar y liberar las estructuras de datos paralelas utilizadas por el código CUDA, debido a que administrar y utilizar la memoria en GPU tiene mayor complejidad que trabajar con la memoria principal utilizada en programas secuenciales convencionales. La clase está implementada en distintos módulos, que corresponden a:

- Inicialización de estructuras de datos paralelos y comunicación con la GPU.
- Inicialización de la configuración inicial.
- Creación de triangulación de Delaunay en CGAL.
- Funciones de I/O para entregar datos sobre las posiciones, triangulación y reporte de errores.
- *Wrappers* de invocación de los *kernels* CUDA que realizan la simulación.

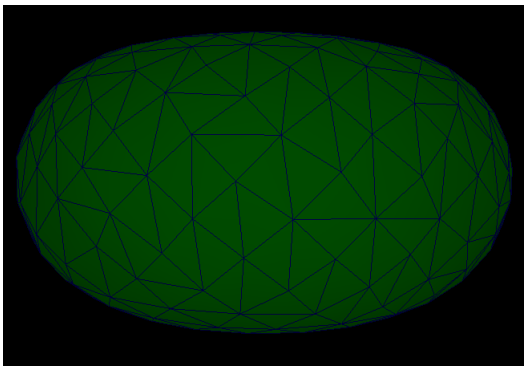
La inicialización de la configuración inicial depende de valores generados aleatoriamente, los cuales se obtienen desde la GPU. Estos valores no se generan a través de una llamada a un *kernel*,



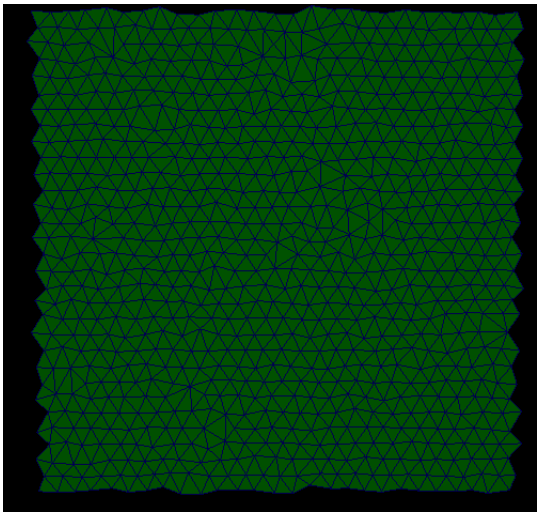
(a) Vista general.



(b) Vista frontal.



(c) Vista lateral.



(d) Vista aplanada.

Figura 4.3: Triangulación de Delaunay en 2 dimensiones bajo condiciones de borde periódicas. Las líneas corresponden a arcos de la triangulación.

sino que se obtienen mediante la *Host API* de la librería cuRAND. La clase `RandGen` implementa este proceso, permitiendo obtener una cantidad finita de números aleatorios generados en una sola llamada a la librería. Esta clase no depende de `Particle_System` y puede ser utilizada por cualquier otro programa.

4.3. Estructuras de datos

Esta sección comprende las estructuras utilizadas tanto en la parte secuencial del programa como en la ejecución paralela.

4.3.1. Estructuras para ejecución paralela

La estructura principal del programa reside completamente en memoria global de GPU, y está organizada como Estructura de Arreglos o SoA (*Structure of Arrays*). La estructura encapsula todos los arreglos utilizados tanto en la integración (posiciones y tipos de las partículas) como en la triangulación de Delaunay (información de arcos y triángulos).

Debido a que las posiciones de cada partícula están compuestas de dos coordenadas y siempre se necesita de ambas para realizar cálculos (no tiene sentido operar únicamente con x o y), se utilizan los tipos de vectores provistos por CUDA en lugar de almacenar los valores de punto flotante consecutivamente o en arreglos distintos. Esto consiste en almacenar el par (x_i, y_i) como un dato de tipo `float2` o `double2`, lo que permite a los threads traer la posición a registro y devolverla a la memoria global con una sola instrucción en lugar de cargar y guardar x e y por separado.

Las actualizaciones de posición para cada partícula (debido al cálculo de fuerzas y corrección de traslapes) se pueden computar simultáneamente si los resultados de estas funciones se escriben en un arreglo diferente que funcione como *buffer* para las posiciones [11]. Esto permite que los *kernels* puedan escribir sus resultados inmediatamente en memoria global sin tener que incurrir en el costo de sincronizar los *threads* antes de la escritura. De lo contrario, un *thread* que haya computado la velocidad de su partícula tendría que esperar a que todas las demás hayan leido la posición anterior antes de escribir el nuevo valor, para no tener problemas de concurrencia.

Las estructuras utilizadas dependen directamente de N (posición y atributos de cada partícula), o bien corresponden a las listas de triángulos y arcos, las cuales también son lineales con respecto a

N para triangulaciones en dos dimensiones. Luego, el algoritmo desarrollado utiliza $O(N)$ espacio en total.

4.3.2. Constantes de simulación

Para encapsular los parámetros que permanecen constantes durante toda la vida de una misma instancia de simulación, se agrupan en una estructura `SimParameters` que reside en memoria principal y de GPU. A diferencia de las partículas, estos parámetros se cargan en la memoria constante, a la cual se puede acceder desde cualquier *kernel* con un costo de acceso menor al de traer datos en memoria global de GPU.

4.4. Algoritmos

El algoritmo que realiza la simulación ejecuta cada uno de sus pasos completamente en paralelo, pero de manera secuencial entre sí. La función de integración invoca exactamente un *kernel* CUDA por paso de tiempo, mientras que las demás funciones pueden realizar múltiples llamadas a sus *kernels* respectivos en un mismo paso. Las funciones de integración y corrección de traslapes son las únicas que realizan trabajo propio de la simulación, que corresponde a actualizar las posiciones de cada partícula y asegurar las propiedades del sistema. Las funciones que operan sobre la triangulación de Delaunay sirven para mantener la validez de esta estructura en cada paso de tiempo, y así ser utilizada por la función de corrección de traslapes.

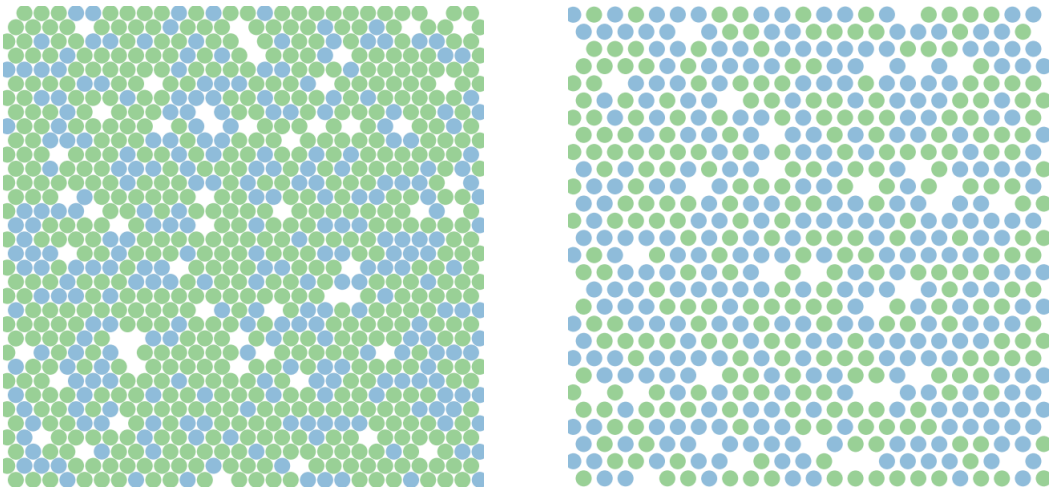
Algoritmo 1 Simulación de sistema de partículas

Entrada: $P = \{p_1, ..., p_N\}$ lista de posiciones de partículas
Salida: $P = \{p_1, ..., p_N\}$ lista de posiciones actualizadas en un paso de tiempo

```
1: procedure RUNSIMULATION(P)
2:     Generar la configuración inicial de las N partículas
3:     for t ← 0 to T_f do
4:         Integrar las N partículas en t + Δt
5:         Corrección de triángulos invertidos
6:         Actualizar triangulación de Delaunay
7:         Corrección de traslapes
8:     end for
9: end procedure
```

4.4.1. Generación de la configuración inicial

Las posiciones del sistema se inicializan como una red triangular con N^* vértices, en donde cada vértice corresponde a una partícula. De este total se eliminan al azar $N^* - N$ partículas para llegar al número de objetos especificado en la entrada. Además, para evitar comportamientos anómalos producidos por partículas perfectamente alineadas entre sí, a las posiciones en la red se les suma un ruido aleatorio para obtener las posiciones en $t = 0$, las que se envían posteriormente a la GPU para realizar la simulación. Ambos procesos se hacen generando valores aleatorios en la GPU por medio de la clase `RandGen`, generando configuraciones iniciales como en la figura 4.4, en donde las partículas no están exactamente alineadas con respecto a sus vecinos.



(a) Configuración 0. (b) Configuración 1.

Figura 4.4: Posiciones iniciales de las partículas del sistema en $t = 0$ para $N = 896$. Ambas configuraciones corresponden a las descritas en la tabla 5.1. Los colores de las partículas indican su tipo.

4.4.2. Integración para fuerza de largo alcance

La implementación del método *n-body* para la fuerza de largo alcance está basada en la implementación paralela en CUDA [14], en la cual se aprovecha la memoria compartida de cada bloque para cargar las partículas por grupos desde memoria global. La optimización desarrollada consiste en utilizar la función intrínseca de *warp shuffle*, la cual permite a un *thread* obtener datos guardados en los registros de otros *threads* que pertenezcan a un mismo *warp*. A cada *thread* se le asigna un número de pista que lo identifica con respecto a los demás *threads* del *warp*, lo cual les permite a cada uno leer una partícula distinta desde memoria global. Luego, cada miembro del *warp* se turna

en propagar los datos de su partícula correspondiente a los demás, quienes pueden recibir estos datos usando la instrucción `__shfl()`, entregando como argumento el número de pista del *thread* que está en el turno de ofrecer sus datos.

Algoritmo 2 Integración del Sistema de Partículas

Entrada: P_0 arreglo de partículas $(x_i, y_i, \alpha_i, \mu_i)$
Salida: P_1 partículas con posiciones actualizadas

1: **procedure** INTEGRATE(P_0, P_1)
2: **for** $i \leftarrow 0$ **to** $|P|$ **do**
3: $l_i \leftarrow$ `threadIdx.x` & 31
4: $\vec{b_i} \leftarrow P_0[i]$
5: $\vec{v_i} \leftarrow 0$
6: **for** $j \leftarrow 0$ **to** $|P|$; $j \leftarrow j + 32$ **do**
7: $\vec{b_j} \leftarrow P_i[j + l_i]$
8: **for** $k \leftarrow 0$ **to** 32 **do**
9: $\vec{b_k} =$ `_shfl`$(\vec{b_j}, k)$
10: $\vec{r_{ik}} \leftarrow$ `dist`$(\vec{b_i}, \vec{b_k})$
11: $\vec{v_i} \leftarrow \vec{v_i} - \vec{r_{ik}} \cdot \frac{\alpha_i \mu_k}{r_{ik}^3}$
12: **end for**
13: `_syncthreads()`
14: **end for**
15: **end for**
16: $P_1[i] \leftarrow \vec{b_i}\vec{v_i}\Delta t + \vec{\xi_i}\sqrt{T\Delta t}$
17: **end procedure**

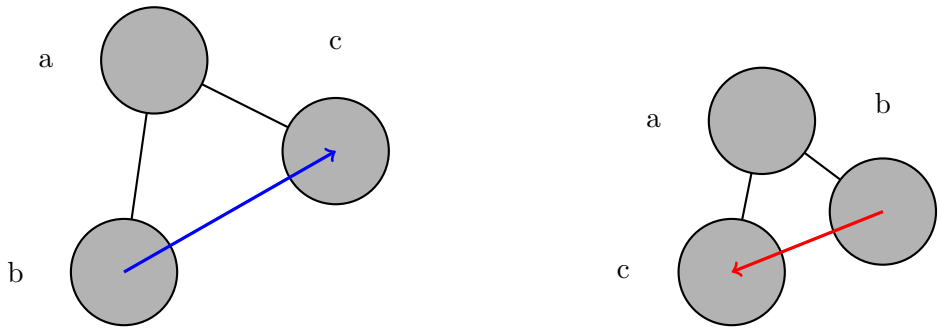
Una vez que todo el *warp* ha compartido los datos entre sus miembros, cada uno obtiene una partícula desde memoria global hasta completar el siguiente grupo de elementos, repitiendo el mismo procedimiento. La principal ventaja de este proceso es la mayor eficiencia de accesos a memoria, ya que la mayor parte del tiempo los *threads* están compartiendo datos a nivel de registro en lugar de realizar consultas más costosas en memoria global. Además, como los *threads* de un *warp* se ejecutan en paralelo, no es necesario que se sincronizen entre sí, a diferencia de la implementación en memoria compartida.

Verificación topológica

Existe un caso especial de traslape que puede producirse cuando el movimiento de una partícula es excesivamente grande, típicamente debido a un valor de Δt demasiado alto en la integración. Puede ocurrir que una partícula haya atravesado el volúmen de otra durante su desplazamiento, aún cuando las posiciones finales no indiquen un traslape entre ellas. Esto no tiene sentido físico y

por lo tanto invalida la simulación, pero es difícil de detectar eficientemente.

Un beneficio extra de la triangulación de Delaunay consiste en utilizar su topología para verificar eficientemente si un par de partículas se ha atravesado entre sí. Como los arcos de la triangulación son orientados, se puede comparar la posición $\vec{r_{12}} = \vec{e_2} - \vec{e_1}$ antes y después de realizar la integración, en donde $\vec{e_2}, \vec{e_1}$ son las partículas que componen el arco. Si $\vec{r_{12}}^0 \cdot \vec{r_{12}}^1 < 0$, entonces la dirección general de las posiciones es distinta y las partículas se atravesaron entre sí. Cuando esto ocurre, se aborta la simulación sin posibilidad de reanudarla, ya que el movimiento que produjo este error no tiene sentido físico (una partícula no puede pasar por sobre otra, debido a los volúmenes excluidos de ambas). En este caso, la solución consiste en disminuir el valor de Δt especificado por el usuario y reintentar la simulación.



(a) Situación inicial.

(b) Tras la integración.

Figura 4.5: Detección de movimiento inválido. El vector azul $\vec{r_{bc}}^0$ corresponde a la distancia entre partículas. Después del desplazamiento, el vector en rojo $\vec{r_{bc}}^1$ cambia su sentido general con respecto a la situación inicial, indicando que hubo un traslape más fuerte entre ambas partículas.

La figura 4.5 muestra el cambio de sentido en las distancias entre un par de partículas que se traspasaron entre sí. Esta función no se realiza en la versión del programa utilizada para las pruebas realizadas en la sección 5, ya que solo sirve para probar valores de Δt y hacer *debugging*. Sin embargo, sí se utilizó para verificar que los resultados preliminares fueran correctos.

4.4.3. Actualización de la triangulación

La actualización de las posiciones debido al paso de integración puede producir distintos problemas en la triangulación almacenada:

1. Existen triángulos que no cumplen la propiedad del circuncírculo vacío, por lo que la triangu-

lación deja de ser de Delaunay.

2. Existen triángulos cuya normal apunta en la dirección opuesta, por lo que ya no pueden ser recorridos en orden CCW.

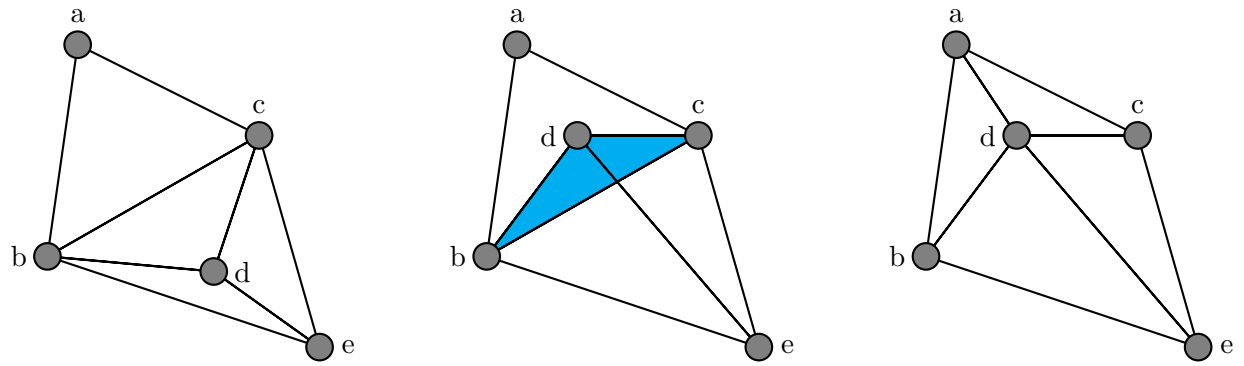
3. Existen arcos que se cruzaron entre sí, por lo que el grafo subyacente deja de ser planar (y por lo tanto, deja de ser triangulación).

El primer problema se puede solucionar a través de una serie de operaciones de giros (o *edge-flips*) de los arcos que no cumplen con la condición de Delaunay [3]. El algoritmo de Lawson [9] recibe una triangulación cualquiera y entrega como salida la triangulación de Delaunay, pero para utilizar su implementación secuencial hay que incurrir en un gran sobrecosto de transferir los datos a memoria principal y de vuelta a la GPU. Esto, sumado a que el algoritmo es de complejidad $O(n^2)$, hace que el costo de esta estrategia sea prohibitivo.

Por lo tanto, se utiliza la solución paralela diseñada en [12] e implementada en la librería **cleap**, la cual se realiza completamente en GPU y evita la necesidad de utilizar el bus de datos en cada paso de tiempo. Este algoritmo de *edge-flipping* paralelo produce una triangulación *quasi-Delaunay*, ya que algunos giros de arco pueden ser ignorados debido al parámetro de tolerancia utilizado en el cálculo de la condición de Delaunay. Sin embargo, el error reportado con respecto a las implementaciones secuenciales existentes es menor a 0.05 %, lo que significa que la cantidad de *edge-flips* que no se hicieron es mínima.

4.4.4. Corrección de triángulos invertidos

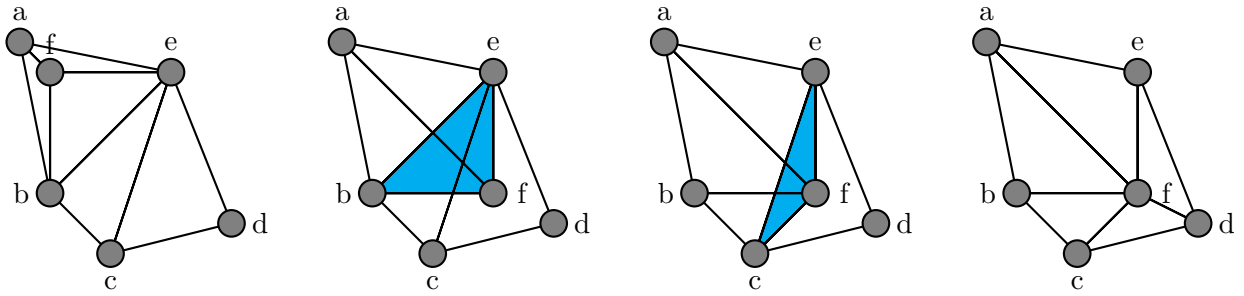
Como el algoritmo de *edge-flipping* resuelve el primer problema enunciado anteriormente, resta resolver los otros dos problemas que pueden invalidar la estructura de vecindad a usar en la corrección de traslapes. En realidad los dos casos son producto de un mismo fenómeno, que ocurre cuando el movimiento de una partícula hace que ésta atraviese uno o más arcos en la triangulación de Delaunay.



(a) Situación inicial. (b) Desplazamiento de d sobre el arco **bc**. (c) *Edgeflip* entre **bc** y **ad**.

Figura 4.6: Resolución de triángulo invertido. El polígono que se invirtió producto del desplazamiento de la partícula d está sombreado en color cian.

Es posible resolver este problema a traves de una secuencia de uno o más *edge-flips*, notando en la figura 4.6 que el arco cruzado es el que debe girarse. Una vez realizado el *flip*, tanto el triángulo invertido como el cruce de arcos desaparece, reparando la triangulación. La malla reparada no necesariamente cumple la propiedad de Delaunay, por lo que la corrección de triángulos invertidos debe hacerse antes de ejecutar el algoritmo de *edge-flipping*.



(a) Situación inicial. (b) Desplazamiento de f. (c) *Edgeflip* entre **be** y **cf**. (d) *Edgeflip* entre **ce** y **df**.

Figura 4.7: Resolución de triángulo invertido con corrección de dos arcos.

La figura 4.7 muestra la secuencia de *edge-flips* que elimina tanto los triángulos invertidos cuando la partícula se mueve hacia un triángulo más allá de su vecindad inmediata. No se ha demostrado que esté procedimiento sea válido cuando una partícula cruza 3 o más arcos, ni tampoco si es posible generalizarlo a cualquier cantidad de cruces de arcos. Asimismo, no se pudo obtener una implementación que permita identificar los arcos que deben ser corregidos cuando una partícula atraviesa dos o más de ellos y reproducir el procedimiento mostrado en la figura 4.7. Sin embargo,

esto no supone un problema mayor para las simulaciones estudiadas, ya que las magnitudes de los movimientos son suficientemente pequeñas para que todas las inversiones de triángulos se produzcan por partículas que cruzan solo un arco.

Detección de triángulos invertidos

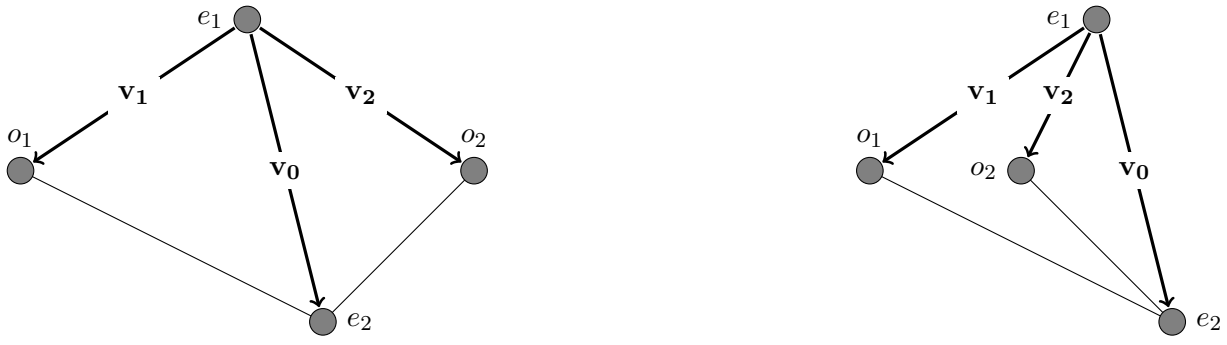
La inversión de un triángulo puede producirse por dos razones: la orientación de uno de sus arcos se invirtió (ver figura 4.5), o bien un vértice se movió hacia el otro lado del semiplano definido por el arco que componen los otros dos vértices. En la sección 4.4.2 se mostró que la primera condición no corresponde a un movimiento válido, por lo que se excluye del análisis. Por lo tanto, la verificación de triángulo invertido equivale a verificar si un vértice cruzó alguno de los arcos que lo rodean hacia un triángulo vecino, con lo que el problema se reduce a verificar si un punto se ubica dentro de un triángulo. Para realizar esto de forma eficiente se utilizan coordenadas baricéntricas:

$$\mathbf{v_0} = \mathbf{e_2} - \mathbf{e_1} \qquad d_2 = \mathbf{v_2} \times \mathbf{v_0}$$

$$\mathbf{v_1} = \mathbf{o_1} - \mathbf{e_1} \qquad s_2 = \mathbf{v_1} \times \mathbf{v_0}$$

$$\mathbf{v_2} = \mathbf{o_2} - \mathbf{e_1} \qquad t_2 = \mathbf{v_2} \times \mathbf{v_1}$$

$$flip(\mathbf{e_1}, \mathbf{e_2}, \mathbf{o_1}, \mathbf{o_2}) = \begin{cases} s_2 <= 0 \ \& \ t_2 <= 0 \ \& \ s_2 + t_2 >= d_2, & \text{si } d_2 < 0 \\ s_2 >= 0 \ \& \ t_2 >= 0 \ \& \ s_2 + t_2 <= d_2, & \text{si no} \end{cases}$$



(a) Situación regular. (b) Triángulo invertido.

Figura 4.8: Detección de tríangulo invertido utilizando coordenadas baricéntricas.

La figura 4.8 muestra los vectores utilizados en el predicado que determina si el arco original debe ser girado. En (b) se muestra una configuración para la cual el predicado es verdadero, indicando que

el arco (e_1, e_2) debe ser reemplazado con (o_1, o_2), que es el arco formado por los vértices opuestos a ambos lados del arco original.

Sin embargo se debe verificar si o_1 está dentro del triángulo (e_1, e_2, o_2), o bien o_2 está dentro del triángulo (e_1, e_2, o_1), ya que ambos sucesos hacen necesario un *edge-flip*. Los valores d_2, s_2, t_2 utilizados en las ecuaciones anteriores se pueden utilizar para calcular d_1, s_1, t_1 que permiten hacer la verificación equivalente para o_1,:

$$d_1 = s_2 \qquad s_1 = d_2 \qquad t_1 = -t_2$$

Esto es equivalente a definir $\mathbf{v_1}$ como $\mathbf{v_2}$ y viceversa en las ecuaciones anteriores, por lo que se puede realizar ambas verificaciones de forma eficiente. Otra ventaja de esté metódo es que no necesita recibir los triángulos en orden, por lo que no se necesitan accesos extra al arreglo de triángulos en memoria global. Además, las estructuras de datos definidas para el algoritmo de *edge-flip* permiten acceder a los 4 vértices involucrados en la verificación.

4.4.5. Corrección de traslapes

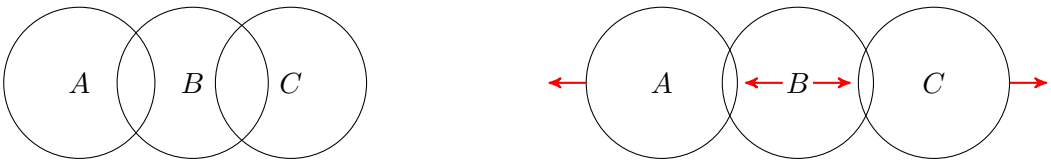
La corrección de traslapes implementada usa la información topológica contenida en los arcos de la triangulación de Delaunay para obtener de forma rápida cuál es la vecindad de una partícula que puede estar traslapando con ella. El *kernel* correspondiente considera un *thread* por arco, obteniendo las posiciones de ambos extremos para evaluar si existe un traslape. Si lo hay, se evalúan los desplazamientos de acuerdo a las ecuaciones mostradas en la sección 3.1.2 y se suman al desplazamiento total en el arreglo de salida. Como una misma partícula puede ser parte de varios arcos a la vez, la suma del desplazamiento debe hacerse de forma atómica para evitar problemas de concurrencia.

Algoritmo 3 Corrección de Traslapes

Entrada: P_0 posiciones iniciales, E arcos de la triangulación
Salida: P_1 desplazamientos sobre cada partícula

```
1:  procedure CORRECTOVERLAPS(P_0, P_1, E)
2:      for i ← 0 to |E| do
3:          e_i ← E[i]
4:          b_i ← P_0[e_i.x]
5:          b_j ← P_0[e_i.y]
6:          r_ij ← dist(b_i, b_j)
7:          if r_ij < σ then
8:              δ ← σ − r_ij
9:              atomicAdd(P_1[e_i.x], −δ · r_ij)
10:             atomicAdd(P_1[e_i.y], δ · r_ij)
11:         end if
12:     end for
13: end procedure
```

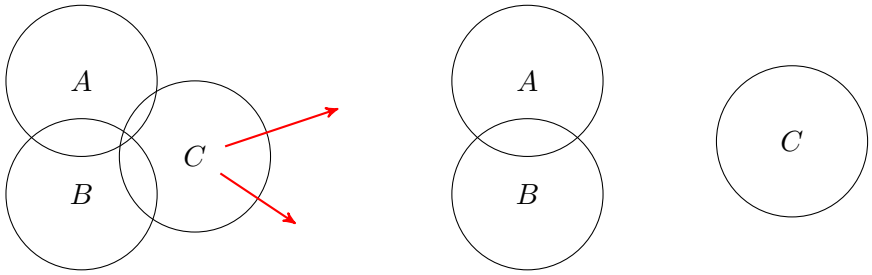
Una vez calculados los desplazamientos, un siguiente *kernel* los suma a las posiciones de cada partícula y aplica las condiciones de borde periódicas (si una partícula no participó en traslapes, el desplazamiento sumado a ella es nulo). Al actualizar las posiciones, es posible que se produzcan traslapes (ver figura 4.9) con nuevas partículas o que los traslapes actuales no hayan sido corregidos por completo, por lo que el proceso se debe realizar de forma iterativa hasta que no se detecten traslapes en el sistema.



(a) Situación inicial. (b) Tras la corrección.

Figura 4.9: Corrección de traslapes. Al aplicar el desplazamiento neto sobre B producto de A y C, puede ocurrir que los traslapes no queden resueltos.

Al sumar desplazamientos en un mismo sentido como se muestra en la figura 4.10, es posible que la partícula se esté moviendo mucho más de lo necesario para corregir el traslape, lo que puede llevar a la particula desplazada a penetrar de forma importante a otras partículas. Estos traslapes deberán ser corregidos en la siguiente iteración, lo que producirá un efecto de rebote que puede terminar llevando el sistema a una configuración incorrecta.



(a) Situación inicial. (b) Tras la corrección.

Figura 4.10: Corrección de traslapes en paralelo. El desplazamiento sobre C producto de los traslapes con A y B es mayor que el necesario, por lo que se trunca su magnitud.

La solución consiste en truncar el desplazamiento por la magnitud heurística $\sigma/4$, lo que previene movimientos demasiado grandes que produzcan situaciones como la descrita anteriormente. Este algoritmo no es equivalente a la implementación secuencial para la corrección de traslapes, por lo que no corresponde a una paralelización de ella. Por lo mismo, las posiciones finales de las partículas pueden ser distintas al ejecutar ambos algoritmos con entradas idénticas. Este algoritmo se implementó tanto en la solución desarrollada como en las existentes previamente, que no consideraban el truncado del desplazamiento neto. Esto impedía realizar simulaciones con pasos de tiempo grandes, ya que se producía el efecto de rebote descrito anteriormente.

4.4.6. Integración para fuerza de corto alcance

El cálculo de fuerzas para interacciones de corto alcance se hace utilizando listas de Verlet creadas y actualizadas en paralelo, con la estructura presentada en la sección 3.5. Los *threads* acceden a un arreglo de tamaño $N \cdot vec_{max}$ que contiene las listas de vecinos de cada partícula, en donde cada lista tiene un tamaño fijo vec_{max}. Esta constante depende del radio de corte r_c hasta donde se calcula la fuerza, por lo que a mayor valor de r_c, el tamaño de las listas de vecinos debe ser mayor.

Los elementos de la lista de vecinos corresponden a los índices de las partículas en el arreglo correspondiente, por lo que existe un sobrecosto de realizar accesos aleatorios en P, debido a que los elementos continuos en V no necesariamente lo están en P. Sin embargo, la cantidad de accesos a memoria realizados es menor que en el algoritmo para fuerza de largo alcance, debido a que solo se necesita comparar las partículas más cercanas en lugar de evaluar todo el sistema por cada partícula.

Algoritmo 4 Integración del Sistema de Partículas (fuerza de corto alcance)

Entrada: P_0 arreglo de partículas $(x_i, y_i, \alpha_i, \mu_i)$, C cantidades de vecinos por partícula, V lista de vecinos
Salida: P_1 partículas con posiciones actualizadas

```
1: procedure INTEGRATE(P_0, P_1)
2:     for i ← 0 to |P| do
3:         b_i ← P_0[i]
4:         v_i ← 0
5:         for j ← 0 to C[i] do
6:             k = V[i · vec_max + j]
7:             b_k = P_0[k]
8:             r_ik ← dist(b_i, b_k)
9:             if then r_ik ≤ r_c
10:                v_i ← v_i − r_ik · (α_i μ_k) / r_ik^7
11:            end if
12:        end for
13:    end for
14:    P_1[i] ← b_i v_i Δt + ξ_i √(T Δt)
15: end procedure
```

4.5. Interfaz de usuario

El programa se ejecuta desde la línea de comandos, al igual que la implementación secuencial existente. Se suministra como entrada el número de partículas, el tamaño del área de simulación, las concentraciones y valores de α, μ para cada tipo de partícula y el resto de constantes involucradas. Durante la simulación, el programa informa continuamente la cantidad de pasos de tiempo simulados y el tiempo físico que representan, hasta completar la cantidad de iteraciones especificada y traer de vuelta los resultados en GPU hacia un archivo de salida con la configuración final. Las visualizaciones de los sistemas simulados no forman parte del programa, y se obtienen con `Processing 2.2.1` leyendo los archivos producidos como resultado de las simulaciones.

También se puede leer una configuración inicial a partir de un archivo de texto plano previamente generado por una ejecución prueba del programa. En este caso, sigue siendo necesario entregar los parámetros de simulación en la línea de comandos.

Capítulo 5

Validación de la solución

Aquí se presentan los resultados de los experimentos que respaldan la solución conseguida, especificando la metodología y equipos utilizados para la medición. Además, se interpretan los resultados obtenidos para verificar si son consistentes con lo esperado y pueden ser explicados a partir de las propiedades de los sistemas estudiados.

5.1. Experimento

Para validar las soluciones desarrolladas, se evaluó el rendimiento de las implementaciones para fuerzas de corto y largo alcance, midiendo la ejecución de una iteración de simulación (que corresponde a un paso de tiempo). También se midió la cantidad de iteraciones para la corrección de traslapes, es decir, las iteraciones necesarias para corregir todos los traslapes. Para la fuerza de largo alcance, además se midió la cantidad promedio de iteraciones de *edge-flips* por paso de tiempo, ya sea para corregir triángulos invertidos o para actualizar la triangulación.

Entrada: Se generaron entradas para 11 valores distintos de N y 5 configuraciones de parámetros distintas, las cuales se describen en la tabla 5.1. Las posiciones iniciales de las partículas se generan aleatoriamente de acuerdo al método descrito en el capítulo 4, manteniendo constante para todas las ejecuciones el valor de semilla utilizado para iniciar el generador de números aleatorios.

config	ϕ_1	α_1	μ_1	ϕ_2	α_2	μ_2	ρ
0	0.7	1	1	0.3	1	-1	0.79
1	0.5	1	1	0.5	1	-1	0.52
2	0.5	1	1	0.5	-1	-2	0.52
3	0.5	-1	1	0.5	1	-2	0.79
4	0.5	1	1	0.5	-1	-4	$8.73 \cdot 10^{-2}$

Tabla 5.1: Configuraciones usadas en las pruebas, en donde cada configuración está identificada por un dígito y tiene dos tipos de partículas. ϕ_i corresponde a la fracción de partículas del tipo i, α_i, μ_i son valores utilizados en el cálculo de fuerzas y ρ es la fracción de volúmen del sistema ocupado por partículas.

Cada configuración tiene dos tipos de partículas con valores α_i, μ_i y una concentración ϕ_i, que corresponde a la fracción de partículas de ese tipo presente en el sistema. Además existe una densidad ρ del sistema, que corresponde a la fracción del espacio ocupada por partículas y está determinada por:

$$\rho = \frac{N\pi(\sigma/2)^2}{L^2},$$

por lo que para mantener ρ constante en cada configuración, se obtiene $L(N) = \sqrt{\frac{N\pi(\sigma/2)^2}{\rho}}$.

Los valores de N parten en 2^{10}, aumentando el exponente en 1 hasta llegar a $N = 2^{20}$; esto se hace para aprovechar toda la capacidad de la GPU en cada simulación. Por último, los valores para $\Delta t = 0.01, T = 0.01, \sigma = 1.0$ se mantienen constantes para todas las configuraciones y valores de N.

Sistema: Se utilizó una tarjeta de video Tesla K40c con 15 multiprocesadores. Además se utilizó un procesador Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz en el mismo equipo para comparar el rendimiento de la implementación secuencial.

Compilación: Se compiló el programa utilizando `nvcc V7.5.17` con las opciones `--std=c++11 -gencode arch=compute_30,code=sm_30`, mientras que para la parte secuencial se utilizó `g++ 4.9.2` con las opciones `--std=c++11 -O3`.

Métrica: Se ejecutaron simulaciones de 100 iteraciones, reportando el promedio de tiempo y la cantidad media de iteraciones para las mediciones respectivas. La razón para considerar el promedio

se debe a que las partículas pueden demorar algunas iteraciones en comenzar a colisionar más frecuentemente con el resto del sistema.

5.2. Resultados

A continuación se presentan las visualizaciones y gráficos obtenidos como resultado del experimento anteriormente descrito.

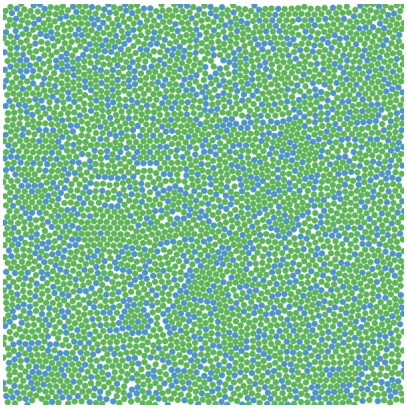
5.2.1. Simulación de sistemas de partículas

Las configuraciones descritas en la tabla 5.1 pueden parecer similares entre sí, ya que todas simulan solo dos tipos de partículas e incluso algunas de ellas comparten algunos valores. Esto puede llevar a pensar que los sistemas representados pueden ser igualmente similares, por lo que las visualizaciones en la figura 5.1 permiten reconocer las diferencias entre cada configuración.

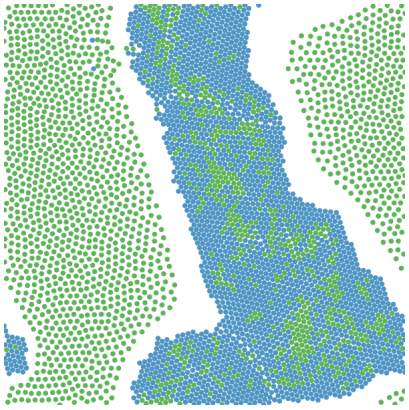
Se puede notar que C_0 es la única configuración en la que las partículas se distribuyen uniformemente a lo largo del sistema, mientras que en todas las demás existen espacios vacíos. Esto permite suponer que el costo de la corrección de traslapes puede variar según la configuración, ya que en algunas de ellas se aprecian grupos de partículas más compactas en el espacio.

C_1 es similar a C_3, ya que en ambos casos se crean grupos con mayoría de alguno de los dos tipos de partículas. Además, se forma una especie de membrana que separa los dos grupos y previene que se formen contactos entre ellos. Las diferencias consisten en que C_3 es más densa y las partículas celestes en ella se compactan más, a diferencia de las partículas verdes en C_1 que no se contactan entre sí.

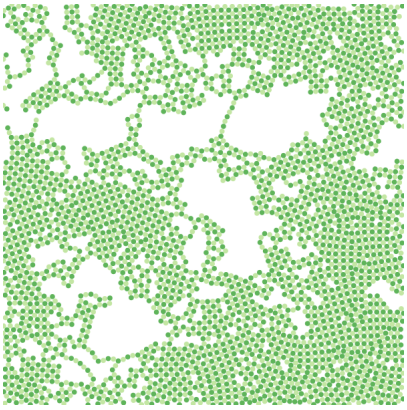
En C_3 se pueden apreciar filamentos que unen grupos de partículas a través de las zonas vacías. También se puede notar que las partículas de cada tipo se intercalan entre sí de forma muy regular, y que las partículas de un tipo solo se encuentran en contacto con objetos del otro tipo. Finalmente, C_4 corresponde a una configuración con densidad muy baja con respecto a las demás, pero presenta un entramado entre ambos tipos de partículas similar a lo que ocurre en C_2



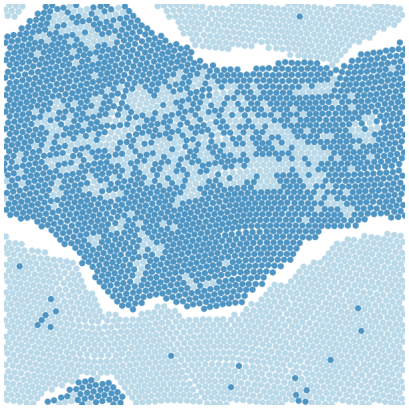
(a) Configuración 0



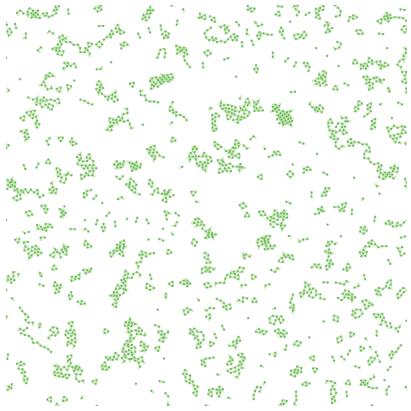
(b) Configuración 1



(c) Configuración 2



(d) Configuración 3



(e) Configuración 4

Figura 5.1: Sistemas de partículas tras 10000 iteraciones con $N = 4096$ y $dt = 0.01$. Los colores de las partículas denotan el tipo al que pertenecen. Las configuraciones de cada sistema y los valores respectivos para cada tipo de partículas corresponden a los descritos en la tabla 5.1.

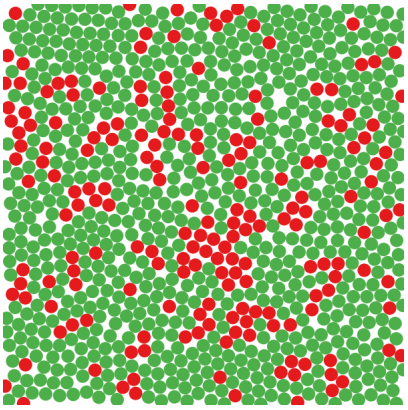
5.2.2. Localidad de corrección de traslapes

Las visualizaciones anteriores permiten validar visualmente que la corrección de traslapes implementada cumple con su objetivo. Sin embargo, también se desea examinar el supuesto formulado en la sección anterior, que consiste en detectar las diferencias en la corrección de traslapes para las configuraciones anteriores. Una forma de detectar esto es visualizando la localidad de la corrección en cada caso, es decir, mirar qué tan lejos se propagan la corrección de traslapes a través de los sistemas de partículas respectivos.

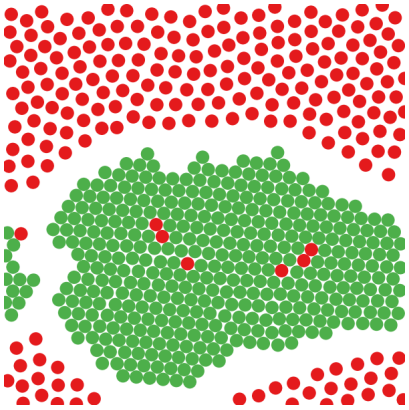
Para cada configuración, se visualizó el resultado final de 10000 pasos de tiempo para cada configuración, marcando con verde las partículas que participaron en alguna corrección de traslapes durante el último paso de tiempo simulado. Las partículas que no participaron en traslapes se marcaron en rojo, por lo que todo el sistema queda marcado con algún color. Se repiten las visualizaciones para distintos Δt, pues se espera que a menor valor del paso de tiempo, los movimientos producto de la integración sean más pequeños y haya menos traslapes que corregir en total. Los resultados obtenidos se presentan en las figuras 5.2, 5.3 y 5.4.

5.2.3. Gráficos

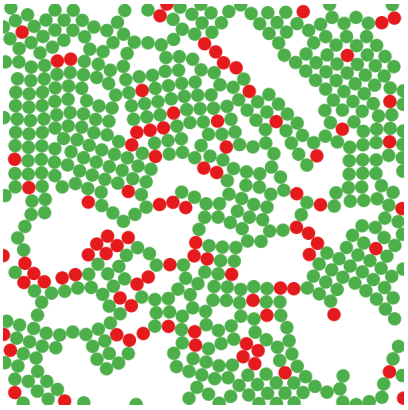
A continuación se presentan los resultados de rendimiento del experimento mencionado al inicio de este capítulo para la solución implementada, iniciando por el promedio de tiempo (real) de un paso de tiempo (físico) en la figura 5.5 para las fuerzas de largo y corto alcance. Si bien las interacciones son distintas y no comparables entre sí, ambas simulaciones utilizan el mismo método de corrección de traslapes, que corresponde al tema principal de análisis.



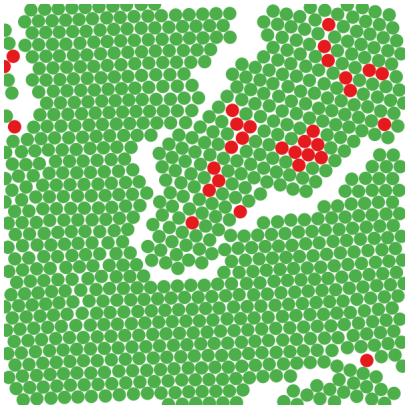
(a) Configuración 0



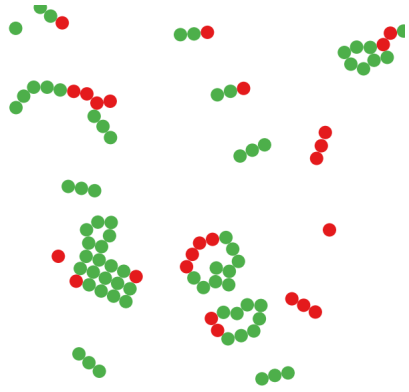
(b) Configuración 1



(c) Configuración 2

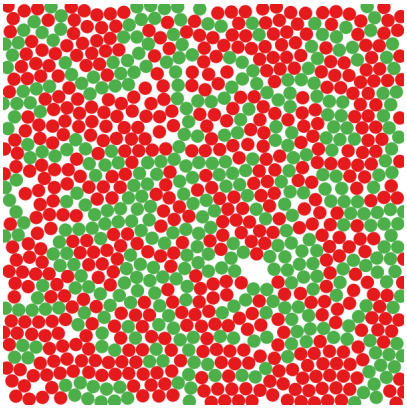


(d) Configuración 3

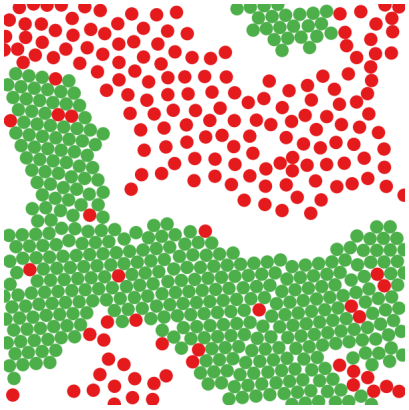


(e) Configuración 4

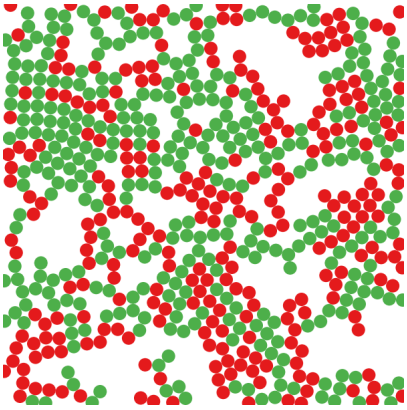
Figura 5.2: Gráficos de localidad para corrección de traslapes tras 10000 iteraciones con $dt = 0.01$. Las partículas verdes participaron en al menos una corrección de traslapes en un mismo paso de tiempo, mientras que las rojas no lo hicieron. Las configuraciones de cada sistema corresponden a las descritas en la tabla 5.1.



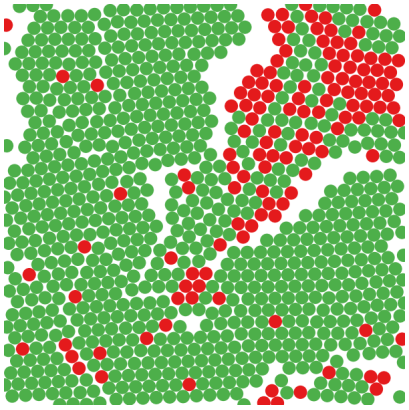
(a) Configuración 0



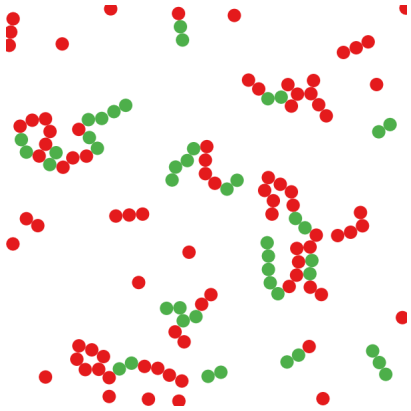
(b) Configuración 1



(c) Configuración 2

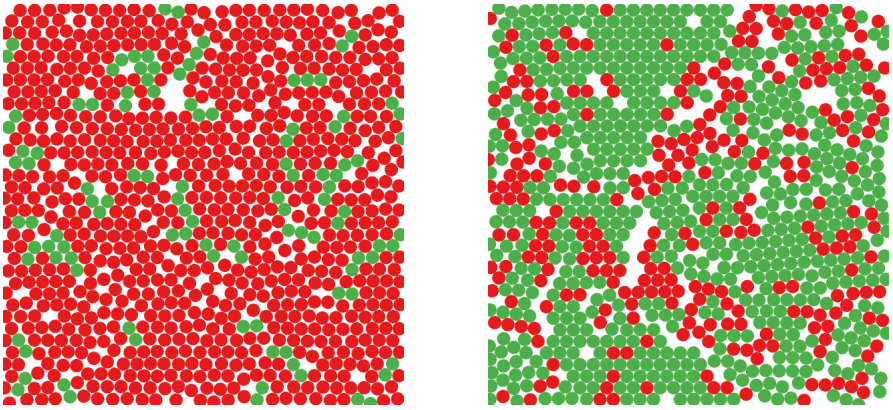


(d) Configuración 3



(e) Configuración 4

Figura 5.3: Gráficos de localidad de corrección de traslapes para $dt = 0.001$, en donde los demás parámetros son los mismos que en la figura 5.2.



(a) Configuración 0 (b) Configuración 3

Figura 5.4: Gráficos de localidad de corrección de traslapes para $dt = 0.0001$, en donde los demás parámetros son los mismos que en la figura 5.2.

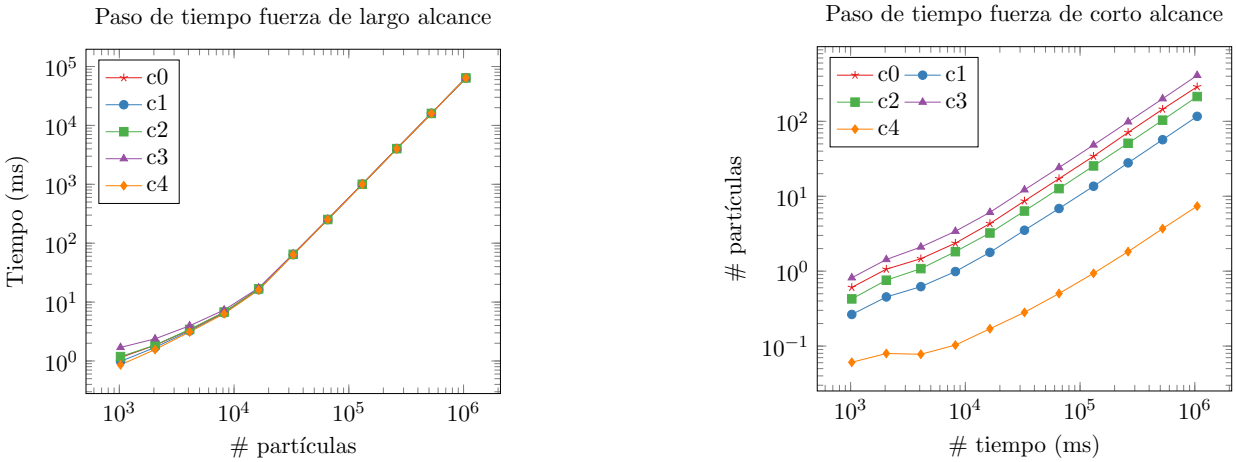


Figura 5.5: Tiempos de ejecución en milisegundos para los algoritmos de largo y corto alcance. Las configuraciones de cada sistema y los valores respectivos para cada tipo de partículas corresponden a los descritos en la tabla 5.1.

Para el caso de largo alcance, el costo de la simulación depende de cada configuración solo al comienzo, y desde $N \sim 10^4$ en adelante es dominado por el cálculo de fuerzas. Por otro lado, para la fuerza de corto alcance la influencia de las configuraciones se hace más notoria y se mantiene constante a medida que aumenta el número de partículas. En la figura 5.6 se intenta despejar los prefactores para las fuerzas de corto y largo alcance, que corresponden a los valores de dificultad para corregir los traslapes en cada configuración. Se debe notar que las magnitudes de ambas fuerzas estudiadas son distintas, lo que produce la diferencia entre las curvas del tiempo de ejecución en cada caso.

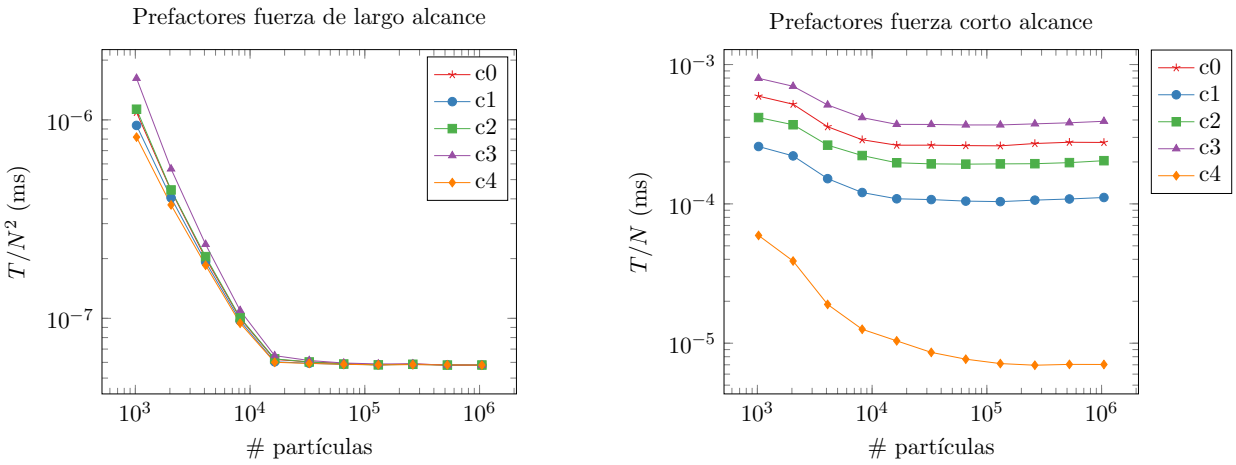


Figura 5.6: Influencia de las distintas configuraciones sobre el tiempo de ejecución para simular un paso de tiempo físico, con los mismos parámetros que los utilizados en la figura 5.5.

En ambos casos se obtiene el mismo orden de configuraciones, partiendo desde menor a mayor complejidad: C_4, C_1, C_2, C_0, C_3. Esto es consistente con los resultados de los gráficos de localidad mostrados anteriormente, ya que en C_3 la mayor parte del sistema participa en la corrección de traslapes, mientras que en C_2 casi la mitad del sistema queda excluido debido a la repulsión entre partículas del mismo tipo. Sin embargo, esto no explica por qué C_4 tiene el menor prefactor de complejidad por un margen muy amplio con respecto al resto, aún cuando la mitad de sus partículas participan en traslapes al igual que en C_2. Para dilucidar esto, en la figura 5.7 se muestran las cantidades promedio de iteraciones de corrección de traslapes, ya que estos valores están directamente relacionados con el tiempo de ejecución en el algoritmo.

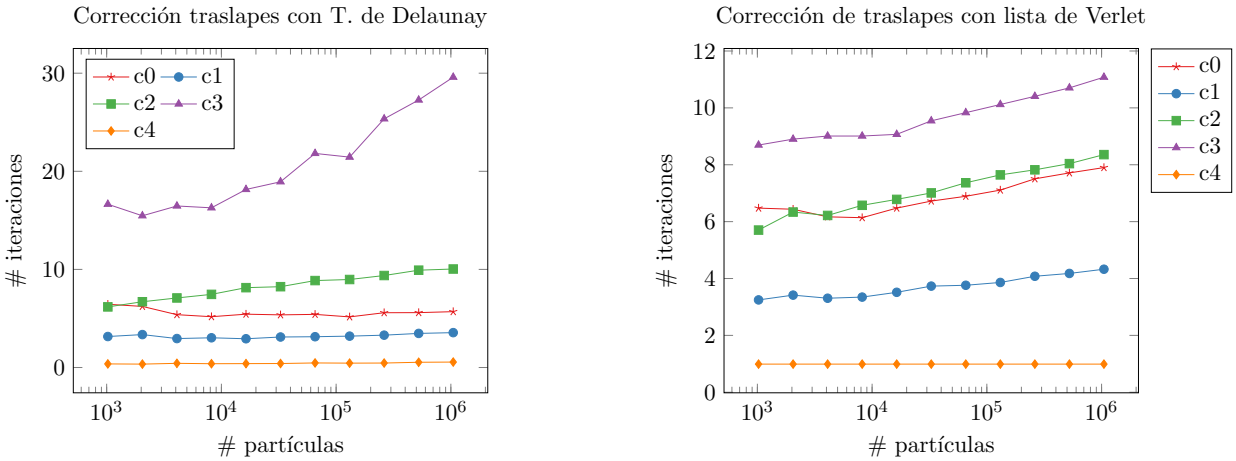


Figura 5.7: Promedio de iteraciones por paso de tiempo para fuerzas de largo y corto alcance, con los mismos parámetros que en la figura 5.5.

Aquí se puede ver que si bien la mayoría de las partículas en C_4 participa en traslapes, el sistema queda completamente corregido en la primera iteración. También se observa que las curvas para la fuerza de largo alcance son constantes en C_4, C_1 y C_0, es ligeramente creciente en C_2 y es mucho mayor al resto en C_3 (además de tener el mayor crecimiento).

También se midió el rendimiento del algoritmo para actualizar la triangulación de Delaunay, midiendo el promedio de las iteraciones de *edge-flip* realizadas para corregir triángulos invertidos o satisfacer la condición de Delaunay. No se realizó el gráfico correspondiente para la fuerza de corto alcance, pues esa implementación no utiliza triangulaciones.

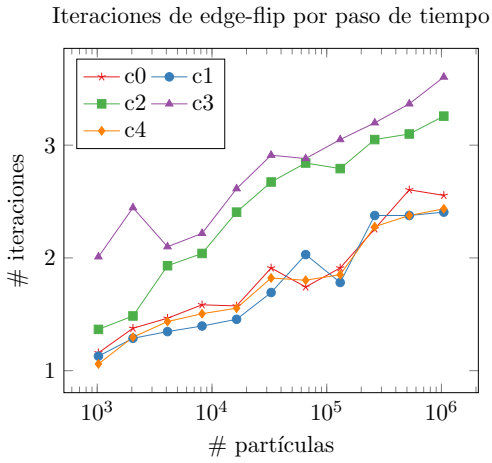


Figura 5.8: Promedio de iteraciones de *edge-flips* por paso de tiempo para algoritmo de fuerza de largo alcance, con los mismos parámetros que en la figura 5.5.

Las curvas obtenidas en la figura 5.8 son menos regulares que en los gráficos anteriores, pero se sigue apreciando una tendencia general. A diferencia del gráfico para corrección de traslapes, en donde C_4 presenta una cantidad muy inferior a las demás configuraciones, aquí la curva es comparable con C_0 y C_1. Esto se debe a que la triangulación representada en C_4 presenta una gran cantidad de astillas o *slivers*, los cuales se pueden ver en la figura 5.9 y que corresponden a triángulos muy delgados, con ángulo mínimo pequeño y que aparecen debido a la distancia entre vértices. Luego, de acuerdo a la condición de triángulo invertido vista en la sección 4.4.4, es más probable que se produzcan triángulos invertidos en C_4 que en las demás configuraciones, las cuales presentan triángulos más equiláteros.

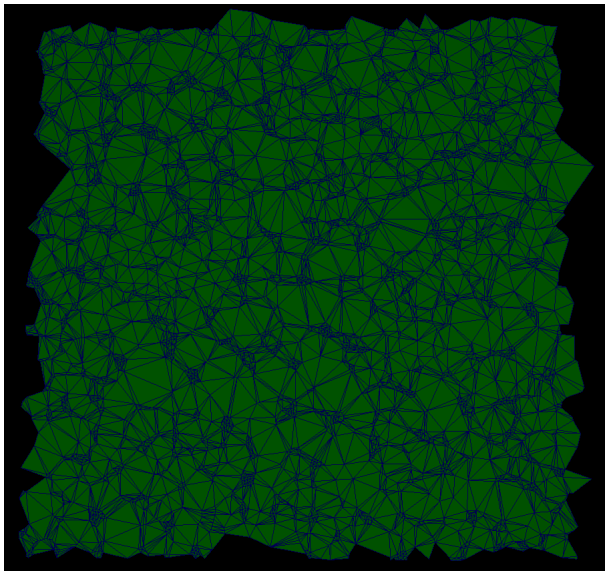


Figura 5.9: Triangulación de Delaunay con $N = 4096$ para la configuración 4 descrita en la tabla 5.1.

Finalmente, se compararon los algoritmos de *n-body* utilizados en las distintas implementaciones disponibles, sin considerar la corrección de traslapes. `shuffle` corresponde a la implementación desarrollada en este trabajo y descrita en la sección 4.4.2, `sharedMem` es el algoritmo utilizado en la solución paralela previa ya existente, y `CPU` corresponde a la solución secuencial, ambas descritas en la sección 3.5. Este es el único gráfico en donde se compara la solución secuencial y el algoritmo de lista de Verlet no se considera, ya que esta última simula una fuerza de corto alcance distinta a la de largo alcance en los demás algoritmos.

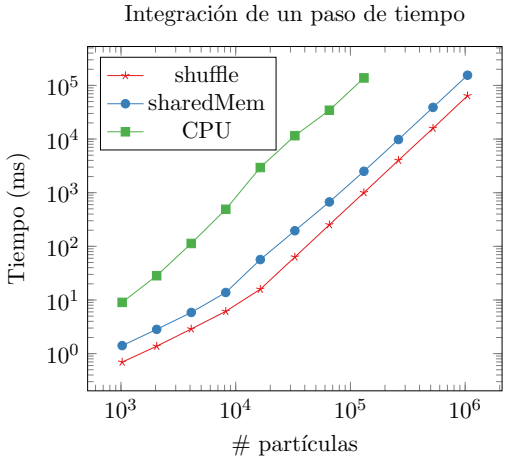


Figura 5.10: Comparación entre los tiempos de ejecución en milisegundos para las implementaciónes del método *n-body* de $O(N^2)$, utilizando la configuración C_0 descrita en la tabla 5.1.

Se aprecia una mejora de hasta $\times 2.41$ para $N = 10^6$ entre la solución paralela previa y la nueva implementación, que se mantiene constante para los demás valores de N debido a que ambos son implementaciones del mismo algoritmo de costo cuadrático. También cabe destacar que el tiempo del algoritmo secuencial para $N \sim 10^5$ es comparable al algoritmo paralelo implementado con una cantidad de partículas mayor en dos órdenes de magnitud.

5.3. Análisis

El tiempo de ejecución para un paso de tiempo físico es dominado por el cálculo de fuerzas en el caso de largo alcance, mostrando que la influencia de la corrección de traslapes y las distintas configuraciones deja de ser importante más alla de $N = 10^4$. Por el contrario, para la fuerza de corto alcance, el factor de complejidad de cada configuración se mantiene constante a medida que N aumenta. Esto permite verificar que el costo de la corrección de traslapes es $O(N)$.

El promedio de iteraciones de *edge-flip* en el algoritmo de largo alcance aumenta linealmente y de forma similar para todas las configuraciones, manteniendo la tendencia en que C_3 y C_4 son los casos más difíciles de resolver y, por lo tanto, tienen prefactores de complejidad más altos. Si bien la cantidad de iteraciones aumenta con N, sigue siendo despreciable con respecto al tiempo total de un paso de simulación, por lo que no es una prioridad para optimizar.

El promedio de iteraciones de traslape aparece constante para las configuraciones C_0, C_1 y C_4, aumenta ligeramente para C_2 y es muy superior para C_3. Este comportamiento significa que C_3 la configuracion produce un sistema percolante, es decir, una corrección local de traslapes termina propagándose a todo el sistema. Lo mismo ocurre en menor medida para C_2, donde los traslapes probablemente se propagan a través de los filamentos que se conectan todo el sistema. Estos casos muestran que la densidad no es el único factor que incide en el tiempo de la corrección de traslapes, ya que C_0 y C_3 presentan la misma fracción de empaquetamiento.

En las visualizaciones de localidad para la corrección de traslapes, se verifica que para la configuración 3 la mayor parte del sistema participa en traslapes, lo que se traduce en más iteraciones de corrección. Por otro lado, en C_1 casi la mitad de las partículas permanece en rojo, lo que es consistente con el hecho de que se necesita iterar menos de la mitad de las veces que en C_3 para corregir todos sus traslapes. Finalmente, se verifica el supuesto que al bajar la magnitud de Δt se producen menos traslapes por paso de tiempo, debido a que los movimientos de partículas bajan

consecuentemente. Esto no quiere decir que utilizar pasos de tiempo bajos reduzca el tiempo de ejecución (de hecho, lo aumenta), ya que el costo del algoritmo es dominado en más del 85 % por el calculo de fuerzas de acuerdo al análisis de *profiling*.

Capítulo 6

Conclusiones

Aquí se hace un resumen del trabajo realizado y los objetivos logrados, tomando en consideración los resultados obtenidos en la sección 5. También se comenta la experiencia obtenida con la realización de este trabajo, así como las herramientas utilizadas que se consideraron cruciales durante su desarrollo. Además, se enuncian ejemplos del trabajo futuro que puede realizarse a partir de esta solución.

6.1. Resumen

En este trabajo se diseñaron e implementaron algoritmos para simular interacciones de corto y largo alcance, utilizando listas de Verlet y triangulaciones de Delaunay respectivamente para almacenar las vecindades de cada partícula. Ambos algoritmos fueron implementados en CUDA, y realizan la simulación completamente en GPU sin necesidad de transferir datos en los pasos intermedios, dejando únicamente la inicialización del sistema como ejecución secuencial. Los algoritmos además comparten la misma estrategia de resolución de traslapes, evitando realizar comparaciones inútiles con partículas lejanas que no tienen posibilidad de participar en un traslape.

La implementación con triangulación de Delaunay utiliza un algoritmo de *edge-flipping* en paralelo, para mantener las propiedades topológicas que permiten realizar la corrección de traslapes eficientemente. Esto consiste en evaluar colisiones únicamente entre partículas que estén conectadas entre sí por arcos de la triangulación, sumando a una misma partícula las contribuciones de todas las partículas cercanas que traslapan con ella. El desplazamiento neto debe ser truncado por $\sigma/4$ para evitar movimientos excesivos que hagan a una partícula colisionar con otros objetos en la dirección

que se hizo el desplazamiento, lo cual no afecta negativamente el rendimiento de esta función. Estos movimientos no son reales, sino que simulan la propiedad de volúmen excluido entre las partículas.

Para ambas implementaciones se realizaron pruebas con cinco configuraciones distintas, las que corresponden a casos de estudio reales y a partir de los cuales se pretende realizar mediciones. El tiempo de ejecución para la fuerza de largo alcance es dominado por el cálculo de fuerzas $O(N^2)$ al igual que en la implementación secuencial, pero con tiempos mucho menores para los mismos valores de N. La corrección de traslapes logra hacerse de forma eficiente, pero el tiempo de ejecución de la simulación termina siendo dominado por el algoritmo de cálculo de fuerzas. También se comprueba que la corrección de traslapes se hace de forma local para la mayoría de los casos, pero existen configuraciones que obligan a propagar la corrección a través de todo el sistema.

6.2. Recuento de objetivos

Se cumplió el objetivo de diseñar e implementar un algoritmo paralelo que mejore sustancialmente a la solución secuencial, presentando tiempos de ejecución menores en dos órdenes de magnitud a partir de $N = 2^14$ con respecto a la primera. Esto permite simular sistemas de hasta 100 veces más partículas en la misma cantidad de tiempo que la solución secuencial. Sin embargo, debido a la complejidad computacional del algoritmo de largo alcance, no es posible realizar simulaciones con $N > 10^6$ de forma tan eficiente como se hubiera deseado, ya que los tiempos de una sola iteración son del orden de minutos. Por otro lado, el uso de memoria utilizada crece linealmente con respecto a N, por lo que es poco probable que este tema sea un problema.

También se pudo comprobar la factibilidad de utilizar la triangulación de Delaunay y el algoritmo de *edge-flip* paralelo para resolver la corrección de traslapes eficientemente, lo que proporciona una aplicación práctica a un algoritmo que no la tenía hasta antes de este trabajo. Esto abre la posibilidad de usar la triangulación de Delaunay para resolver otros problemas relacionados, como el cálculo de fuerzas de corto alcance o simulaciones de *n-body* aproximadas.

Finalmente, se realizaron exitosamente los experimentos que le dan sustento práctico a lo anterior, pudiendo reconocer relaciones como la influencia de las distintas configuraciones sobre el tiempo de ejecución del programa, el peor caso para la resolución de traslapes y la verificación de que el algoritmo implementado es efectivamente local con respecto a las partículas.

6.3. Aprendizaje

La mayor ganancia en experiencia se obtuvo en la utilización de CUDA, ya que además de los algoritmos implementados en este trabajo, se aprendieron optimizaciones e idiomas propios del lenguaje, uso de las herramientas de *profiling*, ventajas y desventajas distintos tipos de memoria y modos de acceso, entre otros. Sin dudas el nivel de dominio sobre la plataforma es mucho mayor que al comienzo de este trabajo, cuando solo se tenía conocimientos básicos a partir de programas de prueba y ejemplos de manual.

Aprender a utilizar CUDA no solo consistió en programar con un nuevo lenguaje, sino que además incluye acostumbrarse a un paradigma de programación y a todo lo que eso implica, ya que algunas de las estrategias utilizadas para diseñar algoritmos secuenciales no son válidas para el caso paralelo. De hecho, hay algoritmos como la corrección de traslapes implementada que no son directamente paralelizables a partir de la implementación secuencial, por lo que se debe rediseñar todo desde cero.

Esta experiencia además demostró que es imprescindible mantener un control riguroso sobre el *testing* y la detección de errores (por ejemplo, utilizando el *debugger* de CUDA), ya que la complejidad de la aplicación desarrollada y la posibilidad de encontrar problemas crece mucho más rápido comparado a la mayoría de aplicaciones secuenciales.

Por otro lado, este proyecto resultó como una buena aproximación al mundo de la investigación, al haber presentado resultados novedosos que pueden servir como punto de partida a soluciones más complejas o eficientes que los encontrados. Asimismo, se espera que el programa producido pueda servir para la investigación sobre partículas coloidales que motivó el desarrollo de este proyecto.

6.4. Trabajo futuro

Existe un cuello de botella considerable en el tiempo de ejecución, debido al algoritmo de n-body cuadrático y que impide simular eficientemente cantidades de partículas superiores al millón. Como las distancias entre objetos aumentan a medida que se consideran sistemas de mayor tamaño, las contribuciones de partículas lejanas son cada vez menores y se hace innecesario calcularlas con tanta precisión. De aquí surge la necesidad de implementar un algoritmo de aproximación, similar a la simulación de Barnes-Hut pero utilizando triangulaciones de Delaunay con distintas resoluciones en

lugar de particionar el espacio. Esto significa que los nodos de una triangulación de nivel superior pueden representar a varias partículas, y que la fuerza entre objetos lejanos pueda aproximarse como una interacción entre la partícula y un nodo de menor resolución.

También es posible utilizar la triangulación de Delaunay en GPU para implementar interacciones de corto alcance, extendiendo las estructuras de datos para acceder eficientemente a todas las partículas vecinas o sus vecinos de nivel superior. Una forma de lograr esto es extendiendo la estructura de Delaunay para moverse eficientemente por la triangulación: por ejemplo, asociando cada arco a un triángulo adyacente respectivo o implementando *half-edges*. Esto además podría usarse para mejorar la detección de triángulos invertidos, ya que actualmente tiene un alcance limitado.

Otra extensión posible corresponde a implementar la capacidad de realizar simulaciones en 3 dimensiones, debido a que los componentes que forman parte de la solución son todos extensibles al caso 3D, incluyendo la triangulación de Delaunay periódica en 3D [5] y el algoritmo de *edge-flip* paralelo.

Por otro lado, se puede aprovechar la interoperabilidad entre CUDA y OpenGL para producir visualizaciones de la simulación directamente desde la GPU, sin necesidad de transferir los datos a memoria principal (con el tremendo sobrecosto que implica) ni modificar considerablemente las estructuras de datos paralelas existentes en la implementación actual. Esto permitiría visualizar en tiempo real sistemas del orden de 10^4 partículas [14], con propósitos ilustrativos o para detectar errores.

Apéndices

Aquí se muestran las tablas que contienen los valores mostrados en los gráficos de la sección 5.

A . Tablas

N	C_0	C_1	C_2	C_3	C_4
1024	1.14	0.98	1.19	1.7	0.86
2048	1.85	1.7	1.86	2.38	1.56
4096	3.37	3.23	3.43	3.96	3.11
8192	6.71	6.51	6.77	7.35	6.34
16384	16.67	16.21	16.77	17.41	16.12
32768	64.79	63.78	64.73	65.86	63.64
65536	$2.54 \cdot 10^2$	$2.52 \cdot 10^2$	$2.53 \cdot 10^2$	$2.56 \cdot 10^2$	$2.52 \cdot 10^2$
131072	$1 \cdot 10^3$	$1 \cdot 10^3$	$1 \cdot 10^3$	$1.01 \cdot 10^3$	$1 \cdot 10^3$
262144	$4.05 \cdot 10^3$	$4.03 \cdot 10^3$	$4.03 \cdot 10^3$	$4.05 \cdot 10^3$	$4.02 \cdot 10^3$
524288	$1.6 \cdot 10^4$	$1.6 \cdot 10^4$	$1.6 \cdot 10^4$	$1.6 \cdot 10^4$	$1.6 \cdot 10^4$
1048576	$6.38 \cdot 10^4$	$6.39 \cdot 10^4$	$6.4 \cdot 10^4$	$6.39 \cdot 10^4$	$6.39 \cdot 10^4$

Tabla 6.1: Tiempos de ejecución promedio en milisegundos de un paso de tiempo para fuerza de largo alcance. N es la cantidad de partículas simuladas y $C_0 - C_4$ corresponden a las configuraciones de cada sistema, descritas en la tabla 5.1.

N	C_0	C_1	C_2	C_3	C_4
1024	6.45	3.16	6.18	16.64	0.37
2048	6.23	3.36	6.7	15.47	0.35
4096	5.39	2.95	7.09	16.47	0.43
8192	5.18	3.03	7.46	16.28	0.39
16384	5.44	2.93	8.14	18.15	0.4
32768	5.37	3.11	8.24	18.93	0.41
65536	5.42	3.14	8.86	21.82	0.47
131072	5.16	3.2	8.97	21.45	0.45
262144	5.58	3.3	9.38	25.35	0.46
524288	5.59	3.48	9.92	27.27	0.53
1048576	5.69	3.55	10.04	29.59	0.55

Tabla 6.2: Cantidad de traslapes promedio por paso de tiempo para fuerza de largo alcance, con los mismos parámetros que en la tabla 6.1.

N	C_0	C_1	C_2	C_3	C_4
1024	1.16	1.13	1.37	2.01	1.06
2048	1.38	1.29	1.49	2.45	1.3
4096	1.47	1.35	1.93	2.1	1.44
8192	1.58	1.4	2.04	2.22	1.5
16384	1.57	1.46	2.41	2.61	1.55
32768	1.91	1.69	2.67	2.91	1.82
65536	1.74	2.03	2.84	2.88	1.8
131072	1.91	1.78	2.79	3.05	1.85
262144	2.26	2.38	3.05	3.2	2.28
524288	2.6	2.38	3.1	3.37	2.38
1048576	2.55	2.41	3.26	3.6	2.44

Tabla 6.3: Cantidad de *edge-flips* promedio por paso de tiempo para fuerza de largo alcance, con los mismos parámetros que en la tabla 6.1.

N	C_0	C_1	C_2	C_3	C_4
1024	0.61	0.26	0.43	0.82	$6.08 \cdot 10^{-2}$
2048	1.06	0.45	0.76	1.43	$7.96 \cdot 10^{-2}$
4096	1.47	0.62	1.08	2.1	$7.77 \cdot 10^{-2}$
8192	2.36	0.99	1.82	3.41	0.1
16384	4.32	1.78	3.23	6.1	0.17
32768	8.65	3.52	6.35	12.18	0.28
65536	17.17	6.86	12.64	24.15	0.5
131072	34.2	13.6	25.38	48.32	0.94
262144	71.09	27.89	50.92	98.39	1.82
524288	$1.45 \cdot 10^2$	56.84	$1.04 \cdot 10^2$	$2 \cdot 10^2$	3.7
1048576	$2.89 \cdot 10^2$	$1.17 \cdot 10^2$	$2.14 \cdot 10^2$	$4.11 \cdot 10^2$	7.38

Tabla 6.4: Tiempos de ejecución promedio en milisegundos por paso de tiempo para fuerza de corto alcance, con los mismos parámetros que en la tabla 6.1.

N	C_0	C_1	C_2	C_3	C_4
1024	6.48	3.25	5.7	8.69	0.99
2048	6.44	3.42	6.34	8.9	0.99
4096	6.17	3.31	6.22	9.01	0.99
8192	6.14	3.35	6.57	9.01	0.99
16384	6.48	3.51	6.78	9.07	0.99
32768	6.72	3.73	7.01	9.54	0.99
65536	6.89	3.76	7.37	9.83	0.99
131072	7.11	3.86	7.64	10.12	0.99
262144	7.5	4.08	7.82	10.41	0.99
524288	7.71	4.18	8.04	10.7	0.99
1048576	7.9	4.33	8.36	11.08	0.99

Tabla 6.5: Cantidad de traslapes promedio por paso de tiempo para fuerza de corto alcance, con los mismos parámetros que en la tabla 6.1.

N	shuffle	sharedMem	CPU
1024	0.69	1.41	9.03
2048	1.38	2.83	28.36
4096	2.88	5.86	$1.13 \cdot 10^2$
8192	6.14	13.79	$4.91 \cdot 10^2$
16384	15.87	56.68	$2.95 \cdot 10^3$
32768	63.19	$1.95 \cdot 10^2$	$1.15 \cdot 10^4$
65536	$2.52 \cdot 10^2$	$6.73 \cdot 10^2$	$3.44 \cdot 10^4$
131072	$1 \cdot 10^3$	$2.51 \cdot 10^3$	$1.38 \cdot 10^5$
262144	$4.02 \cdot 10^3$	$9.8 \cdot 10^3$	
524288	$1.59 \cdot 10^4$	$3.89 \cdot 10^4$	
1048576	$6.41 \cdot 10^4$	$1.55 \cdot 10^5$	

Tabla 6.6: Comparación de los tiempos de ejecución promedio en milisengundos entre los distintos métodos de n-body cuadrático. Los tiempos del algoritmo secuencial no se midieron para todos los valores de N para no detener por un tiempo prolongado el servidor remoto en el que se realizaron las pruebas.

Bibliografía

[1] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[2] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the gpu processor. *J. Comput. Phys.*, 231(7):2825–2839, April 2012.

[3] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd edition, 2008.

[4] M. Burtscher and K. Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU Gems '11: GPU Computing Gems Emerald Edition*, chapter 6, pages 75–92. 2011.

[5] Manuel Caroli and Monique Teillaud. 3D periodic triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.8.1 edition, 2016.

[6] NVIDIA Corporation. *CUDA C Programming Guide*. Santa Clara, CA, USA, 2015.

[7] K. A. Hawick and D. P. Playne. Hard-sphere collision simulations with multiple gpus, pcie extension buses and gpu-gpu communications. In *Proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing - Volume 127*, AusPDC '12, pages 13–22, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc.

[8] Nico Kruithof. 2D periodic triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.8.1 edition, 2016.

[9] Charles L. Lawson. Transforming triangulations. *Discrete Mathematics*, 3(4):365 – 372, 1972.

[10] Tyson J. Lipscomb, Anqi Zou, and Samuel S. Cho. Parallel verlet neighbor list algorithm for gpu-optimized md simulations. In Sanjay Ranka, Tamer Kahveci, and Mona Singh, editors, *BCB*, pages 321–328. ACM, 2012.

[11] Cristobal Navarro, Nancy Hitschfeld, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics (CiCP)*, 15:285–329, Feb 2014.

[12] Cristobal A. Navarro, Nancy Hitschfeld-Kahler, and Eliana Scheihing. A GPU-based Method for Generating quasi-Delaunay Triangulations based on Edge-flips. In Sabine Coquillart, Carlos Andújar, Robert S. Laramee, Andreas Kerren, and José Braz, editors, *GRAPP/IVAPP*, pages 27–34. SciTePress, 2013.

[13] NVIDIA. *CURAND Library: Programming Guide, Version 7.0*, 2015.

[14] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with cuda. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 31, pages 677–695. Addison-Wesley, 2008.

[15] Andrew J. Proctor, Cody A. Stevens, and Samuel S. Cho. Gpu-optimized hybrid neighbor/cell list algorithm for coarse-grained md simulations of protein and rna folding and assembly. In *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, BCB'13, pages 633:633–633:640, New York, NY, USA, 2013. ACM.

[16] Maxi San Miguel and Raúl Toral. *Stochastic Effects in Physical Systems*, pages 35–127. Springer Netherlands, Dordrecht, 2000.

[17] Rodrigo Soto and Ramin Golestanian. Self-assembly of catalytically active colloidal molecules: Tailoring activity through surface chemistry. *Phys. Rev. Lett.*, 112:068301, Feb 2014.

[18] Rodrigo Soto and Ramin Golestanian. Self-assembly of active colloidal molecules with dynamic function. *Phys. Rev. E*, 91:052304, May 2015.

[19] P. Strating. Brownian dynamics simulation of a hard-sphere suspension. *Phys. Rev. E*, 59:2175–2187, Feb 1999.

[20] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.8.1 edition, 2016.

[21] L. Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, 159(1):98–103, 1967.