Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

PROYECTO DE GRADO
INGENIERÍA EN COMPUTACIÓN

PARALELISMO APLICADO AL ESTUDIO DE MEDIOS GRANULARES

Abril 2009

LAURA HEREDIA
PABLO RICHERI

Supervisor: Sergio Nesmachnow
Co-Supervisor: Gonzalo Tancredi

PARALELISMO APLICADO AL ESTUDIO DE MEDIOS GRANULARES

RESUMEN

Este trabajo describe las técnicas de procesamiento de alta *performance* para implementar algoritmos DEM (Método de los elementos discretos) capaces de ejecutar en un ambiente paralelo-distribuido (se pondrá foco en la ejecución en cluster).

En el presente trabajo se evaluaron distintas herramientas, que implementan tanto la versión serial como paralela de algoritmos DEM. La versión serial introduce en los conceptos básicos empleados para la simulación del DEM, como las listas de Verlet. Se estudiaron dos versiones paralelas, y una de ellas fue extendida, con el propósito de modelar fenómenos basados en la física de medios granulares, para entender los procesos colisionales de asteroides.

Palabras clave: algoritmos DEM, paralelismo, partículas, listas de Verlet.

ÍNDICE

CAPÍTULO 1
Introducción

El Método de los Elementos Discretos (DEM) es una aplicación computacional de métodos numéricos, con el propósito de estudiar problemas de ingeniería que involucran un conjunto de partículas y la representación de la totalidad de las interacciones entre las mismas. Se construyen simulaciones para modelar diversos fenómenos físicos, con el objetivo de demostrar teorías en ese ámbito.

Cuando se comenzó a estudiar el Método de los Elementos Discretos, se identificó la existencia de varios productos que lo implementaban. Varias comunidades ya habían emprendido la construcción de algoritmia en este campo de la ingeniería. Además varias implementaciones ya consideraban en su diseño la aplicación de técnicas de paralelismo.

El presente trabajo tuvo como objetivo el estudio de herramientas disponibles que implementan algoritmos DEM. Se estudió tanto implementaciones paralelas como secuenciales. En la etapa siguiente a la evaluación de los distintos productos, se realizó una puesta en productivo de una de las herramientas, incluyendo instalación y configuración de la misma y de los productos con los cuales mantiene una relación de dependencia. La herramienta seleccionada es la descripta en el capítulo 5, la cual se denomina *ESyS-Particle*, fue elegida por ser la que reunía la mayor cantidad de características deseables, para la construcción de simulaciones con aplicación de la física de medios granulares. Esta aplicación ofrece una implementación paralela del motor de las simulaciones. Para que el tiempo de ejecución de las simulaciones sea razonable se requiere de hardware paralelo. Para ello, el centro de cómputos de la Facultad de Ciencias puso a disposición el hardware para computación de alto desempeño, denominado cluster Ciencias. El cluster consta de 8 nodos, donde cada nodo está conformado por 8 procesadores, lo que implica que se dispone de 64 unidades de procesamiento.

En una etapa posterior, se identificó la necesidad de extender la aplicación *ESyS-Particle*, para que fuera factible la representación de fenómenos vinculados con los procesos de colisiones de asteroides. La extensión consistió en la creación de un nuevo tipo de interacción entre partículas esféricas, denominada esferas viscoelásticas, considerando la acción de la fuerza de fricción. La versión de *ESyS-Particle* que se dejó funcionando en el cluster Ciencia incorpora la nueva interacción.

A continuación se detalla el contenido de cada capítulo del presente documento.

El capítulo 2 presenta los conceptos genéricos de la computación de alta *performance* y su aplicación para el diseño de algoritmos DEM paralelos y distribuidos, con el propósito de obtener un mejor desempeño computacional. Se finaliza el capítulo mencionando el estudio de los productos alternativos existentes que implementan algoritmia DEM.

La primera herramienta estudiada, la cual implementa una versión paralela de los algoritmos DEM, es descripta en el capítulo 3. El nombre de la aplicación es *Tremolo* y consta de dos componentes, el primer componente implementa la lógica de las simulaciones y el otro componente se encarga de la interfaz gráfica con la cual interactúa el usuario.

La versión serial de los algoritmos DEM, es explicada en el capítulo 4, y corresponde a una aplicación descripta en el libro de *Computational Granular Dynamics*. En esta parte se incluye tanto una implementación simple pero ineficiente del cálculo de la fuerzas de interacción de corto alcance como una implementación eficiente de las mismas.

En el capítulo 5 se presentan los detalles acerca de la aplicación *ESyS-Particle*, la cual implementa algoritmos DEM paralelos. Con esta aplicación se construyó un marco de trabajo, con el objetivo de que sea posible la construcción de simulaciones, para el estudio de fenómenos de la física de medios granulares. Como aporte en esta temática se extendió el código de la herramienta para soportar un nuevo tipo de interacción.

Por último, en el capítulo 6 se presentan las principales líneas de investigación para dar continuidad al trabajo realizado. También se concluye sobre los aspectos más relevantes de cada capítulo tratado.

CAPÍTULO 2
Procesamiento de alto desempeño aplicado a DEM

2.1 Introducción

DEM es un campo de la ingeniería cuyo propósito es la simulación de materias discretas. DEM obtiene el resultado global de la simulación a partir de los resultados obtenidos sobre cada partícula. Sin embargo, la aplicabilidad de las técnicas DEM se ve reducida por el gran tamaño o complejidad de los problemas a abordar, que requieren la utilización de varios miles de partículas y una gran cantidad de evaluaciones de funciones que calculan las fuerzas de interacción entre ellas.

El capítulo brinda una descripción de los conceptos de la computación de alto desempeño y un análisis de la aplicación de las distintas técnicas de la programación paralela con el objetivo de mejorar el rendimiento de algoritmos DEM. Se comienza con una introducción de los distintos paradigmas de la computación paralela, para el diseño de algoritmos capaces de sacar provecho de los múltiples recursos computacionales existentes. Luego se realiza un estudio de las técnicas de la programación paralela aplicadas al diseño de los algoritmos DEM, incluyendo un análisis de fortalezas y debilidades. Finalmente se menciona el relevamiento de aplicaciones DEM para realizar simulaciones de medios granulares.

2.2 Computación de alto desempeño

La computación de alto desempeño consiste en la aplicación de una familia de técnicas, que se utilizan para lograr ejecutar tareas concurrentes, con el fin de reducir el tiempo de procesamiento. Se debe contar con una arquitectura de hardware paralelo que permita el procesamiento simultáneo de las tareas. La cantidad requerida de circuitos para el procesamiento paralelo es mayor que en el procesamiento serial, pero en la actualidad esto no es un problema, ya que los costos de hardware son relativamente bajos.

Los modelos de programación paralela permiten diseñar la solución de problemas complejos, que requieren de un sistema con una capacidad de procesamiento superior. Estos sistemas son conocidos como multiprocesadores, y están conformados por múltiples unidades de procesamiento conectadas por una red de interconexión que posibilita la transferencia de datos y mensajes de control. La complejidad del problema puede estar dada por el volumen de datos a manejar o por la cantidad de cálculos a realizar, que sería imposible de implementar utilizando el modelo de programación secuencial clásico. En las siguientes secciones se hablará de las arquitecturas de hardware, los modelos de programación paralela que permiten el diseño, implementación y ejecución de aplicaciones paralelas y distribuidas, y por último se describirá como se puede evaluar el desempeño de los algoritmos paralelos. Los conceptos que serán explicados en las secciones siguientes se basan en las referencias [3, 4, 5, 6].

2.2.1 Arquitecturas paralelas

En cualquier sistema multiprocesador, los distintos procesadores que trabajan sobre diferentes partes del problema, se deben comunicar entre sí para intercambiar información. Existen dos tendencias que se diferencian en el mecanismo de comunicación: sistemas con memoria compartida y sistemas con memoria distribuida. A continuación, se presentará cada modelo.

En un sistema de memoria compartida existe una única memoria para almacenar datos, que es común a los procesadores involucrados (ver figura 2.1). Trabaja al igual que la maquina de Von Neumann, pero es una versión escalada respecto a la cantidad de procesadores. Este modela impacta también a nivel de procesos, debido a que varios de ellos pueden comunicarse realizando escrituras y lecturas coordinadas en la memoria común. Para ello, cualquier proceso puede leer y escribir en memoria tan solo ejecutando una única instrucción de máquina. Este modelo se ha hecho popular, debido a que la manera en que se comunican es simple y rápida. Además, es un modelo fácil de entender desde punto de vista programático, que se puede utilizar para resolver una amplia gama de problemas.

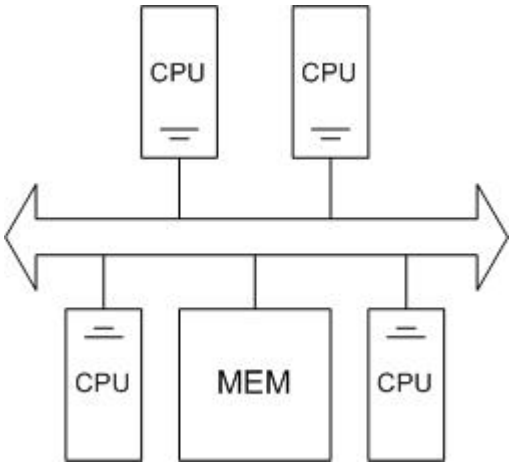

Figura 2.1: Sistema de memoria compartida.

La principal desventaja se presenta cuando ocurre un alto número de peticiones desde los distintos procesadores, llevando al bus a su límite de capacidad de transmisión, convirtiéndolo en el principal cuello de botella del sistema. Para solucionar este problema se introdujo el concepto de la memoria cache. El objetivo de la memoria cache es disminuir la cantidad de accesos a la memoria común, liberando así parte de la carga de trabajo del bus. A cada procesador se le asigna una memoria cache de uso exclusivo. El uso de caches locales trae consigo nuevos problemas a tener en cuenta en el diseño. Cuando un procesador escribe una palabra en memoria, este puede estar inutilizando porciones de cache de otros procesadores. Esto es debido a que las copias locales de los otros procesadores no conocen la actualización realizada. Para solucionar este problema el hardware debe emplear un mecanismo de coherencia de cache, que consiste en avisar al resto de los procesadores cuando hay una actualización.

El diseño de sistemas de memoria distribuida consiste en varios procesadores con su propia memoria interconectados entre sí (ver figura 2.2). Se diferencia respecto al diseño anterior, en que cada procesador posee una memoria local privada, a la cual accede a través de una única instrucción de lectura/escritura. La memoria de cada

procesador no puede ser accedida por el resto de los procesadores. Por lo tanto, cada procesador tiene su propio espacio de direccionamiento. Dado que ahora no es posible que los procesadores se comuniquen mediante una memoria global, es que surge la necesidad de un nuevo mecanismo de comunicación. Este mecanismo es implementado mediante el envío de mensajes, a través del uso de la red de intercomunicación. Al igual que en el otro diseño, este modelo también impacta a nivel de proceso. Ahora la comunicación entre procesos va a estar regida por la invocación de las primitivas de capa de aplicación *send* y *receive*. Notar que el contar con un bus de datos compartido es más eficiente desde el punto de vista de la rapidez, que los tiempos de comunicación debido al pasaje de mensajes. Como contrapartida en un sistema de memoria distribuida no existe el problema de la coherencia de cache.

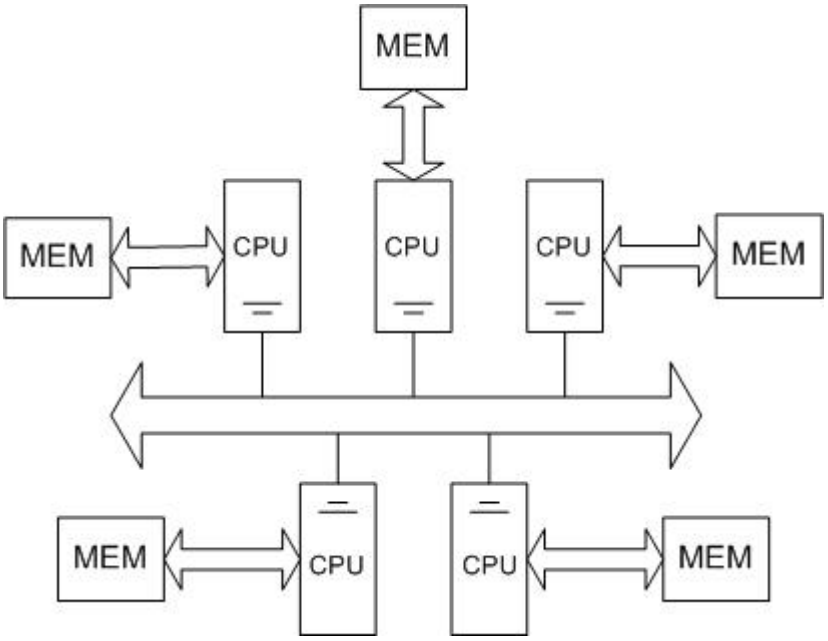

Figura 2.2: Sistema de memoria distribuida.

En sistemas de memoria distribuida, la ubicación óptima de los datos en las distintas memorias es un problema a resolver a nivel de aplicación, que deben tener en cuenta los programadores. Esto hace que el diseño de una aplicación sea más complejo que en sistemas de memoria compartida. A pesar de la complejidad a nivel programático, el construir sistemas de memoria distribuida es más sencillo y barato. Esta arquitectura paralela permite escalar en cientos y miles de procesadores de manera simple, pudiendo resolver así problemas que requieren un gran poder de cálculo.

Una clasificación especial dentro de los sistemas de memoria distribuida se denomina cluster. Se define el termino cluster como grupo de computadores completos interconectados entre sí, trabajando en forma conjunta como si fueran una unidad. El termino computador completo hace referencia a que cada computador del cluster (denominado nodo) puede funcionar de forma independiente. Una característica notable es que es posible armar un cluster de gran tamaño, superando así la potencia de los computadores independientes. Un cluster puede consistir de decenas de computadores, donde cada computador es un sistema multiprocesador de memoria compartida. Otras características son la escalabilidad incremental y una buena relación costo beneficio. La escalabilidad incremental se refiere a que es sencillo agregar nuevos nodos al cluster

una vez ya armado. La relación costo/beneficio indica que obtener un mayor poder de cálculo no implica un incremento abrupto en los costos.

2.2.2 Clasificación de arquitecturas paralelas

Desde 1972 la clasificación de los tipos de arquitecturas paralelas más utilizada es la taxonomía de Flynn. La clasificación se realiza dependiendo de si el flujo de datos y/o el flujo de instrucciones se procesa en paralelo. Se define como flujo de instrucciones al conjunto de instrucciones secuenciales que son ejecutadas por un único procesador, y como flujo de datos al flujo secuencial de datos requeridos por el flujo de instrucciones. La clasificación de Flynn se basa en la diferencia entre el desempeño de la unidad de control y el de la unidad de procesamiento de datos. Enfatiza en las características de desempeño del sistema de computadoras más que en las interconexiones estructurales y de operación. De esta manera, Flynn define cuatro modelos (ver figura 2.3): SISD (una única instrucción y un único flujo de datos – el clásico computador con un solo procesador), SIMD (una única instrucción y un flujo múltiple de datos), MISD (múltiples instrucciones y un único flujo de datos) y MIMD (múltiples instrucciones y flujo múltiple de datos).

		Instrucciones	
		Único flujo	**Flujo múltiple**
Datos	**Único flujo**	SISD	MISD
	Flujo múltiple	SIMD	MIMD

Figura 2.3: Taxonomía de Flynn.

SISD se basa en el diseño convencional de Von Neumann, el mismo tiene un único flujo de instrucciones y datos. Representa la organización de una computadora que contiene una unidad de control, una unidad de procesador y una unidad de memoria. Un único procesador ejecuta una única secuencia de instrucciones que trabaja con una única memoria donde los datos son almacenados.

El modelo SIMD es comúnmente utilizado en la resolución de problemas científicos que involucran tanto estructuras de datos vectoriales como un alto poder de cálculo. Representa una organización que consta de varias unidades de procesamiento bajo la supervisión de una única unidad de control. Todos los procesadores reciben la misma instrucción desde la unidad de control, pero operan sobre diferentes conjuntos de datos. Como ejemplo de este tipo de sistema se encuentran las computadoras vectoriales con hardware escalar y vectorial.

En la clasificación MISD, un único flujo de datos es utilizado por varios flujos de instrucciones al mismo tiempo. Esta organización es solo de interés teórico, dado que hasta el momento no existe un computador que implemente este modelo.

Por último, MIMD es la clasificación en la cual varios procesadores independientes trabajan juntos con diferentes conjuntos de datos. Esta organización se refiere a un sistema de computadoras capaz de procesar múltiples programas al mismo tiempo, la mayoría de los sistemas de memoria compartida y distribuida pueden clasificarse en esta categoría.

2.2.3 Modelos de programación paralela

El diseño de aplicaciones paralelas consiste en encontrar oportunidades de ejecución paralela. Una buena práctica es dividir en particiones más pequeñas tanto los cálculos asociados con el problema, como los datos con los que se operan. Existen dos técnicas de programación paralela: descomposición por dominio y descomposición funcional. La descomposición por dominio, se centra en encontrar las particiones apropiadas para los datos del problema. El enfoque de la descomposición funcional consiste en descomponer de manera apropiada los cálculos a ser realizados. Estas son técnicas complementarias que pueden ser aplicadas a un problema o parte de el, para obtener diferentes algoritmos paralelos. Se busca evitar la duplicación de cálculos y datos, es decir, definir tareas que dividen tanto los cálculos como datos en conjuntos disjuntos. Sin embargo, puede valer la pena el replicar tanto cálculos como datos si esto reduce los requerimientos de comunicación.

La técnica de descomposición por dominio divide los datos del problema en conjuntos más pequeños, preferentemente de igual tamaño. Los procesadores realizan los mismos cálculos sobre diferentes conjuntos de datos. Los aspectos fundamentales a tener en cuenta son, la comunicación entre procesos que manejan datos pertenecientes a regiones vecinas y el criterio adoptado para la división de datos.

La descomposición funcional inicialmente se centra en los cálculos a ser realizados, en lugar de los datos manipulados por estos. Si se tiene éxito en la división de los cálculos en procesos disjuntos, se procede a examinar los datos requeridos por estos. Si los conjuntos de datos requeridos son disjuntos, se puede decir que la partición esta terminada. Sin embargo, estos conjuntos se pueden solapar, en cuyo caso es necesaria una considerable cantidad de comunicaciones para evitar la replicación de datos. Si esto sucede se debe considerar un enfoque de descomposición por dominio.

Por otro lado, se tiene que optar por un modelo de comunicación entre procesos, cuya función es determinar como se comunican y/o sincronizan los procesos involucrados en el diseño de la solución. Existen tres posibles paradigmas: modelo maestro-esclavo, modelo cliente-servidor y el modelo *peer to peer*. Se describirá con mayor detalle el modelo maestro-esclavo porque es el utilizado en el presente trabajo.

El modelo maestro-esclavo es uno de los paradigmas de comunicación más simple. En este modelo se generan un conjunto de subproblemas y procesos que los resuelven. Existe un proceso distinguido denominado maestro y varios procesos idénticos denominados esclavos. El proceso maestro es quien controla a los procesos esclavos, y estos últimos son los que procesan la información. El proceso maestro es quien crea los procesos esclavos y les envía datos. La única comunicación desde los esclavos hacía el maestro es con el propósito de entregar los resultados de la tarea asignada. En la mayoría de los casos hay poca o nula comunicación entre los esclavos. La distribución de tareas a los esclavos es fundamental para el desempeño de aplicaciones que utilicen este paradigma.

Por otro lado, se tiene el modelo cliente-servidor, el cual puede ser utilizado para comunicar procesos que ejecutan en el mismo equipo, pero en realidad fue concebido para comunicar procesos distribuidos en la red.

El modelo *peer to peer* es un paradigma de comunicación donde no existe una jerarquía. Todos los procesos tienen las mismas capacidades para establecer comunicaciones con otros procesos.

Para implementar el modelo de comunicación, se dispone de bibliotecas que ofrecen un mayor nivel de abstracción para resolver la comunicación, a partir del pasaje de mensajes. Los procesos se comunican entre si a partir del envío y recepción de mensajes. MPI (*Message Passing Interface*) es un ejemplo de este tipo de bibliotecas. En la mayoría de las implementaciones de MPI, una cantidad fija de procesos es creada en la inicialización del programa, y solo un proceso es creado por procesador. Los procesos pueden utilizar operaciones para la comunicación punto a punto, para enviar un mensaje desde un proceso dado a otro. También, un grupo de procesos pueden invocar operaciones de comunicación colectivas para realizar acciones globales como por ejemplo un *broadcast*. Todas estas operaciones se pueden encontrar en la versión bloqueante, no bloqueante, sincrónica y asíncrona. Las operaciones para recibir mensajes tienen dos variantes, bloqueante y no bloqueante. Una descripción más detallada sobre MPI puede ser encontrada en [7, 8]. Desde la creación del estándar MPI se han implementado varias versiones por distintas organizaciones, ejemplos de ello son MPICH, LAMP, OpenMPI, etc.

2.2.4 Desempeño

El objetivo de la programación paralela no es diseñar procesos buscando optimizar una única medida tal como el tiempo de ejecución. Un buen diseño debe contemplar varios aspectos a optimizar tales como los requerimientos de memoria, tiempo de ejecución, costos de implementación, mantenimiento, requerimientos y costos de hardware, etc. El diseñador de algoritmos paralelos también, debe tener en cuenta factores como la simplicidad, eficiencia y portabilidad entre otros.

En está sección se describen los aspectos principales de los modelos de evaluación del desempeño de una aplicación paralela y distribuida, tomando como referencia el capítulo 3 del libro de Ian Foster [3].

2.2.4.1 Modelo de evaluación del desempeño

Los modelos de evaluación del desempeño son utilizados para comparar la eficiencia de diferentes algoritmos, evaluar escalabilidad para identificar cuellos de botella y otras ineficiencias. Los modelos descriptos aquí, consideran a cada métrica, como lo es el tiempo de ejecución, como función del tamaño del problema N, procesadores disponibles P, número de procesos U, entre otras características que dependen del algoritmo y del hardware sobre el que se ejecuta. En la ecuación 2.1 se presenta a modo de ejemplo el cálculo de la métrica tiempo de ejecución.

$$T = f(N, P, U, ...)$$

Ecuación 2.1: Cálculo del tiempo ejecución.

Se define tiempo de ejecución de un programa paralelo, como el tiempo entre el inicio de ejecución del primer proceso hasta el fin de ejecución del último proceso. Durante la ejecución, cada proceso alterna entre los siguientes estados: procesamiento efectivo, comunicación y ocioso. Por lo tanto, el tiempo de ejecución puede ser calculado como se muestra en la ecuación 2.2.

$$T = T_{PROC} + T_{COM} + T_{ocioso}$$

Ecuación 2.2: Componentes del tiempo de ejecución.

Si se tiene *p* tareas ejecutando en *p* procesadores, el tiempo total de ejecución está dado por la ecuación 2.3, que se muestra a continuación.

$$T = \frac{1}{P}(\sum_{i=0}^{P-1} T_{proc}^{i} + \sum_{i=0}^{P-1} T_{com}^{i} + \sum_{i=0}^{P-1} T_{ocioso}^{i})$$

Ecuación 2.3: Cálculo del tiempo total de ejecución.

El tiempo de cálculo normalmente depende de la complejidad y dimensión del problema. Si el algoritmo paralelo replica cálculos, entonces el tiempo de cálculo también va a depender del número de procesos requeridos. En entornos distribuidos, la heterogeneidad de los equipos hace que el tiempo de cálculo pueda variar de acuerdo a los procesadores disponibles. El tiempo de cálculo también va a depender de las características tanto de los procesadores como de las memorias.

El tiempo de comunicación depende de la ubicación de los procesos y datos. Se pueden distinguir dos tipos de comunicación: comunicación intra e inter procesador. El primer tipo se refiere a la comunicación entre procesos que corren sobre el mismo hardware de procesamiento, mientras que el otro tipo se refiere a procesos que corren sobre distinto hardware de procesamiento. El costo de la comunicación interprocesador es el tiempo necesario para el establecimiento de la conexión (latencia), y el tiempo necesario para la transferencia de información, que es determinado por el ancho de banda del canal de comunicación. En la ecuación 2.4 se muestra el tiempo requerido para enviar un mensaje de *L* bits, considerando que T_{TR} es el tiempo necesario para transferir un bit.

$$T = latencia + T_{TR}L$$

Ecuación 2.4: Tiempo requerido para el envío de un mensaje.

El tiempo que un proceso permanece ocioso se debe al no determinismo de la ejecución de los procesos involucrados en la resolución del problema, ya que el orden de las instrucciones ejecutadas varía en cada corrida. El objetivo del diseñador de algoritmos paralelos debe ser minimizar el tiempo ocioso de los procesos. Los procesos pueden estar ociosos debido a la ausencia de recursos de cálculo disponibles, o por ausencia de datos sobre los cuales operar. Para solucionar estos problemas, se puede utilizar técnicas de balanceo de carga para distribuir los cálculos a realizar entre los procesadores disponibles, o rediseñar la aplicación para distribuir los datos de una forma más conveniente.

2.2.4.2 Eficiencia y speedup

El tiempo de ejecución varía dependiendo del tamaño del problema y de los recursos disponibles. Por lo tanto, por sí solo no siempre es una métrica adecuada para evaluar el desempeño de los algoritmos paralelos. Por esta razón normalmente se utilizan dos métricas relativas, para la evaluación del desempeño de los algoritmos paralelos, que utilizan como dato de entrada el tiempo de ejecución: el *speedup* y la eficiencia.

El término *speedup* se define como una medida de la mejora de rendimiento obtenida, a partir de una aplicación paralela ejecutando sobre una cantidad de procesadores, respecto a la mejor aplicación serial existente (respecto al tiempo de ejecución), que resuelve el mismo problema. La fórmula con la que se calcula la métrica correspondiente al *speedup* absoluto se muestra en la ecuación 2.5. T_0 denota el tiempo de ejecución del mejor algoritmo serial que se conoce, y T_N es el tiempo de ejecución del algoritmo paralelo utilizando N procesadores.

$$S_N = \frac{T_0}{T_N}$$

Ecuación 2.5: Fórmula para el cálculo del speedup absoluto.

La eficiencia es la fracción de tiempo en que los procesadores se encuentran haciendo algo útil. En ocasiones se utiliza como una métrica para evaluar la calidad de los algoritmos paralelos. Refleja la efectividad con la que un algoritmo paralelo utiliza los recursos disponibles sin importar el tamaño del problema. La ecuación 2.6 define la eficiencia relativa, teniendo en cuenta que T_1 es el tiempo de ejecución utilizando un único procesador, y T_N es el tiempo utilizando N procesadores.

$$E_{relativa} = \frac{T_1}{NT_N}$$

Ecuación 2.6: Fórmula para el cálculo de la eficiencia relativa.

La ecuación 2.7 presenta la relación que vincula a la eficiencia con el *speedup*.

$$S_{relativo} = PE_{relativa}$$

Ecuación 2.7: Relación entre eficiencia y speedup.

Muchas veces no se conoce el mejor algoritmo serial que resuelva un problema dado, por lo que se utilizan como definición alternativa de *speedup* y eficiencia las ecuaciones 2.6 y 2.7 respectivamente. Estos valores son útiles para estudiar la escalabilidad de un algoritmo paralelo.

En la teoría, si se duplica la cantidad de procesadores, el tiempo de ejecución debería reducirse a la mitad. Sin embargo, debido a varios factores como demoras debido a las comunicaciones, *overhead* por intercambio de datos, cuellos de botella en el acceso a los recursos, entre otros, hacen que la mejora introducida no resulte en un crecimiento lineal del *speedup*. Otro factor por el cual no se logra una aceleración perfecta se desprende a partir de la ley de Amdahl. Amdahl observó que todo programa

consta de fracciones de código serial y paralelizables. De esta forma concluye que la aceleración tendrá como cota inferior el tiempo de ejecución de la fracción serial.

Una característica deseable en algoritmos paralelos es la escalabilidad. La escalabilidad representa una forma simple de mejorar el desempeño a partir de agregar más procesadores a la plataforma paralela. Sin embargo, estas incorporaciones no se pueden realizar de forma indiscriminada, para evitar la creación de cuellos de botellas.

2.3 DEM

Los medios granulares están formados por un cierto número de objetos macroscópicos (llamados granos) que interaccionan por medio de contactos temporales o permanentes. Todos los materiales que se presentan en forma de granulados (cereales, arena...) o polvos (talco, harina...) son estudiados por la física de medios granulares.

Los procesos que involucran medios granulares se han estudiado experimentalmente, mediante el desarrollo de experiencias de laboratorio que reproducen los fenómenos, y en las últimas décadas en forma numérica. El método de los elementos discretos (*Discrete Element Method* - DEM) simula el comportamiento mecánico de un medio formado por un conjunto de partículas las cuales interaccionan entre sí a través de sus puntos de contacto. La disposición de las partículas dentro del conjunto global del sistema o medio es aleatoria, por lo que se pueden formar medios con diferentes tamaños de partículas distribuidos a lo largo del conjunto, idealizando de este modo la naturaleza granular de los medios que usualmente se analiza y se simula mediante esta técnica numérica. El DEM se basa en:

- modelar las partículas como elementos discretos que conforman el sistema complejo.
- modelar el desplazamiento (independientemente) y la interacción entre partículas.
- utilizar la mecánica del cuerpo rígido para modelar cada partícula (los elementos discretos se consideran elementos rígidos en sí).

Habitualmente, las partículas pueden tener cualquier forma, son elásticas y no sufren de deformación permanente. Mientras que las fuerzas de contacto, consisten de un componente elástico lineal normal y otro componente elástico lineal tangencial.

Dos casos interesantes dentro de las simulaciones de medios granulares son los procesos de segregación de rocas por tamaño como producto de sismos producidos en colisiones de asteroides, y la producción de nubes de polvo a baja velocidad relativa como producto de la aceleración inducida por un sismo generado a partir de una colisión. Para el estudio de estos fenómenos se necesita de la aplicación de técnicas DEM en condiciones de muy baja gravedad, y la evolución orbital de partículas eyectadas desde la superficie a baja velocidades relativas. Ambos ítems requieren de simulaciones numéricas, y en el caso particular de la evolución orbital ya existen experiencias de aplicación de técnicas de simulación.

Algoritmos y programas secuenciales para simulaciones DEM de sistemas de tamaño medio, compuestos de hasta unos pocos miles de partículas, pueden ser ejecutados en computadores personales. La potencia de cálculo requerida para simulaciones de sistemas más grandes, supera ampliamente las capacidades de un computador personal si uno espera obtener resultados en un tiempo razonable. Para estos casos es recomendable la aplicación de técnicas de procesamiento paralelo y distribuido con el objetivo de obtener mejores tiempos de ejecución.

2.3.1 Reseña histórica

Las simulaciones mediante DEM han sido empleadas y validadas en diversas áreas, como: geomecánica (rocas, tierras), agricultura (transporte y manejo de grano y semillas), farmacia (manejo de polvo y píldoras), química (fluidización, ruptura de partículas), ingeniería de procesos (*peening*, efecto de virutas, aglomeración), petróleo y gas (bloqueo de conductos por la arena), medioambiente y biología (purificación de agua, sangre), entre otras.

A lo largo de la historia de DEM los hitos a destacar hasta el momento son:

- 1960´s: aparecen los primeros modelos continuos.
- 1971: primera aparición de DEM para el estudio de problemas mecánicos en rocas (pocas partículas muy grandes).
- 1979: se generaliza la aplicación para partículas más pequeñas.
- 1992: se materializan códigos de DEM que permiten:
  - Desplazamientos finitos y rotaciones de los elementos discretos.
  - Reconocer nuevos contactos a medida que el cálculo avanza.

En los años 90 y en la siguiente década, las publicaciones y referencias al DEM han aumentado exponencialmente.

2.4 Técnicas de paralelismo aplicadas a algoritmos DEM

La idea detrás de la paralelización es partir un problema matemático en partes más pequeñas que puedan ser procesadas en paralelo, es decir, al mismo tiempo e independientemente por cada computador. El método de descomposición del problema juega un papel decisivo en la eficiencia del programa paralelo. Una de las principales dificultades es mantener balanceada la carga entre los procesadores involucrados en el cálculo (hay que evitar tener procesadores ociosos) y minimizar la cantidad de comunicaciones necesarias entre los procesadores participantes.

Frecuentemente, se clasifican a las computadoras paralelas en función a su flujo de datos e instrucciones. De acuerdo a este esquema, la computadora serial clásica de Von-Neumann pertenece a la clase *single instruction single data* (SISD). Mientras que los algoritmos aquí presentados pertenecen a computadoras de la clase *multiple instruction multiple data* (MIMD). Tales computadores consisten de número de procesadores que son equipados con una memoria local y son capaces de intercambiar información entre ellos. Todos los procesadores ejecutan el mismo programa, sin embargo, controlados y coordinados por un procesador individual.

Comenzando con un programa serial se van a presentar dos estrategias de paralelización de la técnica DEM aplicada a materiales granulares. En la figura 2.4 se presenta un bosquejo del algoritmo serial. Las distintas estrategias de paralelización pueden ser distinguidas principalmente por el tipo de descomposición de datos que realizan. Sin embargo, todas las técnicas presentadas se basan en el pasaje de mensajes para la comunicación de los procesos.

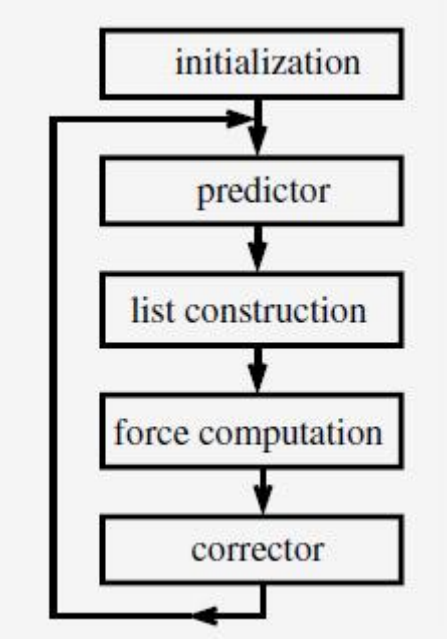

Figura 2.4: Bosquejo del algoritmo serial. *Tomada de [1].*

2.4.1 Modelo de descomposición funcional aplicado a DEM

La idea del modelo de descomposición funcional es guardar toda la información sobre todas las partículas en la memoria individual de cada procesador. Luego, cada procesador puede calcular las trayectorias de un subconjunto arbitrario de partículas.

Considerando la figura 2.4, si a cada procesador se le asigna un mismo número de partículas, se puede obtener un balance de carga óptimo para el predictor y corrector, los cálculos de las fuerzas externas y el movimiento de las partículas debido a las condiciones de borde. Dado que estos cálculos son independientes para cada partícula, la distribución de partículas es arbitraria.

Este principio no se aplica en los cálculos de fuerzas partícula-partícula. Aquí cada procesador no es responsable de un cierto número de partículas, sino en cambio, las entradas de las listas de interacción (por ejemplo, listas de Verlet que se verán en el capítulo 4) son distribuidas de tal manera, que cada procesador sea responsable por el mismo número de contactos entre partículas.

Los cálculos de las listas de interacciones también pueden ser paralelizados: cada procesador genera la lista para el mismo número de partículas en una distribución arbitraria.

Por lo tanto, excepto para la inicialización y la salida de datos, todas las partes del programa pueden ser paralelizadas. Como se muestra en la figura 2.5, la técnica de descomposición funcional requiere de mucha comunicación entre los procesadores. La comunicación es necesaria tres veces en cada paso:

1. Luego de predecir las posiciones, velocidades y derivadas de mayor orden en función del tiempo. Para partículas esféricas, las coordenadas, velocidades lineales y angulares de las partículas son difundidas hacia todos los procesadores. Para tipos de partículas más complejos, también se debe enviar datos adicionales (todas las características de las partículas que estén sujetas a la ecuación del movimiento de Newton).
2. Cuando se calcula una nueva lista de interacción, se debe transmitir a todos los demás procesadores.
3. Las fuerzas de interacción de las partículas tienen que ser obtenidas para calcular las fuerzas que actúan sobre cada partícula individual.

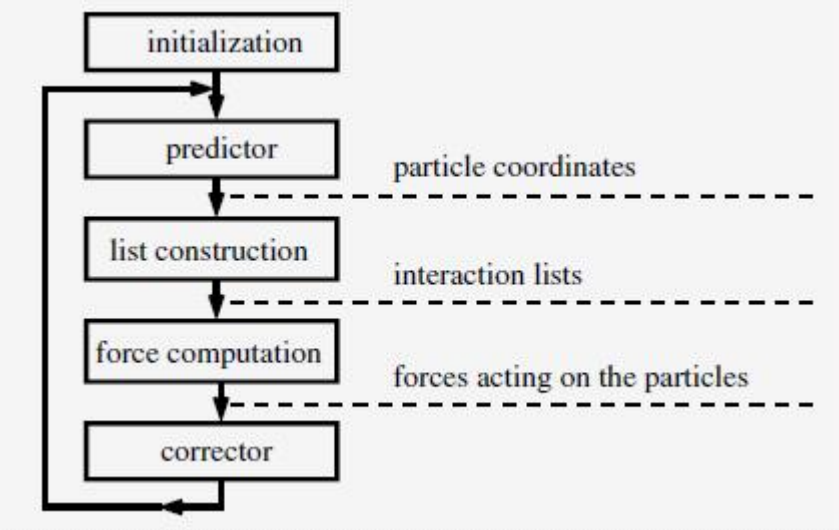

Figura 2.5: Bosquejo del programa paralelo. *Tomada de [1].*

Los altos requerimientos de comunicación que este método impone, restringe la aplicación de algoritmos a computadoras con un canal de comunicación altamente optimizado entre los procesadores.

Los esfuerzos de comunicación se incrementan rápidamente con el aumento del número de procesadores, por lo tanto, el programa se vuelve ineficiente. Entonces, el modelo de descomposición funcional no es adecuado para computadores con muchos procesadores usando canales poco eficientes de comunicación, tales como la mayoría de los clusters de PC.

2.4.2 Modelo de descomposición por dominio aplicado a DEM

El modelo de descomposición por dominio requiere de una cantidad significantemente menor de comunicaciones; sin embargo, su implementación y balanceo de carga es mucho más complicado.

Para este método de paralelización el área de simulación espacial es subdivido en sub-áreas (regiones) que son asignadas a los procesadores. Por lo tanto, las trayectorias de las partículas en una región son calculadas por el procesador correspondiente. Cada procesador mantiene en su memoria los radios, posiciones, velocidades y otros datos de las partículas que pertenecen a su región. En cuanto una partícula abandona su región actual, los datos de esa partícula son transferidos al procesador que es responsable por la región hacia la cual la partícula se movió. Para el cálculo de las fuerzas que actúan sobre las partículas que están ubicadas cerca de límite de una región, se necesitan los datos de las partículas cercanas en las regiones adyacentes. Por lo tanto, además de los datos de todas las partículas en la región, los procesadores también necesitan los datos de las partículas en los alrededores de la región, cuyo tamaño depende del tamaño de la partícula (ver figura 2.6, en gris se muestra la porción extra de partículas necesaria para el procesador asignado a esa región). Así, se puede considerar que las regiones se solapan mutuamente. En cada paso de tiempo la información de las partículas que están ubicadas en la región de solapamiento es intercambiada entre los procesadores. Por lo tanto, cada procesador tiene la información necesaria para calcular las trayectorias de las partículas que están ubicadas en su región, independientemente de los otros procesadores.

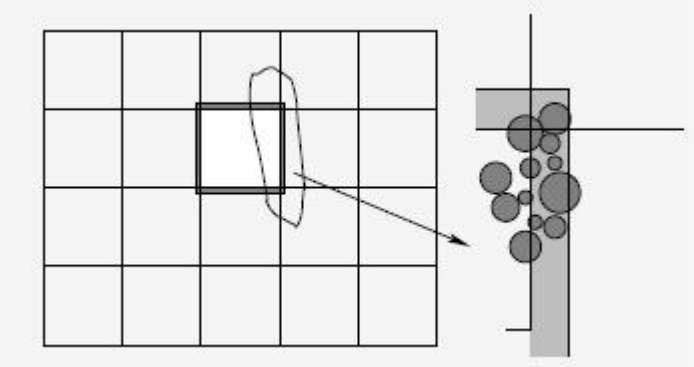

Figura 2.6: Descomposición del área de simulación en regiones. *Tomada de [1].*

Los datos de las partículas en las secciones de solapamiento son intercambiados luego del paso predictor. De esta forma, para cada partícula que cambia de región, los datos son enviados al procesador correspondiente y la administración de la partícula es tomada por este procesador. Cada partícula que ingresa a la región de un procesador es agregada a la lista de Verlet (este concepto será explicado en el capítulo 4) de las partículas alojadas ahí. En cuanto la fase de comunicación es completada, el siguiente paso de la iteración puede ser realizado. Un bosquejo del algoritmo se presenta en la figura 2.7.

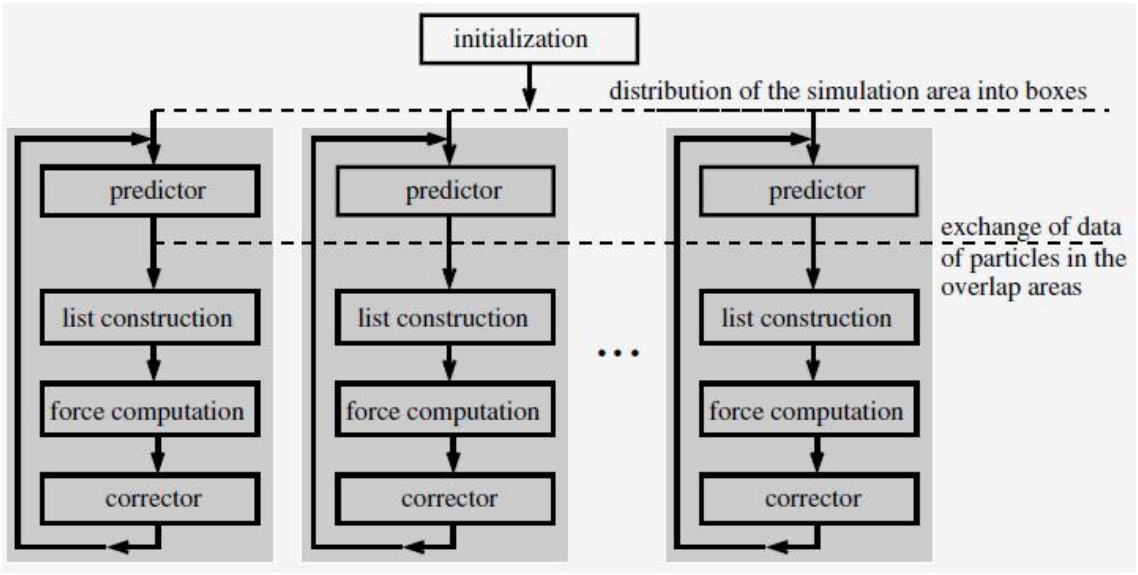

Figura 2.7: Bosquejo del método de descomposición por dominio. *Tomada de [1].*

La eficiencia del algoritmo paralelo depende en gran manera de cómo se realiza la descomposición espacial del área de simulación. Si las regiones son muy pequeñas, el número de partículas que cambian de región incrementa y por lo tanto, la carga de comunicación también, lo cual disminuye la *performance*. En cambio, cuanto más grandes son las regiones, más serial se vuelve el algoritmo y se pierde el paralelismo deseado. El valor óptimo de sub-áreas en las que se puede descomponer el dominio depende de la cantidad de procesadores disponibles y del problema a resolver.

Para mantener la carga de comunicaciones baja es útil buscar flujos regulares de partículas. En el caso de que las partículas se agrupen contra un sector del dominio de simulación, es útil considerar regiones dinámicas. Con las regiones dinámicas el tamaño y forma de las regiones se puede ir adaptando, para asegurar un correcto balanceo de carga entre los procesadores. Sino, unos pocos procesadores pueden estar calculando la mayoría de las trayectorias, mientras que los demás procesadores están ociosos. En la figura 2.8 se muestra el balanceo de carga automático para el caso de un proceso de sedimentación.

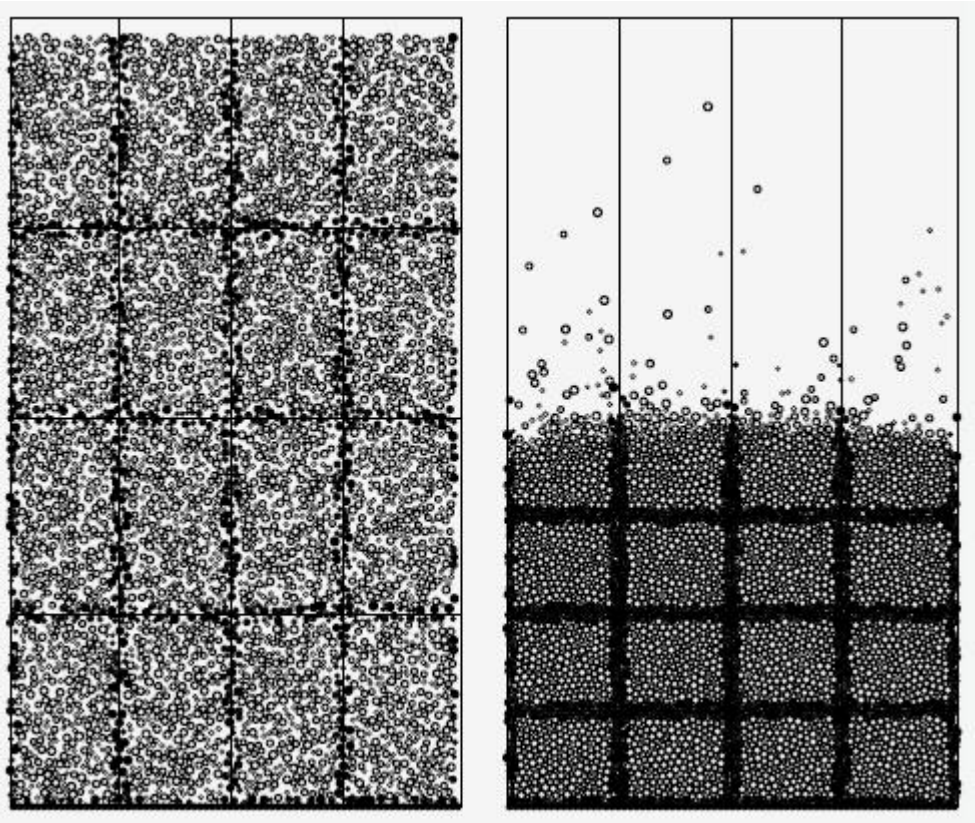

Figura 2.8: Balanceo de carga dinámico. *Tomada de [1].*

La redistribución del área de simulación necesaria, para alcanzar un balance de carga óptimo en sistemas con partículas distribuidas de manera no homogénea, siempre viene acompañada de serias complicaciones algorítmicas. Existe un truco simple para evitar una redistribución del área de simulación, llamado dispersión de datos (*data scattering*), en el cual el área de simulación es subdivida en regiones más pequeñas de manera que a cada procesador se le asignan *n* regiones en lugar de una. Estas *n* celdas o regiones, deben estar distribuidas tal que, la baja cantidad de partículas en algunas regiones afecten por igual a todos los procesadores. La figura 2.9 muestra esta subdivisión del área de simulación asumiendo que se cuenta con cuatro procesadores.

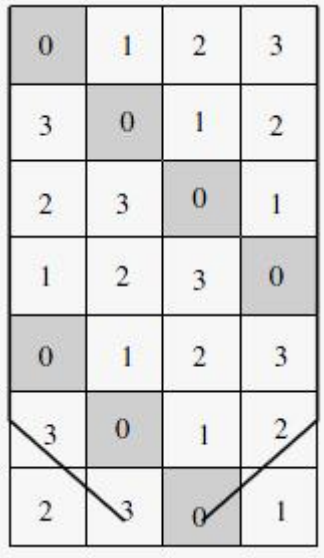

Figura 2.9: Aplicación del método data scattering. *Tomada de [1].*

El espacio es subdivido en 4x7 celdas iguales. Inicialmente las regiones son igualmente pobladas por partículas. Con el paso del tiempo las regiones superiores son vaciadas, esto sin embargo, no afecta la carga total de trabajo de los procesadores. La desventaja de este método es que hay incremento de las zonas de solapamiento lo cual implica esfuerzos extras de comunicación, disminuyendo así la eficiencia. Por lo tanto, el método de dispersión de datos esta limitado a sistemas relativamente grandes, donde el número total de partículas dentro de cada región es grande comparado con el número de partículas en la zona de solapamiento.

2.5 Relevamiento de productos DEM

En el presente trabajo se realizará el estudio de diversas herramientas que implementan algoritmos DEM, con el propósito de ser utilizadas para simulaciones de fenómenos de medios granulares. Los procesos de segregación por tamaño debido a vibraciones (conocido como “efecto nueces de Brasil”) son un caso particular de fenómenos presentes en medios granulares. El objetivo del estudio de este tipo de fenómenos es para aplicarlo al estudio de procesos colisionales de asteroides, constituyendo una aplicación novedosa de esta técnica.

Los productos que implementan algoritmos DEM paralelos se describen en detalle en los capítulos 3 y 5. El modelo de comunicación utilizado por ambos productos, es el maestro-esclavo. Las bibliotecas que implementan la comunicación entre procesos son MPICH y OpenMPI respectivamente.

En el capítulo 4, se realizará el relevamiento de una herramienta que implementa la versión serial de la algoritmia DEM. Esta herramienta explica los principios básicos que se deben tener en cuenta en la programación de simulaciones de medios granulares a partir de DEM. Al no tener en cuenta el paralelismo, es más simple entender como se representan los distintos componentes de una simulación.

CAPÍTULO 3
Tremolo

3.1 Introducción

La simulación numérica de la evolución de materiales granulares ha sido simulado recientemente con el Método de los Elementos Discretos (DEM). La técnica de Dinámica Molecular (DM), es un caso particular de DEM, debido a que las partículas son representadas mediante esferas.

Tremolo es un software construido para la simulación a partir de técnicas DM basado en los fundamentos teóricos desarrollados en [2], que combina algoritmos para la evaluación de potenciales de corto y largo alcance. Estos algoritmos son implementados utilizando un modelo paralelo, sobre computadoras con memoria distribuida (SIMD) con MPI (*message passing interface*) para la comunicación de los procesos. Con esto logra una mejor distribución en la carga de trabajo consiguiendo así importantes mejoras en los tiempos de ejecución.

Tremolo posee una interfaz gráfica amigable llamada *TremoloGUI*, la cual ofrece distintas funcionalidades que serán descriptas en las siguientes secciones del capítulo. En la siguiente sección de este capítulo se explicarán los pasos a seguir para la correcta instalación del software, el cual consta de dos módulos instalables, el componente lógico y quien implementa la interfaz gráfica.

Esta herramienta ha sido desarrollada por el *Institute for Numerical Simulation*, Universidad de *Bonn*, Alemania. La versión de la herramienta que se describe en este capítulo corresponde a una versión *open source*, bajo la licencia *gnu*, que actualmente ya no se encuentra disponible. Existe otra versión posterior a la presentada, cuya licencia no esta disponible de forma gratuita, ni siquiera para fines de investigación.

3.2 Instalación

La instalación de este software tiene dependencias con otras herramientas, que se listan a continuación:

- MPICH - una implementación de MPI.
- FFTW - transformadas de Fourier discretizadas para realizar cálculos de fuerzas de largo alcance.
- QT - biblioteca multiplataforma para desarrollar interfaces gráficas de usuario.
- VMD - Visual Molecular Dynamics es un programa de visualización para dibujar, animar y analizar grandes sistemas de partículas utilizando gráficos 3D.

La ruta a los archivos binarios de cada producto debe ser agregada al *path* del usuario.

Para la instalación de *Tremolo* en un sistema operativo *CentOS* sobre un sistema multiprocesador de memoria compartida, se deben descomprimir los dos componentes, tanto el motor lógico como la interfaz grafica. Luego, dentro de la carpeta de cada componente del producto se deben correr los siguientes comandos:

1. ./configure
2. make
3. make install

En la instalación del componente gráfico, antes de ejecutar el paso 3, se debe crear en */path_instalacion/doc/* un archivo vacío de extensión *.html*. Esto es para corregir un error en el proceso de instalación del producto. A continuación, se debe agregar al *path* la ruta */usr/local/bin*. La invocación al programa se realiza desde la consola mediante el comando *tremologui*.

3.3 Algoritmo para potenciales de corto alcance

El método *linked cell* es utilizado para la evaluación de fuerzas y energías de corto alcance, que disminuyen rápidamente con la distancia. Este método es fácil de implementar y al mismo tiempo es eficiente. La idea del método es dividir el área de simulación en subáreas uniformes, llamadas celdas. Los lados de las celdas miden como mínimo el radio de *cutoff* del potencial, de esta forma las interacciones se limitan a la celda en cuestión y sus celdas adyacentes. Teniendo estructuras de datos adecuadas para las partículas de cada celda, las partículas dentro del radio de *cutoff* pueden ser accedidas de forma más rápida. Todo esto conlleva a un cálculo eficiente de la fuerza. La descomposición en celdas encaja bien con la descomposición del dominio de simulación en subdominios para la paralelización. El dominio es descompuesto de tal forma que los bordes de los subdominios coinciden con los bordes de las celdas de la descomposición *linked cell*.

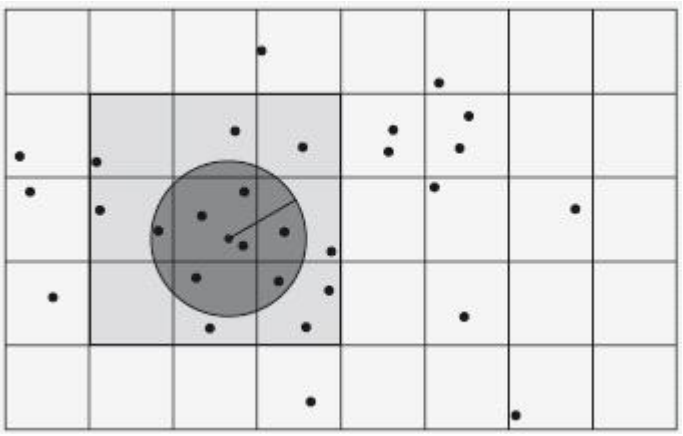

Figura 3.1: Bosquejo del método *linked cell.* *Tomada de [2].*

En la figura 3.1, el dominio de simulación es descompuesto en celdas cuadradas de tamaño r_{cut} x r_{cut}. El área oscura muestra la influencia en el dominio de una partícula para un radio de *cutoff* r_{cut}, que a su vez está contenida en un área de celdas de 3 x 3.

Cada procesador trabajará con las partículas que se encuentren dentro del subdominio asignado. El método de *linked cell* es ejecutado en cada procesador y calcula las fuerzas, energías, nuevas posiciones, y velocidades para las partículas dentro del subdominio. Pero para calcular las fuerzas sobre algunas de las partículas dentro del subdominio, el procesador necesita las posiciones de las partículas dentro de la distancia r_{cut} que pueden estar ubicadas en celdas de subdominios adyacentes. Para almacenar los datos de las partículas de los subdominios adyacentes, cada subdominio es extendido en una celda en cada dirección. Por lo tanto, cada procesador necesita datos de las partículas en celdas adyacentes a los bordes del subdominio, para poder calcular las fuerzas sobre las partículas en su propio subdominio. Los datos de las partículas de los subdominios adyacentes que son necesarios para el cálculo de la fuerza, deben ser intercambiados antes de realizar el cálculo. También puede suceder que las partículas abandonen el subdominio pasando a otro adyacente, estás partículas son asignadas a un procesador diferente.

Para intercambiar datos entre procesadores, cada procesador construye paquetes con la información de las partículas que se encuentran en la frontera del subdominio. Estos paquetes son enviados sobre la red a los procesadores apropiados. Estos procesadores reciben los paquetes, extraen los datos y los insertan en las estructuras adecuadas. Los datos son intercambiados en la menor cantidad posible de pasos de comunicación entre procesadores, dado que el establecimiento de conexión entre procesadores lleva un tiempo relativamente largo. Se envían solo los datos necesarios dado que el intercambio de datos consume un tiempo mayor que la ejecución de instrucciones.

La comunicación se realiza a partir del intercambio de mensajes explícitos entre programas ejecutándose en diferentes procesadores. Para ello, se desarrolló un único programa que corre sobre cada procesador de forma independiente, pero solo trabaja sobre los datos propios. Para implementar la sincronización y comunicación de procesos se utilizó la biblioteca *Message Passing Interface* (MPI).

3.4 Algoritmos para potenciales de largo alcance

Tremolo ofrece varios algoritmos para el cálculo de potenciales de largo alcance, entre ellos se encuentra el método de Barnes-Hut, el cual será explicado en esta sección. Para una explicación más detallada de los conceptos presentados en está sección dirigirse a [2, 9, 10].

En la figura 3.2 se puede apreciar la interfaz gráfica de la herramienta, donde se puede elegir que tipo de algoritmo se desea usar, o inclusive no usar ninguno. Cada algoritmo tiene su propio conjunto de parámetros necesarios, los cuales se van activando según la elección realizada.

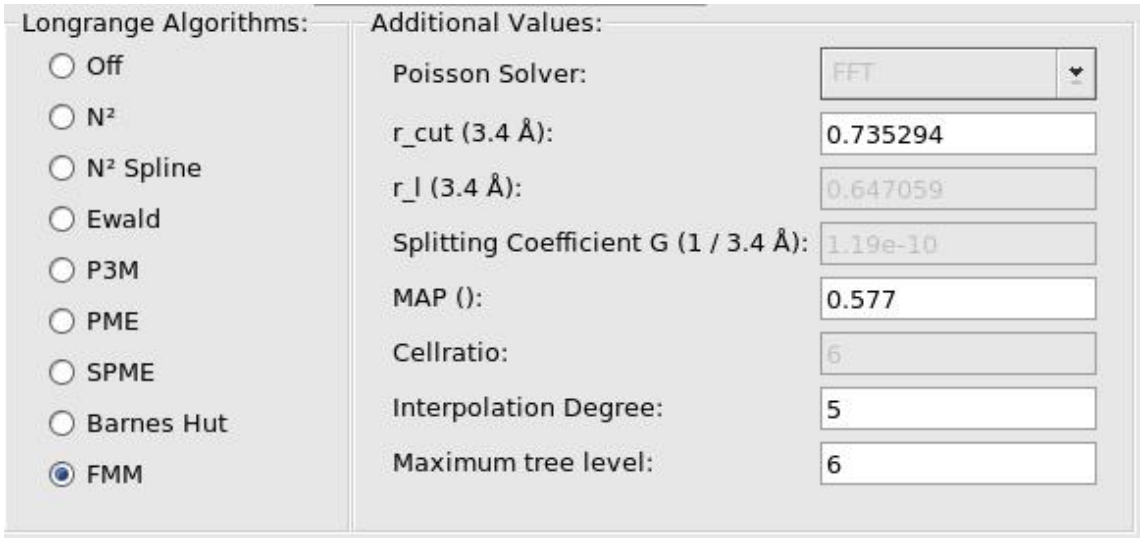

Figura 3.2: Algoritmos de largo alcance disponibles en Tremolo.

Los primeros dos algoritmos, se los denomina algoritmos partícula–partícula (PP), consisten simplemente en calcular en forma directa todas las interacciones entre las partículas. Si se toma en cuenta la interacción de cada par de partículas el cálculo directo requiere un total de $O(N^2)$ operaciones, dado que para N partículas hay $N(N-1)$ interacciones. Este tipo de cómputo es razonable para sistemas que involucran unos

pocos cientos de partículas, pero como el tiempo de cálculo se incrementa rápidamente con el número de partículas, las simulaciones numéricas con miles de partículas resultan computacionalmente costosas y aquellas con millones de partículas resultan imposibles de realizar. Entonces, para tales sistemas, se debe considerar un método alternativo al cálculo directo.

Los métodos Ewald, P3M, PME y SPME, son categorizados como algoritmos partícula–malla (PM), los cuales determinan potenciales como el gravitatorio, generado por la distribución de N partículas, sobre las celdas de una malla regular que cubre la región a considerar. El cálculo es realizado resolviendo numéricamente sobre la malla, la ecuación de Poisson donde Φ es el potencial (ver Ecuación 3.1).

$$-\Delta\Phi(x) = \frac{1}{\varepsilon_0}\rho(x) \quad en\ R^3$$

Ecuación 3.1: Ecuación de Poisson.

A partir de la discretización de la ecuación diferencial parcial se obtiene un sistema lineal de ecuaciones. *Tremolo* resuelve este sistema con el método directo de la transformada rápida de Fourier (FFT). La fuerza sobre una determinada partícula es calculada interpolando el potencial Φ sobre la malla en la posición de la partícula. Utilizando los procedimientos apropiados el número total de operaciones para el cálculo de las interacciones con éste método es de orden $O(N + M \log M)$ donde M es el número de celdas en la malla. Puesto que en la práctica, debido a limitaciones de memoria en las computadoras disponibles, $M << N$, el costo computacional del método resulta proporcional a N. De este modo, el cálculo de las interacciones se realiza con muchas menos operaciones que con el método PP.

Los métodos de malla son satisfactorios si las partículas se encuentran distribuidas en forma aproximadamente uniforme sobre una región rectangular del espacio, pero resultan imprecisos cuando la distribución de partículas no es uniforme. Existe una alternativa que efectúa el cálculo sin la utilización de mallas, y con menor esfuerzo que el método PP. Tal método es denominado algoritmo de árbol jerárquico octal. Este algoritmo explota el hecho de que el potencial gravitatorio decae inversamente proporcional a la distancia, con lo cual las partículas interactúan fuertemente con sus vecinos más cercanos, pero resulta necesaria menos información para describir sus interacciones con partículas más distantes. Así mientras la interacción de una partícula con otras partículas cercanas es evaluada en forma directa como una interacción partícula–partícula, la interacción con las partículas más distantes puede considerarse como una interacción directa entre la partícula y dicho conjunto de partículas distantes. El conjunto de partículas lejanas es representado como una partícula cuya masa es la total del grupo colocada en el centro de masa del mismo. Para llevar a cabo esta idea en forma eficiente, el algoritmo construye una estructura jerárquica de árbol a partir de una división particular del espacio. Esta estructura de árbol provee una manera sistemática de determinar el grado de cercanía geométrica entre dos partículas sin calcular explícitamente la distancia entre cada par de partículas. El resultado del método es reducir efectivamente el número total de operaciones en el cálculo de las interacciones a $O(N \log N)$. Los métodos de Barnes-Hut y FMM que se encuentran en la figura 3.2 son ejemplos de algoritmos de este tipo.

El método de árbol más simple para el cálculo de potenciales y fuerzas de largo alcance es el denominado Barnes-Hut. En sus comienzos fue desarrollado para problemas astrofísicos. Normalmente en este tipo de problemas existe un gran número partículas, con una distribución de densidad no homogénea. Las partículas de estos sistemas interactúan entre si mediante potenciales gravitatorios. El método de Barnes-Hut utiliza árboles de manera que el dominio de simulación es recursivamente dividido en subdominios de igual tamaño (celdas), hasta que cada celda contenga como máximo una partícula (ver figura 3.3). Luego, las celdas resultantes se corresponden con los nodos del árbol. Los nodos internos del árbol representan celdas con varias partículas. Para distribuciones no homogéneas de partículas, este enfoque da lugar a árboles no balanceados.

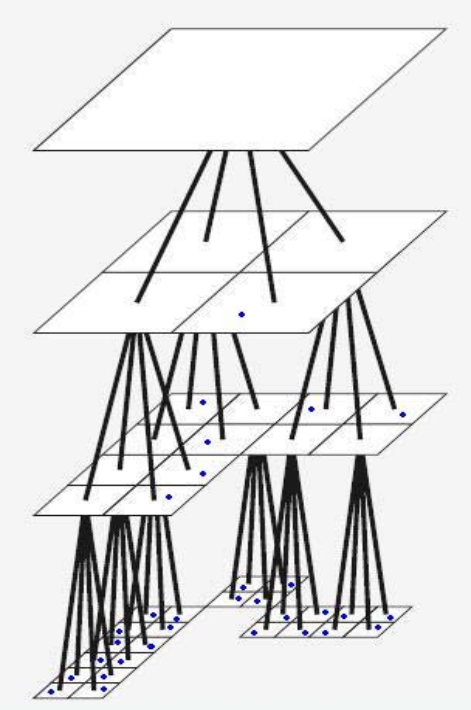

Figura 3.3: División de dominio con el algoritmo de árbol de Barnes-Hut.
Tomada de [2].

El método de Barnes-Hut se basa en la idea de que el efecto de la atracción gravitacional de varias partículas en una celda lejana, es esencialmente el mismo efecto que el de una partícula más grande en el centro de masa de la celda. Por lo tanto, todas las interacciones con estas partículas lejanas pueden ser modeladas a partir de una única interacción con una pseudo-partícula. La posición de la pseudo-partícula es el centro de masa de las partículas de la celda y la masa de la pseudo-partícula es la masa total de todas las partículas de la celda.

Este método básicamente consiste de tres pasos. La construcción del árbol, el cálculo de las pseudo-partículas y el cálculo de las fuerzas. Las pseudo-partículas representan celdas con más de una partícula, y por lo tanto se los asocia con nodos internos del árbol, mientras que las partículas reales son almacenadas en las hojas del árbol. Las coordenadas geométricas de una pseudo-partícula están dadas por el promedio de las coordenadas de todas las partículas en la celda, ponderadas por su masa. La masa de la pseudo-partícula es la masa total de todas las partículas de la celda.

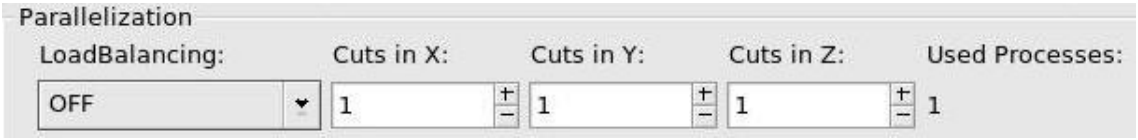

Figura 3.4: Interfaz de configuración de los parámetros para la ejecución paralela de los algoritmos de largo alcance.

Para los algoritmos PP y PM se puede elegir la cantidad de particiones a realizar en cada uno de los ejes de coordenadas (ver figura 3.4). Una vez confirmada la cantidad de subdivisiones, la herramienta notifica el total de procesos necesarios para la descomposición del dominio indicada. Por otro lado, en el caso de los algoritmos de árbol como el de Barnes-Hut y FMM, se indica a la herramienta el total de procesos con los que se desea trabajar, y es responsabilidad del algoritmo elegido realizar una apropiada división de dominio de simulación.

A continuación, se discutirá el enfoque teórico en el cual se basa la herramienta, para la implementación paralela de los métodos de árbol orientados a sistemas con memoria distribuida. Es de suma importancia para obtener un método eficiente una apropiada distribución de los datos entre los procesos. Los métodos de árbol son apropiados para distribuciones no homogéneas de partículas. Por lo tanto, una simple descomposición uniforme del dominio en general conduce a una pérdida de desempeño en la paralelización. Tal perdida de *performance* es el resultado del desbalance de carga. En la figura 3.5 se muestra un ejemplo de una distribución no uniforme de partículas en el dominio de simulación.

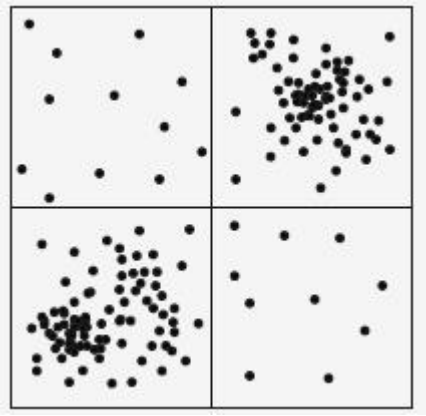

Figura 3.5: Una distribución no uniforme de partículas en el dominio de simulación y una partición en cuatro subdominios de igual tamaño. *Tomada de [2].*

En este caso se puede observar que una partición del dominio en cuatro subdominios iguales conduce a una asignación desbalanceada de las partículas en los procesos, y por lo tanto una distribución desbalanceada de los cálculos a realizar en los procesadores. Este desbalance de carga perjudica la eficiencia del método paralelo. Además, dado que las partículas se mueven de acuerdo a la ley de Newton en el tiempo, la asignación óptima de las partículas sobre los procesos también debe cambiar dinámicamente a lo largo del tiempo.

Para la distribución de los nodos del árbol en los distintos procesos, no se puede utilizar la descomposición por dominio en los nodos superiores y cercanos a la raíz. Primero se debe descender un cierto número de niveles en el árbol, para alcanzar un

punto en el cual el número de nodos o subárboles sea mayor o igual al número de procesos. En este punto, uno o varios subárboles (y sus correspondientes subdominios y partículas) pueden ser asignados a cada proceso. Aquí, el número de partículas por proceso debe ser aproximadamente el mismo para evitar el desbalance de carga. Todas las operaciones que pueden ser ejecutadas independientemente sobre los subárboles son ejecutadas por los procesos apropiados. De esta manera, tal distribución en subárboles puede ser interpretada como una descomposición por dominio adaptada a la densidad de partículas. Sin embargo, en el método de Barnes-Hut, los cálculos no solo son ejecutados por los nodos hoja que corresponden a partículas, sino también por nodos internos que corresponden a pseudo-partículas.

Para realizar el balance de carga se utiliza una estrategia de división de dominio, que saca provecho de la estructura del árbol construido. De esta forma se asignan subárboles completos a cada proceso de manera de minimizar la comunicación.

3.5 Parámetros de la simulación

DEM es una familia de métodos numéricos para calcular el movimiento de grandes números de partículas como moléculas o granos que se rigen por las leyes de la mecánica clásica, específicamente por la ley de Newton. *Tremolo* permite la construcción de simulaciones a partir de la aplicación de la técnica numérica denominada dinámica molecular (DM), la cual es un caso particular de DEM, cuando las partículas son moléculas esféricas.

En dinámica molecular, las diversas configuraciones del sistema se generan a partir de una configuración inicial mediante la integración de las ecuaciones del movimiento de Newton. El resultado es una trayectoria que especifica la posición y la velocidad de todas las partículas del sistema a lo largo del tiempo. Por tanto, en este método de simulación, una vez conocida la energía potencial de interacción intermolecular y las condiciones iniciales (normalmente NVE, número de partículas, volumen y energía constantes; aunque también son posibles NVT, a temperatura constante, y NPT, a presión constante), se calculan las fuerzas que actúan sobre cada partícula. La aplicación permite definir la configuración inicial del sistema de partículas mediante el ingreso de valores desde la interfaz gráfica, que corresponden a los valores iniciales de los parámetros de la simulación.

El método de simulación empleado por la aplicación requiere de la especificación de las condiciones de borde del sistema, el método de integración de las ecuaciones del movimiento, el ensamble termodinámico a utilizar, el potencial empleado para las interacciones y la esfera de corte (*cut-off*) del potencial utilizado en el método *Linked Cell* (visto en la sección 3.3). El requerimiento a especificar que depende del sistema a simular son las posiciones iniciales de las partículas. A continuación, se desarrolla cada una de estas especificaciones requeridas por la herramienta.

Las condiciones de borde del sistema definen el comportamiento molecular en las fronteras del dominio. Dependiendo del problema específico ciertas condiciones son impuestas sobre los bordes del dominio de simulación. En sistemas periódicos, como por ejemplo cristales, es natural imponer condiciones de periodicidad sobre los bordes.

Las condiciones periódicas también son utilizadas en problemas no periódicos para compensar el tamaño limitado del dominio de simulación. En este caso, el sistema es extendido de manera periódica a todo R^3. Las partículas que abandonan el dominio por un lado vuelven a entrar al mismo desde el lado opuesto. Además, las partículas cercanas ubicadas en lados opuestos del dominio interactúan entre si. Condiciones de borde de rebote aparecen cuando el dominio se comporta como una caja cerrada. Por lo tanto, una partícula que se acerca a una cierta distancia del borde, es sujeta a una fuerza repulsiva que la hace rebotar. También existen condiciones de salida que son usadas en bordes donde las partículas pueden abandonar el dominio de simulación. Mientras que las condiciones de entrada permiten que nuevas partículas se incorporen al dominio de simulación en ciertos momentos.

La energía potencial es una función de las posiciones de todas las partículas en el sistema. Dada la complicada naturaleza de está función, no hay solución analítica para las ecuaciones del movimiento; estas deben ser resueltas numéricamente. Los algoritmos numéricos disponibles en la aplicación para la integración de las ecuaciones del movimiento son Verlet, Beeman-Verlet y Velocity-Beerman-Verlet. La aplicación ofrece dos categorías de potenciales, los que actúan sobre sistemas de partículas enlazadas y no enlazadas. A continuación, se presenta en la figura 3.6 el listado de potenciales disponibles según la categoría.

Potenciales	
No enlazados	Lennard-Jones Lennard-Jones Spline Lennard-Jones Spline² Coulomb_ERFC Stillinger-Weber Stillinger-Weber 3 body Brenner
Enlazados	Bond Angles Torsions Improper Torsions

Figura 3.6: Listado de potenciales disponibles en Tremolo.

Para un sistema que se encuentra térmica y mecánicamente aislado, la energía total del mismo es constante en el tiempo. Sin embargo, en algunas simulaciones, la energía o la temperatura deben ser cambiadas a lo largo del tiempo. Por ejemplo, se puede necesitar la habilidad de controlar la temperatura del sistema para el estudio de fenómenos físicos o químicos tales como los cambios de estados. *Tremolo* ofrece varios métodos para ajustar y cambiar la temperatura de un sistema a lo largo del tiempo. Considerando N como el número partículas, V el volumen del dominio de simulación, T la temperatura y E la energía del sistema, tenemos los siguientes ensambles ofrecidos por la aplicación:

- NVE hace referencia a volumen y energía constantes.
- NVT indica volumen y temperatura constantes.
- NPT establece presión y temperatura constantes.
- NPE significa presión y energía constantes.

Para la visualización 3D de la salida de la simulación se puede utilizar varias herramientas como VMD, PyMOL y RasMol. Esto es posible porque *Tremolo* implementó una *interface* con cada una de estas herramientas de visualización. Desde la interfaz gráfica se puede seleccionar una de ellas, de la cual se recomienda la utilización de VMD, pues con PyMOL o RasMol solo se podrán ver *frames* individuales y no el video completo resultado de la simulación.

3.6 Ejemplos de aplicaciones

En esta sección se presentan imágenes de simulaciones realizadas con *Tremolo* en el campo de la bioquímica, biofísica, nanotecnología y ciencias de materiales, las cuales han sido obtenidas de [15].

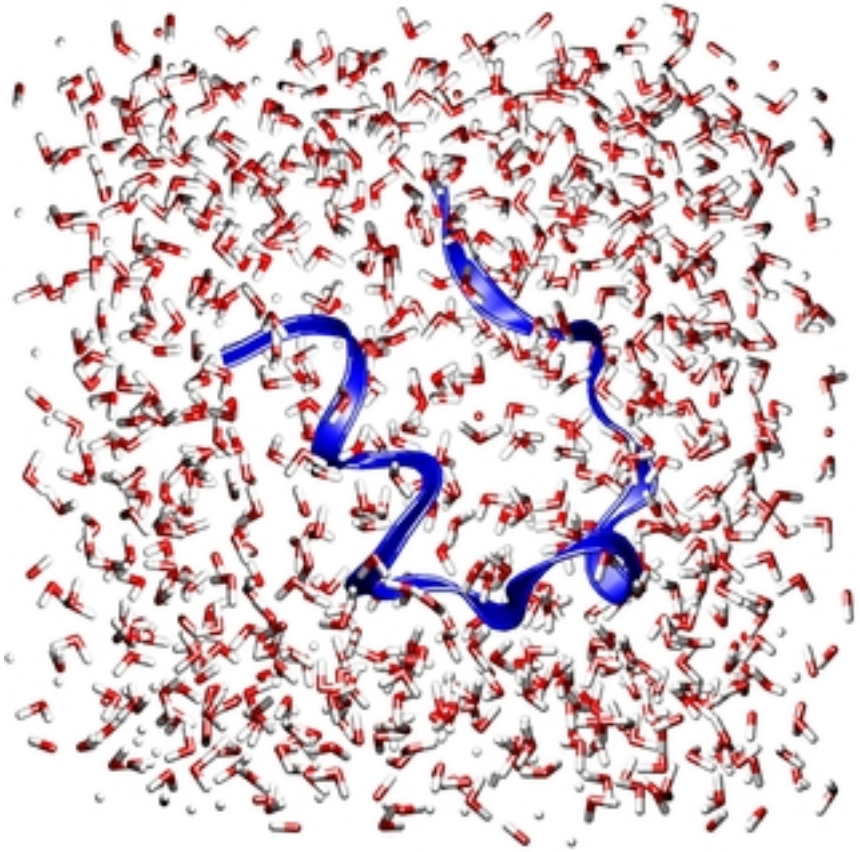

Figura 3.7: Vista de la estructura de una proteína en el agua.

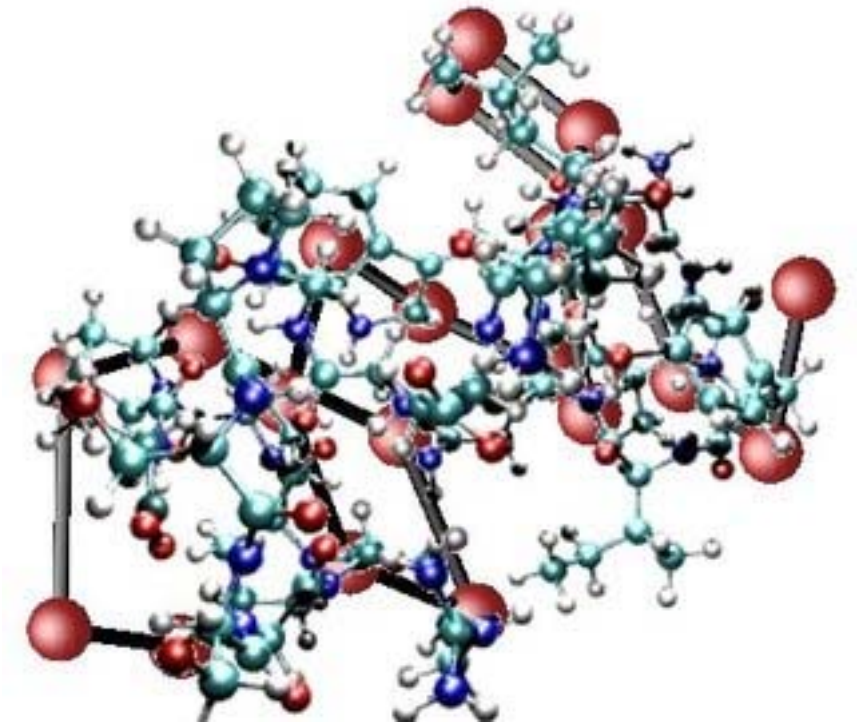

Figura 3.8: Plegamiento de péptidos por métodos de optimización global.

3.7 Conclusión

Algunas de las ventajas presentes en esta herramienta son, la configuración de los parámetros de las simulaciones se realiza muy cómodamente desde la interfaz gráfica, sin tener necesidad de editar archivos de configuración, y además trae incorporada la implementación de la ejecución en paralelo de los algoritmos de la dinámica molecular. Sin embargo, posee carencias importantes desde el punto de vista físico, debido a que los potenciales ofrecidos son de naturaleza química. Lo cual dificulta lograr el objetivo del presente trabajo, que es construir un marco de trabajo para posibilitar la realización de simulaciones de la física de medios granulares.

Se evaluó la posibilidad de extender las funcionalidades de la herramienta de modo de contemplar interacciones físicas. Sin embargo, al momento de realizar este estudio ya no se disponía de apoyo por parte de los creadores, ni de una comunidad, ya que la solución pasó de ser código libre a ser propietario. Además, no se disponía de manera gratuita de ningún tipo de documentación del producto. Por lo tanto, se decidió continuar investigando alternativas más alineadas al objetivo del presente trabajo.

Dado que la herramienta no fue aceptada desde el punto de vista funcional, ya que no se adecuaba a las necesidades del presente trabajo, no se procedió a instalar la misma sobre un cluster. Las pruebas de concepto fueron realizadas sobre un único computador con varios procesadores.

CAPÍTULO 4
Software
Computational Granular Dynamics

4.1 Introducción

En este capítulo se desarrollará los distintos aspectos sobre la aplicación del libro *Computational Granular Dynamics* (CGD), que se puede encontrar en la referencia *[1]*.

La *performance* de cualquier programa que implementa Dinámica Molecular es determinada por el cálculo de la fuerza de interacción entre las partículas.

En este capítulo serán presentados dos enfoques para el cálculo de la fuerza:

- Un método simple pero ineficiente del cálculo de la fuerza, que provee un completo panorama del programa a partir de un mínimo esfuerzo de programación.
- El método de la lista de Verlet.

Los programas aquí presentados tienen una estructura modular. La primera implementación (cálculo simple de la fuerza) sirve como referencia. El código para métodos más eficientes se basa en el programa anterior, donde partes del programa simple son reemplazados por funciones más eficientes. De una manera similar el tipo básico de partícula *sphere* puede ser reemplazado por tipos de partículas más complejos.

4.2 Esquema de integración de Gear

El algoritmo de Gear es un método que consta de dos pasos básicos: predictor y corrector, se caracteriza principalmente por su estabilidad numérica, en la integración de la ecuación del movimiento de Newton para sistemas granulares. El mismo es más complicado cuando se lo compara con otros esquemas de integración. Sin embargo, solo una pequeña parte del tiempo de cálculo se utiliza en la integración de las ecuaciones del movimiento. La mayor parte del tiempo de cálculo es usado en la evaluación de las fuerzas que actúan sobre las partículas en cada paso de tiempo. Consecuentemente, el utilizar un algoritmo menos complicado de integración no ahorra mucho tiempo de cálculo. Por el contrario, se desperdicia tiempo de cálculo al haber usado un método de integración menos estable numéricamente, dado que para alcanzar una precisión parecida, se necesita un paso de tiempo más pequeño lo cual incrementa el número de evaluaciones de fuerzas.

El algoritmo de Gear tiene otra importante ventaja sobre muchos otros esquemas de integración: en cada paso de tiempo solo se requiere una evaluación de la fuerzas de interacción. Por lo tanto, hay una enorme ganancia en la eficiencia dado que la costosa evaluación de fuerzas es realizada con menor frecuencia.

4.3 Estructura del programa

Se especificará el algoritmo de dinámica molecular, para la simulación de un sistema granular de partículas esféricas. En está sección se presenta una simple pero completa implementación del programa. Este programa sirve como base para en un futuro:

- Mejorar la eficiencia de los algoritmos y
- Utilizar modelos de partículas más complejos

Estas modificaciones requieren pequeños cambios en los programas que serán descriptos. Se comenzará la discusión con la implementación de los algoritmos más básicos de la dinámica molecular. El código de este programa esta estructurado en cinco archivos como se muestra en la figura 4.1.

common.h common.sphere.cc	Contiene el código, las declaraciones de todas las variables globales y funciones que, son utilizadas en más de un archivo fuente. Estos archivos son comunes para todos los posibles algoritmos de cálculo de fuerza y modelos de partículas.
simple.cc	Contiene el código para realizar un simple (pero ineficiente) cálculo de la fuerza que, será mejorado a posteriori.
sphere.h sphere.cc	Declaración e implementación de las partículas de la simulación mediante la clase *sphere*. Esta clase contiene la información que representa a una partícula esférica y las funciones para computar su dinámica.

Figura 4.1: Lista de archivos de CGD.

La organización del código en estos cinco archivos refleja la estructura modular del programa. Las simulaciones de la dinámica molecular, utilizan diferentes métodos de cálculo de fuerza y distintos modelos de partículas que pueden ser fácilmente construidos combinando los archivos fuentes apropiados. Sin embargo, está organización tiene un inconveniente: los archivos no pueden ser descriptos de forma secuencial para lograr una explicación sistemática del código. En virtud de una explicación sistemática, el código es descripto independientemente de cómo fue organizado en los archivos.

El archivo de cabecera *common.h* (ver figura 4.2) contiene todas las declaraciones que son comunes a todos los programas de simulación, por ejemplo, para todos los tipos de partículas y todos los métodos de cálculo de fuerza. Declaraciones que son usadas en un único archivo fuente no se encuentran en *common.h*.

```
1   #ifndef _common_h
2   #define _common_h
3
4   #include <math.h>
5   #include <fstream>
6   #include <vector>
7   #include <set>
8   #include <string>
9
10  using namespace std;
11
12  extern double lx, ly;
13  extern double x_0, y_0;
14
15  extern unsigned int no_of_particles;
16  extern double Time;
17  extern ofstream fphase;
18  extern ofstream fenergy;
19
20  void step();
21  void make_forces();
22  void integrate();
23  void init_algorithm();
24  void phase_plot(ostream & os);
25  #endif
```

Figura 4.2: Contenido del archivo common.h. *Tomado de [1].*

Luego, de incluir las cabeceras de las clases necesarias proporcionadas por el lenguaje, se define el tamaño del sistema de coordenadas *lx*, *ly* y la posición del origen a partir de *x_0*, *y_0*. A continuación, se declaran las siguientes variables globales: *no_of_particles* lleva la cuenta del número total de partículas, *Time* representa el tiempo transcurrido, y los archivos de salida *fphase* y *fenergy* para salvar la posición, velocidad, etc. en cada paso y registrar la energía del sistema respectivamente. Por último, se declaran las firmas de las funciones que serán usadas en más de un archivo fuente.

En la figura 4.3 se visualiza una tabla donde se muestra entre paréntesis rectos el archivo fuente que implementa cada una de las funciones declaradas en *common.h*.

Función [archivo]	Descripción
init_algorithm() [simple.cc]	Inicializa todos los datos específicos para el cálculo de la fuerza.
step() [simple.cc]	Avanza el paso de la simulación.
integrate() [common.sphere.cc]	Implementa la ecuación de movimiento de Newton a partir del algoritmo de Gear (esquema predictor-corrector).
make_forces() [simple.cc]	Calcula la fuerza resultante sobre cada partícula.
phase_plot() [common.sphere.cc]	Salva la posición y velocidad actual, entre otros valores, en un archivo de salida.

Figura 4.3: Funciones declaradas en common.h.

La parte del código común a todos los programas esta contenida en el archivo *comnon.sphere.cc* y se muestra en la figura 4.4. Lo primero que incluye es el encabezado de la clase *sphere.h*. Esta clase describe las propiedades de las partículas esféricas y las funciones específicas para su manipulación.

```
1   #include "common.h"
2   #include "sphere.h" /* <===== replace this line if necessary */
3   using namespace std;
4
5   double Time=0, timestep;
6   int nstep, nprint, nenergy;
7   Vector G;
8   double lx, ly, x_0, y_0;
9   vector<sphere> particle; /* <===== replace this line if necessary */
10  unsigned int no_of_particles;
11
12  ofstream fphase("phase.dat"), flast("lastframe.dat"), fenergy("energy.dat");
13
14  void init_system(char * fname);
15  double total_kinetic_energy();
```

Figura 4.4: Primer fragmento de código de common.sphere.cc. *Tomado de [1].*

Las variables *nstep*, *timestep* y *nprint* hacen referencia a la cantidad de pasos ejecutados, la duración de cada paso y la cantidad de pasos entre dos salidas de datos sucesivas respectivamente. Los datos que se pueden salvar son la posición de la partícula y la velocidad entre otros valores. El intervalo entre dos cálculos sucesivos de la energía total en el sistema es controlado por la constante *nenergy*. La variable *G* representa la gravedad. El vector *particle* contiene toda la información acerca de las partículas esféricas. Cuando se quiere modelar partículas con otra forma, se debe reemplazar el tipo *sphere* por el tipo deseado. Las variables *no_of_particles*, *lx*, *ly*, *x_0*, *y_0* y *Time* son definidas aquí, pero ya han sido declaradas en *common.h*. A los archivos de salida *fphase* y *fenergy* se les asigna el nombre "*phase.dat*" y "*energy.dat*", respectivamente.

Se incluye las firmas para las funciones *init_system()* y *total_kinetic_energy()*. La función *init_system()* inicializa la simulación a partir de un archivo de entrada, cuyo nombre es pasado por parámetro. La función *total_kinetic_energy()* calcula la energía cinética del sistema. Estas funciones son utilizadas únicamente en este archivo, por lo que no es necesario incluirlas en *common.h*.

```
16  int main(int argc, char ** argv)
17  {
18  if(argc!=2){
19         cerr << "usage: " << argv[0] << " particle_initfile\n";
20         exit(0);
21  }
22  fenergy.precision(10);
23  init_system(argv[1]);
24  init_algorithm();
25  phase_plot(fphase);
26  for(int i=0;i<nstep;i++){
27         step();
28         if((i+1)%nprint==0){
29                cout << "phase_plot: " << i+1 << " " << particle.size()
                   << " particles\n";
30                phase_plot(fphase);
31         }
32         if((i+1)%nenergy==0){
33                fenergy << Time << "\t" << total_kinetic_energy() << endl;
34         }
35  }
36  phase_plot(flast);
37  }
```

Figura 4.5: Segundo fragmento de código de common.sphere.cc. *Tomado de [1].*

En la figura 4.5 se muestra la implementación de la función *main*, la cual primero comprueba la correcta invocación (líneas de 18 a 21), que debe contener el nombre del archivo de inicialización como único parámetro. El archivo de entrada contiene las posiciones y velocidades iniciales de todas las partículas. Si el programa es invocado erróneamente con otra cantidad de argumentos, se muestra en pantalla una breve descripción, y a continuación se termina el programa.

La llamada a la función *init_system()* es con el propósito de leer desde un archivo de entrada la posición inicial, velocidad, y las derivadas de mayor orden respecto al tiempo de cada partícula. Además, inicializa algunas variables específicas del sistema. La descripción de está función se realiza en la sección 4.5.

La llamada a la función *init_algorithm()* no tiene ningún efecto, ya que se esta implementando un algoritmo simple. Para el caso de algoritmos más sofisticados, como el descripto en la sección 4.7, está función sirve para la inicialización de algunas variables específicas del algoritmo.

En la iteración principal del programa, comenzando por la línea 27, se llama a la función *step() nstep* veces. En intervalos de *nprint* pasos, se llama a la función *phase_plot()* para que realice el volcado de datos, como la posición y velocidad actual de cada partícula y otros valores en el archivo de salida *fphase*. Cada *nenergy* pasos, se calcula la energía cinética total del sistema y es escrita en el archivo *fenergy*.

4.4 Algoritmo simple para el cálculo de fuerza

Las partes esenciales para el cálculo de la fuerza son implementadas en el archivo *simple.cc*. Como se muestra en la figura 4.6, este archivo comienza incluyendo a *common.h* debido a que implementa funciones definidas allí. También incluye a *sphere.h*, que es el encabezado de la clase que modela a partículas de forma esférica. En caso de requerir otra forma más compleja de partículas, se debe reemplazar la palabra *sphere* por el nombre del encabezado de la clase correcta.

```
1    #include "common.h"
2    #include "sphere.h" /* <===== replace this line if necessary */
3
4    using namespace std;
5
6    extern vector<sphere> particle; /* <=== replace this line if necessary */
```

Figura 4.6: Primer fragmento de código de simple.cc. *Tomado de [1].*

En la figura 4.7, se muestra la implementación de las funciones *init_algorithm()* y *step()*, para la versión simple del algoritmo de dinámica molecular presentada en esta sección.

```
7    void init_algorithm()
8    {
9    }
10
11   void step()
12   {
13          integrate();
14   }
```

Figura 4.7: Segundo fragmento de código de simple.cc. *Tomado de [1].*

Estas funciones son utilizadas cuando se cuenta con un algoritmo más eficiente de cálculo de fuerza, como se verá en la sección 4.7.

La función *integrate()*, invocada por *step()*, resuelve la ecuación del movimiento de Newton para partículas móviles de acuerdo al algoritmo de Gear. Las partículas móviles son aquellas cuyo movimiento esta determinado por fuerzas. El tipo de una partícula puede ser determinado mediante *ptype()* y para las partículas móviles el tipo es cero.

Las simulaciones de la dinámica molecular, descritas en este capítulo, siempre son realizadas con condiciones de borde periódicas. Mediante la definición de muros estacionarios o móviles, otras condiciones pueden ser aplicadas en los bordes.

```
38      void integrate()
39      {
40      for(unsigned int i=0;i<particle.size();i++){
41             if(particle[i].ptype()==0) {
42                    particle[i].set_force_to_zero();
43                    particle[i].predict(timestep);
44             } else {
45                    particle[i].boundary_conditions(i,timestep, Time);
46             }
47      }
48      make_forces();
49      for(unsigned int i=0;i<particle.size();i++){
50             if(particle[i].ptype()==0) particle[i].correct(timestep);
51      }
52      for(unsigned int i=0;i<particle.size();i++){
53             particle[i].periodic_bc(x_0, y_0, lx, ly);
54      }
55      Time+=timestep;
56      }
```

Figura 4.8: Tercer fragmento de código de simple.cc. *Tomado de [1].*

Como se muestra en la figura 4.8, para cada partícula, primero se inicializa la fuerza en cero (línea 42). Para las partículas de tipo 0 la función *predict()* es invocada para realizar la primera parte del algoritmo de Gear. Luego, usando la estimación de las posiciones y velocidades de la partícula, *make_forces()* evalúa las fuerzas que actúan sobre las partículas. En el segundo paso del algoritmo predictor-corrector, *correct()*, es ejecutada para las partículas móviles (tipo cero). Esta función corrige la predicción de las posiciones, velocidades y derivadas de mayor orden de la partícula mediante el cálculo de fuerzas. Finalmente el tiempo transcurrido en la simulación *Time* es actualizado.

Las partículas que están en reposo o que se mueven de manera arbitraria son usadas para modelar las fronteras del sistema, se las conoce como *partículas-muro*. Su tipo es distinto de cero. El movimiento de las *partículas-muro* no es gobernado por la ecuación de Newton del movimiento. Para determinar las posiciones y velocidades de tales partículas, la función *Boundary_conditions()* en la línea 45 es invocada para todas las partículas de tipo distinto de cero.

La subdivisión de las *partículas-muro* en diferentes tipo (1, 2,…) permite la propagación de diferentes *partículas-muro* con diferentes trayectorias. Diferentes formas de partículas (por ejemplo, partículas esféricas o más complicadas) requieren distintos tipos de condiciones de borde, por lo tanto, *boundary_conditions()* es un miembro de la clase partícula.

Basado en las predicciones de las posiciones y velocidades de las partículas, la función *make_forces()* calcula la fuerza que actúa sobre todas las partículas. La implementación más simple de esta función es presentada en la figura 4.9.

```
15    void make_forces()
16    {
17    for(unsigned int i=0;i<particle.size()-1;i++){
18           for(unsigned int k=i+1;k<particle.size();k++){
19                  if((particle[i].ptype()==0) || (particle[k].ptype()==0)){
20                         force(particle[i],particle[k],lx,ly);
21                  }
22           }
23    }
24    }
```

Figura 4.9: Cuarto fragmento de código de simple.cc. *Tomado de [1].*

Para cada par de partículas *(i,k)*, si *i* o *k* es de tipo cero, la función *force()* es invocada, la cual calcula la fuerza de interacción y la adiciona a la fuerza total de cada partícula. La función *force()* es especifica del modelo de partículas utilizado, por lo tanto, su implementación se encuentra en la clase partícula.

El método simple para calcular la fuerza de interacción es presentada aquí solo para obtener un programa completo de dinámica molecular que represente, con mínimo esfuerzo, la base para construir algoritmos más eficientes. Este algoritmo es adecuado solo para sistemas muy pequeños, con un par de docenas de partículas. Para grandes cantidades de partículas, es muy ineficiente dado que la mayoría de los *N(N-1)/2* pares de partículas no interactúan y pueden ser excluidas de los cálculos de fuerzas. Esta idea, lleva a mejoras significantes en la *performance* de la simulación.

4.5 La clase partícula: s*phere*

Las partículas esféricas son representadas como objetos de la clase *sphere*. Esta clase contiene toda la información que caracteriza a una partícula y todos los métodos que participan en el cálculo de su dinámica. La implementación de la clase partícula consiste de dos archivos, s*phere.h* y s*phere.cc*. Además en el cabezal de la clase partícula, se define la función *normalize()*.

En la figura 4.10, se muestra el archivo cabecera *sphere.h*, el cual contiene todas las declaraciones de la clase *sphere*. Los operadores >>, <<, la función *Distance()* y *force()* son declaradas como *friends*, lo que permite el acceso a los datos privados de la clase. Estas funciones no son miembros de la clase dado que no operan sobre una partícula, sino que operan en pares de partículas. La función *Distance()*, calcula la distancia espacial de dos partículas. Como una excepción, se escribe el nombre de esta función con una D mayúscula, para evitar conflicto con la función *distance()* de la *Standard Template Library*. Dado que las condiciones de borde periódicas son simuladas, la función *normalize()* (de la línea 8 a la 13) es definida para seleccionar la imagen periódica adecuada de una coordenada.

```
1   ifndef _sphere_h
2   #define _sphere_h
3
4   #include <iostream>
5   #include "Vector.h"
6   using namespace std;
7   inline double normalize(double dx, double L)
8   {
9          while(dx<-L/2) dx+=L;
10         while(dx>=L/2) dx-=L;
11         return dx;
12  }
13
14  class sphere {
15         friend istream & operator >> (istream & is, sphere & p);
16         friend ostream & operator << (ostream & os, const sphere & p);
17         friend double Distance(const sphere & p1, const sphere & p2,
18                                double lx, double ly){
19                double dx=normalize(p1.rtd0.x()-p2.rtd0.x(),lx);
20                double dy=normalize(p1.rtd0.y()-p2.rtd0.y(),ly);
21                return sqrt(dx*dx+dy*dy);
22         }
23
24         friend void force(sphere & p1, sphere & p2, double lx, double ly);
25  public:
26         sphere(): rtd0(null), rtd1(null), rtd2(null), rtd3(null), rtd4(null)
27         {}
28         Vector & pos() {return rtd0;}
29         Vector pos() const {return rtd0;}
30         double & x() {return rtd0.x();}
31         double x() const {return rtd0.x();}
32         double & y() {return rtd0.y();}
33         double y() const {return rtd0.y();}
34         double & phi() {return rtd0.phi();}
35         double phi() const {return rtd0.phi();}
36         double & vx() {return rtd1.x();}
37         double vx() const {return rtd1.x();}
38         double & vy() {return rtd1.y();}
39         double vy() const {return rtd1.y();}
40         double & omega() {return rtd1.phi();}
41         double omega() const {return rtd1.phi();}
42         const Vector & velocity() const {return rtd1;}
43         double & r() {return _r;}
44         double r() const {return _r;}
45         double m() const {return _m;}
46         int ptype() const {return _ptype;}
47         void predict(double dt);
48         void add_force(const Vector & f){_force+=f;}
49         void correct(double dt);
50         double kinetic_energy() const;
51         void set_force_to_zero(){_force=null;}
52         void boundary_conditions(int n, double timestep, double Time);
53         void periodic_bc(double x_0, double y_0, double lx, double ly);
54  private:
55         double _r, _m, _J;
56         int _ptype;
57         double Y,A,mu,gamma;
58         Vector rtd0,rtd1,rtd2,rtd3,rtd4;
59         Vector _force;
60  };
61  #endif
```

Figura 4.10: Contenido del archivo sphere.h. *Tomado de [1].*

Las variables privadas de la clase son: su radio *_r*; el momento de inercia *_J*; su masa *_m*; el tipo de partícula *_ptype*; las constantes del material *Y*, *A*, *mu*, y *gamma*, que son necesarias para calcular la fuerza de interacción de la partícula; así como toda la información necesaria para describir su movimiento, por ejemplo, el vector posición *rtd0*, el vector velocidad *rtd1*, y los vectores *rtd2, rtd3* y *rtd4* de las derivadas de mayor orden respecto al tiempo usadas por el esquema de integración de Gear. El vector *_force* representa la fuerza total que actúa sobre la partícula dado su interacción con otras partículas móviles y *partículas-muro*. Esta fuerza no toma en cuenta la gravedad, al contrario de la función *correct()*.

La posición y velocidad de la partícula puede ser accedida y modificada tanto por la funciones de la forma *double & x()* como por *double x() const*. Finalmente la declaración de la clase contiene la definición de algunas funciones simples y los prototipos de funciones más complejas las cuales son explicadas en la figura 4.11.

Función	Descripción
add_force()	Adiciona un valor a la fuerza total *_force* de la partícula.
ptype()	Devuelve el tipo de la partícula. Las partículas de tipo 0 están sujetas a la ecuación del movimiento de Newton, las partículas de otros tipos conforman los muros.
predict()	Primer paso del algoritmo de Gear.
correct()	Segundo paso del algoritmo de Gear.
force()	Calcula la fuerza de interacción de dos partículas y suma su resultado a la fuerza total de ambas.
boundary_conditions()	Calcula la posición y velocidad de las partículas de tipos distintos de cero.
periodic_bc()	Asegura el cumplimiento de las condiciones de borde periódicas.
kinetic_energy()	Calcula la energía cinética de la partícula.
<<	Operador de salida.
>>	Operador de entrada.

Figura 4.11: Funciones de sphere.h.

Las funciones *predict()* y *correct()* (ver figura 4.12) implementan el esquema de integración de Gear; *predict()* calcula las posiciones, velocidades y derivadas de mayor orden respecto al tiempo a partir del desarrollo del polinomio Taylor evaluado en los valores actuales. La función *correct()* corrige la posición y derivadas respecto al tiempo estimadas, usando las fuerzas calculadas en la función *force()*.

```
1   #include "sphere.h"
2   #include <assert.h>
3
4   extern Vector G;
5
6   void sphere::predict(double dt)
7   {
8           double a1=dt;
9           double a2=a1*dt/2;
10          double a3=a2*dt/3;
11          double a4=a3*dt/4;
12
13          rtd0 += a1*rtd1 + a2*rtd2 + a3*rtd3 + a4*rtd4;
14          rtd1 += a1*rtd2 + a2*rtd3 + a3*rtd4;
15          rtd2 += a1*rtd3 + a2*rtd4;
16          rtd3 += a1*rtd4;
17  }
18
19  void sphere::correct(double dt)
20  {
21          static Vector accel,corr;
22          double dtrez = 1/dt;
23          const double coeff0=double(19)/double(90)*(dt*dt/double(2));
24          const double coeff1=double(3)/double(4)*(dt/double(2));
25          const double coeff3=double(1)/double(2)*(double(3)*dtrez);
26          const double coeff4=double(1)/double(12)*(double(12)*(dtrez*dtrez));
27
28          accel=Vector((1/_m)*_force.x()+G.x(),
29          (1/_m)*_force.y()+G.y(),
30          (1/_J)*_force.phi()+G.phi());
31          corr=accel-rtd2;
32          rtd0 += coeff0*corr;
33          rtd1 += coeff1*corr;
34          rtd2 = accel;
35          rtd3 += coeff3*corr;
36          rtd4 += coeff4*corr;
37  }
```

Figura 4.12: Primer fragmento de código de sphere.cc. *Tomado de [1].*

En la figura 4.13, la función *force()* calcula la fuerza ejercida por dos esferas, *p1* y *p2*, entre sí. Notar que esta función tiene que ser definida para todos los modelos de partículas utilizados.

```
38    void force(sphere & p1, sphere & p2, double lx, double ly)
39    {
40            double dx=normalize(p1.x()-p2.x(),lx);
41            double dy=normalize(p1.y()-p2.y(),ly);
42            double rr=sqrt(dx*dx+dy*dy);
43            double r1=p1.r();
44            double r2=p2.r();
45            double xi=r1+r2-rr;
46
47            if(xi>0){
48                    double Y=p1.Y*p2.Y/(p1.Y+p2.Y);
49                    double A=0.5*(p1.A+p2.A);
50                    double mu = (p1.mu<p2.mu ? p1.mu : p2.mu);
51                    double gamma = (p1.gamma<p2.gamma ? p1.gamma : p2.gamma);
52                    double reff = (r1*r2)/(r1+r2);
53                    double dvx=p1.vx()-p2.vx();
54                    double dvy=p1.vy()-p2.vy();
55                    double rr_rez=1/rr;
56                    double ex=dx*rr_rez;
57                    double ey=dy*rr_rez;
58                    double xidot=-(ex*dvx+ey*dvy);
59                    double vtrel=-dvx*ey + dvy*ex + p1.omega()*p1.r()-
                                    p2.omega()*p2.r();
60                    double fn=sqrt(xi)*Y*sqrt(reff)*(xi+A*xidot);
61                    double ft=-gamma*vtrel;
62
63                    if(fn<0) fn=0;
64                    if(ft<-mu*fn) ft=-mu*fn;
65                    if(ft>mu*fn) ft=mu*fn;
66                    if(p1.ptype()==0) {
67                            p1.add_force(Vector(fn*ex-ft*ey, fn*ey+ft*ex, r1*ft));
68                    }
69                    if(p2.ptype()==0) {
70                            p2.add_force(Vector(-fn*ex+ft*ey, -fn*ey-ft*ex, -r2*ft));
71                    }
72            }
73    }
```

Figura 4.13: Segundo fragmento de código de sphere.cc. *Tomado de [1].*

Desde la línea 48 hasta la 51, los parámetros del material para la colisión son calculados tomando los parámetros del material de las partículas en colisión. La función *force()* continua con el cálculo de la fuerza normal y el componente tangencial. Luego, para las partículas de tipo cero, la fuerza de interacción es sumada a la fuerza total de cada partícula mediante la invocación de *add_force()*.

Los muros de nuestra simulación son construidos a partir de partículas para modelar una superficie rugosa. El movimiento de estas partículas no es gobernado por su interacción con otras partículas, pero siguen una trayectoria predecible. Esto no afecta solo a los muros externos del área de simulación, sino también a otros objetos que no se mueven de acuerdo a la ley de Newton.

Para muros fijos, estas partículas no se pueden mover sin importar que fuerzas actúen sobre ellas. Para muros oscilantes, las *partículas-muro* siguen una trayectoria sinusoidal. Para distinguir las partículas cuyo movimiento es determinado por las fuerzas de interacción de aquellas que obedecen condiciones de borde, se utiliza la variable *_ptype* (se accede mediante la función *ptype()*). Por convención, las partículas móviles de materiales granulares son de tipo 0; los otros tipos son usados para distinguir las partículas que están sujetas a condiciones de borde.

En la figura 4.14, la función *boundary_conditions()* espera tres parámetros: el índice *n* de la partícula, el paso de integración, y el tiempo actual de simulación.

```
74    void sphere::boundary_conditions(int n, double timestep, double Time)
75    {
76            switch(ptype()){
77            case(0): break;
78            case(1): break;
79            case(2): {
80                    x()=0.5-0.4*cos(10*Time);
81                    y()=0.1;
82                    vx()=10*0.4*sin(Time);
83                    vy()=0;
84            } break;
85            case(3): {
86                    double xx=x()-0.5;
87                    double yy=y()-0.5;
88                    double xp=xx*cos(timestep)-yy*sin(timestep);
89                    double yp=xx*sin(timestep)+yy*cos(timestep);
90
91                    x()=0.5+xp;
92                    y()=0.5+yp;
93                    vx()=-yp;
94                    vy()= xp;
95                    omega()=1;
96            } break;
97            case(4): {
98                    x()=0.5+0.1*cos(Time) + 0.4*cos(Time+2*n*M_PI/128);
99                    y()=0.5+0.1*sin(Time) + 0.4*sin(Time+2*n*M_PI/128);
100                   vx()=-0.1*sin(Time) - 0.4*sin(Time+2*n*M_PI/128);
101                   vy()= 0.1*cos(Time) - 0.4*cos(Time+2*n*M_PI/128);
102                   omega()=1;
103           } break;
104           case(5): {
105                   y()=0.1+0.02*sin(30*Time);
106                   vx()=0;
107                   vy()=0.02*30*cos(30*Time);
108           } break;
109           case(6): {
110                   int i=n/2;
111                   y()=i*0.02+0.1+0.02*sin(30*Time);
112                   vx()=0;
113                   vy()=0.02*30*cos(30*Time);
114           } break;
115           default: {
116                   cerr << "ptype: " << ptype() << " not implemented\n";
117                   abort();
118           }
119           }
120   }
```

Figura 4.14: Tercer fragmento de código de sphere.cc. *Tomado de [1].*

Los siete tipos de partículas implementados sirven solo a modo de ejemplo. En general, cada problema requiere su propio conjunto de tipos de partículas de acuerdo a las condiciones de borde del problema en cuestión. A continuación se explican cada uno de los tipos de partículas de ejemplo:

- *tipo 0*: Las partículas granulares de este tipo son aquellas cuyo movimiento es determinado por las interacciones con otras partículas, de acuerdo a la ley de Newton del movimiento. El movimiento de estas partículas es calculado por las funciones *predict()* y *correct()*, resolviendo así la ecuación del movimiento para la partícula.
- *tipo 1*: La función *boundary_conditions()* no realiza cambio alguno sobre la posición y velocidad de este tipo de partículas. Estas partículas son ideales para la representación de muros estáticos. Las funciones *predict()* y *correct()* no son invocadas para estas partículas, en contraste con las de tipo 0.
- *tipo 2*: Las partículas oscilan en dirección horizontal, siguiendo una trayectoria de la forma: $x(t) = [0.5 - 0.4 \cos(10\ t/seg)]\ m$.
- *tipo 3*: Las partículas giran con una frecuencia de 1/seg. alrededor del centro x = 0,5 m, y = 0,5 m, por lo tanto, simulando la rotación de un contenedor de forma arbitraria.
- *tipo 4*: Estas partículas son dispuestas formando un círculo (de radio = 0.4m), cuyo centro realiza una rotación de radio 0.1m con una frecuencia de 1/seg. Este tipo de movimiento es usado para modelar un cilindro que realiza un movimiento en forma de espiral.
- *tipo 5*: Las partículas oscilan verticalmente con una frecuencia $\omega = 30/seg$ y amplitud de 10 ± 2 cm.
- *tipo 6*: las partículas realizan el mismo movimiento que las partículas de tipo 5, pero alrededor del eje vertical en $y = (2i + 10)cm$ siendo i el índice de la partícula dividido dos (los decimales son truncados).

Se puede fácilmente definir más tipos de partículas agregando más sentencias de tipo case en la función *boundary_conditions()*. Los ejemplos mostrados, son con el objetivo de demostrar como se implementan nuevas condiciones de borde.

La función *periodic_bc()* implementa condiciones de borde periódicas (ver figura 4.15). Es decir, coloca la partícula en la posición de su imagen periódica dentro del área de simulación si traspasa los límites del sistema.

```
121  void sphere::periodic_bc(double x_0, double y_0, double lx, double ly)
122  {
123         while(rtd0.x()<x_0) rtd0.x()+=lx;
124         while(rtd0.x()>x_0+lx) rtd0.x()-=lx;
125         while(rtd0.y()<y_0) rtd0.y()+=ly;
126         while(rtd0.y()>y_0+ly) rtd0.y()-=ly;
127  }
```

Figura 4.15: Cuarto fragmento de código de sphere.cc. *Tomado de [1].*

Como ejemplo de procesamiento de datos durante la simulación, se cuenta con la función *kinetic_energy()* ilustrada en la figura 4.16, la cual devuelve la energía cinética de la partícula.

```
128   double sphere::kinetic_energy() const
129   {
130           return _m*(rtd1.x()*rtd1.x()/2 + rtd1.y()*rtd1.y()/2)
131           + _J*rtd1.phi()*rtd1.phi()/2;
132   }
```

Figura 4.16: Quinto fragmento de código de sphere.cc. *Tomado de [1].*

En la figura 4.17 se visualiza la implementación de los operadores << y >>, los cuales se utilizan para la entrada y salida de datos de las partículas respectivamente, como la posición de la partícula, las derivadas respecto al tiempo y otros valores específicos de la partícula. El operador << envía toda la información disponible de la esfera a un flujo de salida (un archivo o la pantalla). Esta función es utilizada en *phase_plot()* para guardar el estado actual del sistema en un archivo de salida, el cual luego será usado para extraer los resultados de la simulación, o para recomenzar la simulación usando las velocidades, posiciones y demás valores de las partículas como valores iniciales.

```
133   istream & operator >> (istream & is, sphere & p)
134   {
135           is >> p.rtd0 >> p.rtd1
136                   >> p._r >> p._m >> p._ptype
137                   >> p.Y >> p.A >> p.mu >> p.gamma
138                   >> p._force
139                   >> p.rtd2 >> p.rtd3 >> p.rtd4;
140           p._J=p._m*p._r*p._r/2;
141           return is;
142   }
143   ostream & operator << (ostream & os, const sphere & p)
144   {
145           os << p.rtd0 << " " << p.rtd1 << " ";
146           os << p._r << " " << p._m << " " << p._ptype << " ";
147           os << p.Y << " " << p.A << " " << p.mu << " " << p.gamma << " ";
148           os << p._force << " ";
149           os << p.rtd2 << " " << p.rtd3 << " " << p.rtd4 << "\n" << flush;
150           return os;
151   }
```

Figura 4.17: Sexto fragmento de código de sphere.cc. *Tomado de [1].*

Para completar la descripción de la versión básica del programa para la Dinámica Molecular de partículas esféricas en dos dimensiones, se mencionarán algunas de las funciones que sirven para la inicialización del sistema y la extracción de datos.

La función *init_system()* visualizada en la figura 4.18, inicializa la simulación. Primero lee los parámetros globales del sistema como *G* (gravedad), *Time*, *nstep*, *timestep*, *nprint*, *nenergy*, *x_0, y_0, lx y ly*, desde el archivo de inicialización.

```
57    void init_system(char * fname)
58    {
59    ifstream fparticle(fname);
60    while(fparticle.peek()=='#'){
61           string type;
62           fparticle >> type;
63           if(type=="#gravity:"){
64                  fparticle >> G.x() >> G.y() >> G.phi();
65                  fparticle.ignore(100,'\n');
66                  cout << "gravity: " << G << endl;
67           } else if(type=="#Time:"){
68                  fparticle >> Time;
69                  fparticle.ignore(100,'\n');
70                  cout << "Time: " << Time << endl;
71           } else if(type=="#nstep:"){
72                  fparticle >> nstep;
73                  fparticle.ignore(100,'\n');
74                  cout << "nstep: " << nstep << endl;
75           } else if(type=="#timestep:"){
76                  fparticle >> timestep;
77                  fparticle.ignore(100,'\n');
78                  cout << "timestep: " << timestep << endl;
79           } else if(type=="#nprint:"){
80                  fparticle >> nprint;
81                  fparticle.ignore(100,'\n');
82                  cout << "nprint: " << nprint << endl;
83           } else if(type=="#nenergy:"){
84                  fparticle >> nenergy;
85                  fparticle.ignore(100,'\n');
86                  cout << "nenergy: " << nenergy << endl;
87           } else if(type=="#lx:"){
88                  fparticle >> lx;
89                  fparticle.ignore(100,'\n');
90                  cout << "lx: " << lx << endl;
91           } else if(type=="#ly:"){
92                  fparticle >> ly;
93                  fparticle.ignore(100,'\n');
94                  cout << "ly: " << ly << endl;
95           } else if(type=="#x_0:"){
96                  fparticle >> x_0;
97                  fparticle.ignore(100,'\n');
98                  cout << "x_0: " << x_0 << endl;
99           } else if(type=="#y_0:"){
100                 fparticle >> y_0;
101                 fparticle.ignore(100,'\n');
102                 cout << "y_0: " << y_0 << endl;
103          } else {
104                 cerr << "init: unknown global property: " << type << endl;
105                 abort();
106          }
107   }
108   while(fparticle){
109          sphere pp;
110          fparticle >> pp;
111          if(fparticle){
112                 particle.push_back(pp);
113          }
114   }
115   no_of_particles=particle.size();
116   cout << no_of_particles << " particles read\n" << flush;
117   }
```

Figura 4.18: Primer fragmento de código de common.sphere.cc. *Tomado de [1].*

Luego de leer el último parámetro del sistema, que sucede cuando se encuentra la primera línea que no empiece con *"#"*, se comienzan a leer las características de cada partícula desde el archivo de entrada.

4.6 Salida de datos

Dependiendo del problema, el programa escribirá datos de salida que, son derivados a partir de la posición, velocidad, aceleración, etc. de la partícula, y de propiedades de la partícula como la masa, el radio, y el momento de inercia.

Un ejemplo de una función de salida es *total_kinetic_energy()*, mostrada en la figura 4.19, que devuelve la energía cinética total del sistema.

```
118   double total_kinetic_energy()
119   {
120   double sum=0;
121   for(unsigned int i=0;i<particle.size();i++){
122          if(particle[i].ptype()==0){
123                 sum+=particle[i].kinetic_energy();
124          }
125   }
126   return sum;
127   }
```

Figura 4.19: Segundo fragmento de código de common.sphere.cc. *Tomado de [1].*

Con frecuencia se desea analizar el resultado de una simulación fuera del tiempo de ejecución. El análisis de datos fuera de línea es particularmente ventajoso si el resultado de la simulación no es previsible. Además, la visualización de datos en línea con frecuencia no es posible debido al alto costo computacional, cada *frame* individual es producido a un ritmo muy lento como para generar una animación fluida. Por lo tanto, la función *phase_plot()* (ver implementación en la figura 4.20) es necesaria para extraer los datos, como las posiciones, velocidades, radios, masas, y derivadas de mayor orden (es decir, toda aquella información que es leída desde el archivo de inicialización). La función *phase_plot()* es invocada por el programa principal en intervalos regulares de tiempo.

```
128   void phase_plot(ostream & os)
129   {
130          os << "#NewFrame\n";
131          os << "#no_of_particles: " << no_of_particles << endl;
132          os << "#compressed: no\n";
133          os << "#type: sphereXYPhiVxVyOmegaRMFixed25\n";
134          os << "#gravity: " << G.x() << " " << G.y() << " " << G.phi() << endl;
135          os << "#Time: " << Time << endl;
136          os << "#timestep: " << timestep << endl;
137          os << "#EndOfHeader\n";
138          for(unsigned int i=0;i<particle.size();i++){
139                 os << particle[i];
140          }
141          os << flush;
142   }
```

Figura 4.20: Tercer fragmento de código de common.sphere.cc. *Tomado de [1].*

Al principio en la función *phase_plot*(), se escribe la información global en un formato de lenguaje natural. Esto sirve para dos propósitos. Primero, a menudo es beneficioso si los datos de la simulación y los correspondientes encabezados se almacenan juntos, ya que contribuye al mantenimiento del registro de resultados de la simulación. Si los encabezados y los resultados de la simulación son almacenados en diferentes archivos, y el archivo de encabezados se perdiera, entonces, los resultados se vuelven inservibles. Segundo, los cabezales pueden ser usados por herramientas de procesamiento de datos.

4.7 Cálculo eficiente de la fuerza

El programa de Dinámica Molecular descripto en la sección 4.2 es ineficiente desde el punto de vista computacional. La eficiencia de cualquier programa de Dinámica Molecular está principalmente determinada por el cálculo eficiente de las fuerzas que actúan sobre las partículas. Se va a explicar mediante un ejemplo: se estudiará el caso de una simulación de N = 1.000 partículas. En cada paso todas las posibles parejas de partículas tienen que ser consideradas con respecto a su fuerza de interacción, por lo tanto, son necesarias $N(N - 1)/2 \approx 500.000$ cálculos de fuerza. Para interacciones de corto alcance, la mayoría de estos cálculos son innecesarios, dado que las partículas están a una gran distancia entre ellas.

Para aproximadamente un mismo tamaño de partículas, cada partícula puede estar en contacto a lo sumo con otras 6 partículas, por lo tanto, sólo son necesarios 3N = 3.000 cálculos. Para el esquema original de cálculo (que se aplicó en la sección anterior), por lo menos, se calculan 166 parejas de interacciones innecesarias. Por lo tanto, es necesario aplicar métodos para reducir el número de parejas de interacciones que se consideran en cada paso. Dado que las fuerzas entre partículas granulares son de corto alcance, el cálculo de la fuerza puede ser restringido a los pares de partículas que son vecinas.

Se necesita identificar que partículas son vecinas cercanas, sin embargo, esto no es trivial, ya que todos los pares de partículas cercanos tienen que ser considerados. Aunque sólo uno de ellos no fuera identificado, los resultados de la simulación dejan de ser válidos. La solución intuitiva para comprobar si todos los pares de partículas son vecinos o no, lleva de nuevo a cerca de 500.000 cálculos, lo que originalmente se quería evitar. En esta sección, se va a describir un método eficiente para el cálculo de las fuerzas, denominado el algoritmo de Verlet.

4.7.1 Listas de Verlet

La idea del algoritmo de Verlet está basada en una simple propiedad de la dinámica de las partículas: las relaciones de vecindad entre las partículas cambian muy lentamente, es decir, dos partículas que están cerca una de la otra en un momento dado se van a mantener como vecinos por lo menos durante algunos pocos pasos siguientes.

En el momento de la inicialización, se determina la relación de vecindad entre partículas, es decir, se calcula la distancia de todos los pares de partículas cercanas. Se considerarán que dos partículas son vecinas si la distancia de sus superficies es menor

que una constante predefinida, a la cual se la denomina *verlet_distance*. Para el caso de partículas esféricas, este criterio puede ser expresado mediante la ecuación 4.1.

$$(| r_i - r_j | - R_i - R_j) < verlet\_distance.$$

Ecuación 4.1: Condición de vecindad entre partículas.

Cada partícula tiene su propia lista (lista de Verlet) que contiene sus vecinos. Para inicializar de manera eficiente las listas de Verlet, se define una grilla que cubre el área de simulación. El tamaño de la celda es más grande que el de la partícula de mayor tamaño. Para la construcción de las listas, solo se tendrá en cuenta a aquellos pares de partículas que residen en la misma celda o en celdas adyacentes. Este procedimiento garantiza la detección de todos los pares cercanos.

Las entradas simétricas en las listas de Verlet (si A es vecino de B entonces B es vecino de A) son evitadas por la restricción de que, la lista de Verlet de la partícula *i* solo contiene vecinos con índice $j < i$. La lista de Verlet de una partícula es, por lo tanto, un conjunto de índices de partículas. La estructura de datos requerida debe permitir tomar rápidas decisiones sobre si una partícula *j* pertenece a la lista de Verlet de una partícula *i*. Las listas de Verlet de todas las partículas son organizadas como un vector de tales listas.

Para el cálculo de las fuerzas de interacción de las partículas, solo son tenidos en cuenta los pares de partículas que pertenezcan a una de las listas de Verlet de las partículas involucradas. Por lo tanto, la lista de Verlet para cada partícula *i* es recorrida, calculando la fuerza de interacción entre *i* y cada entrada *j* de la lista. El algoritmo de Verlet es implementado en el archivo *verlet.cc*.

```
1   #include "common.h"
2   #include "sphere.h" /* <===== replace this line if necessary */
3
4   extern vector<sphere> particle; /* <===== replace this line if necessary */
5
6   vector<sphere> safe; /* <===== replace this line if necessary */
7   double Timesafe;
8
9   double verlet_ratio = 0.6, verlet_distance = 0.00025, verlet_grid = 0.05;
10  double verlet_increase = 1.1;
11  int vnx, vny;
12  double dx, dy;
13  vector<vector<vector<int> > > celllist;
14  vector<set<int> > verlet;
15
16  bool verlet_needs_update();
17  bool make_verlet();
18  bool do_touch(int i, int k);
```

Figura 4.21: Primer fragmento de código de verlet.cc. *Tomado de [1].*

Al comienzo de la especificación de *verlet.cc* mostrado en la figura 4.21, se declara el vector *particle*, al igual que en la versión simple del algoritmo de cálculo de fuerza. Las siguientes declaraciones son específicas del algoritmo de Verlet. El vector *safe* y la variable *Timesafe* sirven como un respaldo (para *particle* y *Time* respectivamente) en el caso de que las listas de Verlet se modifiquen de manera incorrecta. Los valores de *verlet_ratio*, *verlet_increase*, y *verlet_distance* rigen el

rendimiento del algoritmo. Finalmente las funciones *verlet_needs_update()*, *make_verlet()*, y *do_touch()* son declaradas.

Las funciones *init_algorithm()*, *step()*, y *make_forces()* que han sido introducidas en la sección 4.3, tienen que ser implementadas de acuerdo al algoritmo de Verlet. Dado que *make_forces()*, contiene la esencia del algoritmo de Verlet, se comenzará con la implementación de esta función, la cual se muestra en la figura 4.22. Anteriormente se había descripto la función *make_forces()* para calcular las fuerzas, considerando todos los pares de partículas.

```
19    void make_forces()
20    {
21    for(unsigned int i=0;i<particle.size();i++){
22           set<int>::iterator iter;
23           for(iter=verlet[i].begin();iter!=verlet[i].end();iter++){
24                  force(particle[i],particle[*iter], lx, ly);
25           }
26    }
27    }
```

Figura 4.22: Segundo fragmento de código de verlet.cc. *Tomado de [1].*

Dada una partícula *i*, en lugar de considerar a todas las otras partículas como potenciales compañeros de interacción, sólo las partículas que pertenecen a su lista Verlet van a ser consideradas. Se debe elegir un valor adecuado del parámetro *verlet_distance*, porque se corre el riesgo de que la lista de Verlet quede pequeña, lo que implica que el número de compañeros de interacción podría ser mucho menor que para el algoritmo simple. Las listas de Verlet son construidas en la función *make_verlet()* (ver implementación en la figura 4.23).

```
28    bool make_verlet()
29    {
30    bool ok=true;
31
32    verlet.resize(no_of_particles);
33    for(int ix=0;ix<vnx;ix++){
34           for(int iy=0;iy<vny;iy++){
35                  celllist[ix][iy].clear();
36           }
37    }
38    for(unsigned int i=0;i<no_of_particles;i++){
39           int ix=int((particle[i].x()-x_0)/dx);
40           int iy=int((particle[i].y()-y_0)/dy);
41           celllist[ix][iy].push_back(i);
42    }
43    for(unsigned int i=0;i<no_of_particles;i++){
44           set<int> oldverlet=verlet[i];
45           verlet[i].clear();
46           int ix=int((particle[i].x()-x_0)/dx);
47           int iy=int((particle[i].y()-y_0)/dy);
48           for(int iix=ix-1;iix<=ix+1;iix++){
49           for(int iiy=iy-1;iiy<=iy+1;iiy++){
50                  int wx = (iix+vnx)%vnx;
51                  int wy = (iiy+vny)%vny;
52                  for(unsigned int k=0;k<celllist[wx][wy].size();k++){
53                         int pk=celllist[wx][wy][k];
54                         if(pk<(int)i){
55                         if(Distance(particle[i],particle[pk], lx, ly)<
56                         particle[i].r()+particle[pk].r()+verlet_distance){
57                         if((particle[i].ptype()==0) ||
58                         (particle[pk].ptype()==0)){
59                                verlet[i].insert(pk);
60
61                                if(oldverlet.find(pk)==oldverlet.end()){
62                                if(do_touch(i,pk)) {
63                                       ok=false;
64                                }
65                                }
66                         }
67                         }
68                         }
69                  }
70           }
71           }
72    }
73    return ok;
74    }
```

Figura 4.23: Tercer fragmento de código de verlet.cc. *Tomado de [1].*

Al principio de la función *make_forces()* se especifica el número de listas de Verlet que existirán. Luego, las partículas son ordenadas dentro de una grilla de tamaño *dx* x *dy*. En cada celda de la grilla hay una lista de partículas residiendo allí. Estas listas son organizadas en un array de dos dimensiones denominado *celllist* (líneas 38-42).

Las listas de Verlet son construidas en el ciclo principal (líneas 43-72). Dado que se van a identificar nuevas interacciones para la partícula *i*, se procede a copiar la lista de Verlet a la variable *oldverlet* antes de inicializarla con la lista vacía. Los índices *ix* e *iy* que son calculados en las líneas 46 y 47, son utilizados para seleccionar la celda en la cuál reside la partícula *i*. En los ciclos que comienzan en las líneas 48 y 49, la celda (*ix*,*iy*) y sus celdas adyacentes son visitadas, en búsqueda de partículas *pk* que sean vecinas cercanas a la partícula *i*. Para evitar entradas simétricas en las listas de

Verlet, todas las partículas con $pk > i$ son descartadas. Para todas las partículas $pk < i$, en las líneas 55-58 se verifica si la distancia entre las partículas i y pk es menor que *verlet_distance* y si alguna de las dos partículas es de tipo 0. Si se cumple, el índice pk es agregado a la lista de Verlet de la partícula i. Si pk no se encuentra en la lista *oldverlet* (es decir, pk es un nuevo vecino) y las partículas se tocan entre si (*do_touch(i,pk)==true*), un error a ocurrido dado que las partículas han sido reconocidas como vecinos muy tarde. Tan pronto como este error ocurre, a la variable booleana *ok* se le asigna valor *false*. Si este error no ocurre con ninguna partícula, finaliza la construcción de las listas de Verlet y la variable *ok* mantiene su valor inicial de *true*. La función *make_verlet()* retorna la variable *ok*, es decir, *true* para indicar una construcción exitosa de las listas de Verlet y *false* para reportar un error.

```
75    bool do_touch(int i, int k)
76    {
77           return (Distance(particle[i],particle[k],lx,ly)
78                  <particle[i].r()+particle[k].r());
79    }
```

Figura 4.24: Cuarto fragmento de código de verlet.cc. *Tomado de [1].*

Como se muestra en la figura 4.24, la función *do_touch(int i, int k)* retorna *true* si las partículas están contacto.

Durante la simulación, hay cambios en las relaciones de vecindad de las partículas, por lo tanto, las listas de vecinos tienen que ser reconstruidas cada cierto tiempo. La función *verlet_needs_update()* es quien decide si es necesario reconstruir las listas de Verlet, dependiendo de cuan lejos han viajado las partículas desde el momento en que las listas vigentes fueron construidas. La lista de Verlet de una partícula i debe contener en todo momento a todos los vecinos j con $j < i$. Esto asegura que dos partículas i y j nunca se van a tocar si no son conocidos como vecinos. Por lo tanto, y considerando que R_k y r_k es el radio y el centro de una partícula k respectivamente, se tiene que la relación expresada por la ecuación 4.2 es requerida para todos los pares de partículas *(i,j)* que no son vecinas. Esta condición provee un criterio para la reconstrucción de las listas de Verlet.

$$|\vec{r_i} - \vec{r_j}| - R_i - R_j > 0$$

Ecuación 4.2: Condición de contacto entre partículas.

La figura 4.25 ilustra la siguiente idea: asumiendo que al momento en que se construyen las listas de Verlet, las superficies de las partículas que se encuentran a una distancia mayor a *verlet_distance* (ver ecuación 4.3) no son considerados vecinos.

$$|\vec{r_i} - \vec{r_j}| - R_i - R_j > verlet\_distance$$

Ecuación 4.3: Condición de vecindad entre partículas.

Si las listas de Verlet son actualizadas antes de que una de las partículas haya viajado una distancia de *verlet_distance/2* desde que las listas fueron construidas, las partículas no pueden colisionar sin haber sido identificadas primero como vecinas.

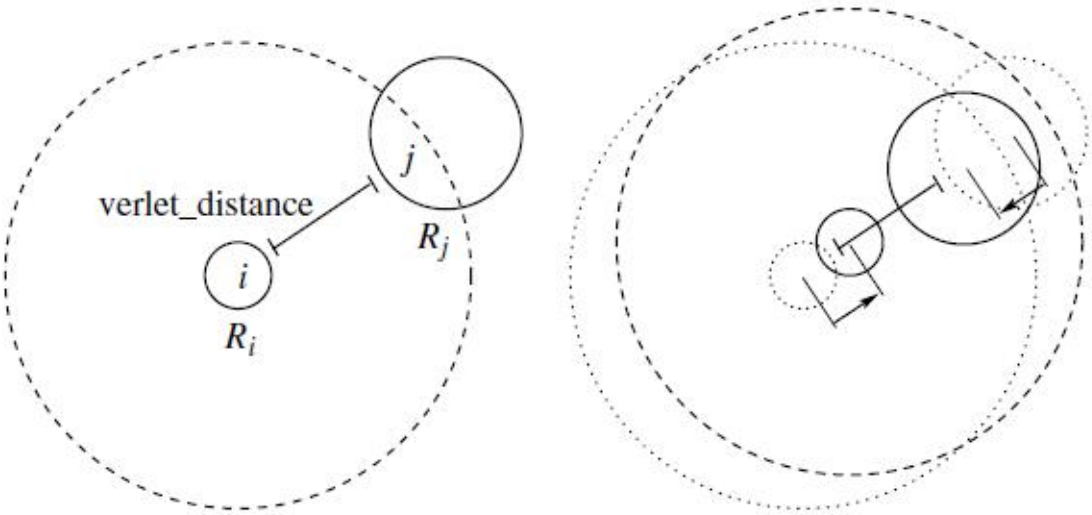

Figura 4.25: Verificación de la validez de las listas de Verlet. *Tomada de [1].*

En la parte izquierda de la figura 4.25, las partículas *i* y *j* no son reconocidas como vecinos dado que la distancia de sus superficies es mayor que *verlet_distance*. En el lado derecho se ilustra el peor caso, es cuando las partículas se acercan entre sí directamente, viajando a la misma velocidad. Tan pronto como una de las partículas haya viajado la distancia *verlet_distance/2* (las flechas de la imagen indican tal desplazamiento), las listas de Verlet tienen que ser reconstruidas. Las partículas *i* y *j* en este escenario son consideradas como vecinas.

Las posiciones de las partículas en el momento en que las listas de Verlet son construidas son almacenadas en la variable *safe*. Esta variable es usada para determinar la distancia que han viajado las partículas desde el momento de la construcción de las listas. Dado que los elementos de *safe* son copias exactas de las partículas, el calcular una distancia deseada puede ser determinada como cualquier otra distancia invocando, *Distance(particle[i],safe[i])*.

Como se muestra en la figura 4.26, la función *verlet_needs_update()* implementa el método descrito con una optimización adicional. En el caso de que la lista de Verlet tenga que ser reconstruida esta función toma el valor *true*.

```
1    bool verlet_needs_update()
2    {
3    for(unsigned int i=0;i<no_of_particles;i++){
4           if(Distance(particle[i],safe[i],lx,ly)>=verlet_ratio*verlet_distance){
5                  return true;
6           }
7    }
8    return false;
9    }
```

Figura 4.26: Quinto fragmento de código de verlet.cc. *Tomado de [1].*

En la práctica se encuentra que el criterio explicado de actualización de listas es muy estricto. El mismo se asegura de que ninguna colisión se pase por alto, aún en el peor caso, que se da cuando:

1. Al momento de construir las listas de Verlet dos partículas tienen una distancia un poco mayor que *verlet_distance* (es decir, no son clasificadas como vecinas).

2. Estas partículas se acercan entre sí de manera directa.
3. Estas partículas son las más rápidas de todo el sistema.
4. Ambas partículas se mueven con la misma velocidad absoluta (es decir, ambas viajan la distancia *verlet_distance/2*).

Sin embargo, este caso es muy poco probable que suceda. Por lo tanto, las listas pueden ser reconstruidas con un criterio más débil: las listas son reconstruidas cuando la partícula más rápida ha viajado una distancia de *verlet_ratio * verlet_distance*, donde *verlet_ratio > 0.5*. Encontrar el valor correcto requiere de experiencia, en muchos casos *verlet_ratio = 0.6* es apropiado. El impacto de esta optimización no es muy grande, dado que la construcción de las lista de Verlet ya es eficiente de por si; normalmente solo se gasta un pequeño porcentaje del total de cálculos en la construcción de las listas de Verlet.

Cuando *verlet_ratio > 0.5*, se puede dar el caso en que ocurra una colisión entre dos partículas que no fueron reconocidas como vecinas anteriormente, es decir, el par no esta contenido en ninguna de las listas de Verlet. Esta situación es fatal, dado que tales partículas no pueden experimentar ninguna fuerza de interacción y por lo tanto, se penetran mutuamente. Una vez detectado el problema en *make_verlet()*, se resuelve restaurando las posiciones, velocidades y las derivadas de mayor orden respecto al tiempo de las partículas a valores previos almacenados y recomenzando la simulación desde ese punto con un valor mayor de *verlet_distance* multiplicándolo por las por la constante *verlet_increase*. Típicamente se elije *verlet_increase = 1.1*.

El método descripto está implementado en la función *step()*. Aquí, se ha reemplazado la función *step()* del método simple de cálculo de la fuerza.

```
1     void step()
2     {
3     bool ok=true, newverlet=false;
4
5     if(verlet_needs_update()){
6            ok = make_verlet();
7            newverlet=true;
8     }
9     if(!ok){
10           cerr << "fail: going back from " << Time << " to " << Timesafe << endl;
11           particle=safe;
12           Time=Timesafe;
13           verlet_distance*=verlet_increase;
14           make_verlet();
15    }
16    if(newverlet && ok){
17           safe=particle;
18           Timesafe=Time;
19    }
20    integrate();
21    }
```

Figura 4.27: Sexto fragmento de código de verlet.cc. *Tomado de [1].*

Las listas de Verlet son construidas en la línea 6 de la figura 4.27. Si *make_verlet()* retorna el valor *true* (indica que no hay error), el tiempo transcurrido del sistema es guardado en la variable *Timesafe* y el estado de las partículas (posición, velocidad, y derivadas de mayor orden respecto al tiempo) es alojado en la variable *safe*. En el caso de que *make_verlet()* devuelva *false*, estos datos son usados para restablecer el estado previo del sistema donde no ha ocurrido un error. También, el valor de la

variable *verlet_distance* es multiplicado por *verlet_increase*. Luego, la simulación recomienza a partir de los datos restaurados.

Finalmente, se concluye esta sección con la implementación de *init_algorithm()* mostrada en la figura 4.28.

```
22      void init_algorithm()
23      {
24              safe=particle;
25              Timesafe=Time;
26              vnx=int(lx/verlet_grid);
27              vny=int(ly/verlet_grid);
28              if(vnx==0) vnx=1;
29              if(vny==0) vny=1;
30              dx=lx/vnx;
31              dy=ly/vny;
32              celllist.resize(vnx);
33              for(int i=0;i<vnx;i++) celllist[i].resize(vny);
34              make_verlet();
35      }
```

Figura 4.28: Séptimo fragmento de código de verlet.cc. *Tomado de [1].*

Dado que las listas de Verlet son necesarias previas al primer paso, la función *make_verlet()* es invocada en la función de inicialización. En este caso el valor de retorno de dicha función es ignorado, porque no se presentan solapamientos de partículas en la inicialización del sistema.

4.8 Conclusión

Se estudió en detalle la implementación serial de los algoritmos DEM (diseñada por Pöschel y Schwager, 2005) presentada en este capítulo, debido a que con su empleo fue posible realizar la simulación de un fenómeno de interés, a partir de la aplicación de la física de medios granulares. El fenómeno simulado es el de producción de nubes de polvo a baja velocidad relativa, como producto de la aceleración inducida por un sismo generado por una colisión. Para ello, fue necesario introducir leves modificaciones en los algoritmos explicados a lo largo del capítulo. El estudio de este fenómeno consistió en realizar una experiencia virtual, en la cual se utilizó una caja virtual 2D. La misma se realizó con una caja de 60 cm de largo conteniendo 21.165 partículas de 0,4 mm a 0,6 mm de radio. En la figura 4.29 se presentan imágenes del desarrollo de la simulación a lo largo del tiempo para el experimento. La simulación de unos pocos segundos insumió cinco días de ejecución.

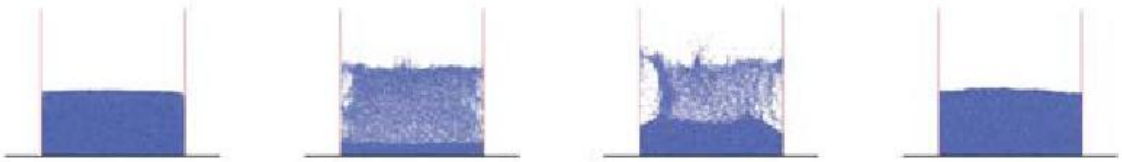

Figura 4.29: Experiencia numérica con 21.165 partículas de 0,4-0,6 mm de radio.
Tomado de [16].

Cabe destacar que el programa presentado en este capítulo es considerado una excelente introducción a varios de los conceptos y algoritmos empleados para la simulación del Método de los Elementos Discretos (DEM). Sin embargo, este programa no es construido utilizando técnicas de procesamiento paralelo, para el cálculo de las fuerzas de interacción. Esto hace que no sea posible modelar un sistema con una gran cantidad de partículas, ni realizar cálculos de interacciones de largo alcance en un tiempo razonable. Si bien se optimiza el cálculo de la fuerza de corto alcance, a partir del método de las listas de Verlet, esto no es suficiente para lograr mayores tiempos de simulación en sistemas de mediano o gran porte. Además, la representación de las magnitudes vectoriales de las partículas, como el radio, velocidad, entre otras, es en dos dimensiones. Dificultando visualizar los resultados de las simulaciones, alejado de la realidad que se pretende modelar.

CAPÍTULO 5
ESyS-Particle

5.1 Introducción

ESyS-Particle es un paquete de software para modelados numéricos basados en partículas. El software implementa el Método de los Elementos Discretos (DEM), una técnica ampliamente utilizada para el modelado de procesos que involucran grandes deformaciones, flujos granulares y/o fragmentación. *ESyS-Particle* esta diseñado para ejecutar sobre supercomputadoras paralelas, clusters o computadores con varios núcleos corriendo Linux como sistema operativo base. El motor de simulación (escrito en C++) implementa una descomposición espacial del dominio mediante el uso de *Message Passing Interface* (MPI). Una API en Python provee flexibilidad en el diseño de modelos numéricos, en la especificación de los parámetros del modelo, la lógica de las interacciones y el análisis de los datos de la simulación.

Las características relevantes de la herramienta son:

- Motor de simulación paralelo basado en MPI.
- API en Python para la preparación y ejecución de simulaciones.
- Preparación mediante *scripts* de modelos geométricos.
- Partículas esféricas rotacionales y no rotacionales.
- Mallas triangulares para especificación de condiciones de borde y pared.
- Visualización de partículas utilizando *POV-Ray* y *VTK*.
- Variedad de interacciones partícula-partícula y partícula-pared:
    - Repulsión elástica lineal entre partículas no enlazadas en contacto.
    - Repulsión elástica lineal entre pares de partículas enlazadas.
    - Interacciones fricciónales tanto rotacionales como no rotacionales entre partículas no enlazadas.
    - Enlaces rotacionales implementando torsión y coeficientes de rigidez.

ESyS-Particle ha sido desarrollado dentro del *Earth Systems Science Computational Centre (ESSCC)* en la Universidad de Queensland, Brisbane, Australia desde 1994. El software fue denominado en sus comienzos como *Lattice Solid Model*, y más tarde se lo llamó *LSMEarth* antes de que Australian Computational Earth Systems Simulator *(ACcESS)* comenzara el financiamiento del desarrollo del software en el año 2002. Desde 2002 al 2007 y gracias al financiamiento económico el desarrollo ha avanzado rápidamente hasta convertirse en un software DEM de uso comercial, distribuido gratuitamente bajo la licencia de software libre versión 3.0. Al día de hoy el desarrollo de *ESyS-Particle* continúa siendo financiado por el gobierno australiano, contando con la colaboración de desarrolladores de la universidad RWTH en Aachen, Alemania.

En las siguientes secciones se hablará de cómo *ESyS-Particle* resuelve las etapas de pre-procesamiento y post-procesamiento de datos, la instalación de la herramienta y de sus dependencias, las consideraciones para la construcción de una simulación, el uso de la técnica de división del dominio para la ejecución en paralelo, y por último se describirá como se extendió el software para incluir un nuevo tipo de interacción entre partículas. Los conceptos en este capítulo presentados sobre la herramienta *ESyS-Particle* se basan en [11, 13, 14].

5.2 Diseño de la aplicación

ESyS-Particle es una implementación paralela del Método de los Elementos Discretos (DEM) que, se caracteriza por ser *open source*. Esta disponible gratuitamente[1], puede ser modificada y extendida. Se basa en una arquitectura de memoria distribuida usando MPI. Corre desde simples PC de escritorio hasta grandes clusters, bajo el sistema operativo Linux. Mientas que el tamaño del problema sea escalado adecuadamente con la cantidad de procesadores, el beneficio de agregar procesadores es casi lineal. Soporta varios millones de partículas en las simulaciones, en pruebas realizadas por los desarrolladores de la herramienta, llegaron a utilizar más de 10 millones de partículas.

Esta diseñado para trabajar con otros programas individuales, que por separado se encargan del pre-procesamiento de los datos que son la entrada de la simulación y post-procesamiento de los resultados de la misma. Esto posibilita que sea más fácil el desarrollo y mantenimiento de las distintas herramientas, en lugar de un paquete que incluya todo. Las funcionalidades de *ESyS-Particle* se manejan mediante *scripts* en Python y no posee una GUI interactiva. El uso de Python como lenguaje de *scripting* proporciona una programación orientada a objetos, es fácil de utilizar, posee una extensa librería estándar, y además ya existe una implementación de la interfaz entre Python y el motor en C++ de *ESyS-Particle* (mediante la librería *Boost-Python*).

ESyS-Particle esta compilado como una librería compartida y es utilizada como un módulo de Python, los *scripts* importan las librerías como mecanismo para hacer accesibles las funciones exportadas por *ESyS-Particle*. Las funciones de los *scripts* llaman a las funciones de C++ para inicializar el modelo, preparar las interacciones, mover los objetos, salvar datos, etc.

[1] Se puede obtener en: http://launchpad.net/esys-particle/2.0/esys-particle.v2.0.stable/+download/ESyS-Particle-2.0.tar.gz.

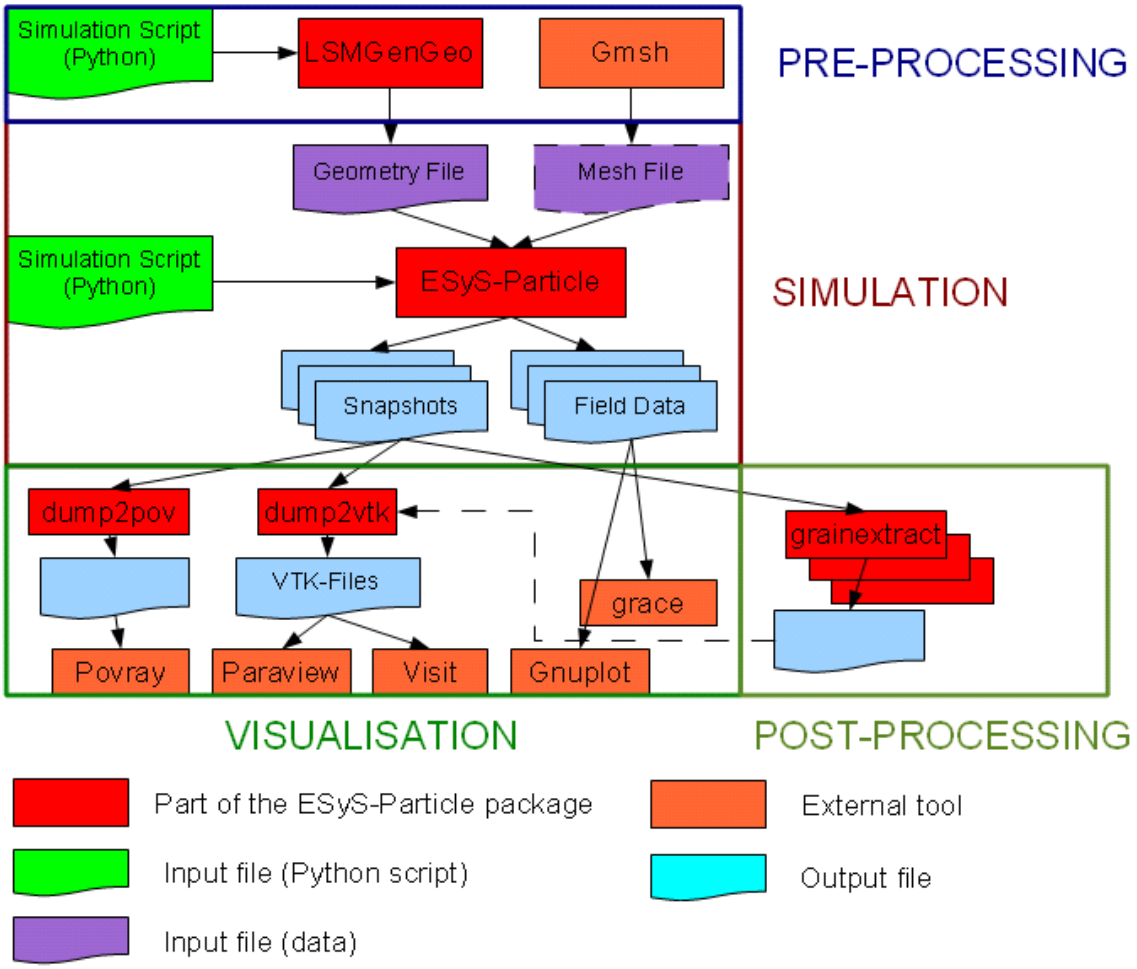

Figura 5.1: Procesamiento de la entrada y salida de la simulación. *Tomada de [13].*

Para realizar el pre-procesamiento se dispone de dos aplicaciones (ver figura 5.1), las cuales permiten generar grupos de partículas que serán la entrada de la simulación. La aplicación *LSMGenGeo* permite la generación de paquetes de partículas, utilizada dentro de los *scripts* en Python, fácilmente extensible y viene incluida como parte de la distribución de *ESyS-Particle*. *Gmsh* es una herramienta que permite la construcción de figuras más complejas mediante la utilización de mallas triangulares, se caracteriza por ser *open source* y es parte de algunas distribuciones de Linux (Debian, Ubuntu, entre otros). Además, los archivos de salida generados por *Gmsh* deben ser convertidos al formato de *ESyS-Particle*. Notar que *Gmsh* no se distribuye junto con *ESyS-Particle*.

En la etapa de post-procesamiento se cuenta con dos mecanismos para extraer datos, los *Snapshots* y los *FieldSavers*. Los *Snapshots* permiten tomar imágenes a modo de foto cada cierto período de tiempo elegido arbitrariamente, obteniendo así una salida gráfica mediante archivos de extensión *png* durante la ejecución. Los *FielSavers* proveen un mecanismo para, selectivamente salvar ciertos datos en archivos denominados *Field Data* (como posiciones, fuerzas actuantes, energía en los enlaces de partículas, cantidad de enlaces rotos, etc.) a disco para su posterior procesamiento. En la figura 5.1 se muestra la correspondencia entre los mecanismos de salida de datos soportados por *ESyS-Particle* y las distintas aplicaciones para la visualización 3D.

5.3 Instalación

En esta sección, se especifican los pasos que se deben seguir para la instalación de *ESyS-Particle*, en una arquitectura de NFS *(Network File System)*. La misma posibilita que distintos sistemas pertenecientes a una misma red accedan a ficheros remotos como si fueran locales. El sistema NFS está dividido al menos en dos componentes principales: un servidor y varios clientes, que conforman un cluster. Estos clientes pueden acceder de forma remota a los datos que se encuentran en el servidor, como se muestra en la figura 5.2.

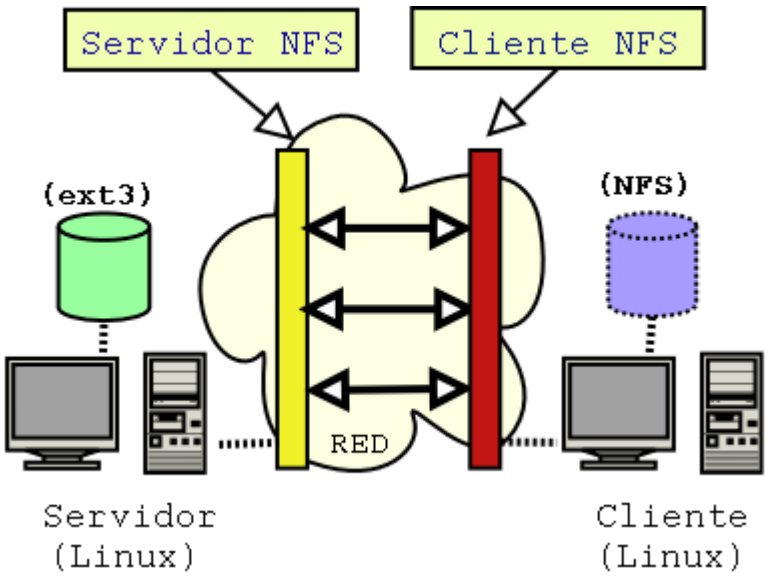

Figura 5.2: Diagrama de la arquitectura NFS.

Es importante que los pasos de instalación que se enumeran a continuación, se realicen en el orden en que aparecen para una correcta instalación. Los pasos del 1 al 6 inclusive deben ser ejecutados en cada máquina de cluster, del paso 7 en adelante solo se deben correr en el nodo maestro (o servidor) del sistema NFS.

1. Instalar *Centos* 5.3 con las últimas actualizaciones.

2. Instalar las *Development Libraries* y *Development Tools*: (como superusuario)
   a. Clic en "Applications -> Add/Remove Software" para invocar el *Package Manager*.
   b. Clic en la pestaña "Browse".
   c. Clic en "Development" en el panel de la izquierda.
   d. Marcar "Development Libraries" y "Development Tools" en el panel derecho.
   e. Clic en "Apply".

3. Esto instala los siguientes paquetes requeridos por *ESyS-Particle*:
   - python-2.4.3-24
   - python-devel-2.4.3-24
   - subversion-1.4.2-4
   - autoconf-2.59-12
   - automake-1.9.6-2.1

- libtool-1.5.22-6.1
- gcc-4.1.2-44
- boost-1.33.1-10
- boost-devel-1.33.1-10

4. Instalar *OpenMPI*:
   a. Clic en la pestaña "List" en el *Package Manager*.
   b. Instalar el paquete openmpi-devel-1.2.7-6 que aparece en la lista.

5. Esto también instala:
   - openmpi-1.2.7-6
   - openmpi-libs-1.2.7-6
   - mpi-selector-1.0.1-1

6. Establecer a *OpenMPI* como la implementación MPI predeterminada:
   a. Invocar "mpi-selector-menu" en una terminal.
   b. Siguiendo las instrucciones que aparecen en pantalla ingresar los valores "1s" y luego "1u".

7. Descargar e instalar (como súper usuario) los paquetes *RPM* cppunit-1.12.0-3 y cppunit-devel-1.12.0-3. Esta instalación de *CentOS* esta basada en *Red Hat Enterprise Linux 5*, por lo tanto hay que descargar las versiones "el5". (http://dag.wieers.com hostea un repositorio *RPM*.) Instalar los paquetes invocando "rpm –i <paquete.rpm>" en una terminal.

8. Descargar, compilar e instalar la ultima versión del código fuente de *POV-Ray* (http://www.povray.org/download/):
   a. ./configure --prefix=/path/to/install COMPILED_BY="your name <email@address>"
   b. make
   c. make install
   d. Agregar a la variable $PATH la ruta /path/to/install/bin

9. Descargar, compilar e instalar la ultima versión del código fuente de *cmake* (http://www.cmake.org/):
   a. ./bootstrap --prefix=/path/to/install
   b. make
   c. make install
   d. Agregar a la variable $PATH la ruta /path/to/install/bin

10. Descargar, compilar e instalar la ultima versión del código fuente de *VTK* (http://www.vtk.org/):
    a. Descomprimir VTK.
    b. mkdir /path/to/install
    c. cd /path/to/install
    d. ccmake ../path/to/VTK/unzip/files
    e. En la nueva interfaz interactiva que aparece presionar “c” para realizar una configuración inicial.
    f. Ajustar las opciones de instalación:
       i. CMAKE_INSTALL_PREFIX = /path/to/install

ii. VTK_USE_N_WAY_ARRAYS = ON
iii. VTK_USE_RENDERING = OFF

g. Volver a presionar “c” para regenerar la configuración:
   i. Si llega a aparecer algún error, volver al paso f para ajustar otras opciones y continuar.
h. Una vez obtenida la configuración deseada, presionar “g”.
i. make
j. Agregar a la variable $PATH la ruta /path/to/install/bin

11. Descargar la ultima versión de *ESyS-Particle*, en la terminal ejecutar: "svn checkout https://svn.esscc.uq.edu.au/svn/esys3/lsm/trunk esys-particle"

12. Dentro de la carpeta recién creada esys-particle editar el archivo configure.ac y reemplazar la línea "AX_BOOST_BASE([1.34.1])" por "AX_BOOST_BASE([1.33.1])".

13. Ahora se está en condiciones de instalar *ESyS-Particle* en el sistema
    a. ./autogen.sh
    b. ./configure --prefix=/path/to/install CC=mpicc CXX=mpic++ --without-epydoc
    c. make -j X (donde X es el número de núcleos con el que se desea compilar, esto es solo a efectos de acelerar el proceso de compilación)
    d. make install

14. Actualizar las siguientes variables de entorno modificando el archivo .bashrc en el home del usuario:
    a. Agregar la ruta a la carpeta bin de la instalación de *ESyS-Particle* a $PATH
    b. Agregar la ruta a la carpeta lib de la instalación de *ESyS-Particle* a $LD_LIBRARY_PATH.
    c. Agregar la ruta a la carpeta /lib/python2.4/site-packages de la instalación de *ESyS-Particle* a $PYTHONPATH.

Si por algún motivo, se necesita recompilar la herramienta, entonces se debe realizar de la siguiente manera:

1. make uninstall
2. make distclean
3. ./autogen.sh
4. ./configure CC=mpicc CXX=mpic++ --without-epydoc
5. make -j X (donde X es el número de núcleos en la máquina)
6. make install (como súper usuario)

5.4 Componentes de la simulación

Para construir una simulación utilizando *ESyS-Particle* se deben llevar a cabo los pasos que se detallan a continuación:

1. realizar la configuración inicial de partículas
2. establecer las condiciones de borde y las interacciones de partículas
3. ejecución de la simulación
4. visualización de los resultados

Para realizar el primer paso, *ESyS-Particle* provee un mecanismo simple para la construcción de un bloque de partículas con posiciones y radios aleatorios. Utiliza un algoritmo iterativo para insertar partículas dentro de un volumen determinado. El algoritmo puede ser visualizado en la figura 5.3 y se explica a continuación:

1. Ubicar un número aleatorio de partículas dentro del volumen asegurando que no se solapen entre sí.
2. Identificar tres partículas adyacentes
   a) computar el baricentro del tetraedro definido por las tres partículas
   b) computar el radio de una partícula que sea tangencial a las otras tres
   c) si el radio está dentro del rango especificado, y la partícula está completamente dentro del volumen prescrito, se inserta la partícula
3. Repetir el paso dos hasta que el número de inserciones fallidas alcance un tope especificado.

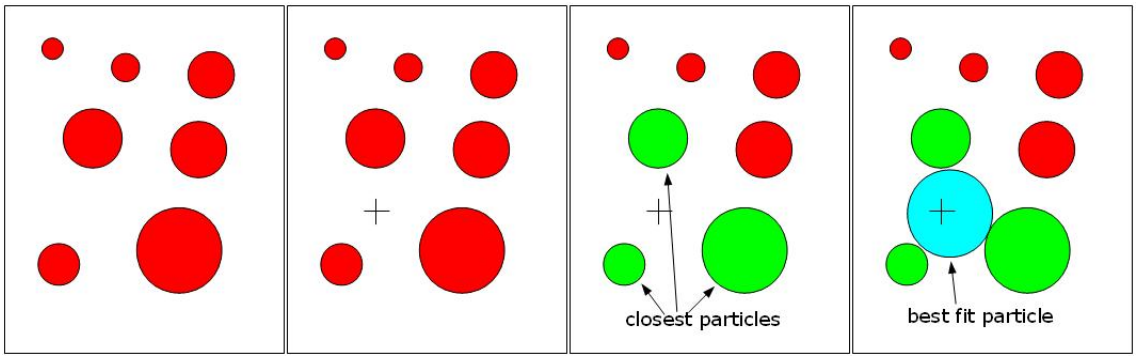

Figura 5.3: Algoritmo para la construcción de un bloque de partículas. *Tomada de [13].*

En el paso dos, lo primero que se debe hacer es crear un objeto de simulación llamado *LsmMpi*. Este objeto provee medios para definir y correr una simulación, y puede ser pensado como un contenedor al cual se le agregan componentes de simulación tales como una lista de partículas, paredes, diferentes tipos de interacciones y salida de datos.

Las partículas que provee la herramienta son de forma esférica. Existen dos tipos de partículas esféricas: rotacionales y no rotacionales. Una partícula queda definida una vez que se especifica el radio, posición del centro de masa y la masa. Se utilizan listas de vecinos y distancias de Verlet para la detección de interacciones y contactos entre partículas.

ESyS-Particle provee interacciones tanto con enlaces, como sin enlaces entre esferas que pueden ser tanto rotacionales como no rotacionales. A su vez, permite definir contactos físicos debidos a la fricción o a las interacciones elástico lineal. También logra representar el modelo continuo elástico lineal como interacciones elástico lineal entre partículas enlazadas.

Observar que en el caso de enlaces no rotacionales participan tanto fuerzas de compresión como de tensión. En enlaces rotacionales intervienen además fuerzas como torsión, flexión y ruptura. En la figura 5.4 se muestran los distintos tipos de fuerzas involucradas en las interacciones partícula-partícula.

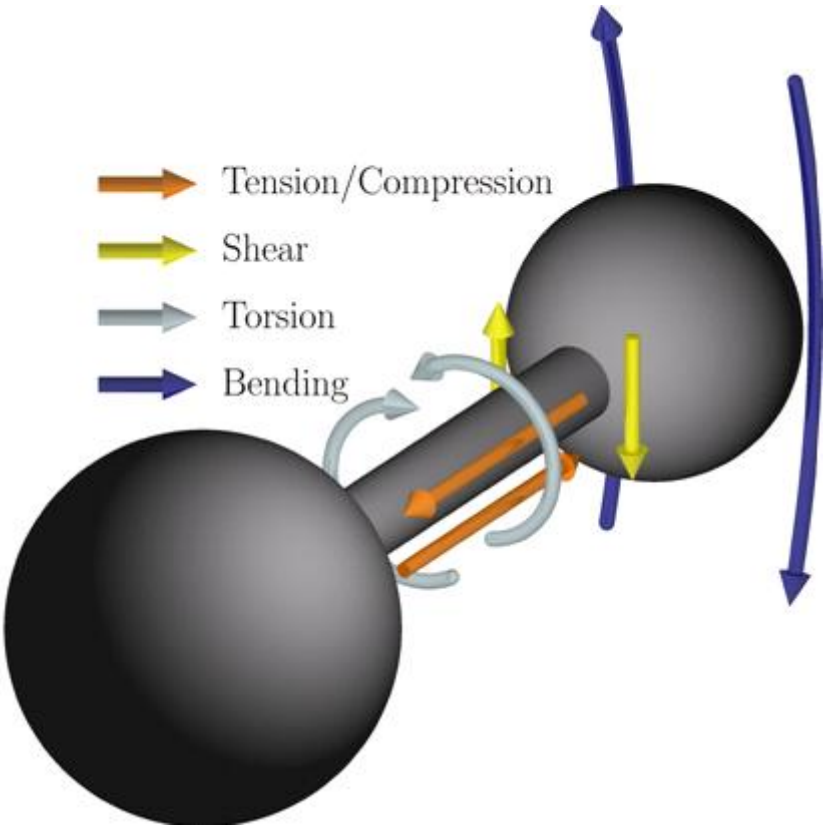

Figura 5.4 Tipos de fuerzas involucrados en una interacción partícula-partícula.
Tomada de [13].

A través, de la representación de enlaces con posterior rompimiento progresivo, es que se logra simular el fenómeno de fractura. En la figura 5.5 se observa la simulación de este fenómeno usando *ESyS-Particle*.

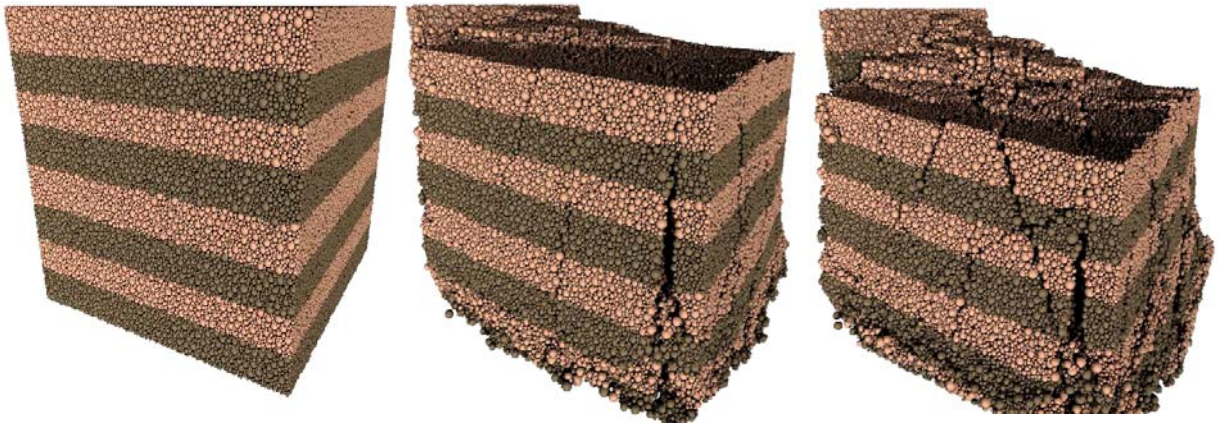

Figura 5.5: Simulación del fenómeno de fractura. *Tomada de [13].*

Las partículas no sólo interactúan con otras partículas, sino que también lo hacen con objetos fijos incluidos en el modelo. Estos objetos fijos no están sujetos a las leyes del movimiento, sino que se rigen por las condiciones de borde. Los objetos fijos disponibles son:

- Muro: un plano infinito determinado por una posición y un vector normal
- Malla lineal: un segmento de malla lineal puede ser utilizado para representar una superficie lineal.
- Malla triangular: puede ser utilizada para la representación de superficies 3D.

La herramienta provee una clase llamada *Runnable*, que le permite a los usuarios programar mediante *scripts* las tareas que se van a realizar antes y después de cada paso en la simulación. Entre las tareas que se pueden realizar se encuentran el control de muros (cambios en los desplazamientos, en las fuerzas, etc.), salida de datos a disco y movimientos no inerciales de partículas. También se dispone de elementos llamados *FieldSavers*, los cuales proveen mecanismos avanzados para el almacenamiento de información, tanto escalar como vectorial en intervalos regulares. Los *FieldSavers* se clasifican en dos categorías, los de partículas y los de muro. Los *FieldSavers* de partículas permiten almacenar datos como la energía cinética y los vectores tanto de desplazamiento como de velocidad. Los *FieldSavers* de muros permiten guardar las fuerzas aplicadas sobre el muro y los desplazamientos que experimenta.

Para realizar el paso 3, que hace referencia a la ejecución de la simulación, se debe ejecutar el comando *mpirun* incluyendo la definición del numero de procesos y el archivo que contiene las direcciones *ip* de los distintos nodos del cluster.

Por último, se dispone de dos librerías que son las encargadas de generar la salida para la visualización de la simulación: *VTK* y *POV-Ray*. *VTK* permite la creación de archivos con la salida de la simulación en formatos especiales que, sirven de entrada para paquetes de visualización como Paraview, Visit, entre otros. Por otro lado, *POV-Ray* genera la salida en su propio formato, y ofrece la interpretación de los archivos para una visualización de alta calidad de la simulación.

5.5 Paralelismo

ESyS-Particle utiliza la técnica paralela de descomposición espacial del dominio. Lo que básicamente significa que el dominio entero de simulación es divido en subdominios más pequeños, cada uno de ellos asignados a un procesador diferente.

Se utiliza una variable denominada *mpiDimList* para especificar la manera en la cual, el dominio de simulación será divido entre los procesadores. El número y tamaño de cada subdominio esta determinado por *mpiDimList*=[*x*,*y*,*z*]. Por ejemplo, si se toma *mpiDimList*=[3,2,1], entonces, el dominio de simulación va a ser dividido en tres partes en el eje *x* y dos partes en el eje *y*, obteniendo así un total de seis subdominios.

Los cálculos de las fuerzas y desplazamientos de las partículas en un determinado subdominio son asignados a un único procesador. La variable *numWorkerProcesses* especifica la cantidad de procesos MPI utilizados para los cálculos que son requeridos. Si se tienen seis subdominios, es necesario asignar *numWorkerProcesses* = 6 y cuando se ejecuta la simulación se debe informarle a *mpirun* que se desea utilizar un proceso maestro y seis procesos esclavos: *mpirun -np 7*.

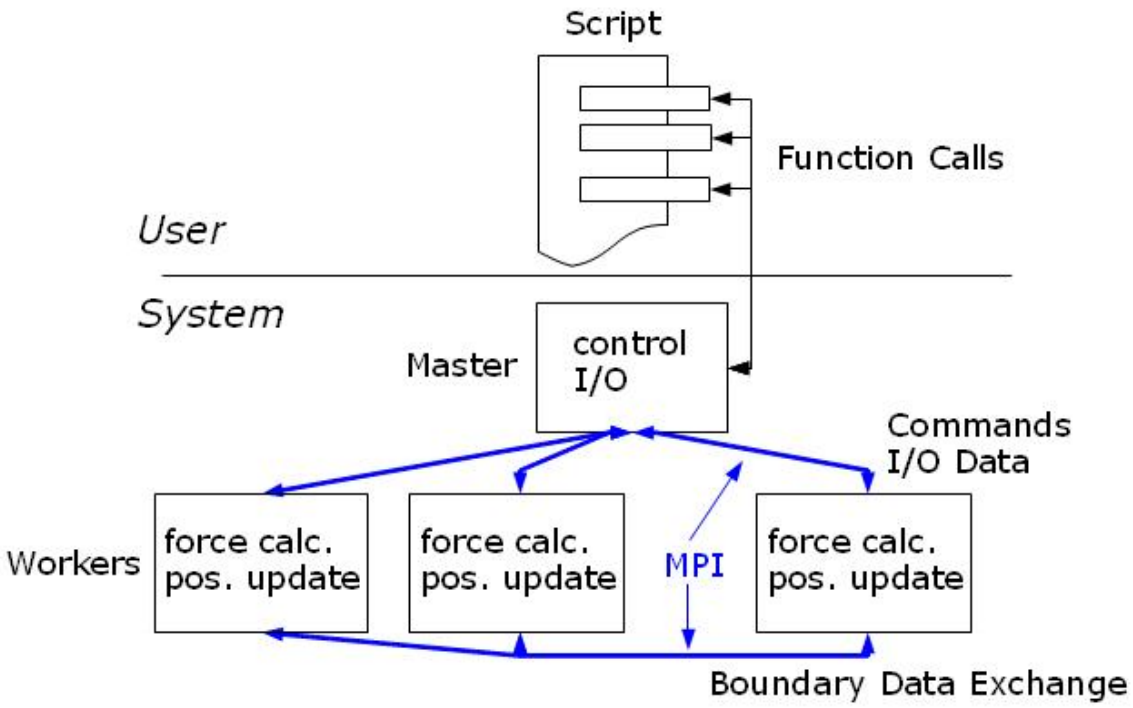

Figura 5.6 Esquema de funcionamiento de ESyS-Particle. *Tomada de [13].*

Cuando un dominio es subdivido de esta manera, los procesos esclavos (quienes realizan los cálculos) necesitan comunicarse entre ellos para determinar las posiciones de partículas cerca de los bordes de los subdominios, y también para informarle a otro procesador cuando una partícula se mueve de un subdominio a otro. Este esquema de funcionamiento es ilustrado en la figura 5.6. En general, en simulaciones paralelas, la comunicación es muy costosa en términos de tiempos de cómputo y debería ser evitada cuando sea posible. La descomposición en subdominios espaciales también tiene problemas, particularmente para simulaciones que involucran grandes desplazamientos de partículas. El tamaño y distribución de un subdominio no cambia durante una

simulación pero las partículas pueden comenzar en una pequeña porción del dominio y luego moverse por grandes distancias hacia otras partes del dominio. En tales casos se debe tener cuidado en como uno subdivide el dominio, sino uno se esta arriesgando a tener un gran número de procesadores sin cálculos para realizar (por que no hay partículas en su subdominio por grandes periodos de tiempo). Aún cuando un subdominio no tiene partículas, el proceso esclavo todavía esta involucrado en comunicaciones con su maestro, incrementando así el tiempo total de ejecución de la simulación.

Cuando uno desea correr una simulación en paralelo, dos preguntas tienen que ser consideradas:

1. ¿Cómo puedo subdividir el dominio para que, en promedio, todos los subdominios contengan aproximadamente el mismo número de partículas durante la simulación?
2. ¿Cada subdominio tiene suficiente número de partículas?

La segunda pregunta es probablemente la más importante. Si cada subdominio contiene solo unas pocas partículas, entonces el tiempo para computar las fuerzas y desplazamientos va a ser muy pequeño pero el tiempo de comunicación entre los procesos esclavos y el maestro va a ser grande. Dado que la comunicación es mucho más lenta que los cálculos matemáticos, incrementar el número de procesadores esclavos solo va a incrementar el tiempo total de ejecución, no lo va a disminuir como uno podría esperar. Es aconsejable asegurarse que cada procesador contenga un número razonable de partículas, una situación aceptable es cuando cada procesador es responsable de por lo menos 5.000 partículas.

Para responder la primera pregunta también hay que estudiar cada caso en particular. Si se considera el caso de un cubo de partículas no enlazadas que se desmorona por efecto de la gravedad, las partículas se mueven predominantemente de manera vertical (eje *y*) con rotaciones laterales al centro de masa del cubo. Si se subdivide el dominio en la dirección *y* (*mpiDimList*=[1,2,1]), entonces, todas las partículas estarían en el subdominio superior al comienzo de la simulación y terminarían en el subdominio inferior al finalizar la simulación. En promedio, solo un procesador estaría haciendo los cálculos de una determinada etapa de la simulación. Agregándole a esto los costos extra de comunicación, sería de esperar que los tiempos de ejecución incrementen.

Sin embargo, si se subdivide el dominio por la dirección *x* o *z* y no en el sentido del eje *y* (por ejemplo: *mpiDimList* = [2,1,1]), entonces, en promedio, cada subdominio va a contener el mismo número de partículas. En este caso se podría ver una disminución del tiempo de ejecución. Sin embargo aún va a haber esfuerzo de comunicación por las partículas que pasan de un procesador a otro dado que el cubo rota sobre su centro de masa.

5.6 Esferas viscoelásticas

Para modelar esferas viscoelásticas fue necesario la incorporación de dos nuevas interacciones al producto *ESyS-Particle*. En una de las interacciones se tuvo en cuenta la acción de la fricción y en la otra se omite. Agregar un nuevo tipo de interacción entre partículas en *ESyS-Particle* es relativamente sencillo. Sin embargo, dada la arquitectura paralela maestro-esclavo y la interfaz Python, es necesario agregar pequeños trozos de código a lo largo de todo el código fuente. Existen cuatro tipos de interacciones en *ESyS-Particle* que difieren en la forma en que son implementados:

- Interacciones no enlazadas entre pares de partículas.
- Interacciones enlazadas entre pares de partículas.
- Interacciones entre partículas y muros o mallas.
- Campos de fuerza tales como la gravedad, que son implementados como interacciones de una sola partícula.

Las interacciones no enlazadas son aquellas generadas dinámicamente para partículas que entran en contacto durante la simulación, sin depender de enlaces previos. En esta sección se muestra como crear una interacción no enlazada, basado en [12], necesaria para la representación del modelo viscoelástico con fricción.

5.6.1 La interacción y sus parámetros

Para implementar nuevas interacciones, se necesita de un archivo cabecera y un archivo de implementación, en nuestro caso son:

- HertzianViscoElasticFrictionInteraction.h
- HertzianViscoElasticFrictionInteraction.cpp

Estos archivos deben estar alojados en el subdirectorio "*Model*" de la estructura de archivos de *ESyS-Particle*. Para incluir estos archivos en el proceso de compilado de *ESyS-Particle*, hay que agregar tanto el .h como el .cpp al archivo *Makefile.am* que se encuentra en el mismo subdirectorio.

Para la implementación de un nuevo tipo de interacción, se necesitan dos clases, una para representar los parámetros que describen las interacciones (clase *Interaction Group parameters*) y una clase encargada del manejo de la interacción (clase *Interaction*).

Toda clase *Interaction Group parameters* (IGP) hereda de la clase *AIGParam*. Entonces, el archivo en el que se define una nueva clase IGP tiene que incluir a "Model/IGParam.h". Por lo tanto, la clase necesita declarar las siguientes funciones y variables:

- Un constructor sin parámetros.
- Un constructor que reciba como parámetro el nombre del grupo de interacción y todos los parámetros propios de la interacción. El tipo del nombre es *STL-string*.
- Una función *getTypeString()* que retorne el nombre de la interacción como un *STL-string*.

- Los atributos necesarios para la interacción.

En general, los parámetros de la interacción son implementados como atributos públicos, lo cual no es la forma correcta de hacerlo desde el punto de vista de los principios de orientación a objetos, pero ahorra muchos llamados a funciones disminuyendo el *overhead* de acceso a los mismos.

En nuestro caso la clase IGP necesita de seis atributos, definidos en el archivo HertzianViscoElasticFrictionInteraction.h, mostrado en la figura 5.7.

```
class CHertzianViscoElasticFrictionIGP : public AIGParam
{
public:
  CHertzianViscoElasticFrictionIGP();

  CHertzianViscoElasticFrictionIGP(const std::string &name, double A, double E,
double nu, double fricCoef, double shearK, double dT);

  virtual std::string getTypeString() const {return "HertzianViscoElasticFriction";}

  void setTimeStepSize(double dt);

  double m_A;  // Dissipative constant
  double m_E;  // Young's modulus
  double m_nu; // Poisson ratio
  double mu;   // Friction coefficient
  double k_s;  // Shear coefficient
  double dt;
};
```

Figura 5.7: Clase CHertzianViscoElasticFrictionInteractionIGP.

La implementación de los constructores se muestra en la figura 5.8 (ver el archivo HertzianViscoElasticFrictionInteraction.cpp).

```
CHertzianViscoElasticFrictionIGP::CHertzianViscoElasticFrictionIGP()   :   AIGParam(),
m_A(0.0), m_E(0.0), m_nu(0.0), mu(0.0), k_s(0.0), dt(0.0)
{
}

CHertzianViscoElasticFrictionIGP::CHertzianViscoElasticFrictionIGP(const   std::string
&name, double A, double E, double nu, double fricCoef, double shearK, double dT)
  : AIGParam(name),
       m_A(A),
       m_E(E),
       m_nu(nu),
    mu(fricCoef),
    k_s(shearK),
    dt(dT)
{
}
```

Figura 5.8: Constructores de CHertzianViscoElasticFrictionInteractionIGP

Como se comentó anteriormente, la otra clase a implementar (ver figura 5.9) es la encargada de las interacciones, la cual es responsable de:

- el calculo de las fuerzas entre dos partículas interactuando entre si,
- aplicar las fuerzas calculadas sobre estas partículas, y
- proveer a las funciones *FieldSaver* datos relevantes de la interacción tales como energía potencial, cantidad de interacciones, etc.

Toda clase de interacción contempla tanto los aspectos que tienen que ver con influencias externas, sobre una única partícula (como la gravedad), como las interacciones entre dos partículas. Toda aquella clase que implemente interacciones entre dos partículas, como la presentada aquí, heredan de la clase *APairInteraction*.

Una clase de interacción debe contener como mínimo, las siguientes funciones:

- Un constructor que recibe 2 punteros a partículas y una referencia a la clase IGP.
- Una función *calcForces()* que calcula las fuerzas de la interacción y las aplica a las partículas.
- Las funciones necesarias para salvar datos.

```
class CHertzianViscoElasticFrictionInteraction : public APairInteraction {
 public: // types
  typedef CHertzianViscoElasticFrictionIGP ParameterType;

  typedef double (CHertzianViscoElasticFrictionInteraction::* ScalarFieldFunction)()
                const;
  typedef std::pair<bool,double> (CHertzianViscoElasticFrictionInteraction::*
                CheckedScalarFieldFunction)() const;
  typedef Vec3 (CHertzianViscoElasticFrictionInteraction::* VectorFieldFunction)()
                const;

  static ScalarFieldFunction getScalarFieldFunction(const string&);
  static CheckedScalarFieldFunction getCheckedScalarFieldFunction(const string&);
  static VectorFieldFunction getVectorFieldFunction(const string&);

 protected:
  double m_A;     //!< Dissipative constant
  double m_E;     //!< Young's modulus
  double m_nu;    //!< Poisson ratio
  double m_r0;    //!< equilibrium distance
  double m_mu;    //!< coefficient of friction
  double m_ks;    //!< shear stiffness (Cundall)
  double m_dt;    //!< time step
  Vec3 m_Ffric;   //!< current frictional force
  Vec3 m_force_deficit; //!<difference between fric. Force & force necessary for slip
  Vec3 m_cpos; //!< contact position
  Vec3 m_normal_force; //!< current normal force
  double m_dn; //!< caching displacement for E_pot
  bool m_is_slipping; //!< static/dynamic status of the interaction
  bool m_is_touching; //!< contact status of the interaction
  double m_E_diss; //!< dissipated energy

 public:
  CHertzianViscoElasticFrictionInteraction();
  CHertzianViscoElasticFrictionInteraction(CParticle*,CParticle*);
  CHertzianViscoElasticFrictionInteraction(CParticle*,CParticle*,const
                CHertzianViscoElasticFrictionIGP&);
  virtual ~CHertzianViscoElasticFrictionInteraction();

  static string getType() {return "HertzianViscoElasticFriction";};

  virtual void calcForces();
  virtual bool isPersistent();
  void setTimeStepSize(double dt);

  std::pair<bool,double> getAbsFrictionalForce() const;
  std::pair<bool,double> getAbsFN() const;
  std::pair<bool,double> getAbsMuFN() const;
  std::pair<bool,double> getSlipVelocity() const;
  std::pair<bool,double> getNormalStress() const;
  std::pair<bool,double> getMaxFricStress() const;
  std::pair<bool,double> getAbsFrictionalStress() const;

  double getAbsForceDeficit() const;
  double getPotentialEnergy() const;
  double getSlipping()const;
  double getSticking()const;
  double getDissipatedEnergy() const;
  double Count() const;
  Vec3 getForce() const;
  Vec3 getNormalForce() const;
  virtual Vec3 getPos() const {return m_cpos;};
  std::pair<bool,double> getMuEff(const Vec3&,const Vec3&) const;
  std::pair<bool,double> getMuEffXY() const {return
                        getMuEff(Vec3(1.0,0.0,0.0),Vec3(0.0,1.0,0.0));};
  std::pair<bool,double> getMuEffXZ() const {return
                        getMuEff(Vec3(1.0,0.0,0.0),Vec3(0.0,0.0,1.0));};
  friend std::ostream& operator<<(std::ostream&,const
                        CHertzianViscoElasticFrictionInteraction&);
  friend class TML_PackedMessageInterface;
};
```

Figura 5.9: Clase CHertzianViscoElasticFrictionInteraction.

5.6.2 Cálculo de la fuerza

La fuerza de interacción para esferas elásticas fue inferida por Heinrich Hertz como una función de la deformación ξ y los parámetros del material Y (módulo de Young) y υ (radio de Poisson):

$$F_{el}^{n} = \frac{2Y\sqrt{R^{eff}}}{3(1-\nu^2)}\xi^{3/2}$$

Ecuación 5.1: Cálculo de la fuerza normal en interacciones elásticas.

A partir de la ecuación 5.2, se puede inferir R^{eff} como el radio efectivo de las esferas en colisión. Con R_i y R_j se denotan los radios de las partículas que intervienen en la interacción.

$$\frac{1}{R^{eff}} = \frac{1}{R_i} + \frac{1}{R_j}$$

Ecuación 5.2: Expresión de la que se deriva el radio efectivo.

En la ecuación 5.3 se presenta el resultado que fue generalizado para el contacto de partículas viscoelásticas. La constante disipativas A es una función de la viscosidad del material.

$$F^{n} = \frac{2Y\sqrt{R^{eff}}}{3(1-\nu^2)}\left(\xi^{3/2} + A\sqrt{\xi}\frac{d\xi}{dt}\right)$$

Ecuación 5.3: Cálculo de la fuerza normal en interacciones viscoelásticas.

Teniendo en cuenta que las esferas no se pueden solapar, pero si deformar, es que se introduce un cambio en la ecuación 5.4 de manera de evitar este fenómeno.

$$F^{n} = \max\left\{0, \left[\frac{2Y\sqrt{R^{eff}}}{3(1-\nu^2)}\left(\xi^{3/2} + A\sqrt{\xi}\frac{d\xi}{dt}\right)\right]\right\}$$

Ecuación 5.4: Cálculo de la fuerza normal que evita el solapamiento de las esferas.

La implementación del cálculo de la fuerza se muestra en la figura 5.10.

```
1   void CHertzianViscoElasticFrictionInteraction::calcForces(){
3   // calculate distance
4   Vec3 D=m_p1->getPos()-m_p2->getPos();
5   double dist=D*D;
6   double eq_dist=m_p1->getRad()+m_p2->getRad();
7   // check if there is contact
8   if(dist<(eq_dist*eq_dist))
9   { // contact -> calculate forces
10          //--- viscoelastic force ---
11          double R_ij=1.0/(1.0/m_p1->getRad()+1.0/m_p2->getRad());
12          dist=sqrt(dist);
13          m_dn=eq_dist-dist;
14          Vec3 dir=D.unit();
16          //Calculate d m_dn / dt
17          double ex=D.X()/dist;
18          double ey=D.Y()/dist;
19          double ez=D.Z()/dist;
20          double dvx=m_p1->getVel().X()-m_p2->getVel().X();
21          double dvy=m_p1->getVel().Y()-m_p2->getVel().Y();
22          double dvz=m_p1->getVel().Z()-m_p2->getVel().Z();
23
24          double m_dn_dot=-(ex*dvx+ey*dvy+ez*dvz);
25          double norm_m_normal_force=(2.0*m_E*sqrt(R_ij))/(3.0*(1.0-
26          m_nu*m_nu))*(pow(m_dn,1.5)+m_A*sqrt(m_dn)*m_dn_dot);
27          m_normal_force = norm_m_normal_force < 0 ? Vec3(0.0,0.0,0.0) :
                                dir*norm_m_normal_force;
28          Vec3 pos=m_p2->getPos()+(m_p2->getRad()/eq_dist)*D;
29          // apply viscoelastic force
30          m_p1->applyForce(m_normal_force,pos);
31          m_p2->applyForce(-1.0*m_normal_force,pos);
32
33          //--- frictional force ---
34          // particle movement since last timestep
35          const Vec3 d1=m_p1->getVel()*m_dt;
36          const Vec3 d2=m_p2->getVel()*m_dt;
37          Vec3 dFF=m_ks*(d2-d1);
38          // Compute tangential part by subtracting off normal component.
39          dFF -= ((dFF*D)/(D.norm2()))*D;
40          m_Ffric+=dFF;
41          const double FfricNorm = m_Ffric.norm();
42          const double forceNorm = m_normal_force.norm();
43          // decide static/dynamic
44          if (FfricNorm > forceNorm*m_mu)
45          { // tangential force greater than static friction -> dynamic
46                  m_Ffric=m_Ffric*((m_mu*forceNorm)/FfricNorm);
47                  m_force_deficit=Vec3(0.0,0.0,0.0);
48                  m_is_slipping=true;
49                  m_E_diss=m_mu*fabs(m_normal_force*(d2-d1)); // energy dissipated
50          }
51          else if (FfricNorm > 0.0)
52          { // static friction
53                  m_is_slipping=false;
54                  m_E_diss=0.0; // no energy dissipated
55          }
56          else
57          { // no frictional force -> force deficit=mu*F_n
58                  m_is_slipping=false;
59                  m_E_diss=0.0; // no energy dissipated
60          }
61          m_p1->applyForce(m_Ffric, pos);
62          m_p2->applyForce(-1.0*m_Ffric, pos);
63          m_cpos=pos;
64          m_is_touching=true;
65  }
66  else
67  { // no contact -> all forces are 0
68          m_Ffric=Vec3(0.0,0.0,0.0);
69          m_normal_force=Vec3(0.0,0.0,0.0);
70          m_is_slipping=false;
71          m_is_touching=false;
72          m_E_diss=0.0; // no energy dissipated
73  }
74  }
```

Figura 5.10: Función para el cálculo de fuerzas de la interacción.

Las líneas 3-6 determinan si la distancia actual entre las partículas es menor que la suma de sus respectivos radios, es decir, si las partículas están en contacto. Si están en contacto, lo cual se determina en la línea 6, la fuerza de interacción es calculada (líneas 25, 26 y 27 implementan la ecuación 1.2) y es aplicada a las partículas en las líneas 30 y 31. Luego, a partir de la línea 33 hasta la línea 64 se calcula la fuerza de fricción a partir del componente normal, aplicando el coeficiente de fricción estático o dinámico según corresponda. En cambio, si las partículas no se encuentran en contacto, el valor del componente normal, tangencial de la fuerza vale cero. Como consecuencia de ello, el componente disipativo de la energía también valdrá cero.

La interacción tiene dos magnitudes de interés que, pueden ser accedidos en las funciones de *FieldSaver* desde el *script* de simulación, la fuerza y la energía potencial. Para ello, dos funciones deben ser implementadas, *getForce()* que devuelve un vector Vec3, y *getPotencialEnergy()* que devuelve un escalar (de tipo *double*).

5.6.3 Modelo paralelo maestro-esclavo

Con el fin de crear una instancia de un grupo de interacción de un tipo determinado, una serie de funciones se necesitan en el maestro y los esclavos. En el maestro, las funciones se necesitan para armar un paquete con los parámetros del grupo de interacción y enviarlo a los esclavos. Por otro lado, las funciones en el esclavo son necesarias para recibir y desempaquetar los parámetros, y crear una instancia del grupo de interacción.

5.6.3.1 Funciones del maestro

Es necesario agregar en el archivo *LatticeMaster.h* (en el subdirectorio *Parallel*) la importación al archivo cabecera de la nueva interacción mediante *#include*, en nuestro caso sería: *#include "Model/HertzianViscoElasticFrictionInteraction.h"*. Además es necesario agregar las siguientes funciones, declaradas en *LatticeMaster.h* e implementadas en *LatticeMaster.cpp*.

- Una función para agregar el grupo de interacción para todas las partículas: *void addPairIG(const CHertzianElasticIGP&)*.
- Una función para agregar el grupo de interacción solo para las partículas con etiquetas definidas: *void addTaggedPairIG(const CHertzianElasticIGP&,int,int,int,int)*.

Para ser más preciso, estas funciones arman el paquete de parámetros, los envían a los esclavos y le indican a los esclavos que agreguen el grupo de interacción.

Estas dos funciones (ver figuras 5.11 y 5.12) consisten tanto, de la declaración de un objeto para la construcción de un mensaje, el agregado de los parámetros necesarios, como de un *broadcast* a todos los esclavos.

```
void CLatticeMaster::addPairIG(const CHertzianViscoElasticFrictionIGP &prms)
{
  IGPCommand igpCmd(getGlobalRankAndComm());
  igpCmd.appendIGP(prms);
  igpCmd.append(prms.m_A);
  igpCmd.append(prms.m_E);
  igpCmd.append(prms.m_nu);
  igpCmd.append(prms.mu);
  igpCmd.append(prms.k_s);
  igpCmd.append(prms.dt);
  igpCmd.broadcast();
}
```

Figura 5.11: Implementación de la función AddPairIG

```
void CLatticeMaster::addTaggedPairIG(const CHertzianViscoElasticFrictionIGP &prms,
                              int tag1, int mask1, int tag2 ,int mask2)
{
  TaggedIGPCommand igpCmd(getGlobalRankAndComm());
  igpCmd.appendIGP(prms);
  igpCmd.append(prms.m_A);
  igpCmd.append(prms.m_E);
  igpCmd.append(prms.m_nu);
  igpCmd.append(prms.mu);
  igpCmd.append(prms.k_s);
  igpCmd.append(prms.dt);
  igpCmd.append(tag1);
  igpCmd.append(mask1);
  igpCmd.append(tag2);
  igpCmd.append(mask2);

  igpCmd.broadcast();
}
```

Figura 5.12: Implementación de la función AddPairIG

5.6.3.2 Funciones de los esclavos

En los esclavos, se necesitan de funciones para recibir y desempaquetar los parámetros, y crear el grupo de interacción. Para este propósito, los siguientes agregados tienen que ser realizados en el archivo *SubLattice.hpp* (ver figura 5.13):

- Se debe incluir el archivo cabecera del nuevo tipo de interacción, *#include "Model/HertzianViscoElasticFrictionInteraction.h"*.
- Dentro de la función *doAddPIG()* añadir el código necesario para crear el grupo de interacción, es decir:
    - Adicionar un condicional para el nombre del nuevo tipo de interacción: *if(type=="HertzianViscoElasticFriction")*.
    - Extraer los parámetros del grupo de interacción del mensaje.
    - Si hay que crear un grupo de interacción etiquetado, también es necesario extraer los parámetros *tag* y *mask* desde el mensaje.

- Crear un "*pair interaction storage*" del grupo de interacción. Para elegir el tipo apropiado de la clase "*pair interaction storage*" (PIS), es necesario distinguir si la interacción contiene datos persistentes que necesiten ser intercambiados entre los diferentes procesos esclavos, si las partículas involucradas en la interacción se mueven a un subdominio diferente.

```
.....
} else if(type=="HertzianViscoElasticFriction") {
    CHertzianViscoElasticFrictionIGP hvefigp;
    hvefigp.m_A=param_buffer.pop_double();
    hvefigp.m_E=param_buffer.pop_double();
    hvefigp.m_nu=param_buffer.pop_double();
    hvefigp.mu=param_buffer.pop_double();
    hvefigp.k_s=param_buffer.pop_double();
    hvefigp.dt=param_buffer.pop_double();
    if(tagged){
      int tag1=param_buffer.pop_int();
      int mask1=param_buffer.pop_int();
      int tag2=param_buffer.pop_int();
      int mask2=param_buffer.pop_int();
      new_pis=new
ParallelInteractionStorage_NE_T<T,CHertzianViscoElasticFrictionInteraction>(m_ppa,hve
figp,tag1,mask1,tag2,mask2);
    }else{
      new_pis=new
ParallelInteractionStorage_NE<T,CHertzianViscoElasticFrictionInteraction>(m_ppa,hvefi
gp);
    }
    res=true;
.....
```

Figura 5.13: Fragmento de código de los esclavos

5.6.4 Extensiones de la interfaz python

De forma de ser capaz de usar un nuevo tipo de interacción desde un *script* de simulación, la interfaz Python para *ESyS-Particle* necesita dos extensiones, un contenedor (*wrapper*) Python para la clase IGP y otro contenedor Python para la invocación que instancia el grupo de interacción.

De forma de poder crear una instancia de la clase IGP, es necesario exponer la clase implementada en la sección 5.6.1 a la interfaz Python. Para ello, es necesario extender los archivos *InteractionParamsPy.h* e *InteractionParamsPy.cpp* de la siguiente manera:

- Se debe incluir el archivo cabecera del nuevo tipo de interacción, *#include "Model/HertzianViscoElasticFrictionInteraction.h"* en el archivo *InteractionParamsPy.h*.
- Añadir la definición de la clase contenedora para la clase IGP (en el ejemplo en *InteractionParamsPy.h*). La clase contenedora (ver figura 5.14) únicamente necesita tener, una función constructora que tome la misma cantidad de argumentos que la clase contenida.

- Agregar la implementación del constructor en el archivo *InteractionParamsPy.cpp* (ver figura 5.15), el cual en la mayoría de los casos, simplemente pasa los argumentos al constructor de la clase contenida.
- Nuevamente en *InteractionParamsPy.cpp*, se debe agregar la clase contenedora a la interfaz, la cual es exportada a Python mediante la librería *boost*. Las dos funciones mostradas en la figura 5.16 son los constructores, y la función *getName* que es heredada de la clase contenida.

```
class HertzianViscoElasticFrictionPrmsPy : public CHertzianViscoElasticFrictionIGP
    {
    public:
      HertzianViscoElasticFrictionPrmsPy(const
std::string&,double,double,double,double,double);
    };
```

Figura 5.14: Clase contenedora de la clase IGP.

```
HertzianViscoElasticFrictionPrmsPy::HertzianViscoElasticFrictionPrmsPy(const
std::string &name,
                                          double A,
                                          double E,
                                          double nu,
                                          double dynamicMu,
                                          double shearK)
      : CHertzianViscoElasticFrictionIGP(name, A, E, nu, dynamicMu, shearK, 0.0)
    {}
```

Figura 5.15: Implementación del constructor de la clase contenedora.

```
boost::python::class_<HertzianViscoElasticFrictionPrmsPy,boost::python::bases<Interac
tionPrmsPy> >("HertzianViscoElasticFrictionPrms","Parameters for Hertzian
viscoelastic contact interactions with friction.", boost::python::init<const
std::string &, double, double, double, double, double>(
          (
            arg("name"),
            arg("A"),
            arg("E"),
            arg("nu"),
            arg("dynamicMu"),
            arg("shearK")
          ),
          "@type name: string\n"
          "@kwarg name: Name assigned to this group of interactions.\n"
          "@type A: float\n"
          "@kwarg A: Damping constant used for force calculation.\n"
          "@type E: float\n"
          "@kwarg E: Young's modulus used for force calculation.\n"
          "@type nu: float\n"
          "@kwarg nu: poisson ratio used for force calculation.\n"
          "@type dynamicMu: float\n"
          "@kwarg dynamicMu: friction coefficient"
          " when contact is sliding.\n"
          "@type shearK: float\n"
          "@kwarg shearK: spring constant used when calculating linear"
          " viscoelastic shear force.\n"
        )
      )
      .def(
        "getName",
        &HertzianViscoElasticFrictionPrmsPy::getName,
        boost::python::return_value_policy<boost::python::copy_const_reference>()
      )
      ;
```

Figura 5.16: Implementación del constructor de la clase contenedora.

5.6.4.1 Exportando la función para crear el grupo de interacción

De forma de ser capaz de crear el grupo de interacción, es necesario agregar una función a la interfaz Python que llama a *LatticeMaster::addPairIG* con los parámetros apropiados. Para implementar esta función, la clase *LsmPmiPy* necesita ser extendida de la siguiente forma:

- Agregar una declaración adelantada de la clase *HertzianViscoElasticFrictionPrmsPy* en el archivo *LsmMpiPy.h*.
- Añadir en el archivo *LsmMpiPy.h* la declaración de la función *createHertzianViscoElasticFrictionIG*.
- Implementar la función anterior en el archivo *LsmMpiPy.cpp* (ver figura 5.17).
- Agregar la variante correspondiente de "*createInteractionGroup*" a la interfaz *boost* que se encuentra en la función *exportLsm()* en *LsmPmiPy.cpp* (ver figura 5.18).

```
void LsmMpiPy::createHertzianViscoElasticFrictionIG(
      const HertzianViscoElasticFrictionPrmsPy &prms
    )
    {
         HertzianViscoElasticFrictionPrmsPy p = prms;

      p.setTimeStepSize(getTimeStepSize());
      getNameTypeMap()[p.getName()] = p.getTypeString();
      getLatticeMaster().addPairIG(p);
    }
```

Figura 5.17: Implementación de la función createHertzianViscoElasticFrictionIG dentro del archivo LsmMpiPy.cpp.

```
  .def(
        "createInteractionGroup",
        &LsmMpiPy::createHertzianViscoElasticFrictionIG,
        (arg("prms"))
      )
```

Figura 5.18: Código para la definición del nuevo grupo de interacción que estará disponible para la interfaz python.

5.7 Simulaciones de medios granulares

En esta sección se mostrarán los resultados obtenidos con la aplicación *ESyS-Particle*. Para los sistemas con grandes cantidades de partículas (del orden de miles), se utilizó la infraestructura del cluster Ciencias. El mismo consta de un máximo de 8 nodos con 8 procesadores cada uno. En cada problema la cantidad de procesadores a utilizar depende de la cantidad de partículas intervinientes en la simulación. A partir de las experiencias realizadas por los desarrolladores de *ESyS-Particle* se recomienda asignarle a cada procesador más de 5.000 partículas. De otra forma, en lugar de ganar performance se estaría perdiendo debido al exceso de comunicaciones entre los procesadores involucrados.

El primer paso previo a simular fenómenos con grandes cantidades de partículas, consiste en realizar simples experiencias. Estas se realizan con pocas partículas, y cuyo resultado es conocido. El objetivo es, ajustar los distintos parámetros de las interacciones de interés, que participan en las simulaciones de fenómenos de medios granulares.

Los experimentos que se realizaron para el ajuste de parámetros fueron:

1) Rebote de una partícula que cae
2) Colisión de dos partículas a 45 grados
3) Colisión rasante de dos partículas
4) Simulación del Newton Cradle o balancín de Newton: las 4 bolitas colgadas de hilos que se pegan entre si.

Una vez que se obtiene el conjunto de valores apropiados para los parámetros de las interacciones, se procedió a realizar simulaciones relacionadas con los siguientes problemas:

1) Segregación por tamaño (efecto Nueces de Brasil) en diferentes condiciones de gravedad. Para eso se consideró una caja con partículas pequeñas y una grande cuyo fondo sube súbitamente repetidas veces.
2) Colisión de un proyectil contra un aglomerado de partículas. Un proyectil de 10 m contra un objeto formado por 90.000 partículas de entre 2,5 m y 12,5 m formando un cuerpo de 500,0 m de diámetro.

En la simulación del efecto Nueces de Brasil participaron 400 partículas de 0,25 m de radio pegadas a la base (ubicada en la parte inferior del dominio de simulación), 6.241 partículas de 0,25 m de radio y 1 partícula de 1,0 m de radio. La base se desplaza en la dirección del eje *y* positivo a una velocidad de 0,1 m/s durante 0,5 s. El movimiento ascendente de la base se repite cada 15 s. La simulación tuvo una duración de 5.000 s, mientras que el paso de integración elegido fue de $5x10^{-5}$ y el tiempo de ejecución fue de 14 días con un solo procesador. Se utilizó un solo procesador debido a las dimensiones del problema.

Al comienzo de este proyecto, el efecto Nueces de Brasil se había logrado simular con el código serial modificado de CGD (presentado en el capítulo 4). La ejecución tardaba aproximadamente un día por cada 100 segundos de simulación con un total de 1.800 partículas. Con *ESyS-Particle* se ha logrado una notoria mejora en los

tiempos de ejecución del fenómeno, ya que con cuatro veces más partículas que en un principio, por cada día de ejecución se logran cerca de 357 segundos de simulación. Es decir, se tiene tres veces más poder de cálculo con cuatro veces más cantidad de partículas.

En segundo lugar tenemos un nuevo caso de estudio, que consiste en la simulación de la colisión del proyectil contra un aglomerado de partículas. Se utilizaron 88.570 partículas de 2,5 m a 12,5 m de radio para formar el aglomerado y 1 partícula de 10,0 m de radio como proyectil impactando a 5.000 m/s. El tiempo total de simulación fue de 10 s, el paso de integración utilizado fue de 10^{-5} s, y el tiempo de ejecución requerido fue de 45 horas con 8 procesadores.

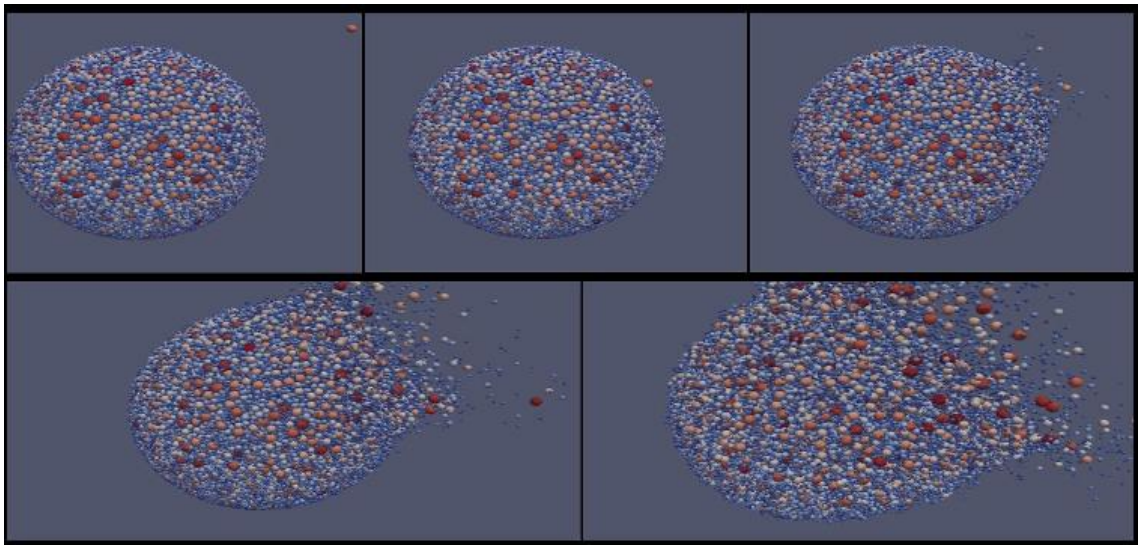

Figura 5.19: Colisión de un proyectil contra un aglomerado de partículas. *Tomada de [17].*

En la figura 5.19 se muestra el transcurso de la simulación, a partir de capturas de pantalla de distintos momentos de tiempo. Los colores indican los distintos tamaños de las partículas. Los tonos azules indican partículas de tamaño pequeño, mientras que cuanto más roja es la partícula más grande es su radio. El color naranja del proyectil que impacta contra el aglomerado se debe a que este posee 10,0 m de radio, siendo una de las partículas de mayor tamaño de la simulación.

5.8 Conclusión

ESyS-Particle implementa un motor potente para simulaciones de fenómenos de medios granulares. Esto se debe a que utiliza la técnica paralela de descomposición por dominios para el cálculo de la fuerza e implementa interacciones de naturaleza física de gran interés y aplicación para los fenómenos físicos de medios granulares.

Se pretende utilizar la herramienta para el estudio de dos fenómenos:

- Procesos de segregación de rocas por tamaño como producto de sismos producidos en colisiones de asteroides.
- Producción de nubes de polvo a baja velocidad relativa como producto de la aceleración inducida por un sismo generado por una colisión.

Estos fenómenos van a permitir una mayor comprensión de los procesos colisionales de asteroides, basados en la aplicación de la Física de Medios Granulares. Para la implementación de estos fenómenos es necesario la aplicación de técnicas DEM en condiciones de muy baja gravedad. Para ello se construyo una infraestructura a partir de la implantación de *ESyS-Particle* en el cluster de Ciencias. La cual es posible utilizar como base, para la generación de simulaciones de sistemas de partículas de gran tamaño. Este ámbito de trabajo también permite la incorporación de nuevas interacciones entre partículas como ser, la fuerza de atracción mutua entre partículas, con un algoritmo de árbol o de potencial gravitatorio medio.

Los avances obtenidos a partir de la incorporación de esta herramienta son:

1) Las simulaciones son 3D y no solamente 2D como en un principio.
2) En un comienzo se trabajaba con un máximo de 2.000 partículas, ahora se tienen simulaciones con 90.000 partículas y potencialmente podrían ser más.
3) Se pueden testear diferentes tipos de interacciones.
4) Se mejoró notablemente la visualización de los resultados.
5) Mejoras sustanciales en el poder de cálculo.
6) Se pueden incorporar nuevas interacciones de interés de manera sencilla.

Dado que la herramienta no incluye el modelo de esferas viscoelásticas, que es necesario para la simulación de los fenómenos mencionados anteriormente, se implementaron dos tipos de interacción: esferas viscoelásticas sin fricción y con fricción. En la sección 5.6 se describió paso a paso como se implementó las esferas viscoelásticas con fricción, siendo esto un ejemplo de mejora o expansión que es posible realizar con la herramienta.

CAPÍTULO 6
Conclusiones y trabajo futuro

6.1 Introducción

En este capítulo se presenta un resumen de las conclusiones del trabajo realizado. Asimismo, se comentan los trabajos pendientes a ser realizados en una etapa posterior que complemente el trabajo desarrollado.

6.2 Conclusiones

A lo largo del presente trabajo se han estudiado los conceptos de la computación de alto desempeño, la aplicación de técnicas paralelas para la implementación de algoritmos DEM, y el relevamiento de tres herramientas que implementan algoritmos DEM.

En primer lugar se relevó las principales características de la herramienta *Tremolo*. Esta aplicación es una implementación paralela 3D del modelo DEM, con la posibilidad de configurar los parámetros de las simulaciones desde la interfaz gráfica. Una característica interesante es la implementación de algoritmos para la evaluación de potenciales de largo y corto alcance. La incorporación de nuevos potenciales, que fueran de naturaleza física, se transforma en una tarea compleja, debido a la falta de documentación disponible de la aplicación. Además, el conjunto de potenciales implementados por la aplicación son de origen químico, por lo que se decidió seguir investigando alternativas.

El diseño de los algoritmos DEM, estudiado en el capítulo 4, fue utilizado para la simulación 2D de los siguientes fenómenos:

- Procesos de segregación de rocas por tamaño como producto de sismos producidos en colisiones de asteroides (conocido como “efecto nueces de Brasil”).
- Producción de nubes de polvo a baja velocidad relativa como producto de la aceleración inducida por un sismo generado a partir de una colisión.

Estos fenómenos son simulados a través de la aplicación de la física de medios granulares. Debido a que los algoritmos DEM implementados por Pöschel y Schwager en el 2005 se corresponden con una versión serial, el poder de cómputo está limitado a unos pocos segundos de simulación de ambos fenómenos. Los resultados de estas experiencias fueron publicados bajo el título "*Activity-like appearance of asteroids caused by impact processes*" en el Congreso de Río de Janeiro (2009) [16].

La última herramienta relevada fue *ESyS-Particle*, la cual implementa un motor paralelo 3D, con un conjunto básico de interacciones de naturaleza física. Con *ESyS-Particle* fue posible simular la colisión de un proyectil contra un aglomerado de partículas. Los resultados de esta experiencia fueron presentados bajo el título "Física de Medios Granulares con aplicación a procesos de impacto" en el Taller de Ciencias Planetarias realizado en La Plata, Argentina (2010) [17]. La ejecución de la simulación fue realizada en el cluster Ciencias, sobre la plataforma implantada, resultado de la investigación de este proyecto. Conjuntamente, fue posible comprobar el poder de cómputo de la arquitectura paralela disponible en la Facultad de Ciencias (cluster Ciencias), para el estudio de fenómenos físicos complejos (en particular de la física de medios granulares) con grandes cantidades de partículas. Notar que la mejora más notoria, redunda en mayores tiempos de simulación obtenidos en el modelado de fenómenos.

Durante el desarrollo del proyecto se estableció un importante contacto con otro grupo de trabajo e investigación sobre implementación paralela de algoritmos DEM. El grupo de trabajo es encabezado por Dion Weatherley (University of Queensland) y Steffen Abe (RWTH Aachen University), quienes son responsables del desarrollo, mantenimiento y soporte de *ESyS-Particle*.

Por último, cabe destacar que se extendió el código de *ESyS-Particle*, incorporando el modelo de esferas viscoelásticas, necesario para la simulación del "efecto nueces de Brasil". Con este aporte, resultado de la investigación realizada, se está contribuyendo a que se obtengan resultados satisfactorios tanto, desde el punto de vista de la calidad de resultados físicos que serán obtenidos, como desde el punto de vista de la mejora de eficiencia computacional alcanzada por esta versión paralela de los algoritmos DEM frente a la contraparte serial estudiada.

6.3 Trabajo futuro

Como parte del trabajo de investigación, es importante identificar aquellas líneas de investigación de trabajo a futuro para dar continuidad al esfuerzo invertido. Para ello, esta sección pretende mostrar el trabajo futuro que es necesario realizar para seguir avanzando en el conocimiento y resultados de la aplicación de técnicas paralelas en la física de medios granulares.

A partir de los resultados obtenidos en las experiencias realizadas con *ESyS-Particle*, surge la principal línea de investigación, que consiste en la implementación de atracción gravitacional mutua entre las partículas (autogravedad). Para ello, se debe extender el código de *ESyS-Particle* con un algoritmo de árbol o de potencial gravitatorio medio. En el capítulo 3 del presente trabajo, se explicaron los conceptos teóricos del algoritmo de árbol más simple, denominado Burnes-Hut. Este método fue desarrollado en sus comienzos para la evaluación de potenciales astrofísicos. Por lo tanto, puede servir como un punto de partida en la investigación e implementación de la autogravedad. La otra posibilidad es la implementación de la autogravedad, a través de un algoritmo de potencial medio. El método consiste en dividir el dominio en una malla regular, donde en cada celda se aplique un potencial gravitatorio medio. La idea es que el valor del potencial asociado a cada celda, se actualice de forma periódica. Dependiendo de en que región del dominio se encuentre una partícula, es la gravedad a

la cual será sometida. La finalidad de la implementación de la autogravedad es obtener una simulación del "efecto de las nueces de Brasil" más fiel a la realidad.

Otra línea de investigación, es estudiar como lograr una representación más cercana a la realidad del momento del impacto entre un proyectil y un aglomerado de partículas. En la actualidad, la colisión causa que el proyectil salga disparado al rebotar contra el aglomerado. El efecto observado en la realidad, demuestra que el proyectil queda incrustado en el aglomerado, sufriendo una deformación a causa de la fuerza del impacto, existiendo la posibilidad de ser destruido en el proceso. Para ello, se sugiere extender el código de *ESyS-Particle*, ya que se cuenta con el apoyo de los creadores de la herramienta, lo cual permite una mejor visibilidad y comprensión del código.

Referencias bibliográficas

[1] Computational Granular Dynamics: Models and Algorithms. T. Pöschel & T. Schwager, (Springer-Verlag, Berlin Heidelberg, 2005) ISBN: 978-3-540-21485-4.

[2] Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications. M, Griebel, S. Knapek, G. Zumbusch (Springer-Verlag, Berlin Heidelberg, 2007). ISBN: 978-3-540-68094-9.

[3] Designing and Building Parallel Programs. Ian Foster. Addison-Wesley, 1995. ISBN 0-201-57594-9.

[4] William Stallings, Computer Organization and Architecture 5th Edition, Prentice Hall, 2000, ISBN: 0130812943.

[5] Andrew S. Tanenbaum, Structured Computer Organization 4th Edition, Prentice Hall, 1998, ISBN: 0130959901.

[6] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 3rd Edition, Morgan Kaufmann Publishing Co., 2002, ISBN: 1558607242.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing, 22(6):789-828, 1996.

[8] W. Gropp, E. Lusk, and A. Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA, 1994. ISBN 0-262-57104-8.

[9] Santamaría, P. (2009). Algoritmo del árbol jerárquico para el cómputo de la interacción gravitatoria de muchos cuerpos, Facultad de Ciencias Astronómicas y Geofísicas, Universidad Nacional de La Plata, Argentina.

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

[11] Dion Weatherley (2009). ESyS-Particle v2.0 Users Guide, Earth Systems Science Computational Centre, University of Queensland.

[12] Steffen Abe (2009). How to add a new type of interaction to ESyS-Particle, RWTH Aachen University.

[13] Dion Weatherley, Steffen Abe (2009). Short Course "DEM Simulations in Geoscience using ESySParticle", RWTH Aachen University.

[14] ESyS-Particle. ESyS-Particle resource information page. URL: https://launchpad.net/esys-particle, consultada en setiembre de 2009.

[15] Tremolo. Tremolo resource information page. URL: http://wissrech.ins.uni-bonn.de/research/projects/tremolo/, consultada en abril de 2009.

[16] Andrea Maciel, Gonzalo Tancredi, Marcelo Traverso, Departamento de Astronomía, Facultad de Ciencias, y Observatorio Astronómico Los Molinos, Montevideo, Uruguay. Presentación bajo el título "Activity-like appearance of asteroids caused by impact processes", realizada en el Symposium #263 "ICY BODIES IN THE SOLAR SYSTEM", International Astronomical Union, 3-7 Agosto 2009, Rio de Janeiro, Brasil.

[17] G.Tancredi, A. Maciel, I. Elgue, S.Bruzzone, S.Roland, Depto. Astronomía, Fac. Ciencias, Observatorio Astronómico Los Molinos, PEDECIBA-Física, Montevideo, Uruguay. Presentación bajo el título "Física de Medios Granulares con aplicación a procesos de impacto", realizada en el V Taller de Ciencias Planetarias, 23 - 26 Febrero 2010, La Plata, Argentina.