UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Optimización de la Simulación de N Cuerpos Computacionales con Atracción Gravitacional sobre Intel Xeon Phi KNL

AUTORES: Ezequiel Moreno

DIRECTOR: Enzo Rucci, Adrian Pousa

CODIRECTOR:

ASESOR PROFESIONAL:

CARRERA: Licenciatura en Informática.

Resumen

En la comunidad HPC, el uso de aceleradores se ha consolidado como estrategia para mejorar el rendimiento de los sistemas al mismo tiempo que la eficiencia energética. Recientemente, Intel introdujo Knights Landing (KNL), la segunda generación de aceleradores Xeon Phi. Este trabajo se enfoca en la paralelización de la simulación de N cuerpos computacionales sobre un acelerador Xeon Phi KNL. Esta simulación, requiere de alto poder computacional para ser procesada con un tiempo de respuesta aceptable. Comenzando por una implementación secuencial, se muestra cómo es posible que la implementación paralela alcance 2355 GFLOPS a través de diferentes optimizaciones.

Palabras Clave

Xeon Phi, Knights Landing, N body, AVX-512, MCDRAM, HPC.

Conclusiones

A partir de los resultados obtenidos, se considera que se ha cumplido con el objetivo planteado inicialmente. Partiendo de una implementación secuencial, se han aplicado y analizado diferentes técnicas de optimización que permiten obtener una solución de alto rendimiento para el problema de estudio sobre los procesadores Xeon Phi KNL. La implementación desarrollada es capaz de alcanzar un pico de rendimiento de 2355 GFLOPS y se encuentra disponible para beneficio de la comunidad científica

Trabajos Realizados

Inicialmente, se estudió en detalle la arquitectura de los aceleradores Xeon Phi KNL, y el problema de N cuerpos computacionales con atracción gravitacional, incluyendo la bibliografía existente en la temática. A continuación, se diseñaron y desarrollaron diferentes soluciones paralelas al problema estudiado que puedan ejecutarse en aceleradores Xeon Phi KNL, considerando diferentes optimizaciones aplicables. Luego, se midió el rendimiento considerando diferentes escenarios en cada caso y se realizó un análisis de los resultados. Finalmente, se comparó la propuesta llevada a cabo y sus resultados con otros existentes en la literatura.

Trabajos Futuros

Considerando los resultados obtenidos, se espera avanzar en la implementación de métodos avanzados para esta simulación. También en el desarrollo de soluciones optimizadas para otras simulaciones de la física y áreas afines sobre multiprocesadores.

Dado que las GPUs son el acelerador dominante en la actualidad, interesa realizar una comparación de rendimiento y eficiencia energética entre estas arquitecturas.

Fecha de la presentación: Mayo 2020

Optimización de la Simulación de N Cuerpos Computacionales con Atracción Gravitacional sobre Intel Xeon Phi KNL

Ezequiel Tomás Moreno.

Facultad de Informática.

Universidad Nacional de La Plata.

Director: Dr. Enzo Rucci.

Director: Dr. Adrián Pousa.

Tesina presentada para obtener el grado de
Licenciado en Informática
27 de mayo de 2020

Resumen

En la comunidad HPC, el uso de aceleradores se ha consolidado como estrategia para mejorar el rendimiento de los sistemas al mismo tiempo que la eficiencia energética. Recientemente, Intel introdujo Knights Landing (KNL), la segunda generación de aceleradores Xeon Phi. Entre sus características destacadas, se puede mencionar su gran cantidad de núcleos, la incorporación de las instrucciones vectoriales AVX-512 y la integración de una memoria de alto ancho de banda. El presente trabajo se basa en una implementación paralela de la simulación de N-cuerpos computacionales con atracción gravitacional en un acelerador Xeon Phi de la arquitectura Knights Landing. Además de representar la base de un gran número de aplicaciones de la astrofísica, esta simulación requiere de alto poder computacional para ser procesada con un tiempo de respuesta aceptable. Comenzando por una implementación secuencial, se muestra cómo es posible que la implementación paralela alcance 2355 GFLOPS a través de la aplicación incremental de diferentes optimizaciones, orientadas a lograr un óptimo rendimiento en la arquitectura utilizada.

Agradecimientos

A mis padres, Teresa y Sergio, por el apoyo y sustento durante todos estos años.

A mis directores, Enzo y Adrian, por su paciencia y conocimientos, que parecen infinitos.

Al instituto III-LIDI por brindarme los medios y recursos para realizar esta tesina.

Al instituto LINTI por darme la oportunidad de iniciarme en docencia, investigación y extensión durante mi carrera.

A mis amigos de siempre, por estar en las buenas y en las malas.

A Rob, por su compañía en este tiempo difícil.

Finalmente, mi agradecimiento más grande a la universidad pública y en particular a la Universidad Nacional de La Plata, por darme la oportunidad de estudiar una carrera.

Índice.

1. Introducción

En primer lugar, se presenta la motivación de esta tesina (Sección 1.1). Luego se enuncian los objetivos y metodología a emplear (Sección 1.2), luego los resultados obtenidos (Sección 1.3) y luego las publicaciones generadas por el presente trabajo (Sección 1.4). Por último, se describe la organización del documento (Sección 1.5).

1.1 Motivación

Durante décadas, el desarrollo de plataformas HPC (High performance computing) estuvo focalizado casi únicamente en mejorar el rendimiento de las mismas. Esto provocó un crecimiento exponencial en los requerimientos de potencia de estos sistemas, lo que a su vez los llevó a consumir enormes volúmenes de energía eléctrica (tanto para alimentación como refrigeración). En consecuencia, el costo económico también se ve agravado.

El problema del consumo energético se presenta como uno de los mayores obstáculos para el diseño de sistemas que sean capaces de alcanzar la escala de los Exaflops. Por lo tanto, la comunidad científica está en la búsqueda de diferentes maneras de mejorar la eficiencia energética de los sistemas de cómputo de altas prestaciones. Una tendencia reciente para incrementar el poder computacional y al mismo tiempo limitar el consumo de potencia de estos sistemas consiste en incorporarles aceleradores y coprocesadores,como pueden ser las GPUs de NVIDIA y AMD o los coprocesadores Xeon Phi de Intel. Estos sistemas híbridos que emplean diferentes recursos de procesamiento son capaces de obtener mejores cocientes FLOPS/Watt [1].

Recientemente, Intel ha presentado la segunda generación de sus procesadores Xeon Phi, con nombre clave Knights Landing (KNL). Entre sus principales características, se pueden mencionar la gran cantidad de núcleos con soporte para hyper-threading, la incorporación de las instrucciones vectoriales AVX-512 y la integración de una memoria de alto ancho de banda [2].

Entre las áreas que se ven afectadas por los problemas actuales de los sistemas HPC se encuentra la física, debido a que cuenta con un número creciente de aplicaciones que requieren de cómputo de altas prestaciones para alcanzar tiempos de respuesta aceptables. Una de esas aplicaciones es el clásico problema de la simulación de N cuerpos computacionales, la cual aproxima en forma numérica la evolución de un sistema de cuerpos en el que cada uno interactúa con todos los restantes [3].

El uso más conocido de esta simulación quizás sea en la astrofísica, donde cada cuerpo representa una galaxia o una estrella particular que se atraen entre sí debido a la fuerza gravitacional. Sin embargo, también se ha empleado en otras áreas muy diferentes. Por ejemplo, para el plegado de proteínas en la biología computacional [4] o para la iluminación global de una imagen en computación gráfica [5].

Existen diferentes métodos para computar la simulación de los N cuerpos [6]. La forma más sencilla se denomina directa (o all-pairs) y consiste en evaluar todas las

interacciones entre todos los pares de cuerpos. Es un método de fuerza bruta que posee alta demanda computacional ($O(n^2)$). Debido a su complejidad computacional, la versión directa sólo es empleada cuando la cantidad de cuerpos es moderada, o bien para computar las interacciones entre cuerpos cercanos en combinación con una estrategia para los que se encuentran lejanos entre sí. Esta segunda opción es el enfoque empleado por métodos avanzados que permiten simular la interacción entre una gran cantidad de cuerpos, como pueden ser el de Barnes-Hut (O(n * log(n))) o el Fast Multipole Method(O(n)). Por lo tanto, al acelerar la versión directa no solo se mejora a la misma sino también a las otras que la emplean como componente.

De manera de poder computar simulaciones de N cuerpos con tiempos de respuesta aceptables, resulta necesario desarrollar nuevas soluciones computacionales que sean capaces de aprovechar las arquitecturas HPC actuales. Por ese motivo, esta tesina se plantea como objetivo optimizar el rendimiento de las implementaciones para computar el problema de los N cuerpos con atracción gravitacional en la arquitectura Intel Xeon Phi KNL.

1.2 Objetivo y metodología

El objetivo de esta tesina consiste en optimizar soluciones para la simulación de N cuerpos computacionales con atracción gravitacional sobre aceleradores Xeon Phi KNL. Para ello se realizarán las siguientes actividades:

- Se estudiará el problema de la simulación de N cuerpos computacionales con atracción gravitacional y sus requisitos computacionales.
- Se examinará la arquitectura de los aceleradores Xeon Phi KNL, los modelos de programación y las técnicas de optimización aplicables.
- Se relevará la bibliografía existente en la temática a partir de la búsqueda en bases de datos especializadas.
- Se diseñarán y desarrollarán diferentes soluciones paralelas al problema estudiado que puedan ejecutarse en aceleradores Xeon Phi KNL, considerando diferentes optimizaciones aplicables.
- Se medirá el rendimiento considerando diferentes escenarios en cada caso y se realizará un análisis de los mismos.
- Se comparará la propuesta llevada a cabo y sus resultados con otros existentes en la literatura.

1.3 Resultados obtenidos

Entre los resultados obtenidos, se pueden mencionar:

- Una implementación optimizada para computar la simulación de N cuerpos computacionales con atracción gravitacional sobre aceleradores Xeon Phi KNL. Como se mencionó en la Sección 1.1, esta implementación no sólo representa

una solución en sí misma, sino que puede beneficiar a otras que la empleen como componente. Para beneficio de la comunidad científica, los algoritmos se encuentran disponibles en un repositorio web público [https://github.com/e-moreno/Gravitational_N_Bodies_Xeon_Phi_KNL].

- Una evaluación de rendimiento de aceleradores Xeon Phi KNL para resolver la simulación de N cuerpos computacionales con atracción gravitacional. Este análisis no sólo permite evidenciar qué optimizaciones tienen mayor impacto en la mejora de rendimiento sino también apreciar el potencial de esta arquitectura para el problema de estudio.

1.4 Publicaciones

Esta tesina ha servido como base para dos publicaciones científicas:

- "Simulación de N Cuerpos Computacionales sobre Intel Xeon Phi KNL", E. Rucci, E. Moreno, M. Camilo, A. Pousa, and F. Chichizola. En: Actas del XXV Congreso Argentino de Ciencias de la Computación (CACIC 2019), ISBN: 978-987-688-377-1, págs. 194-204, 2019. Disponible en http://sedici.unlp.edu.ar/handle/10915/90463

- "Optimization of the N-body Simulation on Intel's Architectures Based on AVX-512 Instruction Set", E. Rucci, E. Moreno, A. Pousa, and F. Chichizola, En: Computer Science – CACIC 2019. Revised Selected Papers, ISBN:978-3-030-48325-8, Springer International Publishing, págs. 37-52, 2020. Disponible en: https://link.springer.com/chapter/10.1007/978-3-030-48325-8_3.

1.5 Organización

El resto del documento se organiza de la siguiente forma:

- En el capítulo 2 se introducen los conceptos físicos y matemáticos necesarios para computar el problema de los N cuerpos con atracción gravitacional, se mencionan los distintos algoritmos posibles para este problema y se detalla la versión secuencial del método directo.

- En el capítulo 3, en primer lugar se detalla el contexto histórico de los procesadores, que llevó al surgimiento de los aceleradores y luego se mencionan las características de estos. A continuación se presenta la arquitectura de los aceleradores Intel Xeon Phi Knights Landing, los modelos de programación disponibles para los mismos y las técnicas de optimización para dicha arquitectura.

- En el capítulo 4, se explica la implementación base del algoritmo y se detallan todas las sucesivas optimizaciones que incrementalmente se agregaron hasta llegar a una versión final. A continuación se detallan las pruebas experimentales y los resultados obtenidos en las mismas y por último se analizan trabajos relacionados.

- En el capítulo 5 se detallan las conclusiones del presente trabajo y se mencionan las posibles líneas de trabajo futuras.

2. Simulación de N cuerpos Computacionales con Atracción Gravitacional

El presente capítulo introduce los conceptos físicos y matemáticos necesarios para computar el problema de los N cuerpos con atracción gravitacional, así como los pasos del algoritmo secuencial. En la sección 2.1 se describe el problema general de los N cuerpos. Luego, en la sección 2.2, se detallan los fundamentos físicos y matemáticos de los cálculos del algoritmo. En la subsección 2.3, se presenta el problema específico de los N cuerpos con atracción gravitacional y los diferentes algoritmos para computarlo. En la subsección 2.4 se describe en detalle los pasos de computación del algoritmo naive. Por último en la sección 2.5 se resume el contenido del capítulo.

2.1 Problema general

Las computaciones de partículas, en contraste con otros modelos de computación como el de grillas, se utilizan para modelar sistemas que consisten de partículas que ejercen fuerzas entre sí. El problema de los N cuerpos, quizás el caso más característico de la computación de partículas, es la tarea de predecir como N masas puntuales en espacio libre, en un sistema cerrado, se mueven de acuerdo a determinadas fuerzas (atracción electroestática[1], atracción gravitacional[2], etc). Para el problema con N = 2 cuerpos, la solución general en términos de funciones algebraicas es bien conocida. Sin embargo, para resolver el problema de los N cuerpos con N > 2, la computación nos brinda la única aproximación posible.

2.2 Fundamentos físicos de la atracción gravitacional

La física subyacente a la simulación es fundamentalmente la mecánica Newtoniana. La simulación se realiza en 3 dimensiones espaciales y la atracción gravitacional entre dos cuerpos C_1 y C_2 se computa usando la ley de gravitación universal de Newton, mostrada en la Fig. 1:

$$F = \frac{G * m_1 * m_2}{r^2}$$

Fig. 1 - Ecuación de gravitación universal de Newton.

[1] Sean las dos cargas puntuales q_1 y q_2 separadas una distancia r, que se encuentran en reposo con respecto al origen O del sistema de referencia inercial. La fuerza que la carga q_1 ejerce sobre q_2 se denomina fuerza electrostática y viene dada por la ley de Coulomb. [http://www2.montes.upm.es/dptos/digfa/cfisica/electro/fuerza_electr.html]

[2] Atracción entre dos cuerpos, dada por la ecuación de gravitación universal de Newton, definida en “Philosophiae Naturalis Principia Mathematica”.

En esta ecuación: F corresponde a la magnitud de la fuerza gravitacional entre los cuerpos, G corresponde a la constante de gravitación universal, m_1 corresponde a la masa del cuerpo C_1, m_2 corresponde a la masa del cuerpo C_2, y r corresponde a la distancia Euclídea entre los cuerpos C_1 y C_2 como se ve en la Fig. 2. Cuando N es mayor a 2, la fuerza de gravitación sobre un cuerpo se obtiene con la sumatoria de las fuerzas de gravitación ejercidas por los N - 1 cuerpos restantes.

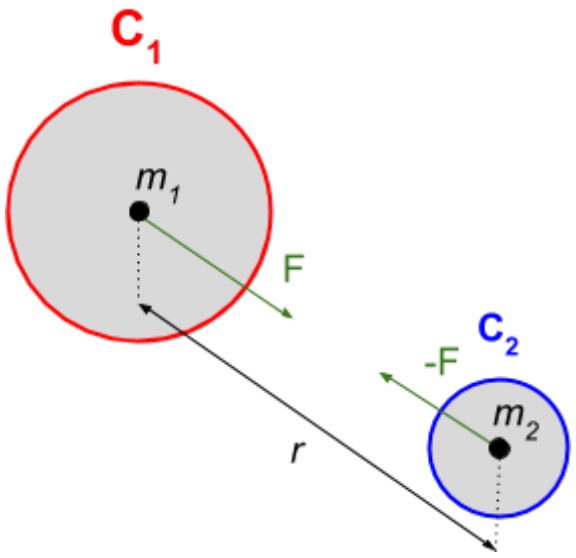

Fig. 2 - Diagrama de fuerza de atracción gravitacional entre dos cuerpos.

La fuerza de atracción se traduce entonces en una aceleración del cuerpo mediante la aplicación de la segunda ley de Newton, la cual está dada por la ecuación mostrada en la Fig. 3:

$$F = m \times a \quad \Leftrightarrow \quad a = \frac{F}{m}$$

Fig. 3 - Segunda ley de Newton.

Donde *F* es el vector fuerza, calculado utilizando la magnitud obtenida con la ecuación de gravitación y la dirección y sentido del vector que va desde el cuerpo afectado hacia el cuerpo que ejerce la atracción. De la ecuación anterior, se concluye que se puede calcular la aceleración de un cuerpo dividiendo la fuerza total por su masa. Se puede notar que mediante estas ecuaciones la magnitud de la fuerza es igual para ambos cuerpos, pero la aceleración experimentada es mayor desde el cuerpo de menor masa hacia el de mayor masa, ya que para obtener la aceleración se divide la fuerza por la masa del cuerpo afectado. Teniendo la aceleración del cuerpo, se puede obtener la diferencia de posición del mismo, mediante las ecuaciones del movimiento, también parte de la mecánica Newtoniana. Para un grado de libertad (una coordenada espacial), estas ecuaciones son las mostradas en la Fig. 4:

$$\frac{\partial x}{\partial t} = v, \quad \frac{\partial v}{\partial t} = a$$

Fig. 4 - Ecuaciones del movimiento de la mecánica Newtoniana.

El significado de estas ecuaciones es simple. Por un lado, la velocidad ***v*** es la pendiente de la ecuación de posición con respecto al tiempo en un punto (la derivada en este punto), es decir, el ritmo de cambio de la posición con respecto al tiempo. Por otro lado, la aceleración a es el análogo para la gráfica de velocidad con respecto al tiempo, es decir, el ritmo de cambio de la velocidad con respecto al tiempo (la derivada).

Para poder obtener la diferencia de posición experimentada por el cuerpo, partiendo de la aceleración obtenida mediante la segunda ley de Newton, debemos despejar x de estas ecuaciones diferenciales. Para lograr esto, realizamos la operación inversa a la derivada, es decir, la integral de la aceleración en un punto (instante de tiempo). Esta nos dará la velocidad y, al integrar la velocidad en ese punto, obtendremos la diferencia de posición que utilizaremos posteriormente para mover al cuerpo.

Para el cálculo de dicha integral, tendremos que utilizar lo que se conoce como *integrador numéric*o, el cual es un algoritmo que permite aproximar la integral de una función en un punto de forma numérica con un error asociado. Estos integradores se clasifican de acuerdo con su error asociado, lo que se conoce como orden del integrador. Este es el grado del error expresado en función del intervalo de tiempo sobre el que se integra. A modo de ejemplo supongamos que utilizamos un integrador numérico para integrar sobre un intervalo de tiempo h, el error de dicho integrador siendo h^n, entonces decimos que el integrador es de enésimo orden. El orden de un integrador es mejor mientras más grande sea la potencia n, ya que h es menor a la unidad, por lo tanto el error decrece al aumentar el orden [7].

El primer ejemplo de integrador que consideramos es el integrador de Euler, este aproxima la integral de las ecuaciones de movimiento como se ve en la Fig. 5:

$$x_{n+1} = x_n + h \cdot v_n$$
$$v_{n+1} = v_n + h \cdot a_n$$
$$[a_n = F(x_n) / m]$$

Fig. 5 - Integrador de Euler, v → velocidad, a → aceleración, subíndice → instante de tiempo.

En esta ecuación, x_{n+1} es la posición en el instante de tiempo siguiente, v_{n+1} es la velocidad en el instante de tiempo siguiente, x_n es la posición actual, v_n la velocidad actual, a_n es la aceleración actual, y h es el instante de tiempo transcurrido. Este integrador es de primer orden, ya que su error es de orden h (lineal).

Una mejora que puede aplicarse a este integrador es realizar el cálculo de la velocidad teniendo en cuenta la fuerza calculada con la nueva posición del cuerpo, este algoritmo se denomina integrador de Euler-Cromer [8], mostrado en la Fig. 6:

$$x_{n+1} = x_n + h \cdot v_n$$
$$v_{n+1} = v_n + h \cdot a_{n+1}$$
$$[a_{n+1} = F(x_{n+1}) / m]$$

Fig. 6 - Integrador de Euler - Cromer, v → velocidad, a → aceleración, subíndice → instante de tiempo.

El integrador de Euler-Cromer, sigue siendo de primer orden, pero con una pequeña modificación al cálculo de x_{n+1}, para considerar la velocidad en el punto medio del intervalo en lugar del valor actual, este se transforma en un mejor algoritmo, el algoritmo de leapfrog [7]. Este algoritmo define las integrales de la posición y la velocidad como se ve en la Fig. 7:

$$x_{n+1} = x_n + h \cdot v_{n+\frac{1}{2}}$$
$$v_{n+\frac{3}{2}} = v_{n+\frac{1}{2}} + h \cdot a_{n+1}$$
$$[a_{n+1} = F(x_{n+1}) / m]$$

Fig. 7 - Algoritmo leapfrog, v → velocidad, a → aceleración, subíndice → instante de tiempo.

Con este planteamiento nos surge la incógnita de cómo obtener la velocidad en el punto medio del intervalo $\boldsymbol{v_{1/2}}$ en la primera iteración del algoritmo, la aproximación más simple es utilizar medio paso del método de Euler, por única vez, para obtener este valor, como se ve en la Fig. 8.

$$v_{\frac{1}{2}} = v_0 + \frac{1}{2} \cdot h \cdot a_0$$

Fig. 8 - Aproximación a $v_{1/2}$ por método de Euler.

Con todas estas consideraciones, se construye un método denominado velocity verlet [9], que nos permite aproximar la integral a las ecuaciones de movimiento y es de segundo orden. Sus ecuaciones, se detallan en la Fig. 9:

$$v_{n+\frac{1}{2}} = v_n + \frac{1}{2} . h . a_n$$

$$x_{n+1} = x_n + h . v_{n+\frac{1}{2}}$$

$$v_{n+1} = v_{n+\frac{1}{2}} + \frac{1}{2} . h . a_{n+1}$$

$$[a_n = F(x_n) / m]$$

$$[a_{n+1} = F(x_{n+1}) / m]$$

*Fig. 9 - Ecuaciones del método de integración **velocity verlet**.*

Este método es la implementación del esquema de integración Leapfrog, en el cual una mitad del cambio de posición emplea la velocidad *vieja* mientras que la otra considera la velocidad *nueva*. La justificación para utilizar este método de integración numérica, está dada por que el mismo es un método simple, mantiene la estabilidad global de las ecuaciones de Newton (es simpléctico) y es un integrador de segundo orden, lo que lo hace más preciso que otros integradores numéricos como el método de Euler. La demostración de estas propiedades [7] escapa del objetivo del presente trabajo y puede hallarse en la bibliografía.

2.3 Problema de los N cuerpos con atracción gravitacional

En la versión de este problema que calcula la atracción gravitacional entre N cuerpos, se simula la evolución de un sistema compuesto por estos cuerpos durante una cantidad de tiempo determinada. Dados la masa y el estado inicial (velocidad y posición) de cada cuerpo, se simula el movimiento del sistema a través de instantes discretos de tiempo. En cada uno de ellos, todo cuerpo experimenta una aceleración que surge de la atracción gravitacional del resto, lo que afecta a su estado.

Para realizar las simulaciones el primer enfoque que podemos tomar, es el algoritmo directo, que calcula las interacciones entre cada par de cuerpos utilizando la ecuación de Newton, y realiza la suma de todas las fuerzas que se ejercen sobre cada cuerpo para obtener la fuerza total resultante que será responsable del movimiento de los mismos. En cada iteración se realizan los cálculos de fuerzas y luego se aplican dichas fuerzas como aceleración mediante la integración de las ecuaciones de movimiento de Newton. Este algoritmo es de orden N^2 ya que, las interacciones de cada uno de los N cuerpos con los N-1 restantes pueden expresarse como se muestra en la Fig. 10:

$$N \,.\, (N-1) \;=\; N^2 - N \;=\; O(N^2)$$

Fig. 10 - Aproximación al orden del algoritmo.

Todos los demás algoritmos para el problema de los N cuerpos con atracción gravitacional toman este de base, utilizando distintas propiedades para realizar menos cálculos, y por lo tanto este algoritmo constituye un buen punto de partida para aplicar optimizaciones, ya que todas las optimizaciones que se apliquen sobre este algoritmo, servirán para las demás versiones.

La magnitud de la fuerza de atracción gravitacional es inversamente proporcional al cuadrado de la distancia entre los cuerpos, por lo tanto, si dos cuerpos están suficientemente alejados, la fuerza entre ellos es despreciable. Newton descubrió este hecho y mostró que un grupo de cuerpos puede ser tratados como un único cuerpo en relación a otros más alejados. En particular, un grupo de cuerpos B puede ser aproximado a un solo cuerpo b, que tiene la masa total de todos los cuerpos pertenecientes a B y se encuentra localizado en el centro de masa de B. Entonces, la fuerza entre un cuerpo distante i, y los cuerpos en B puede aproximarse como la fuerza entre i y b [3] Estas ideas de Newton permitieron la creación de métodos aproximados para simulaciones de N cuerpos que reducen la cantidad de cálculos y el orden de la versión directa.

El primero de los métodos alternativos para el cálculo de N cuerpos, denominado algoritmo de Barnes-Hut [10], es de orden N * Log(N). Este algoritmo utiliza árboles para representar la distribución espacial de los cuerpos (quadtree para dos dimensiones y octree para tres). El otro algoritmo, que también utiliza árboles, es conocido como Fast Multipole Method [11] (FMM), y para distribuciones uniformes de cuerpos, su tiempo de ejecución es de orden lineal. Si bien estos métodos permiten computar la simulación de manera eficiente con una mayor cantidad de cuerpos, aún necesitan de la versión directa para hacerlo. Una descripción más profunda de estos algoritmos queda fuera del alcance de este trabajo.

2.4 Algoritmo secuencial de N cuerpos

El código secuencial para resolver el problema de los N cuerpos en su versión directa consiste de los siguientes pasos, para cada cuerpo:

1. Calcular las fuerzas de gravitación ejercidas por todos los otros cuerpos.

2. Sumar componente a componente (cada coordenada espacial) las fuerzas al total.

Y luego, también para cada cuerpo:

3. Calcular las integrales para transformar fuerza en desplazamiento.

4. Aplicar el desplazamiento al cuerpo.

Podemos ver estos pasos en el pseudocódigo de la Fig. 11:

```
Para cada cuerpo de i = 1 a N:

    Para cada cuerpo de j = 1 a N:

        Calcular la fuerza ejercida por j sobre i.
        Sumar a fuerzas que afectan a i.

    Calcular desplazamiento del cuerpo i.

    Mover cuerpo i.
```

Fig. 11 - Pseudocódigo de N cuerpos secuencial (versión directa).

En este código podemos identificar dependencias de datos entre los pasos de la computación. En primer lugar, el cálculo de desplazamiento de los cuerpos, depende del cómputo de las fuerzas en el paso anterior. En segundo lugar,el cálculo de las fuerzas de una iteración, depende de la actualización de la posición de los cuerpos de la iteración inmediatamente anterior.

Esta es la base sobre la cual partimos para aplicar las optimizaciones, y constituye la versión secuencial del método naive de N cuerpos.

2.5 Resumen

El problema de los N cuerpos puede ser considerado un problema clásico de computación de partículas. La versión de este problema en astrofísica (el problema de los N cuerpos con atracción gravitacional) utiliza la ley universal de gravitación de Newton y las ecuaciones de movimiento de la mecánica Newtoniana para simular la evolución de cuerpos astronómicos a lo largo del tiempo. En esta versión, se computa la evolución de los cuerpos, calculando el cambio de posición de los mismos en cada instante discreto de tiempo. La distancia entre estos intervalos discretos de tiempo, DT, se toma como un parámetro de la simulación.

Por la utilización de las ecuaciones de movimiento Newtonianas, se requiere aproximar la integral de una función en un punto, para lo cual existen varias aproximaciones. Comparando las ventajas y desventajas de los distintos integradores numéricos (entre ellos el nivel de error asociado), se justificó la elección del integrador velocity verlet, que es una implementación del esquema de integración leapfrog**,** por sobre los algoritmos de Euler y Euler-Cromer. La elección se fundamenta en que el primero presenta características que favorecen el cómputo de las ecuaciones de movimiento sin agregar excesiva complejidad al código.

Además de la versión directa, existen otros métodos para el cálculo del problema de los N cuerpos. Entre los más conocidos, se encuentran los métodos que utilizan árboles para optimizar los cálculos, mencionados en la sección 2.3. Si bien estos algoritmos son más eficientes, especialmente con una gran cantidad de cuerpos, están basados en el método directo y lo usan como componente para implementar su estrategia. Es por eso que se elige el enfoque directo sobre las otras variantes. Al

acelerar la versión directa no solo se mejora a la misma sino también a las otras variantes que la emplean como componente.

3 Intel Xeon Phi Knights Landing

El presente capítulo presenta a los aceleradores Xeon Phi Knights Landing de Intel. En la sección 3.1 se describe la evolución histórica de los procesadores que terminó impulsando la llegada de los multicore y luego los aceleradores. En la sección 3.2, se detalla la arquitectura de la primera generación de Xeon Phi, Knights Corner. Finalmente, en la sección 3.3 se describe la arquitectura Knights Landing, los modelos de programación que esta soporta y las técnicas de optimización específicas para esta arquitectura.

3.1 Contexto histórico

Históricamente, la mejora de rendimiento de los procesadores estuvo dada por el aumento en el número de transistores en el chip (por la ley de Moore) y en la frecuencia del reloj. Este aumento en la cantidad de transistores en el chip, y en la frecuencia del reloj de dichos procesadores, permitió que se implementen técnicas de procesamiento avanzadas que se tradujeron en mejoras en el rendimiento de las aplicaciones. Por un lado, el mayor número de transistores, permitió que se implementen técnicas que permiten explotar paralelismo a nivel de instrucciones (Instruction Level Parallelism, ILP). Entre esas técnicas se encuentran: Pipelining, Arquitectura superescalar, Instrucciones Single Instruction Multiple Data[3], ejecución fuera de orden, Predicción de saltos y ejecución especulativa, simultaneous multithreading y memorias cache más grandes. Por otro lado, el incremento de la frecuencia del reloj permitió que un procesador pueda ejecutar (teóricamente), más instrucciones por segundo [12].

Este proceso de mejora, alcanzó un límite en la década del 2000, debido a dos causas principales, en primer lugar resultaba más difícil extraer más ILP (Instruction Level Parallelism), de los programas y además no fue posible continuar incrementando la frecuencia del reloj de los procesadores principales. En primer lugar, debido a que el consumo de potencia y su disipación en forma de calor llegó a límites insostenibles para el correcto funcionamiento de los circuitos integrados [13].

Debido a esto, se volvió necesario buscar alternativas de diseño que permitieran continuar elevando el rendimiento de los procesadores. En este punto, en lugar de seguir aumentando la complejidad de la organización interna del chip, las empresas fabricantes comenzaron a integrar dos o más núcleos computacionales más simples en un solo chip. Si bien estos núcleos son individualmente más limitados y menos veloces, al combinarlos se obtienen mejoras significativas en el rendimiento global del procesador. A este tipo de procesadores se los denominó multicore.

Al introducirse estos procesadores, se volvió una necesidad que las aplicaciones exploten explícitamente el paralelismo para poder aprovechar la potencia del hardware

[3] I*nstrucciones SIMD*: Estas instrucciones permiten aplicar la misma operación sobre un conjunto de datos diferentes al mismo tiempo. Tantos los procesadores actuales de Intel como los de AMD cuentan con instrucciones SIMD. En los procesadores Intel, se conocen como SIMD Streaming Extensions (SSE) para 128 bits o Advanced Vector Extensions (AVX) para 256 y 512 bits.

subyacente, tanto el paralelismo de datos como el de tareas. El paralelismo de datos se basa en procesamiento de múltiples datos de manera simultánea. Por otro lado, el paralelismo de tareas consiste en ejecutar múltiples tareas independientes en paralelo.

Desde su origen, los procesadores multicore fueron sofisticando su diseño en las sucesivas familias de procesadores de propósito general. Los primeros procesadores de este tipo, consistían de prácticamente dos procesadores mononúcleo en el mismo chip. Las sucesivas generaciones han aumentado el número de núcleos e incorporado caches L2 y L3, que son compartidas por todos los núcleos o un subconjunto de ellos.

Como se mencionó en la sección 1.1, el uso de aceleradores se presenta como una alternativa exitosa a la hora de reducir el consumo energético de los sistemas HPC. En la actualidad existe una amplia variedad de aceleradores, siendo los más populares las GPUs (mayormente de Nvidia), seguidos por los manycore Xeon Phi de intel. Esta tendencia se puede notar en la última edición del TOP500 [14] (Edición Noviembre 2019), donde 136 equipos cuentan con GPUs, 22 con Xeon Phi y 12 con alguna otra clase.

3.2 Primera generación - Knights corner

Los coprocesadores Xeon Phi, de Intel, fueron presentados en el año 2012 como propuesta para el segmento de aceleradores en HPC. Los Xeon Phi de primera generación, se componen de hasta 61 núcleos pentium x86, que cuentan con unidades vectoriales extendidas de 256 bits, mediante el conjunto de instrucciones Knights Corner (KNC). Cada núcleo también cuenta con Hyper-Threading[4] (cuatro hilos hardware por núcleo) y tiene integrada una cache L1 (32 kb de datos y 32 kb de instrucciones) y una cache L2 completamente coherente (512 kb de datos y 512 kb de instrucciones).

El Xeon Phi KNC soporta hasta 8 controladores de memoria, con dos canales GDDR5 cada uno, y se conecta al host a través de un bus PCIe de segunda generación.

Los procesadores Xeon Phi, admiten dos modos de ejecución diferentes, el modo nativo permite emplear al Xeon Phi como un sistema de cómputo autónomo y ejecutar programas utilizando solo los recursos de este, el otro modo disponible es el modo offload, como coprocesador, el cual es análogo al funcionamiento de otros aceleradores como las GPU. En el modo offload, el host ejecuta la porción secuencial del código de la aplicación e invoca a la ejecución de determinados kernels en el Xeon Phi.

3.3 Segunda generación - Knights Landing

Knights landing es el nombre elegido por intel para la arquitectura de la segunda generación de procesadores Intel Xeon Phi [2]. Los procesadores Xeon Phi Knights Landing, son procesadores *manycore* que proveen un paralelismo masivo de threads[5] y de datos con alto ancho de banda de memoria . Estos procesadores, tienen una tasa

[4] Hyper-Threading, que es el nombre comercial de Intel para SMT (Simultaneous Multi Threading), es detallado en la sección 3.3.5.1.
[5] A lo largo del presente trabajo, los términos thread e hilo se utilizan de manera intercambiable.

de operaciones de punto flotante por segundo (FLOPS) por núcleo, superior a los Xeon contemporáneos, además de incluir un nuevo conjunto de instrucciones vectoriales (detallado en la sección 3.3.1). A diferencia de sus antecesores (KNC), los procesadores KNL, son autónomos, pueden arrancar sistemas operativos por su cuenta y conectarse directamente a la red.

3.3.1 Arquitectura.

El diseño de Knights landing, que podemos apreciar en la Fig. 12, tiene el concepto de 'tile', que es la unidad básica de replicación. Cada tile consiste de dos cores, dos unidades de procesamiento vectorial por core y una cache L2 de 1M compartida entre ambos núcleos. Estas tiles están físicamente replicadas 38 veces, pero solo pueden estar activas a la vez 36 de ellas, por lo que en total el procesador consta de 72 cores y 144 unidades de procesamiento vectorial. Un tile tiene, soporte de 4 hyper threads por core, cache de ejecución fuera de orden más grande, mayor ancho de banda de cache L1 y L2, instrucciones vectoriales AVX-512, TLB y cache L1 más grandes.

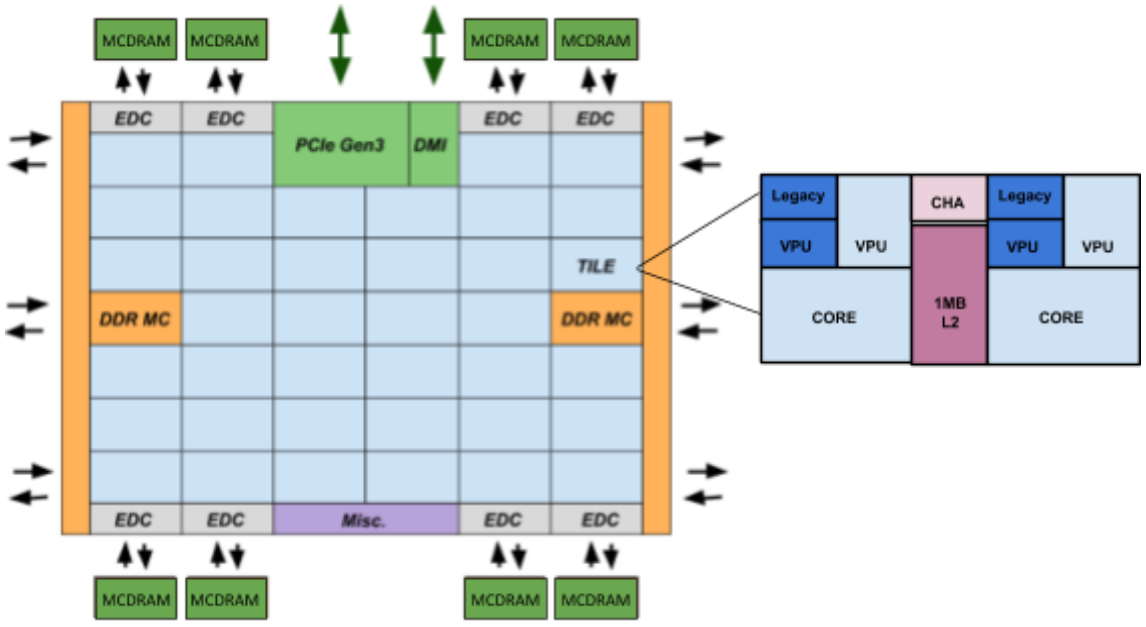

Fig. 12 - Arquitectura de Xeon Phi KNL.

Dos cores en la misma tile, tienen acceso a sus cache L2 locales y sus unidades de procesamiento locales, para acceder a datos en otras cache, memoria principal, o realizar operaciones de entrada / salida, necesitan comunicarse con otras partes mediante la interconexión en chip. Las tiles están interconectadas por una red de dos dimensiones con coherencia de cache. La red de interconexión utiliza el protocolo de coherencia de cache MESIF [15], para mantener la coherencia entre la cache de las distintas tiles.

El set de instrucciones de la arquitectura, es similar al resto de los procesadores x86; agregando los nuevos grupos de instrucciones sobre el soporte para la instrucciones de base de la arquitectura x86/x86-64 equivalentes a los procesadores Xeon recientes.

Inicialmente existían los conjuntos de instrucciones SIMD denominados Streaming SIMD Extensions (SSE), diseñados por Intel e introducidos en 1999 en su procesador Pentium 3 [16]. Luego de SSE, aparecieron las instrucciones Advanced Vector Extensions (AVX), que extendieron el tamaño de los operandos a 256 bits. Este set de instrucciones fue creado por Intel en 2008 e introducido en su arquitectura Sandy Bridge [17]. Los procesadores Knights Landing soportan todas los conjuntos de instrucciones legacy, incluyendo las diferentes versiones de SSE y AVX, e introduce uno nuevo llamado AVX-512 de 512 bits. En la Fig. 13 podemos observar la evolución del ancho de banda de las instrucciones vectoriales.

Fig. 13 - Evolución de ancho de instrucciones vectoriales.

Como se mencionó anteriormente, la arquitectura Knights Landing introduce *Advanced Vector Extensions* de 512 bits (Intel AVX-512) que consiste de instrucciones vectoriales que realizan operaciones SIMD sobre hasta 512 bits en simultáneo. Estas instrucciones, surgen como una evolución de las instrucciones AVX de 256 bits. Además del soporte de instrucciones AVX-512, la arquitectura dispone de 32 registros lógicos, 8 registros de máscara para '*predicación vectorial*'[6] e instrucciones de gather y scatter para cargar y almacenar datos esparcidos. Una instrucción AVX-512 puede realizar 8 operaciones de suma/multiplicación con operandos de doble precisión (16 FLOPs) o 16 operaciones con operandos de simple precisión (32 FLOPs). Las instrucciones de AVX-512 se dividen en cuatro categorías, '*AVX-512 Foundation instructions*', qué son las instrucciones básicas recién mencionadas, '*AVX-512 Conflict Detection Instructions*', '*AVX-512 Exponential and Reciprocal Instructions*' y '*AVX-512 Prefetch Instructions*'.

[6] **Predicación**: en ciencias de la computación, la predicación es una característica arquitectónica que provee una alternativa a las transferencias condicionales de control implementadas por instrucciones de máquina como ramificaciones condicionales, llamadas condicionales, retornos condicionales y tablas de ramificación. La predicación funciona ejecutando instrucciones de ambos caminos de ejecución y solo permite que las instrucciones del camino tomado modifiquen el estado arquitectural.

Knights landing nos provee de una variedad de configuraciones, que tradicionalmente, estaban cableadas en el hardware y eran decisiones de diseño que no podían modificarse. Estas configuraciones, permiten a Knights Landing comportarse como máquinas distintas dependiendo de las configuraciones que elegimos desde la BIOS. Específicamente, los modos de memoria y modos de cluster, nos brindan flexibilidad para ajustar el funcionamiento a los distintos modelos de programación que necesitemos utilizar. Y a su vez las aplicaciones pueden ser modificadas para sacar provecho de estos modos también.

3.3.2 Modos de memoria

Los procesadores Knights Landing, tienen dos tipos de memoria: MCDRAM (Multi Channel Dynamic Random Access Memory) y DDR. Estos dos tipos en conjunto proveen un alto ancho de banda y una gran capacidad. La memoria puede ser configurada en uno de entre tres modos, que podemos observar en la Fig. 14:

1. Flat mode:
   Donde la MCDRAM es tratada como memoria estándar en el mismo espacio de direcciones que la DDR.
2. Cache mode:
   Donde la MCDRAM es utilizada como cache para la DDR.
3. Hybrid:
   Donde una parte de la MCDRAM es utilizada como cache y el resto en modo flat.

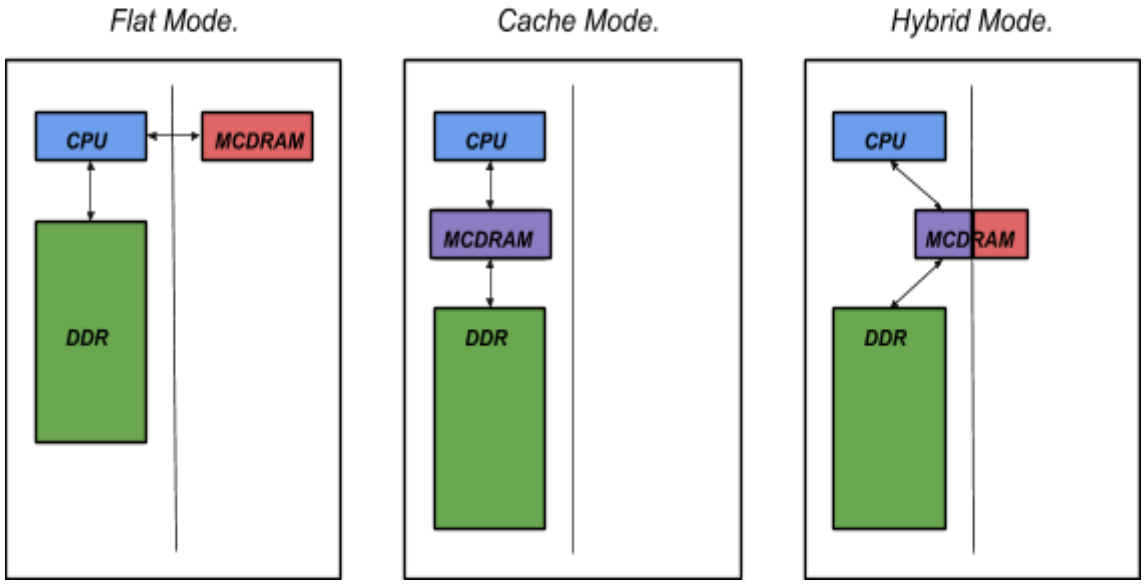

Fig. 14 -Modos de memoria disponibles en Xeon Phi KNL.

La memoria DDR provee alta capacidad externa al paquete del Knights Landing. Hay dos controladoras DDR4 en lados opuestos del chip, cada una controlando 3 canales, cada canal soporta un módulo DIMM de hasta 2400 MHz.

La MCDRAM es una memoria apilada en 3 dimensiones que no es más rápida para un único acceso que la memoria principal, pero puede soportar mucho más ancho de banda. Por lo tanto cuando esta es empleada como cache o utilizada directamente para almacenar los datos más usados por una aplicación, se puede lograr una aceleración sustancial. Esta memoria consiste de 8 dispositivos integrados on package, de 2GB cada uno para un total de 16GB de memoria de alto ancho de banda.

Se tienen varias formas para trabajar con la memoria MCDRAM:

- Con el modo numactl (NUMA control utility), no se debe cambiar el código de la aplicación. La utilización de este modo hace que todas las alocaciones de memoria del programa se hagan en la MCDRAM.

- Con la utilización de la librería autohbw, la cual tampoco requiere cambios en el código de la aplicación (salvo la inclusión/configuración de la librería) se puede hacer que las alocaciones de un cierto rango de tamaños se realicen en la MCDRAM. Los segmentos de datos y stack no son alocadas en la MCDRAM.

- Por último, con la utilización de la librería memkind, se puede mapear alocaciones de memoria específicas a la MCDRAM. Sin embargo, requiere cambiar el código de la aplicación para su uso.

3.3.3 Modos de clúster

Los modos de cluster son formas de particionar el chip en regiones virtuales de cores con la intención de mantener la mayor parte de las comunicaciones locales a cada región. El objetivo del particionamiento es reducir la latencia e incrementar el ancho de banda de las comunicaciones en la matriz. Asimismo, la arquitectura permite mejorar la localidad de los datos procesados por cada partición realizando afinidad entre las regiones y las direcciones de memoria. Dos de estos modos tienen subdivisiones. Estos modos son:

1. Modo ***All to All.***

2. Modo ***Quadrant.***

    a. ***Quadrant.***
    b. ***Hemisphere.***

3. Modo ***Sub NUMA Clustering*** (SNC).

    a. ***SNC/4.***
    b. ***SNC/2.***

El modo All to All, es el más general de los 3. Este modo es el de menor rendimiento global, pero puede ser usado con cualquier configuración de memorias DDR. Es el

único modo que puede usarse cuando las memorias DDR no son idénticas en capacidad. En este modo todas las caches son visibles a todos los procesadores.

En el modo Quadrant, el chip es dividido en 4 cuadrantes virtuales. Las comunicaciones se limitan al cuadrante en el que se encuentran y se reduce la latencia comparado con el modo All to All. Este modo es transparente para el software en el sentido de que no se necesita modificarlo para beneficiarse del modo. El único requerimiento para poder utilizar este modo es que la configuración sea simétrica, es decir, que todas las memorias DDR tengan la misma capacidad.

En el modo SNC-4, se expone cada cuadrante como un cluster NUMA (Non Uniform Memory Access) independiente, con coherencia de cache. Estos nodos son visibles al sistema operativo mediante tablas ACPI[7] de la BIOS. Este modo no es enteramente transparente al programador. El programa debe estar optimizado para accesos NUMA para poder beneficiarse de este modo.

Los modos Hemisphere y SNC-2 son equivalentes a Quadrant y SNC-4 respectivamente, con la diferencia de que en lugar de dividir el chip virtualmente en cuatro partes, lo dividen en dos.

El modo por defecto es Quadrant (a excepción de cuando las memorias DDR no son de la misma capacidad, en cuyo caso el único modo disponible es All to All). Para la mayoría de los programas, desde el punto de vista de la aplicación los modos All to All, Quadrant y Hemisphere serán indistinguibles. Las consideraciones a tener en cuenta para el programador, surgen en comparar, estos modos previamente mencionados, con los modos SNC-4/SNC-2.

La relación de los distintos tipos de memoria con los modos de clúster, mencionados previamente, se da de la siguiente manera: independientemente del modo de clúster seleccionado, toda la memoria estará disponible para todos los cores en cualquier momento. Además, toda la memoria es completamente coherente a nivel de cache, es decir, que todos los cores tienen la misma visibilidad de los datos que están en cada lugar.

La diferencia fundamental radica en si cada tipo de memoria (DDR - MCDRAM) es de acceso uniforme o no uniforme (UMA/NUMA). En el modo Quadrant, cada tipo de memoria es de acceso uniforme, por lo tanto la latencia desde cualquier core a cualquier posición de memoria del mismo tipo es esencialmente la misma. En SNC-4 cada tipo de memoria es NUMA, los cores y memorias se dividen en cuatro cuadrantes y tienen menor latencia para acceder a memoria ‘cercana’ (dentro del mismo cuadrante) y mayor latencia para la memoria ‘lejana’ (de un cuadrante diferente). Los modos restantes, Hemisphere y SNC-2 son variaciones de los anteriores, en los que los cores y las memorias se dividen en dos partes en lugar de cuatro. En estos modos

[7] ACPI (Advanced Configuration and Power Interface) es un estándar que controla el funcionamiento de la BIOS. Este estándar expone información del hardware instalado al sistema operativo mediante tablas [https://www.intel.com/content/www/us/en/standards/processor-vendor-specific-acpi-specification.html].

en comparación con los anteriores, puede haber mayor latencia por lo que se reduce el ancho de banda efectivo.

3.3.4 Modelos de programación

La arquitectura Knights Landing, por ser compatible con arquitecturas x86, soporta la utilización de los modelos de programación paralela clásicos como son OpenMP, MPI[8], Intel Thread Building Blocks[9], e incluso Pthreads[10].

El hecho de que la arquitectura KNL soporte estos modelos de programación representa una ventaja respecto a las GPUs, ya que en estas, los algoritmos deben ser codificados de una forma que refleje la arquitectura para lograr un rendimiento óptimo. Como los modelos de programación de las GPU son muy diferentes a los de los procesadores tradicionales, los programadores se ven obligados a aprender detalles técnicos específicos de las arquitecturas y su programación, lo que conlleva mayor dificultad.

Por ser utilizado para el presente trabajo, procedemos a describir el primero de estos modelos. Open Multi-Processing (OpenMP) es una API que provee un conjunto de directivas, funciones de librerías y variables de entorno para la programación de aplicaciones paralelas sobre arquitecturas de memoria compartida, con implementaciones disponibles para los lenguajes C, C++ y Fortran. Se puede decir que OpenMP es multiplataforma, ya que posee versiones para diversas arquitecturas y sistemas operativos, dando portabilidad a las aplicaciones paralelas que la utilizan.

La utilización de OpenMP, permite paralelizar programas mediante de la adición de directivas al código secuencial. Usando estas directivas podemos: crear un conjunto de hilos, repartir tareas paralelas independientes entre ellos y sincronizarlos para el acceso a datos compartidos.

OpenMP utiliza un enfoque de bloque estructurado y el modelo fork-join. Como podemos ver en la Fig. 15, el programa comienza con un único hilo de ejecución (denominado maestro). Cuando se encuentra la primera directiva parallel, que define un bloque paralelo, se crea un conjunto de hilos que ejecutan el mismo código, los cuales pueden coordinarse para repartir el trabajo utilizando las directivas para tal fin. Al final del bloque paralelo existe una barrera de sincronización implícita y sólo el hilo maestro continuará con la ejecución del programa. El código encerrado por un bloque paralelo se denomina *región paralela* y puede comprender tanto paralelismo de datos como de tareas.

---

[8] MPI (Message Passing Interface Standard): Es un estándar de librería de pasaje de mensajes basada en el consenso del foro de MPI. [https://computing.llnl.gov/tutorials/mpi/]

[9] Pthreads es una librería de C que permite el manejo de threads para ejecución paralela [https://computing.llnl.gov/tutorials/pthreads/].

[10] Thread Building Blocks es una librería de C++ para programación paralela con memoria compartida y computación heterogénea [https://software.intel.com/en-us/tbb].

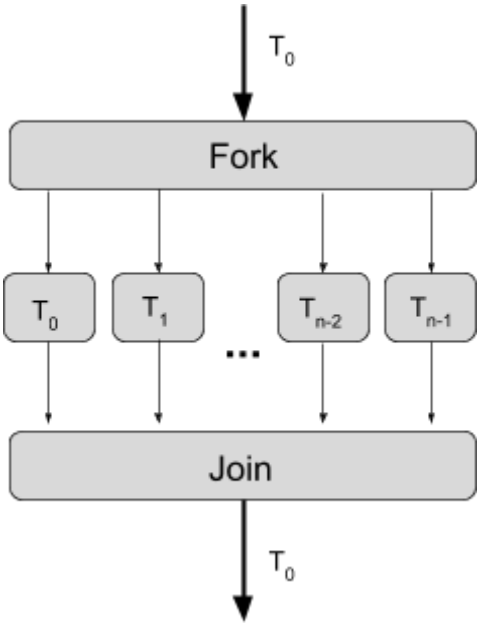

Fig. 15 - Modelo Fork - Join de OpenMP.

Existe otra directiva de paralelización relevante a los fines del presente trabajo: la directiva for, utilizada para dividir las iteraciones de un bucle entre los threads. La forma de asignar iteraciones a los threads se define mediante la cláusula schedule, la cual admite las siguientes políticas:

- La política *static* divide las iteraciones en partes iguales que llamaremos bloques y asigna éstos a los threads mediante el algoritmo round-robin. El tamaño de estos bloques se puede definir mediante un parámetro, y de no estar definido explícitamente, se divide el total de iteraciones entre la cantidad de threads para definir dicho tamaño.

- La política *dynamic* divide las iteraciones en bloques y luego los asigna en tiempo de ejecución a los threads a medida que se liberan (bajo demanda). En caso de no estar definido el tamaño de bloque, el valor por defecto es de una iteración.

- La política *guided* comienza con un tamaño de bloque de trabajo para asignar y luego reduce exponencialmente el tamaño del mismo a medida que asigna bloques a los threads. De esta forma buscar evitar el desbalance de carga cuando la cantidad de iteraciones no es exactamente divisible por la cantidad de threads. Esta política, toma como parámetro el tamaño mínimo al que puede reducir los bloques, que por defecto es de una iteración.

- La política *runtime* permite que se determine el tipo de scheduling y el tamaño de las partes a asignar mediante una variable de entorno en tiempo de ejecución.

- Por último, la política *auto*, delega la elección de políticas de scheduling al compilador o al entorno de ejecución.

Cuando no se especifica el scheduling en la directiva for, el tipo por defecto depende de la implementación y no está definido en el estándar [18].

Los hilos dentro de un bloque paralelo, se ejecutarán dentro del mismo espacio de direcciones y mediante cláusulas específicas de openMP podrán compartir el acceso a variables declaradas en el mismo (variables definidas como *shared*); también se permite que una variable sea designada como privada a un hilo (variables definidas como *private*), en este caso cada hilo tendrá una copia que usará mientras dure la región paralela.

Una de las mejoras más importantes en la versión 4.0 del estándar de OpenMP consiste en la inclusión de la directiva simd para vectorización guiada, la cual permite indicarle explícitamente al compilador qué bucles vectorizar.

En resumen, OpenMP nos ofrece directivas para paralelizar iteraciones de una estructura de control iterativa (for en C) y secciones paralelas, donde todo el código es paralelizado. Además nos provee de directivas que afectan la visibilidad de las variables entre los threads, directivas de sincronización entre los mismos y por último, funciones y variables de entorno para acceder a información del entorno y los threads en tiempo de ejecución.

3.3.5 Técnicas de optimización

Por la arquitectura, descrita anteriormente, del Xeon Phi KNL, podemos aplicar ciertas técnicas de optimización para lograr el mejor rendimiento de los programas que corren en el mismo.

3.3.5.1 Hyper-Threading

Definimos multithreading por hardware, como la capacidad que tienen algunas microarquitecturas de mantener el estado de ejecución de múltiples hilos, sin depender de los sistemas operativos, multiplicando el hardware para este fin. El resto de los recursos del procesador en general son compartidos, pero algunas unidades pueden estar replicadas. Esta capacidad, permite que los cambios de contexto entre threads se hagan mucho más rápido, y se pueda aprovechar mejor el procesador cuando alguno de los threads está haciendo una operación bloqueante.
La tecnología de Hyper-Threading [19], es una implementación específica de multithreading por hardware. Hyper-Threading hace que un único procesador físico, funcione como varios procesadores lógicos; los recursos físicos de ejecución se comparten y el estado arquitectural se duplica para los procesadores lógicos. Desde una perspectiva de software o arquitectura, esto significa que los sistemas operativos y programas pueden ejecutar varios threads por procesador físico. Desde una perspectiva de microarquitectura, esto significa que las instrucciones de los procesadores lógicos persistirán y se ejecutarán simultáneamente en los recursos de ejecución compartidos.

Particularmente, respecto al Hyper-Threading en Intel Xeon Phi KNL, la elección de cuantos threads por núcleo utilizar es altamente dependiente de la aplicación. Sin embargo, como principio general, siempre es recomendable mantener las funcionalidades de hyper threading encendidas, aunque se utilice un solo thread por núcleo físico.

Cuando de la ejecución con Hyper-Threading se trata, podemos diferenciar dos tipos de performance:,la performance por thread individual y la performance agregada de todos los threads. En cuanto a la performance individual, en KNL, la máxima performance individual por thread se logra al ejecutar un thread por core. Sin embargo, la performance agregada crecerá, hasta el uso de 4 threads por core. El uso de tres threads por core, generalmente, no será óptimo ya que la microarquitectura de Intel Xeon Phi KNL, particiona a los recursos en cuartos cuando se utilizan 3 o 4 threads en el mismo core. Debido a esto, una configuración con 3 threads por core, tendrá un menor aprovechamiento de recursos y podemos concluir que generalmente las configuraciones óptimas serán de 1, 2 o 4 threads por core, dependiendo de la aplicación.

Las limitaciones prácticas de capacidad de memoria o paralelismo pueden limitar la escalabilidad al incrementar los threads utilizados por core. Por esto se recomienda evaluar la cantidad de threads por núcleo para cada problema en particular y no tomar estas recomendaciones como una regla absoluta.

Las librerías de tiempo de ejecución de Intel, tienen la capacidad de controlar cómo se mapean los threads de openMP a las unidades físicas de procesamiento [20]. Las políticas de mapeo se denominan afinidad de los threads y se pueden controlar mediante la variable de entorno KMP_AFFINITY. La afinidad restringe la ejecución de ciertos threads a un subconjunto de las unidades físicas de procesamiento. Los tipos de afinidad de threads posibles son:

- None: No mapea los threads a ningún contexto en particular. Este es el modo de afinidad por defecto.

- Balanced: Distribuye los threads en las unidades físicas de procesamiento hasta que todas las unidades tengan al menos un thread. Esta política además asegura que si varios threads se mapean al mismo procesador físico utilizando multithreading por hardware, estos tengan ids de thread cercanos entre sí.

- Compact: Esta afinidad mapea al thread N+1 en el contexto de threads más cercano posible al contexto donde se mapea el thread N.

- Scatter: Distribuye los threads de la manera más uniforme posible entre las unidades de procesamiento disponibles.

- Explicit: Mapea los threads a una lista de Proc IDs específica, indicada mediante el modificador proclist=<lista>, que es obligatorio para este modo.

- Disabled: Deshabilita completamente la interfaz de afinidad de threads.

Estas políticas, que permiten controlar el mapeo de threads a procesadores físicos, pueden ser útiles para mejorar el rendimiento de los programas paralelos. La mejora

proviene de lograr que hilos que comparten recursos (ej. datos) o se comunican frecuentemente, se ejecuten en contextos de ejecución contiguos, obteniendo de esta forma una mejor localidad de datos o un tiempo menor de comunicación.

3.3.5.2 Localidad de datos

Como se describió en la Sección 3.3.1, la arquitectura de KNL tiene múltiples niveles de cache con diferentes velocidades de acceso. Mientras más datos del conjunto de trabajo de un procesador se encuentren en cache, menos accesos a memorias más lentas (DDR o MCDRAM) serán requeridos para su procesamiento, lo que llevará a una mejora del rendimiento.

Debido a la arquitectura de memoria cache de KNL, es recomendable que los programas exploten la localidad espacial y temporal de los datos con los que trabajan. La localidad temporal de los datos refiere a el principio de las memorias cache por el cual los datos recientemente utilizados se mantienen en la memoria por determinado tiempo, ya que es probable que se vuelvan a utilizar. La localidad espacial de los datos es el principio por el cual las memorias cache traen toda la “localidad” de un dato (el dato solicitado más una cierta cantidad de celdas vecinas) al ir a buscar dicho dato en la memoria principal, ya que es probable que datos contiguos sean consultados sucesivamente en un intervalo corto de tiempo.

Aprovechar las propiedades de localidad de la memoria cache hace que, cuando el procesador necesite un dato, con alta probabilidad estará disponible en la cache y se podrá consumir en menor tiempo. El programador tendrá que trabajar en la organización de los datos con los que trabaja el algoritmo y en la estructura del código para poder explotar de manera óptima estas propiedades de la memoria cache.

3.3.5.3 Vectorización

La vectorización nos permite realizar una misma operación sobre múltiples datos en paralelo, reduciendo así el tiempo de ejecución requerido por las operaciones que pueden ser vectorizadas. Como los procesadores de la arquitectura KNL tienen múltiples unidades vectoriales por core, su aprovechamiento provee grandes oportunidades de mejora para las aplicaciones que sean capaces de usarlas.

Como se mencionó en la sección 3.3.1, esta arquitectura introduce las instrucciones AVX-512, las cuales proveen el cuádruple de la performance de las instrucciones SSE y el doble de las instrucciones AVX-256, por lo que se recomienda utilizar AVX-512 siempre que sea posible para maximizar el número de operaciones resueltas en paralelo.

La vectorización, es particularmente útil cuando se aplica sobre bucles que realizan operaciones sobre datos, iterando uno por uno sobre estos. Mediante directivas al compilador, se puede transformar estos bucles en bucles más cortos, que realizan las operaciones de forma vectorial sobre varios datos a la vez, obteniendo una ganancia en tiempo de ejecución. Por ejemplo, para un bucle que itera sobre un arreglo de N elementos, sumando un valor a cada uno, puede reducirse el tiempo de ejecución aproximado del mismo a N/16 para datos de simple precisión y N/8 para datos de doble precisión mediante el uso de las instrucciones AVX-512. La mejora en cada caso

proviene del hecho de que este conjunto de instrucciones puede operar sobre 16 datos de simple precisión o 8 de doble precisión a la vez (512 bits). La Fig. 16 ejemplifica la mejora que se puede lograr con el uso de operaciones vectoriales.

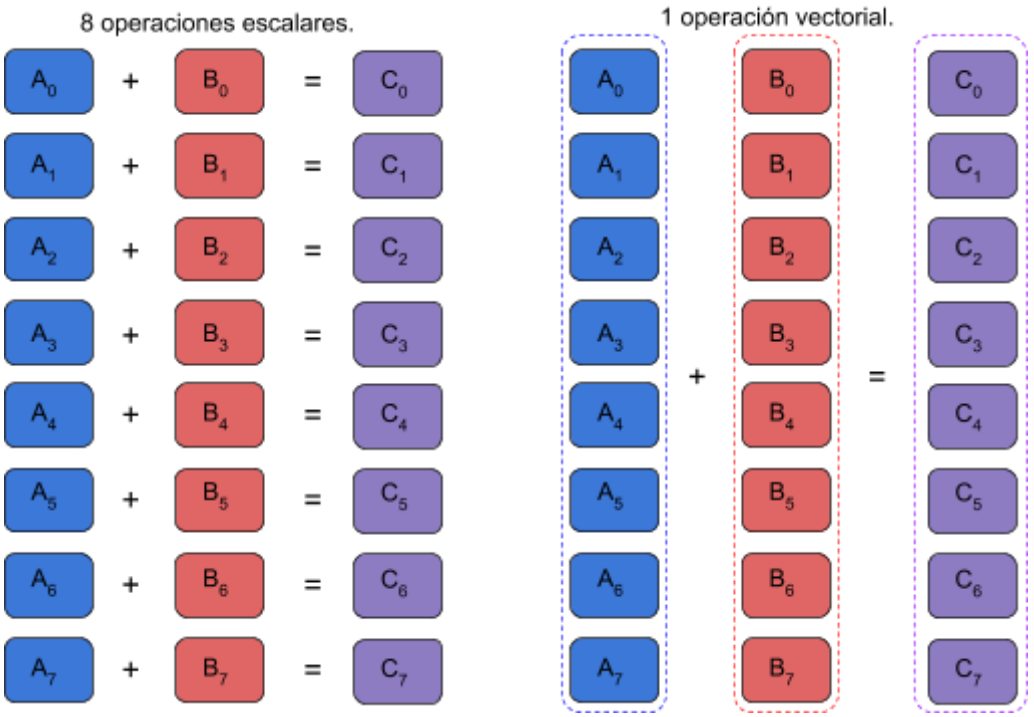

Fig. 16 - Operaciones escalares contra operaciones vectoriales.

En general,la ejecución de operaciones vectoriales resulta conveniente por sobre las operaciones escalares. Sin embargo, hay un límite inferior en el cual el código vectorial puede no ser mejor que su versión escalar. Una aproximación muy simple es, dado N el ancho vectorial del conjunto de instrucciones utilizado y F_{size} el tamaño del tipo de dato con el que se trabaja: Los bucles que iteran más de N/F_{size} veces sobre datos, serán más rápidos en forma vectorial: para bucles más cortos, la performance será similar en algunos casos; y mientras más pequeño el tamaño del bucle, aumentará la probabilidad de que las instrucciones escalares sean mejores. Por ejemplo para las instrucciones AVX-512 y el tipo de dato float, N/F_{size} será 512/64 = 16, y por lo tanto, convendrá vectorizar los bucles que iteran más de 16 veces.

El alineamiento de datos (data alignment) es un método para forzar al compilador a almacenar los datos en memoria con límites específicos de tamaño en bytes [21]. Esto se hace para aumentar la eficiencia de la carga de datos entre el procesador y la memoria. Esto sucede debido a que los procesadores están diseñados para mover datos eficientemente entre desde y hacia memoria en direcciones que son múltiplos específicos de un número de bytes. Por lo tanto, en el caso del Xeon Phi KNL, es deseable forzar al compilador a cargar los datos en memoria en direcciones que sean múltiplos de 64 bytes.

Además de cargar los datos en posiciones alineadas en memoria, el compilador también puede optimizar los accesos cuando es sabido que estos están alineados. Por defecto el compilador no puede saber ni asumir esto, por lo que se le indica que los accesos a datos son alineados mediante pragmas/directivas para que el compilador pueda generar el código óptimo.

3.3.5.4 MCDRAM

La memoria MCDRAM, como se explicó en la Sección 3.3.2, es una memoria que ofrece un mayor ancho de banda y menor capacidad que la memoria DDR más comúnmente utilizada.

La optimización del rendimiento mediante el uso de la MCDRAM puede lograrse utilizando la misma para los datos más accedidos por un programa, cuyo tamaño exceda el de la cache L2 de los procesadores. De esta forma, al acceder a los datos en la memoria MCDRAM y no en la memoria DDR se obtendrá ganancias en performance.

Cabe destacar que la memoria MCDRAM no tiene menor latencia por acceso que la memoria DDR. Por lo tanto, sólo se lograrán optimizaciones por el uso de esta memoria cuando la aplicación requiera un ancho de banda mayor al que la memoria DDR tradicional puede proveer para consumir datos.

3.4 Resumen

El uso de aceleradores se presenta como una alternativa exitosa a la hora de mejorar el rendimiento y, al mismo tiempo, reducir el consumo energético de los sistemas HPC. Entre las opciones disponibles se encuentran la segunda generación de procesadores Xeon Phi de Intel, denominados KNL. Estos procesadores nos proveen de múltiples núcleos de procesamiento, conjuntos de instrucciones vectoriales de gran ancho, gran capacidad de memoria RAM y una memoria adicional con un alto ancho de banda.

La arquitectura de Intel Xeon Phi Knights Landing, tiene el concepto de tile, que agrupa dos cores físicos y una memoria caché entre estos. El tile es la unidad básica de replicación y hay 38 de ellos, pero a lo sumo pueden estar activos 36. La comunicación entre estos se realiza mediante una conexión en forma de matriz y mediante la configuración de los denominados modos de cluster, se pueden sectorizar dichas comunicaciones en mitades o cuartos de la matriz, para mejor localidad, o permitir que todos los nodos se comuniquen con todos. El set de instrucciones de la arquitectura se basa en x86 pero agrega mejoras, como el conjunto de instrucciones vectoriales AVX-512 de 512 bits.

Knights landing soporta dos tipos de memoria: la memoria DDR tradicional, de alta capacidad, y la nueva memoria MCDRAM, que provee un alto ancho de banda. Esta arquitectura permite configurar el modo en el que cooperan estas memorias entre tres variantes. En el primer modo de memoria, denominado flat, DDR y MCDRAM funcionan independientemente y se puede alocar memoria en ambas. En el modo cache, la memoria MCDRAM funciona como cache de la memoria DDR. Finalmente, en el modo hybrid, una parte de la MCDRAM es cache y la otra funciona como una memoria independiente.

La arquitectura Knights Landing soporta múltiples modelos de programación, de los cuales se eligió para el presente trabajo OpenMP, el cual funciona bajo el modelo fork-join y permite definir paralelismo funcional o de datos mediante directivas.

Por las características de esta arquitectura surgen varias optimizaciones que pueden hacerse a los programas que corren en la misma, para mejorar su rendimiento. Entre estas optimizaciones tenemos el uso de varios threads/hilos por núcleo físico. También es importante la explotación de los principios de localidad de las memorias cache. Es vital el uso de las instrucciones vectoriales de mayor ancho de banda posible para realizar múltiples operaciones en simultáneo. Y por último, aprovechar la memoria MCDRAM, lo cual nos provee mejoras en aplicaciones que requieren alto ancho de banda.

4. Optimización de la simulación de N cuerpos computacionales con atracción gravitacional sobre Intel Xeon Phi KNL

En el presente capítulo se introduce la implementación paralela desarrollada y se analiza su rendimiento en un procesador de la arquitectura Intel Xeon Phi KNL. En la sección 4.1, se presenta la solución paralela y se detallan las optimizaciones aplicadas. Luego, en la sección 4.2, de presentan y analizan los resultados experimentales obtenidos. Por último, en la sección 4.3, se describen los trabajos relacionados.

4.1 Implementación

En esta sección se introducen las mejoras aplicadas al algoritmo directo de N cuerpos en su versión secuencial, hasta llegar a la versión paralela más optimizada.

4.1.1 Implementación secuencial base

Inicialmente se desarrolló una implementación secuencial del método directo de N cuerpos, la cual servirá como referencia para evaluar las mejoras introducidas por las técnicas de optimización posteriores. En la Fig. 17 se muestra el código de esta implementación.

```
1  // Para cada instante discreto de tiempo
2  for (t = 1; t <= D; t += DT){
3      // Para cada cuerpo que experimenta una fuerza
4      for (i = 0; i < N; i++){
5          // Componentes de la fuerza del cuerpo i
6          forcesx[i] = forcesy[i] = forcesz[i] = 0.0;
7          // Para cada cuerpo que ejerce una fuerza
8          for (j = 0; j < N; j++){
9              //  Ley de atracción gravitacional de Newton
10             dx = xpos[j] - xpos[i]; dy = ypos[j] - ypos[i]; dz = zpos[j] - zpos[i];
11             dsquared = (dx*dx) + (dy*dy) + (dz*dz) + SOFT;
12             F = G * masses[i] * masses[j]; d32 = 1/pow(dsquared, 3.0/2.0);
13             // Calcular la fuerza total
14             forcesx[i] += F*dx*d32; forcesy[i] += F*dy*d32; forcesz[i] += F*dz*d32;
15         }
16         // Calcular aceleración
17         ax = forcesx[i] / masses[i]; ay = forcesy[i] / masses[i]; az = forcesz[i] / masses[i];
18         // Calcular velocidad
19         dvx = (xvi[i] + (ax*DT/2.0)); dvy = (yvi[i] + (ay*DT/2.0)); dvz = (zvi[i] + (az*DT/2.0));
20         // Calcular posición
21         dpx[i] = dvx * DT; dpy[i] = dvy * DT; dpz[i] = dvz * DT;
22         // Actualizar velocidad
23         xvi[i] = dvx; yvi[i] = dvy; zvi[i] = dvz;
24     }
25     // Actualizar posiciones
26     for(i = 0; i < N; i++){
27         xpos[i] += dpx[i]; ypos[i] += dpy[i]; zpos[i] += dpz[i];
28     }
29 }
```

Fig. 17 - Código de la versión secuencial.

El primer bucle de esta implementación se encarga de iterar sobre el total de los instantes de tiempo de la simulación. El siguiente bucle (línea 4) itera sobre el total de cuerpos. El bucle más interno (línea 8) itera nuevamente sobre el total de cuerpos para calcular las N^2 interacciones posibles y así calcular las fuerzas. Al finalizar este bucle más interno, se calculan los desplazamientos correspondientes en el bucle que lo contiene (líneas 17 a 23). Por último, se desplazan los cuerpos en bucle final (línea 26).

4.1.2 Análisis de paralelismo disponible

A la hora de paralelizar el algoritmo, podemos optar por varias versiones. La versión de memoria compartida y las versiones de memoria distribuida, con sus distintos modelos, constituyen las alternativas disponibles de algoritmos paralelos. La versión de memoria compartida es la que tiene mayor facilidad de programación y menor overhead de comunicaciones. Además, es la que mejor se ajusta a la arquitectura objetivo del presente trabajo, ya que el Xeon Phi es un multiprocesador de memoria compartida.

La implementación de la versión directa de N cuerpos, en el modelo de memoria compartida, simplemente almacena la información de los cuerpos como variables globales de tal forma que todos los threads puedan acceder a ella. Los cuerpos se dividen en bloques de tamaño N / T (para N cuerpos y T threads) y cada thread calcula las interacciones de sus cuerpos con todos los demás. A continuación, se actualizan las posiciones y velocidades de acuerdo a las ecuaciones de movimiento de la mecánica Newtoniana previamente mencionada. La Fig. 18 muestra el pseudo-código de esta solución.

```
Para cada hilo x del total T:

    Para cada cuerpo de i = x * (N/T) a (x+1)(N/T):

        Para cada cuerpo de j = 1 a N:

            Calcular la fuerza ejercida por j sobre i.
            Sumar a fuerzas que afectan a i.

        Calcular desplazamiento del cuerpo i.
        Mover cuerpo.
```

Fig. 18 - Pseudocódigo del algoritmo paralelo.

Si bien este pseudocódigo nos permite ver la estructura del problema, resulta necesario sincronizar algunos de sus cálculos para garantizar su correctitud. Cualquier versión paralela requiere que todos los cuerpos terminen de calcular las fuerzas para comenzar a desplazar los cuerpos, ya que sino se corre el riesgo de mover un cuerpo antes de que otro de los threads haya utilizado la posición en el instante de tiempo anterior. Lo mismo pasa con el movimiento de los cuerpos y el cálculo de la fuerza del siguiente instante de tiempo. Es por eso que se debe esperar a que todos los threads

muevan sus cuerpos antes de usar esas posiciones para el cálculo de las fuerzas de la siguiente iteración.

Para solucionar los inconvenientes que pueden causar interferencia en los cálculos, se requiere utilizar sincronización de barreras, la cual implica sincronizar a todos los threads al final de cada iteración de un algoritmo (el resultado de una iteración es requerido para computar la siguiente). Esta sincronización es denominada de esa forma porque el punto de retardo al final de la iteración representa una barrera a la que todos los threads deben llegar antes que cualquiera de ellos pueda continuar. Las barreras pueden ser necesarias al principio, al final o en pasos intermedios de un loop.

En el caso particular de este algoritmo la sincronización de barreras se requiere en dos lugares del algoritmo paralelo. En primer lugar, para evitar que ningún thread intente mover sus cuerpos antes de que los demás hayan terminado de utilizar las posiciones actuales para computar la fuerza. En segundo lugar, para lograr que ningún thread intente avanzar a la siguiente iteración hasta que todas las posiciones hayan sido actualizadas. Por lo anterior, en este algoritmo tenemos una barrera en el medio de la iteración (al salir del loop más interno), y otra al final de la iteración, al finalizar de mover los cuerpos correspondientes a cada thread.

Con todas estas consideraciones, el algoritmo queda como se muestra en la Fig. 19:

```
Para cada thread x del total T:

    Para cada cuerpo de i = x * (N/T) a (x+1)(n/T):

        Para cada cuerpo de j = 1 a N:

            Calcular la fuerza ejercida por j sobre i.
            Sumar a fuerzas que afectan a i.

        BARRERA 1

        Calcular desplazamiento del cuerpo i.
        Mover cuerpo.

        BARRERA 2
```

Fig. 19 - Pseudocódigo del algoritmo paralelo, con barreras.

La estructura paralela del programa entonces, puede graficarse como se muestra en la Fig. 20, en la cual podemos ver que inicialmente todos los threads computan en paralelo las fuerzas que les corresponden a cada uno. Luego, los threads esperan en la primera barrera hasta que todos hayan terminado de computar el cálculo de fuerzas y luego proceden a aplicar esas fuerzas como movimiento de los cuerpos, para lo que primero deben aplicar las leyes de la mecánica Newtoniana. Una vez que los threads terminan de aplicar el desplazamiento sobre los cuerpos que les corresponden, esperan en la segunda barrera para hasta que todos hayan terminado, para avanzar a la siguiente iteración de la simulación.

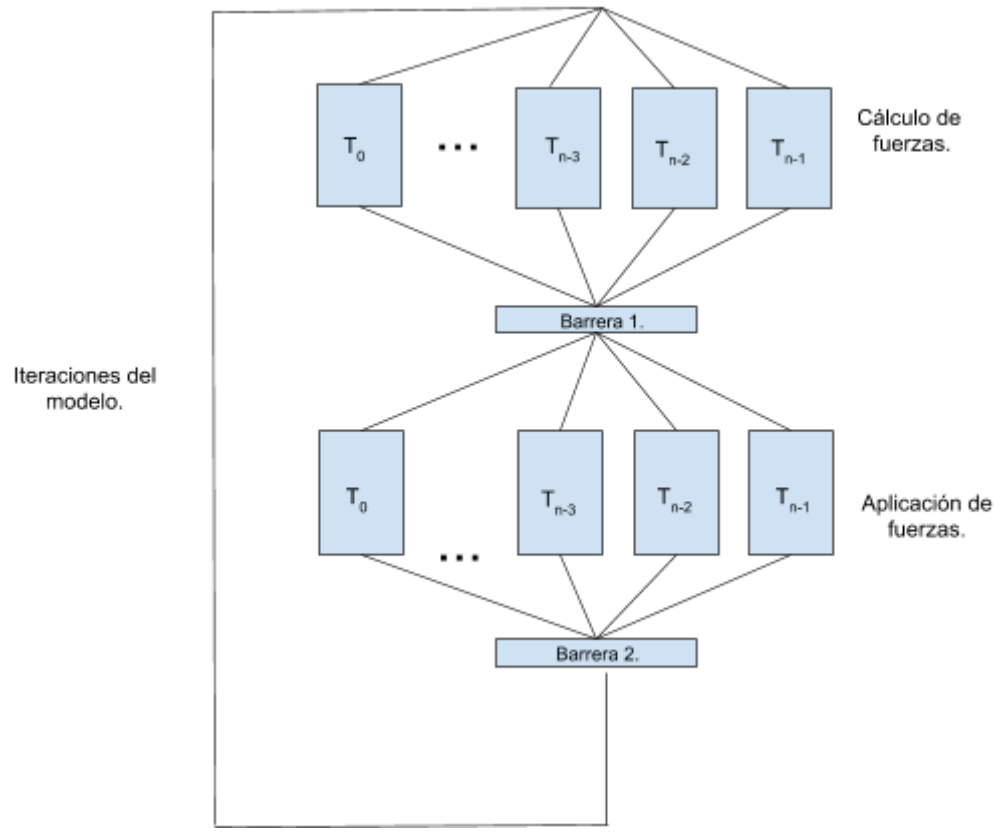

Fig. 20 - Diagrama del funcionamiento del algoritmo (T → threads).

4.1.3 Multihilado

Esta optimización consiste en introducir paralelismo a nivel de hilos a través de directivas OpenMP. Paralelizando el bucle que realiza los cálculos de fuerza por separado del que mueve los cuerpos, se respetan las dependencias del problema ya que un cuerpo no se puede mover hasta que el resto no haya terminado de calcular sus interacciones y tampoco puede avanzar al paso siguiente hasta que todos los demás hilos hayan completado el paso actual. Para este fin se utiliza la directiva for con las cláusulas schedule y private. La opción static para la cláusula schedule permite distribuir equitativamente la cantidad de cuerpos entre los hilos logrando un balance de carga de costo mínimo. Por último, el modificador de acceso private para la variable del for más interno, hace que cada hilo tenga su propia copia de la variable, ya que las iteraciones sobre esta son independientes y de utilizar la variable de manera compartida entre los threads, generaría interferencia. Con todas estas consideraciones, los bucles de las líneas 4 y 26 de la Fig. 17 son paralelizados mediante la inserción de directivas parallel for. Como podemos ver en la Fig. 21, estas directivas se añadieron en las líneas 4 y 27, con sus respectivas cláusulas. La sincronización implícita de la directiva parallel for garantiza el cumplimiento de las dependencias de datos del problema.

```
1  // Para cada instante discreto de tiempo
2  for (t = 1; t <= D; t += DT){
3      // Para cada cuerpo que experimenta una fuerza
4      #pragma omp parallel for schedule(static) private(j)
5      for (i = 0; i < N; i++){
6          // Componentes de la fuerza del cuerpo i
7          forcesx[i] = forcesy[i] = forcesz[i] = 0.0f;
8          // Para cada cuerpo que ejerce una fuerza
9          for (j = 0; j < N; j++){
10             //  Ley de atracción gravitacional de Newton
11             dx = xpos[j] - xpos[i]; dy = ypos[j] - ypos[i]; dz = zpos[j] - zpos[i];
12             dsquared = (dx*dx) + (dy*dy) + (dz*dz) + SOFT;
13             F = G * masses[i] * masses[j]; d32 = 1/pow(dsquared, 3.0/2.0);
14             // Calcular la fuerza total
15             forcesx[i] += F*dx*d32; forcesy[i] += F*dy*d32; forcesz[i] += F*dz*d32;
16         }
17         // Calcular aceleración
18         ax = forcesx[i] / masses[i]; ay = forcesy[i] / masses[i]; az = forcesz[i] / masses[i];
19         // Calcular velocidad
20         dvx = (xvi[i] + (ax*DT/2.0)); dvy = (yvi[i] + (ay*DT/2.0)); dvz = (zvi[i] + (az*DT/2.0));
21         // Calcular posición
22         dpx[i] = dvx * DT; dpy[i] = dvy * DT; dpz[i] = dvz * DT;
23         // Actualizar velocidad
24         xvi[i] = dvx; yvi[i] = dvy; zvi[i] = dvz;
25     }
26     // Actualizar posiciones
27     #pragma omp parallel for schedule(static)
28     for(i = 0; i < N; i++){
29         xpos[i] += dpx[i]; ypos[i] += dpy[i]; zpos[i] += dpz[i];
30     }
31 }
```

Fig. 21 - Código de la primera versión paralela.

4.1.4 Optimizaciones escalares

En general, las funciones de las librerías estándar de c++, están sobrecargadas [22] para permitir que al llamar a una función se llame a la función correcta dependiendo del tipo de datos de los argumentos. Por ejemplo en la librería Math, esperaríamos que al llamar a pow(a,b) dependiendo del tipo de los argumentos a y b, se llame a la función pow para simple o doble precisión. Sin embargo, como las funciones de la librería Math no están sobrecargadas en el espacio global de nombres, cuando llamamos a pow(a,b) con argumentos de simple precisión, se invoca una función de doble precisión. Este tipo de llamadas, genera conversiones de tipos innecesarias que pueden añadir demoras en los cálculos. La solución para esto es llamar a la función powf cuando el tipo de dato es float y pow cuando el tipo de dato es double. En la línea 13 de la Fig. 22 se puede apreciar el uso de la instrucción powf. En las sucesivas implementaciones, se optó por definir constantes de compilador nombradas como POW y DATATYPE para poder definir en tiempo de compilación si se usa float o double.

Una situación similar, ocurre al definir constantes de punto flotante. Se puede especificar cuando una constante de punto flotante es de tipo float mediante una ‘f’ al final del valor de dicha constante, por ejemplo 2.0f. Cuando una constante de punto flotante se define sin especificar si es float o double, la constante es de tipo double por defecto. Esto hace que cuando no se especifica el tipo de una constante de punto flotante, pero se asigna esta a variables float, genere conversiones de tipo innecesarias. Para evitar las conversiones innecesarias, hay que definir las constantes con el tipo que les corresponde, por ejemplo, si necesitamos una constante con el valor 2.0, en el código especificaremos 2.0f si es float y 2.0 si es double. Podemos ver esto en las líneas 13 y 20 de la Fig 22.

Otra posible optimización consiste en reemplazar las divisiones que tienen una constante como denominador, por multiplicaciones con el inverso multiplicativo de dicha constante. Un ejemplo de esto son las divisiones por dos en la línea 20 de la Fig. 22, donde DT/2.0 puede reemplazarse por DT*0.5f. Esto puede mejorar el rendimiento en algunos casos, por que las multiplicaciones de punto flotante son más eficientes que las divisiones. Sin embargo, esta optimización puede no generar mejoras en el rendimiento ya que muchas veces el compilador aplica este tipo de optimizaciones, y el código generado para DT/2.0 sería el mismo que para DT*0.5f.

```
1  // Para cada instante discreto de tiempo
2  for (t = 1; t <= D; t += DT){
3      // Para cada cuerpo que experimenta una fuerza
4      #pragma omp parallel for schedule(static) private(j)
5      for (i = 0; i < N; i++){
6          // Componentes de la fuerza del cuerpo i
7          forcesx[i] = forcesy[i] = forcesz[i] = 0.0f;
8          // Para cada cuerpo que ejerce una fuerza
9          for (j = 0; j < N; j++){
10             //  Ley de atracción gravitacional de Newton
11             dx = xpos[j] - xpos[i]; dy = ypos[j] - ypos[i]; dz = zpos[j] - zpos[i];
12             dsquared = (dx*dx) + (dy*dy) + (dz*dz) + SOFT;
13             F = G * masses[i] * masses[j]; d32 = 1/powf(dsquared, 1.5f);
14             // Calcular la fuerza total
15             forcesx[i] += F*dx*d32; forcesy[i] += F*dy*d32; forcesz[i] += F*dz*d32;
16         }
17         // Calcular aceleración
18         ax = forcesx[i] / masses[i]; ay = forcesy[i] / masses[i]; az = forcesz[i] / masses[i];
19         // Calcular velocidad
20         dvx = (xvi[i] + (ax*DT*0.5f)); dvy = (yvi[i] + (ay*DT*0.5f)); dvz = (zvi[i] + (az*DT*0.5f));
21         // Calcular posición
22         dpx[i] = dvx * DT; dpy[i] = dvy * DT; dpz[i] = dvz * DT;
23         // Actualizar velocidad
24         xvi[i] = dvx; yvi[i] = dvy; zvi[i] = dvz;
25     }
26     // Actualizar posiciones
27     #pragma omp parallel for schedule(static)
28     for(i = 0; i < N; i++){
29         xpos[i] += dpx[i]; ypos[i] += dpy[i]; zpos[i] += dpz[i];
30     }
31 }
```

Fig 22 - Optimizaciones escalares.

Como última consideración de esta sección, la performance entre el uso de sqrt y pow (con exponente fraccionario) puede diferir, dependiendo de la implementación de ciertas instrucciones para estas operaciones en el hardware del procesador. Debido a esto, en el presente trabajo se exploraron ambas alternativas. En la Fig. 23 podemos ver un ejemplo de la sustitución de powf por sqrtf, debido a la equivalencia entre [$x^{3/2}$] y [$x^{1/2} * x^{1/2} * x^{1/2}$].

```
8          // Para cada cuerpo que ejerce una fuerza
9          for (j = 0; j < N; j++){
10             //  Ley de atracción gravitacional de Newton
11             dx = xpos[j] - xpos[i]; dy = ypos[j] - ypos[i]; dz = zpos[j] - zpos[i];
12             dsquared = (dx*dx) + (dy*dy) + (dz*dz) + SOFT;
13             F = G * masses[i] * masses[j]; d12 = 1.0f/sqrtf(dsquared); d32 = d12 * d12 * d12;
14             // Calcular la fuerza total
15             forcesx[i] += F*dx*d32; forcesy[i] += F*dy*d32; forcesz[i] += F*dz*d32;
16         }
17         // Calcular aceleración
18         ax = forcesx[i] / masses[i]; ay = forcesy[i] / masses[i]; az = forcesz[i] / masses[i];
19         // Calcular velocidad
20         dvx = (xvi[i] + (ax*DT*0.5f)); dvy = (yvi[i] + (ay*DT*0.5f)); dvz = (zvi[i] + (az*DT*0.5f));
```

Fig 23 - Alternativa de cálculos con sqrt.

4.1.5 Vectorización

Como se mencionó en la sección 3.3.5.3, la vectorización es una de las técnicas de optimización más importantes para esta arquitectura. Una adecuada vectorización del código permitirá realizar grandes cantidades de operaciones en paralelo. En la Fig. 24 podemos ver el código de esta versión.

```
1  // Para cada instante discreto de tiempo
2  for (t = 1; t <= D; t += DT){
3      // Para cada cuerpo que experimenta una fuerza
4      #pragma omp parallel for schedule(static) private(j)
5      for (i = 0; i < N; i++){
6          // Componentes de la fuerza del cuerpo i
7          forcesx[i] = forcesy[i] = forcesz[i] = 0.0;
8          // Para cada cuerpo que ejerce una fuerza
9          #pragma omp simd aligned
10         for (j = 0; j < N; j++){
11             //  Ley de atracción gravitacional de Newton
12             dx = xpos[j] - xpos[i]; dy = ypos[j] - ypos[i]; dz = zpos[j] - zpos[i];
13             dsquared = (dx*dx) + (dy*dy) + (dz*dz) + SOFT;
14             F = G * masses[i] * masses[j]; d32 = 1/POW(dsquared,1.5);
15             // Calcular la fuerza total
16             forcesx[i] += F*dx*d32; forcesy[i] += F*dy*d32; forcesz[i] += F*dz*d32;
17         }
18         // Calcular aceleración
19         ax = forcesx[i] / masses[i]; ay = forcesy[i] / masses[i]; az = forcesz[i] / masses[i];
20         // Calcular velocidad
21         dvx = (xvi[i] + (ax*DT*0.5)); dvy = (yvi[i] + (ay*DT*0.5)); dvz = (zvi[i] + (az*DT*0.5));
22         // Calcular posición
23         dpx[i] = dvx * DT; dpy[i] = dvy * DT; dpz[i] = dvz * DT;
24         // Actualizar velocidad
25         xvi[i] = dvx; yvi[i] = dvy; zvi[i] = dvz;
26     }
27     // Actualizar posiciones
28     #pragma omp for simd aligned schedule(static)
29     for(i = 0; i < N; i++){
30         xpos[i] += dpx[i]; ypos[i] += dpy[i]; zpos[i] += dpz[i];
31     }
32 }
```

Fig. 24 - Código vectorizado.

El reporte de optimización del compilador ICC permite identificar qué bucles son vectorizados en forma automática. A partir del mismo, se pudo saber que el compilador detecta dependencias falsas en algunas operaciones, imposibilitando la generación de instrucciones SIMD. En consecuencia, para garantizar la vectorización de operaciones se optó por un enfoque guiado a través del uso de la directiva simd de OpenMP 4.0. En particular, los bucles vectorizados son los de las líneas 9 y 28 de la Fig. 22, las directivas simd se insertaron en esas mismas líneas como se puede apreciar en la Fig. 24, siendo la última combinada con la directiva parallel for, como se mencionó en la sección anterior. Por último, para favorecer el uso de instrucciones SIMD, se alinearon los datos a 64 bytes en su alocación, agregando la cláusula aligned a la directiva simd.

4.1.6 Procesamiento por bloques

Para explotar la localidad de datos, es posible implementar un procesamiento por bloques de los cuerpos. Para ello, en primer lugar, se debe iterar sobre el total de cuerpos de a porciones de tamaño fijo denominados bloques, y luego iterar sobre los elementos de ese bloque realizando el procesamiento. Para un rendimiento óptimo, la elección del tamaño de bloque debe considerar las características de la arquitectura,

como pueden ser el tamaño de la memoria cache y las capacidades de las unidades vectoriales.

```
1  // Para cada instante discreto de tiempo
2  for (t = 1; t <= D; t += DT){
3      // Para cada bloque de cuerpos de tamaño BS
4      #pragma omp parallel for schedule(static) private(i,j)
5      for(ii = 0; ii < N; ii+=BS){
6          // Componentes de la fuerza del cuerpo i
7          forcesx[BS] = forcesy[BS] = forcesz[BS] = {0.0};
8          // Para cada cuerpo que ejerce una fuerza
9          for(j = 0; j < N; j++){
10             // Para cada cuerpo que experimenta una fuerza
11             #pragma omp simd aligned
12             for (i = ii; i < ii+BS; i++){
13                 //  Ley de atracción gravitacional de Newton
14                 dx = xpos[j] - xpos[i]; dy = ypos[j] - ypos[i]; dz = zpos[j] - zpos[i];
15                 dsquared = (dx*dx) + (dy*dy) + (dz*dz) + SOFT;
16                 F = G * masses[i] * masses[j]; d32 = 1/POW(dsquared,1.5);
17                 // Calcular la fuerza total
18                 forcesx[i-ii] += F*dx*d32; forcesy[i-ii] += F*dy*d32; forcesz[i-ii] += F*dz*d32;
19             }
20         }
21         #pragma omp simd aligned
22         for (i = ii; i < ii+BS; i++){
23             // Calcular aceleración |M = F * A| ---> |A = F / M|
24             ax = forcesx[i-ii] / masses[i]; ay = forcesy[i-ii] / masses[i]; az = forcesz[i-ii] / masses[i];
25             // Calcular velocidad
26             dvx = (xvi[i] + (ax*DT*0.5)); dvy = (yvi[i] + (ay*DT*0.5)); dvz = (zvi[i] + (az*DT*0.5));
27             // Calcular posición
28             dpx[i] =  dvx * DT; dpy[i] =  dvy * DT; dpz[i] =  dvz * DT;
29             // Actualizar velocidad
30             xvi[i] = dvx; yvi[i] = dvy; zvi[i] = dvz;
31         }
32     }
33     // Actualizar posiciones
34     #pragma omp parallel for simd aligned
35     for(int i = 0; i < N; i++){
36         xpos[i] += dpx[i]; ypos[i] += dpy[i]; zpos[i] += dpz[i];
37     }
38 }
```

Fig. 25 - Código de la versión paralela al aplicar procesamiento por bloques.

En la Fig. 25 se muestra el pseudo-código de la implementación paralela por bloques. El cambio con respecto a la implementación base de la Fig. 17, consiste en desdoblar el bucle i (línea 4) y colocarlo dentro del bucle j (línea 8). En consecuencia, se reemplaza un bucle por otros dos: uno que itera sobre todos los bloques (línea 5) y uno más interno que itera sobre los cuerpos de cada bloque (línea 12).

De esta manera se minimiza el tráfico entre cache y memoria principal al aumentar la cantidad de veces que se usa cada dato en el bucle más interno.

4.1.7 Desenrollado de bucles

El desenrollado de bucles es una técnica de optimización que puede mejorar el rendimiento de un programa, desenrollando las iteraciones de bucles en operaciones individuales, esto puede hacerse manualmente por el programador, o automáticamente por el compilador mediante la directiva unroll.

En particular, se encontró beneficioso desenrollar completamente el bucle de variable j, de la implementación presentada en la Fig. 25 (línea 9), además del que actualiza las posiciones de los cuerpos posteriormente (línea 35). Esto se puede apreciar en la Fig. 26 donde las directivas unroll de openMP aparecen en las líneas 9 y 36.

```
// Para cada instante discreto de tiempo
for (t = 1; t <= D; t += DT){
    // Para cada bloque de cuerpos de tamaño BS
    #pragma omp parallel for schedule(static) private(i,j)
    for(ii = 0; ii < N; ii+=BS){
        // Componentes de la fuerza del cuerpo i
        forcesx[BS] = forcesy[BS] = forcesz[BS] = {0.0};
        // Para cada cuerpo que ejerce una fuerza
        #pragma unroll(BLOCKSIZE)
        for(j = 0; j < N; j++){
            // Para cada cuerpo que experimenta una fuerza
            #pragma omp simd aligned
            for (i = ii; i < ii+BS; i++){
                //  Ley de atracción gravitacional de Newton
                dx = xpos[j] - xpos[i]; dy = ypos[j] - ypos[i]; dz = zpos[j] - zpos[i];
                dsquared = (dx*dx) + (dy*dy) + (dz*dz) + SOFT;
                F = G * masses[i] * masses[j]; d32 = 1/POW(dsquared,1.5);
                // Calcular la fuerza total
                forcesx[i-ii] += F*dx*d32; forcesy[i-ii] += F*dy*d32; forcesz[i-ii] += F*dz*d32;
            }
        }
        #pragma omp simd aligned
        for (i = ii; i < ii+BS; i++){
            // Calcular aceleración |M = F * A| ---> |A = F / M|
            ax = forcesx[i-ii] / masses[i]; ay = forcesy[i-ii] / masses[i]; az = forcesz[i-ii] / masses[i];
            // Calcular velocidad
            dvx = (xvi[i] + (ax*DT*0.5)); dvy = (yvi[i] + (ay*DT*0.5)); dvz = (zvi[i] + (az*DT*0.5));
            // Calcular posición
            dpx[i] =  dvx * DT; dpy[i] =  dvy * DT; dpz[i] =  dvz * DT;
            // Actualizar velocidad
            xvi[i] = dvx; yvi[i] = dvy; zvi[i] = dvz;
        }
    }
    // Actualizar posiciones
    #pragma omp parallel for simd aligned
    #pragma unroll(BLOCKSIZE)
    for(int i = 0; i < N; i++){
        xpos[i] += dpx[i]; ypos[i] += dpy[i]; zpos[i] += dpz[i];
    }
}
```

Fig. 26 - versión con loop unrolling.

4.2 Resultados experimentales

En esta sección se describe el entorno utilizado para las pruebas experimentales, las pruebas que se realizaron y los resultados obtenidos.

4.2.1 Plataforma.

Todas las pruebas fueron realizadas en un sistema equipado con un Intel Xeon Phi 7230 de 68 núcleos (que además cuenta con 4 hilos hardware por núcleo), 192 GB de memoria RAM y 16 GB de memoria MCDRAM. En ese sentido, el procesador fue usado en modo cluster All to all y la memoria MCDRAM en modo Flat. Se eligió este modo de cluster en particular, debido a que es el único disponible cuando las memorias RAM son de diferente tamaño. Respecto al software, el sistema operativo es Ubuntu 16.04.3 LTS mientras que el compilador es el ICC (versión 19.0.0.117).

4.2.2 Pruebas realizadas.

Para las pruebas se partió de una versión base del algoritmo, y se analizó el impacto de aplicar cada optimización incrementalmente sobre el mismo. Se evaluaron múltiples optimizaciones específicas, incluidas en las optimizaciones mencionadas en la sección

4.1. Estas optimizaciones se fueron aplicando una por una en archivos fuente distintos, resultando en 10 archivos fuente que se compilaron a 14 ejecutables distintos, estas versiones fueron numeradas del 0 a 13, siendo 0 la versión base secuencial y 10 la versión más optimizada. Las versiones 11-13 representan diferentes variantes de la versión 10, considerando cambios en el tipo de datos o la precisión en el cómputo de operaciones con flotantes. Además de las optimizaciones algunas versiones difieren en el tipo de datos, variando entre double o float. En la Fig. 27 podemos ver el detalle de las diferentes versiones.

Versión	Multithreading	Vectorización	Procesamiento por bloques.	Alineación de datos.	Desenrollado de bucles.	Tipo de dato.	Flag fp mode	Descripción
0	No	No	No	No	No	Float	No	Versión secuencial.
1	Sí	No	No	No	No	Float	No	Versión paralela base.
2	Sí	No	No	No	No	Float	No	Optimización constantes y funciones float.
3	Sí	No	No	No	No	Float	No	Optimización divisiones float.
4	Sí	No	No	No	No	Float	No	Sqrt en lugar de pow.
5	Sí	SSE	No	No	No	Float	No	Vectorizada SSE
6	Sí	AVX-2	No	No	No	Float	No	Vectorizada AVX2
7	Sí	AVX-512	No	No	No	Float	No	Vectorizada AVX-512
8	Sí	AVX-512	No	Sí	No	Float	No	Vectorizada + alineación.
9	Sí	AVX-512	Sí	Sí	No	Float	No	Versión por bloques.
10	Sí	AVX-512	Sí	Sí	Sí	Float	No	Versión desenrollado de bucles.
11	Sí	AVX-512	Sí	Sí	Sí	Float	Sí	Desenrollado + fp-mode.
12	Sí	AVX-512	Sí	Sí	Sí	Double	No	Double.
13	Sí	AVX-512	Sí	Sí	Sí	Double	Sí	Double + fp-mode.

Fig. 27 - Detalle de las optimizaciones.

En tiempo de compilación, para utilizar las instrucciones vectoriales AVX2 (versión 6) y AVX-512 (versiones 7-13), se emplearon los flags -xAVX2 y - xMIC-AVX512, respectivamente, mientras que para acelerar el cómputo de operaciones en punto flotante se usó el flag -fp-mode fast=2 (versiones 11 y 13). Además, se empleó el comando numactl para poder explotar la memoria MCDRAM (no requiere modificaciones al código fuente).

A la hora de realizar las pruebas se utilizó la cantidad fija de 100 iteraciones y se varió el número de threads T={64, 128, 192, 256} para las versiones paralelas. Por último, se introdujo variación en la carga de trabajo al usar diferentes números de cuerpos: N = {65536, 131072, 262144, 524288, 1048576}

4.2.3 Resultados.

Para evaluar el rendimiento se emplea la métrica GFLOPS (mil millones de FLOPS), utilizando la fórmula mostrada en la Fig. 28. Donde N es el número de cuerpos, I es el número de pasos, T es el tiempo de ejecución (en segundos) y el factor 20 representa la cantidad de operaciones en punto flotante requerida por cada interacción[11].

$$GFLOPS = \frac{20 * N^2 * I}{T * 10^9}$$

Fig. 28 - Fórmula del cálculo de GFLOPS.

En la Fig. 29 se pueden ver los rendimientos al variar el tipo de afinidad y el número de hilos cuando N=65536. Al emplear paralelismo a nivel de hilos, el rendimiento mejora considerablemente, notando que un mayor número de hilos lleva a mejores prestaciones (excepto con afinidad scatter).

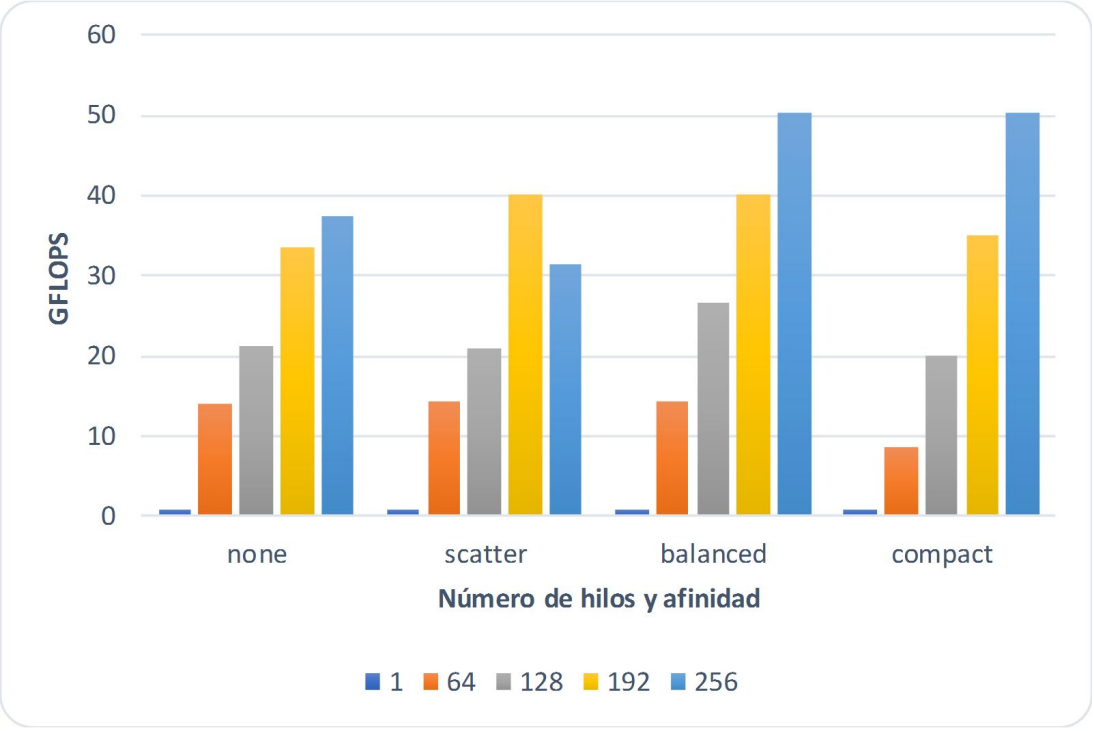

Fig. 29 - Rendimientos obtenidos para diferentes tipos de afinidad y número de hilos cuando N=65536.

[11] Convención ampliamente aceptada en la literatura disponible sobre este problema.

Respecto a la afinidad, se puede apreciar que resulta conveniente elegir alguna de las estrategias disponibles en lugar de delegar la distribución en el sistema operativo (none). A diferencia de scatter, balanced y compact garantizan la contigüidad de hilos OpenMP con identificadores consecutivos, lo que favorece a la comunicación de los datos que cada uno requiere[12].

En cuanto a las optimizaciones escalares, podemos ver el resultado de aplicar la primera optimización en la Fig. 30. Al eliminar las conversiones innecesarias, podemos observar una pequeña mejora de hasta aproximadamente un 14%.

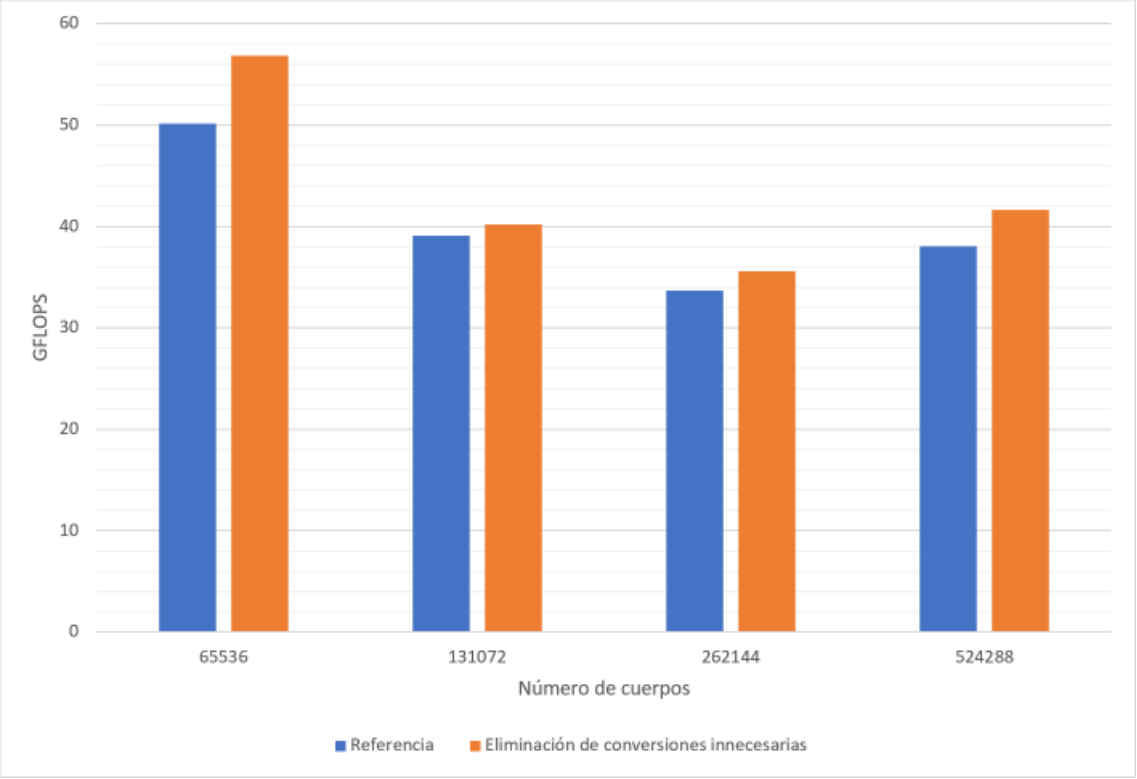

Fig. 30 - Rendimientos para las distintas optimizaciones escalares.

En cuanto al resto de las optimizaciones escalares, podemos observar el resultado de las mediciones en la Fig. 31. En primer lugar, cabe mencionar que las 3 versiones sujetas a comparación, implementan la eliminación de las conversiones innecesarias mostrada en la Fig. 30. Se tomó como referencia la versión que incluye la primera optimización, para poder mostrar que entre las 3 distintas optimizaciones no hay diferencias de rendimiento significativas. Este hecho se debe principalmente a que el compilador puede estar realizando alguna de estas optimizaciones de forma automática, invisibilizando los cambios en el código de alto nivel.

[12] Como todos los núcleos del procesador se encuentran en el mismo paquete, balanced y compact producen la misma asignación cuando se emplean 4 hilos por núcleo.

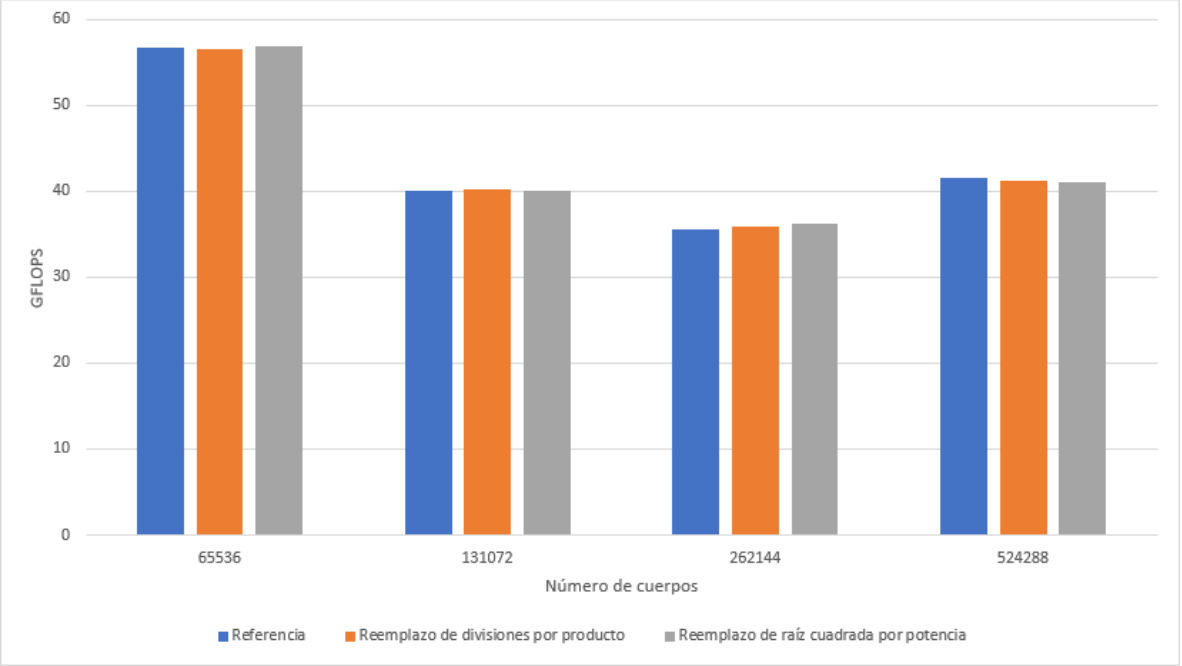

Fig. 31 - Comparación de optimizaciones escalares.

Como se mencionó previamente, el compilador no es capaz de vectorizar todas las operaciones por su cuenta. Se puede notar en la Fig. 32 que, al forzar la vectorización de operaciones, se produce una mejora de aproximadamente 3.9×.

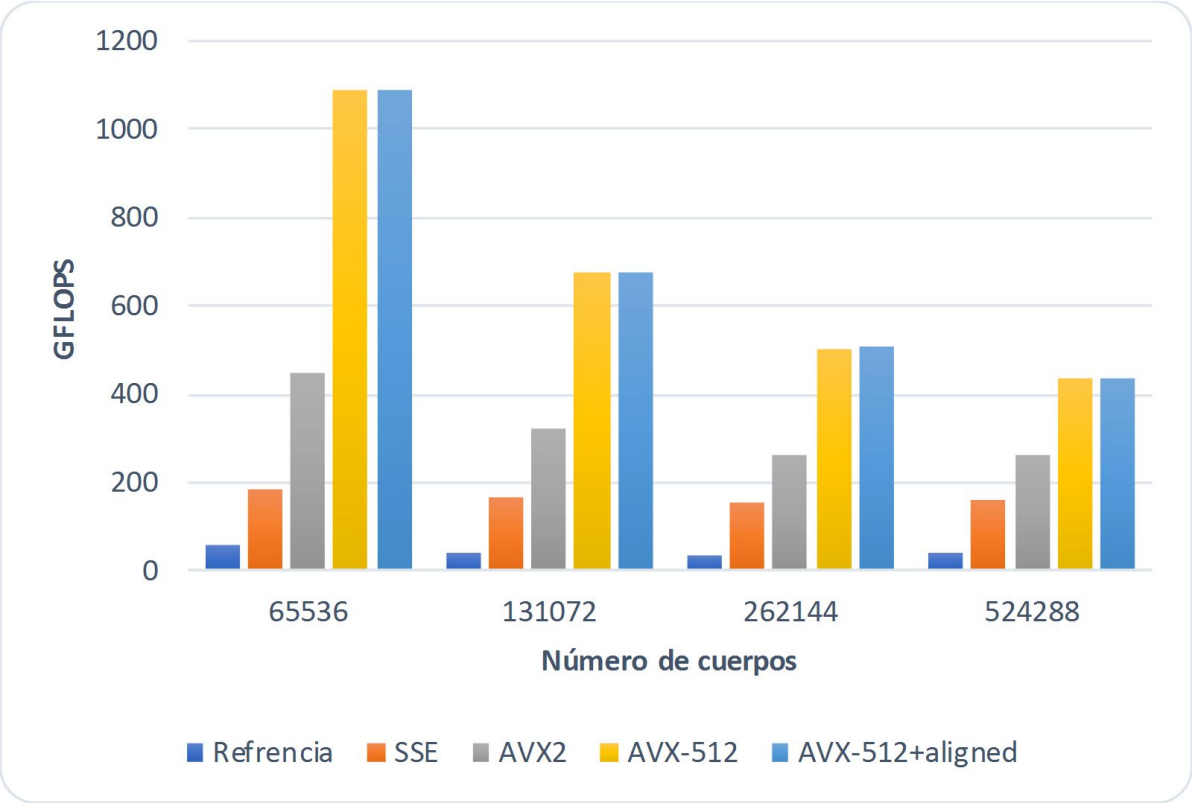

Fig. 32 - Rendimientos obtenidos para diferentes niveles de vectorización al variar el número de cuerpos (N).

Al agregar los flags -xAVX2 y -xMIC-AVX512, el compilador emplea las instrucciones AVX2 y AVX-512, respectivamente. Como estas extensiones tienen mayor ancho vectorial, el rendimiento se incrementa aún más, logrando mejoras de 7.4× para AVX2 y de 15.1× para AVX-512. Por lo tanto, resulta claro que este problema se beneficia de instrucciones vectoriales más anchas. Adicionalmente, no se observan mejoras significativas por el alineamiento de datos a memoria.

Como se puede apreciar en la Fig. 33, la técnica de procesamiento por bloques aumenta considerablemente el rendimiento, siendo mayor la ganancia a medida que la cantidad de cuerpos crece. En particular, se logra una mejora promedio de 2.9× y una máxima de 4.1× (con BS=16, siendo BS el tamaño del bloque). En forma adicional, las prestaciones mejoran aproximadamente un 9% al desenrollar los bucles mencionados en la sección 4.1.6.

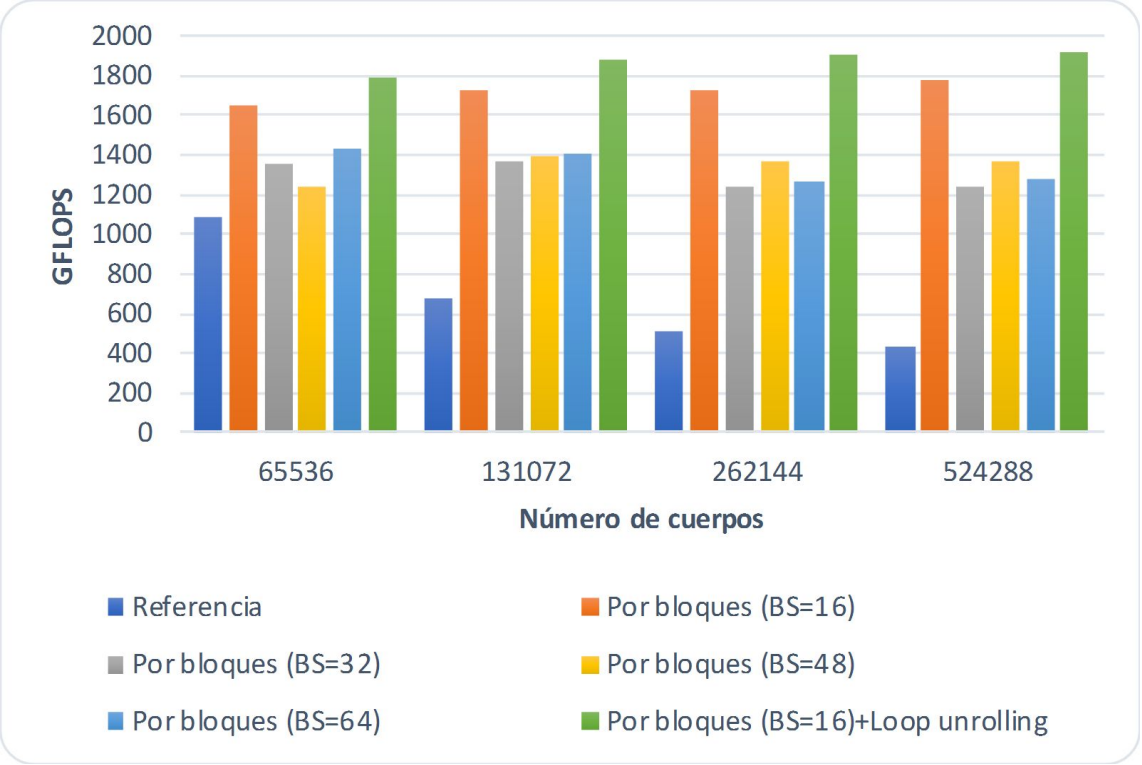

Fig. 33 - Rendimientos obtenidos para la técnica de bloques y desenrollado de bucles al variar la carga de trabajo (N).

En la Fig. 34 se muestran los rendimientos obtenidos para la relajación de precisión al variar el tipo de dato y la carga de trabajo (N). Se puede notar que las prestaciones mejoran un 22% en promedio al aplicar la optimización del compilador para este fin. En sentido contrario, el rendimiento decae aproximadamente un 60% al duplicar la precisión numérica usando el tipo de dato double.

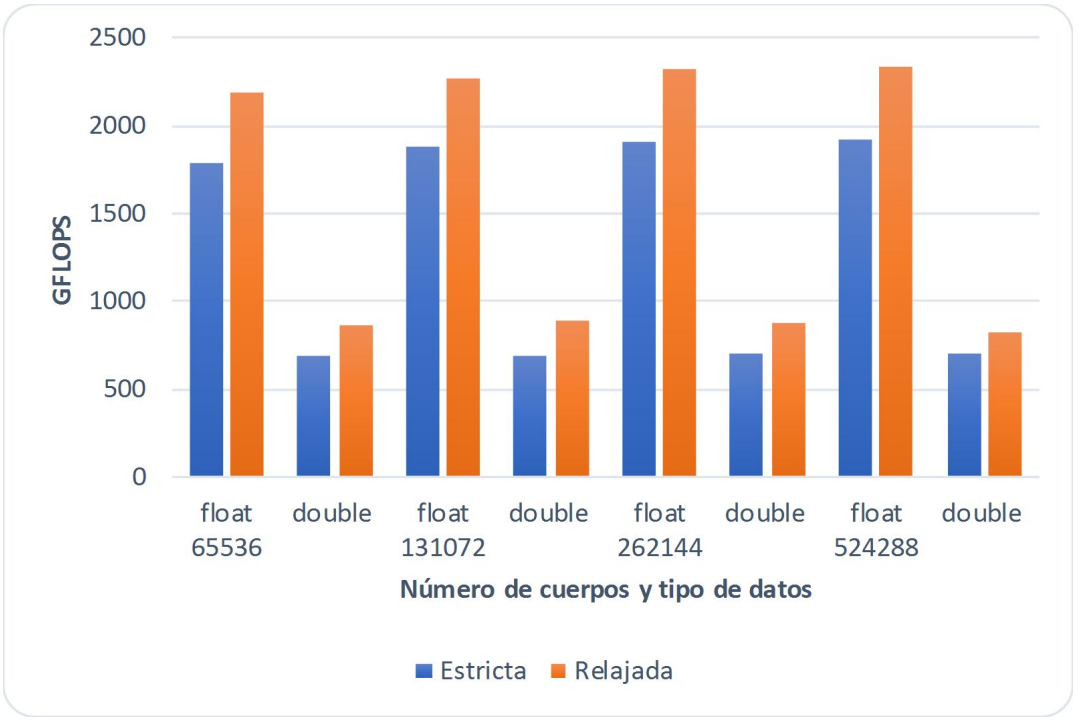

Fig. 34 - Rendimientos obtenidos para la relajación de precisión al variar el tipo de dato y la carga de trabajo (N).

Por último, a partir de la Fig. 35, se puede notar que el rendimiento se mantiene (casi) constante al aumentar el número de cuerpos, obteniendo como máximo 2355 GFLOPS. También se puede apreciar una pequeña mejora cercana al 2% por el uso de la memoria MCDRAM. Este resultado es similar al observado en [23], estando relacionado con el hecho de que el rendimiento de esta aplicación se ve más influenciado por la latencia de la memoria que por el ancho de banda.

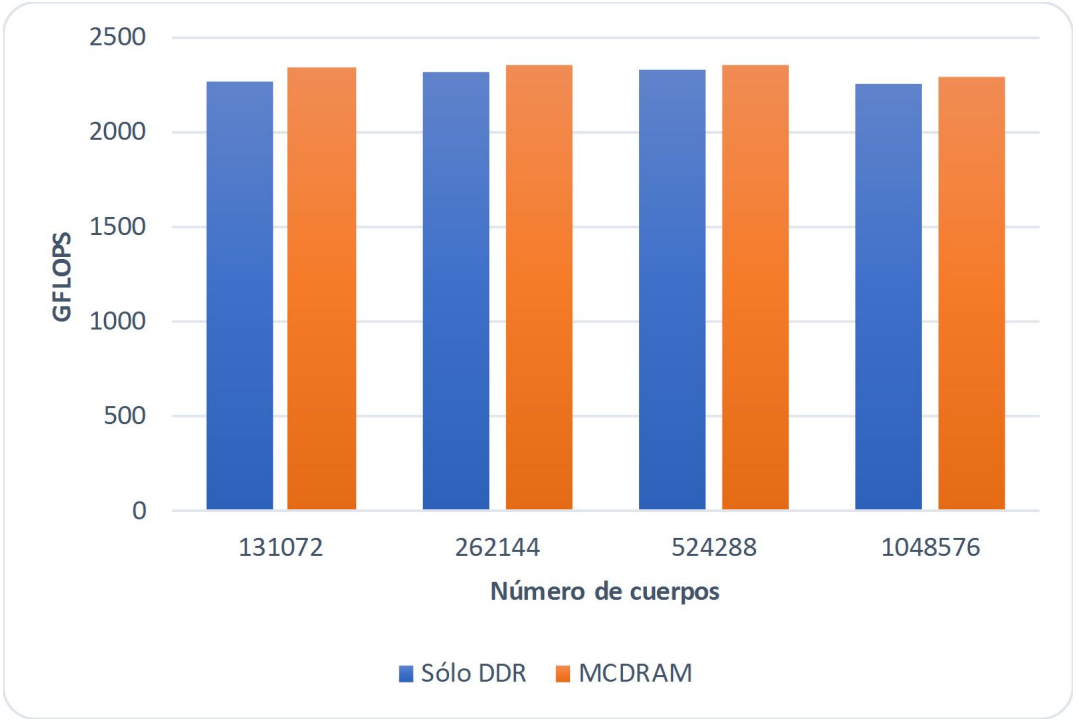

Fig. 35 - Rendimientos obtenidos para la explotación de MCDRAM al variar la carga de trabajo

4.3 Trabajos relacionados

La aceleración de la simulación de N cuerpos ha sido ampliamente estudiada en la literatura. Sin embargo, pocos trabajos lo hacen sobre la arquitectura Xeon Phi, usando en su mayoría la primera generación de esta familia (KNC) [6] [24] [25]. Con respecto a KNL, sólo se puede mencionar el trabajo [23], el cual presenta algunas similitudes y diferencias con la presente investigación. Al igual que en este artículo, los autores estudian la paralelización de la versión directa de la simulación mostrando las mejoras obtenidas a través de diferentes optimizaciones, aunque la implementación final alcanza un pico de rendimiento mayor (2875 GFLOPS). A diferencia de esta investigación, el trabajo prioriza las optimizaciones para KNL, haciendo algunas simplificaciones en el cálculo de la simulación, como emplear un mecanismo de integración más sencillo (menos operaciones) y sólo computar un único paso en el tiempo. Al usar la misma métrica de evaluación[13], sobreestiman la cantidad de FLOPS obtenidos. En relación con el análisis de prestaciones, en este artículo se consideraron varios aspectos adicionales como la cantidad y la afinidad de los hilos, el uso de todos los conjuntos de instrucciones vectoriales del KNL, la búsqueda del tamaño de bloque óptimo, el impacto del aumento de precisión por el uso del tipo double, además de la variación en la cantidad de cuerpos.

4.4 Resumen

Para la implementación de optimizaciones al problema de los N cuerpos sobre Intel Xeon Phi Knights Landing del presente trabajo, se comenzó a trabajar sobre una implementación secuencial del método directo de N cuerpos. Luego de la implementación secuencial se realizó un análisis de los puntos a tener en cuenta a la hora de paralelizar el algoritmo. El análisis de paralelismo, llevó a una primera implementación paralela utilizando OpenMP. Luego, se aplicaron diferentes optimizaciones sobre esta versión en forma incremental, como las de vectorización, procesamiento por bloques y desenrollado de bucles.

Para las pruebas experimentales se tomaron como variables la cantidad de hilos y afinidad de los mismos, el tamaño del problema expresado como la cantidad de cuerpos N y la precisión de los datos y operaciones. Se tomaron mediciones con diferentes valores para estas variables. Además de estos se hicieron pruebas con y sin el uso de la memoria MCDRAM. La métrica elegida para las pruebas experimentales fue la de GFLOPS, es decir la cantidad en miles de millones de operaciones de punto flotante por segundo.

Entre los resultados más destacados podemos mencionar la mejora lograda con 256 hilos mediante el uso de los modos de afinidad balanced y compact; el notable incremento de rendimiento por el uso de instrucciones AVX-512 por sobre las escalares y aprovechamiento de la localidad de datos mediante el procesamiento por bloques; y la gran diferencia entre el uso de datos de simple precisión para los cálculos

[13] También asumen que se realizan 20 operaciones de punto flotante por interacción pero el número real es inferior por usar un esquema de integración más simple (menos operaciones).

contra los de doble precisión (más del doble) así como la mejora por el uso de precisión relajada en los cálculos. Por último, resulta importante destacar el pico de rendimiento de 2355 GFLOPS logrado.

El presente trabajo se destaca de otros similares por diferentes cuestiones. En primer lugar, se realizó una implementación del algoritmo de N cuerpos con atracción gravitacional a diferencia de un método genérico de N cuerpos que requiere menos operaciones. Adicionalmente, se utilizó un integrador numérico más preciso que requiere realizar más cálculos. Finalmente, en cuanto al análisis de prestaciones, se consideraron varias cuestiones adicionales como la afinidad de hilos, el uso de los distintos conjuntos de instrucciones vectoriales, la búsqueda del tamaño de bloque óptimo y el impacto de la utilización de doble precisión en comparación con simple precisión en los cálculos de punto flotante

5. Conclusiones y trabajos futuros

En la comunidad HPC, el uso de aceleradores se ha consolidado como estrategia para mejorar el rendimiento de los sistemas al mismo tiempo que la eficiencia energética. Recientemente, Intel introdujo Knights Landing (KNL), la segunda generación de aceleradores Xeon Phi. Entre sus características destacadas, se puede mencionar su gran cantidad de núcleos, la incorporación de las instrucciones vectoriales AVX-512 y la integración de una memoria de alto ancho de banda.

Un área afectada por la necesidad de resolución eficiente de problemas con alta demanda computacional es la física, donde se sitúa el problema de la simulación de N cuerpos computacionales. Estas simulaciones se han convertido en una gran herramienta para entender las interacciones gravitacionales entre partículas que van desde pocos cuerpos hasta galaxias completas. Debido a esto, el presente trabajo se enfoca en la optimización de la simulación de N cuerpos computacionales con atracción gravitacional sobre un acelerador Intel Xeon Phi KNL.

Para lograr una solución optimizada, se comenzó por una versión secuencial a la que se le fueron aplicando incrementalmente diferentes técnicas de optimización, en particular considerando aspectos específicos de la arquitectura de soporte.

En el análisis realizado sobre los resultados obtenidos de las diferentes pruebas se concluyó que:

- En general, aumentar el número de hilos llevó a un mejor rendimiento.
- En cuanto a la afinidad de hilos, se encontraron diferencias significativas entre las distintas opciones, por lo que es un factor que no se debe ignorar al momento de ejecutar una aplicación paralela.
- La vectorización de operaciones representó un factor fundamental para la mejora de rendimiento. En ese sentido, se lograron aceleraciones cercanas al número de operaciones simultáneas que cada repertorio SIMD permite, a un bajo costo de programación.
- La explotación de la localidad de datos mediante el procesamiento por bloques resultó otro aspecto clave para obtener alto desempeño. No sólo permitió incrementar los FLOPS obtenidos en cada caso sino también que el rendimiento escale al aumentar la cantidad de cuerpos.
- Cuando la precisión en los resultados finales, no es una prioridad, tanto el uso de los tipos de datos de simple precisión, como las opciones del compilador que permiten relajar la precisión de las operaciones matemáticas pueden ofrecer mejoras significativas en el rendimiento. En sentido contrario, duplicar la precisión puede reducir el rendimiento por debajo de la mitad.

- El uso de la memoria MCDRAM puede no proveer mejoras significativas en el rendimiento, especialmente cuando la aplicación no tiene una alta demanda de ancho de banda.

A partir de los resultados obtenidos, se considera que se ha cumplido con el objetivo planteado inicialmente. Partiendo de una implementación secuencial, se han aplicado y analizado diferentes técnicas de optimización que permiten obtener una solución de alto rendimiento para el problema de estudio sobre los procesadores Xeon Phi KNL. La implementación desarrollada es capaz de alcanzar un pico de rendimiento de 2355 GFLOPS y se encuentra disponible para beneficio de la comunidad científica.

Entre los trabajos futuros, se pueden mencionar dos posibles líneas de investigación:

- Considerando los resultados obtenidos, se espera avanzar en la implementación de métodos avanzados para esta simulación. También en el desarrollo de soluciones optimizadas para otras simulaciones de la física y áreas afines sobre multiprocesadores.

- Dado que las GPUs son el acelerador dominante en la actualidad, interesa realizar una comparación de rendimiento y eficiencia energética entre estas arquitecturas.

6. Referencias

[1] M. B. Giles y I. Reguly, «Trends in high-performance computing for engineering calculations», *Phil. Trans. R. Soc. A*, vol. 372, n.º 2022, p. 20130319, ago. 2014, doi: 10.1098/rsta.2013.0319.

[2] J. Jeffers, J. Reinders, y A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[3] G. R. Andrews, *Foundations of multithreaded, parallel, and distributed programming*. Reading, Mass: Addison-Wesley, 2000.

[4] P. Freddolino, C. Harrison, Y. Liu, y K. Schulten, «Challenges in protein folding simulations: Timescale, representation, and analysis. Nature Physics, 6, 751», *Nature physics*, vol. 6, pp. 751-758, oct. 2010, doi: 10.1038/nphys1713.

[5] R. G. Guide y S. Chandran, «Global Illumination for Point Models», *Future Work*, p. 75.

[6] R. Yokota y M. Abduljabbar, «N-Body methods.», en *High Performance Parallelism Pearls - Multicore and Many-core Programming Approaches*, Morgan-Kaufmann, 2015, pp. 175-183.

[7] P. Young, «Physics 115/242 The leapfrog method and other "symplectic" algorithms for integrating Newton's laws of motion», [En línea]. Disponible en: http://physics.ucsc.edu/~peter/242/leapfrog.pdf.

[8] A. Cromer, «Stable solutions using the Euler approximation», *American Journal of Physics*, vol. 49, n.º 5, pp. 455-459, may 1981, doi: 10.1119/1.12478.

[9] M. Tuckerman, B. J. Berne, y G. J. Martyna, «Reversible multiple time scale molecular dynamics», *J. Chem. Phys.*, vol. 97, n.º 3, pp. 1990-2001, ago. 1992, doi: 10.1063/1.463137.

[10] J. Barnes y P. Hut, «A hierarchical O(N log N) force-calculation algorithm», *Nature*, vol. 324, n.º 6096, pp. 446-449, dic. 1986, doi: 10.1038/324446a0.

[11] L. Greengard y V. Rokhlin, «A Fast Algorithm for Particle Simulations», p. 13, jun. 1986.

[12] John Hennessy y David Patterson, *Computer Architecture - 6th Edition*, 6th Edition. Morgan Kaufmann, 2017.

[13] V. Eijkhout, E. Chow, y R. van de Geijn, «Introduction to High Performance Scientific Computing», p. 535, 2014.

[14] «TOP500 List - November 2019 | TOP500 Supercomputer Sites». https://www.top500.org/list/2019/11/ (accedido abr. 27, 2020).

[15] J.R. Goodman y H.H.J. Hum, «MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects (2004)», nov. 2004, [En línea]. Disponible en: https://www.cs.auckland.ac.nz/~goodman/TechnicalReports/MESIF-2004.pdf.

[16] Keith Diefendorff, «MICROPROCESSOR REPORT. THE INSIDERS' GUIDE TO MICROPROCESSOR HARDWARE», vol. 13, n.º 3, mar. 1999, [En línea]. Disponible en: http://docencia.ac.upc.edu/ETSETB/SEGPAR/microprocessors/pentium3%20(mpr).pdf.

[17] David Kanter, «Intel's Sandy Bridge Microarchitecture», *Real world technologies*, sep. 25, 2010. https://www.realworldtech.com/sandy-bridge/6/ (accedido dic. 20, 2019).

[18] A. Grama, Ed., *Introduction to parallel computing*, 2nd ed. Harlow, England ; New York: Addison-Wesley, 2003.

[19] D. T. Marr *et al.*, «Hyper-Threading Technology Architecture and Microarchitecture», p. 12, 2002.

[20] «Thread Affinity Interface». https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-

developer-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-library-support/thread-affinity-interface-linux-and-windows.html (accedido may 19, 2020).
[21] «Data Alignment to Assist Vectorization». https://software.intel.com/content/www/us/en/develop/articles/data-alignment-to-assist-vectorization.html (accedido may 19, 2020).
[22] corob-msft, «Function Overloading». https://docs.microsoft.com/en-us/cpp/cpp/function-overloading (accedido may 21, 2020).
[23] A. Nikolaev y R. Asai, «N-Body simulation.», en *Intel Xeon Phi Processor HPC - KNL edition.*, Morgan Kaufmann, 2016, p. 638.
[24] Andrey Vladimirov y Vadim Karpusenko, «Test-driving Intel R Xeon PhiTM coprocessors with a basic N-body simulation», p. 15.
[25] B. Lange y P. Fortin, «Parallel Dual Tree Traversal on Multi-core and Many-core Architectures for Astrophysical N-body Simulations», en *Euro-Par 2014 Parallel Processing*, Cham, 2014, pp. 716-727.