Facultad
de
Ciencias

Optimización de aplicaciones y equilibrio de carga en sistemas heterogéneos con Xeon Phi
(Application performance optimization and load balancing in Xeon Phi based heterogeneous systems)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Adrián Herrera Arcila

Director: José Luis Bosque Orero

Septiembre - 2017

Índice general

Índice de figuras

Índice de cuadros

Agradecimientos

A mis padres por confiar en mí, dejarme tanto acertar como errar en mis decisiones y aprender del proceso; por apoyarme, animarme y aguantarme día tras día; por subirme a hombros cuando quiero tocar las nubes y por bajarme al suelo suavemente cuando me doy cuenta de que no puedo.

A mis amigos por estar siempre ahí para compartir y disfrutar momentos juntos, para ayudarme cuando lo he necesitado y para abstraerme de las dificultades.

A todos los profesores que me han enseñado a apreciar el aprendizaje durante estos años y que se han esforzado en transmitirme su pasión, especialmente a José Luis Bosque, director de esta investigación que ha confiado en mí para llevarla a cabo, me ha guiado en los momentos en los que no veía un camino a seguir y me ha inspirado para esforzarme en no dejar de creer en ella.

Finalmente a todas las personas que se levantan por la mañana con la intención de ayudar y aportar a la sociedad, ya sea a su familia, a su comunidad o a su país, gracias a ellos creo en el progreso de la humanidad y estoy feliz de sumarme a través de mi pasión mientras sea útil.

Resumen

La curiosidad del ser humano es innata e infinita, para muchos forma parte del sentido de la vida, pero a medida que se exploran y resuelven misterios, surgen otros de mayor magnitud y complejidad; hoy en día muchos de los estudios e investigaciones que tratan de arrojar luz sobre estos problemas requieren herramientas software extremadamente complejas (simulaciones, análisis numérico, predicciones, ...), y por consiguiente tienen asociado un coste computacional alto.

Para enfrentarse a ello, los sistemas de alto rendimiento han pasado de un modelo de potencia bruta a uno colaborativo, en el que diversos componentes hardware trabajan de manera distribuida para ejecutar esas herramientas, asumiendo de forma paralela el coste total.

En este sentido han surgido nuevos dispositivos hardware específicamente diseñados para este modelo; en concreto, Intel ha desarrollado el Xeon Phi, un acelerador de decenas de núcleos de proceso que se encarga de tareas con un alto grado de paralelismo.

Esta investigación trata de informar del valor de esta pieza para las necesidades actuales y de cómo optimizar las aplicaciones para ejecutar sobre ella y sacar el máximo rendimiento; además se introducirán una serie de métricas, métodos y algoritmos para aprovechar el poder computacional combinado en un entorno heterogéneo compuesto por un procesador tradicional Xeon y este acelerador, un proceso conocido como equilibrio de carga.

En este entorno se realizarán un conjunto de pruebas sobre dos aplicaciones típicas en la computación de alto rendimiento, la simulación N-body y el filtrado de imágenes, en las que se pondrán en práctica los métodos desarrollados.

Por último, se analizarán los resultados de estas pruebas, donde serán explicadas las mejoras producidas por cada uno de ellos sobre los programas elegidos, así como las principales razones que han conducido a esos incrementos de rendimiento.

Palabras clave: paralelismo, acelerador, coste o carga computacional, sistema heterogéneo, equilibrio de carga, Intel Xeon Phi.

Abstract

Human beings' curiosity is inborn and infinite, for many it takes part in their meaning of life conception, but while we get further through solving unanswered questions and mysteries, more and more versions of these pop up with higher degree of both scale and complexity; nowadays a considerable fraction of the studies and investigations going on require extremely complex software tools (simulations, numerical analysis, forecasting, ...), which in turn have a huge computational cost associated to them.

In order to face this issue, high performance systems have transitioned from a pure power model to a rather colaborative one, in which big numbers of diverse hardware subsystems work in a distributed way to execute these mentioned tools, assuming the total cost in a parallel manner.

Having this into account, new specific hardware for this model is being created; in particular, Intel has developed the Xeon Phi, an accelerator possesing dozens of processing cores which takes care of applications with high degrees of parallelism.

This investigation tries to transmit the value of the Xeon Phi for today's needs, and gives advise on how to optimize software for it, squeezing its performance to higher levels; moreover a set of metrics, methods and algorithms will be introduced to exploit the combined computational power in a heterogeneous environment formed by a traditional Xeon processor and this accelerator, a process being known as load balancing.

Within this environment, a test set will be applied to two main applications in high performance computing, N-body simulation and image filtering, this set reflecting the implementation of previously introduced methods.

At last, the results of the tests will be analysed, and both the improvements in performance due to these methods as well as the reasons driving to them will be explained.

Keywords: parallelism, accelerator, computational cost or complexity, heterogeneous system, load balancing, Intel Xeon Phi.

Capítulo 1

Introducción

1.1. Sistemas heterogéneos

La ley de Moore expresa que el número de transistores en un microchip dobla su magnitud cada dos años, aproximadamente; las consecuencias de esto son, entre otras, las mejoras a nivel de microarquitectura, como el aumento de componentes (puertas lógicas, registros, contadores, ...) o la optimización de su distribución; esto, a su vez, tiene un impacto en el rendimiento del hardware, que es capaz de producir mejores resultados para mismas cargas.

Normalmente este efecto se produce por la reducción del tamaño del transistor y por consiguiente el aumento de su densidad en el circuito, no obstante los límites físicos están frenando la tendencia; con tamaños de 10 nanómetros planificados para 2017, cada vez son más notables los problemas asociados con esta métrica, como el filtrado de energía o la complejidad de manufacturación; por otro lado el consumo energético está conteniendo el incremento de las frecuencias de reloj en los procesadores, uno de los parámetros fundamentales de mejora de rendimiento.

A raíz de esto se han generado una serie de conceptos y técnicas para sobrepasar estas limitaciones, que radican en dos principios fundamentales: la división y la simultaneidad en el procesado de la carga computacional.

Desde el procesador multinúcleo hasta los sistemas heterogéneos, la transición hacia ejecuciones distribuidas y colaborativas de los programas ha supuesto una serie de retos a los programadores; entre ellos está la necesidad de adaptar sus aplicaciones y, aún más importante, su concepción de la programación.

Particularmente en los sistemas heterogéneos, los componentes hardware participantes presentan diferencias a nivel arquitectural, cuyas consecuencias impactan negativamente tanto a la compatibilidad de los modelos de programación como al propio proceso de compilación (diferentes arquitecturas, distintas instrucciones).

Una combinación especialmente típica en estos sistemas es la utilización de un acelerador para complementar a la unidad de proceso principal; el acelerador normalmente se encarga de tareas con alto grado de paralelismo en su ejecución, ya que su característica fundamental es el elevado número de núcleos de proceso; estos núcleos son distintos de los de un procesador tradicional, individualmente son menos potentes, pero al ser más tienden a generar mayor rendimiento cuando el procesado es divisible.

Para paliar el sobrecoste temporal y económico de introducir nuevas metodologías de programación en pos de obtener beneficios de estos componentes, Intel ha lanzado un tipo de acelerador

denominado Xeon Phi, que permite a los programadores utilizar sus paradigmas tradicionales para conseguir esa mejora de rendimiento.

1.2. Objetivos y plan de trabajo

Esta investigación se ha centrado en la programación de un entorno heterogéneo compuesto por un procesador Intel Xeon y un acelerador Intel Xeon Phi; la idea principal es caracterizar las capacidades del acelerador, en qué aspectos destaca y la importancia de su correcta programación; además se introducen mecanismos para la repartición o equilibrio de carga entre ambos componentes, comparando el rendimiento con el de la ejecución individual y señalando los sobrecostes de comunicación y sincronización.

El enfoque es fundamentalmente didáctico, de tal forma que el lector pueda observar no sólo las consecuencias de la utilización de un sistema heterogéneo, sino el motivo de los métodos de programación, optimización y equilibrio utilizados, así como el tipo de aplicaciones a las que está destinado cada uno de ellos.

Los objetivos concretos del proyecto han sido los siguientes:

- Optimización de aplicaciones para el acelerador Intel Xeon Phi, mejora de rendimiento de programas existentes a través de las características arquitecturales de este componente; a partir de la implementación secuencial de un algoritmo, adaptar el código mediante modelos de programación tradicionales para conseguir un rendimiento óptimo en el acelerador.

- Implementación de mecanismos para equilibrar la carga computacional en un sistema heterogéneo compuesto por un procesador Xeon y un acelerador Xeon Phi; a través de la descarga de datos, conseguir la co-ejecución de un programa en ambos componentes, mejorando aún más los resultados de la implementación optimizada; cada uno de estos mecanismos estará enfocado a un tipo de aplicación concreto para el cual ofrecerá la máxima efectividad.

Para conseguirlos, se ha llevado a cabo el siguiente plan de trabajo:

- Recopilación de información sobre el Intel Xeon Phi, su arquitectura, modelos de programación y diferencias claves con los procesadores multinúcleo tradicionales.

- Exposición de los principales aspectos a tener en cuenta a la hora de optimizar los programas para su utilización en el acelerador Xeon Phi, en concreto la paralelización y la vectorización de las fases computacionales.

- Introducción al equilibrio de carga, clasificación en dos grandes grupos, carga regular e irregular, con ejemplos de aplicaciones concretas; identificación de métodos específicos para cada uno de ellos, diferencias, ventajas y desventajas.

- Desarrollo de aplicaciones bajo un modelo de programación de memoria compartida compatible entre el procesador y el acelerador, OpenMP; implementación de los métodos previamente introducidos, utilizando el modelo de descarga u *offload* para la repartición de la carga computacional.

- Pruebas sobre el sistema real, generación de resultados y análisis de estos para la verificación y validación de la viabilidad en los métodos aplicados; rendimiento y *speedups* conseguidos.

1.3. Estructura del documento

Además del actual capítulo de introducción, el documento consta de 5 capítulos adicionales configurados según la siguiente estructura:

- Capítulo 2: **Estado del arte**. Se realiza una primera exposición de los conceptos reflejados durante el resto del proyecto: tendencias y necesidades en la computación de alto rendimiento, aceleradores y el Xeon Phi, aspectos arquitecturales, modelos de programación y algoritmos de equilibrio de carga.

- Capítulo 3: **Optimización de aplicaciones para el Xeon Phi**. Se profundiza en los principales factores a tener en cuenta a la hora de optimizar las aplicaciones para el Xeon Phi, en concreto se explican los fundamentos de la paralelización y la vectorización, así como los recursos prácticos (opciones de compilación, directivas, variables de entorno) necesarios para activarlos y utilizarlos eficazmente; además, se destacan algunos puntos importantes adicionales que mejorarán el rendimiento de las aplicaciones; finalmente se expone un ejemplo práctico que resume el capítulo e integra los métodos introducidos.

- Capítulo 4: **Algoritmos de equilibrio de carga**. Explicación del concepto de equilibrio de carga, qué consecuencias tiene (sobrecostes de comunicación, diferencia temporal, ...) y división de la carga en dos grandes grupos, regular e irregular; se presentan tres métodos para intentar obtener el ratio óptimo de división de trabajo: un análisis estático mediante interpolación de las curvas de coste computacional, el método de bisección aplicado al ratio, y un algoritmo dinámico de partición y distribución de segmentos a procesar en tiempo de ejecución; se discutirá el tipo de carga al que estará dirigido cada uno y los posibles resultados de su implementación.

- Capítulo 5: **Evaluaciones y resultados**. Una breve introducción al entorno real de pruebas con sus características más importantes; se detalla el desarrollo e implementación de todos los métodos y algoritmos mencionados en los capítulos 3 y 4 para un conjunto de aplicaciones concretas, señalándose los puntos clave del código y las herramientas utilizadas (Intel Language Extensions for Offload, OpenMP, compilador ICC, ...); finalmente se realiza un grupo de pruebas basadas en estas implementaciones, y se analizan los resultados obtenidos, contrastando su viabilidad y efectividad.

- Capítulo 6: **Conclusiones y trabajos futuros**. En este último capítulo se exponen una serie de conclusiones sobre el proceso de realización de este proyecto: dificultades, objetivos logrados, aprendizaje, ...; además, se indicarán brevemente algunos conceptos relevantes sobre los que trabajar para, en un futuro, mejorar los desarrollos y el valor de los resultados.

Capítulo 2

Estado del arte

La computación de alto rendimiento se suele asociar a grandes sistemas capaces de alcanzar miles de millones de operaciones por segundo, de tal forma que pueden ejecutar aplicaciones realmente complejas en un tiempo razonable; éstas normalmente están relacionadas con ámbitos científicos y tecnológicos: simulación de entornos dinámicos de partículas, estudios del origen del universo, predicción del clima y catástrofes naturales, pruebas nucleares, ...[1] Actualmente, la supercomputadora más potente a nivel mundial ha logrado un rendimiento de unos 100 petaflops[1] en los benchmarks estándar, y se está trabajando para alcanzar la escala del exaflop en los próximos años.

No obstante, alrededor de esta idea de alcanzar la máxima capacidad de cómputo posible, han surgido nuevas preocupaciones que han afectado a la dirección del desarrollo de hardware y software para HPC considerablemente.

Con las limitaciones de la ley de Moore ya se había observado que la tendencia hacia arquitecturas paralelas era una forma no sólo de romper el muro del mononúcleo, sino de incrementar la eficiencia energética del sistema; los últimos avances en este campo han traído de la mano las denominadas arquitecturas *manycore*, que pasan a utilizar decenas, incluso centenas de núcleos de proceso; éstos son de naturaleza simple, y su rendimiento en aplicaciones secuenciales no es bueno, ya que excluyen muchas de las características técnicas de los cores modernos (ejecución OoO, pipelines profundas, ...), pero esto a su vez tiene la ventaja de hacerlos extremadamente eficientes en su consumo, mientras que logran alcanzar rendimientos elevados en aplicaciones paralelas debido a su número (ver figura 2.1).

Por otro lado, se ha insistido en la importancia de la accesibilidad a los sistemas actuales, en su mayoría distribuidos y heterogéneos; paradigmas de programación como MPI u OpenCL permiten el desarrollo de aplicaciones que pueden utilizar los distintos recursos del sistema de forma conjunta, pero son complicados y requieren conocimiento técnico de las arquitecturas subyacentes, así como de modelos de programación adecuados.

Teniendo en cuenta que las personas que hacen uso de estos sistemas no son especialistas ni siquiera en el campo de la informática (físicos, matemáticos, médicos, ...), es necesario ofrecerles una alternativa para adaptar sus aplicaciones sin que se tengan que formar extensivamente para ello; en este sentido, Intel ha hecho un movimiento significativo, ya que ha conseguido crear una arquitectura manycore compatible con modelos de programación tradicionales como los utilizados en los multinúcleo x86, la llamada Intel MIC.

[1] Un petaflop equivale a mil billones europeos (10^{15}) de operaciones en punto flotante.

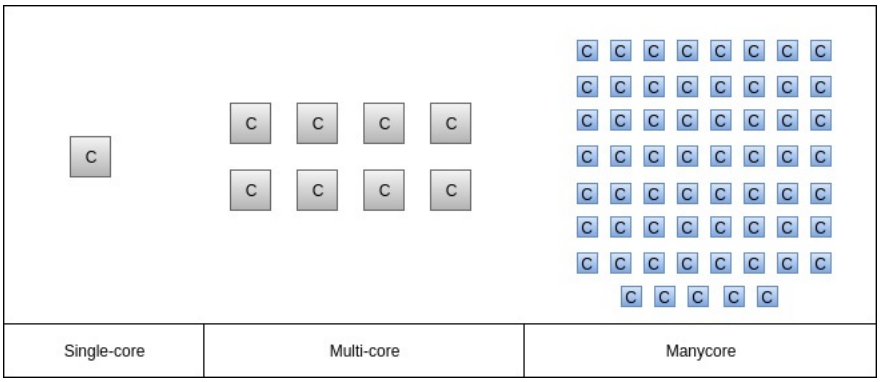

Figura 2.1: Los núcleos del manycore son mucho más sencillos, el rendimiento viene de su número.

2.1. Intel Xeon Phi

Intel Xeon Phi es el nombre comercial dado a una serie de procesadores basados en Intel MIC que vio la luz por primera vez en junio de 2013, cuando el Tianhe-2 entró en la lista Top500 como la supercomputadora más rápida del mundo utilizando este hardware[2].

La primera versión recibió el nombre en clave *Knights Corner*, y es sobre la que se ha llevado a cabo esta investigación; el Knights Corner se considera un coprocesador o acelerador, un tipo de hardware pensado para complementar a las CPUs tradicionales y asistir con las tareas de alto grado de paralelismo; otros ejemplos de aceleradores actualmente presentes en el mercado son las GPUs (ver figura 2.2).

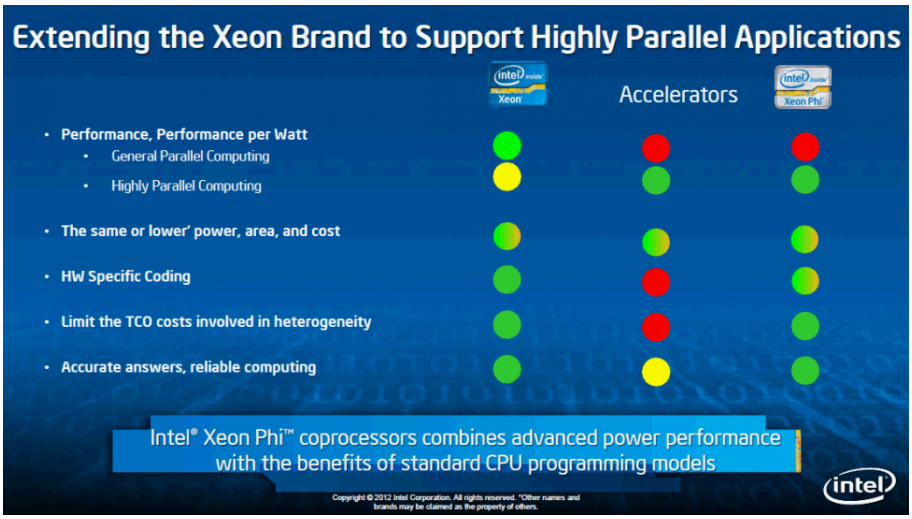

Figura 2.2: Intel destaca la reducción de costes asociados al desarrollo de software debido a la compatibilidad del Phi. Aquí "accelerators" hace referencia a las GPUs. Fuente: Intel.

2.1.1. Aspectos clave y arquitectura

El Intel Xeon Phi Knights Corner (KNC), debido a su naturaleza de acelerador, necesita un procesador tradicional para poder funcionar; la conexión y comunicación entre ambos se lleva a cabo mediante un bus PCI Express v2.

Para poder utilizarlo es necesario instalar una capa de software adicional denominada Intel MPSS (Manycore Platform Software Stack), que consiste fundamentalmente en un sistema Li-

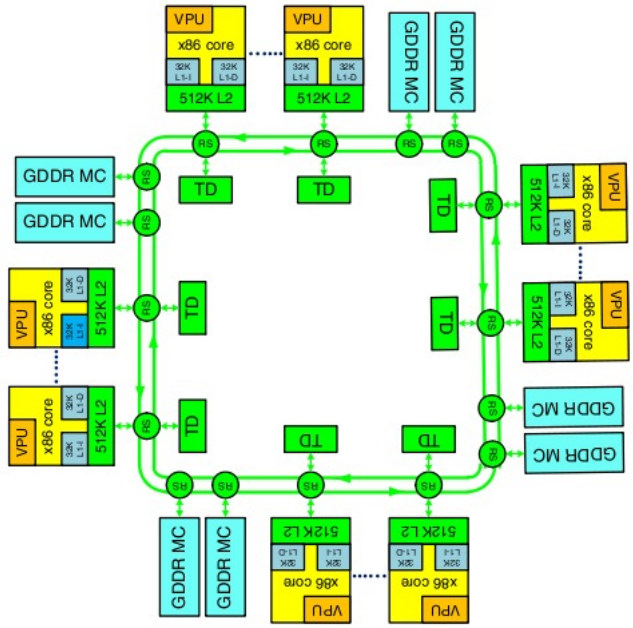

Figura 2.3: Arquitectura del Knights Corner. Fuente: [5]

nux y utilidades para la correcta comunicación y ejecución de aplicaciones; una vez instalada es posible acceder al acelerador mediante un túnel SSH desde la CPU.

Su arquitectura está basada en la ya mencionada Intel MIC; los fundamentos de ésta son la integración de muchos núcleos de proceso en un único chip, en concreto KNC posee 61, y cada uno de ellos es capaz de ejecutar 4 hilos hardware mediante SMT, lo que constituye un total de 244 posibles hilos concurrentemente en el sistema; estos núcleos están conectados entre sí a través de un anillo bidireccional de alto rendimiento, y tienen acceso a 8 controladores de memoria, con un máximo de 16 GB de GDDR5 en el sistema y ancho de banda de 352 GB/s; esta memoria es compartida entre los núcleos a través del anillo y presenta coherencia de caché[3] en todo el chip.

Cada uno de los núcleos ejecuta instrucciones en orden por dos cauces, lo que significa que puede albergar dos líneas de ejecución simultáneas y hasta 4 hilos concurrentemente gracias al SMT; funcionan a 1.2 GHz y poseen 32 KB de caché de instrucciones y otros 32 KB de datos de L1[4]; la L2 es de 512 KB en cada uno, pero es compartida entre todos, haciendo un total de 30 MB de capacidad; su microarquitectura está fundamentada en los cores x86 de Intel Pentium y poseen soporte para 64 bits (ver figura 2.3).

Un aspecto fundamental de la arquitectura del KNC además de su número de cores, es el soporte para operaciones vectoriales; cada núcleo posee una estructura vectorial con anchura de registro de 512 bits, capaz de ejecutar instrucciones SIMD de hasta 8 operaciones en coma flotante de precisión doble o 16 de simple simultáneamente; es importante destacar que la ISA vectorial del KNC, denominada Intel IMCI (Initial Many Core Instructions), no es compatible ni equivalente a las extensiones vectoriales SSE y AVX de las CPUs tradicionales, por tanto la programación vectorial explícita no es recomendable.

2.2. Modelo de programación

La paralelización del código y distribución de las distintas partes entre las unidades de proceso, así como la correcta utilización del subsistema vectorial son los dos pilares que sustentan la mejora de rendimiento ofrecida por el KNC; los modelos de programación y APIs tradicionales suelen ser incompatibles con los utilizados en los aceleradores, por ejemplo NVIDIA tiene su propia interfaz de programación para las GPUs, CUDA, y las soluciones como OpenCL, que tratan de homogeneizar el sistema desde el punto de vista software, son relativamente complejas[6].

En este sentido, KNC, al presentar la misma microarquitectura que los procesadores x86, supone la posibilidad de realizar código compatible en ambos sistemas hardware con un mismo modelo o API; sumado a esto, el hecho de que su jerarquía de memoria sea compartida y coherente, hace prácticamente inmediata la paralelización ligera haciendo uso de hilos.

Todo esto nos lleva a la elección de paradigmas que utilicen estas ventajas de forma sencilla para facilitar la accesibilidad antes mencionada, como es el caso de OpenMP[7].

2.2.1. OpenMP: memoria compartida

OpenMP es una API para la implementación de programas paralelos mediante hilos software; los hilos de ejecución son pequeñas secuencias de instrucciones independientes pertenecientes a un mismo proceso; éstos pueden ser ejecutados simultáneamente en un mismo sistema de memoria compartida, donde comparten los recursos y se comunican a través del espacio de memoria asignado a su proceso padre.

Su programación es relativamente sencilla, ya que utiliza una colección de directivas, funciones y variables de entorno que abstraen al usuario de la gestión de los hilos en el sistema; la API está disponible para los lenguajes de programación C, C++ y Fortran, siendo el primero de ellos el utilizado en este trabajo.

Cada hilo de ejecución posee un espacio de memoria privado para realizar sus operaciones y guardar las variables no compartidas, mientras que el resto de datos se encuentra en la memoria global del proceso padre, donde se realiza la comunicación y la sincronización; esta última es fundamental para la correcta ejecución del programa, y, a pesar de que OpenMP la abstrae en su mayoría, a veces es necesario expresarla explícitamente mediante puntos de sincronización (ver figura 2.4).

Un programa en OpenMP empieza como un proceso con un hilo de ejecución maestro; si no existe ninguna directiva, la aplicación se ejecutará de forma secuencial en un núcleo, en el que se planificará el hilo maestro; el trabajo del programador es detectar las partes del código altamente paralelizables y añadir las directivas correspondientes para la creación y manejo de los hilos; esto es lo que se conoce como un modelo "fork-join": en la parte secuencial ejecuta un único hilo maestro; cuando se llega a la parte paralela se generan tantos hilos como el programador haya especificado (fork) y éstos la ejecutan, cuando termina la parte paralela se reúnen los resultados (join) y vuelve a ejecutar el maestro (ver figura 2.5).

En la pieza de código 2.1 podemos ver un ejemplo de C y OpenMP con las directivas fundamentales.

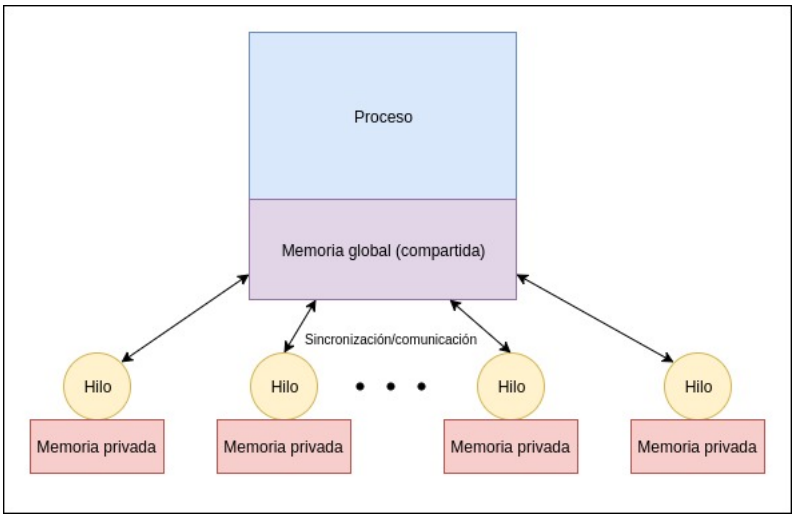

Figura 2.4: Modelo de memoria en OpenMP.

```
// Parte secuencial.
int a = 1, b = 2;

// Parte paralela.
#pragma omp parallel num_threads( 8 ) private( a ) shared( b )
{
    #pragma omp master
    {
        int numthreads = omp_get_num_threads( );
        printf( "%d\n", numthreads );
    }
    #pragma omp barrier

    #pragma omp for simd
    for( int i = 0; i < 10; i++ )
    {
        a += 1;
        b *= 2;
    }
}

//Parte secuencial.
printf( "Privada: %d, compartida: %d\n", a, b );
```

Listing 2.1: Ejemplo de código C con directivas OpenMP.

Todas las directivas comienzan con *#pragma omp* y continúan con su acción concreta, seguida de sus opciones.

En este caso, el hilo maestro comienza el programa y crea dos variables enteras, a con valor de 1 y b con valor de 2; posteriormente llega a la directiva *parallel*, la cual se encarga de hacer el fork y crear tantos hilos como los especificados en la opción *num_threads*, es decir, 8; la variable a va a la memoria privada de cada hilo mediante la opción *private*, mientras que la variable b va a la memoria global del proceso padre a través de la opción *shared*, y por lo tanto es compartida entre los hilos.

A partir de la directiva *parallel*, el mismo código es ejecutado por todos los hilos activos; la sección *master* hace que sólo el hilo maestro ejecute esa parte del código, así que el resto de hilos se la saltarán hasta llegar a *barrier*, que constituye un punto de sincronización explíci-

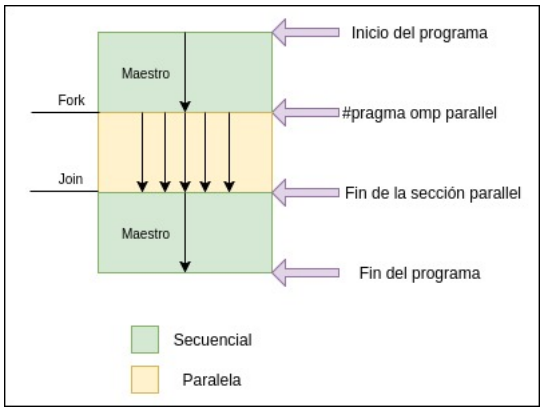

Figura 2.5: Modelo de ejecución en OpenMP.

to: mientras que todos los hilos no hayan ejecutado hasta esa barrera, no se reanuda el programa.

La directiva compuesta *for simd* es una combinación de dos directivas básicas: *for* señala que las iteraciones del bucle a continuación se dividirán entre el número de hilos disponibles y se ejecutarán en paralelo, mientras que *simd* indica al compilador que, si es posible, se utilicen las instrucciones vectoriales para el contenido del bucle.

OpenMP ofrece muchas más directivas, así como funciones en tiempo de ejecución que permiten modificar el entorno, más adelante se verán las más relevantes a la hora de programar para la arquitectura MIC.

2.2.2. Modo nativo

Hemos destacado que, a diferencia de las GPUs, KNC es un sistema en sí mismo, ejecuta una capa Linux y es posible acceder a él desde el host Xeon; esta característica es de alta importancia, ya que nos permite la ejecución nativa de aplicaciones en el coprocesador.

Existen varias ventajas que nos ofrece el ejecutar de forma nativa; entre ellas, hay dos que son fundamentales:

- No existe transferencia de datos entre el host y el acelerador; al tener control total del entorno de ejecución, el acceso a los recursos es inmediato y no se requiere de comunicación ni sincronización entre los dos sistemas; KNC puede trabajar sólo.
- El compilador, al producir código nativo para el KNC, es capaz de fijarse en la arquitectura MIC para generar código que se aproveche de las características beneficiosas de ésta, como es el conjunto de instrucciones IMCI.

2.2.3. Modo descarga (offload)

Si bien es cierto que el modo nativo nos permite aprovechar al máximo el potencial del KNC, es importante destacar que mientras éste está trabajando, el Xeon no participa en la ejecución del programa; es posible ejecutar distintas aplicaciones en los dos sistemas para mantenerlos ocupados, pero ¿existe alguna manera de que ambos puedan participar en una misma tarea?

Tanto el compilador de Intel como OpenMP en su revisión 4.0 y posteriores, introducen un modo de programación para aceleradores, el denominado *offload*, y que precisamente se encarga de responder a ésta pregunta.

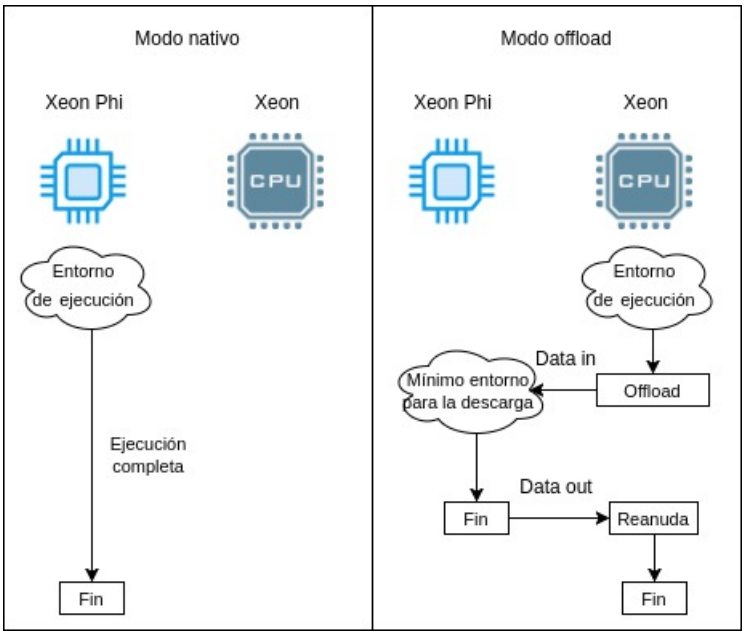

Figura 2.6: Nativo vs. Offload.

El modo *offload*, en castellano *descarga*, permite al host asignar partes de la tarea al KNC, de tal forma que las secciones de código secuencial se ejecutan en la parte del host, mientras que las más paralelizables son delegadas al acelerador (ver figura 2.6); este modo es, en esencia, el utilizado en la programación de GPUs.

En ocasiones, la magnitud de cómputo de una aplicación puede ser considerable incluso para un coprocesador como KNC; en estos casos, es bastante común que mientras el acelerador se encarga de ello, el host no esté haciendo nada, ya que está esperando los resultados y por tanto se desperdician recursos.

En este sentido, es realmente interesante explorar la posibilidad de repartir la carga computacional entre ambos sistemas; para poder obtener un rendimiento óptimo, el tiempo empleado para procesar la carga por cada sistema ha de ser el mismo, y para ello es el programador el que ha de equilibrar esta repartición, enviando más datos al componente más rápido y menos al más lento.

2.3. Algoritmos de equilibrio de carga

Existen cientos de aplicaciones para HPC, y cada una de ellas puede estar implementada de muchas maneras distintas, por lo tanto es muy complicado aportar un mecanismo genérico de equilibrio de carga cuando no conoces el comportamiento de la aplicación sobre el sistema heterogéneo.

En este proyecto se han desarrollado dos formas distintas de afrontar este problema, cada una de ellas enfocada a un tipo de carga computacional: regular e irregular (ver figura 2.7).

2.3.1. Pre-análisis estático

La carga computacional regular se caracteriza por ser constante a lo largo del conjunto de datos; esto quiere decir que una unidad de carga cualquiera da lugar al mismo número de ope-

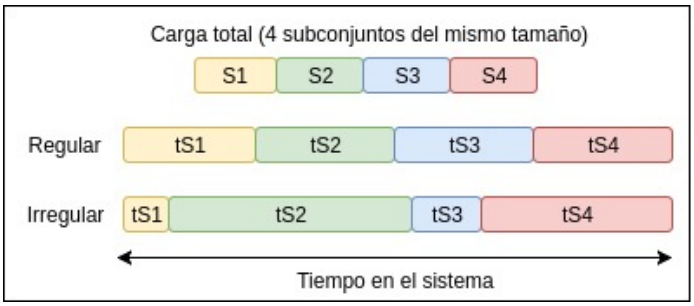

Figura 2.7: Diferencias entre carga regular e irregular.

raciones en el sistema que el resto de ellas.

Normalmente balancear la carga implica asignar un subconjunto de datos a cada sistema, y debido a la regularidad es deducible que cualquier subconjunto de datos del mismo tamaño será procesado en el mismo tiempo; esto nos permite despreocuparnos de qué intervalos de datos procesará cada parte del sistema, centrándonos en la cantidad o magnitud.

En este sentido, el pre-análisis estático se encarga de ofrecer al programador una serie de pautas para poder obtener el ratio óptimo de magnitud de carga para cada sistema.

2.3.2. Dinámico en tiempo de ejecución

A veces, las operaciones llevadas a cabo dentro del núcleo de cómputo son condicionadas por factores intrínsecos a la etapa de ejecución; un ejemplo típico para verlo más claro es el procesado por filas de una matriz dispersa: cuando los valores de los elementos son 0, no se realiza operación alguna, por tanto la carga de los subconjuntos de filas es variable respecto del número de elementos que tenga cada una de ellas.

Este tipo de carga se suele denominar irregular, y siguiendo con la definición utilizada anteriormente, distintos subconjuntos de datos del mismo tamaño no tienen por qué ejecutarse en el mismo tiempo.

Debido a esto, el análisis en pos de encontrar un ratio óptimo en función de las cantidades se torna complicado, ya que no existe una tendencia definitiva; a raíz de ello, es interesante pensar en una alternativa adaptable, un algoritmo en tiempo de ejecución que ofrezca carga a medida que los sistemas vayan acabando su trabajo.

Capítulo 3

Optimización de aplicaciones para el Xeon Phi

La fase de procesado de las aplicaciones con un alto coste computacional derivado de grandes cantidades de datos, normalmente implica la ejecución de unas pocas operaciones sobre millones de elementos; en este sentido, uno de los conceptos más importantes hoy en día en el mundo de la HPC es la **escalabilidad**, que se refiere a la capacidad de una aplicación de adaptar su ejecución sobre varios procesadores, de tal forma que estas operaciones se lleven a cabo de forma paralela sobre los distintos datos, y sin sufrir una pérdida significativa de rendimiento a causa de la repartición.

Dos de los principales caminos para conseguir esta característica son la paralelización de los programas y la utilización de paradigmas de programación vectoriales para aprovechar la arquitectura del mismo nombre; el Xeon Phi es un componente hardware especializado en ambos métodos, y por tanto es mucho más escalable que un procesador multinúcleo tradicional (ver figura 3.1), por tanto si queremos exprimir su rendimiento tendremos que modificar nuestras aplicaciones, que gracias a la programabilidad compatible, beneficiará tanto al host Xeon como al acelerador; en este capítulo veremos unas pautas generales para aprovechar al máximo ambos aspectos.

Las opciones, variables y comandos utilizados en las explicaciones son condicionadas al uso del lenguaje C, al compilador de Intel ICC y a las librerías de OpenMP[8] provistas por Intel; la utilización de GCC no es recomendable para generar ejecutables del Xeon Phi debido a ciertas limitaciones[9].

3.1. Identificando las cargas de cómputo paralelizables

Este proyecto se centra en aplicaciones masivamente paralelas en datos, en las que normalmente el coste computacional gira en torno a los bucles de procesado; estos recorren estructuras de datos y realizan operaciones sobre sus elementos, frecuentemente modificando su contenido o utilizándolo para generar uno nuevo; las operaciones realizadas cumplen un papel muy importante en la posibilidad de paralelizar estos bucles (ver figura 3.2), ya que pueden presentar **dependencias**; este problema se da cuando las operaciones de una iteración necesitan el resultado de otra, lo que impide la ejecución simultánea y la ganancia de rendimiento.

Existen diferentes maneras de reducir las dependencias, y por suerte los compiladores modernos se encargan en gran parte de esto[11], no obstante a la hora de programar es importante mantener los bucles limpios para facilitar el trabajo al compilador; la recomendación es utilizar

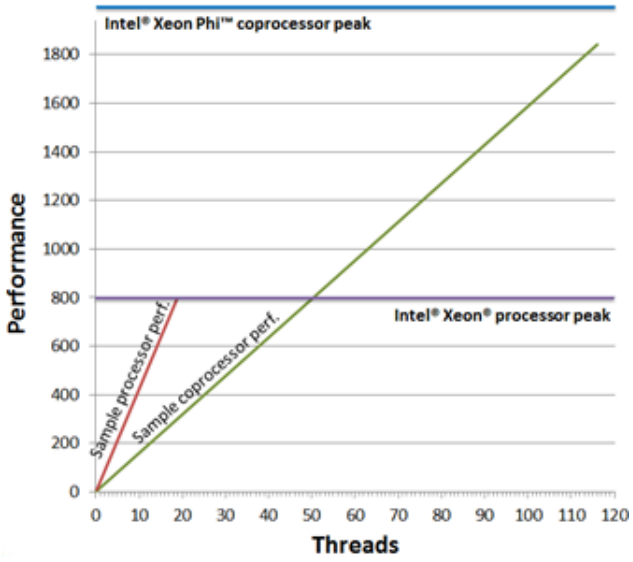

Figura 3.1: Escalabilidad del Xeon Phi respecto al Xeon. Fuente: [10].

las optimizaciones de éste siempre que sea posible, en los casos de GCC e ICC esto se puede conseguir con **-O3** como opción de línea de comandos.

3.2. Paralelizando el código

Una vez sabido que la carga es en efecto paralelizable, es necesario indicar al compilador cómo hacerlo de la forma deseada; en este caso la manera tradicional de llevarlo a cabo con OpenMP nos sirve para ambos sistemas debido a la compatibilidad existente; en el caso del Xeon Phi es necesario compilar con la opción **-mmic**, para que el ejecutable generado sea compatible con la arquitectura MIC; es importante no especificar el número de hilos en el código del programa, ya que el acelerador puede soportar muchos más que el Xeon y por tanto son valores distintos; para controlar esto, es fundamental configurar el entorno de ejecución del Xeon Phi.

Para poder ejecutar una aplicación nativa sobre el Phi hay que conectarse a él a través de un túnel SSH, y una vez dentro se presentará una interfaz de consola Linux tradicional; la fase de compilación se realiza en el host Xeon, y el ejecutable resultante se copia al Xeon Phi con herramientas como SCP[12]; en Linux, el entorno de ejecución se configura a través de *variables de entorno*, y para el caso del paralelismo existen dos en concreto fundamentales: **OMP_NUM_THREADS** y **KMP_AFFINITY**.

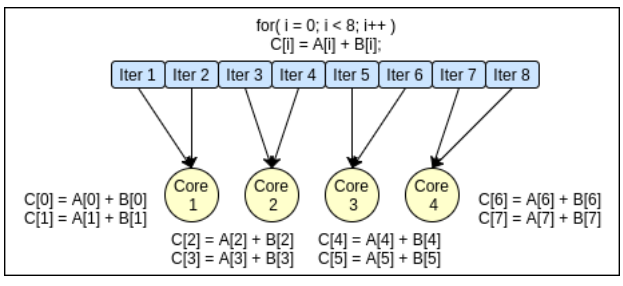

Figura 3.2: Paralelización de un bucle.

Con OMP_NUM_THREADS es posible especificar el número de hilos software que OpenMP generará al llegar a la directiva *#pragma omp parallel*; en general, el máximo rendimiento es conseguido con un número de hilos igual al máximo soportado por el sistema; existen varias versiones del Xeon Phi, con diferente número de núcleos de proceso, por eso siempre es recomendable cerciorarse de este dato in situ; para ello, es posible obtenerlo con el comando **nproc**, o si no se dispone de éste con **cat /proc/cpuinfo | grep siblings | head -n 1**; en este caso el número de hilos soportados es de 61 núcleos x 4 hilos hardware que hacen un total de 244, por tanto se especificará esta variable con un valor de **244**.

KMP_AFFINITY permite especificar la afinidad de los hilos software al hardware subyacente; este es un concepto de bajo nivel de abstracción pero de alta importancia, y viene a definir la manera en la que se van a planificar los hilos software en las unidades y núcleos de proceso.

En general existen tres opciones principales, aunque se pueden hacer configuraciones personalizadas:

- Compacta (*compact*): los hilos software se van asignando a los hilos hardware de cada núcleo de forma que se utilicen todos los recursos de un núcleo antes de planificar sobre el siguiente; de esta manera, los hilos software estarán alojados en el menor número de núcleos posible; esto hace que se desperdicien recursos de otros núcleos.

- Dispersa (*scatter*): los hilos software se asignan a los hilos hardware de tal forma que ocupen el máximo número de núcleos posible; si tenemos 65 hilos, del 0 al 60 irán a los núcleos 0 al 60, mientras que el 61 se planificará de nuevo sobre el núcleo 0 (ciclo); esto aumenta la utilización del sistema, pero se pierde el beneficio de compartir caché entre los hilos, ya que cada uno estará en un núcleo distinto.

- Equilibrada[13] (*balanced*): fundamentalmente como la dispersa, se prioriza la utilización de los núcleos, pero en este caso los hilos software que se encargan de conjuntos de datos consecutivos se planifican sobre el mismo núcleo, de tal forma que exista la compartición de caché de la opción compacta.

Al utilizar el máximo número de hilos, realmente la afinidad disminuye su relevancia, ya que la utilización será siempre máxima, la ganancia surgirá de las dependencias entre hilos y su cercanía; la opción más recomendable para el caso general es que esta variable posea el valor **balanced**.

3.3. Vectorización y SIMD con Intel IMCI

Además de repartir el trabajo entre las distintas unidades de proceso o hilos hardware, existe una forma complementaria de producir paralelismo de datos y mejorar el rendimiento: las extensiones vectoriales; también llamadas SIMD (una instrucción, múltiples datos), permiten ejecutar una misma operación sobre un conjunto de elementos denominado *vector* (no confundir con la estructura de datos del mismo nombre) simultáneamente (ver figura 3.3).

El número de estos elementos depende de la *anchura de registro* vectorial; en el caso del Xeon Phi, el conjunto de instrucciones IMCI tiene una anchura de 512 bits, y soporta tanto números enteros como en coma flotante; por ejemplo un número en coma flotante de doble precisión ocupa en el lenguaje C 8 bytes, 64 bits, por tanto se podrá ejecutar una operación simultánea sobre 8 de estos ($512/64 = 8$); en el caso de los multi-core tradicionales, a pesar de que hoy en día se están adoptando nuevas tecnologías como AVX-512, se utilizan extensiones vectoriales con menor anchura de registro, como son SSE (128 bits) o AVX (256 bits), por tanto

para aplicaciones vectorizables el Xeon Phi dará mejores resultados.

Y, ¿qué se considera una aplicación vectorizable? De nuevo surge el problema de las dependencias; para poder ejecutar simultáneamente una operación sobre varios datos, no pueden existir dependencias entre los elementos participantes y/o el resultado de sus operaciones, ya que no respetaría el modelo de consistencia secuencial; en el caso del paralelismo, cuando existe alguna dependencia ha de ser tratada con una etapa de sincronización, en la que, a través del establecimiento de un orden de ejecución de los hilos, se garantice esta consistencia, sin embargo en el caso de la vectorización es un requerimiento que no existan.

Otra restricción importante es el *stride* o desplazamiento de acceso a las estructuras de datos; la vectorización sólo se puede realizar sobre datos consecutivos, si existe un desplazamiento de acceso, de tal forma que en la iteración i se acceda al elemento m y en la $i+1$ se acceda al elemento $n > m+1$ la vectorización no será posible; esto suele pasar cuando los desplazamientos de acceso son un derivado del índice de iteración, por ejemplo se accede a la estructura con desplazamiento $estructura[2 * i]$.

El compilador ICC tiene la capacidad de auto-vectorizar el código de un programa a partir de las optimizaciones -O2 en adelante, no obstante es bastante conservador en este aspecto, en el caso de que no esté seguro de las dependencias existentes o no en un fragmento de código, no realizará el proceso; en este sentido, es importante echar una mano al compilador cuando el programador esté seguro de que no existen dependencias, y para ello son proporcionados mecanismos como *#pragma ivdep*, que indica al compilador que ignore las dependencias asumidas, o en el caso de OpenMP 4.0+ las directivas *#pragma omp simd*; existen múltiples tutoriales en la red, además es posible profundizar en el tema con artículos como [14].

Es importante destacar la existencia de instrucciones *intrinsic*, que son llamadas a funciones que realizan específicamente una operación vectorial perteneciente a un conjunto de instrucciones vectoriales concreto; en el caso de IMCI, por ejemplo *_m512d __cdecl _mm512_add_pd(_m512d v2, _m512d v3)* suma dos vectores de elementos en coma flotante de doble precisión; estas instrucciones permiten acceder al proceso de vectorización desde un nivel más bajo, y por tanto especificar de forma más concreta lo que se quiere realizar sobre el sistema, no obstante el inconveniente que tienen y por lo que no son recomendables es porque son inherentes a la arquitectura del Xeon Phi, lo cual implica que no podrán compilarse para el host Xeon, y romperán la principal ventaja del acelerador, la programabilidad compatible.

La vectorización es complementaria al paralelismo; cada unidad de proceso posee 32 registros vectoriales, en total 128 en cada núcleo, que también posee 1 unidad de procesamiento vectorial;

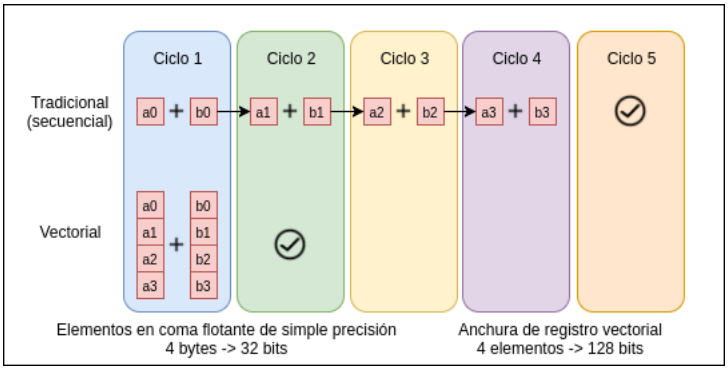

Figura 3.3: Funcionamiento de la vectorización.

en este sentido, para obtener el máximo rendimiento es importante integrar ambas técnicas.

3.4. Otros aspectos relevantes

Gracias a la abstracción de los paradigmas de programación como OpenMP, las directivas de los compiladores y las optimizaciones llevadas a cabo por estos, paralelizar y vectorizar un programa para el Xeon Phi no resulta extremadamente complejo, no obstante se pueden tener en cuenta pequeños aspectos para optimizar aún más las aplicaciones.

3.4.1. Alineamiento de estructuras de datos

En sistemas GNU/Linux y programación en C, cuando se dispone de estructuras de datos de gran magnitud, se suele utilizar memoria dinámica o *heap*; las llamadas más comunes para solicitar un bloque de memoria a utilizar son *malloc* y *calloc*.

Para computadoras de 64 bits, el bloque devuelto por estas llamadas consiste en una dirección de memoria múltiplo de 16; esto quiere decir que el primer elemento de la estructura que va a ser contenida en ese bloque estará localizado en una posición de memoria divisible por 16.

Las memorias caché poseen un *tamaño de línea* que representa la unidad mínima de transacción entre la memoria principal y éstas; tanto en el caso del host Xeon como el del acelerador este tamaño es de 64 bytes.

Las transacciones de memoria principal a caché son mucho más costosas que las de caché a registros, por tanto es muy importante reducirlas; para ello, es fundamental el correcto alineamiento de los bloques de memoria solicitados, en concreto se deberá alinear cualquier estructura al tamaño de línea de caché, 64; de esta forma, se consigue que el primer elemento de la estructura siempre empiece en una nueva línea de caché, de tal forma que el grueso de la estructura ocupe el mínimo número de líneas posible, y por tanto suponga el mínimo número de transacciones (ver figura 3.4).

Además de ser importante para minimizar las transacciones de memoria principal a caché, el alineamiento tiene un impacto en el proceso de vectorización, ya que se reproduce el mismo problema en las transacciones de caché a registros: el registro vectorial de 64 bytes (IMCI) accederá a la estructura de datos, si se contiene en una única línea de caché traerá todos sus elementos, si se contiene en dos necesitará dos registros y por tanto dos operaciones vectoriales.

El registro vectorial del Xeon en el caso de este trabajo es de una anchura de 32 bytes (AVX), mientras que el del acelerador sabemos que es de 64 bytes, no obstante al estar alineado a 64 está consecuentemente también a 32 (2 accesos a la misma línea de caché con desplazamiento 0 y 32).

Para poder asignar posiciones de memoria alineadas a estructuras de datos, existen llamadas especiales, en el caso de Intel nos ofrecen **_mm_malloc** y en el estándar C11 se define **aligned_alloc**, ambas análogas a la hora de compilar; además de alinear la memoria, es necesario indicar al compilador que está alineada para que la utilice en el proceso de vectorización, en el caso de Intel nos proporcionan la directiva *#pragma vector aligned* y en OpenMP 4.0+ la opción *aligned(nombre_estructura : alineamiento)* para la directiva *#pragma omp simd*.

3.4.2. Índices de los subconjuntos de datos

Al paralelizar un bucle se asigna un subconjunto de iteraciones a cada hilo software; estos hilos, como se ha visto, estarán distribuidos por los distintos núcleos de proceso del acelerador,

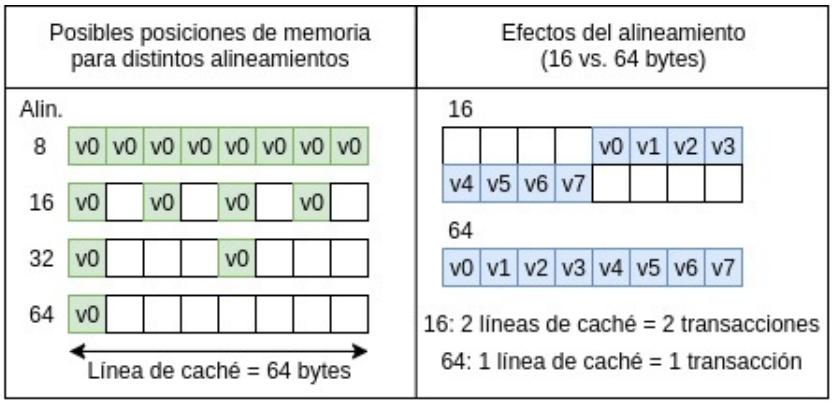

Figura 3.4: Fundamentos del alineamiento.

y por tanto accederán a distintas memorias caché L1.

Los accesos a las estructuras de datos dentro de los bucles son dependientes en la mayoría de los programas del índice de la iteración que se está ejecutando en ese momento, o en su defecto de un desplazamiento sobre ese índice; en este sentido, cada hilo accederá a un subconjunto de la estructura de datos definido por los índices de acceso y del tamaño del chunk[1] especificado para el planificador.

Estos accesos traerán partes de la estructura de datos de memoria principal a la caché del núcleo de proceso en el que se aloja el hilo; como antes, es deseable reducir el número de estos accesos, así que se habrá de recurrir de nuevo al recurso del alineamiento: se trata de que cada hilo reciba un índice inicial alineado a 64 bytes, de tal forma que pueda traer el subconjunto de datos en el menor número de transacciones.

El tamaño de chunk es medido en iteraciones por tanto hay que tener en cuenta el tamaño de los elementos de la estructura de datos, si son enteros de 4 bytes, el tamaño de chunk tendrá que ser múltiplo de 16 iteraciones (64 div. 4), si son flotantes de doble precisión el elemento es de 8 bytes, por tanto chunk múltiplo de 8 iteraciones (64 div. 8).

Es relativamente común el fenómeno denominado *false sharing* o falsa compartición, en el que dos o más hilos en dos o más núcleos diferentes acceden a subconjuntos de la estructura de datos contenidos en líneas de caché comunes; cuando un hilo de un núcleo accede a su subconjunto trae una línea de caché y la modifica; cuando el otro hilo accede al suyo, y precisamente ha de acceder a esa línea de caché común con otro desplazamiento, no está en su memoria caché, ya que está alojado en un núcleo diferente, y por tanto se origina un conflicto de coherencia.

En la figura 3.5 se puede observar un ejemplo de este fenómeno; un vector de 14 elementos va a ser procesado por 2 hilos, cada uno de ellos se encargará de 7 elementos; estos dos hilos están alojados en dos núcleos distintos, con cachés L1 privadas.

El vector no está alineado a 64 bytes, y por tanto ocupa 3 líneas de caché en vez de 2; en la segunda de estas líneas, los elementos de $v3$ a $v6$ están asociados al hilo 1, mientras que de $v7$ a $v10$ lo están al hilo 2; esta línea es, por lo tanto, común a ambos hilos y supondrá una falsa compartición en las cachés privadas, con un coste adicional de coherencia.

Establecer una distribución de hilos favorable a los accesos de memoria y un alineamiento de

[1] El tamaño de chunk es el número de iteraciones de los subconjuntos a asignar a los hilos software.

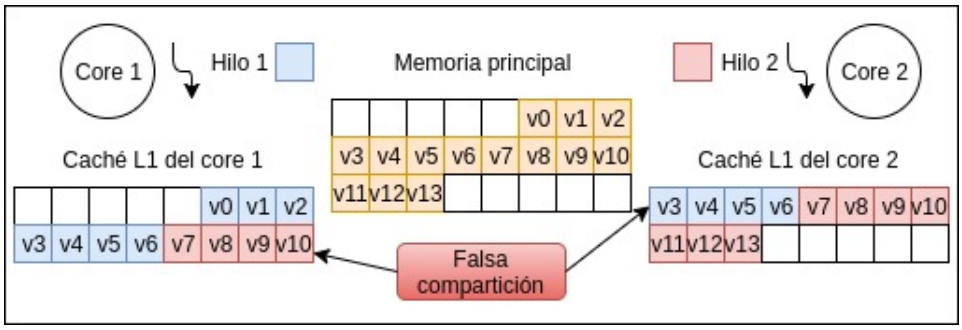

Figura 3.5: Falsa compartición.

```
int main( )
{
    const int N = 5000;
    const double K = 2.0;
    int i;
    double *A, *B;

    A = ( double * ) aligned_alloc( 64, N * sizeof( double ) );
    B = ( double * ) aligned_alloc( 64, N * sizeof( double ) );

    lee_datos( A, B );

    for( i = 0; i < N; i++ )
    {
        A[i] = A[i] + K * B[i];
    }

    muestra_resultados( A );

    return 0;
}
```

Listing 3.1: DAXPY base.

los índices reducen su impacto; para informarse más detalladamente sobre la falsa compartición se recomienda leer [15].

3.5. Ejemplo práctico

Una vez explicados los conceptos más importantes para la optimización de aplicaciones en el Xeon Phi, se procede a exponer un pequeño ejemplo que sirva de base para la adaptación de otras aplicaciones; éste se va a basar en la rutina DAXPY, la cual realiza la siguiente operación en un bucle: $A[i] = A[i] + K * B[i]$, donde los vectores A y B, así como la constante K son números en coma flotante de doble precisión.

El código base se puede observar en el fragmento de código 3.1; se ha utilizado *aligned_alloc* para alinear los dos vectores al tamaño de línea de caché y reducir así los accesos a memoria principal.

Primero es necesario identificar si el bucle, que es el principal factor de coste computacional, es paralelizable; la iteración con operación $A[i] = A[i] + K * B[i]$ es independiente del resto de iteraciones, ya que no hace referencia a los resultados de éstas, por tanto el bucle es paralelizable; un contraejemplo podría ser una operación de tipo $A[i] = A[(i-1)\,\%N] + K * B[i]$, en la que el

resultado actual ($A[i]$) depende del resultado de la iteración anterior ($A[i-1]$).

En el fragmento de código 3.2 se puede observar el bucle paralelizado con OpenMP; el tipo de datos *double* es de 8 bytes en el lenguaje C, por tanto en una línea de caché de 64 bytes entrarán 8 elementos, así que el tamaño de chunk ha de ser múltiplo de 8 para tener subíndices alineados y reducir la falsa compartición.

La planificación especificada es estática, ya que la carga computacional de cada iteración es la misma, una operación DAXPY; los chunks para este tipo de planificación son asignados a los hilos por orden de identificación, y cuando se ha asignado un chunk a cada hilo, el siguiente chunk se ofrece de nuevo al primer hilo (*round-robin*[2]); en este sentido, surge la reflexión de si es mejor un chunk pequeño (de una línea de caché), o uno grande (de varias líneas).

```
#pragma omp parallel for private( i ) schedule( static, 16 )
for( i = 0; i < N; i++ )
{
    A[i] = A[i] + K * B[i];
}
```

Listing 3.2: DAXPY paralelo.

Dejando de lado el aspecto práctico para las pruebas realizadas, existe algún factor teórico que favorece el chunk grande; el Xeon Phi soporta *prefetching*, una técnica para reducir las latencias de la jerarquía de memoria, mediante la cual se traen a la caché bloques de memoria antes de ser utilizados por el programa.

El prefetching se basa en el patrón de acceso a memoria, es decir, predice los bloques que se van a utilizar, así, si el programa pide una línea de memoria L, se hace prefetch de la L+1, ya que es probable que sea utilizada; si el chunk es de una línea de caché, el prefetching no servirá, ya que la siguiente línea es otro chunk para otro hilo, sin embargo si es de varias líneas, esta técnica incrementaría el rendimiento; para conocer más sobre el prefetching se puede recurrir a [16].

En este caso el tamaño de las estructuras de datos es $N = 5000$; como el Xeon Phi posee 244 hilos hardware, se habrán de generar 244 hilos software; si se divide 5000 entre 244, el chunk es de 20 iteraciones para cada hilo; como 20 no es múltiplo de 8 se busca el anterior número que sí lo sea, en este caso 16.

Si se utilizase 24 como tamaño de chunk, algunos hilos se quedarían sin carga de trabajo, ya que $5000/24 = 208$, es decir, con 208 - 209 hilos se cubrirían todos los datos, sin embargo con un tamaño de 16 se completa el primer ciclo de asignación en el elemento $244 * 16 = 3904$, y de nuevo se asigna el chunk [3905 - 3920] al primer hilo.

Es apreciable que 16 no divide a 5000, el resto de esta división es 8, lo cual quiere decir que existirá un pequeño chunk de 8 iteraciones [4993 - 5000] restante, lo que se denomina el *remainder* y que se suele aislar del resto de iteraciones por el compilador para poder vectorizar íntegramente el bucle principal.

Es importante destacar que el tamaño de chunk se puede especificar en una variable, por

[2] Algoritmo de planificación equitativo, que asigna de uno en uno los elementos a los recipientes desde el primero hasta el último, y empieza de nuevo desde el primero cuando se ha completado un ciclo de asignación.

tanto se puede adaptar tanto al tamaño de las estructuras de datos como a los hilos a generar (si se ejecuta en el host Xeon serán 24, en el Phi 244).

El código ya está paralelizado, por último es necesario vectorizarlo; en el fragmento de código 3.3 es posible observar el resultado final; la directiva *simd* indica al compilador que el bucle es vectorizable, mientras que la opción *aligned* informa de que todos los subconjuntos de datos de A y B están alineados al tamaño de línea de caché, 64 bytes.

```
#pragma omp parallel for simd private( i ) schedule( static, 16 )
aligned( A, B : 64 )
for( i = 0; i < N; i++ )
{
    A[i] = A[i] + K * B[i];
}
```

Listing 3.3: DAXPY paralelo y vectorizado.

Este es un ejemplo muy básico y que favorece tanto la paralelización como la vectorización, no obstante puede funcionar de referencia a la hora de adaptar las aplicaciones; si se consiguen simplificar los bucles, eliminar dependencias y alinear accesos, se pueden optimizar de una forma similar a ésta; para ello, puede resultar interesante informarse sobre las transformaciones de bucles en artículos como [17].

Los efectos en el rendimiento de una aplicación derivados de estas optimizaciones suelen ser significativos, siempre y cuando el programa sea tanto paralelizable como vectorizable; en el apartado 5.4.1 se puede observar la magnitud de las mejoras para una aplicación que cumple estas dos premisas.

Capítulo 4

Algoritmos de equilibrio de carga

En las aplicaciones de alto grado de paralelismo de datos, la principal manera de utilizar un sistema heterogéneo es repartir esta carga de datos entre los distintos componentes; por ello, un buen parámetro para establecer cuán adaptada está una aplicación al sistema sobre el que ejecuta, es el grado de utilización de sus recursos.

En el anterior capítulo se presentaban una serie de factores a tener en cuenta para exprimir al máximo el rendimiento individual tanto del host Xeon como, y más importante, del acelerador Xeon Phi; este capítulo se centrará en explicar y ofrecer métodos para la explotación de ambos sistemas simultáneamente, consiguiendo rendimientos aún superiores a los programas optimizados.

4.1. ¿En qué consiste el equilibrio de carga? Algunas aclaraciones

Se puede definir la carga computacional de una aplicación como la cantidad de datos a procesar, y su coste como el tiempo que emplea un sistema hardware concreto en llevar a cabo tal tarea; este coste es condicionado por diversos factores: diseño del software, *utilización de la arquitectura hardware*, fase de compilación, entorno de ejecución y recursos disponibles, ...

El equilibrio de carga se enfoca en maximizar uno de estos factores, la utilización del sistema hardware subyacente; actualmente y como se ha mencionado en los capítulos introductorios, existe una proliferación de sistemas distribuidos heterogéneos con distintos componentes hardware que colaboran en las tareas; además de utilizar de forma óptima las facilidades de las arquitecturas de estos (cubierto en el capítulo 3), es importante que todos colaboren de una manera activa durante la ejecución de los programas.

En este sentido existen dos maneras de maximizar esa actividad: división de una tarea en subtareas y repartición de éstas (paralelismo de tareas), y división de los datos a procesar en subconjuntos de datos que se asignan a las partes del sistema (paralelismo de datos) (ver figura 4.1); esta investigación se ha centrado en el equilibrio de carga a través del paralelismo de datos, no obstante estas dos vías son complementarias, por tanto no se descartan trabajos futuros que involucren el paralelismo de tareas.

En un entorno heterogéneo, los distintos componentes poseen diferentes potencias computacionales; en base a esto, el trabajo realizado se ha centrado en repartir estos datos de tal forma que el dispositivo más rápido reciba mayor carga que el lento; así, se pretende que ambos componentes terminen el trabajo a la vez, alcanzando el equilibrio mencionado y evitando

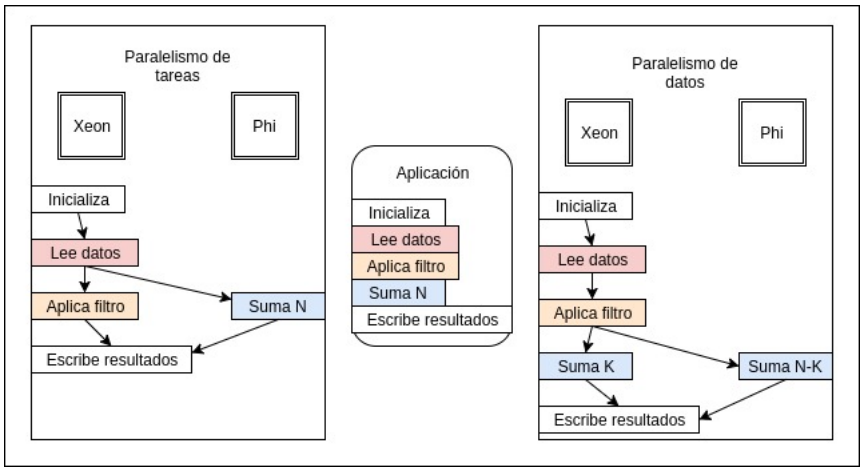

Figura 4.1: Distintas formas de equilibrar la carga.

pérdidas.

4.2. Tipos de aplicaciones según su carga computacional

Como se ha señalado previamente, las aplicaciones de HPC son muy diversas computacionalmente, tanto en sus modelos como en sus implementaciones, así que un primer paso para facilitar esta tarea es encontrar un modo de clasificarlas.

Es especialmente interesante una clasificación que permita distinguir las aplicaciones en función del tratamiento que dan a sus datos, de tal forma que posteriormente se pueda establecer una relación entre métodos de equilibrio adecuados para cada clase.

4.2.1. Aplicaciones con carga regular

Los programas caracterizados por este tipo de carga tratan los datos de entrada de una manera genérica o no discriminatoria, lo cual quiere decir que no se contemplan casos especiales en los que se podrían ahorrar operaciones, bien porque se sepa a priori que esto no es posible/viable o porque no se ha tenido en cuenta en la fase de desarrollo.

La ventaja de programar este tipo de aplicaciones es que son sencillas de desarrollar, reduciendo el coste del proceso así como las habilidades necesarias para llevarlo a cabo; además, facilitan su equilibrio de carga, ya que no existen restricciones al dividir los datos, sabemos que cualquier subconjunto se va a procesar de la misma forma y por tanto en el mismo tiempo.

Ejemplos de este grupo son abundantes, entre ellos la mayoría de algoritmos en su versión clásica: multiplicación de matrices, ordenamiento rápido, árboles mínimos en grafos (Prim, Kruskal), etc.

4.2.2. Aplicaciones con carga irregular

Los algoritmos de carga regular funcionan muy bien para estructuras de datos densas, que poseen información relevante en la mayoría de sus elementos; no obstante, existen ciertos problemas que generan estructuras dispersas o *sparse*[18], matrices, vectores y otros contenedores en los que gran parte de sus elementos contienen información nula (ver figura 4.2).

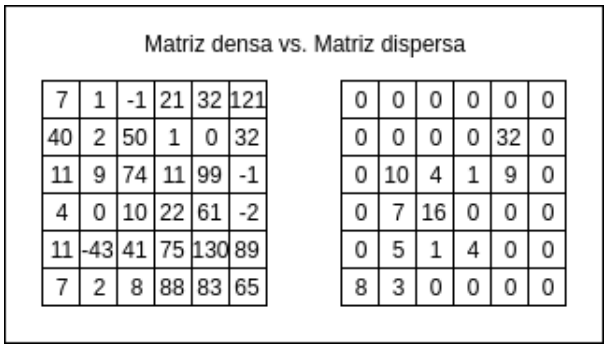

Figura 4.2: Diferencias en las cantidades de información.

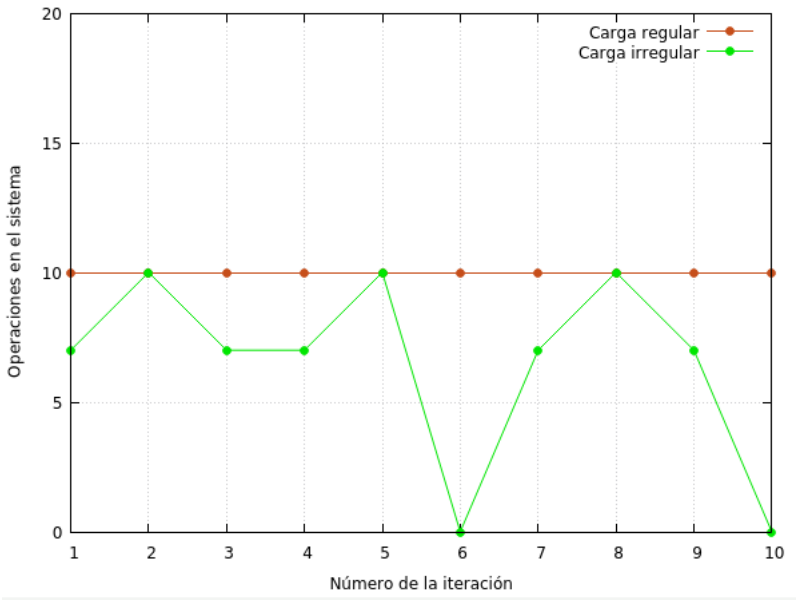

Figura 4.3: Diferencias entre la carga regular y la irregular en un bucle de procesado.

Para estos casos, un algoritmo de carga regular es ineficiente, ya que realiza un grueso de operaciones innecesarias que tienen impacto en el rendimiento del programa; es por ello que se suelen desarrollar versiones de los algoritmos clásicos específicamente para este tipo de estructuras[19]; normalmente estas versiones poseen condiciones intrínsecas en cada iteración de los bucles de procesado, condicionales o índices dependientes (ver fragmento de código 4.1) que desequilibran la carga (ver figura 4.3).

```
for( j = 0; j < N; j++)
    for( i = 0; i < j; i++ )
        for( k = i; k < j; k++ )
            C[i][j] += A[i][k] * B[k][j];
```

Listing 4.1: Ejemplo de índices dependientes.

Como ejemplos de estas aplicaciones son cualquiera de las versiones adaptadas de los algoritmos clásicos, el procesado de matrices triangulares superiores, la resolución de ecuaciones en derivadas parciales[20], etc.

4.3. Pre-análisis estático

Ya se ha mencionado que el aspecto fundamental de las aplicaciones de carga regular es que dos subconjuntos de datos del mismo tamaño tardan en procesarse lo mismo sobre el sistema, aportan la misma carga; esta propiedad es fundamental, ya que su consecuencia es que existe un único ratio óptimo de división para cualquier combinación de subconjuntos de datos.

4.3.1. La transferencia de datos, ¿es un problema?

Cuando se habla de dividir los datos (en concreto las iteraciones de los bucles de procesado), se hace referencia a transferir un porcentaje de esos datos al acelerador, que los procesará y los transferirá de vuelta; como se ha visto, el entorno de ejecución de descarga es mínimo e independiente del perteneciente al programa principal (incluye la asignación de espacio y recursos para la ejecución en el Xeon Phi), por tanto requiere de una serie de procesos adicionales que suponen un coste temporal.

Una descarga de datos puede dividirse en las siguientes fases:

1. **Inicialización** de la comunicación con el acelerador: establece las conexiones necesarias y prepara el entorno.
2. **Asignación de memoria** en el entorno de descarga: reserva espacio de memoria para el subconjunto de datos que se va a recibir, así como para los que se van a transferir de vuelta.
3. **Transferencia de entrada** hacia el acelerador: se envían los datos a procesar a través del enlace PCI-e.
4. **Procesado** de los datos en el acelerador: éste trabaja sobre su entorno, definiendo y utilizando las variables locales necesarias, transformando los datos recibidos y generando el resultado a devolver.
5. **Transferencia de salida** hacia el procesador: el acelerador devuelve por el enlace PCI-e los datos del resultado.
6. **Liberación de memoria** en el entorno de descarga: las posiciones de memoria utilizadas son desenlazadas del entorno del programa.

El impacto de este proceso va a depender fundamentalmente de la cantidad de datos que se transfieran a/desde el acelerador: cuantos más datos se muevan, mayor será el sobrecoste de la descarga, pero también se aprovechará la potencia del acelerador en la fase de procesado.

Al no ser trivial, se ha procedido a un análisis del modelo de transferencia de memoria; para ello se han hecho distintas pruebas de descarga para varios valores de tamaño de datos, de los que han surgido los resultados observables en el cuadro 4.1, cuya relación se puede ver en la figura 4.4.

Se ha instrumentalizado el código de forma que se puede medir experimentalmente el tiempo empleado en cada una de las fases contempladas, exceptuando la de procesado, ya que depende de la aplicación que descargue; los resultados obtenidos están expresados en milisegundos (ms) y se han verificado haciendo un total de 30 pruebas por cada uno de los datos, cuya media aritmética aparece en la tabla.

Es posible observar cómo el grueso del sobrecoste de transferencia de datos es en su mayoría asignación de espacio de memoria (T_{alloc}); para 1000 MB esta fase ronda los 3 segundos de coste adicional, mientras que el resto de fases apenas alcanzan los 200 milisegundos (aproximadamente

Cuadro 4.1: Análisis del modelo de transferencia de memoria. Resultados en milisegundos (ms).

	Inic.	Asignación de mem.	Transf. entrada	Transf. salida	Liberación de mem.
1 MB	196,651	6,909	0,183	0,205	0,948
5 MB	192,268	18,539	0,898	1,089	0,96
10 MB	194,986	34,25	1,722	2,113	0,969
15 MB	195,221	49,655	2,519	3,223	1,107
30 MB	189,125	94,857	4,872	6,285	0,993
50 MB	191,498	154,704	7,998	10	1,035
75 MB	192,852	229,518	11,944	15,438	1,232
100 MB	192,076	304,445	15,882	20,794	1,146
150 MB	192,243	451,077	23,758	31,187	1,095
200 MB	189,663	599,172	31,601	41,043	1,138
250 MB	193,079	752,747	39,499	52,705	1,377
300 MB	195,997	893,943	47,395	63,027	1,255
400 MB	192,6	1187,953	63,065	84,381	1,269
500 MB	193,094	1477,555	78,801	107,198	1,455
750 MB	192,905	2239,438	118,778	146,307	1,524
1000 MB	195,682	2975,133	158,045	201,607	1,617

un 85 % del total); esto sugiere centrar la atención en minimizar el tiempo de esta fase.

Por otra parte, se puede ver que la inicialización (T_{init}) es constante, con un valor medio de unos 195 ms, lo cual para transferencias pequeñas es un factor a considerar, suponiendo el 96 % en el caso de 1 MB y el 89 % en el de 5 MB; cabe destacar que la fase de inicialización sucede **una única vez** por ejecución del programa, por tanto para transferencias consecutivas sólo afecta a la inicial.

También es notable la asimetría entre el enlace de entrada (T_{in}) y de salida (T_{out}), existiendo en grandes transferencias desviaciones de decenas de milisegundos; esto puede ser debido a la diferencia en el tipo y jerarquía de memoria entre el KNC y el Xeon, lo que se traduce a distintas velocidades de escritura, no obstante no es una diferencia alarmante excepto en casos en los que se transfieran ingentes cantidades de datos desde el KNC.

Por último el tiempo de liberación (T_{free}) de memoria puede considerarse despreciable, situándose entre 1 y 2 milisegundos de sobrecoste.

La naturaleza lineal de las fases permite diseñar un modelo de predicción sencillo pero útil para el programador, ya que puede estimar el sobrecoste de tranferencia de su aplicación, y calcular la viabilidad del modo offload.

Este modelo está definido de forma general en la ecuación 4.1, donde se distinguen las fases de la descarga; por otro lado, en la ecuación 4.2 se especifican los valores concretos para el caso de la investigación, donde x son los megabytes (MB) asignados en el entorno de descarga, y r_{in} y r_{out} son dos constantes que representan el ratio de los datos asignados que son transferidos hacia y desde el Phi, respectivamente; la versión compacta se puede encontrar en la ecuación 4.3.

$$T_{transf\_general} = T_{init} + T_{alloc} + T_{in} + T_{out} + T_{free} \tag{4.1}$$

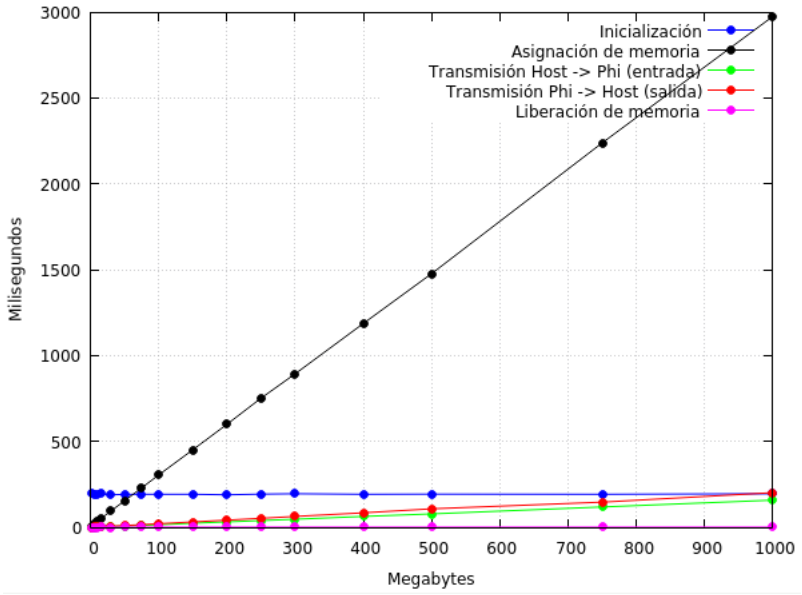

Figura 4.4: Comparación del impacto de las fases de transferencia.

(4.2)
$$T_{transf\_fases} = (195)+(2{,}9693*x+4{,}9813)+(0{,}1579*r_{in}*x+0{,}0682)+(0{,}2013*r_{out}*x+0{,}9083)+(1{,}3)$$

(4.3)
$$T_{transf\_comp} = (0{,}1579 * r_{in} + 0{,}2013 * r_{out} + 2{,}9693) * x + 202{,}258$$

4.3.2. Un modelo de cómputo, primera aproximación

Una vez explorado el impacto de la transferencia de memoria en el rendimiento global, se procede a plantear una manera de proporcionar al programador un ratio de equilibrio óptimo para cualquier aplicación de carga regular; la idea es llegar a la igualdad expresada en la ecuación 4.4, donde ambos dispositivos terminan de procesar su carga al mismo tiempo (en el caso del acelerador se incluye el sobrecoste de transferencia); si se consiguen expresar los tiempos de cómputo/procesado en función de los datos tal y como con el tiempo de transferencia, será posible establecer ese ratio.

(4.4)
$$T_{cómputo\_xeon} = T_{cómputo\_phi} + T_{transferencia}$$

Para poder realizar este planteamiento es necesario identificar una métrica común a todas las aplicaciones sobre la que basar el modelo de cómputo, y una posibilidad es que ésta sea la complejidad temporal algorítmica[21]; esta métrica establece una relación entre el rendimiento temporal de una aplicación en un sistema y el tamaño de los datos de entrada.

Normalmente se expresa en cotas superiores asintóticas o notación *big O*, que ofrece un límite superior del número de operaciones que realizará un algoritmo (ver figura 4.5); además tiene una propiedad muy importante y es que es inherente a la aplicación, lo cual quiere decir que será la misma en cualquier subsistema hardware.

Para entender mejor esto se presenta el siguiente ejemplo: para recorrer un vector de n elementos, se necesita un bucle de n iteraciones y por tanto su complejidad temporal será de *O(n)* operaciones; si queremos recorrer una matriz cuadrada de nxn elementos, se programarán dos

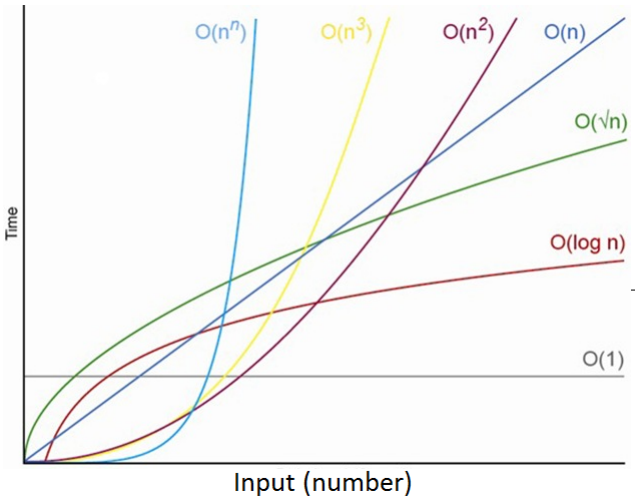

Figura 4.5: Complejidad temporal. La variable n representa el tamaño del problema. Fuente: [22].

bucles de n iteraciones cada uno, donde cada iteración del bucle externo realizará n del bucle interno, constituyendo una complejidad de $O(n^2)$ operaciones.

Una complejidad de $O(n)$ indica que el algoritmo posee una ecuación temporal de tipo $t = a * n + b$, tanto en el Xeon como en el KNC, no obstante las constantes a y b variarán en función de la velocidad del subsistema; asumiendo que el KNC es más rápido computando que el host Xeon, $a_{phi} < a_{xeon}$; lo mismo se aplica para $O(n^2)$, donde $t = a * n^2 + b * n + c$, y es extensible al resto de complejidades.

De aquí se obtiene uno de los puntos clave del análisis, y es que la diferencia de rendimiento entre el Xeon y el KNC es mayor cuanto mayor sea la complejidad temporal de la aplicación y el tamaño de los datos; un escenario no muy alejado de la realidad que nos puede servir como argumento es el siguiente:
Un algoritmo A tarda $T = 7*n+1$ en el host Xeon y $T = 2{,}3*n+1{,}5$ en el KNC (c. temporal O(n)); otro algoritmo B tarda $T = 2{,}1*n^2+0{,}7*n+2$ en el host Xeon y $T = 1{,}6*n^2+0{,}5*n+4$ en el KNC (c. temporal O(n^2)); para un tamaño de datos $n = 5$, la diferencia absoluta de tiempo entre el Xeon y el Phi para A es de $36 - 13 = 23$ ms, mientras que para B es de $58 - 46{,}5 = 11{,}5$ ms; se puede observar que para este valor de n, el algoritmo lineal se aprovecha más del Phi que el cuadrático; no obstante, si aumentamos el valor a $n = 100$, la diferencia para A pasa a ser de $701 - 231{,}5 = 469{,}5$ ms, y en el caso de B $21072 - 16054 = 5018$ ms.

Es evidente que el algoritmo cuadrático se aprovecha mucho más del rendimiento del Phi cuando crece el tamaño de los datos, ya que ejecuta por más tiempo, sin embargo el aspecto importante a obtener de este sencillo análisis es preguntarse hasta qué punto es viable descargar algoritmos lineales teniendo en cuenta que el sobrecoste de transferencia también es lineal.

Buscando los polinomios interpoladores

La idea de la interpolación para este contexto es obtener un polinomio a partir de un subconjunto de muestras sobre la función temporal a interpolar F; es una técnica de análisis numérico útil para aproximar valores de una función.

En este caso se busca un método para obtener el ratio óptimo respecto a distintos valores de tamaño de datos; esto es debido a que, deduciendo de lo explicado en el anterior apartado, un

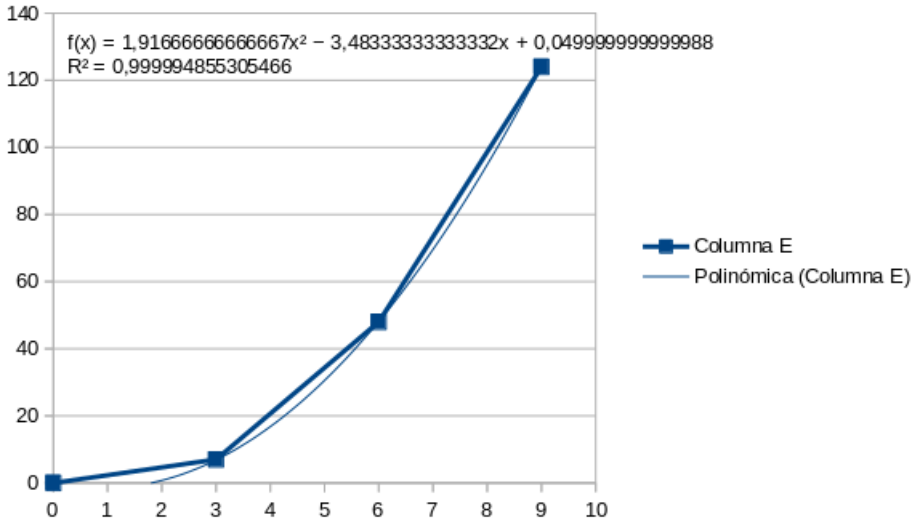

Figura 4.6: Interpolación de una función de coste computacional O(n^2) en LibreOffice Calc. Los percentiles elegidos sobre el intervalo [0,10] han sido 0/30/60/90 %. El polinomio $f(x)$ es el resultado aproximado de la función real, y R^2 es el coeficiente de determinación.

mayor tamaño de datos aumentará linealmente el sobrecoste de transferencia, pero no lo hará así la diferencia de rendimiento en los tiempos de cómputo; si esta diferencia crece a mayor velocidad que la transferencia, el ratio incrementará también a favor del Xeon Phi (más datos para el acelerador, asumiendo que éste siempre es más rápido).

El primer paso es definir un intervalo para el tamaño de datos que se va a utilizar; por ejemplo, si se sabe que se van a utilizar vectores de 10k hasta 100k elementos, no es necesario interpolar el polinomio para valores fuera de ese subconjunto; si es muy variable, la precisión de este método disminuirá, y los ratios serán menos efectivos; en ese caso, podrían obtenerse múltiples ratios para distintos intervalos de tamaño de datos.

Una vez definido el intervalo, se procederá a aplicar mediciones sobre los percentiles[1] más significativos; dependiendo del tiempo que se desee invertir en el análisis, se realizarán más o menos muestras, cuantas más muestras el polinomio interpolado será más representativo de la función F de coste computacional (mayor coeficiente de determinación[2]); lo ideal es realizar muestras sobre percentiles suficientemente separados, para obtener información sobre más áreas de la función; para realizar la interpolación se puede recurrir a herramientas como LibreOffice Calc o a lenguajes matemáticos como R (ver figura 4.6).

El proceso hay que realizarlo tanto para el Xeon como para el Xeon Phi, de tal forma que se obtengan sus dos ecuaciones de modelo de coste computacional; las unidades de éstas han de ser acordes al modelo de transferencia: la variable independiente x será el tamaño del problema en MB, mientras que la variable dependiente y será el tiempo transcurrido en milisegundos.

Una vez modeladas las tres funciones temporales, es necesario adecuarlas a la igualdad expresada en 4.4, resultando una ecuación general reflejada en 4.5 para complejidades **polinómicas**.

- Las constantes k representan los coeficientes del polinomio interpolado de grado n.

[1] Valor (del tamaño de datos) bajo el cual se encuentra el X % del intervalo

[2] Mide la calidad del modelo respecto a la función real, y por tanto su capacidad de predecir valores dentro del intervalo. Se representa como R^2.

- R es el ratio óptimo, que depende directamente del valor de x para la igualdad, y representa el porcentaje de datos que se descargará al acelerador.
- Dim es la dimensión de las estructuras de datos, si son vectores Dim tomará el valor 1, si son matrices será igual a 2, esto es necesario para ajustar el impacto de la transferencia de datos.

$$\begin{aligned} & k_{n-xeon} * ((1-R)*x)^n + k_{(n-1)-xeon} * ((1-R)*x)^{n-1} + ... + k_{0-xeon} = \\ & k_{n-phi} * (R*x)^n + k_{(n-1)-phi} * (R*x)^{n-1} + ... + k_{0-phi} + \\ & (0,1579 * r_{in} + 0,2013 * r_{out} + 2,9693) * R * Dim * x + 202,258 \end{aligned} \tag{4.5}$$

El usuario puede adaptar esta ecuación a sus mediciones, y utilizarla al principio de un programa para calcular de manera aproximada el ratio óptimo para su aplicación dado un tamaño de datos desconocido en tiempo de compilación.

Nota: una aplicación con complejidad temporal polinómica de grado n tendrá n bucles anidados; las expresiones $((1-R)*x)^i$ y $(R*x)^i$ asumen que las iteraciones de **todos** los bucles anidados se reducen en $(1-R)$ y R respectivamente, por eso el propio ratio también se eleva a la potencia; existen aplicaciones que reducen las iteraciones del bucle externo únicamente, manteniendo la carga interna inmutable (ver caso práctico en 5.3.2); en estos casos el ratio se ha de extraer de la potencia, adaptando las expresiones a $(1-R)*x^i$ y $R*x^i$.

4.3.3. Identificación de algunos problemas, un cálculo más exacto

El valor del método de interpolación reside tanto en su adaptabilidad a distintos tamaños de datos como en su sencillez; a pesar de esto, tiene algunas desventajas que es importante entender y destacar:

- Asume estructuras de datos dimensionalmente homogéneas, es decir, vectores del mismo tamaño o matrices de números de columnas y filas similares; por ejemplo, si los datos de entrada estuviesen compuestos por matrices NxM, donde $N >> 0$, $M >> 0$ y $(N-M) >> 0$, caso típico en el procesado de imágenes, la complejidad temporal dependería de dos variables y sería $O(N*M)$, cuyo efecto replantearía fundamentalmente el modelo.
- En algunos casos, los modelos de las interpolaciones pueden ser poco representativos, ya que han tenido en cuenta mediciones sesgadas por eventos del sistema que no se han podido contemplar a priori (tamaños de problema que llenan la memoria caché, peticiones de recursos copadas e impacto de las comparticiones entre hilos, ...); en este sentido, el ratio obtenido no será una buena aproximación al óptimo y el rendimiento será notablemente menor.

El modelo obtenido en esta investigación está definido para complejidades polinómicas, no obstante se podrá adaptar con facilidad a otras complejidades (logarítmica, exponencial, ...), ya que el único factor a variar es la morfología de las ecuaciones computacionales.

Este apartado se enfocará en reducir la inexactitud de los resultados obtenidos, de tal forma que se pueda asegurar al usuario un ratio óptimo independiente de los efectos adversos en el entorno de ejecución; para ello, como siempre, se asume que el acelerador es más rápido que el host Xeon.

Acabar al mismo tiempo, el método de bisección aplicado al desfase temporal

En el apartado 4.3.2 se ha buscado de forma teórica el ratio raíz de la ecuación 4.5; buscar la raíz de esa igualdad se puede abstraer a un concepto de más alto nivel, que es que el tiempo

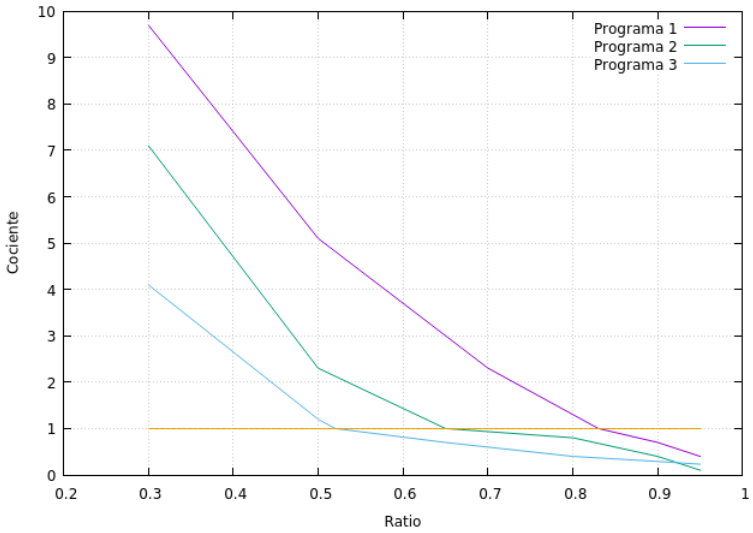

Figura 4.7: Ratio óptimo para un cociente de 1.0 en tres programas distintos.

empleado en el Xeon sea igual que el invertido en el Phi, o lo que es lo mismo, que terminen su trabajo a la vez.

Como se ha explicado antes, emplear modelos es asumir un margen de error que puede llegar a ser considerable; en este sentido, es interesante alejarse de ellos y actuar sobre el escenario real para poder obtener resultados más fiables; para ello, se propone un método clásico y bastante sencillo para obtener el ratio óptimo a través de mediciones sobre distintos ratios, el denominado *método de bisección*.

Éste trabaja en encontrar las raíces de una ecuación sobre una única variable; en este caso la variable será el **cociente temporal** entre ambos sistemas ($T_{xeon}/T_{phi+transf.}$); véase el siguiente ejemplo para facilitar la comprensión:

Una aplicación tarda 10 segundos en el Xeon y 4 en el Phi (incluido el tiempo de transferencia); si el ratio R es $1{,}0 - \epsilon$, el programa se descarga prácticamente en su totalidad sobre el acelerador, y el cociente es $(0{,}0 + \epsilon)/4{,}0 \ = 0{,}0$; si R es $0{,}0 + \epsilon$, la aplicación se ejecutará íntegramente en el Xeon, por tanto el cociente será de $10{,}0/(0{,}0 + \epsilon) \ = \infty$.

Se desea un ratio R_{opt} que genere un cociente $T_{xeon}/T_{phi} = 1{,}0$, lo que significa que ambos tiempos son iguales, por ejemplo $T_{xeon} = 3$ y $T_{phi} = 3$, cuyo tiempo total paralelo es el máximo de ambos, $3 < 4$ segundos que tardaba el Phi trabajando sólo.

El aspecto general de la gráfica del cociente temporal se puede observar en la figura 4.7; el método de bisección ayuda a alcanzar el ratio para un cociente de 1 (R_{opt}) a partir de dos ratios iniciales, R_{inf} y R_{sup}; el primero de ellos genera un cociente por encima de 1, mientras que el segundo genera un cociente por debajo de 1; la primera iteración de este método aplica el ratio intermedio entre estos $R_{temp} = (R_{sup} - R_{inf})/2$, que genera un cociente C; si C es 1 (o muy cerca de este valor[3]) termina y el ratio óptimo $R_{opt} = R_{temp}$; si es mayor que 1, $R_{inf} = R_{temp}$, si es menor que 1, $R_{sup} = R_{temp}$, y en ambos casos se repite la iteración, reduciendo el intervalo de ratios posibles.

Este método devuelve un ratio óptimo real, no obstante es más lento que la interpolación y

[3]Existe un margen de aproximación configurable para evitar efectos del entorno de ejecución sobre las mediciones que hagan que nunca se alcance un cociente de 1; cuanto menor sea ese margen, más representativo es el ratio.

además requiere un análisis por cada tamaño del problema, por eso está pensado para aplicaciones que se vayan a ejecutar muchas veces y sobre tamaños de problema limitados.

La utilización del cociente en vez de la diferencia temporal se debe a que es relativo al coste del programa; una diferencia absoluta de 2 segundos en programas que tardan 30 minutos en ejecutarse es insignificante, no obstante en una aplicación como la utilizada anteriormente, que tardaba 10 segundos en el Xeon y 4 en el Phi, es desde luego relevante; por el contrario, un cociente de 1.02 en el primer programa supone una diferencia absoluta de 20-30 segundos, mientras que en el segundo de algunas decenas de milisegundos, adaptando su impacto.

4.4. Dinámico en tiempo de ejecución

Una aplicación de carga irregular presenta bucles cuyas iteraciones tienen un coste considerablemente distinto; esto genera un problema fundamental en la fase de reparto de carga, ya que, dependiendo de las iteraciones que reciba un sistema, ofrecerá un rendimiento mayor o menor.

Casos típicos de aplicaciones con este comportamiento son las que trabajan con estructuras de datos dispersas, matrices triangulares, simulaciones adaptativas[23], etc.; paralelizar estas aplicaciones, sobre todo cuando los datos de entrada difieren significativamente entre ejecuciones, implica asumir una distribución de la carga subóptima.

Para ejecuciones en un único sistema, OpenMP ofrece métodos de planificación alternativos al ya mencionado y utilizado *static*; en concreto la planificación *dynamic* o dinámica va asignando *chunks* de iteraciones a los hilos según van terminando su trabajo, de tal forma que se maximice la actividad de estos y por tanto minimice el tiempo transcurrido; este tipo de métodos supone un sobrecoste en tiempo de ejecución, ya que el planificador habrá de consumir recursos del sistema para poder actuar.

En este caso se propone un algoritmo similar para equilibrar la carga entre el Xeon y el acelerador Phi, de tal forma que sea posible abstraer las diferencias de carga entre iteraciones y conseguir un buen rendimiento en la co-ejecución.

4.4.1. Funcionamiento del algoritmo

La primera fase es la división de los datos de entrada en paquetes de un tamaño fijo denominados *segmentos*; el segmento es la unidad mínima de trabajo a asignar a un sistema participante; en el host Xeon existirán dos hilos, uno para cada sistema, encargados de asignar segmentos a estos hasta que la carga total de trabajo haya terminado.

La sincronización entre estos dos hilos se hará a través de un **índice global** de segmento, que identificará el próximo segmento a procesar; este índice será una variable compartida, y por tanto su acceso se hará de forma atómica, protegido por una región crítica[24].

Estos dos hilos tendrán un pequeño impacto sobre el rendimiento del procesado de segmentos en el host Xeon, ya que utilizarán la CPU para su objetivo, tal y como lo hace el planificador del runtime[4] en OpenMP; se puede ver una representación del algoritmo en la figura 4.8; destacar que el *segmento resto* se trata como uno común, pero su carga es menor; su existencia se debe a que el tamaño de los datos no tiene por qué ser divisible por la longitud de segmento.

[4]Es el tiempo de ejecución de un programa, en el caso de OpenMP el planificador de chunks se ejecutada en este tiempo como un proceso auxiliar, a la vez que el programa.

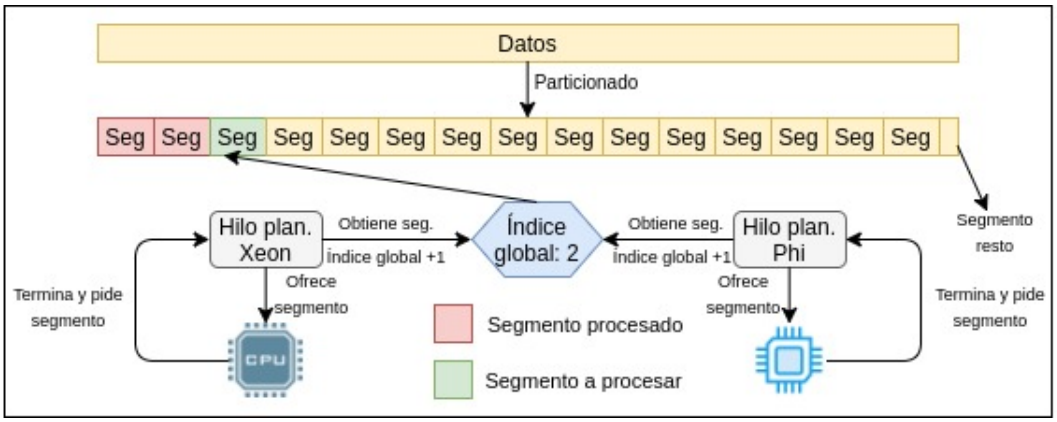

Figura 4.8: Algoritmo dinámico de asignación de segmentos.

Existen técnicas como el *padding*, que consiste en añadir datos no significativos hasta que el tamaño total obtenga la mencionada propiedad, no obstante en este caso no es necesario y probablemente contraproducente.

4.4.2. Efectos de la longitud de segmento elegida

La razón por la que hasta ahora no se ha ofrecido una longitud de segmento fija es por sus dependencias con la aplicación; existen tanto ventajas como inconvenientes si la elección es una longitud grande o pequeña:

- Una longitud grande reducirá el sobrecoste de asignación de segmentos, ya que el número de estos será menor; por otro lado aumentará la desigualdad entre ejecuciones, lo cual es problemático especialmente si la aplicación es de un coste temporal alto, ya que los últimos segmentos pueden tener una diferencia de carga apreciable y producir un desfase de finalización entre sistemas inadecuado.
- Una longitud pequeña evita las grandes diferencias entre procesado de segmentos, lo que impactará en un desfase de finalización aceptable, no obstante la planificación de segmentos supondrá un mayor sobrecoste en el host Xeon; además, un factor importante a tener en cuenta es que una longitud muy pequeña puede suponer un tiempo de transferencia al acelerador igual o mayor que el de procesado.

En este sentido, se recomienda probar con distintas longitudes hasta encontrar el rendimiento deseable.

4.4.3. Reutilización de la memoria en el acelerador, reduciendo el sobrecoste

En el apartado 4.3.1 se enumeraron las distintas fases de la transferencia al acelerador; en concreto se pudo observar que la asignación de memoria suponía un coste diferencial respecto al resto de fases, llegando al orden de segundos para transferencias grandes.

El hecho de que la longitud de segmento sea fija aporta una ventaja clara, y es que el espacio de memoria requerido por un segmento es del mismo tamaño para todos ellos; esto permite reciclar la memoria asignada, eliminando esta fase para las transferencias consecutivas, lo cual es posible gracias a que la memoria dinámica del Xeon Phi es **persistente** entre descargas.

Como reflexión, es posible que en el caso de tranferencias de elevada cantidad de datos un particionado en varias subtransferencias con reutilización de espacio de descarga genere un rendimiento mayor.

Capítulo 5

Evaluaciones y resultados

El objetivo de este apartado es ofrecer al lector una visión del potencial del Xeon Phi sobre aplicaciones típicas de HPC; para ello se han llevado a cabo una serie de pruebas cuya misión es verificar la practicidad de las ideas desarrolladas en los capítulos anteriores.

En términos generales, estas evaluaciones están destinadas a mediciones de rendimiento, entendiéndose éste como el desempeño temporal de una aplicación sobre un sistema; el concepto fundamental asociado a este tipo de mediciones es el **speedup** o aceleración, que expresa la mejora de rendimiento en la ejecución de una aplicación sobre dos configuraciones o sistemas distintos.

La medición se ha realizado en todos los casos directamente sobre el núcleo de cómputo del programa, dejando de lado el resto de fases tradicionales (definiciones de variables, manejo de memoria, entradas y salidas, ...), exceptuando la propia descarga al acelerador, que es obviamente un factor intrínseco de su etapa de procesado; la intención es mostrar la capacidad del acelerador en tareas en las que está especializado, si se contemplasen otras secciones existiría un sesgo en los resultados debido a las leyes de impacto secuencial[25].

5.1. Descripción del entorno de evaluación

Todo el trabajo realizado en esta investigación, incluido este apartado de pruebas se ha realizado sobre la siguiente configuración:

- Un nodo principal denominado "Batel", consistente en dos procesadores Intel Xeon E5-2620[26] conectados a una placa de doble zócalo[1]; cada uno de ellos posee 6 núcleos de proceso con 2 hilos hardware, lo que hace un total de 24 hilos en el nodo. El núcleo funciona a 2 GHz de frecuencia de reloj base, con un pico de 2,5 GHz teórico con cargas máximas (empíricamente se ha observado un máximo de 2,3 GHz), tiene 15 MB de caché L3 y ejecuta fuera de orden; la conexión entre los dos procesadores se realiza a través de QPI hasta $7{,}2 * 10^9$ transferencias por segundo, lo cual permite observar al sistema como un nodo global de un único procesador de 12 núcleos. Posee un total de 16 GB de memoria DDR3 a 42,6 GB/s de ancho de banda máximo, y presenta soporte para extensiones vectoriales AVX con 256 bits de anchura de registro. El sistema operativo que ejecuta sobre el nodo es un CentOS Linux 7.2.

- Un coprocesador Intel Xeon Phi 7120P[27] con la arquitectura descrita en 2.1.1, conectado a través de un bus PCIe al nodo "Batel" y accesible a través de este último mediante un túnel SSH; el coprocesador corre un sistema independiente del ejecutado en el nodo, en

[1] El zócalo es la zona de conexión para la CPU en la placa base, una placa de dos zócalos permite la conexión de dos procesadores.

este caso un Linux de la familia Red Hat/CentOS con versión de núcleo 4.4, asociado a una capa software MPSS 3.8.1 que se instala en el host.

- El compilador de Intel para C/C++ (ICC) con versión 17.x; es fundamental utilizar este compilador si se quiere reproducir el experimento u obtener resultados equivalentes, ya que sus procesos de auto-vectorización, diagnóstico y optimización son muy superiores en general a los del compilador libre de GNU (GCC); en concreto para Intel MIC este último es incapaz de utilizar las instrucciones IMCI para auto-vectorizar. ICC está instalado en el host Xeon, por tanto las aplicaciones ya sean para el host o para el coprocesador se compilarán desde ahí, copiando los ejecutables en el caso del KNC. Destacar que las directivas relacionadas con las instrucciones vectoriales (*simd*) y la descarga (*offload*) ofrecidas por OpenMP aparecen a partir de la revisión 4.0, y ésta es soportada oficialmente a partir de la versión 15.0 de ICC, por tanto versiones del compilador inferiores a ésta no son viables.

5.2. Aplicaciones utilizadas

Para probar la viabilidad de los métodos se han utilizado dos aplicaciones, una de carga regular y otra de irregular, de tal forma que sea posible observar los efectos de cada idea sobre el tipo de carga a la que está dirigida.

5.2.1. Carga regular: simulación N-body

Esta aplicación se encarga de reproducir los efectos gravitacionales entre un conjunto de partículas o *cuerpos* en un espacio tridimensional; cada una de éstas tiene una masa definida y, en un momento determinado, una posición y una velocidad en el espacio; el algoritmo modifica estos dos últimos valores acorde a los efectos gravitacionales del resto de cuerpos para cada partícula, y lo hace en intervalos de tiempo definidos dt.

Es una herramienta muy utilizada en astrofísica para estudiar la evolución de grupos de cuerpos celestes (estrellas, planetas, galaxias, ...) condicionados por las fuerzas de la gravedad; en las mediciones llevadas a cabo se utilizarán distintos tamaños de problema definidos por el número de cuerpos participantes en la simulación.

En general, para cada intervalo dt o Δt el algoritmo calculará la variación de la velocidad y la posición de cada partícula atendiendo a las ecuaciones 5.1 y 5.2 respectivamente, derivadas de las leyes de Newton.

$$\Delta \vec{v}_i = G * \Delta t * \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{m_j}{r_{ij}^3} * \vec{r}_{ij} \tag{5.1}$$

$$\Delta \vec{r}_i = \vec{v}_i * \Delta t \tag{5.2}$$

Sin entrar en más detalles sobre este problema (si es de interés, se puede obtener más información en libros como [28]), se puede observar una representación visual del escenario de la simulación en la figura 5.1 y el código base para un intervalo (una iteración) sobre Δt en el apéndice A.

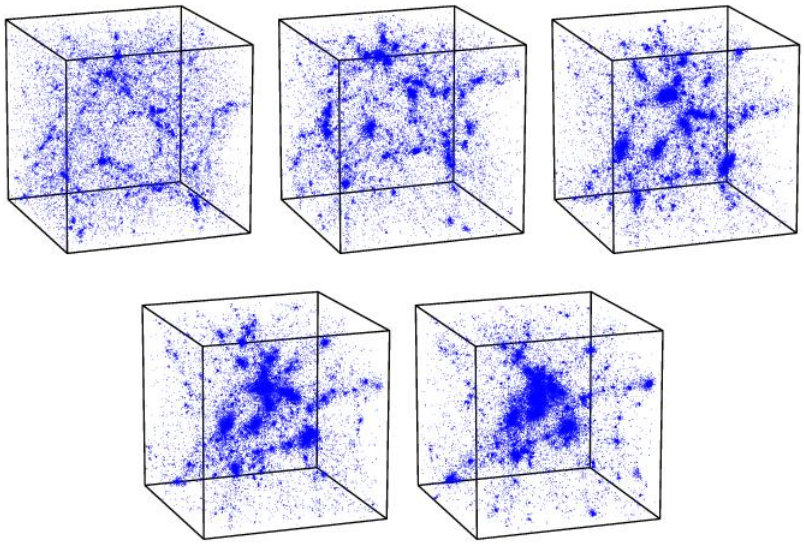

Figura 5.1: Evolución de la simulación N-body en la que los cuerpos tienden a colapsar debido a las fuerzas gravitacionales. Fuente: [29]

5.2.2. Carga irregular: filtrado específico de cualidades en una imagen

El procesamiento digital de imágenes es una de las aplicaciones clásicas a la hora de utilizar arquitecturas paralelas; una imagen es normalmente representada mediante una matriz de píxeles[2] cuyos valores están relacionados con las cualidades presentes en ella, como es el color (luminosidad, tono, saturación, ...); el procesado o filtrado de una imagen consiste en modificar ciertas características de ésta para hacerla más apropiada de cara a su utilización en otras tareas; ejemplos de esto pueden ser el suavizado de bordes o la eliminación de ruido.

Existen muchas maneras de filtrar una imagen ya que depende fundamentalmente de lo que se quiera obtener como resultado; en este caso, se va a trabajar con un procesado concreto que va a ser adecuado para el tipo de aplicación que estamos buscando; en efecto se va a modificar la saturación o intensidad de un color c en una imagen siguiendo un patrón determinado.

La imagen en sí será una matriz A de dimensiones MxN, donde M es el número de píxeles horizontales (filas) y N el de verticales (columnas); cada elemento de esta matriz reflejará la intensidad de c en ese píxel, que vendrá definida por un valor decimal s, donde $s = 1{,}0$ es la máxima saturación y $s = 0{,}0$ es la mínima.

El filtro de modificación será un vector B de dimensión N donde cada elemento con índice i tendrá un valor $B[i] = |N/2 - i|/(N/2)$, de tal forma que los elementos se acerquen a 1,0 a medida que se alejan del centro del vector; la operación de filtrado consistirá para cada píxel $A[i][j]$ en sumar a su valor de s los valores $A[i][j] * B[x]$, donde $x < j$ y restarle $A[i][j] * B[y]$ donde $y > j$, finalmente adecuándolo con el módulo al intervalo $[0{,}0, 1{,}0]$; así, los valores de las columnas centrales se mantendrán o variarán mínimamente su saturación, mientras que los de las columnas exteriores sufrirán cambios considerables (ver figura 5.2).

Si la intensidad es mínima ($s = 0{,}0$), $A[i][j] = 0{,}0$, por tanto la modificación será una operación inútil; recordando lo mencionado en el apartado 4.2.2, en estos casos hay que considerar si es posible evitar el sobrecoste de esa operación, sobre todo teniendo en cuenta que pueden

[2]Menor unidad de cualidades homogéneas en una imagen.

existir imágenes en las que la presencia de c sea baja (los elementos de la matriz son 0.0 en su mayoría); para ello, se va a utilizar una técnica denominada CSR, del inglés *Compressed Sparse Row*, fila dispersa comprimida.

CSR transforma una matriz en tres vectores: un vector de valores val de dimensión S, donde S es el número de elementos no nulos de la matriz, que contendrá todos estos en orden de recorrido por filas; otro vector de punteros a fila $rptr$ de dimensión M, donde M es el número de filas de la matriz en cuestión, y que contiene en el elemento $rptr[i]$ un índice j que indica que $val[j]$ es el primer elemento de la fila i; y por último un vector col de dimensión S que contiene en el elemento $col[j]$ el índice k de la columna de $val[j]$; para poder entender esto, se facilita el siguiente ejemplo:

$$A = \begin{pmatrix} 0{,}6 & 0{,}0 & 0{,}0 & 0{,}0 & 0{,}6 \\ 0{,}0 & 0{,}2 & 0{,}5 & 0{,}2 & 0{,}0 \\ 0{,}0 & 0{,}0 & 0{,}3 & 0{,}0 & 0{,}0 \\ 0{,}0 & 0{,}8 & 0{,}0 & 0{,}8 & 0{,}0 \\ 0{,}0 & 0{,}0 & 0{,}8 & 0{,}0 & 0{,}0 \end{pmatrix} \qquad B = \begin{pmatrix} 1{,}0 \\ 0{,}5 \\ 0{,}0 \\ 0{,}5 \\ 1{,}0 \end{pmatrix}$$

La matriz A es la representación de una imagen, en la que el color c se presenta en las intensidades reflejadas en cada píxel; B es el vector de filtrado a aplicar sobre A; ésta es una matriz dispersa, posee muchos elementos nulos, así que se utilizará CSR para transformarla:

$$val = \begin{pmatrix} 0{,}6 & 0{,}6 & 0{,}2 & 0{,}5 & 0{,}2 & 0{,}3 & 0{,}8 & 0{,}8 & 0{,}8 \end{pmatrix}$$
$$rptr = \begin{pmatrix} 0 & 2 & 5 & 6 & 8 \end{pmatrix}$$
$$col = \begin{pmatrix} 0 & 4 & 1 & 2 & 3 & 2 & 1 & 3 & 2 \end{pmatrix}$$

Atendiendo a la explicación del CSR, para $i = 1$, por ejemplo, $rptr[1] = 2$, lo cual quiere decir que $val[2] = 0{,}2$ es el primer elemento no nulo de la fila $i = 1$, y $col[2] = 1$ es la columna de ese elemento; se puede observar el pseudocódigo utilizado en (1); el bucle externo a paralelizar en este algoritmo itera sobre el número de elementos contenidos en el segmento de filas a procesar, cuya naturaleza es variable; habrá filas (y segmentos) con más elementos (e iteraciones) que otros, y por tanto la carga será de carácter irregular; esto implica que, dependiendo de los segmentos que acceda cada componente, tardará más o menos en acabar su trabajo.

Es importante entender el funcionamiento de aplicaciones como la descrita para comprender el valor de los algoritmos dinámicos y la razón de su existencia; la carga irregular es un problema en las aplicaciones paralelas que afecta considerablemente al rendimiento global de éstas, para aprender más sobre él se recomiendan lecturas como [30], que realizan un estudio de cómo identificarla y clasificarla.

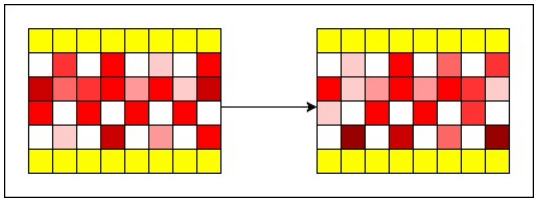

Figura 5.2: Filtrado de una imagen de 8x6 píxeles para c rojo. Se observa cómo la saturación central se mantiene y la exterior varía.

Algorithm 1 Cómputo del filtrado con CSR.

```
for i = rptr[offset] to rptr[tam_seg] - rptr[offset] do
  res = 0.0
  for j = 0 to col[i] do
    res += val[i] * filtro[j]
  end for
  for j=col[i] + 1 to N do
    res -= val[i] * filtro[j]
  end for
  val[i] = abs( res % 1.0 )
end for
```

5.3. Desarrollo de las pruebas

En total se han llevado a cabo cuatro evaluaciones para observar los efectos de los cuatro principales conceptos mencionados en esta investigación: optimización de una aplicación para el acelerador Xeon Phi, equilibrio de carga estático mediante interpolación, estático por método de bisección y dinámico.

5.3.1. Optimización de la simulación N-body para el Xeon Phi

Partiendo del código secuencial presentado en el anexo A, se han realizado dos iteraciones sobre él para mejorar su rendimiento.

Primero se ha procedido a paralelizar el programa y a configurar su entorno de ejecución (ver fragmento de código 5.1); en cada iteración del algoritmo existen dos bucles principales, el de actualización de las velocidades y el de actualización de las posiciones; a simple vista, estos bucles podrían fusionarse en uno sólo, no obstante se van a mantener aislados para facilitar al compilador los posteriores procesos de vectorización.

```
#pragma omp parallel private( i, j, dvx, dvy, dvz, dx, dy, dz, dist2, mjOverDist3 )
{
    #pragma omp for schedule( static )
    for( i = 0; i < particulas; i++ )
    {
        ...
        vx[i] += dvx * dtG;
        vy[i] += dvy * dtG;
        vz[i] += dvz * dtG;
    }
    #pragma omp for schedule( static )
    for( i = 0; i < particulas; i++ )
    {
        x[i] += vx[i] * dt;
        y[i] += vy[i] * dt;
        z[i] += vz[i] * dt;
    }
```

Listing 5.1: Bucles de proceso paralelizados.

La directiva *#pragma omp parallel* asume por defecto dos factores importantes:

i) si no se especifica su opción *num_threads(x)*, generará tantos hilos como los especificados en *OMP_NUM_THREADS*; como se desea utilizar todo el potencial del Phi para este programa,

el valor de esta variable será 244:

"[user@hostname dir]$ export OMP_NUM_THREADS=244"

ii) cualquier variable que no se incluya ni en la opción *private* ni en la *shared* es por defecto **compartida**; en este caso (y en la mayoría), las estructuras de datos de entrada (x, vx, m, ...) son compartidas, ya que distintos hilos escriben sobre distintas posiciones de memoria y por tanto no hay conflictos; otras variables compartidas son las llamadas de "sólo lectura", como por ejemplo el tamaño de las estructuras *particulas*, lo cual ahorra hacer una copia de ésta a cada hilo; privadas son todas las variables que el hilo utiliza para hacer sus cálculos, y se definen explícitamente en la opción *private*.

La planificación de los bucles es estática con la opción *schedule(static)*; se utiliza esta opción porque la carga de ambos bucles es **regular**, y por tanto el trabajo de cada hilo será prácticamente el mismo.

Por último se establece la distribución de los hilos en el Phi a *balanced*, que es la opción que generalmente va mejor en este sistema (ver apartado 3.2):

"[user@hostname dir]$ export KMP_AFFINITY=balanced"

El código resultante completo se puede observar en el anexo B.

Una vez paralelizado se procede a vectorizar, así como a optimizar algunos aspectos; si bien se suele aplicar la paralelización a los bucles externos, la vectorización se realiza sobre los internos, ya que son estos los que contienen las operaciones fundamentales; esto es importante, un bucle con una variedad de operaciones considerable o con estructuras anidadas no es candidato a ser vectorizado, ya que el coste de vectorizarlo es más alto que el coste base.

En el fragmento de código 5.2 se pueden observar los cambios más importantes aplicados tanto al proceso principal como a la simulación N-body.

```
int main( int argc, char **argv )
{
    ...
    x = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    y = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    z = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    vx = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    vy = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    vz = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    m = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    ...
}

void newton_opt( const size_t particulas, const double dt, double *x, double *y,
    double *z, double *vx, double *vy, double *vz, const double *m )
{
    ...
    #pragma omp parallel private( i, j, dvx, dvy, dvz, dx, dy, dz,
    dist2, mjOverDist3 )
    {
        #pragma omp for schedule( static )
        for( i = 0; i < particulas; i++ )
        {
            ...
```

```
            #pragma omp simd aligned( m : 64 )
            for( j = 0; j < i; j++ )
            {
                ...
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );
                ...
            }

            #pragma omp simd
            for( j = i + 1; j < particulas; j++ )
            {
                ...
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );
                ...
            }
            ...
        }

        #pragma omp for simd schedule( static )
        for( i = 0; i < particulas; i++ )
        {
            ...
        }
    }
}
```

Listing 5.2: Optimizaciones y vectorización.

Los bucles internos y el de actualización de posiciones han sido vectorizados mediante *#pragma omp simd*.

La utilización de *aligned_alloc*(64U, ...) permite alinear las estructuras de datos a 64 bytes, que es el tamaño de línea de caché del Phi; esto mejora los accesos como se comentó en el apartado 3.4.1, pero además permite utilizar la opción *aligned*(m : 64) de la directiva *simd*; esto le indica al compilador que los accesos iniciales de ese bucle están alineados a 64 para la estructura m, lo que reduce los movimientos a registros vectoriales desde caché; como vemos el acceso a m depende únicamente de j, que en ese bucle empieza en 0, por tanto alineado.

En el bucle de $i+1$ a *particulas* esto **no** se puede asumir, ya que $m[i+1]$ es dependiente del hilo que está ejecutando ese bucle; este último caso se aplica a los accesos del resto de estructuras de datos, todas dependientes de i y por tanto del hilo que accede; para paliar esto veíamos en el apartado 3.4.2 técnicas como el ajuste de tamaño de chunk, no obstante después de aplicar éstas empíricamente se ha observado un rendimiento **menor** para esta aplicación.

La razón de esto puede deberse al tratamiento del compilador: con las optimizaciones activas, éste modifica el código desarrollado de manera significativa, y estos ajustes pueden impedirle realizar esos procesos con éxito; de nuevo, esto ha sido comprobado para esta aplicación en concreto, la técnica de alineamiento de índices sigue teniendo una base teórica y debería ser considerada en cualquier implementación.

El código completo vectorizado así como el programa principal se encuentra en el anexo C; además del código es fundamental el proceso de **compilación**, que difiere para cada versión de la aplicación:

Secuencial:
"[user@hostname dir]$ icc -mmic -no-vec -o phi_nbs.out -O2 -std=c11 -w3 n_body_seq.c n_body_utils.c"

Paralela:
"[user@hostname dir]$ icc -mmic -no-vec -o phi_nbp.out -O2 -qopenmp -qopt-report=5
-qopt-report-phase=openmp -qopt-report-file=phi_nbp.opt.rpt -std=c11 -w3 n_body_par.c n_body_utils.c"

Optimizada (paralela y vectorizada):
"[user@hostname dir]$ icc -mmic -o phi_nbo.out -O2 -qopenmp -qopt-report=5
-qopt-report-phase=vec,openmp -qopt-report-file=phi_nbo.opt.rpt -std=c11 -w3 n_body_opt.c n_body_utils.c"

La opción *mmic* es común a todas e indica al compilador que la arquitectura destino es Intel MIC (nativo).
no-vec desactiva la vectorización para optimizaciones hasta *O2*, si se utilizase *O3* esta opción no podría ser aplicada.
std=c11 es el estándar del lenguaje C sobre el que se desea que trabaje el compilador, es necesario para poder utilizar *aligned_alloc()*.
qopenmp activa la detección de directivas OpenMP y por tanto la paralelización del código.

La familia *qopt-report-** genera un reporte de los aspectos indicados en *phase*, en este caso *openmp* y/o *vec*, con un nivel de detalle 5 en el fichero especificado en *file*; este reporte contiene información sobre los bucles paralelizados y vectorizados, los accesos alineados, el *speedup* estimado derivado de las optimizaciones y demás datos; se ha utilizado durante la investigación para observar el comportamiento del compilador así como para ajustar el código y producir mejores reportes; se puede ver un ejemplo en el anexo D.

5.3.2. Equilibrio de carga estático mediante interpolación para N-body

En esta sección el código base es el de la simulación N-body optimizada que se ha obtenido previamente; como se puede observar, el núcleo de cómputo de esta aplicación tiene tres bucles, dos externos y uno interno anidado sobre el primero de estos; el tamaño del problema es el valor de *particulas*, por tanto a efectos de complejidad temporal, $N = particulas$; los dos bucles externos iteran sobre N mientras que el interno sobre $N-1$ (se salta la partícula i), por tanto se podría expresar su coste como $N*(N-1)+N = N^2$, lo que refleja finalmente una complejidad de $O(N^2)$.

Una vez sabido que su complejidad es polinómica, en concreto cuadrática, se procede a medir el tiempo para diferentes tamaños de problema en ambos componentes hardware; en este caso se ha optado por definir un intervalo de datos $N = [0, 1000000]$, y se utilizarán los percentiles en 30, 60 y 90 %; antes de nada se ha de compilar la aplicación para el host Xeon, y aquí es donde se ve el valor del acelerador, ya que no es necesario modificar el código en ningún aspecto, simplemente adecuar la línea de compilación:

"[user@hostname dir]$ icc -march=native -o nbo.out -O2 -qopenmp -qopt-report=5
-qopt-report-phase=vec,openmp -qopt-report-file=nbo.opt.rpt -std=c11 -w3 n_body_opt.c n_body_utils.c"

La opción *march=native* es necesaria para que el compilador utilice las características de la arquitectura subyacente, en este caso *Sandy Bridge*, que a efectos prácticos pone a disposición de éste las instrucciones vectoriales *AVX* (256 bits de anchura de registro).

Nota: cuando se mida el rendimiento nativo de la aplicación en el Xeon Phi es necesario especificar un valor de 240 hilos para OMP_NUM_THREADS y *compact* para KMP_AFFINITY;

Cuadro 5.1: Mediciones en milisegundos del tiempo de ejecución del núcleo N-body.

	Xeon	Xeon Phi
(30 %) 16,0217 MB	71018,3	7239,3
(60 %) 32,0435 MB	284755,4	34536,4 7339,33
(90 %) 48,0652 MB	641709	89367,1

esto es así porque el modo descarga reserva un núcleo de proceso del acelerador para gestionar el entorno, por tanto es necesario utilizar únicamente 60 núcleos.

El problema N-body consta de 7 vectores (*x, y, z, vx, vy, vz, m*), por tanto el tamaño del problema en MB para un valor de particulas N es $(N * 8(double) * 7)/(1024 * 1024)$; en el cuadro 5.1 se pueden observar las mediciones obtenidas para los percentiles mencionados en MB; con estos datos es posible interpolar ambas ecuaciones de cómputo (ver figura 5.3), que se reflejan en las expresiones 5.3 y 5.4.

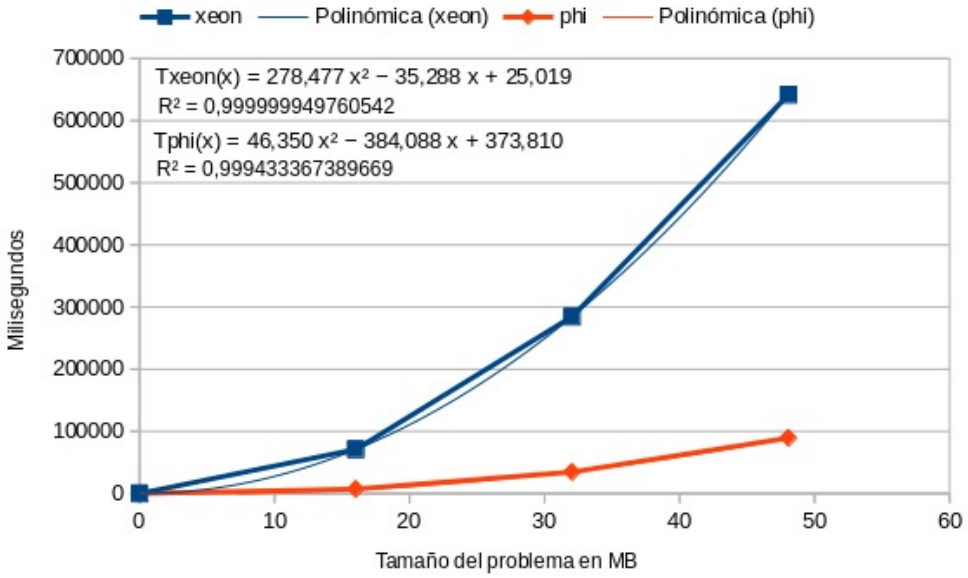

Figura 5.3: Interpolación de las ecuaciones temporales de ambos componentes hardware para la simulación N-body.

$$T_{xeon} = 278{,}47694 * (1 - R) * x^2 - 35{,}28814 * (1 - R) * x + 25{,}01859 \tag{5.3}$$

$$T_{phi} = 46{,}35002 * R * x^2 - 384{,}0883 * R * x + 373{,}80984 \tag{5.4}$$

En este caso concreto ha de hacerse una pequeña adaptación de la ecuación de transferencia; los vectores *(x, y, z, m)* se transfieren completos independientemente del ratio asociado al acelerador, ya que el algoritmo necesita aplicar a la partícula i el efecto de todas las demás; *(vx, vy, vz)* sí que se transfieren en parte, ya que sólo dependen de la propia partícula analizada; además, para *(x, y, z)* sólo se devolverá del Phi las posiciones modificadas, no el vector entero, y m no se devuelve; es posible observar la ecuación adaptada en 5.5.

$$\begin{aligned} T_{transf} = {} & (0{,}1579 * ((3/7) * R + (4/7)) + 0{,}2013 * (6/7) * R \\ & + 2{,}9693 * ((3/7) * R + (4/7))) * x + 202{,}258 \end{aligned} \tag{5.5}$$

Finalmente se llega a la igualdad deseada $T_{xeon} = T_{phi} + T_{transf}$ en la expresión 5.6; a través de esta ecuación es posible aproximar los ratios óptimos para distintos tamaños de problema dentro o cerca del intervalo[3].

$$
\begin{aligned}
&278{,}47694 * (1 - R) * x^2 - 35{,}28814 * (1 - R) * x + 25{,}01859 = \\
&46{,}35002 * R * x^2 - 384{,}0883 * R * x + 373{,}80984 \\
&+ (0{,}1579 * ((3/7) * R + (4/7)) + 0{,}2013 * (6/7) * R \\
&+ 2{,}9693 * ((3/7) * R + (4/7))) * x + 202{,}258
\end{aligned} \tag{5.6}
$$

Se procede a obtener el ratio óptimo para $N = 1000000$ como ejemplo del desarrollo; el tamaño del problema en MB para tipo de datos *double* es $x = (1000000 * 8 * 7)/(1024 * 1024) = 53{,}40576$; si se sustituye y se despeja R, se obtiene un valor de 0,875669, que es el porcentaje de datos que ha de procesar el acelerador para optimizar la combinación de ambas piezas.

Se ha probado con tamaños de problema distintos, por ejemplo para $x = 20$ MB el valor de R es 0,90561, mientras que para $x = 10$ MB el valor de R es 0,951309; la tendencia es a menor tamaño de problema, más datos al acelerador, ya que la ganancia de atribuir carga al procesador disminuye.

Las posiciones originales han de ser descargadas al acelerador previamente a que el procesador las modifique, ya que el cálculo en ese caso sería erróneo; esto conlleva añadir el sobrecoste de inicialización, asignación de memoria y transferencia de los vectores *(x, y, z)* hacia el acelerador de forma secuencial; el impacto de esta situación es considerable, por eso es importante que las aplicaciones descargadas sean lo más independientes posible.

Una manera de solucionarlo es subdividir las operaciones de la carga: en la simulación N-body las posiciones se modifican en el segundo bucle, el bucle O(N^2) sólo modifica las velocidades, que son independientes en cada componente, por tanto es posible procesar éstas mientras se envían los datos al Phi, y establecer un punto de sincronización antes de procesar las posiciones; esto se denomina "solapamiento de cómputo con comunicaciones", es una técnica muy utilizada, y a la vez muy valorada en aquellas aplicaciones en las que puede utilizarse.

Para realizar la descarga se han utilizado las Intel Language Extensions for Offload (LEO), un conjunto de directivas que permiten especificar las acciones a realizar en este contexto; se puede observar una muestra del código resultado en el fragmento 5.3.

Existen dos tipos de directivas principales, *#pragma offload_transfer* y *#pragma offload*; la primera se encarga únicamente de manejar los datos (asignación, entrada, salida, ...), mientras que la segunda indica al compilador que la llamada posterior se ejecutará en el acelerador; la opción *target(mic : 0)* indica el dispositivo al que descargar, en este caso el MIC número 0 (pueden existir varios aceleradores, incluso coexistir con GPUs).

Las opciones aceptadas por Intel LEO para los tipos de transferencia son *in*, *out*, *inout* y *nocopy*, cada una pensada para definir la dirección de los datos; para transferir conjuntos de datos como vectores (*arrays*) o punteros, es necesario especificar una longitud de datos en la opción *length*, así como indicar la asignación de memoria (*alloc_if*) o la liberación de ésta (*free_if*) en el acelerador; por ejemplo, la primera directiva le indica al compilador que asigne memoria en el acelerador (*alloc_if(1)*) para los vectores *(x, y, z, m)* de longitud *particulas*, y para los vectores *(vx, vy, vz)* de longitud *phi_datos*, pero sin copiar el contenido original del procesador (*nocopy*).

[3]Para un intervalo grande como $[0, 1000000]$ se podrían aproximar valores mayores si se han escogido percentiles cercanos al límite superior.

```
#pragma offload_transfer target( mic : 0 )
nocopy( x, y, z, m : length( particulas ) alloc_if( 1 ) free_if( 0 ) )
nocopy( vx, vy, vz : length( phi_datos ) alloc_if( 1 ) free_if( 0 ) )
#pragma offload_transfer target( mic : 0 )
in( x, y, z, m : length( particulas ) alloc_if( 0 ) free_if( 0 ) )
in( vx, vy, vz : length( phi_datos ) alloc_if( 0 ) free_if( 0 ) )
#pragma offload target( mic : 0 )
in( particulas, phi_datos, dt )
nocopy( x, y, z, m : length( particulas ) alloc_if( 0 ) free_if( 0 ) )
nocopy( vx, vy, vz : length( phi_datos ) alloc_if( 0 ) free_if( 0 ) )
newton_vel( particulas, 0U, phi_datos, dt, x, y, z, vx, vy, vz, m );
```

Listing 5.3: Descarga con Intel LEO.

Es necesario habilitar la función a llamar por *#pragma offload* como disponible en el acelerador; para ello se definen los prototipos como accesibles desde éste tal y como se expresa en el fragmento 5.4; el código completo es observable en el anexo E; si se quiere aprender más sobre el modelo de descarga y cómo se gestiona internamente es recomendable leer [31].

```
#pragma offload_attribute( push, target( mic ) )
void newton_vel( const size_t particulas, const size_t lim_inf, const size_t lim_sup
    , const double dt, const double *x, const double *y, const double *z, double *vx
    , double *vy, double *vz, const double *m );
#pragma offload_attribute( pop )
```

Listing 5.4: Definición de prototipo disponible en el acelerador.

Cabe destacar que el entorno de ejecución del KNC utilizado por la descarga es configurable; esto es importante, ya que permitirá planificar la disposición de los recursos desde el propio host; para ello existe una variable de entorno en el procesador, *MIC_ENV_PREFIX*, la cual toma el valor de una cadena de caracteres con la que se identificarán las variables de entorno del acelerador. Por ejemplo, si se establece "export MIC_ENV_PREFIX=MIC", es posible configurar los hilos y la afinidad del KNC desde el procesador haciendo "export MIC_OMP_NUM_THREADS=240" y "export MIC_KMP_AFFINITY=compact".

5.3.3. Equilibrio de carga estático mediante el método de bisección para N-body

Este apartado se basa en la ejecución del programa de descarga desarrollado en la anterior sección para distintos ratios, siguiendo el método de bisección para aproximar el valor óptimo de éste parámetro.

Un aspecto muy importante a la hora de realizar un conjunto de mediciones es la **independencia** entre ellas; cuando un sistema procesa una tarea, el estado de los recursos asociados a ésta es distinto al comenzar a ejecutarla y al terminar; las memorias caché contendrán partes de las estructuras de datos e instrucciones a ejecutar, los predictores de saltos se habrán adecuado al patrón del programa, el acelerador tendrá conexiones abiertas y entornos inicializados, ... En definitiva, si se plantea un bucle de mediciones en el interior del programa, el rendimiento de las consecutivas será mayor, ya que la estructura de los recursos será favorable.

Para evitar esto podemos se puede pensar en dos opciones: realizar una "medición de calentamiento", de tal forma que se descarte su resultado pero se estructuren los recursos homogéneamente para el resto de mediciones, o medir el rendimiento del ejecutable desde otro programa; la principal desventaja del primer caso es que se evalúa el rendimiento de la aplicación "calentada", cuando en el escenario real esto no va a ser así, por tanto se ha procedido a utilizar el segundo

caso.

El algoritmo de medición se ha diseñado como un *shell script* para Bash; éste es un intérprete de comandos utilizado ampliamente en los entornos Linux para comunicarse con el sistema, y es el principal medio para ejecutar los programas desde la terminal; un *shell script* no es más que un conjunto de comandos con una estructura definida que Bash interpretará y ejecutará; a continuación se puede observar el algoritmo desarrollado:

```
# Prepara el entorno del procesador y el acelerador.
export OMP_NUM_THREADS=24
export KMP_AFFINITY=compact
export MIC_ENV_PREFIX=MIC
export MIC_OMP_NUM_THREADS=240
export MIC_KMP_AFFINITY=compact

# Define el intervalo de ratios y calcula sus valores objetivo.
SUP="0.99"
INF="0.01"
MARGEN="0.05"
FSUP=$(./bis_nb.out $SUP)
FINF=$(./bis_nb.out $INF)
FIN=0
while [ $FIN -eq 0 ]
do
    # Calcula el nuevo ratio y su valor objetivo.
    ACT=$(printf "%.3f\n" $(bc <<< "scale=3;($SUP + $INF) / 2.0"))
    FACT=$(./bis_nb.out $ACT)
    echo "Superior: [$SUP, $FSUP] Inferior: [$INF, $FINF] Actual: [$ACT, $FACT]"
    ABSDIFF=$(bc <<< "$FACT - 1.0" | sed -e 's/-//g')
    if [ $(bc <<< "$ABSDIFF < $MARGEN") -eq 1 ]
    then
        # Si los tiempos son suficientemente cercanos termina.
        FIN=1
    elif [ $(bc <<< "$FACT < 1.0") -eq 1 ] && [ $(bc <<< "$FACT > $FSUP") -eq 1 ]
    then
        SUP=$ACT
        FSUP=$FACT
    elif [ $(bc <<< "$FACT < $FINF") -eq 1 ]
    then
        INF=$ACT
        FINF=$FACT
    fi
done
echo "Ratio óptimo: $ACT"
```

Por defecto, Bash únicamente trabaja con números enteros; los valores temporales están expresados en coma flotante, así que se han delegado los cálculos de estos a la herramienta *bc*, que actúa como un lenguaje de apoyo.

El intervalo inicial de ratios ha sido de 0,01 (todo el trabajo en el Xeon) a 0,99 (todo en el Phi), y el tamaño del problema para la simulación N-body ha constado de 1000000 de partículas; el límite de diferencia entre el ratio perfecto ($T_{xeon}/T_{phi} = 1{,}0$) y el medido es de 0,05, supo-

```
[aherrera@batel estatico-biseccion]$ source biseccion.bash
Superior: [0.99, 0.081] Inferior: [0.01, 422.895] Actual: [0.500, 7.571]
Superior: [0.99, 0.081] Inferior: [0.500, 7.571] Actual: [0.745, 2.563]
Superior: [0.99, 0.081] Inferior: [0.745, 2.563] Actual: [0.867, 1.212]
Superior: [0.99, 0.081] Inferior: [0.867, 1.212] Actual: [0.928, 0.608]
Superior: [0.928, 0.608] Inferior: [0.867, 1.212] Actual: [0.897, 0.886]
Superior: [0.897, 0.886] Inferior: [0.867, 1.212] Actual: [0.882, 1.024]
Ratio óptimo: 0.882
```

Figura 5.4: Ejecución del script de bisección.

niendo una aproximación razonablemente buena, no obstante éste es configurable a través de la variable *MARGEN*; la variación de este parámetro impactará en el resultado, a menor límite más cerca estará la medición del óptimo, pero tardará más en realizar el análisis.

El ejecutable es *bis_nb.out*, que mide el tiempo de cálculo de las posiciones y las velocidades, para cada componente, calcula su ratio y lo devuelve; el código fuente es prácticamente el del anexo E, al que se le han añadido instrucciones de medición de tiempo.

Finalmente es posible observar un ejemplo de ejecución de este algoritmo para un millón de partículas en la figura 5.4.

Los pares $[a, b]$, donde a es el ratio y b la relación entre los tiempos (T_{xeon}/T_{phi}), comprenden los límites superior e inferior del intervalo de búsqueda, así como el valor actualmente analizado; en este caso el ratio óptimo de equilibrio de carga es $R = 0{,}882$ con una relación entre los tiempos de 1,024, donde $abs(1{,}024 - 1{,}0) = 0{,}024 < 0{,}05$.

5.3.4. Equilibrio de carga dinámico para filtrado de imagen con representación matricial dispersa

La implementación de este método se ha realizado sobre un matriz 20000x20000, con un cociente de dispersión de 0,9, lo cual quiere decir que un 10 % de los elementos son 0, en concreto $(1{,}0 - 0{,}9) * 20000 * 20000 = 4 * 10^7$ elementos.

Se han desarrollado dos versiones, una nativa para ejecutar sobre un único componente, y otra de descarga; en concreto se va a explicar esta última, ya que es la más compleja e introduce algunos conceptos nuevos; en cualquier caso, se puede observar todo el código en el anexo F.

El algoritmo principalmente se divide en las siguientes fases:

- **Generación de datos**: las funciones *init_datos* e *init_datos_v2* crean la matriz 20kx20k con distintas probabilidades de que el elemento sea 0, en el primer caso es totalmente aleatoria mediante llamadas a *get_random*, en el segundo a través del operador módulo sobre los índices de las iteraciones, lo que genera un patrón predecible y un cociente de dispersión de 0,9.
- **Transformación a CSR**: la función *transf_CSR* convierte la matriz en los tres vectores de CSR: *val*, *rptr* y *col*; como los tamaños de *val* y *col* son variables, se ha asignado memoria para su máximo teórico, que sería 20kx20k, es decir, que ningún elemento sea 0 (cociente de dispersión 1.0). Por cada elemento de la matriz: si no es 0 se inserta en *val* y se suma 1 al total de elementos; su valor de columna, el índice j del bucle de recorrido, se inserta en *col*; cada vez que varía el índice i, se actualiza *rptr[i]* al número total de elementos en ese momento, que es precisamente el índice del primer elemento de esa fila i.

- **Filtrado del vector *val* respecto al vector *filtro***: una vez obtenidos estos tres vectores, se define un tamaño de segmento *tam_seg*, que indica el número de filas que se analiza por cada paquete de procesado; se tendrá un puntero *global* que señala el siguiente segmento a procesar, así como dos hilos que acceden a este puntero, uno de ellos se encargará del procesado local de segmentos y otro de la descarga al Phi; para cada segmento se llama a la función *filtrado*, que realiza las operaciones pertinentes y modifica los valores de *val* dentro del segmento.

La fase a evaluar es el filtrado, ya que es el núcleo de cómputo, por tanto va a profundizarse un poco más sobre su funcionamiento; los dos hilos tienen que sincronizarse para acceder al próximo segmento a procesar, la variable puntero *global* es compartida, cada vez que se adjudiquen un segmento, tienen que leer esta variable y sumarle el tamaño de segmento para el próximo acceso.

El problema surge cuando existen dos accesos al mismo tiempo: se tiene que el valor de *global* en un instante t_0 es P; si ambos hilos leen al mismo tiempo en t_0, ambos procesarán el segmento señalado por P, lo cual conduce a una redundancia indeseada; es más, cuando escriban posteriormente $P = P + tam\_seg$ es posible que se sumen 2 tamaños de segmento (uno por cada hilo) y el señalado por $P + tam\_seg$ nunca se llegue a procesar; es necesario pues que la operación de lectura y escritura se haga atómicamente, es decir, en una sola **sección crítica**.

OpenMP permite hacer esto mediante la directiva *#pragma omp critical* tal como se ve en el fragmento 5.5; esta sección se ejecuta una vez antes de comenzar a filtrar segmentos y después de cada segmento procesado, siendo ejecutada por un único hilo al mismo tiempo; si ambos llegan a la vez, uno entrará a la sección y el otro esperará hasta que el que está dentro salga para poder acceder.

La variable *sig* es una copia local del puntero compartido, de tal forma que se pueda utilizar durante el procesado sin tener que acceder a la variable *global* y generar más situaciones críticas; en general, estas secciones suponen un sobrecoste considerable cuando se accede con bastante frecuencia, por eso los tamaños de segmento pequeños tienden a sufrir por ello.

```
#pragma omp critical
{
    sig = global;
    global += tam_seg;
}
```

Listing 5.5: Sección crítica.

Una vez tengan el puntero al segmento a procesar, cada hilo llamará a *filtrado*; en el caso del Xeon Phi, se necesitan transferir los datos, como ya se vio en la descarga del N-body; la propiedad que se destacaba de la transferencia para esta aplicación era que el tamaño fijo del segmento permitía reutilizar la memoria asignada en el acelerador.

En concreto, se definen tres sub-vectores *d_val*, *d_rptr* y *d_col* tal y como se ve en el fragmento 5.6; la opción *__declspec(target(mic))* indica que la variable a continuación es definida únicamente en el acelerador, a diferencia de los vectores de la simulación N-body, que se definían en ambos entornos.

```
__declspec( target( mic ) ) double *d_val;
__declspec( target( mic ) ) size_t *d_rptr;
__declspec( target( mic ) ) size_t *d_col;
```

Listing 5.6: Vectores definidos en el Xeon Phi.

A estos vectores no se les asigna memoria dinámica con *malloc*, sino que se manejan desde el propio entorno de descarga que ofrecen las directivas de Intel LEO; el fragmento 5.7 refleja

la creación de memoria de estos tres vectores: el tamaño de *d_val* y *d_col* será de $tam\_seg * N$, que es el máximo posible con todos los elementos de las filas en el segmento no nulos, mientras que el de *d_rptr* será tam_seg+1, siendo ese "+1" necesario para el funcionamiento del algoritmo.

La opción *targetptr* permite inicializar la memoria únicamente en el Phi, lo cual presenta la sinergía buscada con *__declspec(target(mic))*; hay que destacar la utilización de *[0 : tamaño]* en sustitución de *length(tamaño)* para definir la longitud de la estructura, son equivalentes.

```
max_elem_transf = tam_seg * N;
#pragma offload_transfer target( mic : 0 )
nocopy( d_val[0 : max_elem_transf] : alloc_if( 1 ) free_if( 0 ) targetptr )
#pragma offload_transfer target( mic : 0 )
nocopy( d_col[0 : max_elem_transf] : alloc_if( 1 ) free_if( 0 ) targetptr )
#pragma offload_transfer target( mic : 0 )
nocopy( d_rptr[0 : tam_seg + 1] : alloc_if( 1 ) free_if( 0 ) targetptr )
```

Listing 5.7: Creación de los vectores d_x en el acelerador.

Ahora que la memoria en el Phi está lista, es necesario realizar las transferencias; una descarga tradicional como las realizadas en la simulación N-body supone que el vector existe tanto en el Xeon como en el Xeon Phi (por ejemplo *vx* existía en ambos); en este caso, sin embargo, en el Xeon se aloja el vector *val* completo, con un tamaño de MxN, mientras que en el Phi existe un sub-vector *d_val* de tamaño tam_segxN, lo que se repite para *col* y *rptr*.

Por ello, es necesario adaptar las transferencias, de tal forma que en la entrada se pasen partes de los vectores del Xeon a los sub-vectores del Phi, y en las salidas los sub-vectores completos a sus posiciones en los vectores grandes; es posible ver estas transferencias en el fragmento 5.8.

```
// Entradas.
elems_transf = rptr[sig + tam_seg] - rptr[sig];
#pragma offload_transfer target( mic : 0 )
in( val[rptr[sig] : elems_transf] :
into( d_val[0 : elems_transf] ) alloc_if( 0 ) free_if( 0 ) targetptr )
#pragma offload_transfer target( mic : 0 )
in( col[rptr[sig] : elems_transf] :
into( d_col[0 : elems_transf] ) alloc_if( 0 ) free_if( 0 ) targetptr )
#pragma offload_transfer target( mic : 0 )
in( rptr[sig : tam_seg + 1] :
into( d_rptr[0 : tam_seg + 1] ) alloc_if( 0 ) free_if( 0 ) targetptr )

// Salidas.
#pragma offload_transfer target( mic : 0 )
out( d_val[0 : elems_transf] :
into( val[rptr[sig] : elems_transf] ) alloc_if( 0 ) free_if( 0 ) targetptr )
```

Listing 5.8: Transferencias adaptadas.

La notación de longitud $[o : E]$ para las transferencias significa que se envían/reciben E elementos, desde el elemento o; por ejemplo, $in(data[4 : 2])$ significa que se envían 2 elementos desde $data[4]$, es decir, $data[4]$ y $data[5]$.

Por otro lado, la opción *into* permite especificar la estructura en el otro componente en la que se recibirán los datos; para este caso en concreto, por cada segmento se envían los elementos en *val* desde el primero de la fila inicial (*rptr[sig]*) hasta el primero de la fila inicial del siguiente segmento (*rptr[sig + tam_seg]*) no incluido; la longitud entonces es la expresada en *elems_transf*; se almacenan en el sub-vector *d_val* del Xeon Phi con la misma longitud pero lógicamente sin desplazamiento; el procedimiento para *rptr* y *col* es similar.

En el caso de la salida, se transfiere todo el subvector *d_val* ya filtrado en el acelerador a la misma posición que la especificada en la entrada, de tal forma que los elementos procesados sustituyan los originales; es muy importante destacar que siempre que se manejen estructuras únicamente definidas en el entorno del Phi hay que especificar la opción *targetptr*.

Estas transferencias reutilizan totalmente los vectores *d_x*, por tanto se está ahorrando la fase de asignación y liberación de memoria en cada transferencia, reduciendo considerablemente su sobrecoste; la liberación únicamente ocurre al final del procesado.

Cada vez que un hilo termina de procesar un segmento, pasa de nuevo por la sección crítica y actualiza el puntero *global*; cuando quede un único segmento por procesar, el *segmento resto*, los hilos abandonan el bucle y éste se procesa aparte; esto es debido a que el último segmento es un caso especial que obligaba a introducir un extra en la ejecución de los demás segmentos, por tanto se ha aislado.

5.4. Análisis de los resultados

En esta última sección se hará un breve análisis de los resultados obtenidos, cuyo objetivo no es tanto avalar los métodos desarrollados si no ofrecer al lector una carta de presentación al potencial del Xeon Phi basada en resultados experimentales.

Se recuerda que los análisis se han llevado a cabo sobre los núcleos de cómputo de los programas, excluyendo inicialización de memoria, entrada-salida y demás acciones complementarias en el host que no representan los objetivos para los que está diseñado el acelerador; el número de repeticiones por cada dato medido ha sido de mínimo 10, aumentando este valor para resultados inestables, de tal forma que las medias sean altamente representativas.

5.4.1. Simulación N-body optimizada

Se han realizado una serie de mediciones sobre la simulación N-body con 100000 partículas; se ha comparado el rendimiento del algoritmo secuencial, paralelo y optimizado tanto en el Xeon como en el Xeon Phi; los resultados son observables en la tabla 5.2, y la relación entre ellos en la figura 5.5.

Cuadro 5.2: Resultados en segundos para la simulación N-body con 100000 partículas.

	Seq.(1)	Par.(2)	Speedup 1 a 2	Opt. (par. + vect.)(3)	Speedup 2 a 3
Xeon	180,6948	16,4416	x10,990	7,926	x2,074
Xeon Phi	753,4312	5,405	x139,395	0,889	x6,08

En el caso del algoritmo secuencial se observa como el Xeon rinde significativamente mejor que el Xeon Phi, aproximadamente 4,15 veces más rápido; como se comentaba al comienzo de la investigación, a nivel de núcleo de proceso individual, el multicore tiene más capacidad, ya que posee, entre otras, mayor frecuencia de reloj (2 GHz respecto a 1.2 GHz del Phi), una estructura de caché más sofisticada (3 niveles respecto a los 2 del acelerador, con 15 MB en L3) y ejecución fuera de orden.

El algoritmo paralelo ofrece una mejora sustancial en ambos casos; para el Xeon, el speedup obtenido respecto al secuencial es de x11, mientras que para el Phi es de aproximadamente x140; los speedups teóricos se corresponden con el número de núcleos en cada componente, 12 en el Xeon y 61 en el Phi, no obstante la utilización de SMT con hilos hardware permite sobrepasarlos.

En el caso del acelerador, es capaz de ejecutar 2 hilos por núcleo simultáneamente en orden, así como utilizar los 2 restantes (4 en total por núcleo) cuando estos se bloquean (fallos de caché, predicciones de saltos erróneas, etc.); en el caso del Xeon no se llega a sobrepasar, ya que la ejecución fuera de orden permite un tiempo secuencial suficientemente bueno, con lo que el impacto del speedup es menor.

Finalmente la versión optimizada, en la que se ha vectorizado el código como principal factor de mejora, brinda uno resultados muy favorables al acelerador; el Xeon consigue un speedup de x23 respecto a su ejecución secuencial y de x2 respecto a la paralela; el Phi, por su parte, consigue una mejora de x848 respecto a la secuencial y de x6 respecto a la paralela, bajando del segundo en el tiempo total consumido.

En esta versión final, el acelerador es aproximadamente 9 veces más rápido que el host Xeon, con una mejora absoluta de 7 segundos; en el caso de la vectorización, el speedup obtenido depende fundamentalmente de la anchura de registro vectorial, que en el caso del Phi son 8 números en coma flotante de doble precisión (IMCI 512 bits, x8 speedup teórico respecto a la paralela), y en el del Xeon son 4 (AVX 256 bits, x4 speedup teórico respecto a la paralela).

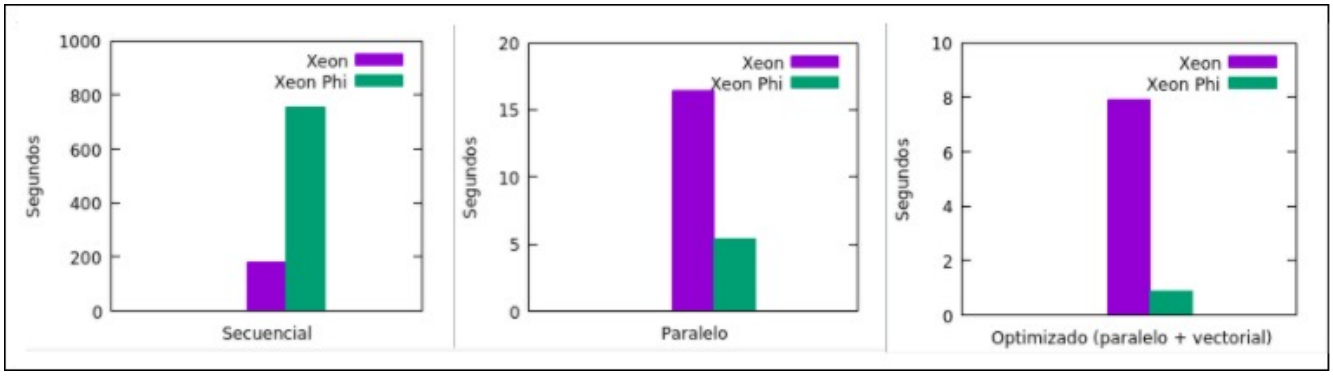

Figura 5.5: Relación entre los dos componentes para las tres iteraciones del algoritmo.

En general se puede observar que, si la aplicación es suficientemente paralelizable y vectorizable (como es el caso de la simulación N-body), el Xeon Phi ofrece un rendimiento muy superior al procesador Xeon, principalmente por su cantidad de núcleos de proceso y su mayor anchura de registro vectorial; por otra parte, si el programa presenta un número considerable de secciones secuenciales, es bastante posible que el acelerador sea más lento debido a las limitaciones individuales de sus núcleos.

5.4.2. Pre-análisis estático en la simulación N-body: interpolación y bisección

En el desarrollo de las aplicaciones se obtuvieron dos ratios para cada uno de los métodos; en el caso de la interpolación, el ratio fue de 0,875669, mientras que para la bisección (más exacto) fue de 0,882; como se puede observar, la diferencia entre ambos es minúscula; si se considera el resultado de la bisección como el ideal, podría decirse que el modelo de ecuaciones del método de interpolación es altamente representativo del caso real, con la ventaja de ser más rápido de obtener.

Para un millón de partículas, se han llevado a cabo tres mediciones: el caso nativo en el Phi, el colaborativo con el ratio de interpolación y el colaborativo con el ratio de bisección; los resultados se pueden visualizar en la figura 5.6; destacar que la ejecución nativa en el Phi, a pesar de representar un ratio de 1,0, no asume el sobrecoste de descargar los datos, el entorno

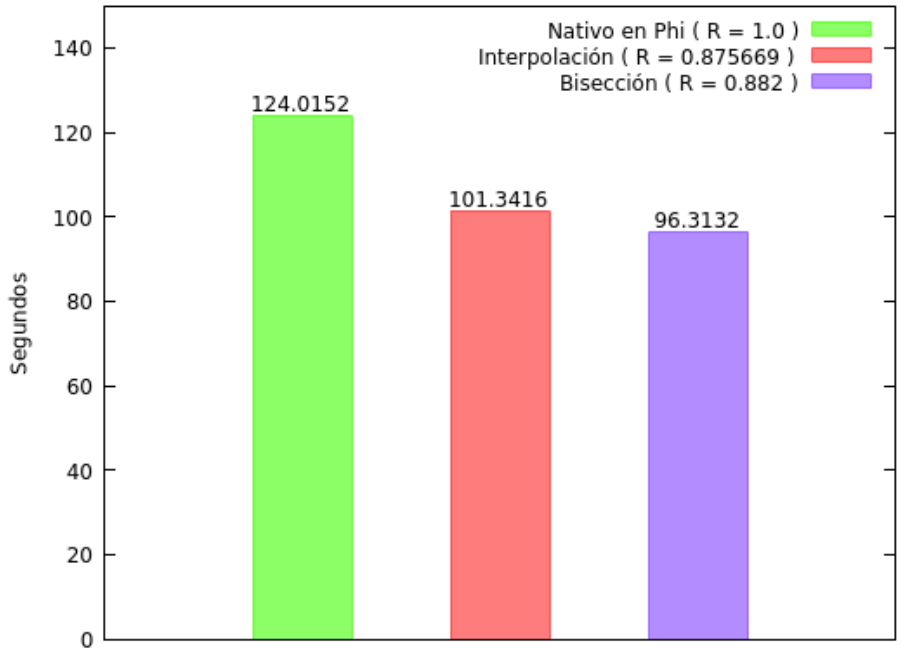

Figura 5.6: Tiempo empleado en realizar la simulación para los distintos ratios.

principal está en el acelerador desde el comienzo.

Se observa que existen diferencias considerables entre las tres ejecuciones; empleando el ratio obtenido por el método de interpolación (0,875669) se logra un speedup respecto a la ejecución nativa de 1,22, ahorrando en valor absoluto aproximadamente 22,7 segundos.

Por otro lado el ratio ideal del método de bisección (0,882) brinda un speedup respecto al nativo de 1,29 y una reducción absoluta de aproximadamente 27,7 segundos; se había señalado el hecho de que la diferencia entre los ratios obtenidos por ambos métodos era ínfima, no obstante es capaz de generar una diferencia de 5 segundos en el rendimiento, lo que sugiere la importancia en la exactitud del valor de este parámetro.

En resumen, se puede afirmar la efectividad del modo de descarga u *offload* para los ratios de división de trabajo adecuados; es importante comprender que debe existir al menos una de las siguientes dos condiciones para poder aplicar estos métodos: a) la diferencia de rendimiento entre el Xeon y el Xeon Phi no es alta o b) existe suficiente carga para repartir.

El modo de descarga implica una serie de sobrecostes, transferencia y sincronización fundamentalmente, si el Phi es mucho más rápido que el Xeon para un programa concreto, los ratios de división serán cercanos a 0,9 - 1,0, lo que se convierte prácticamente en una ejecución nativa con estos costes extra.

En estos casos, si la carga es suficientemente intensa, los métodos seguirán siendo efectivos, como puede verse en los resultados de N-body; obsérvese que en un algoritmo de complejidad $O(n^2)$, por cada unidad de n se agregan n operaciones más a la carga, así si $n = 1000$, para un ratio de 0,9 se ahorran $0{,}1 * 1000 * 1000 = 100000$ operaciones, mientras que para $n = 10000$ se evitan $0{,}1 * 10000 * 10000 = 10000000$ operaciones, valor con mucho más impacto en el rendimiento final.

5.4.3. Equilibrio de carga dinámico para el filtrado de imágenes con valores dispersos

Uno de los aspectos clave a la hora de obtener los resultados de este método es el tamaño del segmento; como se indicaba en el apartado 4.4.2, una longitud grande generará un desfase de tiempos de ejecución mayor, y por tanto más probabilidad de que los componentes terminen en momentos distintos, consecuencia de un desequilibrio de carga; una longitud pequeña, por su parte, tendrá un sobrecoste más alto por sincronización y comunicación, pero minimizará ese desfase y generará un mejor equilibrio de tiempos totales.

El segmento es un conjunto de filas en esencia, al trabajar con matrices de 20000x20000, éste ha de ser un número entre 1 y 20000, donde estos son los casos paradigmáticos de las situaciones expresadas en el anterior párrafo.

Se han realizado pruebas con los siguientes tamaños: 32, 64, 128, 256, 350, 512, 1024 y 2048; los resultados obtenidos para cada longitud de segmento se pueden observar en la tabla 5.3 y en la figura 5.7.

Cuadro 5.3: Resultados de rendimiento para el filtrado de imagen con distintas longitudes de segmento.

Long. segmento (filas)	Nativo	32	64	128	256	350	512	1024	2048
Tiempo total (seg)	44,344	49,423	44,081	42,825	42,871	41,538	42,84	48,808	64,153

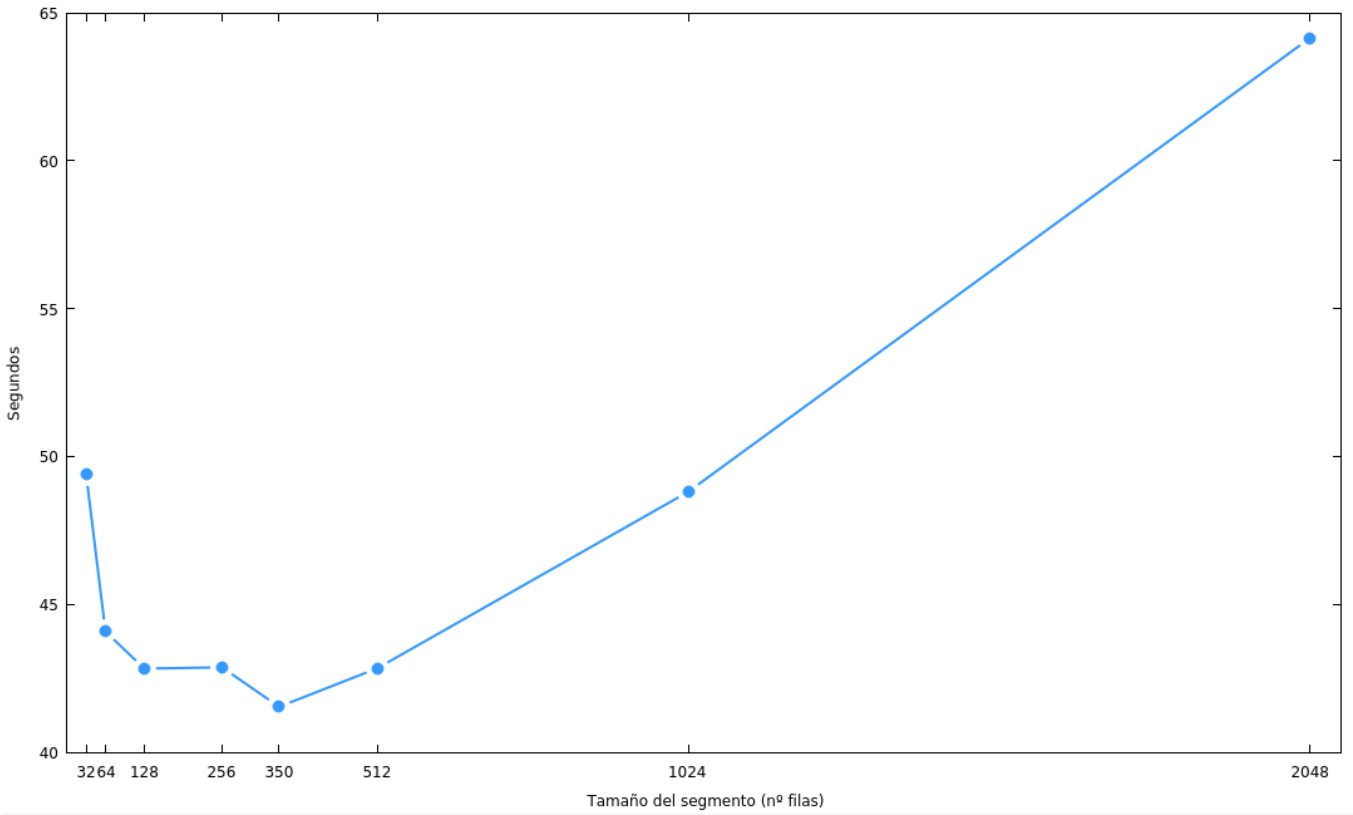

Figura 5.7: Impacto del tamaño de segmento en el algoritmo dinámico.

Se puede apreciar cómo se confirma la base teórica; los pequeños tamaños como 32 o 64 implican más transferencias y más accesos a secciones críticas, lo que, a pesar de que el desfase

de finalización es mínimo, aumenta el tiempo total de ambos componentes, generando un rendimiento subóptimo.

Las grandes longitudes, como 1024 o 2048 reducen el número de segmentos (19 en el primer caso, 9 en el segundo) y por tanto el sobrecoste del control, pero la ejecución de cada uno de los segmentos conlleva un tiempo empleado muy distinto en cada componente, lo que aumenta drásticamente el desfase de finalización; en concreto, el hecho de que se alcancen valores de 50 - 65 segundos de ejecución se debe a que el Xeon obtiene uno de los últimos segmentos a procesar, y el Phi ha de esperar a que termine para concluir el programa.

Son los valores intermedios [128 - 512] los que generan mejores resultados; en este caso una longitud de 350 filas por segmento es la que mejor ha funcionado, con un tiempo total empleado de 41,538, 2,8 segundos menos que la ejecución nativa en el Phi (44,344); si bien es cierto que es un speedup relativamente pequeño (1,07) comparado con los resultados de los otros métodos, hay que entender que este algoritmo es más complejo y por tanto tiene un coste adicional mayor.

Por otro lado se está trabajando con una carga ligera, en torno a los 45 segundos en el Phi; el verdadero valor de los algoritmos de equilibrio de carga se muestra efectivamente en programas muy pesados, donde las ejecuciones se alargan horas, incluso días, y donde un speedup de 1,07 se traduce a un ahorro significativo.

También es importante destacar que a menor cociente de dispersión, más entropía existirá en el coste de los segmentos, y por tanto más irregularidad, lo que beneficiará a este algoritmo.

En general la longitud de segmento óptima dependerá de la aplicación: si el rendimiento del Xeon Phi es mucho mayor sobre ella que el del Xeon, se intentará reducir este parámetro para minimizar el desfase de finalización, que como se ha visto en el análisis impacta negativamente de forma más contundente que el sobrecoste de control.

Hay que tener en cuenta también que un tamaño muy reducido disminuirá la efectividad de las optimizaciones en tiempo de ejecución (prefetching, vectorización, predicción de saltos) ya que se modifica el conjunto de datos (segmento) cada vez que se sustituye.

Finalmente, si se quiere obtener la longitud óptima de forma metódica se puede aplicar bisección u otros procesos similares.

Capítulo 6

Conclusiones y trabajos futuros

En este último capítulo se expondrán las principales conclusiones del proceso de investigación, incluyendo la consecución de los objetivos, el aspecto personal, dificultades, facultades enriquecidas y valoración del trabajo.

Además se presentarán una serie de ideas para incrementar el valor de la investigación, así como trabajos derivados de ésta a realizar en el futuro.

6.1. Conclusiones

Deduciendo de los resultados obtenidos en el apartado 5.4, el Xeon Phi es un componente hardware que puede conseguir mejoras de rendimiento significativas en aplicaciones con alto grado de paralelización y vectorización; además, el modo descarga permite combinar el poder computacional de éste con el procesador tradicional, reduciendo aún más los tiempos de ejecución.

Si bien es cierto que Intel ha hecho un magnífico trabajo reduciendo la complejidad a la hora de programar aceleradores, un requisito fundamental para poder aprovecharse de ello es comprender en detalle las aplicaciones utilizadas; si se quiere sacar el máximo rendimiento de este componente no se necesitan aprender nuevos paradigmas de programación, pero sí entender la estructura de los programas, los focos de paralelizabilidad y vectorizabilidad, sus complejidades temporales, cuáles son los sobrecostes y efectos de utilizar el acelerador para su ejecución, ...

En esta investigación se han ofrecido pasos concretos para facilitar este requisito y que los usuarios del Xeon Phi comprendan y utilicen el acelerador en las situaciones indicadas.

Se ha conseguido el objetivo de optimizar una aplicación para el Xeon Phi, y que además sirve de base para adaptar otros programas; para ello se ha modificado el algoritmo de la simulación N-body, añadiéndole las siguientes mejoras:

- Se han paralelizado los núcleos de cómputo, sobre todo centrándose en los bucles de procesado externos, a través de directivas OpenMP; con esto, se ha conseguido ejecutar el programa en los 61 núcleos del acelerador, alcanzando un speedup de x140 respecto a la ejecución secuencial.

- Se han vectorizado los bucles de procesado internos, de tal forma que por cada operación en coma flotante de doble precisión se realizan 8 simultáneamente; para ello se ha utilizado también OpenMP con las directivas *#pragma omp simd* y la opción *aligned*; esto ha reportado speedups de hasta x6 respecto a la versión paralela.

Una de las principales ventajas a la hora de programar este sistema es que las optimizaciones realizadas beneficiarán tanto al procesador como al acelerador, lo cual reduce severamente el coste de desarrollo.

Por otro lado se ha conseguido el objetivo de desarrollar mecanismos de equilibrio de carga, aprovechando los diferentes componentes de un sistema heterogéneo y mejorando el rendimiento de la aplicación optimizada.

En concreto se han desarrollado tres métodos, dos estáticos para aplicaciones de carga regular y uno dinámico para cargas irregulares; el fundamento de estos desarrollos ha sido el modo de descarga de datos al acelerador a través de las directivas *offload*, que ha permitido la co-ejecución de los programas sobre el sistema.

En el caso de los estáticos, se ha aproximado un ratio de repartición de datos óptimo a través de interpolación y el método de bisección, señalando las situaciones para las que es adecuado cada uno de ellos.

En el caso del dinámico, la definición de la longitud de segmento ha resultado ser trascendental para obtener el rendimiento deseado; se ha observado que es dependiente de la aplicación y que sus principales efectos son la variación en el desfase de finalización de los componentes y en el sobrecoste de control inherente al algoritmo.

Los métodos estáticos han logrado speedups de hasta x1,29 respecto a la ejecución nativa en el Phi, y los dinámicos de hasta x1,07.

6.2. Valoración personal

En el contexto personal y profesional, pienso que este proyecto me ha aportado bastante valor; primero he aprendido la importancia en la programación para alto rendimiento de comprender en detalle el entorno de actuación; desde la capacidad de cómputo de una unidad de proceso, pasando por la jerarquía de memoria, interconexiones y aspectos arquitecturales distintivos, hasta el diseño, implementación, optimización y compilación de las aplicaciones, interpretando su carga computacional y cómo dividirla.

Por otro lado creo que me ha ayudado en facetas más abstractas como la resiliencia, planificación y toma de decisiones; a lo largo del proceso he tenido dificultades y estancamientos, algunos de varios días incluso semanas, y en este sentido he tenido que tomar una serie de decisiones para pivotar el camino a seguir; ha habido que descartar posibles trabajos, desechar desarrollos que no aportaban valor a la investigación y volver a pasar por los mismos lugares, pero entiendo que esto es parte del proceso, así que aceptarlo y continuar, aunque sea difícil, es la única vía para el éxito.

Personalmente estoy contento con el trabajo realizado, creo que sirve para su propósito y espero que el lector haya obtenido al menos una visión global de cómo utilizar el Xeon Phi y sacarle el máximo partido.

6.3. Trabajos futuros

Sin extender mucho este apartado, se van a enumerar y detallar levemente una serie de mejoras y trabajos futuros para esta investigación.

- **Equilibrio de carga para paralelismo de tareas**: en el apartado 4.1 se observaba la existencia de otro tipo de equilibrio de carga, el de tareas; se pueden plantear una serie de

métodos para asignar las tareas de más intensidad computacional al Xeon Phi mientras que el Xeon realiza las relacionadas con recursos del sistema, como entrada/salida; sumado a esto, se podrían comparar estos métodos a los desarrollados sobre paralelismo de datos.

- **Mecanismos de convergencia alternativos al de bisección**: para la obtención del ratio óptimo en cargas regulares se ha utilizado el método de bisección; éste es de los más simples y, aunque efectivo, existen alternativas que pueden converger más rápido, como el de Newton-Raphson o el de Brent; se podrían implementar estos métodos y comparar el tiempo de análisis con el de bisección.

- **Utilización de OpenMP para la descarga**: a partir de OpenMP 4.0, se introducen una serie de directivas *#pragma omp target* sustitutivas de *#pragma offload* en Intel LEO; sería interesante traducir las implementaciones de descarga a las directivas de OpenMP, ya que éste es un estándar comprendido por múltiples compiladores, y si bien para la utilización del Xeon Phi se recomienda ICC, en un futuro otros como GCC podrían ofrecer un buen desempeño.

- **Integración en una librería**: para facilitar la utilización por parte del usuario, sería deseable que todos estos métodos y algoritmos estuviesen integrados en una librería genérica; a partir de unos parámetros de la aplicación, como el tamaño del problema, la curva computacional o el número de transferencias, se podría automatizar la aplicación de los métodos.

- **Análisis de eficiencia energética**: esta investigación ha llevado a cabo pruebas sobre el rendimiento de las aplicaciones en el Xeon Phi; otro aspecto importante y a la orden del día es el consumo energético en los sistemas; en este sentido, se podría comprobar la eficiencia energética del acelerador respecto a otros componentes como las GPUs o el propio procesador.

- **Equilibrio de carga guiado**: se han desarrollado dos mecanismos de equilibrio de carga, uno estático y otro dinámico; especialmente en este último, se ha observado que el momento crítico donde más impacto tiene el equilibrio es en los últimos segmentos, debido al ya mencionado desfase de finalización. Teniendo esto en cuenta, podría valorarse implementar un algoritmo adaptativo, que al comienzo del programa genere segmentos de gran tamaño, aprovechando los beneficios de estos, pero que cuando llegue a dicho momento crítico reduzca el tamaño de los segmentos para minimizar el desfase final; esto se suele denominar equilibrio de carga guiado.

Glosario

API Application Programming Interface, interfaz de programación de aplicaciones, es un conjunto de subrutinas y procedimientos que abstraen el funcionamiento de una biblioteca software para el programador que la utiliza. 6

Benchmark Programa o conjunto de programas que se ejecutan sobre un sistema con el objetivo de medir el rendimiento de éste en un área concreta, en el caso de las supercomputadoras se suele asociar a las operaciones en coma flotante por segundo. 4

CPU Central Processing Unit, unidad de procesamiento central, es la parte de una computadora encargada de procesar las instrucciones de los programas que se ejecutan sobre ella, así como realizar tareas aritméticas, lógicas, gestión de entradas/salidas, etc. 5

Extensiones vectoriales Son conjuntos de instrucciones (ISAs) que se añaden al de una arquitectura específica para dar soporte al procesado vectorial (SIMD); ejemplos de estos son SSE, AVX, AVX-512 o IMCI. 6

GPU Graphics Processing Unit, unidad de procesamiento de gráficos, es un hardware de un sistema especializado en procesamiento de imágenes y gráficos por computador, aunque es ampliamente utilizado para obtener rendimiento en aplicaciones altamente paralelizables debido a su estructura interna. 5

Hilo de ejecución También llamado subproceso, es la menor secuencia de instrucciones que se puede planificar en un computador por el sistema operativo; normalmente posee un proceso padre del que obtiene y con el que comparte recursos. 5

HPC High Performance Computing, computación de alto rendimiento, se refiere a la agregación del poder computacional en sistemas de gran escala y potencia para la ejecución de tareas de alto coste o carga de cómputo. 4

ISA Instruction Set Architecture, conjunto de instrucciones, es un modelo de instrucciones pensado para una arquitectura hardware concreta, aunque puede ser implementado de distintas maneras. 6

Memoria GDDR5 Graphics Double Data Rate type 5, es un tipo de memoria utilizada específicamente para componentes gráficos, como lo son los aceleradores, y posee un ancho de banda más alto que las memorias tradicionales. 6

MIC Many Integrated Core, diseño por parte de Intel de una arquitectura manycore, pensada para ofrecer alto grado de computación paralela y vectorial. 5

MPI Message Passing Interface, interfaz de paso de mensajes, es un estándar de programación por paso de mensajes pensado para ejecutar en diversas arquitecturas de computación paralelas, haciendo foco especialmente en sistemas distribuidos. 4

OoO Out-of-Order, hace referencia a la capacidad de un núcleo de proceso de ejecutar instrucciones en un orden distinto al natural; la ejecución se lleva a cabo respecto a la disponibilidad de los datos de entrada para la instrucción, evitando así las latencias de los bloqueos en orden. 4

OpenCL Open Computing Language, es un framework o entorno de trabajo destinado a la integración de los distintos tipos de hardware en sistemas heterogéneos. 4

PCI Express Peripheral Component Interconnect Express, interconexión de componentes periféricos de alta velocidad, es un bus serial para la conexión de componentes hardware adicionales en un sistema. La versión 2 alcanza hasta 500 MB/s por línea. 5

QPI QuickPath Interconnect, es una tecnología de conexión punto a punto desarrollada por Intel que permite velocidades de comunicación extremadamente altas entre procesadores en una misma placa, de tal forma que el sobrecoste de acceso a los recursos de un procesador por parte del otro sea ínfimo, pudiéndose considerarse un sistema global de un único procesador combinado. 32

SIMD Single Instruction Multiple Data, es una técnica para explotar el paralelismo a nivel de datos; las arquitecturas que soportan SIMD son capaces de procesar varios datos por instrucción, reduciendo significativamente el tiempo total empleado. 6

SMT Simultaneous MultiThreading, ejecución multi-hilo simultánea, es la capacidad de un núcleo de proceso de albergar más de un hilo concurrentemente, permitiendo en caso de bloqueos que otros hilos utilicen sus cauces de ejecución. 5

SSH Secure SHell, intérprete de órdenes seguro, es un protocolo de comunicación que, entre otras cosas, nos permite acceder de forma remota a otros sistemas para poder operar sobre ellos. 5

Bibliografía

[1] Aspen Systems Inc. *Ejemplos de aplicaciones de HPC para Intel Xeon Phi*. Accedido el 18 de julio de 2017. URL: https://www.aspsys.com/solutions/hpc-applications/phi-applications/.

[2] Oath Inc. (Engadget). *Tianhe-2 supercomputer claims the lead in Top 500 list, thanks its 3.1 million processor cores*. 2013. Accedido el 20 de julio de 2017. URL: https://www.engadget.com/2013/06/17/tianhe-2-supercomputer-claims-the-lead-in-top-500-list-thanks-i/.

[3] Daniel J Sorin, Mark D Hill y David A Wood. «A primer on memory consistency and cache coherence». En: *Synthesis Lectures on Computer Architecture* 6.3 (2011), págs. 1-212.

[4] Jim Handy. *The cache memory book*. Morgan Kaufmann, 1998, págs. 89-94.

[5] Edmond Chow (Georgia Tech College of Computing). *Arquitectura del Knights Corner*. URL: https://www.cc.gatech.edu/~echow/research.html.

[6] Ravishekhar Banger y Koushik Bhattacharyya. *OpenCL programming by example*. Packt Publishing Ltd, 2013.

[7] OpenMP. *Web oficial de OpenMP*. URL: http://www.openmp.org/.

[8] Alina Kiessling. «An introduction to parallel programming with OpenMP». En: *The University of Ediburgh* (2009).

[9] James Reinders. *Knights Corner: Open source software stack*. 2012. Accedido el 2 de agosto de 2017. URL: https://software.intel.com/en-us/blogs/2012/06/05/knights-corner-open-source-software-stack/.

[10] Intel Corp. *Escalabilidad del Xeon Phi respecto del Xeon*. URL: https://software.intel.com/en-us/articles/is-the-intel-xeon-phi-coprocessor-right-for-me.

[11] David F Bacon, Susan L Graham y Oliver J Sharp. «Compiler transformations for high-performance computing». En: *ACM Computing Surveys (CSUR)* 26.4 (1994), págs. 345-420.

[12] Tatu Ylonen Timo Rinne. *Scp Linux man page*. 2013. Accedido el 3 de agosto de 2017. URL: https://linux.die.net/man/1/scp.

[13] Intel Corporation. *Balanced Affinity Type*. 2015. Accedido el 2 de agosto de 2017. URL: https://software.intel.com/en-us/node/522518.

[14] Xinmin Tian y col. «Practical simd vectorization techniques for intel® xeon phi coprocessors». En: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, págs. 1149-1158.

[15] Tor E Jeremiassen y Susan J Eggers. *Reducing false sharing on shared memory multiprocessors through compile time data transformations*. Vol. 30. 8. ACM, 1995.

[16] Sparsh Mittal. «A survey of recent prefetching techniques for processor caches». En: *ACM Computing Surveys (CSUR)* 49.2 (2016), pág. 35.

[17] Kathryn S McKinley, Steve Carr y Chau-Wen Tseng. «Improving data locality with loop transformations». En: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.4 (1996), págs. 424-453.

[18] Timothy A Davis y Yifan Hu. «The University of Florida sparse matrix collection». En: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), págs. 5-6.

[19] Nathan Bell y Michael Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Inf. téc. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[20] Mo Mu y John R Rice. «The structure of parallel sparse matrix algorithms for solving partial differential equations on hypercubes». En: (1990).

[21] Peter Gács y László Lovász. «Complexity of algorithms». En: (1990).

[22] xszaboj (Code Project user). *Complejidad temporal algorítmica*. URL: https://www.codeproject.com/Articles/1012294/Algorithm-Time-Complexity-What-Is-It.

[23] Marsha J Berger y Joseph Oliger. «Adaptive mesh refinement for hyperbolic partial differential equations». En: *Journal of computational Physics* 53.3 (1984), págs. 484-512.

[24] Miloš Milovanović y col. «Transactional memory and OpenMP». En: *International Workshop on OpenMP*. Springer. 2007, págs. 37-53.

[25] Matt Gillespie. «Amdahl's Law, Gustafson's Trend, and the Performance Limits of Parallel Applications». En: *Online: http://software. intel. com/sites/default/files/m/d/4/1/d/8/Gillespie-0053-AAD_ Gustafson-Amdahl_ v1_ _2_. rh. final. pdf* (2008).

[26] Intel Corporation. *Procesador Intel Xeon E5-2620*. Accedido el 11 de agosto de 2017. URL: https://ark.intel.com/es-es/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI.

[27] Intel Corporation. *Coprocesador Intel Xeon Phi 7120P*. Accedido el 11 de agosto de 2017. URL: https://ark.intel.com/es-es/products/75799/Intel-Xeon-Phi-Coprocessor-7120P-16GB-1_238-GHz-61-core.

[28] Sverre J Aarseth y Sverre Johannes Aarseth. *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press, 2003.

[29] Rien van de Weijgaert et al. *Void Hierarchy: Evolution Experiments*. Accedido el 15 de agosto de 2017. URL: https://www.astro.rug.nl/~weygaert/voidhierarchy.II.html.

[30] D. Böhme. *Characterizing Load and Communication Imbalance in Parallel Applications:* Schriften des Forschungszentrums Jülich IAS Series. Hochschulbibliothek der Rheinisch-Westfälischen Technischen Hochschule Aachen, 2014. Cap. 3. ISBN: 9783893369409. URL: https://books.google.es/books?id=q70xAwAAQBAJ.

[31] Chris J Newburn y col. «Offload compiler runtime for the Intel® Xeon Phi coprocessor». En: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, págs. 1213-1225.

[32] James Reinders y James Jeffers. *High Performance Parallelism Pearls Volume One: Multicore and Many-core Programming Approaches*. Morgan Kaufmann, 2014.

[33] Jim Jeffers y James Reinders. *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*. Morgan Kaufmann, 2015.

Anexos

Anexo A

Código base de la simulación N-body (una iteración)

```
/* Iteracion de la simulacion N-body; actualiza las posiciones y las velocidades de
    todas las particulas */
void newton( const size_t particulas, const double dt, double *x, double *y, double
    *z, double *vx, double *vy, double *vz, const double *m )
{
        // Constantes.
        const double G = 6.67384e-11;
        const double dtG = dt * G;

        // Variables.
        double dvx, dvy, dvz, dx, dy, dz, dist2, mjOverDist3;
        size_t i, j;

        // Para cada particula i actualiza posicion y velocidad.
        for( i = 0U; i < particulas; i++ )
        {
            // Variaciones de la velocidad.
            dvx = 0.0;
            dvy = 0.0;
            dvz = 0.0;

            // Particulas 0 a i - 1.
            for( j = 0U; j < i; j++ )
            {
                // Variaciones de distancia entre j e i.
                dx = x[j] - x[i];
                dy = y[j] - y[i];
                dz = z[j] - z[i];

                // Distancia al cuadrado.
                dist2 = dx * dx + dy * dy + dz * dz;

                // Masa de j sobre distancia al cubo.
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );

                // Actualiza las variaciones de la velocidad de i por efecto de j.
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }

            // Particulas i + 1 a N.
            for( j = i + 1U; j < particulas; j++ )
            {
```

```
                dx = x[j] - x[i];
                dy = y[j] - y[i];
                dz = z[j] - z[i];
                dist2 = dx * dx + dy * dy + dz * dz;
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }

            // Actualiza las velocidades para la siguiente iteracion.
            vx[i] += dvx * dtG;
            vy[i] += dvy * dtG;
            vz[i] += dvz * dtG;
        }

        for( i = 0U; i < particulas; i++ )
        {
            // Actualiza las posiciones para la siguiente iteracion.
            x[i] += vx[i] * dt;
            y[i] += vy[i] * dt;
            z[i] += vz[i] * dt;
        }
}
```

Este código ha sido obtenido y adaptado del libro [32], en general es una lectura recomendable para aprender técnicas sobre programación paralela; actualmente existe una versión actualizada en [33].

Anexo B

Código paralelo de la simulación N-body (una iteración)

```
/* Iteracion de la simulacion N-body; actualiza las posiciones y las velocidades de
   todas las particulas */
void newton_par( const size_t particulas, const double dt, double *x, double *y,
   double *z, double *vx, double *vy, double *vz, const double *m )
{
    // Constantes.
    const double G = 6.67384e-11;
    const double dtG = dt * G;

    // Variables.
    double dvx, dvy, dvz, dx, dy, dz, dist2, mjOverDist3;
    size_t i, j;

    // Para cada particula i actualiza posicion y velocidad.
    #pragma omp parallel private( i, j, dvx, dvy, dvz, dx, dy, dz,
    dist2, mjOverDist3 )
    {
        #pragma omp for schedule( static )
        for( i = 0U; i < particulas; i++ )
        {
            // Variaciones de la velocidad.
            dvx = 0.0;
            dvy = 0.0;
            dvz = 0.0;

            // Particulas 0 a i - 1.
            for( j = 0U; j < i; j++ )
            {
                // Variaciones de distancia entre j e i.
                dx = x[j] - x[i];
                dy = y[j] - y[i];
                dz = z[j] - z[i];

                // Distancia al cuadrado.
                dist2 = dx * dx + dy * dy + dz * dz;

                // Masa de j sobre distancia al cubo.
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );

                // Actualiza las variaciones de la velocidad de i por efecto de j.
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }
```

```
            // Particulas i + 1 a N.
            for( j = i + 1U; j < particulas; j++ )
            {
                dx = x[j] - x[i];
                dy = y[j] - y[i];
                dz = z[j] - z[i];
                dist2 = dx * dx + dy * dy + dz * dz;
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }

            // Actualiza las velocidades para la siguiente iteracion.
            vx[i] += dvx * dtG;
            vy[i] += dvy * dtG;
            vz[i] += dvz * dtG;
        }

        #pragma omp for schedule( static )
        for( i = 0U; i < particulas; i++ )
        {
            // Actualiza las posiciones para la siguiente iteracion.
            x[i] += vx[i] * dt;
            y[i] += vy[i] * dt;
            z[i] += vz[i] * dt;
        }
    }
}
```

Anexo C

Código optimizado de la simulación N-body (una iteración) y programa principal

```
/* Iteracion de la simulacion N-body; actualiza las posiciones y las velocidades de
   todas las particulas */
void newton_opt( const size_t particulas, const double dt, double *x, double *y,
   double *z, double *vx, double *vy, double *vz, const double *m )
{
    // Constantes.
    const double G = 6.67384e-11;
    const double dtG = dt * G;

    // Variables.
    double dvx, dvy, dvz, dx, dy, dz, dist2, mjOverDist3;
    size_t i, j;

    // Para cada particula i actualiza posicion y velocidad.
    #pragma omp parallel private( i, j, dvx, dvy, dvz, dx, dy, dz,
    dist2, mjOverDist3 )
    {
        #pragma omp for schedule( static )
        for( i = 0U; i < particulas; i++ )
        {
            // Variaciones de la velocidad.
            dvx = 0.0;
            dvy = 0.0;
            dvz = 0.0;

            // Particulas 0 a i - 1.
            #pragma omp simd aligned( m : 64 )
            for( j = 0U; j < i; j++ )
            {
                // Variaciones de distancia entre j e i.
                dx = x[j] - x[i];
                dy = y[j] - y[i];
                dz = z[j] - z[i];

                // Distancia al cuadrado.
                dist2 = dx * dx + dy * dy + dz * dz;

                // Masa de j sobre distancia al cubo.
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );

                // Actualiza las variaciones de la velocidad de i por efecto de j.
```

```
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }

            // Particulas i + 1 a N.
            #pragma omp simd
            for( j = i + 1U; j < particulas; j++ )
            {
                dx = x[j] - x[i];
                dy = y[j] - y[i];
                dz = z[j] - z[i];
                dist2 = dx * dx + dy * dy + dz * dz;
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }

            // Actualiza las velocidades para la siguiente iteracion.
            vx[i] += dvx * dtG;
            vy[i] += dvy * dtG;
            vz[i] += dvz * dtG;
        }

        #pragma omp for simd schedule( static )
        for( i = 0U; i < particulas; i++ )
        {
            // Actualiza las posiciones para la siguiente iteracion.
            x[i] += vx[i] * dt;
            y[i] += vy[i] * dt;
            z[i] += vz[i] * dt;
        }
    }
}
```

```
/* Procedimiento principal para la simulacion N-body */
int main( int argc, char **argv )
{
    // Constantes.
    const double dt = 0.01;
    const size_t particulas = 300000U;

    // Variables.
    double *x, *y, *z, *vx, *vy, *vz, *m, tiempo;
    FILE *outf;
    size_t i;
    struct timeval ini_t, fin_t;

    // Asigna memoria.
    x = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    y = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    z = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    vx = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    vy = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    vz = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );
    m = ( double * ) aligned_alloc( 64U, particulas * sizeof( double ) );

    // Especifica las posiciones, velocidades y masas iniciales.
    srand( time( NULL ) );
    #pragma omp parallel for schedule( static )
    for( i = 0U; i < particulas; i++ )
    {
```

```
        x[i] = ( double ) ( rand( ) % 50 );
        y[i] = ( double ) ( rand( ) % 50 );
        z[i] = ( double ) ( rand( ) % 50 );
        vx[i] = ( double ) ( rand( ) % 10 );
        vy[i] = ( double ) ( rand( ) % 10 );
        vz[i] = ( double ) ( rand( ) % 10 );
        m[i] = ( double ) ( rand( ) % 1000 );
    }

    // Computa.
    gettimeofday( &ini_t, NULL );
    newton_opt( particulas, dt, x, y, z, vx, vy, vz, m );
    gettimeofday( &fin_t, NULL );
    tiempo = ( double ) ( fin_t.tv_sec - ini_t.tv_sec ) + ( double ) ( fin_t.tv_usec
        - ini_t.tv_usec ) / 1000000.0;
    outf = fopen( "nbopt.txt", "a" );
    fprintf( outf, "%.3f\n", tiempo );
    fclose( outf );

    // Libera memoria.
    free( x );
    free( y );
    free( z );
    free( vx );
    free( vy );
    free( vz );
    free( m );

    // Sale.
    return 0;
}
```

Anexo D

Reporte de compilación

```
LOOP BEGIN at n_body_utils.c(70,9)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at n_body_utils.c(75,13)
   <Peeled loop for vectorization>
      remark #15389: vectorization support: reference x[j] has unaligned access   [ n_body_utils.c(77,22) ]
      remark #15389: vectorization support: reference y[j] has unaligned access   [ n_body_utils.c(78,22) ]
      remark #15389: vectorization support: reference z[j] has unaligned access   [ n_body_utils.c(79,22) ]
      remark #15389: vectorization support: reference m[j] has unaligned access   [ n_body_utils.c(81,31) ]
      remark #15381: vectorization support: unaligned access used inside loop body
      remark #15305: vectorization support: vector length 8
      remark #15309: vectorization support: normalized vectorization overhead 0.630
      remark #15301: PEEL LOOP WAS VECTORIZED
   LOOP END

   LOOP BEGIN at n_body_utils.c(75,13)
      remark #15389: vectorization support: reference x[j] has unaligned access   [ n_body_utils.c(77,22) ]
      remark #15389: vectorization support: reference y[j] has unaligned access   [ n_body_utils.c(78,22) ]
      remark #15389: vectorization support: reference z[j] has unaligned access   [ n_body_utils.c(79,22) ]
      remark #15388: vectorization support: reference m[j] has aligned access   [ n_body_utils.c(81,31) ]
      remark #15381: vectorization support: unaligned access used inside loop body
      remark #15305: vectorization support: vector length 8
      remark #15309: vectorization support: normalized vectorization overhead 0.776
      remark #15300: LOOP WAS VECTORIZED
      remark #15450: unmasked unaligned unit stride loads: 3
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 114
      remark #15477: vector cost: 16.750
      remark #15478: estimated potential speedup: 6.240
      remark #15488: --- end vector cost summary ---
   LOOP END
```

Se puede observar la vectorización del bucle interno de la simulación N-body, y cómo el acceso a *m[j]* es el único alineado; la anchura de registro o *vector length* es 8, 64 bytes entre 8 bytes de un *double*; el *speedup* estimado es de 6,240.

El *peeled loop* es una optimización interna del compilador que extirpa un número K de iteraciones del total N para que $N - K$ sea divisor de la anchura de registro.

Anexo E

Programa de descarga para la simulación N-body y división de las funciones de ésta.

Prototipos de las funciones para la simulación.

```
// Librerias.
#include <stddef.h>

// Prototipos.
#pragma offload_attribute( push, target( mic ) )
void newton_vel( const size_t particulas, const size_t lim_inf, const size_t lim_sup
    , const double dt, const double *x, const double *y, const double *z, double *vx
    , double *vy, double *vz, const double *m );
void newton_pos( const size_t lim_inf, const size_t lim_sup, const double dt, double
     *x, double *y, double *z, const double *vx, const double *vy, const double *vz
    );
#pragma offload_attribute( pop )
```

Funciones para la simulación.

```
// Librerias.
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include "n_body_utils.h"

// Calcula las velocidades.
void newton_vel( const size_t particulas, const size_t lim_inf, const size_t lim_sup
    , const double dt, const double *x, const double *y, const double *z, double *vx
    , double *vy, double *vz, const double *m )
{
    // Constantes.
    const double G = 6.67384e-11;
    const double dtG = dt * G;

    // Variables.
    double dvx, dvy, dvz, dx, dy, dz, dist2, mjOverDist3;
    size_t i, j;

    // Computa.
    #pragma omp parallel private( i, j, dvx, dvy, dvz, dx, dy, dz,
    dist2, mjOverDist3 )
    {
        #pragma omp for schedule( static )
        for( i = lim_inf; i < lim_sup; i++ )
        {
```

```
            dvx = 0.0;
            dvy = 0.0;
            dvz = 0.0;
            #pragma omp simd aligned( m : 64 )
            for( j = 0U; j < i; j++ )
            {
                dx = x[j] - x[i];
                dy = y[j] - y[i];
                dz = z[j] - z[i];
                dist2 = dx * dx + dy * dy + dz * dz;
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }
            #pragma omp simd
            for( j = i + 1; j < particulas; j++ )
            {
                dx = x[j] - x[i];
                dy = y[j] - y[i];
                dz = z[j] - z[i];
                dist2 = dx * dx + dy * dy + dz * dz;
                mjOverDist3 = m[j] / ( dist2 * sqrt( dist2 ) );
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }
            vx[i] += dvx * dtG;
            vy[i] += dvy * dtG;
            vz[i] += dvz * dtG;
        }
    }
}

// Calcula las posiciones.
void newton_pos( const size_t lim_inf, const size_t lim_sup, const double dt, double
     *x, double *y, double *z, const double *vx, const double *vy, const double *vz
    )
{
    // Variables.
    size_t i;

    // Computa.
    #pragma omp parallel for simd private( i ) schedule( static )
    for( i = lim_inf; i < lim_sup; i++ )
    {
        x[i] += vx[i] * dt;
        y[i] += vy[i] * dt;
        z[i] += vz[i] * dt;
    }
}
```

Parte de cómputo del programa principal de descarga.

```
// Computa.
omp_set_nested( 1 );
phi_datos = ( size_t ) ( ratio * ( double ) particulas );
#pragma omp parallel num_threads( 2 )
{
    // Velocidades Xeon.
    if( omp_get_thread_num( ) == 0 )
    {
        newton_vel( particulas, phi_datos, particulas, dt, x, y, z, vx, vy, vz, m );
    }
```

```
    // Velocidades Phi.
    else
    {
        // Inicializacion del entorno.
        #pragma offload_transfer target( mic : 0 )
        // Asignacion de memoria.
        #pragma offload_transfer target( mic : 0 )
        nocopy( x, y, z, m : length( particulas ) alloc_if( 1 ) free_if( 0 ) )
        nocopy( vx, vy, vz : length( phi_datos ) alloc_if( 1 ) free_if( 0 ) )
        // Entrada.
        #pragma offload_transfer target( mic : 0 )
        in( x, y, z, m : length( particulas ) alloc_if( 0 ) free_if( 0 ) )
        in( vx, vy, vz : length( phi_datos ) alloc_if( 0 ) free_if( 0 ) )
        // Ejecucion.
        #pragma offload target( mic : 0 )
        in( particulas, phi_datos, dt )
        nocopy( x, y, z, m : length( particulas ) alloc_if( 0 ) free_if( 0 ) )
        nocopy( vx, vy, vz : length( phi_datos ) alloc_if( 0 ) free_if( 0 ) )
        newton_vel( particulas, 0U, phi_datos, dt, x, y, z, vx, vy, vz, m );
    }

    // Sincronizacion.
    #pragma omp barrier

    // Posiciones Xeon.
    if( omp_get_thread_num( ) == 0 )
    {
        newton_pos( phi_datos, particulas, dt, x, y, z, vx, vy, vz );
    }
    // Posiciones Phi.
    else
    {
        // Ejecucion (persistencia de las transferencias hechas anteriormente).
        #pragma offload target( mic : 0 )
        in( particulas, phi_datos, dt )
        nocopy( x, y, z : length( particulas ) alloc_if( 0 ) free_if( 0 ) )
        nocopy( vx, vy, vz : length( phi_datos ) alloc_if( 0 ) free_if( 0 ) )
        newton_pos( 0U, phi_datos, dt, x, y, z, vx, vy, vz );
        // Salida.
        #pragma offload_transfer target( mic : 0 )
        out( x, y, z : length( phi_datos ) alloc_if( 0 ) free_if( 0 ) )
        out( vx, vy, vz : length( phi_datos ) alloc_if( 0 ) free_if( 0 ) )
        // Liberacion de memoria.
        #pragma offload_transfer target( mic : 0 )
        nocopy( x, y, z, m : length( particulas ) alloc_if( 0 ) free_if( 1 ) )
        nocopy( vx, vy, vz : length( phi_datos ) alloc_if( 0 ) free_if( 1 ) )
    }
}
```

Existen dos hilos, uno se encarga de descargar al acelerador y el otro de la ejecución local en el procesador; *omp_set_nested(1)* es necesario para que se puedan anidar llamadas a la directiva *#pragma omp parallel*, de tal forma que cada hilo pueda generar más "hermanos" para la ejecución paralela.

Anexo F

Implementación nativa y de descarga para el filtrado de matrices dispersas.

Cabeceras de las funciones comunes (sparse_utils.h).

```
// Librerias.
#include <stddef.h>

// Prototipos.
double gen_random( );
void init_datos( double *mat_img, double *filtro, const size_t M, const size_t N );
void init_datos_v2( double *mat_img, double *filtro, const size_t M, const size_t N
    );
size_t transf_CSR( const double *mat_img, double *val, size_t *rptr, size_t *col,
    const size_t M, const size_t N );
#pragma offload_attribute( push, target( mic ) )
void filtrado( double *val, const size_t *rptr, const size_t *col, const double *
    filtro, const size_t tam_seg, const size_t N );
#pragma offload_attribute( pop )
```

Implementación de las funciones comunes (sparse_utils.c); la función *init_datos_v2* no utiliza la llamada a *rand( )*, ya que ésta no es vectorizable por el compilador, y por tanto es más rápida a la hora de generar la matriz.

```
// Librerias.
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "sparse_utils.h"

// Genera un numero aleatorio entre el 0.0 y el 1.0, 0.0 con probabilidad zero_p.
double gen_random( )
{
    // Variables.
    double num, zero_p;

    num = ( double ) ( rand( ) % 10 + 1 ) / 10.0;
    zero_p = ( double ) ( rand( ) % 10 + 1 ) / 10.0;
    if( ( double ) rand( ) < zero_p * ( ( double ) RAND_MAX + 1.0 ) )
    {
        return 0.0;
    }
    else
    {
        return num;
    }
```

```
}

// Inicializa la matriz de la imagen y el filtro.
void init_datos( double *mat_img, double *filtro, const size_t M, const size_t N )
{
    // Variables.
    int tid;
    size_t i, j;

    #pragma omp parallel private( i, j, tid )
    {
        tid = omp_get_thread_num( );
        srand( ( unsigned int ) time( NULL ) ^ ( unsigned int ) tid );
        #pragma omp for schedule( static )
        for( i = 0U; i < N; i++ )
        {
            mat_img[0U + i] = gen_random( );
            filtro[i] = ( double ) ( abs( N / 2 - i ) ) / ( double ) ( N / 2 );
        }
        #pragma omp for schedule( static )
        for( i = 1U; i < M; i++ )
        {
            for( j = 0U; j < N; j++ )
            {
                mat_img[i * N + j] = gen_random( );
            }
        }
    }
}

// Inicializa los datos sin llamar a random.
void init_datos_v2( double *mat_img, double *filtro, const size_t M, const size_t N
    )
{
    // Variables.
    size_t i, j;

    #pragma omp parallel private( i, j )
    {
        #pragma omp for schedule( static )
        for( i = 0U; i < N; i++ )
        {
            mat_img[0U + i] = ( double ) ( ( i ) % 11U ) / 10.0;
            filtro[i] = ( double ) ( abs( N / 2 - i ) ) / ( double ) ( N / 2 );
        }
        #pragma omp for schedule( static )
        for( i = 1U; i < M; i++ )
        {
            for( j = 0U; j < N; j++ )
            {
                mat_img[i * N + j] = ( double ) ( ( i + j ) % 11U ) / 10.0;
            }
        }
    }
}

// Transforma la matriz de la imagen a formato CSR y devuelve el numero de elementos
    .
size_t transf_CSR( const double *mat_img, double *val, size_t *rptr, size_t *col,
    const size_t M, const size_t N )
{
    // Variables.
    double elem;
```

```
    size_t i, j, elem_count;

    elem_count = 0U;
    for( i = 0U; i < M; i++ )
    {
        rptr[i] = elem_count;
        for( j = 0U; j < N; j++ )
        {
            elem = mat_img[i * N + j];
            if( elem >= 0.1 )
            {
                val[elem_count] = elem;
                col[elem_count] = j;
                elem_count++;
            }
        }
    }

    return elem_count;
}

// Filtra la imagen en formato CSR en base al filtro proporcionado.
void filtrado( double *val, const size_t *rptr, const size_t *col, const double *
    filtro, const size_t tam_seg, const size_t N )
{
    // Variables.
    double res;
    size_t i, j;

    #pragma omp parallel private( i, j, res )
    {
        #pragma omp for schedule( static )
        for( i = 0U; i < rptr[tam_seg] - rptr[0U]; i++ )
        {
            res = 0.0;
            #pragma omp simd aligned( filtro : 64 )
            for( j = 0U; j < col[i]; j++ )
            {
                res += val[i] * filtro[j];
            }
            #pragma omp simd
            for( j = col[i] + 1U; j < N; j++ )
            {
                res -= val[i] * filtro[j];
            }
            val[i] = fabs( fmod( res, 1.0 ) );
        }
    }
}
```

Implementación del algoritmo nativo (mod_color_native.c).

```
// Librerias.
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "sparse_utils.h"

// Procedimiento principal.
int main( int argc, char **argv )
{
    // Constantes.
    const size_t M = 20000U, N = 20000U;
```

```
    // Variables.
    double *mat_img, *val, *filtro, tiempo, res;
    FILE *outf;
    size_t *rptr, *col, elementos, i, j;
    struct timeval ini_t, fin_t;

    // Asignacion de memoria.
    mat_img = ( double * ) aligned_alloc( 64U, M * N * sizeof( double ) );
    filtro = ( double * ) aligned_alloc( 64U, N * sizeof( double ) );
    val = ( double * ) aligned_alloc( 64U, M * N * sizeof( double ) );
    rptr = ( size_t * ) aligned_alloc( 64U, M * sizeof( size_t ) );
    col = ( size_t * ) aligned_alloc( 64U, M * N * sizeof( size_t ) );

    // Inicializacion.
    init_datos_v2( mat_img, filtro, M, N );

    // Transformacion a CSR.
    elementos = transf_CSR( mat_img, val, rptr, col, M, N );
    printf( "Sparsity: %.3f\n", ( double ) elementos / ( double ) ( M * N ) );

    // Realiza el filtrado.
    gettimeofday( &ini_t, NULL );
    filtrado( val, rptr, col, filtro, M - 1U, N );
    // Ultima fila.
    #pragma omp parallel private( i, j, res )
    {
        #pragma omp for schedule( static )
        for( i = rptr[M - 1U]; i < elementos; i++ )
        {
            res = 0.0;
            #pragma omp simd aligned( filtro : 64 )
            for( j = 0; j < col[i]; j++ )
            {
                res += val[i] * filtro[j];
            }
            #pragma omp simd
            for( j = col[j] + 1U; j < N; j++ )
            {
                res -= val[i] * filtro[j];
            }
            val[i] = fabs( fmod( res, 1.0 ) );
        }
    }
    gettimeofday( &fin_t, NULL );
    tiempo = ( double ) ( fin_t.tv_sec - ini_t.tv_sec ) + ( double ) ( fin_t.tv_usec
        - ini_t.tv_usec ) / 1000000.0;
    outf = fopen( "nat.txt", "a" );
    fprintf( outf, "%.3f\n", tiempo );
    fclose( outf );

    // Sale.
    return 0;
}
```

Implementación del algoritmo de descarga (mod_color.c).

```
// Librerias.
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "sparse_utils.h"
```

```
// Procedimiento principal.
int main( int argc, char **argv )
{
    // Constantes.
    const size_t M = 20000U, N = 20000U;

    // Variables.
    char outn[20];
    double *mat_img, *val, *filtro, tiempo, res;
    FILE *outf;
    size_t *rptr, *col, elementos, sig, global, max_elem_transf, elems_transf, i, j,
        tam_seg;
    struct timeval ini_t, fin_t;
    // Variables Phi.
    __declspec( target( mic ) ) double *d_val;
    __declspec( target( mic ) ) size_t *d_rptr;
    __declspec( target( mic ) ) size_t *d_col;

    // Asignacion de memoria.
    mat_img = ( double * ) aligned_alloc( 64U, M * N * sizeof( double ) );
    filtro = ( double * ) aligned_alloc( 64U, N * sizeof( double ) );
    val = ( double * ) aligned_alloc( 64U, M * N * sizeof( double ) );
    rptr = ( size_t * ) aligned_alloc( 64U, M * sizeof( size_t ) );
    col = ( size_t * ) aligned_alloc( 64U, M * N * sizeof( size_t ) );

    // Inicializacion.
    init_datos_v2( mat_img, filtro, M, N );

    // Transformacion a CSR.
    elementos = transf_CSR( mat_img, val, rptr, col, M, N );

    // Obtiene el tamano de segmento.
    tam_seg = ( size_t ) atoi( argv[1] );

    // Realiza el filtrado.
    gettimeofday( &ini_t, NULL );
    omp_set_nested( 1 );
    global = 0U;
    #pragma omp parallel num_threads( 2 ) private( sig )
    {
        // El hilo del Xeon Phi inicializa y prepara el entorno para la
            reutilizacion de memoria.
        if( omp_get_thread_num( ) == 1 )
        {
            #pragma offload_transfer target( mic : 0 )
            // El maximo de elementos que va a tener que procesar es tam_seg * N,
                rptr sera como maximo del tamano del segmento + 1.
            max_elem_transf = tam_seg * N;
            #pragma offload_transfer target( mic : 0 )
            nocopy( d_val[0 : max_elem_transf] : alloc_if( 1 ) free_if( 0 )
            targetptr )
            #pragma offload_transfer target( mic : 0 )
            nocopy( d_col[0 : max_elem_transf] : alloc_if( 1 ) free_if( 0 )
            targetptr )
            #pragma offload_transfer target( mic : 0 )
            nocopy( d_rptr[0 : tam_seg + 1] : alloc_if( 1 ) free_if( 0 ) targetptr )
            #pragma offload_transfer target( mic : 0 )
            nocopy( filtro : length( N ) alloc_if( 1 ) free_if( 0 ) )
            // Solo necesitamos transferir el filtro una vez, ya que es inmutable.
            #pragma offload_transfer target( mic : 0 )
            in( filtro : length( N ) alloc_if( 0 ) free_if( 0 ) )
        }
```

```
#pragma omp critical
{
    sig = global;
    global += tam_seg;
}
while( ( sig + tam_seg ) < M )
{
    // Xeon.
    if( omp_get_thread_num( ) == 0 )
    {
        filtrado( &val[rptr[sig]], &rptr[sig], &col[rptr[sig]], filtro,
            tam_seg, N );
    }
    // Xeon Phi.
    else
    {
        // Transfiere los subconjuntos de datos para operar.
        elems_transf = rptr[sig + tam_seg] - rptr[sig];
        #pragma offload_transfer target( mic : 0 )
        in( val[rptr[sig] : elems_transf] :
        into( d_val[0 : elems_transf] ) alloc_if( 0 ) free_if( 0 )
        targetptr )
        #pragma offload_transfer target( mic : 0 )
        in( col[rptr[sig] : elems_transf] :
        into( d_col[0 : elems_transf] ) alloc_if( 0 ) free_if( 0 )
        targetptr )
        #pragma offload_transfer target( mic : 0 )
        in( rptr[sig : tam_seg + 1] :
        into( d_rptr[0 : tam_seg + 1] ) alloc_if( 0 ) free_if( 0 )
        targetptr )
        // Realiza el filtrado.
        #pragma offload target( mic : 0 )
        in( tam_seg, N )
        nocopy( d_val[0 : elems_transf] : alloc_if( 0 ) free_if( 0 )
        targetptr )
        nocopy( d_col[0 : elems_transf] : alloc_if( 0 ) free_if( 0 )
        targetptr )
        nocopy( d_rptr[0 : tam_seg + 1] : alloc_if( 0 ) free_if( 0 )
        targetptr )
        nocopy( filtro : length( N ) alloc_if( 0 ) free_if( 0 ) )
        {
            filtrado( d_val, d_rptr, d_col, filtro, tam_seg, N );
        }
        // Devuelve los valores filtrados.
        #pragma offload_transfer target( mic : 0 )
        out( d_val[0 : elems_transf] : into( val[rptr[sig] : elems_transf] )
        alloc_if( 0 ) free_if( 0 ) targetptr )
    }

    #pragma omp critical
    {
        sig = global;
        global += tam_seg;
    }
}// Fin del bucle de procesado de segmentos.

// Procesa el ultimo segmento.
#pragma omp single
{
    if( M % tam_seg != 0 )
    {
        sig = M - M % tam_seg;
```

```
                filtrado( &val[rptr[sig]], &rptr[sig], &col[rptr[sig]], filtro, M -
                    sig - 1U, N );
            }
            // Ultima fila.
            #pragma omp parallel private( i, j, res )
            {
                #pragma omp for schedule( static )
                for( i = rptr[M - 1U]; i < elementos; i++ )
                {
                    res = 0.0;
                    #pragma omp simd aligned( filtro : 64 )
                    for( j = 0; j < col[i]; j++ )
                    {
                        res += val[i] * filtro[j];
                    }
                    #pragma omp simd
                    for( j = col[j] + 1U; j < N; j++ )
                    {
                        res -= val[i] * filtro[j];
                    }
                    val[i] = fabs( fmod( res, 1.0 ) );
                }
            }
        }

        // Libera la memoria utilizada para persistencia en el Phi.
        if( omp_get_thread_num( ) == 1 )
        {
            #pragma offload_transfer target( mic : 0 )
            nocopy( d_val[0 : max_elem_transf] : alloc_if( 0 ) free_if( 1 )
            targetptr )
            #pragma offload_transfer target( mic : 0 )
            nocopy( d_col[0 : max_elem_transf] : alloc_if( 0 ) free_if( 1 )
            targetptr )
            #pragma offload_transfer target( mic : 0 )
            nocopy( d_rptr[0 : tam_seg + 1] : alloc_if( 0 ) free_if( 1 ) targetptr )
            #pragma offload_transfer target( mic : 0 )
            nocopy( filtro : length( N ) alloc_if( 0 ) free_if( 1 ) )
        }
    }
    gettimeofday( &fin_t, NULL );
    tiempo = ( double ) ( fin_t.tv_sec - ini_t.tv_sec ) + ( double ) ( fin_t.tv_usec
        - ini_t.tv_usec ) / 1000000.0;
    sprintf( outn, "tam_%zu.txt", tam_seg );
    outf = fopen( outn, "a" );
    fprintf( outf, "%.3f\n", tiempo );
    fclose( outf );

    // Sale.
    return 0;
}
```