UNIVERSIDAD DE LA REPUBLICA
URUGUAY

FACULTAD DE INGENIERIA

Computación científica de alto desempeño en sistemas Xeon Phi

Proyecto de grado de la carrera de
Ingeniería en Computación

Autores:
Matías Carro
Pablo Da Costa Porto

Supervisores:
Sergio Nesmachnow
Santiago Iturriaga

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Abril 2023

RESUMEN

Intel Xeon Phi es una línea de coprocesadores cuya arquitectura consiste de entre 57 y 61 núcleos, a una frecuencia de alrededor de 1GHz, que implementan un superconjunto de las instrucciones x86, con la adición de instrucciones SIMD de 512 bits. Cada coprocesador viene contenido dentro de una tarjeta de expansión PCIe, incluyendo hasta 16GB de memoria RAM, y se integra a sistemas existentes corriendo su propio sistema operativo autónomo basado en Linux. Su arquitectura cuenta con compiladores oficiales para C/C++ y Fortran y con implementaciones de varios modelos de programación paralela comunes con otras arquitecturas de memoria compartida, incluyendo el estándar OpenMP y la biblioteca Intel Threading Building Blocks (TBB). Por sus características, es una arquitectura de alto desempeño que, en principio, no requiere conocimientos específicos para su uso y permite el desarrollo de aplicaciones portables. La propuesta de este trabajo es interiorizarse en la arquitectura de Xeon Phi y sus herramientas para el desarrollo de aplicaciones, con énfasis en TBB, buscando identificar las condiciones y prácticas necesarias para su desempeño óptimo, culminando con la implementación de una aplicación como prueba de concepto. La literatura consultada y las pruebas realizadas muestran que, para obtener un buen desempeño, es necesario que las aplicaciones sean escalables a entre 100 y 250 hilos concurrentes, que sus operaciones aprovechen el tamaño completo de las unidades vectoriales, y que saquen provecho de la localidad de memoria. Se conocen múltiples técnicas portables de optimización para favorecer la escalabilidad, la vectorización y el aprovechamiento de la localidad, pero se observa que, en la mayoría de los casos, es necesario usar construcciones específicas de la arquitectura para optimizar la vectorización. La prueba de concepto implementada consistió de una simulación de una red pesca. Con la ayuda del perfilador VTune Amplifier, fue posible llegar a una implementación escalable de la simulación usando TBB. Sin embargo, la implementación final no mostró un buen aprovechamiento de la vectorización y la localidad, quedando pendiente la optimización de estos aspectos.

Índice

1. Introducción

Este proyecto de grado propone estudiar la implementación de modelos de computación paralela en el coprocesador Intel Xeon Phi, una tecnología novedosa que se caracteriza por ejercer paralelismo a través de alrededor de 60 núcleos de procesamiento en un sistema de memoria compartida, soportando los lenguajes y herramientas de programación paralela tradicionales. Se presenta un estudio de la arquitectura del coprocesador y las herramientas con las que cuenta, buscando identificar los problemas a los que se adecúa y las prácticas para optimizar su rendimiento. Finalmente, se implementa un caso de estudio específico, el cual consiste en la simulación del movimiento de una red de pesca.

La computación científica plantea gran diversidad de problemas, los cuales pueden requerir gran cantidad de cálculo, al punto de que las tecnologías tradicionales no sean adecuadas para resolverlos en un tiempo razonable. Para atacar este tipo de problemas surgen las distintas tecnologías de computación de alto desempeño, cuyas arquitecturas son diseñadas específicamente para obtener una capacidad de cómputo acorde [1][6].

Cada tecnología implica un conjunto de herramientas asociadas para su utilización, por lo que cada una requiere aprendizaje específico para su uso, y la obtención de un rendimiento óptimo puede incluso necesitar de un conocimiento a fondo de su arquitectura, de manera de aplicar las prácticas necesarias para esta finalidad. Existiendo una diversidad de tecnologías, cada una con sus características, en simultáneo con la diversidad de problemas a los que potencialmente se aplican, surge también la cuestión de conocer en qué situaciones es útil cada tecnología en particular, es decir, a qué tipo de problemas se adecúa cada una.

El coprocesador Intel Xeon Phi es una tecnología de computación de alto desempeño que cuenta con una arquitectura innovadora, mientras mantiene las herramientas clásicas de los procesadores de memoria compartida tradicionales [28][29][30]. Este proyecto se plantea con el fin de estudiar el dispositivo, su arquitectura y modo de uso, y buscar responder las cuestiones mencionadas en cuanto a esta arquitectura, culminando con el desarrollo y estudio de una aplicación real.

Los problemas que se resuelven en computación científica frecuentemente trabajan sobre conjuntos de datos de gran tamaño y requieren tecnologías de alto desempeño para resolverse. Los clústers son sistemas de alto desempeño armados conectando múltiples computadoras a través de una red. Las computadoras que componen un clúster pueden tener una arquitectura tradicional, pero, para aprovechar la capacidad de cómputo completa del clúster en la resolución de un mismo problema, es necesario usar protocolos de comunicación para coordinar el trabajo de las computadoras, y usar dichos protocolos para el desarrollo de aplicaciones requiere conocimiento específico por parte de los desarrolladores [4][10]. Una tecnología de alto desempeño alternativa son las GPUs (unidades de procesamiento gráfico, por sus siglas en inglés). Las GPUs son unidades de procesamiento adicionales al procesador central de una computadora, originalmente dedicadas a la computación gráfica y el procesamiento de imágenes. Las capacidades de procesamiento paralelo que presentan hizo que empezaran a usarse para resolver problemas de propósito general. Sin embargo, su arquitectura hace que se dificulte obtener un buen desempeño en programas que no presentan un acceso regular a los datos. Además, usarlas requiere aplicar modelos de programación paralela específicos, en ocasiones propietarios [25][26][30].

Intel Xeon Phi fue lanzado al mercado en un contexto de tendencia al crecimiento en la cantidad de núcleos de procesamiento por chip que presentan las nuevas arquitecturas de computadoras de propósito general. Esta tendencia empieza en la segunda mitad de la década de los 2000, cuando se llegó a un límite en la frecuencia de los diseños de procesadores, dado por el aumento exponencial del consumo energético y la generación de calor respecto a la frecuencia. Este límite, junto a otras razones, llevó a que las arquitecturas subsiguientes usaran la estrategia de

multiplicar la cantidad de núcleos de procesamiento por chip para aprovechar las mejoras en las tecnologías de transistores en busca de mejor desempeño. Siguiendo esta tendencia, la arquitectura Many Integrated Core (MIC), implementada por Xeon Phi, es particularmente innovadora en que define unidades de procesamiento de alto desempeño integrando más de 50 de núcleos de procesamiento, muy similares a los de los procesadores x86, pero de bajo rendimiento. Sacrificando el desempeño secuencial, este diseño permite obtener una gran capacidad de cómputo paralelo, a la vez que mantiene una microarquitectura muy parecida a la tradicional [28][29][30].

La generación de Intel Xeon Phi denominada *Knights Corner* está compuesta por coprocesadores de entre 57 y 61 núcleos a entre 1.05GHz y 1.25GHz. Los núcleos son, esencialmente, núcleos x86 con instrucciones SIMD de 512 bits, lo cual le permite superar billón de operaciones de doble precisión por segundo. Siendo un coprocesador, se integra a la placa madre de otra computadora a través de un puerto PCI-e [28][29][30][31].

Los compiladores oficiales de Intel permiten usar los lenguajes C y Fortran en Xeon Phi, sin necesidad de hacer consideraciones particulares, por lo que los programas desarrollados para Xeon Phi son portables a arquitecturas x86. Soporta varios de los modelos de programación paralela populares para sistemas de memoria compartida, incluyendo OpenMP e Intel TBB. Como interfaz, tiene integrado un sistema operativo basado en Linux, al cual se puede acceder, desde el sistema en que está instalado el coprocesador, para ejecutar programas de forma nativa. También tiene, como alternativa, el modo de ejecución *offload*, en el cual un programa que corre en el sistema principal puede delegar partes del código al coprocesador [28][29][30].

Sus capacidades de cómputo vectorizado, el modo de ejecución offload y su formato de tarjeta de expansión hacen que Xeon Phi sea frecuentemente comparado con tecnologías de GPUs. Se afirma que el coprocesador es aplicable a todas las aplicaciones que se desempeñan bien en GPUs, a la vez que es flexible para otros tipos de aplicaciones. Se afirma también, como ventaja respecto a las GPUs, la falta de necesidad de aplicar modelos de programación específicos para la arquitectura en comparación con las demás arquitectuas de propósito general, lo cual facilita el uso y brinda portabilidad [30].

Entre los modelos de programación paralela compatibles con Xeon Phi, se destaca Intel TBB, una biblioteca de C++ que implementa varios patrones paralelos conocidos y es compatible también con las arquitecturas x86. TBB tiene la particularidad de soportar paralelismo anidado, lo que es un factor atractivo para su adopción debido a que permite la modularización arbitraria y, por lo tanto, facilita el desarrollo y fomenta la mantenibilidad del código. Es de particular interés para este trabajo experimentar con el desarrollo de aplicaciones escalables y mantenibles usando TBB en Xeon Phi, con el fin de evaluar la dificultad implicada.

El trabajo realizado consistió en interiorizarse con la arquitectura de Xeon Phi, su modo de uso y los modelos de programación paralela que soporta, buscando identificar sus capacidades, su aplicablidad, y las prácticas necesarias para obtener un buen desempeño, haciendo foco en el modelo de programación Intel TBB. La parte experimental del trabajo consistió en abordar un problema específico con Xeon Phi, partiendo de una versión secuencial y aplicando progresivamente TBB y optimizaciones para mejorar el uso de los recursos de la arquitectura. Se usó el perfilador Intel VTune Amplifier para guiar el proceso de optimización. El problema abordado consistió de una simulación del comportamiento de una red de pesca.

Este documento se divide en cuatro partes, siendo la presente introducción la parte 1. Las siguientes dos partes contienen la mayoría del contenido del documento y están divididas en capítulos. En la parte 2, se exponen los conceptos estudiados, introduciendo la computación científica de alto desempeño, las especificaciones de arquitectura y software de Xeon Phi, el análisis sobre cómo optimizar las aplicaciones para Xeon Phi, y los trabajos relacionados que fueron de utilidad. En la parte 3 se documenta el problema de la red de pesca y las pruebas realizadas a partir de él sobre el coprocesador. La parte 4 cierra el documento con las conclusiones

obtenidas a partir de los conocimientos expuestos y los resultados obtenidos.

2. Marco teórico

La primera parte del trabajo realizado consistió en estudiar la literatura existente relacionada a Xeon Phi, buscando obtener un abordaje profundo de sus características y su utilización. En los siguientes capítulos se documenta el conocimiento relevante adquirido durante ese proceso.

2.1. Computación científica de alto desempeño

Las actividades implicadas por este trabajo se enmarcan dentro de la computación científica de alto desempeño. En este capítulo se introduce dicha disciplina, incluyendo un relevamiento general de las tecnologías usadas para resolver los problemas que plantea y un relato histórico del desarrollo tecnológico.

2.1.1. Definición

La computación científica es el campo multidisciplinario que estudia la aplicación de capacidades de cómputo para la comprensión y resolución de problemas científicos y de ingeniería. Formalmente, se define como *"el estudio de la computación eficiente de métodos constructivos de la matemática aplicada"* [1]. Esto implica el desarrollo de modelos matemáticos de fenómenos del mundo real, el análisis numérico de estos modelos para la construcción de métodos de resolución y la implementación de estos métodos en sistemas de computación reales. Se incluye también el análisis de los métodos en cuanto a su eficiencia y precisión [1].

Los problemas sobre los que se trabaja pueden originarse sobre cualquier rama de la ciencia o la ingeniería. La resolución de estos problemas implica la generación de modelos matemáticos, usualmente sistemas de ecuaciones diferenciales, que luego son resueltos mediante métodos numéricos (esto es, métodos de aproximación basados en cálculos numéricos). Algunos ejemplos de posibles problemas y sus métodos de resolución pueden ser:

- en la mecánica de estructuras, se modela el comportamiento de las estructuras en función de las fuerzas a las que se someten; los resultados se calculan mediante el método de elementos finitos [2].
- SPICE es un simulador que genera ecuaciones diferenciales a partir de especificaciones de circuitos electrónicos; la simulación se realiza resolviendo las ecuaciones mediante el método de Newton [2].
- en finanzas, la ecuación de Black-Scholes determina el valor de compra o venta de un activo según las reglas del mercado que se esté estudiando; se puede resolver mediante el método de elementos finitos o mediante métodos Monte Carlo [2].
- los *problemas de N-cuerpos* consisten en estudiar el comportamiento de un conjunto de partículas; en cada instante de tiempo se identifican las fuerzas sobre cada partícula (como se hace sobre la partícula central de la figura 1) y se integra la ecuación resultante para determinar el desplazamiento de la partícula en el instante siguiente; la integración se puede realizar mediante varios métodos, entre ellos el método Störmer-Verlet y los métodos Runge-Kutta [1][3].

En computación, el *desempeño* es un atributo de calidad asociado a la velocidad de la ejecución de programas; se aplica a [4][5]:

- un sistema computacional, para referirse a la velocidad de la ejecución programas en general o programas con una aplicación específica, frecuentemente en comparación con la velocidad

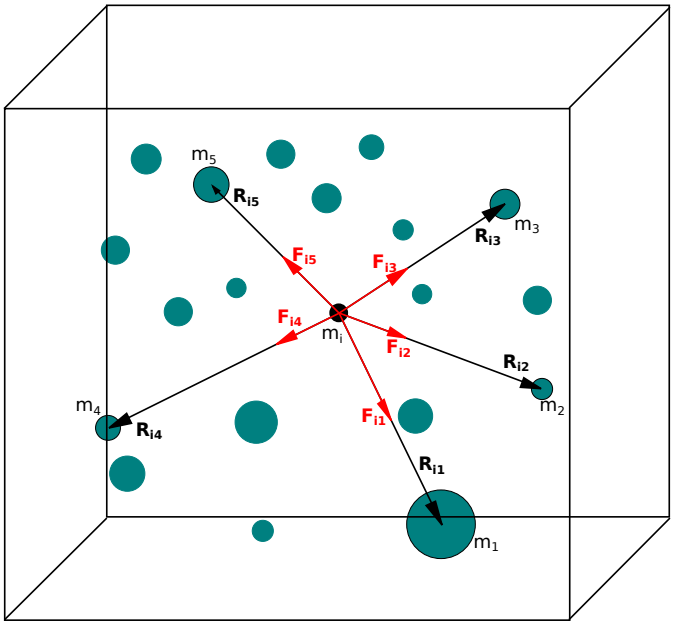


Figura 1: Cálculo de las fuerzas sobre una partícula realizado para resolver un problema de N-cuerpos. (Imagen extraída de Eijkhout et al. con licencia CC BY 3.0 [1].)

de los mismos programas en otros sistemas. Métricas asociadas son el *desempeño pico*, que se mide como la máxima capacidad computacional por unidad de tiempo que el sistema puede permitir, o los resultados de la ejecución de programas de prueba específicos para medir desempeño que tengan en cuenta otros aspectos como el rendimiento de la memoria y la comunicación, como lo son los *benchmarks* LINPACK o SPEC.

- la ejecución de un programa particular en un sistema dado, para referirse a su velocidad de ejecución o a la eficiencia del uso que hace de los recursos del sistema, frecuentemente en comparación con otras ejecuciones que resuelvan el mismo problema en el mismo sistema. Algunas métricas asociadas son el tiempo de ejecución, la tasa de operaciones por segundo del programa, la eficiencia en el uso de la memoria caché, y el *speedup*, que indica la mejora obtenida por el uso de paralelismo en proporción a la ejecución secuencial.

El término computación de alto desempeño (HPC, *high-perfomance computing*) tampoco es un término formalmente definido. Frecuentemente, está directamente asociado al análisis, diseño y uso de supercomputadoras, que son computadoras con velocidad de cómputo, memoria principal y espacio de almacenamiento mucho mayores en comparación a las computadoras de propósito general [6][7]. Sin embargo, el avance tecnológico experimentado por las tecnologías de propósito general lleva a que se pueda hablar de obtener alto desempeño en computadoras personales [8].

Una forma más general para referirse a HPC es a través de los problemas a resolver que abarca. Ejemplo puede ser la predicción numérica del clima, donde la atmósfera es dividida en celdas tridimensionales, para las cuales se debe calcular las distintas condiciones del tiempo atmosférico en cada instante de tiempo en función de las condiciones de la celda y las celdas contiguas en el instante anterior, requiriendo trabajar con regiones grandes (y con una cantidad de operaciones proporcional) para realizar predicciones de varios días. Otro ejemplo puede ser la simulación del comportamiento de un sistema astronómico, cuya complejidad computacional

está dada por la cantidad de cuerpos celestes que la constituyen [10]. Este tipo de problemas, muchos de ellos originados en la computación científica, son frecuentemente llamados problemas *grand challenge* (gran desafío), e implican una gran cantidad de cálculos sobre una gran cantidad de datos, por lo que requieren un poder de cómputo considerable para ser resueltos en un tiempo razonable [6].

En resumen, el estudio de la HPC implica el análisis y la elaboración de arquitecturas e infraestructuras de computación orientadas al cálculo intensivo y el desarrollo y la aplicación de algoritmos ejecutados sobre estas infraestructuras para la resolución de problemas de alta compejidad computacional. Su asociación con las supercomputadoras liga fuertemente a la HPC con la computación paralela, debido a que en la actualidad las arquitecturas de supercomputadoras se apoyan en este paradigma para la obtención de poder de cómputo [9]. Esto lleva a que el estudio de HPC incluya el estudio de tecnologías de computación paralela, y a que la mayor parte de la bibliografía actual de HPC se enfoque en este tema.

2.1.2. Desarrollo histórico

El desempeño general de los sistemas de computación se encuentra en crecimiento gracias al desarrollo sostenido en la tecnología que utilizan. Este desarrollo se puede representar mediante la observación realizada en 1975 por Gordon Moore, que aproximaba que la densidad de transistores de los circuitos integrados se ve duplicada cada dos años. Esta observación es conocida como ley de Moore y, como se puede observar en la figura 2, hasta la actualidad ha mostrado ser una aproximación correcta, llegando a funcionar como una guía de la industria para la planificación del desarrollo de nuevas tecnologías [4][5][30].

Las mejoras en desempeño, además de mejorar la ejecución de los programas existentes, per-

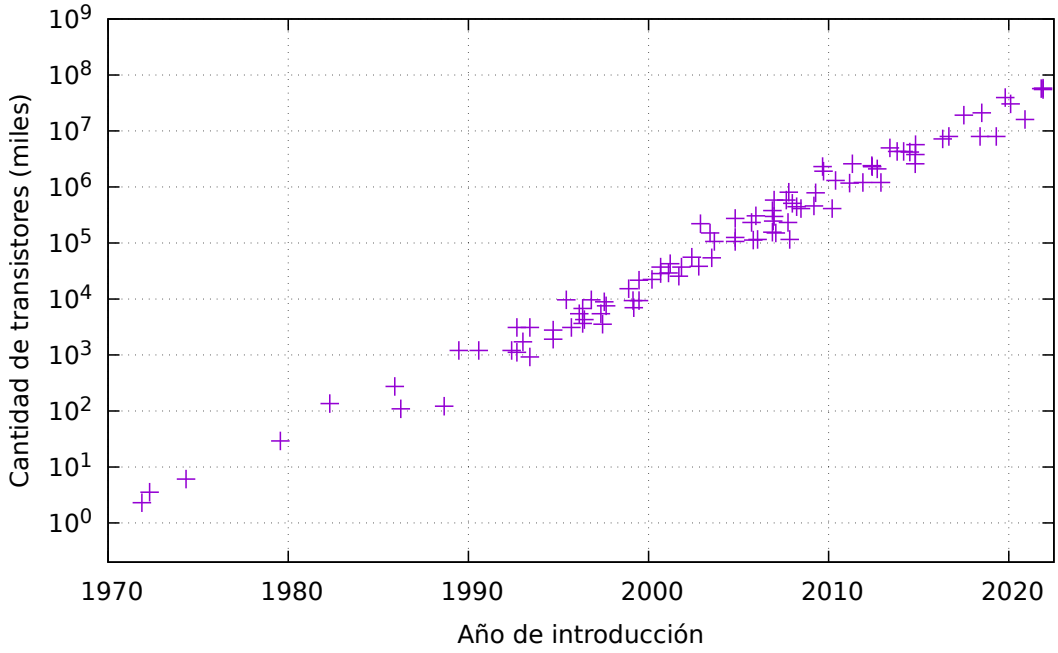


Figura 2: Cantidad de transistores para los modelos de procesadores a través del tiempo, en escala logarítmica. (Datos extraídos de Rupp [58].)

miten y traen consigo la aparición de nuevas aplicaciones o la aplicación de los mismos programas a nuevos dominios más complejos. De hecho, se observa que la demanda por capacidad computacional y de memoria de los programas emergentes crece más rápido que lo que la ley de Moore permite, sin límites a la vista dentro de ciertas áreas como la computación científica. Todo esto sugiere que el desarrollo tecnológico seguirá apuntando a aumentar el desempeño de los sistemas de computación en el futuro [4].

En un principio, las mejoras en la tecnología de transistores permitieron el aumento de la frecuencia de los relojes y el diseño de arquitecturas más complejas. Con el tiempo, la arquitectura Von Neumann inicial fue evolucionando para incorporar técnicas de paralelismo a nivel de instrucción (ILP, *instruction-level parallelism*), de manera de aumentar el número de instrucciones simultáneas que un procesador puede ejecutar, y así multiplicar la tasa de instrucciones ejecutadas por ciclo de reloj. Los procesadores pasaron a contar con múltiples unidades funcionales que pueden ejecutar distintas instrucciones independientes en paralelo, en lo que es llamado ejecución superescalar; además, las instrucciones son segmentadas en etapas discretas, las cuales son ejecutadas por distintos componentes del procesador, de forma que varias instrucciones que se encuentren en etapas distintas puedan ejecutarse simultáneamente, lo que es denominado *pipeline* de instrucciones. Las primeras supercomputadoras, surgidas en las décadas de 1950 y 1960, se basaban en este tipo de técnicas para obtener su excepcional desempeño para la época. Sin embargo, este tipo de técnicas solo generan mejoras de desempeño en factores constantes, es decir, su desarrollo no permite mejoras progresivas en el desempeño. Más allá de que estas técnicas se usen en las arquitecturas actuales, se considera que las mejoras que generan se estancaron y ya se ha llegado a sacar el máximo provecho posible del paralelismo a nivel de instrucción. Por esta razón, para los diseños posteriores se debió explorar nuevas estrategias [4][5][11].

Un cambio paradigmático se dio a mitad de la década de 1970 con la aparición de los procesadores vectoriales [11], cuyas arquitecturas se caracterizan por contar con instrucciones que trabajan sobre conjuntos de datos (vectores) en lugar de datos individuales, siguiendo el paradigma SIMD (*single instruction, multiple data*; única instrucción, múltiples datos). Una instrucción SIMD es utilizada para realizar una única operación sobre los datos de un vector, como puede ser la multiplicación de cada elemento por un escalar o la suma de cada elemento con su correspondiente en otro vector. Los compiladores de estas arquitecturas, con asistencia del programador, son encargados de detectar, en el programa, bucles de operaciones independientes u operaciones explícitamente escritas con notación de arreglo a ser compiladas con instrucciones SIMD [12]. De esta forma, el manejo de la ejecución paralela es completamente responsabilidad de la arquitectura. De hecho, las arquitecturas de los procesadores vectoriales no cuentan estrictamente con tantas unidades funcionales paralelas como el tamaño del vector, sino que basan su desempeño en pocas unidades funcionales con pipelines profundos [13]. Dado este manejo de instrucciones SIMD, los procesadores vectoriales muestran particularmente buen desempeño en programas que presentan paralelismo a nivel de datos, es decir, que trabajan sobre grandes cantidades de datos independientes que se transforman idénticamente, como lo hacen muchos algoritmos de aplicaciones científicas. La computación científica estuvo dominada por este tipo de arquitecturas durante varias décadas [9].

Dado el desarrollo que experimentó la tecnología de las computadoras personales en la década de 1990, el campo de la HPC comenzó a poblarse de arquitecturas basadas en microprocesadores. Estas arquitecturas basaban su diseño en una estrategia diferente: la utilización de múltiples núcleos de procesamiento paralelos [15]. Ejemplos de este tipo de diseños son los procesadores SIMD paralelos, que implementan el mismo paradigma que los procesadores vectoriales pero a través de conjuntos de muchos procesadores coordinados que ejecutan la misma instrucción, cada uno sobre distintos datos [14]; los multiprocesadores y los procesadores multinúcleo, que consisten de múltiples núcleos de procesamiento autónomos dentro de un mismo chip que acceden

a un espacio de memoria común, formando lo que es llamado un sistema de memoria compartida [5][12]; y las arquitecturas MPP (*massively parallel processor*, procesador masivamente paralelo), que consisten en un gran número de procesadores que no comparten memoria y se comunican mediante pasaje de mensajes a través de una red propietaria de alta velocidad [14].

El bajo costo por desempeño que comenzaron a presentar los microprocesadores hizo que estas nuevas arquitecturas paralelas comenzaran a desplazar a los procesadores vectoriales hacia afuera del mercado [8][15]. Según la primera edición de la lista TOP500, la cual se encarga de recabar información estadística sobre las supercomputadoras con mayor desempeño del mundo, en junio de 1993 el 31 % de los sistemas contenidos eran arquitecturas MPP o SIMD paralelas [16]. Diez años después, las arquitecturas vectoriales dejaron de figurar en la lista [15]. En la actualidad, la mayor parte de los sistemas de HPC son clústers, un tipo de arquitectura económica que consiste en la construcción de sistemas distribuidos similares a los MPP pero a través de componentes comerciales, como lo son los procesadores x86 tradicionales y las redes Ethernet o Infiniband [4]. También, muchos sistemas están utilizando aceleradores computacionales, que son dispositivos contenidos dentro de los nodos que funcionan por fuera del procesador central y son utilizados para cálculos específicos; ejemplos de este tipo de dispositivos son las unidades de procesamiento gráfico utilizadas para computación de propósito general (GPGPU, *general-purpose computing on graphics processing units*), los coprocesadores Intel Xeon Phi, y los aceleradores basados en arreglos de compuertas programables en campo (FPGA, *field programmable gate arrays*) [17]. La lista TOP500 de noviembre de 2022 cuenta únicamente con sistemas MPP o clúster, en una proporción de 10.4 % y 89.6 % del total de los sistemas, respectivamente [19].

No solo la conveniencia económica del uso de microprocesadores propició el aumento del paralelismo en los sistemas de alto desempeño, sino que, a partir de la segunda mitad de la década del 2000, las arquitecturas de microprocesadores empezaron una tendencia al aumento en la cantidad de núcleos de procesamiento internos que contienen. Tres razones, conocidas como *barreras*, son consideradas las determinantes de estas tendencias [5]:

- *Barrera de energía*: el consumo de energía aumenta no linealmente con el aumento de la frecuencia de ciclo de reloj de los procesadores. Esto genera que no sea posible seguir aprovechando las posibilidades de aumento de frecuencia que permite la creciente densidad de transistores sin exceder los límites de disipación de calor manejables mediante refrigeración por aire, lo cual además implicaría un uso ineficiente de la energía. En la figura 3 se puede observar cómo el límite de frecuencia es alcanzado alrededor del año 2004. A partir de entonces, las frecuencias se encuentran en su mayoría alrededor de los 3GHz.

- *Barrera del ILP*: como se mencionó anteriormente, se considera que se ha llegado a un límite en la posibilidad de conseguir mejoras de desempeño a través de técnicas de paralelismo a nivel de instrucción. De esta forma, no es posible aprovechar los transistores adicionales que la ley de Moore permite en paralelismo de bajo nivel, por lo que un potencial buen destino para estos transistores puede ser la integración de nuevos núcleos de procesamiento paralelos.

- *Barrera de memoria*: el desempeño de la tecnología usada para memoria principal tiene un crecimiento menor que el de las tasas de cálculo de los procesadores, lo que genera que las aplicaciones se vean limitadas por el desempeño de la memoria. El paralelismo puede ayudar a aprovechar mejor el tiempo correspondiente a la latencia de la memoria, si los recursos paralelos se usan para realizar trabajo adicional mientras se está esperando por un acceso a memoria.

El resultado de estos factores es una tendencia al aumento del paralelismo en los diseños de arquitecturas desde la década de 1990 hasta la actualidad, que se materializa en la multiplicidad

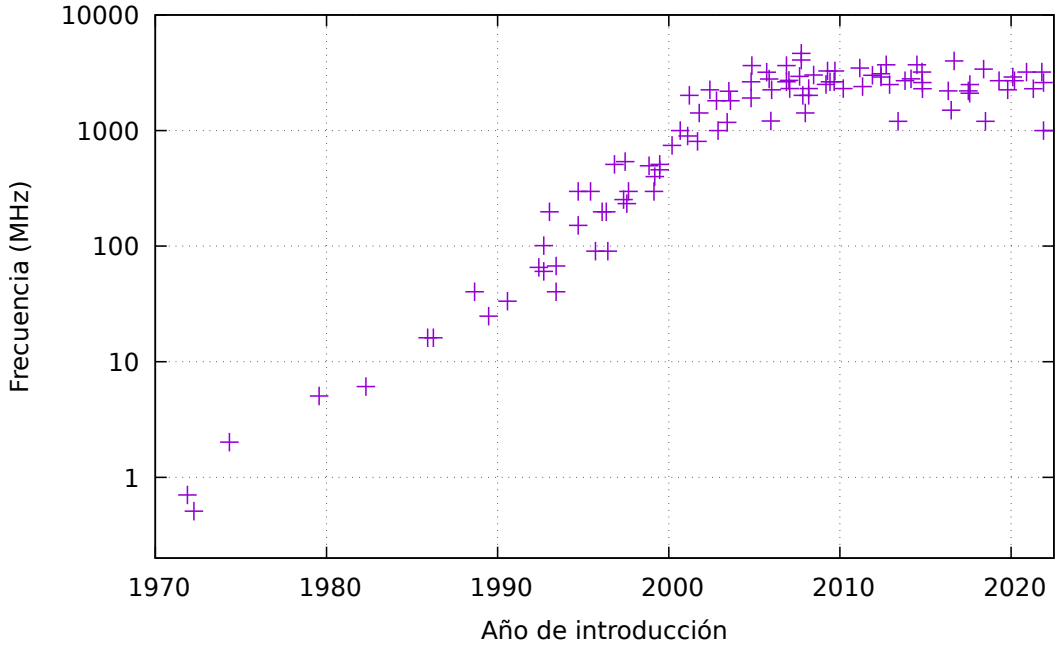


Figura 3: Frecuencias de ciclo de reloj de los modelos de procesadores Intel a través del tiempo. (Datos extraídos de Rupp [58].)

de procesadores por computadora, de núcleos de procesamiento por procesador e incluso de hilos de ejecución simultáneos por núcleo. Los datos de las figuras 4 y 5 sugieren estas tendencias. En la figura 4 se puede observar un crecimiento exponencial de la cantidad núcleos de la primera posición de la lista TOP500 hasta 2016, empezando a partir de 2004, lo que coincide con la fecha en que se llega a la barrera de energía. La figura 5 muestra un aumento exponencial en la cantidad de núcleos lógicos por chip de los modelos de procesadores, también aproximadamente a partir del año 2004.

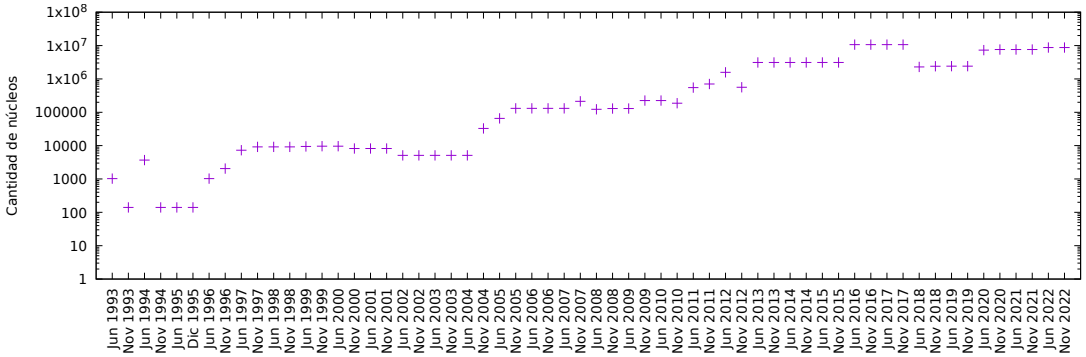


Figura 4: Cantidad de núcleos de la primera posición de la lista TOP500, desde sus inicios a la actualidad. (Datos extraídos de la lista Top500 [18].)

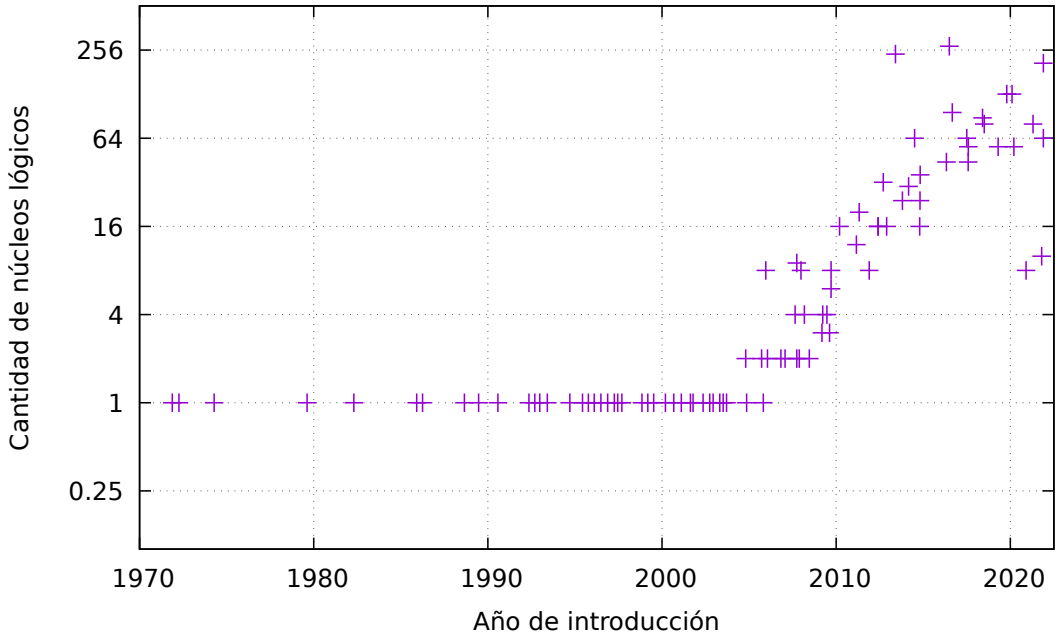


Figura 5: Cantidad de núcleos lógicos (núcleos o hilos de hardware, según si la arquitectura soporta múltiples hilos por núcleo) por chip de procesador para distintos modelos de procesadores a través del tiempo. (Datos extraídos de Rupp [58].)

Según la edición de noviembre de 2022 de la lista Top500, la computadora más rápida es Frontier, contenida en el Laboratorio Nacional Oak Ridge, en Tennessee, Estados Unidos. Es un MPP con un total de 8730112 núcleos con una capacidad de cómputo teórica de 1685.65 petaflops (cuatrillones de operaciones de punto flotante por segundo) [20].

El empleo de arquitecturas paralelas y su desarrollo tecnológico no es un factor transparente a las aplicaciones, sino que afecta a cómo las aplicaciones deben ser desarrolladas. Las aplicaciones que hacen uso de estos sistemas deben programarse para explícitamente indicar qué uso hacen de los recursos paralelos con los que cuenta. Hasta el choque con la barrera de energía, el desempeño de cualquier programa se veía mejorado con la aparición de nuevas tecnologías, las cuales mejoraban el desempeño secuencial aumentando la frecuencia. De esta manera, no era necesario mejorar el diseño del programa para obtener un mejor desempeño, sino que era suficiente con esperar a obtener procesadores más nuevos. Desde el momento en que las arquitecturas dejaron de mejorar su desempeño secuencial y empezaron a basar sus mejoras en agregar más paralelismo, los desarrolladores tienen que encargarse de diseñar sus algoritmos para que hagan buen uso de los recursos paralelos de los sistemas de los que disponen. Sin embargo, un efecto parecido aún puede conseguirse si los algoritmos diseñados son escalables, es decir, se adaptan correctamente a la cantidad de recursos paralelos de los sistemas donde son ejecutados [5][21].

2.1.3. Taxonomía de computación paralela

El término *paralelismo* se usa para referirse a la capacidad de ciertas arquitecturas de computadoras y algoritmos de realizar múltiples operaciones u obtener múltiples resultados de forma simultánea. El paralelismo se obtiene mediante técnicas tanto de hardware como de software: las

arquitecturas paralelas deben contar con recursos que puedan ser empleados de forma paralela, y el software que corre sobre una arquitectura paralela debe estar diseñado para explotar los recursos disponibles [22].

Las técnicas de paralelismo se pueden clasificar en 4 niveles [22]:

- Cuando se opera simultáneamente sobre múltiples datos, se habla de *paralelismo a nivel de datos*. Las instrucciones SIMD, que ejecutan una misma operación sobre todos los datos de un arreglo y están disponibles en muchas arquitecturas actuales, son un ejemplo de este nivel de paralelismo.
- Los procesadores que hacen *pipelining* de instrucciones dividen su ciclo de instrucción en etapas, ejecutando cada etapa en un recurso distinto del procesador. Esto les permite ejecutar múltiples instrucciones de forma simultánea, lo que es conocido como *paralelismo a nivel de instrucción*.
- Los sistemas operativos multitarea pueden ejecutar múltiples programas de forma concurrente. Esto es llamado *paralelismo a nivel de procesos*.
- A la vez, un programa puede usar *paralelismo a nivel de hilos* para ejecutar porciones suyas de forma simultánea.

Los sistemas de computación pueden clasificarse según las formas en las que implementan el paralelismo. La figura 6 muestra una clasificación de arquitecturas paralelas, organizada como un árbol, donde cada categoría puede contener varias subcategorías. El primer nivel del árbol corresponde a la taxonomía más reconocida de sistemas de computación, denominada taxonomía de Flynn [10]:

- Los sistemas clasificados como SISD (*single instruction stream, single data stream*, único flujo de instrucción, único flujo de datos) son sistemas con un único procesador, donde en un momento dado solo puede estar ejecutando un único programa, con su único flujo de instrucción, operando sobre datos individuales. Las arquitecturas **mononúcleo**, como las de todos los microprocesadores de Intel hasta alrededor de 2004 [5], son ejemplos de arquitecturas SISD.

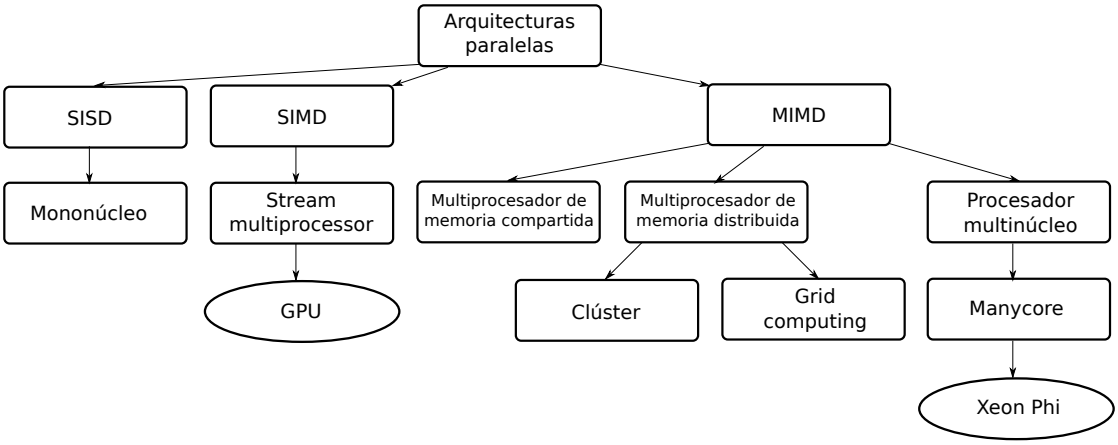


Figura 6: Taxonomía de computación paralela.

- Los sistemas multiprocesador y los sistemas multinúcleo cuentan con múltiples núcleos de procesamiento. Cada núcleo puede estar ejecutando, a la vez que los demás, un programa diferente, con su propio flujo de instrucciones y sus propios datos. Este tipo de sistemas se clasifican como MIMD (*multiple instruction stream, multiple data stream*; múltiples flujos de instrucción, múltiples flujos de datos).
- Existen arquitecturas clasificadas como SIMD (*single instruction stream, multiple data stream*, único flujo de instrucción, múltiples flujos de datos), las cuales están conformadas por muchos procesadores que comparten una única unidad de control. De esta forma, un único flujo de instrucción es ejecutado simultáneamente por todos los procesadores, cada uno sobre sus propios datos.

Más allá de que las arquitecturas SISD puedan presentar ciertos niveles de paralelismo, las clasificaciones de arquitecturas consideradas paralelas son SIMD y MIMD, debido a su utilización de múltiples unidades de procesamiento [10][22]. En el resto de la sección se enumeran distintas clases de arquitecturas de estos tipos y sus características generales.

Un tipo de arquitectura MIMD muy simple desde el punto de vista de su modelo de programación son los **multiprocesadores de memoria compartida**, los cuales consisten de un grupo de procesadores que comparten un espacio de memoria único. En la figura 7 se pueden ver dos diagramas de arquitecturas de este tipo, en las que múltiples procesadores se conectan a módulos de memoria compartidos. La memoria es utilizada para guardar código y datos de forma compartida entre todos los procesadores, permitiéndoles comunicarse entre ellos a través de ella. Es, entonces, un recurso por el que todos los procesadores compiten, y por lo tanto un potencial cuello de botella de desempeño del sistema. Esta situación se da particularmente en el caso de la figura 7a, dado que el bus y el módulo de memoria pueden ser utilizados únicamente por un solo procesador a la vez. Para atender este problema, se plantean arquitecturas como la de la figura 7b, donde el bus y el módulo de memoria único son sustituidos por una red de interconexión entre los procesadores y múltiples módulos de memoria. De esta forma, varios procesadores pueden acceder a la memoria simultáneamente a través de la red de interconexión y realizar múltiples operaciones de lectura o escritura simultáneas en los distintos módulos. Desde el punto de vista de cada procesador, el espacio de memoria es único. Según la arquitectura, acceder a distintos lugares de memoria puede tener distintas demoras. En los sistemas multiprocesador UMA (*uniform memory access*, acceso uniforme a la memoria), acceder a distintos lugares de memoria tiene la misma demora, independientemente del módulo en que se encuentre. En otras arquitecturas, los procesadores tienen memoria local, a la que acceden con menor demora, y

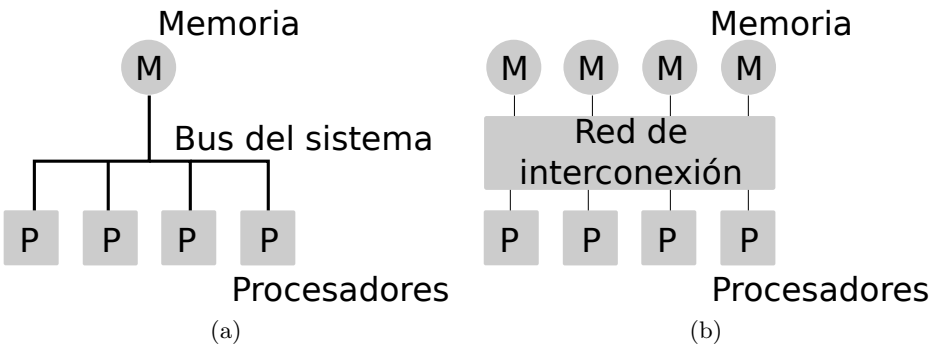


Figura 7: Diagrama de dos posibles arquitecturas de memoria compartida. (Figura basada en imagen extraída de Gebali [22].)

tienen una demora mayor para acceder a las memorias de otros procesadores. A estos casos se les refiere como sistemas multiprocesador NUMA (*non-uniform memory access*, acceso no uniforme a la memoria) [14].

En los sistemas de memoria compartida, cada procesador tiene su propia memoria caché, donde guarda sus instrucciones y datos en uso para un acceso más rápido. Para prevenir inconsistencias en situaciones en que varios procesadores trabajan sobre los mismos datos, estas arquitecturas deben implementar protocolos de coherencia de caché que se encarguen de mantener actualizadas las cachés de todos los procesadores.

Si bien el desarrollo de programas paralelos en sistemas de memoria compartida es muy similar al desarrollo de programas secuenciales, el desarrollador debe ser consciente del uso compartido de la memoria, identificando y controlando el acceso de los datos compartidos para mantener su integridad y resolver conflictos, lo cual puede implementarse mediante la ayuda de bibliotecas con herramientas de sincronización como POSIX u OpenMP [22].

Los sistemas **multinúcleo** son similares a los multiprocesadores de memoria compartida, pero, en lugar de comunicar múltiples procesadores a través de un bus externo compartido, en este caso los múltiples procesadores están contenidos dentro del mismo circuito integrado. El resultado es un sistema con buen desempeño y consumo de energía bajo, en comparación con el alto desempeño y consumo de energía de los sistemas multiprocesador, a la vez que presentan un costo de comunicación interprocesador bajo. Un diagrama de este tipo de arquitecturas puede verse en la figura 8. Los elementos que la integran son conjuntos de núcleos de procesamiento programables de propósito general y otros núcleos aceleradores con propósitos específicos, conectados con módulos de memoria compartida e interfaces de entrada y salida. Los sistemas multinúcleo más simples están conformados por dos o cuatro núcleos; sistemas con más cantidad de núcleos pueden necesitar interconexiones complejas.

Las arquitecturas multinúcleo pueden utilizarse para aplicaciones multitarea de propósito general y para aplicaciones con requerimientos de desempeño como pueden ser la criptografía o el procesamiento de imágenes [22].

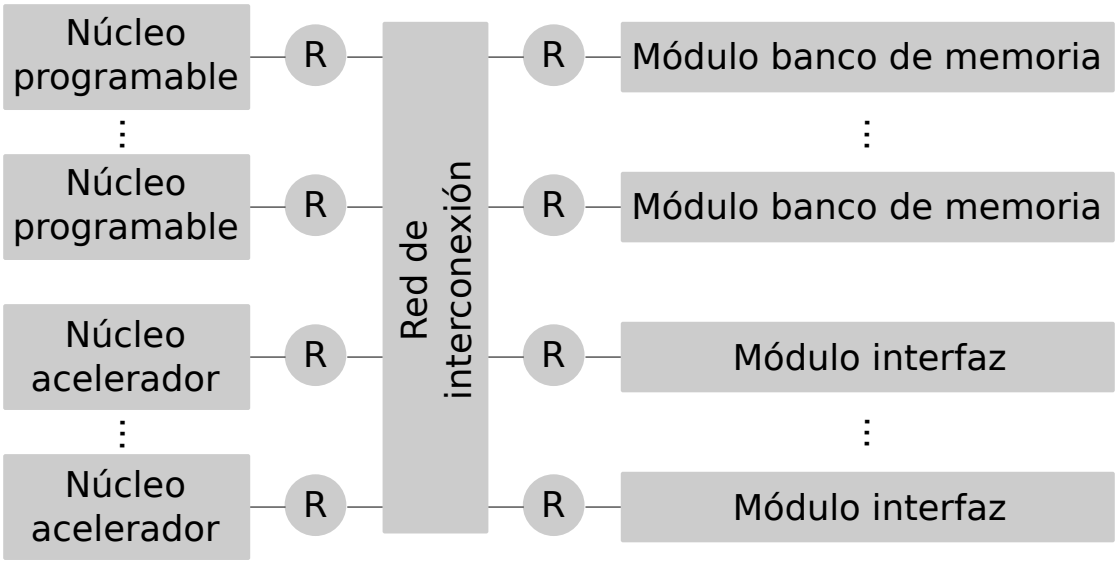


Figura 8: Diagrama de un sistema multinúcleo. (Figura basada en imagen extraída de Gebali [22].)

Para optimizar la eficiencia energética, algunos procesadores multinúcleo están constituidos por núcleos simples, de bajo desempeño, pero integrados en cantidades mayores a 50 dentro de un mismo circuito; este tipo de arquitecturas se denomina **procesadores *manycore***, y el coprocesador **Intel Xeon Phi** se encuentra dentro de esta clasificación. La gran cantidad de paralelismo de estas arquitecturas permite obtener un desempeño alto sin aumentar el consumo energético, pero las aplicaciones que corren sobre estos procesadores deben ser escalables para obtener un buen desempeño [29]. Que una aplicación sea escalable requiere pericia de parte de los desarrolladores de la aplicación y depende de la naturaleza de la aplicación y, frecuentemente, de la arquitectura específica donde está corriendo.

Los **multiprocesadores de memoria distribuida** están conformados, como se muestra en la figura 9, por múltiples procesadores, cada uno con su propio módulo de memoria, comunicados a través de una red de interconexión. La red puede ser tanto de área local como amplia, pudiendo abarcar miles de procesadores. Cada procesador puede acceder directamente a su propia memoria, pero, para acceder a los datos de otro procesador, debe comunicarse con él a través de la red, mediante un protocolo de pasaje de mensajes [22].

La arquitectura de HPC más utilizada actualmente es el **clúster** [4]. Estos sistemas surgieron a fines de la década de 1980 y principios de 1990 con el objetivo de obtener alto desempeño sin el costo y el riesgo de una supercomputadora tradicional. Un clúster es un sistema distribuido que consiste de múltiples nodos con tecnologías comerciales, como pueden ser computadoras personales o estaciones de trabajo con arquitecturas x86, conectados a través de una red de área local estándar capaz de soportar tasas de tráfico altas, como puede ser una red Gigabit Ethernet o Infiniband. Para el funcionamiento de este sistema es necesario un amplio conjunto de herramientas que implementen sus funcionalidades básicas, como lo son el pasaje de mensajes, el manejo de los recursos de cómputo y el sistema de archivos en red, además de otras utilidades para el desarrollo y ejecución de software paralelo [10]. Un diagrama simplificado de un clúster puede verse en la figura 10. En el sistema del diagrama, la máquina cliente funciona como interfaz del clúster, siendo utilizada por los usuarios para el acceso a los recursos de cómputo.

La red que interconecta los recursos de un sistema distribuido puede abarcar un área geográfica amplia. Por ejemplo, se pueden usar muchas computadoras conectadas a través de Internet para resolver un problema en conjunto. En estos casos, el término que se utiliza es **computación en grilla (*grid computing*)** o **computación en la nube (*cloud computing*)**. Para la computación científica, este tipo de sistemas puede manejar problemas de computación a gran escala, como las simulaciones de N-cuerpos, sísmicas, atmosféricas y oceánicas [22].

SM (*stream multiprocessor*) es un tipo de arquitectura asociada a las GPUs (*graphics processing unit*, unidad de procesamiento gráfico), ya que las arquitecturas de GPUs están formadas por varios SMs. Al invocar un programa a ser ejecutado por un SM, el SM distribuye el

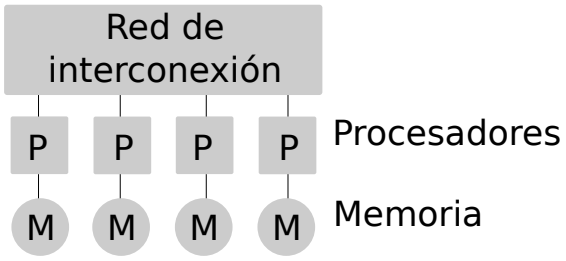


Figura 9: Diagrama de una arquitectura de memoria distribuida. (Figura basada en imagen extraída de Gebali [22].)

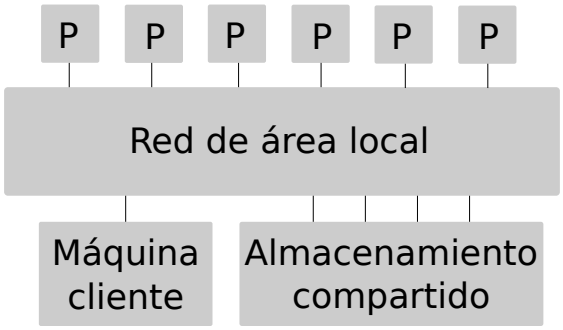


Figura 10: Diagrama simplificado de un clúster. (Figura basada en imagen extraída de Gebali [22].)

trabajo asignado entre los SPs (*streaming processor*) que lo conforman, que son múltiples procesadores con poca lógica de control que ejecutan de forma sincronizada en grupos. El diseño simplificado de los SPs permite integrar una gran cantidad de núcleos de procesamiento por SM, y por lo tanto una gran capacidad de cómputo paralelo [4][22].

Los SMs presentan capacidad para el procesamiento de flujos de datos. En particular, son adecuados para aplicaciones que presenten paralelismo de datos y alta intensidad computacional. Esto incluye no solo aplicaciones gráficas, sino también muchas aplicaciones de propósito general, para lo cual las GPUs también están siendo aplicadas [22].

2.2. Intel Xeon Phi

Xeon Phi presenta una arquitectura innovadora caracterizada por permitir el uso de un conjunto tradicional de herramientas de desarrollo de software. Este capítulo empieza comentando las tendencias tecnológicas que llevaron al desarrollo de Xeon Phi. A continuación, se describen las principales características de la arquitectura MIC. Se describe el software de sistema incluido en el coprocesador y en el nodo en el que se instala, y algunas de las herramientas de desarrollo de software con las cuenta. Finalmente, se mencionan ejemplos del mundo real en los que se usó Xeon Phi.

2.2.1. Antecedentes

En los inicios de los microprocesadores, el desarrollo del desempeño se basaba en cambios arquitectónicos para potenciar la cantidad de trabajo que un procesador realizaba por unidad de tiempo en forma secuencial. Estando la frecuencia de reloj de un procesador directamente relacionada a su velocidad de ejecución secuencial, este desarrollo se vio reflejado en un aumento exponencial en el ciclo de reloj de los procesadores. El aumento de la frecuencia a la que funcionan los procesadores tiene un efecto secundario: el aumento del consumo de energía por parte del procesador. El orden de este aumento de consumo no es lineal con respecto a la frecuencia, sino que tiene orden cúbico. Esto derivó en que, en 2004, el desarrollo tecnológico se topara con la llamada *barrera de energía*, momento en que los consumos de energía alcanzados llegaron a los límites de disipación de calor manejable mediante refrigeración por aire, estancándose en frecuencias de alrededor de 4GHz para los procesadores más rápidos. A partir de entonces, las nuevas arquitecturas de microprocesadores pasaron a basarse en la explotación de paralelismo, es decir, la utilización de múltiples unidades de procesamiento paralelas, para explotar las mejoras de desempeño permitidas por el desarrollo de la tecnología de transistores, sin buscar aumentar

la frecuencia de su ciclo de reloj [5][9]. De esta forma, por ejemplo, una familia de procesadores de núcleo único de 3GHz, según la ley de Moore, puede lanzar un nuevo modelo dos años después que tenga dos núcleos con la misma frecuencia y una arquitectura similar a la original, manteniendo el consumo de energía dentro de los parametros manejables. Este tipo de evolución es la que siguen los diseños de los microprocesadores x86 actuales, como la familia Intel Xeon, los cuales en la actualidad pueden contar con hasta 60 núcleos dentro de un mismo chip [23].

Más allá de que es viable usar procesadores con frecuencias cercanas al límite, resulta en un consumo de energía considerable, adquiriendo una magnitud particularmente importante en sistemas a gran escala con grandes cantidades de procesadores. Surge entonces la preocupación por buscar alternativas con mayor eficiencia energética, es decir, arquitecturas que presenten un menor consumo de energía en proporción a su desempeño, magnitud usualmente medida en FLOPS (*floating-point operations per second*, operaciones de punto flotante por segundo) por watt. Distintas técnicas de paralelismo dentro del procesador son aplicadas para afrontar este problema, como la implementación de unidades vectoriales y de múltiples hilos de hardware por núcleo. Sin embargo, la principal tendencia respecto a la computación energéticamente eficiente surge de una interpretación inversa del criterio utilizado para superar la barrera de energía: la utilización de múltiples núcleos de procesamiento paralelos, pero, en lugar de buscar escalar el desempeño manteniendo la frecuencia de ciclo de reloj de los núcleos, se busca disminuir el consumo energético utilizando frecuencias menores [4]. Un procesador con un único núcleo que funciona a cierta frecuencia puede ser sustituido por múltiples procesadores a menor frecuencia que obtengan un desempeño paralelo equivalente con menor potencia conjunta. Se puede observar el caso hipotético de la figura 11: si una tarea puede ser dividida en dos subtareas paralelas que ejecuten a la mitad de voltaje (y, por lo tanto, a la mitad de frecuencia) pero obteniendo el mismo desempeño en conjunto, entonces la densidad de potencia se disminuye por un factor de 8, y el consumo energético total por uno de 4. Sin embargo, este recurso tiene el costo de implicar una reducción del desempeño secuencial; el provecho de las mejoras en desempeño depende de la posibilidad de adaptación de las aplicaciones a los recursos paralelos disponibles, posibilidad que puede no existir en algunos casos. Los programas o porciones de programa que no puedan adaptarse y dependan fuertemente de la velocidad de ejecución secuencial se verán penalizados por la baja frecuencia de los elementos de procesamiento, presentando un desempeño menor que en alternativas con menos paralelismo y mayor frecuencia.

Ejemplos de arquitecturas que se basan en aplicar gran paralelismo para hacer cálculos energéticamente eficientes son las GPUs. En el caso de la arquitectura Kepler de Nvidia, la GPU está compuesta por 15 multiprocesadores de flujo, cada uno conteniendo 192 núcleos de procesamiento simplificados con una unidad de control compartida [24][25]. Estos dispositivos han demostrado, además de un poder de cómputo superior, una mayor eficiencia energética que las arquitecturas tradicionales para los problemas adecuados [25][26], lo que les ha valido la integración de las supercomputadoras más eficientes según la lista Green500 [27]. Sin embargo,

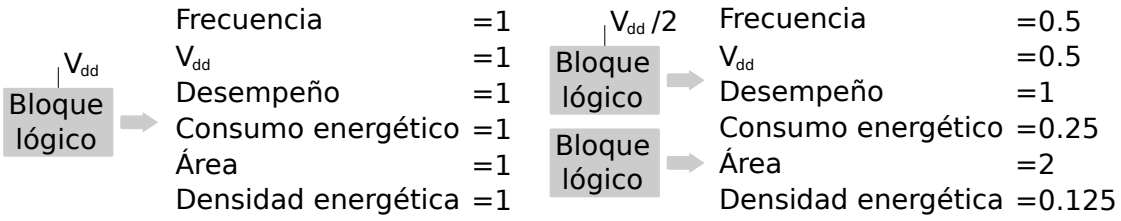


Figura 11: Ejemplo de utilización de paralelismo para mejorar la eficiencia energética. (Figura basada en imagen extraída de Padua et al. [4].)

dadas las radicales diferencias en el diseño de sus arquitecturas, el uso de esta tecnología dista mucho del de las arquitecturas multiprocesador tradicionales, no contando con la versatilidad para la implementación de muchos modelos de computación paralela, ni presentando un desempeño consistente para cualquier tipo de programa. En particular, los problemas que obtienen el mejor desempeño en GPUs son aquellos que presentan paralelismo masivo (al nivel de miles de hilos de ejecución) y simétrico (es decir, los hilos siguen un mismo flujo de ejecución, sin bifurcaciones) [25][28]. Además, tiene un modelo de programación específico, lo que impide la portabilidad de los programas que corre, requiriendo a su vez el conocimiento específico del modelo y las plataformas de software que lo implementan [7].

La familia de procesadores Xeon Phi es el resultado de los esfuerzos realizados por Intel por reducir el consumo de su familia de procesadores Xeon. Nuevamente, la eficiencia de estos dispositivos se basa en el uso de paralelismo y ejecución a baja frecuencia: su arquitectura, denominada Many Integrated Core (MIC), es descrita como un multiprocesador simétrico en un chip, conformada por alrededor 60 núcleos de procesamiento independientes que funcionan sobre memoria compartida ejecutando un mismo sistema operativo. Cada uno de estos núcleos consiste de un procesador x86 simplificado, sin capacidades de ejecución fuera de orden, pero con capacidades para múltiples hilos de ejecución de hardware y soporte de instrucciones SIMD. Su similitud con las arquitecturas x86, sumado a su soporte de los lenguajes y modelos de programación paralela tradicionales, llevan a que sean considerados como aplicables a los mismos modelos que cualquier sistema multiprocesador, adicionalmente presentando buen desempeño en los problemas aplicables a las GPUs, sin la necesidad de aprendizaje específico y además permitiendo el desarrollo de programas portables. Más allá de estas afirmaciones, se reconoce que el desempeño del Xeon Phi no es el mismo para cualquier tipo de problema, y que requiere ciertas consideraciones para obtener un desempeño óptimo [7][28][29].

Tanto la línea de coprocesadores Xeon Phi, denomindada Knights Corner, como su sucesora, la línea de procesadores Knights Landing, están descontinuadas e Intel dejó de darles soporte en enero de 2021. No se producen modelos que sigan la arquitectura MIC en la actualidad [48].

2.2.2. Arquitectura

Intel Xeon Phi se presenta como un sistema basado en una arquitectura manycore contenido dentro una tarjeta de expansión PCIe que se usa como coprocesador del nodo al que está conectado. Los detalles del sistema contenido dentro de la tarjeta dependen del modelo específico del procesador, pero todos siguen la arquitectura Many Integrated Core (MIC), que contiene alrededor de 60 núcleos que funcionan a poco más de 1GHz, con 4 hilos de hardware cada uno. Además de soportar instrucciones x86-64, cuenta con una unidad vectorial que trabaja con operandos de 512 bits. Dentro de la misma tarjeta, vienen incorporados hasta 16GB de memoria RAM GDDR5. Con estas características, Xeon Phi se muestra como un sistema que puede obtener alto desempeño de forma energéticamente eficiente en aplicaciones que muestren paralelismo de datos, teniendo además las ventajas de adaptarse también al paralelismo a nivel de hilos y de ser compatible con las arquitecturas x86 tradicionales, ventajas que no presentan otras arquitecturas masivamente paralelas como pueden ser las GPUs [28][29][30].

Como se puede ver en la figura 12, la organización interna del coprocesador consiste en una interconexión en forma de anillo bidireccional que comunica a los componentes de Xeon Phi, los cuales, para la línea Knights Corner, son entre 57 y 61 núcleos [31], 8 controladores de memoria, y la interfaz de sistema que comunica al coprocesador con el nodo principal mediante el bus PCIe. Conectados al anillo de forma paralela también se encuentran los componentes que conforman el denominado *tag directory*, que se encarga de implementar los protocolos de coherencia de caché [28][29][30].

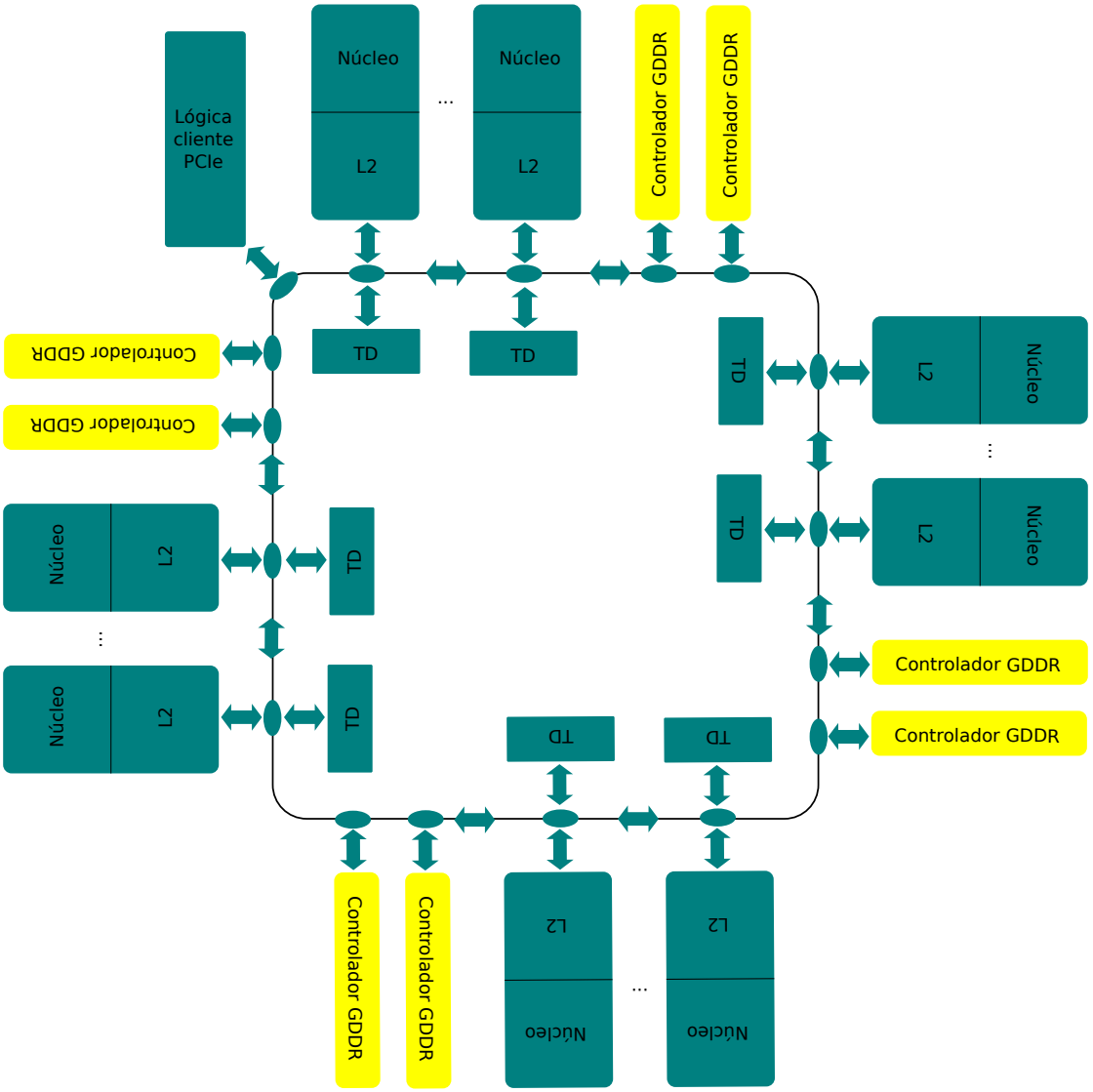


Figura 12: Organización de los núcleos y otros componentes en la arquitectura MIC. (Figura basada en imagen extraída de Jeffers et al. [30].)

La microarquitectura de los núcleos MIC está basada en la arquitectura P54c de Intel, la cual seguían los procesadores Pentium, adaptada para soportar el conjunto de instrucciones x86-64. El núcleo tiene un pipeline de ejecución en orden. Sin embargo, como adición a la arquitectura original, cuenta con 4 hilos de hardware, es decir, puede mantener el contexto de hasta 4 hilos de ejecución simultáneos. Asignar varios hilos a un mismo núcleo permite que, ante la situación de que un hilo tenga una instrucción en espera por una dependencia, el núcleo pueda continuar ejecutando las instrucciones de otros hilos. Las primeras etapas del pipeline se encargan de determinar de qué hilo serán las próximas instrucciones que pasen a la etapa de decodificación. Un detalle de estas primeras etapas es que, si bien pueden pasar hasta dos instrucciones por ciclo a la etapa de decodificación, no permiten que se pasen a decodificar instrucciones de un mismo

hilo en dos ciclos consecutivos, por lo que asignar más de un hilo por núcleo es una condición necesaria para obtener un desempeño óptimo [28][29][30].

Otra adición que hace la arquitectura MIC a los núcleos P54c son sus unidades para la ejecución de operaciones vectoriales. Xeon Phi cuenta con su propio conjunto de instrucciones SIMD, que trabajan sobre registros de 512 bits. El tamaño de los registros que usa como parámetros permite ejecutar 8 operaciones de operandos de 64 bits ó 16 operaciones de 32 bits simultáneas, como pueden ser operaciones de punto flotante de precisión doble o simple, respectivamente. Entre el conjunto de instrucciones SIMD soportadas, incluye [28][29][30]:

- instrucciones de 3 operandos, con 2 de lectura y uno de lectura y escritura, que se pueden usar para operaciones de multiplicación y adición fusionadas.
- *gather* y *scatter*, que son instrucciones que sirven cuando se accede a lugares de memoria que no son contiguos.
- *pack* y *unpack*, que son instrucciones que sirven para comprimir y descomprimir vectores ante la presencia de valores nulos.
- enmascaramiento, para cuando se quieren usar solo algunos elementos de un vector en una operación SIMD.

Todo núcleo cuenta con una memoria caché L1 para instrucciones y otra para datos, ambas de 32KB, y con una caché L2 de 512KB, ambos niveles con líneas de 64 bytes de tamaño y asociatividad en 8 vías. La coherencia entre las cachés L2 de los núcleos es controlada por el *tag directory* (TD), un componente distribuido conectado al anillo de interconexión que usa una combinación personalizada de los protocolos *Modified, Exclusive, Shared, Invalid* (MESI) y *Globally Owned, Locally Shared* (GOLS). Por cada núcleo, hay un componente del TD que vigila las operaciones de memoria del núcleo y actúa para mantener la coherencia: si un lugar de memoria solicitado no está presente en la caché del núcleo, pero sí en la de otro núcleo, se encarga de hacer la petición por los datos al núcleo correspondiente, y si se actualiza un lugar de memoria presente en otras cachés, se encarga de notificar de la actualización a los demás núcleos; también se encarga de enviar la solicitud de escritura a la memoria principal cuando una línea actualizada es liberada por todos los núcleos. El tamaño acumulado de las cachés L2 de todos los núcleos es de alrededor de 30MB, pero es necesario tener en cuenta que el aprovechamiento efectivo es menor si los núcleos comparten muchos datos (512KB en el peor caso, en el que todos los núcleos usan los mismos datos) [28][29][30].

Conectados al anillo de interconexión hay 8 controladores de memoria GDDR5 equiespaciados entre los demás componentes. Los controladores se reparten las direcciones de memoria de forma intercalada para optimizar el ancho de banda de acceso a memoria; teóricamente, entre todos los controladores se puede obtener un ancho de banda conjunto de 352GB/s [28][29][30]; este valor es competivivo en comparación con las arquitecturas de GPUs de la época en que se lanzó Xeon Phi [25]. La línea Knights Corner tiene modelos con 6GB, 8GB y 16GB de memoria [31].

La tarjeta de expansión que contiene al coprocesador se conecta a través de una ranura PCIe 2.0 x16. La interfaz de sistema que permite acceder al bus PCIe también está conectada al anillo de interconexión. Por medio de este bus se permite la comunicación bidireccional entre el sistema anfitrión y el coprocesador a través de sus memorias, y también la comunicación entre varios coprocesadores [28][29][30].

La tarjeta cuenta con un componente denominado *System Management Controller* que se conecta directamente al bus PCIe y sirve para control y monitoreo. Entre otras cosas, tiene sensores térmicos y de potencia y, con el fin de controlar la temperatura del coprocesador, permite limitar la frecuencia máxima de reloj del coprocesador, prender y apagar núcleos específicos

según su nivel de actividad, y controlar el ventilador en los modelos con refrigeración activa. El manejo de este componente está a cargo del sistema operativo del nodo en colaboración con el sistema operativo del coprocesador. Existen modelos con refrigeración activa (la tarjeta tiene un ventilador) y pasiva (usa los controles de frecuencia y voltaje para mantener la temperatura); existen modelos sin solución térmica para casos en que se quiera aplicar refrigeración externa [28][29][30].

El *translation lookaside buffer* (TLB) de Xeon Phi soporta páginas de 4KB, 64KB y 2MB; sin embargo, el sistema operativo ejecutado por Xeon Phi no soporta páginas de 64KB, por lo que solo las otras dos opciones están disponibles. Para páginas de 2MB, tiene dos niveles de TLB, con 8 entradas en el primer nivel y 64 entradas en el segundo, con asociatividad en 4 vías. Para páginas de 4KB y 64KB, L1 guarda 64 y 32 entradas, respectivamente, y L2 guarda entradas de directorio de páginas (en lugar de entradas de tabla de páginas). Usar 2MB como tamaño de página es útil para mejorar el desempeño en aplicaciones que acceden a grandes cantidades de datos contiguos [28][29][30].

2.2.3. Sistema operativo

Siguiendo el criterio de asemejarse a los sistemas tradicionales, Xeon Phi corre un sistema operativo autónomo basado en Linux. Para poder usar el coprocesador, Intel provee el conjunto de componentes de software *Manycore Platform Software Stack* (MPSS), que se instala en el sistema operativo del nodo y es compatible con varias distribuciones de Linux. MPSS incluye el módulo del kernel correspondiente al controlador de dispositivo para el coprocesador, comandos de utilidad para configurar, controlar y monitorear todos los dispositivos Xeon Phi del nodo, y el servicio `mpssd`, que es el principal encargado del proceso de arranque de los coprocesadores. El sistema operativo que corre Xeon Phi es una versión del kernel de Linux adaptada para la arquitectura MIC, adicionada de las bibliotecas de *Linux Standard Base* (LSB, incluyendo las bibliotecas estándar de C y C++) y el entorno minimalista de comandos BusyBox (un conjunto de comandos básicos de Linux) [57]. Por cada coprocesador en funcionamiento dentro un nodo, figura una interfaz de red virtual en el sistema operativo, a través de la cual se puede acceder mediante SSH a una consola en el dispositivo [28][29][30].

El componente principal con el que interactúa el usuario de Xeon Phi del lado del sistema operativo del nodo es el comando `micctrl`, que permite prender y apagar los coprocesadores. Como parte del proceso de arranque, `mpssd` se encarga de copiar el sistema operativo en la memoria RAM del dispositivo e inicializar el sistema de archivos. Dado que el sistema de archivos está alojado en la memoria RAM, cualquier modificación que se le haga se pierde al reiniciar el dispositivo. MPSS da la posibilidad de indicar archivos para ser copiados dentro del sistema de archivos como parte del procedimiento de arranque, lo cual es particularmente útil para hacer modificaciones permanentes a los archivos de configuración del directorio `/etc` (incluyendo el archivo `/etc/passwd` para gestionar identidades de usuarios) o para copiar bibliotecas dentro del directorio `/usr/lib64`. Para simplificar, `micctrl` permite hacer algunas configuraciones temporales del sistema operativo del coprocesador y autogenerar los archivos de configuración necesarios para que se hagan permanentes. Para compartir archivos con el nodo, MPSS permite montar directorios del nodo en el sistema de archivos del coprocesador. Las interfaces virtuales creadas para los coprocesadores pueden configurarse de múltiples formas, pudiendo asignar IPs y MACs fijas o puentear un dispositivo Ethernet del nodo, entre varias posibilidades [28][29][30].

MPSS contiene, tanto en el sistema operativo del nodo como en el del coprocesador, la implementación de una interfaz llamada *Symmetric Communications Interface* (SCIF) para la comunicación DMA de forma bidireccional a través del bus PCIe. Para soportar la biblioteca Intel MPI, MPSS implementa la interfaz *Open Fabrics Enterprise Distribution* sobre SCIF, lo que

permite comunicación de alto desempeño del coprocesador con procesos MPI dentro del mismo nodo; MPI (*Message Passing Interface*) es un protocolo estándar para la comunicación entre procesos mediante pasaje de mensajes. En el caso de contar con un dispositivo Infiniband dentro del nodo, los módulos *Coprocessor Communications Link* (CCL) de MPSS permiten comunicarse con procesos MPI remotos [28][29][30].

2.2.4. Herramientas de desarrollo

Las herramientas para desarrollo de aplicaciones en Xeon Phi originalmente estaban incluidas dentro de la suite de software Intel Composer XE [29][30] y luego dentro de Intel Parallel Studio XE, actualmente llamada Intel oneAPI [34]. Parallel Studio era una suite de software para el desarrollo de aplicaciones de computación paralela; entre las herramientas para dar soporte a Xeon Phi, incluía los compiladores oficiales de Intel para C, C++ y Fortran, bibliotecas paralelas y el perfilador VTune Amplifier [32]. La última versión de Parallel Studio que soportó la línea Knights Corner fue Parallel Studio XE 2017 [33].

Las principales herramientas para el desarrollo de aplicaciones en Xeon Phi son los compiladores oficiales de Intel para los lenguajes C, C++ y Fortran. Los compiladores son los mismos que se usan para arquitecturas Intel 64 que, indicando la bandera de compilación `-mmic`, permiten compilar de forma cruzada aplicaciones que luego se pueden copiar a dispositivos Xeon Phi para ser ejecutadas. Este modo de ejecución es llamado *ejecución nativa*. En general, cualquier programa que se pueda compilar para arquitecturas Intel 64 puede compilarse para arquitecturas MIC con los compiladores oficiales. Además, los compiladores oficiales incluyen operaciones vectoriales y clases específicas de C++ para trabajar con vectores, cuyo uso resulta en código dependiente de la plataforma [28][29][30]; las formas portables de obtener vectorización se comentan en el capítulo 2.4.

El modo de ejecución alternativo al nativo es el modo de ejecución *offload*, en el que, dado un programa que ejecuta en el procesador del nodo principal, ciertas porciones de código del programa son marcadas para ejecutar en coprocesadores disponibles. Este modo de uso es similar al usado para la computación de propósito general en GPUs. El compilador oficial de Intel soporta extensiones propietarias para que el desarrollador indique qué porciones de código son las que ejecutan en los coprocesadores mediante directivas de preprocesador llamadas *pragmas*. Puntualmente, el pragma `offload` indica que el bloque de código siguiente se ejecuta en un coprocesador. Ejecutar en modo offload implica que los datos que se vayan a usar estén presentes en el coprocesador, por lo que existe la posibilidad de indicar la transferencia de datos a través del bus PCIe para las ejecuciones offload. El pragma `offload` permite indicar qué conjuntos de datos deben transferirse para una ejecución, y si deben transferirse desde el procesador principal al coprocesador o en sentido contrario. El pragma `offload_transfer` también puede usarse para indicar transferencias de datos. Todas las operaciones offload se pueden ejecutar de forma bloqueante (el programa principal espera a que la ejecución offload termine antes de continuar) y no-bloqueante (el programa principal sigue y ejecuta en paralelo junto con la ejecución offload), pudiendo indicar operaciones offload no-bloqueantes anteriores que deban terminar antes de ejecutar cada operación. El pragma `offload_wait` sirve para que el programa principal espere por todas las operaciones offload anteriores que estén ejecutando en los coprocesadores indicados [28][29][30]. El listado 1 muestra un ejemplo de uso de un pragma para indicar ejecuciones offload. El pragma `offload` de la línea 13 indica que el siguiente bloque de código debe ejecutarse en Xeon Phi. En los parámetros del pragma se indica que los arreglos `fa` y `fb` y el valor de la variable `a` deben copiarse al coprocesador antes de ejecutar.

Xeon Phi soporta las bibliotecas paralelas OpenMP e Intel TBB y la biblioteca de matemática Intel MKL [28][29][30]. En el capítulo 2.3 se comenta más a fondo sobre cómo se usan.

```
1  int main(int argc, char *argv[] )
2  {
3    float a = 1.1;
4    float *fa,*fb;
5    fa = (float*) malloc(FLOPS_ARRAY_SIZE*sizeof(float));
6    fb = (float*) malloc(FLOPS_ARRAY_SIZE*sizeof(float));
7    for(i=0; i<FLOPS_ARRAY_SIZE; i++)
8    {
9      fa[i] = (float)i + 0.1;
10     fb[i] = (float)i + 0.2;
11   }
12
13   #pragma offload target(mic) in(a) \
14       in(fa:length(FLOPS_ARRAY_SIZE) alloc_if(1) free_if(1)) \
15       in(fb:length(FLOPS_ARRAY_SIZE) alloc_if(1) free_if(1))
16   {
17     int j, k;
18     for(k=0; k<FLOPS_ARRAY_SIZE; k++)
19     {
20       //
21       // scale 1st array and add in the 2nd array
22       // example usage - y = mx + b;
23       //
24       for(j=0; j<LOOP_COUNT; j++)
25       {
26         fa[k] = a * fa[k] + fb[k];
27       }
28     }
29   }
30
31   // ...
32 }
```

Listado 1: Ejemplo de ejecución offload. El programa está basado en código extraído de Jeffers et al. [30].

2.2.5. Perfilador: *VTune Amplifier*

Intel VTune Amplifier es un perfilador incluido dentro de la suite Parallel Studio. VTune sirve para analizar el desempeño individual de aplicaciones o de los procesos concurrentes dentro de un sistema operativo. Usa sus propios controladores del sistema operativo para recolectar información del estado del hardware durante la ejecución de los programas analizados. Entre la información recolectada, VTune incluye datos de eventos sobre el estado de los núcleos del procesador, el uso de caché y el acceso a memoria; los eventos recolectados se pueden asociar a las funciones y líneas del código fuente que están ejecutando cuando se generan. Luego de recolectados los análisis, la información puede observarse en la interfaz gráfica de VTune, organizada en varios tipos de reportes diferenciados por el tipo de problemas en que se enfocan en ayudar a resolver. Por cada métrica que se muestra en un reporte, la interfaz gráfica incluye su definición y una explicación de sus implicancias en el desempeño. En los reportes, se observan resaltadas las métricas que se encuentran por fuera de los valores óptimos, según umbrales por defecto; para cada métrica resaltada, la interfaz da una explicación posible de las razones por las que el valor no es óptimo para el programa y sugerencias para su optimización [35]. VTune Amplifier actualmente se desarrolla bajo el título *VTune Profiler* [36].

VTune permite hacer distintos *tipos de análisis* que tienen propósitos diferentes. Cada análisis recolecta su propio conjunto de métricas [35]. Los tipos de análisis mencionados a continuación son los que VTune puede realizar en Xeon Phi.

El tipo de análisis *Advanced Hotspots* sirve para identificar los *hotspots* del programa, que son las secciones de código en las que el programa pasa más tiempo. Esto puede ser tanto porque son funciones lentas o porque son llamadas muchas veces. En todo caso, mejorar el desempeño de un hotspot debería impactar considerablemente en el desempeño de la ejecución en general [35].

El análisis de tipo *General Exploration* se basa en la recolección de eventos de hardware para mostrar la frecuencia con la que se dan en el transcurso del programa. Puede ser útil como punto de partida para investigar las razones de los problemas de hardware una vez que ya se corrieron análisis para identificar los hotspots. Recolecta métricas sobre uso de memoria (uso de cache, latencia de misses y eficiencia de TLB) y vectorización [35]. Cepeda [37] describe un proceso de optimización específico de Xeon Phi basado en este tipo de análisis.

La documentación de VTune indica que el análisis *Memory Access* sirve para identificar problemas relacionados a la memoria y tiene métricas para las lecturas y los misses en los distintos niveles de memoria, como sus cantidades y la cantidad de ciclos que se pierden. Sin embargo, para Xeon Phi solamente muestra valores relacionados al uso de ancho de banda de memoria [35].

Por último, el tipo de análisis *HPC Performance Characterization* sirve para identificar qué tan bien se usan los recursos de hardware (CPU, memoria y unidades de punto flotante) en una aplicación de cálculo intensivo [35].

La interfaz gráfica organiza las métricas en lo que llama *puntos de vista*. Cada punto de vista muestra su propio grupo de métricas, organizado en varias ventanas que ordenan las métricas de distinta forma. Hay puntos de vista que están disponibles en varios tipos de análisis. Los puntos de vista disponibles son [35]:

- *General Exploration*: Ayuda a identificar dónde no se está haciendo el mejor uso de los recursos de hardware disponibles, a partir de métricas derivadas de los eventos de hardware.
- *Hardware Events*: Muestra estadísticas de los eventos de hardware recolectados.
- *Hardware Issues*: Ayuda a identificar dónde no se está haciendo el mejor uso de los recursos de hardware.

- *Hotspots*: Ayuda a identificar los hotspots del programa.
- *Memory Usage*: Ayuda a entender qué tan efectivamente se usan los recursos de memoria e identificar potenciales problemas de acceso a la memoria.
- *HPC Performance Characterization*: Sirve como punto de partida para observar el análisis que tiene el mismo nombre.

Las ventanas se repiten entre los distintos puntos de vista, mostrando distintas métricas para cada uno. Las ventanas disponibles son [35]:

- *Summary*: Es el punto de entrada de cada punto de vista. Muestra un resumen de las métricas del punto de vista, resaltando con rojo aquellas con valores que indican problemas de desempeño (a partir de umbrales determinados por los desarrolladores de VTune).
- *Sample Count - Hardware Events* y *Event Count - Hardware Events*: Son dos ventanas muy similares. Permiten analizar las cantidades de veces que ocurrieron los eventos de los tipos correspondientes al análisis. Se diferencian en que *Event Count* hace una estimación de los valores en función de los datos recolectados, mientras que *Sample Count* se usa cuando los datos recolectados permiten saber los valores reales. En el panel *Event Count Pane* se muestran las cantidades de veces que se dan los eventos agrupadas por unidad de programa (función, módulo, hilo, etc.). El panel *Timeline Pane* muestra una gráfica de las cantidades por hilo y en total. *Call Stack Pane* muestra los stacks de los que formó parte la unidad seleccionada del *Event Count Pane* y qué porción de la cantidad total de eventos aportó cada stack.
- *Caller/Callee*: Esta ventana sirve para visualizar las métricas de una función junto a las funciones que la llaman en algún momento del programa (*callers*) y las funciones que esa misma función llama (*callees*). Tiene un panel para seleccionar la función que se quiere visualizar; una vez seleccionada, un panel adicional muestra sus callers y otro muestra sus callees. Para cada métrica se muestran dos valores: *Self*, que indica el valor de la métrica contando solo al código de la función seleccionada, sin incluir a las funciones que llama, y *Total*, que incluye los valores de las invocaciones de la función a otras funciones.
- *Top-down Tree*: Esta ventana muestra las mismas métricas que *Caller/Callee* pero en un solo panel organizado según el árbol de llamadas del programa. Generalmente empieza en la función `main`, la cual se puede expandir para mostrar las funciones a la que llama, y así sucesivamente. También divide los valores entre *Self* y *Total*.
- *Bottom-up - Hotspots/Hotspots by CPU Usage*: Esta ventana es como la ventana *Top-down Tree* pero en este caso se listan todas las unidades del programa según la agrupación elegida (por ejemplo, funciones, pero hay otras agrupaciones) y, al expandirlas, se muestran las unidades que llaman a esa unidad (por ejemplo, en el caso de las funciones se forma el árbol de llamadas pero invertido).
- *Bottom-up - Memory Usage*: Esta ventana sirve para identificar secciones de código y objetos de memoria con problemas de memoria. Tiene un panel llamado *Timeline* que muestra histogramas de las métricas. Se puede usar para detectar las zonas problemáticas, marcarlas y elegir que se filtren los valores de la ventana a esas zonas. Luego, se puede pasar al panel *Grid*, que lista las funciones y los objetos de memoria junto a sus métricas para el periodo filtrado. Un tercer panel, *Stack*, muestra los stacks de los que formó parte la función u objeto de memoria seleccionado.

- *Platform*: Esta ventana es más útil para sistemas heterogéneos que para Xeon Phi. En Xeon Phi, la información más relevante que muestra es el uso de CPU.

2.2.6. Trabajos académicos

En principio, los programas que corren en las arquitecturas de memoria compartida x86 pueden correrse en Xeon Phi. Xeon Phi es considerado adecuado para los mismos casos que las GPUs, con la ventaja de tener flexibilidad para adaptarse a otros tipos de problemas. Además, su similaridad con las arquitecturas x86 y el hecho de que soporta algunos de sus lenguajes y bibliotecas reconocidos es potencialmente beneficioso para tener una curva de aprendizaje reducida en su adopción. La portabilidad con x86 de los programas desarrollados para Xeon Phi no solo se da en la compatibilidad, sino que además se afirma que las optimizaciones que se aplican en Xeon Phi también tienen impacto positivo en el desempeño en x86 [30]. Sin embargo, Fang et al. [38] afirman que las optimizaciones del compilador no son suficientes para obtener un buen desempeño en aplicaciones reales con Xeon Phi, que es necesario invertir un esfuerzo considerable en optimización y paralelización, y que, en particular, es necesario el uso de vectorización mediante código no portable.

Reinders et al. [39] recopilan multiples ejemplos de aplicaciones científicas implementadas en Xeon Phi, haciendo análisis del desempeño que presentan y comentando las optimizaciones que se usaron. En particular, incluyen dos ejemplos de problemas de n-cuerpos relativamente simples, en los que se simula el movimiento de un conjunto de partículas en función de las interacciones entre ellas. En el primer ejemplo, se obtiene un desempeño correspondiente al 64 % del desempeño teórico del coprocesador en precisión simple y 51 % del desempeño teórico en precisión doble. Las primeras versiones del programa obtenían un buen desempeño aprovechando todos los hilos de hardware disponibles y aplicando vectorización, pero el desempeño se deterioraba al aumentar el tamaño del problema: se mantenía alto en sistemas de entre 5000 y 30000 partículas, y decaía notoriamente y de forma lineal para sistemas mayores. La versión final implementa optimizaciones de localidad de memoria, aplicando técnicas *cache tiling* y procurando que los hilos de cada núcleo compartan datos. Esta versión muestra un desempeño sostenido, independientemente del tamaño del problema. Pruebas realizadas en el procesador del nodo, de arquitectura x86, muestran que las optimizaciones del programa hechas para Xeon Phi mejoraron en 10 % el desempeño del mismo programa en el procesador. El segundo ejemplo implementa un programa similar y obtiene un desempeño similar. Se llega a un desempeño de alrededor de 1.1TFLOPS de precisión simple usando la vectorización automática generada por el compilador. Se obtiene un desempeño mayor, de 1.4TFLOPS, usando operaciones intrínsecas de vectorización. El programa es una simulación en varios pasos, donde cada paso consiste de un bucle de dos niveles: el nivel exterior itera sobre todas las partículas, calculando su aceleración; en el nivel interior, se itera sobre todas las partículas para calcular su fuerza de interacción con la partícula de la iteración actual del nivel exterior. Las pruebas indican que, para sistemas con cantidad de partículas menor a 16384, el mejor desempeño se obtiene vectorizando el bucle interior, pero que, para sistemas con mayor cantidad de partículas, se obtiene mejor desempeño vectorizando el bucle exterior.

En Wang et al. [28], se muestran dos ejemplos de aplicaciones científicas implementadas usando arquitecturas heterogéneas. La primera aplicación implementa el algoritmo *simultaneous iterative reconstruction technique* (SIRT), usado para generar imágenes de tomografía. Para calcular una imagen de 2048*2048 puntos, el tamaño de los datos necesarios está en el orden de MBs y entran completamente dentro de la memoria del coprocesador. El algoritmo divide los datos en secciones de la imagen final que pueden calcularse de forma independiente. La implementación ejecuta de forma simultánea en el procesador del nodo y en todos los dispositivos MIC disponibles, entre los cuales se reparten las distintas secciones. Para esto, ejecuta un hilo paralelo para

ejecutar en cada dispositivo MIC del nodo y un hilo más, correspondiente al procesador del nodo. El código que ejecuta el hilo de cada dispositivo es prácticamente el mismo, con la particularidad de que el código que ejecuta en los coprocesadores tiene directivas de offload. Para aprovechar el paralelismo disponible dentro de cada dispositivo, el programa usa paralelismo anidado: el hilo de cada dispositivo levanta un hilo anidado por cada hilo de hardware del dispositivo. Las regiones paralelas anidadas, locales a cada dispositivo, consisten de un bucle de tres niveles. El paralelismo a nivel de tareas se obtiene dividiendo el nivel exterior del bucle entre los hilos disponibles en el dispositivo. La vectorización se realiza en el nivel más interior del bucle, y se muestra que, para obtenerla, es necesario modificar el algoritmo secuencial original: se invierten los dos bucles interiores para garantizar que se accede a los datos en el sentido de las direcciones de memoria y se indica que las iteraciones del bucle más interno son independientes mediante el pragma `ivdep`. Según los resultados exprimentales, la versión final del algoritmo, ejecutando en un nodo con un procesador de 16 núcleos y dos Xeon Phi, se obtuvo un speedup de 74.83 frente a la ejecución secuencial dentro del mismo nodo.

La segunda aplicación mostrada en Wang et al. [28] es una implementación de *Large Eddy Simulation*, un método usado en mecánica de los fluidos para el modelado de turbulencias, aplicando el método *Lattice Boltzmann* como algoritmo de resolución. Debido a los requerimientos de memoria de la aplicación, la resolución usa múltiples nodos de un clúster, con varios dispositivos MIC en cada nodo. La implementación del algoritmo distribuido que se muestra está hecha con MPI, invocando un proceso por cada procesador y coprocesador del sistema distribuido. Dentro de cada proceso, se invoca la cantidad de hilos para mantener ocupados a los hilos de hardware disponibles en el dispositivo en que ejecuta. Los procesos que ejecutan en coprocesadores lo hacen mediante ejecución *offload*. El algoritmo consiste de múltiples iteraciones que se deben ejecutar en orden debido a dependencias de datos; parte de los datos son copiados por única vez del procesador al coprocesador al inicio del algoritmo, y en posteriores iteraciones se indica el parámetro `nocopy` en las directivas de offload para que los datos se reutilicen; otra parte de los datos es transferida hacia y desde el coprocesador en cada iteración, debido a que los procesos deben intercambiar parte de sus resultados con los demás antes de continuar. El balanceo de la carga se hace de forma equitativa entre los nodos disponibles. Dentro de cada nodo, debido a que hay diferencias entre la capacidad computacional del procesador y el coprocesador, se le asignan distintos tamaños de carga al procesador y a los coprocesadores; los tamaños de carga se determinaron empíricamente, comparando los tiempos de ejecución del algoritmo en un procesador y un coprocesador. En la publicación se muestran resultados de desempeño de pruebas realizadas ejecutando la aplicación en un sistema compuesto por dos nodos con dos procesadores y dos coprocesadores por nodo, variando la cantidad de nodos y coprocesadores usados en cada ejecución. La ejecución que usó todos los procesadores y coprocesadores disponibles se mostró 156.87 veces más rápida que la ejecución secuencial, y 6.71 veces más rápida que la ejecución paralela en los procesadores de un único nodo; las ejecuciones paralelas en un único nodo usando uno y dos coprocesadores se mostraron 1.7 y 3.4 veces más rápidas, respectivamente, que la ejecución paralela en un único nodo sin usar coprocesadores.

Se llama *dinámica molecular* a casos particulares de los problemas de n-cuerpos en los que se simula un conjunto de átomos o moléculas. Pennycook et al. [40] muestran que los problemas de dinámica molecular se pueden beneficiar del uso de operaciones SIMD con vectores grandes, como los de Xeon Phi. En su artículo, comentan las optimizaciones realizadas a un programa existente para permitir la vectorización en un procesador Xeon y en un coprocesador Xeon Phi. El código no pudo vectorizarse de forma automática por el compilador, por lo que los autores implementaron la vectorización usando las operaciones SIMD intrínsecas de cada arquitectura. Dado que el algoritmo requiere leer y escribir de posiciones de memoria no contiguas, para permitir la vectorización, se aplica una optimización que lee los lugares de memoria de varios operandos de

iteraciones contiguas y los reestructura en un único registro SIMD que luego es usado en operaciones vectoriales, cuyos resultados son, finalmente, escritos en las direcciones de memoria dispersas correspondientes; los autores identifican esta técnica como una transformación *array-of-structs* a *struct-of-arrays*, aunque otros autores identifican de esa forma a una técnica diferente [5][30]. Las instrucciones *gather* y *scatter* de la arquitectura MIC fueron particularmente útiles para aplicar la técnica. También, las instrucciones de empaquetado SIMD se usaron para solucionar un problema de dependencias entre iteraciones del bucle que se procuraba vectorizar. Tanto en el procesador como en el coprocesador, una bifurcación en el bucle vectorizado se solucionó usando enmascaramiento. El tamaño del problema resuelto en las pruebas realizadas se determina por la cantidad de átomos de la simulación; las pruebas fueron desde 32 miles hasta 2048 miles de átomos. En las pruebas realizadas con el algoritmo vectorizado, el coprocesador probó ser 1.42 veces más rápido que el procesador, pero solo para ejecuciones mayores a 256 miles de átomos. Para ejecuciones de tamaños menores, el procesador presentó desempeño equivalente o mejor. El algoritmo vectorizado se mostró escalable, y el mejor desempeño se obtuvo con la ejecución combinada del procesador y el coprocesador para el tamaño de problema más grande. Los autores notan que se obtuvo mejor desempeño agregando un algoritmo de resolución de conflictos de escritura que agregando operaciones redundantes, recurso que se aplica frecuentemente debido a que es efectivo en GPUs.

Todas las aplicaciones mencionadas hasta el momento usan OpenMP para obtener paralelismo a nivel de hilos. Se hizo más difícil encontrar ejemplos de aplicaciones que usen TBB, que también es un modelo soportado por Xeon Phi. Mirsoleimani et al. [41] comentan una implementación de un *árbol de búsqueda Monte Carlo* usando TBB en Xeon Phi. La implementacion resultante demostró escalar con buena eficiencia en Xeon Phi, pero fue superada en desempeño por una solucion de manejo de hilos alternativa y más simple implementada por los autores. No se hacen comentarios sobre vectorización en el artículo. Tousimojarad et al. [42] corrieron tres algoritmos muy simples a modo de *benchmark* para comparar sus implementaciones usando distintos modelos de programación paralela en Xeon Phi. Los resultados muestran un desempeño en general levemente menor para TBB en comparación con OpenMP. Fiksman et al. [43] muestran la optimización de un modelo financiero, originalmente implementado usando OpenMP. Los autores comentan que adaptar la aplicación a TBB permitió un mayor control de la planificación de las tareas paralelas y una reducción de la sobrecarga de trabajo generada por la planificación, cuestiones que se veían complicadas por cómo OpenMP maneja el paralelismo anidado. La adaptación consistió, simplemente, en sustituir las directivas de OpenMP por sus equivalentes de TBB. Una optimización extra fue necesaria para permitir la vectorización y mejorar el uso de caché, la cual consistió en invertir el orden de los bucles para favorecer el acceso a memoria en direcciones contiguas; también se agregó una directiva para ayudar al compilador a vectorizar de forma automática. Los resultados obtenidos para el programa ejecutando en un único nodo con dos procesadores y un coprocesador fueron satisfactorios.

Xeon Phi era usado en la supercomputadora Tianhe-2 del Centro Nacional de Supercomputación en Cantón, China, desarrollada por la Universidad Nacional de Tecnología de Defensa. Tianhe-2 estaba conformada 16000 nodos de cómputo, cada uno conformado por dos procesadores Intel Xeon E5-2600 (de arquitectura Ivy Bridge, con 12 núcleos cada uno), tres coprocesadores Xeon Phi de la familia 3100 (con 57 núcleos cada uno) y 64GB de memoria RAM [44]. Alcanzaba un desempeño máximo de 3386PFLOPS en el benchmark LINPACK [45], y estuvo primera en la lista Top500 desde julio de 2013 a noviembre de 2015 [46]. En la edicion de junio de 2016 de lista Top500, Tiahne-2 fue superada por la supercomputadora Sunway Taihulight, ubicada en el Centro Nacional de Supercomputacion en Wuxi, China, que está conformada principalmente por procesadores SW26010, una arquitectura manycore desarrollada localmente [47]. Hubo sanciones de parte del gobierno de Estados Unidos que impidieron que Intel siguiera comerciando con la

universidad militar que mantenía Tianhe-2, lo que llevó a que en su actualización, Tianhe-2A, Xeon Phi se reemplazara con coprocesadores que siguen una arquitectura propietaria denominada Matrix-2000, también manycore [45].

2.3. Programación paralela: patrones, modelos y bibliotecas

Los lenguajes de programación tradicionales son, en su mayoría, imperativos, y siguen un modelo secuencial. Independientemente del paralelismo a nivel de instrucción que presente la computadora en la que se ejecutan, o del reordenamiento de las instrucciones realizado por el compilador, los programas, desde el punto de vista de quien los desarrolla o quien observa su código, se visualizan como un conjunto de pasos secuenciales que son ejecutados por el procesador en el orden en que se muestran. Este modelo fue adecuado para las arquitecturas de microprocesadores por muchos años. El desarrollo tecnológico de las arquitecturas monoprocesador permitía que los algoritmos se vieran beneficiados de los aumentos de desempeño secuencial sin tener que modificar su diseño. A partir de principios del siglo XXI, con la emergencia de las arquitecturas multinúcleo y el estancamiento de las frecuencias de reloj, obtener un buen desempeño secuencial dejó de ser suficiente para explotar las capacidades disponibles en las nuevas computadoras. Los algoritmos debieron empezar a tener en cuenta de forma explícita el paralelismo para poder aprovechar la capacidad de los múltiples núcleos de procesamiento. Con la tendencia al aumento en la cantidad de núcleos de las arquitecuras emergentes, se hace además necesario que los algoritmos tengan la capacidad de ser escalables para poder aprovechar las mejoras de desempeño de las nuevas arquitecturas [5].

El desarrollo de algoritmos paralelos requiere de construcciones en los lenguajes de programación y los sistemas operativos que permitan expresar paralelismo. Los lenguajes imperativos tradicionales (en particular C y C++) no cuentan con construcciones paralelas nativas. Las interfaces de manejo de hilos de los sistemas operativos permiten la invocación de hilos paralelos, lo que posibilita la implementación de ejecuciones paralelas. Sin embargo, manejar hilos del sistema operativo es una técnica de bajo nivel, que requiere conocimientos específicos del sistema y de la arquitectura, y es difícilmente portable.

El propósito de los modelos de programación paralela es extender los lenguajes existentes, agregándoles expresiones que permitan indicar paralelismo a quien desarrolla un algoritmo. Son interfaces de programación implementadas mediante bibliotecas o extensiones de los compiladores. Los modelos abstraen el hardware y las interfaces de hilos del sistema operativo, simplificando el trabajo de desarrollo de algoritmos paralelos y fomentando la portabilidad de los programas resultantes. Idealmente, estas abstracciones permiten aprovechar el paralelismo disponible sin necesidad de hacer optimizaciones específicas para la plataforma.

Esta sección empieza introduciendo los patrones de programación paralela, que son herramientas conceptuales útiles para diseñar aplicaciones paralelas. Seguido, se describen los principales modelos de programación paralela soportados por Xeon Phi: el estándar OpenMP, conformado por un conjunto de extensiones para los compiladores de C/C++, y la biblioteca Intel TBB para C++. También se comentan otros modelos soportados por Xeon Phi que no fueron estudiados en profundidad. La biblioteca Intel TBB se describe en mayor detalle debido a que estudiar la implementación de aplicaciones paralelas en Xeon Phi con TBB es uno de los propósitos de este trabajo. En particular, se describen las interfaces de TBB usadas para implementar la prueba de concepto descrita en la parte 3.

2.3.1. Patrones de programación paralela

Es frecuente el uso de patrones para resolver problemas recurrentes; un ejemplo son los patrones de diseño de la programación orientada a objetos. Los patrones de programación paralela

son estrategias algoritmicas genéricas (a veces llamadas *esqueletos de algoritmos*) que pueden aplicarse para resolver, de forma paralela, problemas que cumplen ciertas condiciones específicas para cada patrón. Aplicarlos facilita el desarrollo de aplicaciones, dado que son soluciones conocidas que se pueden aplicar frecuentemente, y fomenta la matenibilidad, gracias a que conforman un lenguaje común entre quienes desarrollan aplicaciones paralelas. Muchos modelos de programación paralela brindan implementaciones genéricas de patrones para ser aplicadas a los problemas sin tener que desarrollar el algoritmo genérico [5].

El patrón paralelo más básico es **fork-join**. En este patrón, una tarea invoca a otra tarea para que sea ejecutada de forma paralela; el punto en el cual la tarea es invocada se llama punto de *fork*. Una tarea puede invocar múltiples tareas y, luego, puede esperar a que terminen en el punto de *join*. Básicamente, es una generalización paralela de un conjunto de tareas que, en un programa secuencial, se verían como ejecutadas en orden, y sirve como base para la implementación de los demás patrones [5].

Otro patrón básico es la **anidación** (*nesting*), que es cuando una tarea, que está ejecutando de forma paralela a otras tareas, invoca nuevas tareas paralelas. A pesar de parecer un patrón obvio, en varios modelos de programación paralela no puede aplicarse con total libertad, lo cual dificulta la aplicación de algunos patrones (en particular, aquellos que usan recursión) [5].

Un patrón de los más frecuentes es **map**, que consiste en aplicar exactamente la misma operación en cada uno de los datos de un conjunto de forma paralela; la operación que se ejecuta sobre cada elemento se llama *función elemental*. Para que el patrón sea aplicable, una ejecución de la función elemental no puede tener efectos secundarios sobre las ejecuciones sobre otros elementos. Se puede ver como la paralelización de un caso particular de un bucle en el cual todas las iteraciones son independientes. Map puede generalizarse en otros patrones; por ejemplo, el patrón **stencil**, en el que, dado un conjunto de datos dispuestos geométricamente, el resultado de la función elemental se calcula a partir del elemento actual y un subconjunto de elementos ubicados a desplazamientos fijos respecto al elemento. En la figura 13 se puede ver una representación del patrón stencil. Los bloques verdes representan datos y los azules representan tareas que aplican la funcion elemental. El flujo de datos es de arriba hacia abajo. Cada tarea calcula un resultado a partir de su elemento correspondiente y sus vecinos inmediatos. El conjunto resultado tiene la misma forma que el conjunto de entrada. La práctica de agregar datos nulos para los casos borde se denomina *padding* [5].

Reduce es un patrón que implementa la operación *reducción*, que consiste en combinar todos los elementos de un conjunto en un único elemento mediante una *función combinadora* binaria. La función combinadora define cómo se combinan dos elementos del conjunto o las aplicaciones parciales de la función sobre el conjunto; también puede definir una *identidad*, que corresponde al resultado cuando se aplica sobre un conjunto vacío, y se usa en algunas implementaciones en con-

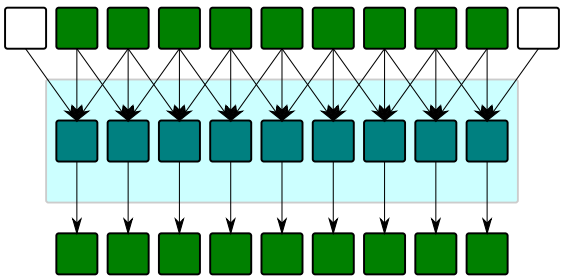


Figura 13: Aplicación del patrón stencil. (Figura basada en imagen extraída de McCool et al. [5].)

diciones borde o como valor inicial. Como se puede observar en la figura 14, en la implementación secuencial de la reducción, empezando por el primer elemento, los elementos se van combinando en orden; en cada paso, se suma el elemento actual al acumulado de los pasos anteriores. Este bucle tiene dependencias entre sus iteraciones, lo que no impide que existan implementaciones paralelas, pero, sin embargo, sí genera que las implementaciones paralelas disponibles requieran que la función combinadora cumpla con ciertas características. Las implementaciones paralelas mostradas en la figura 15, por ejemplo, requieren que la función combinadora sea asociativa (esto es, siendo a, b y c elementos del dominio y la $\oplus$ la función, se cumple $(a \oplus b) \oplus c = a \oplus (b \oplus c)$). La implementación de la izquierda de la figura requiere la mínima cantidad de tareas posibles. Por otro lado, la implementación de la derecha de la figura divide el conjunto en bloques independientes que son reducidos secuencialmente; luego, se reduce el conjunto de resultados de cada bloque para obtener el resultado final. Esta implementación implica una cantidad de tareas mayor, pero igual se usa frecuentemente porque requiere menor comunicación entre las tareas paralelas [5].

Scan es un patrón, similar a reduce, que sirve para calcular las reducciones parciales de un conjunto (es decir, se incluyen todos los valores intermedios de la reducción, no solo el resultado final). Como se puede observar en la figura 16, al igual que con reduce, que haya dependencias entre las iteraciones de su implementación secuencial no implica que no exista una implementación paralela. Sin embargo, en este caso, como contrapartida de permitir escalabilidad, la implementación paralela requiere más trabajo computacional que la implementación secuencial; en el caso de la figura, la implementación secuencial implica 7 tareas, mientras que la paralela implica 11. Existe también una implementación paralela en bloques, representada en la figura 17 [5].

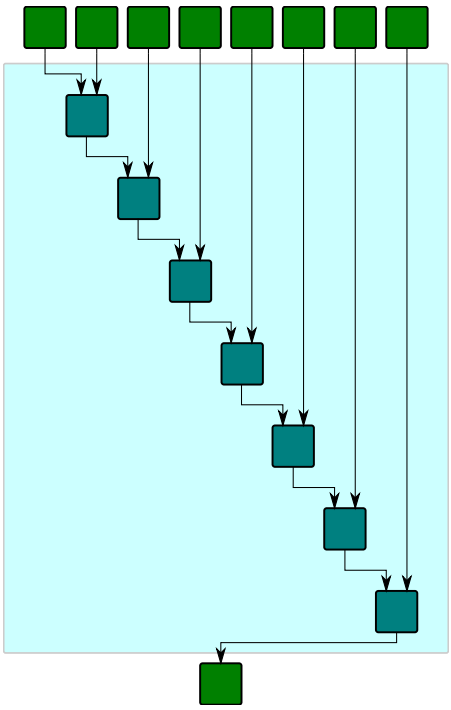


Figura 14: Implementación secuencial de una reducción. (Figura basada en imagen extraída de McCool et al. [5].)

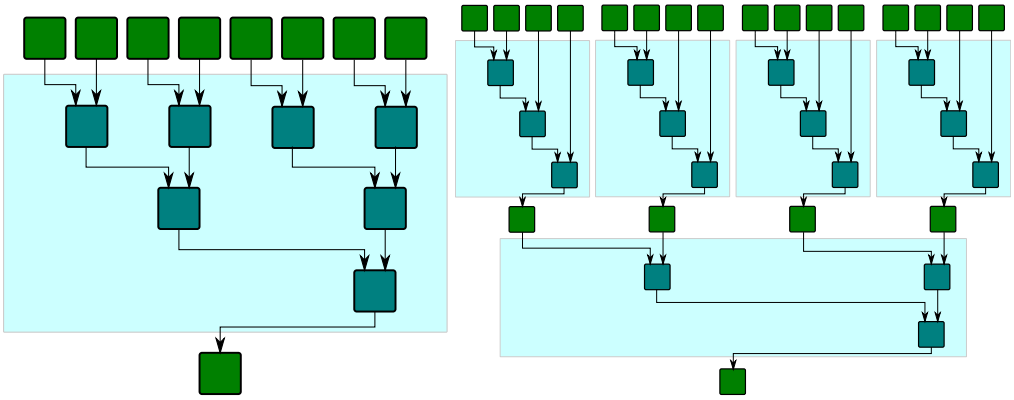


Figura 15: Dos implementaciones paralelas de la reducción. (Figura basada en imágenes extraídas de McCool et al. [5].)

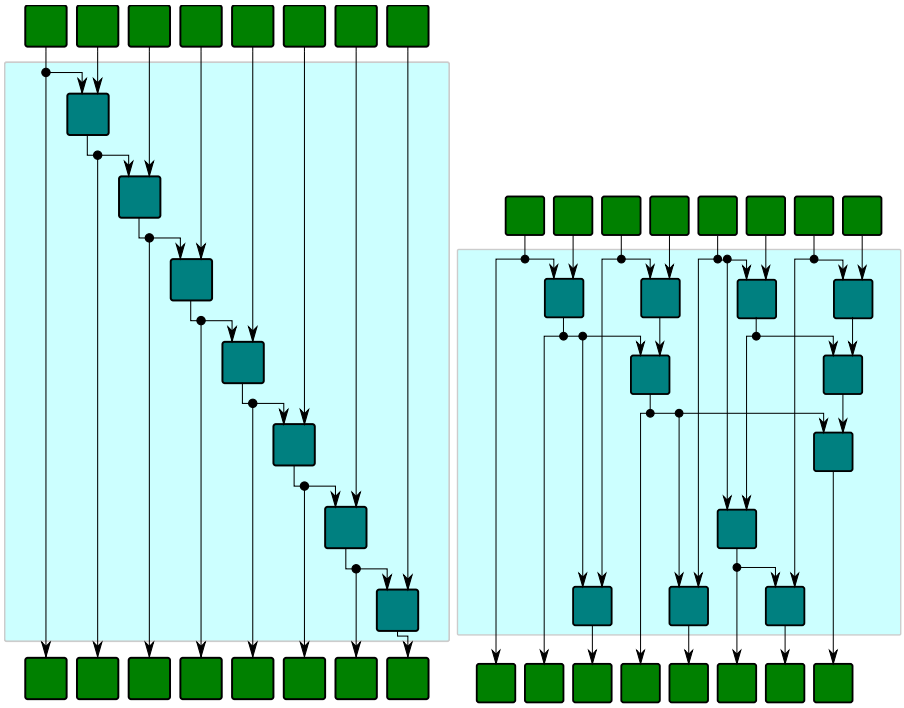


Figura 16: Implementaciones secuencial (izquierda) y paralela (derecha) del patrón scan. (Figura basada en imágenes extraídas de McCool et al. [5].)

Algunos patrones pueden fusionarse con fines de optimización. Por ejemplo, invocaciones sucesivas del patrón map con distintas funciones elementales pueden fusionarse en una sola que combine todas las funciones. De esta forma, se explota mejor la localidad de la memoria. Una ventaja de usar las implementaciones en bloques de reduce y scan es que se pueden fusionar con un map previo, en el caso de reduce, y también con un map posterior, en el caso de scan [5].

Otros patrones paralelos son [5]:

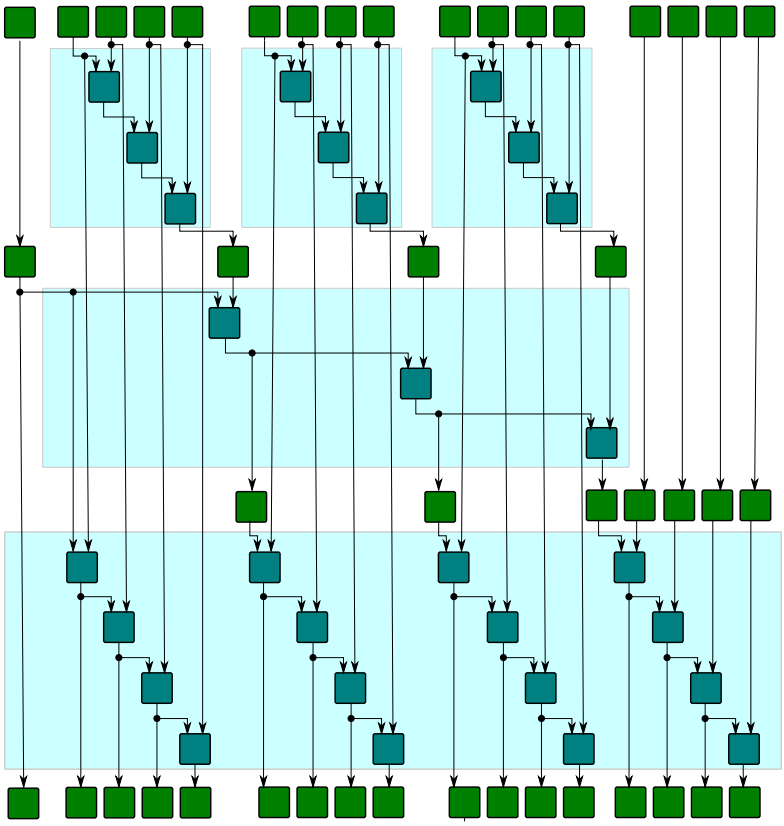


Figura 17: Implementación en bloques de scan paralelo. (Figura basada en imagen extraída de McCool et al. [5].)

- **gather** y **scatter** son patrones relacionados al acceso a memoria; en gather se leen un conjunto de lugares de memoria indizados, y en scatter se escribe un arreglo de elementos en lugares de memoria dispersos.
- **zip**, **unzip**, **shift**, **pack**, **split** y **unsplit** son patrones que se usan para hacer reordenamientos y selecciones de los elementos de un arreglo.
- **pipeline** se usa para procesar datos entrada en varias etapas secuenciales, procesando de forma paralela distintos datos en cada etapa.
- **workpile** es una generalización de map en la que no se conoce de antemano la cantidad de tareas, sino que existe la posibilidad de que una ejecución de la función elemental agregue nuevas ejecuciones a la pila.

2.3.2. Estándar OpenMP

OpenMP es una especificación estandarizada; puede ser implementada por compiladores para el uso de paralelismo de alto nivel en Fortran y C/C++ [50]. Para Xeon Phi, OpenMP es soportado por el compilador oficial de Intel, y se activa al indicar la bandera de compilación `-openmp`. Se basa principalmente en pragmas, que son directivas de precompilación que pueden ser ignoradas si el compilador no las soporta; en caso de que el compilador no soporte OpenMP, las directivas están diseñadas de forma tal que el programa pueda ejecutar secuencialmente con resultados correctos [5][29][30].

En el modelo de ejecución de OpenMP, un hilo principal ejecuta secuencialmente el programa hasta llegar a un bloque de código marcado con un pragma `omp parallel`. En ese momento, se crea un grupo de hilos, los cuales ejecutan el código del bloque marcado, cada uno de forma independiente. La construcción usada más frecuentemente es el pragma `omp parallel for`, que se usa para paralelizar bucles; es una implementación básica del patrón paralelo map. Las iteraciones del bucle son agrupadas en trozos contiguos, los cuales son asignados a los hilos invocados para ejecutar de forma paralela. Para mantener la correctitud del resultado, es necesario que las iteraciones del bucle paralelizado sean independientes entre sí [5][29][30][49]. En el listado 2 se puede ver un ejemplo de uso de este pragma. El pragma `omp parallel for` de la línea 14 indica que las iteraciones del bucle de la línea siguiente van a ser repartidas para ser resueltas por un grupo de `NUM_THREADS` hilos. `private` indica las variables locales a cada hilo; `a` y los arreglos `fa` y `fb` son variables compartidas. `schedule` permite indicar cómo se repartirán las iteraciones entre los hilos; el valor `guided` indica que se repartirá en trozos proporcionalmente al número de hilos, los cuales se asignarán a hilos de forma dinámica.

OpenMP cuenta también con algunas construcciones para casos particulares en los que hay dependencias en las iteraciones; un caso es el pragma `omp parallel reduction`, que es una implementación limitada del patrón reduce [49].

En lugar de crear hilos para cada región paralela, OpenMP maneja un banco de hilos persistente durante la ejecución del programa, a los cuales asigna tareas a ejecutar al llegar a cada región paralela. La cantidad de hilos utilizados para resolver cada región paralela es un parámetro explícito de la región, pudiendo indicarse en tiempo de ejecución. También es configurable la forma en que se asocian los hilos a los núcleos del procesador, pudiendo indicar mediante variables de ambiente si se prefiere esparcir los hilos entre los núcleos u ocupar completamente los hilos de hardware de cada núcleo antes de asignar un hilo a un nuevo núcleo; este concepto es denominado *afinidad*. De una forma parecida, hay controles para indicar cómo se quieren repartir las tareas de una región paralela entre los hilos; esto puede servir para fomentar la localidad en regiones paralelas contiguas que trabajan sobre los mismos datos. Dentro de los pragmas que

```
1  int main(int argc, char *argv[] )
2  {
3    int i, j, k;
4    float a = 1.1;
5    float *fa,*fb;
6    fa = (float*) malloc(FLOPS_ARRAY_SIZE*sizeof(float));
7    fb = (float*) malloc(FLOPS_ARRAY_SIZE*sizeof(float));
8    for(i=0; i<FLOPS_ARRAY_SIZE; i++)
9    {
10     fa[i] = (float)i + 0.1;
11     fb[i] = (float)i + 0.2;
12   }
13
14   #pragma omp parallel for private(j,k) num_threads(NUM_THREADS) schedule(guided)
15   for(k=0; k<FLOPS_ARRAY_SIZE; k++)
16   {
17     //
18     // scale 1st array and add in the 2nd array
19     // example usage - y = mx + b;
20     //
21     for(j=0; j<LOOP_COUNT; j++)
22     {
23       fa[k] = a * fa[k] + fb[k];
24     }
25   }
26
27   // ...
28 }
```

Listado 2: Ejemplo de paralelización de un bucle con OpenMP. El programa está basado en código extraído de Jeffers et al. [30].

indican las regiones paralelas, OpenMP brinda directivas para indicar cuáles variables dentro del bloque de código son compartidas entre los hilos y de cuáles cada hilo guarda su propio valor. Se incluyen construcciones para sincronización entre los hilos, incluyendo barreras, operaciones atómicas y locks [5][49].

Más allá de que está estandarizado, soporta múltiples plataformas y abstrae las interfaces de hilado de bajo nivel, características que simplifican el desarrollo, OpenMP aporta portabilidad de forma limitada. El modelo conceptual en torno a los hilos de ejecución fomenta que quien desarrolla una aplicación paralela tenga presente los hilos de hardware sobre los que se va a ejecutar el programa, lo que es un detalle dependiente de la arquitectura en la que se ejecuta. Este modelado en torno al hilado explícito provoca otra desventaja: el paralelismo anidado solo es soportado de forma limitada, debido a límites que impone OpenMP para evitar la invocación excesiva de hilos de forma recursiva. Esto complica el uso de recursión, que es un recurso usado por muchos algoritmos potencialmente paralelizables [5].

2.3.3. Biblioteca Intel TBB

Intel Threading Building Blocks (TBB) es una biblioteca de programación paralela de C++. Fundamentalmente, está conformada por un conjunto de tipos genéricos que pueden usarse para ejecutar código personalizado de forma paralela. Originalmente surgida para agregar capacidades paralelas a C++, en ese momento inexistentes, incluye implementaciones genéricas de algoritmos paralelos, estructuras de datos concurrentes, funciones de asignación de memoria escalables y directivas de sincronización de hilos. Siendo una biblioteca, puede usarse con cualquier compilador, y es soportada por múltiples plataformas, incluyendo Xeon Phi. Usualmente es usada con la versión C++11 o posterior, debido a que las características de programación funcional agregadas en esa versión de C++ simplifican el uso de las construcciones genéricas de la biblioteca. Es software libre [5][29][30][51].

Al igual que en OpenMP, TBB abstrae las interfaces de hilado del sistema operativo, gestionando internamente su propio banco de hilos. Sin embargo, las interfaces para implementar algoritmos paralelos dispuestas a los usuarios no manejan el concepto de *hilo*, sino que se enfocan en el concepto de *tarea*. Para TBB, una tarea es una porción de trabajo independiente disponible para ejecutar por un hilo cualquiera. Las tareas son creadas en el código y asignadas por TBB a los hilos disponibles. Cada hilo cuenta con una cola de tareas para ejecutar, las cuales va ejecutando en orden. Cuando se queda sin tareas para ejecutar, un hilo puede pasar la última tarea de la cola de otro hilo para su propia cola; por esta característica, el planificador de TBB es clasificado como planificador *work stealing* (robo de trabajo), dado que un hilo que se queda sin trabajo *roba* una tarea de otro hilo [5][30].

En el modelo de ejecución de TBB, se dice que las tareas determinan el *paralelismo potencial*, dado que puede expresarse una cantidad deliberada de tareas independientemente de la capacidad de procesamiento disponible; las tareas que son creadas y no están en procesamiento esperan a ser procesadas en las colas de trabajo de los hilos. Esto aporta portabilidad, dado que no es necesario tener en cuenta la cantidad de hilos sobre la que ejecutarán las tareas a la hora de desarrollar. Esta característica también da soporte al paralelismo anidado, permitiendo que las tareas invoquen nuevas tareas de forma recursiva, sin que haya problemas de sobrecarga de trabajo concurrente. La clasificación de *paralelismo potencial* se contrasta con el *paralelismo obligatorio* implicado por otros modelos, como OpenMP, que trabaja explícitamente con hilos en lugar de tareas [5].

TBB incluye varias interfaces para especificar tareas paralelas. La clase `tbb::task_group` es la forma más básica, pudiendo usarla para indicar porciones arbitrarias de código y generar una tarea que lo ejecute, lo cual sigue el patrón fork-join. Sin embargo, TBB provee implementaciones genéricas de algoritmos paralelos que son útiles para casos frecuentes. La interfaz más simple y

frecuente es la función `tbb::parallel_for`, que es una implementación del patrón paralelo map, y se usa para paralelizar bucles de iteraciones independientes entre sí. `parallel_for` recibe dos parámetros: un rango de tipo `tbb::blocked_range`, que representa los índices del bucle, y una función, la cual debe recibir un parámetro de tipo `blocked_range`. Internamente, TBB se encarga de particionar el rango de los índices y generar una tarea por cada bloque de la partición; la tarea generada ejecuta la función del segundo parámetro (la cual representa la función elemental del patrón map), pasándole como argumento el bloque correspondiente [5][51]. En el listado 3 se puede ver un ejemplo de un bucle secuencial y en el listado 4 su paralelización usando `parallel_for`. El procedimiento `derivar_Nudos` se encarga de derivar todos los nudos de la red y guardar el resultado en el arreglo `nudos_derivados`. En su versión paralela, se invoca a `parallel_for` indicando el rango de los índices del arreglo de nudos (0 a `red->cant_nudos`). La función pasada como segundo parámetro llama a `derivar_Nudo` para los nudos que le toca en función de los índices del bloque que le llega por parámetro.

TBB cuenta con una implementación del patrón reduce: la interfaz `tbb::parallel_reduce`. Sirve para paralelizar código que tiene dependencias entre iteraciones. Nuevamente, `parallel_reduce` recibe un rango de índices como primer parámetro, el cual particiona en bloques. El segundo parámetro es el valor inicial del acumulador. Por cada bloque, TBB crea una tarea

```
1  void derivar_Nudos(NudoDerivado** nudos_derivados, Red* red)
2  {
3    for (int i = 0; i < red->cant_nudos; i++) {
4      nudos_derivados[i] = derivar_Nudo(
5        red->nudos[i],
6        red->barras,
7        red->cant_barras,
8        red->objetos,
9        red->cant_objetos
10     );
11   }
12 }
```

Listado 3: Ejemplo de un bucle secuencial. Este ejemplo está extraído del código de la prueba de concepto de este trabajo.

```
1  void derivar_Nudos(NudoDerivado** nudos_derivados, Red* red)
2  {
3    tbb::parallel_for(
4      tbb::blocked_range<int>(0, red->cant_nudos),
5      [=] (const tbb::blocked_range<int> &rango) {
6        for (int i = rango.begin(); i < rango.end(); i++) {
7          nudos_derivados[i] = derivar_Nudo(
8            red->nudos[i],
9            red->barras,
10           red->cant_barras,
11           red->objetos,
12           red->cant_objetos
13         );
14       }
15     }
16   );
17 }
```

Listado 4: Ejemplo de paralelización del código del listado 3 con TBB usando `parallel_for`.

que ejecuta la función que recibe como tercer parámetro; esta función se encarga de calcular la reducción local del bloque que le toca. Otra función, que es la que recibe como cuarto parámetro, es usada para combinar los resultados de todos los bloques. Para que los resultados sean correctos, `parallel_reduce` requiere que la operacion de reducción sea asociativa; si la operación no lo es (como las operaciones de punto flotante), los resultados pueden ser no deterministas. En caso de que se precise determinismo, se puede usar la alternativa `tbb::parallel_deterministic_reduce`, que tiene menor desempeño [5][51]. Un ejemplo de uso de `parallel_reduce`, usado para paralelizar el código del listado 5, se puede ver en el listado 6. La función `Fuerza_neta` suma todas las fuerzas de un arreglo. Para paralelizarla, se invoca a `parallel_reduce` indicando el rango de los índices del arreglo de fuerzas (0 a `cantidad`). El segundo parámetro es el valor inicial del acumulador: la fuerza nula. La función pasada como tercer parámetro hace la reducción parcial correspondiente al bloque pasado por parámetro, partiendo de un valor inicial. La función pasada como cuarto parámetro solamente suma dos fuerzas y es usada para sumar los resultados parciales de todos los bloques. Se puede notar cómo fueron necesarias operaciones extra de asignación y liberación de memoria en comparación con la versión secuencial.

Una interfaz bastante simple brindada por TBB es `tbb::parallel_invoke`, que permite crear una tarea por cada una de las funciones de una lista de tamaño entre 2 y 10, siguiendo el patrón fork-join. Un ejemplo de su uso puede verse en el listado 7; en la línea 11 del listado, se invoca a `parallel_invoke` indicando una lista de funciones que, al ser independientes, se pueden ejecutar como tareas paralelas. Otras interfaces brindadas por TBB son [5][51]:

- `tbb::parallel_scan`, que implementa el patrón scan.
- `tbb::parallel_do`, que implementa el patrón workpile.
- `tbb::parallel_pipeline`, que implementa el patrón pipeline.

Cuando se usan los algoritmos de TBB que trabajan sobre rangos de tipo `blocked_range`, está la opción de ingresar un parámetro extra a los mencionados, indicando un *partitioner* (particionador), y así poder controlar la forma en la que se particiona el rango. Existen cuatro tipos de particionador [51]:

- `simple_partitioner`, que divide el rango recursivamente hasta llegar a bloques inmediatamente inferiores a la granularidad (*grainsize*) del rango, la cual se puede indicar al crear el rango (por defecto vale 1).

```
1  Fuerza* Fuerza_neta(Fuerza** fuerzas, int cantidad) {
2    Fuerza* neta = nuevo_Vector(
3      (*fuerzas[0])[0], (*fuerzas[0])[1], (*fuerzas[0])[2]
4    );
5    for (i = 1; i < cantidad; i++) {
6      (*neta)[0] += (*fuerzas[i])[0];
7      (*neta)[1] += (*fuerzas[i])[1];
8      (*neta)[2] += (*fuerzas[i])[2];
9    }
10   return neta;
11 }
```

Listado 5: Ejemplo de reducción secuencial. Este ejemplo está extraído del código de la prueba de concepto de este trabajo.

```
Fuerza *Fuerza_neta(Fuerza** fuerzas, int cantidad) {
  return tbb::parallel_reduce(
    tbb::blocked_range<int>(0, cantidad),
    Fuerza_nula(),
    [=] (const tbb::blocked_range<int>&rango, Fuerza* inicial) -> Fuerza* {
      Fuerza * acumulador = nueva_Fuerza(
        (*inicial)[0], (*inicial)[1], (*inicial)[2]
      );
      for (int i = rango.begin(); i < rango.end(); i++) {
        (*acumulador)[0] += (*fuerzas[i])[0];
        (*acumulador)[1] += (*fuerzas[i])[1];
        (*acumulador)[2] += (*fuerzas[i])[2];
      }
      borrar_Vector(inicial);
      return acumulador;
    },
    [=] (Fuerza* una, Fuerza* otra) -> Fuerza* {
      Fuerza* resultado = sumar_Vectores(una, otra);
      borrar_Vector(una);
      borrar_Vector(otra);
      return resultado;
    }
  );
}
```

Listado 6: Ejemplo de paralelización del código del listado 5 con TBB usando `parallel_reduce`.

```
RedDerivada *derivar_Red(Red* red) {
  NudoDerivado** nudos_derivados = MALLOC_ARRAY(
    NudoDerivado*, red->cant_nudos
  );
  BarraDerivada** barras_derivadas = MALLOC_ARRAY(
    BarraDerivada*, red->cant_barras
  );
  ObjetoDerivado** objetos_derivados = MALLOC_ARRAY(
    ObjetoDerivado*, red->cant_objetos
  );
  tbb::parallel_invoke(
    [=] { derivar_Nudos(nudos_derivados, red); },
    [=] { derivar_Barras(barras_derivadas, red); },
    [=] { derivar_Objetos(objetos_derivados, red); }
  );
  return nueva_RedDerivada(
    nudos_derivados,
    barras_derivadas,
    objetos_derivados,
    red->cant_nudos,
    red->cant_barras,
    red->cant_objetos
  );
}
```

Listado 7: Ejemplo de invocación de tareas paralelas con TBB usando `parallel_invoke`. El ejemplo es extraído del código de la prueba de concepto de este trabajo.

- `auto_partitioner`, usado por defecto, que divide el rango recursivamente lo suficiente como para balancear la carga, o hasta llegar a la granularidad.
- `affinity_partitioner`, que es similar al `auto_partitioner` pero conserva la asignación de bloques a hilos para fomentar la localidad de caché.
- `static_partitioner`, que es similar al `affinity_partitioner` pero es determinista.

Las interfaces de asignación de memoria de C/C++ no son escalables: solo un hilo a la vez puede acceder a ellas. Para solucionarlo, TBB implementa su propio *allocator* (asignador de memoria) escalable, `tbb::scallable_allocator`, el cual se puede usar con las colecciones de la STL de C++. TBB también cuenta con una biblioteca *proxy* que, de encontrarse enlazada a un programa, se encarga de redirigir todas las llamadas a las interfaces de memoria de C/C++ (como `malloc` y `free`) a sus equivalentes en `scalable_allocator`. Otro asignador implementado por TBB, que puede ser combinado con `scallable_allocator`, es `tbb::cache_aligned_allocator`, el cual asegura que las asignaciones de memoria están alineadas con las líneas de caché, y no hay asignaciones que compartan datos de la misma línea. El objetivo de esta práctica es evitar casos de *false sharing*, que se dan cuando dos procesadores que no comparten datos sí tienen datos propios dentro de una misma línea de caché, situación que genera movimientos innecesarios de líneas entre las cachés de los procesadores. La contrapartida de usar este tipo de asignación de memoria es que las asignaciones de poco tamaño tienen mayor costo de espacio, el cual es mayormente desperdiciado [5][51].

Por defecto, TBB crea un hilo de ejecución por cada procesador lógico de la plataforma en la que ejecuta, lo que en Xeon Phi equivale a un hilo de TBB por hilo de hardware. Si bien no se recomienda modificar la cantidad de hilos de ejecución, TBB permite establecer un valor diferente si se inicializa un objeto de tipo `tbb::task_scheduler_init` antes de llamar a ninguna otra interfaz de TBB, pasando el número de hilos deseado como parámetro del constructor [61].

2.3.4. Otros modelos soportados

Intel Math Kernel Library (MKL) es una biblioteca que implementa funciones matemáticas para el desarrollo de aplicaciones de computación científica, incluyendo interfaces estándar como BLAS y LAPACK. Es soportada por Xeon Phi, teniendo varios modos de ejecución. Puede usarse directamente llamando a sus funciones, tanto en modo nativo como en modo offload. Un tercer modo de ejecución es el de *offload automático*: las funciones de MKL, cuando son invocadas por un programa cualquiera en un nodo que tiene coprocesadores, son ejecutadas automáticamente por el procesador y los coprocesadores en cooperación. Esta característica solo está habilitada para algunas funciones y solo a partir de ciertos tamaños de datos de los operandos. Por defecto, está desactivada; puede activarse estableciendo el valor de la variable de entorno `MKL_MIC_ENABLE` en 1. La proporción de trabajo que debe repartirse entre el procesador principal y los coprocesadores se puede configurar mediante las variables de entorno `MKL_HOST_WORKDIVISION`, `MKL_MIC_WORKDIVISION` y `MKL_MIC_<N>_WORKDIVISION`, sustituyendo `<N>` por el número de coprocesador que se quiere especificar [29][30].

Message Passing Interface (MPI) es un estándar usado en computación de alto desempeño para desarrollar aplicaciones que ejecutan en sistemas de memoria distribuida. Está definido como un conjunto de funciones que permiten la colaboración entre los procesos del sistema distribuido, a los cuales llama *rangos*. Intel MPI es una implementación que puede usarse en sistemas que tengan nodos con coprocesadores Xeon Phi. La forma más directa de usar Xeon Phi en programas implementados con MPI es ejecutar en modo offload desde un rango que corra en un procesador de un nodo con coprocesadores. Sin embargo, existe también la posibilidad de asignar rangos a los núcleos de los coprocesadores. Este tipo de ejecución heterogénea tiene la complejidad de

tener que balancear la carga de trabajo para adaptarse a las diferencias de desempeño entre los rangos que corren en procesadores y los que corren en coprocesadores; existen soluciones estáticas (repartir cargas fijas de trabajo distintas según el rango) y dinámicas (dividir el trabajo total en tareas autoasignables) para esta situación [30].

2.4. Optimización de desempeño de aplicaciones paralelas

El desempeño de las arquitecturas de computadoras paralelas es objeto de estudio desde antes de la irrupción de los procesadores multinúcleo. El análisis del desempeño paralelo fue evolucionando junto con la adopción y el desarrollo de las computadoras paralelas hasta llegar al día de hoy, en que el uso de arquitecturas masivamente paralelas de memoria compartida, como Xeon Phi, se hizo frecuente. Así como conocer la evolución de las arquitecturas paralelas sirve para entender el estado del arte actual de la computación paralela, conocer la evolución histórica del análisis del desempeño paralelo sirve para entender el desempeño de las arquitecturas actuales y las condiciones para optimizarlo. Este capítulo comienza con una exposición sobre este tema.

Además de los conceptos básicos aplicables a cada arquitectura paralela de memoria compartida, Xeon Phi tiene sus propios aspectos de optimización puntuales. En general, las optimizaciones aplicables a Xeon Phi son aplicables a cualquier arquitectura x86; sin embargo, las características particulares de Xeon Phi hacen que ciertas optimizaciones sean cruciales para obtener un desempeño adecuado. Existen múltiples optimizaciones que son mencionadas frecuentemente en la literatura referida a Xeon Phi. La segunda sección del capítulo se enfoca en enumerar estas optimizaciones.

El desarrollo de la prueba de concepto documentada en la parte 3 implicó varias iteraciones de optimización guiada por perfilación usando VTune Amplifier XE; este proceso está detallado en el apéndice B. Al final de este capítulo, se detallan las métricas de desempeño tenidas en cuenta durante la optimización, que son las usadas para analizar los resultados de las pruebas en la parte 3.

Los ejemplos de código son meramente ilustrativos de los conceptos que representan y no buscan representar problemas realistas.

2.4.1. Análisis de desempeño paralelo

Cuando se quiere evaluar el desempeño de una computadora en la resolución de un problema, habitualmente lo que se quiere es optimizar la latencia de la resolución, es decir, reducir el tiempo que demora la computadora en ejecutar el programa que resuelve el problema. El concepto de *speedup* se usa para referirse a la disminución de la latencia producida por cambios en la forma en que se computa un problema. En particular, cuando se trabaja con computadoras paralelas interesa saber el speedup obtenido al usar cierta cantidad P de unidades de hardware (por ejemplo, núcleos o hilos de procesamiento paralelos), denominadas *workers*, en lugar de una sola:

$$\text{speedup} = S_P = \frac{T_1}{T_P}$$

siendo T_1 la latencia en un worker (es decir, ejecución secuencial) y T_P la latencia en P workers. La relación entre la cantidad de workers y el speedup es denominada *eficiencia*:

$$\text{eficiencia} = \frac{S_P}{P} = \frac{T_1}{PT_P}$$

Frecuentemente, es expresada como porcentaje [5].

La elección de T_1 condiciona la utilidad del speedup y la eficiencia que se calculan usándola como línea base. Si T_1 corresponde a exactamente el mismo algoritmo usado para T_P pero

ejecutado por un solo worker, se dice que el speedup y la eficiencia son *relativos*. Los valores son considerados *absolutos* en el caso de que se use como T_1 a la latencia correspondiente al algoritmo que se desempeñe mejor secuencialmente, el cual no necesariamente escale y, por lo tanto, probablemente no se use para calcular T_P [5].

Cuando la eficiencia es de 100 %, se dice que el algoritmo presenta *speedup lineal*. El speedup lineal puede considerarse un caso ideal de optimización mediante paralelismo, pero se obtiene poco frecuentemente en la práctica. Menos frecuentemente se dan casos de *speedup supralineal*, en general permitidos por condiciones particulares de los problemas (por ejemplo, un algoritmo escalable que trabaja sobre un conjunto de datos que no entra en la caché de un worker pero sí repartido entre las cachés de varios workers) [5].

En general, es normal que el speedup óptimo que se obtenga para un problema corriendo en una computadora sea menor al lineal. La *ley de Amdahl*, formulada en 1967, expone las limitaciones de aumentar el paralelismo para optimizar la latencia de un programa. La ley de Amdahl modela a todos los programas como conformados por dos fracciones: una fracción paralelizable, la cual, en el mejor de los casos, se podrá escalar a la cantidad de workers con los que se cuente, y una fracción secuencial, la cual se ejecutará secuencialmente independientemente de la cantidad de workers. Siendo f_s la fracción secuencial, la latencia obtenida mediante la paralelización con P workers tiene una cota superior [5]:

$$T_P \leq f_s T_1 + (1 - f_s)T_1/P.$$

De igual forma, el speedup también está acotado superiormente:

$$S_P \leq \frac{T_1}{f_s T_1 + (1 - f_s)T_1/P} = \frac{1}{f_s + (1 - f_s)/P}.$$

De esto se desprende que, independientemente de la cantidad de workers usados para optimizar la fracción paralela, el speedup se ve limitado por la fracción secuencial del programa:

$$S_\infty \leq \frac{1}{f_s}.$$

El modelo planteado por la ley de Amdahl se basa en un programa fijo que es escalado a múltiples workers. La *ley de Gustafson-Barsis* es un modelo alternativo, formulado posteriormente, en 1988. Gustafson [52] parte de la observación empírica de que la porción paralelizable de los programas crece linealmente junto con el tamaño del problema, mientras que la porción secuencial se mantiene estática. A partir de esta afirmación, se propone estudiar el speedup a partir de un tiempo de ejecución paralela fijo. Siendo, nuevamente, f_s la fracción secuencial del programa, y f_p la fracción del tiempo de ejecución que se ejecuta de forma paralela, si se asume que la fracción paralela es escalable, el tiempo de ejecución secuencial puede expresarse así:

$$T_1 = f_s T_P + f_p T_P P = f_s T_P + (1 - f_s)T_P P$$

A partir de esto, el speedup paralelo se puede expresar de la siguiente forma:

$$S_P = \frac{f_s T_P + (1 - f_s)T_P P}{T_P}$$

$$= f_s + (1 - f_s)P$$

De esta expresión del speedup paralelo se desprende que, si el tamaño del problema puede aumentar indefinidamente, no hay una cota superior para el speedup que se puede obtener

mediante paralelismo. Esta es una motivación para seguir desarrollando computadoras con cada vez más unidades paralelas. El propósito no es usarlas para resolver los mismos problemas más rápido, sino tener la capacidad de resolver problemas más grandes [5].

La ley de Gustafson-Barsis está relacionada con una característica de la escalabilidad que es importante resaltar: el paralelismo escalable es el paralelismo de datos. La descomposición funcional, es decir, la identificación de varias funciones diferentes que pueden ejecutarse de forma paralela, solo genera mejoras de desempeño puntuales. Los algoritmos que escalan, lo hacen porque son capaces de generar paralelismo acorde con el aumento del tamaño de los datos [5].

Existe un tercer modelo para analizar el desempeño paralelo, que da herramientas más concretas para el análisis de algoritmos paralelos, llamado *modelo work-span*. Se basa en la relación entre dos atributos del algoritmo:

- el trabajo (*work*), que es la latencia del algoritmo cuando se ejecuta con un solo worker;
- y el *span*, que es la latencia correspondiente a cuando se ejecuta en una máquina ideal con infinitos workers.

El modelo se representa gráficamente como en el ejemplo de la figura 18, mostrando las tareas del algoritmo como vértices de un grafo dirigido acíclico, donde las aristas indican las dependencias entre tareas. En esta representación, el trabajo corresponde a la cantidad total de tareas, mientras que el span equivale al camino crítico del grafo [5]. Puntualmente, en el ejemplo de la figura, asumiendo que todas las tareas tienen una latencia igual a 1, el trabajo total es 18 y el span es 6.

Siendo T_1 el trabajo y T_∞ el span de un algoritmo, el speedup paralelo tiene una cota superior determinada por ambas características:

$$S_P = \frac{T_1}{T_P} \leq \frac{T_1}{T_\infty}.$$

Basándose en el modelo work-span, el *lema de Brent* permite determinar una cota superior de la latencia paralela en condiciones ideales. El argumento del lema, en un análisis similar al de la ley

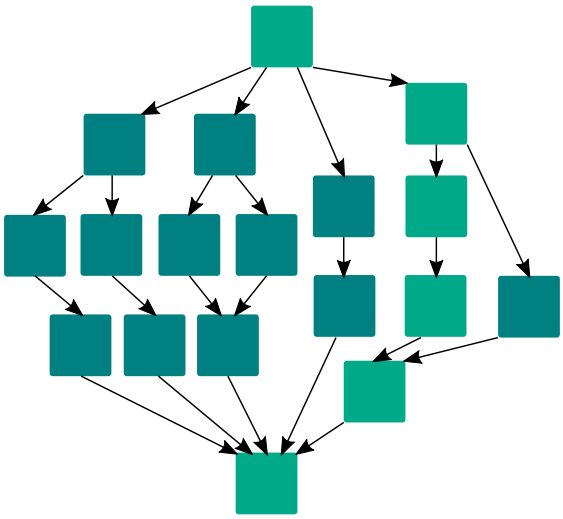


Figura 18: Representación gráfica de un algoritmo mediante el modelo work-span. (Figura basada en imagen extraída de McCool et al. [5].)

de Amdahl, divide el trabajo total T_1 en dos tipos de trabajo: trabajo *perfectamente paralelizable* (es decir, escala a la cantidad de workers disponibles) y trabajo *imperfectamente paralelizable*. El trabajo imperfectamente paralelizable implica una latencia igual al span T_∞, y por lo tanto, el trabajo restante (perfectamente paralelizable) puede implicar una latencia de hasta $T_1 - T_\infty$ con un solo worker en el peor de los casos. Dado que el trabajo perfectamente paralelizable escala a la cantidad de workers, de las observaciones anteriores se desprende la siguiente desigualdad [5]:

$$T_P \leq T_\infty + \frac{T_1 - T_\infty}{P}$$

Partiendo del lema, se pueden observar dos aproximaciones:

$$T_P \approx T_1/P + T_\infty, \text{ si } T_\infty \ll T_1,$$

$$S_P = T_1/T_P \approx P, \text{ si } T_1/T_\infty \gg P.$$

La primera sirve para estimar la latencia cuando el trabajo es mucho mayor que el span. Muestra cómo aumentar el trabajo impacta proporcionalmente la ejecución paralela, y que el span impacta la escalabilidad. De esto se desprende que, al diseñar un algoritmo, es importante enfocarse en reducir el span, el cual es el límite asintótico de la escalabilidad, y que solo conviene aumentar el trabajo si el aumento permite una reducción drástica del span. La segunda aproximación indica que el speedup se aproxima al lineal si hay mucho más paralelismo potencial que del que se dispone del hardware, lo cual se denomina *sobredescomposición*. El paralelismo excesivo se llama *parallel slack* y se define [5]:

$$\text{parallel slack} = \frac{S_\infty}{P} = \frac{T_1}{PT_\infty}.$$

McCool et al. [5] recomiendan apuntar a un valor de parallel slack mayor a 8.

Si bien el modelo work-span es útil para el análisis de algoritmos, es importante tener en cuenta sus restricciones. En primer lugar, asume una asignación del trabajo greedy, en la que a todo worker que termina una tarea se le asigna una tarea inmediatamente siempre que haya trabajo disponible para ejecutar. Esta asunción no siempre se puede tomar. En el caso de TBB, el algoritmo de robo de trabajo que usa como planificador es greedy, aunque no perfectamente. Otra limitación del modelo es que no tiene en cuenta el impacto del acceso a memoria y la comunicación entre workers. El lema de Brent fomenta la sobredescomposición, es decir, la generación de tareas en exceso, superando las capacidades paralelas disponibles; sin embargo, en la práctica es necesario tener en cuenta la sobrecarga de la generación de tareas, dimensionando las tareas para evitar que la sobrecarga impacte escesivamente el trabajo total [5].

Las operaciones de sincronización entre tareas (como lo son los locks y mutexes) son potenciales cuellos de botella que limitan el escalamiento. Por lo tanto, es importante minimizar su uso y medir su impacto antes de usarlas. En general, los patrones de programación paralela evitan usar este tipo de operaciones.

Por fuera del escalado, es fundamental tener en cuenta el impacto del acceso a memoria en el desempeño de los algoritmos. Las memorias caché están construidas en torno a las dos características del principio de localidad: cuando el programa que corre en un procesador accede a un dato, es probable que en un futuro cercano vuelva a acceder al mismo dato (*localidad temporal*) y a datos contiguos (*localidad espacial*). Diseñar algoritmos que tengan en cuenta este principio es clave para obtener un buen desempeño [5].

La *fórmula de Little* es una ecuación que es relevante destacar. Relaciona la concurrencia C con el *throughput* (es decir, la tasa a la que se completan las operaciones) R de operaciones con

latencia L:

$$C = R \cdot L.$$

Esto quiere decir que, para tener un throughput R cuando las operaciones tienen latencia L, se deben estar ejecutando C operaciones simultáneas. Es útil para tener en cuenta en casos en que la latencia es generada por bloqueos en las operaciones, como los accesos de memoria u operaciones de entrada/salida. Lo que se propone es sobredescomponer el problema de forma de generar concurrencia adicional para esconder la latencia, siempre y cuando la concurrencia adicional no genere nuevas penalizaciones que empeoren la latencia [5].

2.4.2. Técnicas de optimización

Diseñar algoritmos escalables es una práctica necesaria para obtener un buen desempeño, tanto en Xeon Phi como en cualquier arquitectura paralela, y los patrones paralelos son técnicas fundamentales para este fin. Por sobre esto, las características específicas de Xeon Phi llevan a que deban cumplirse ciertas condiciones para llegar al desempeño óptimo. Xeon Phi sigue una arquitectura muy particular, con gran cantidad de núcleos, de varios hilos de hardware cada uno, baja frecuencia, un pipeline de ejecución en orden, y unidades vectoriales anchas. Explotar al máximo sus recursos requiere de alta concurrencia, buen aprovechamiento de la memoria caché, y vectorización. Hay un conjunto de técnicas específicas que ayudan a conseguir estas condiciones. Dado que Xeon Phi comparte mucho con los procesadores x86, la mayoría de estas técnicas de optimización tienen un impacto similar al aplicarse en arquitecturas tradicionales [30].

La baja frecuencia de los núcleos de Xeon Phi y su pipeline de ejecución en orden implican que el desempeño secuencial de los núcleos es muy bajo. Obtener un buen desempeño requiere alta concurrencia, por lo que los programas que quieran obtenerlo deben manejar gran cantidad de hilos concurrentes. Cada núcleo tiene 4 hilos de hardware disponibles. No es suficiente con invocar una cantidad de hilos cercana a la cantidad de núcleos: con un criterio relacionado a la fórmula de Little, el próposito de los múltiples hilos de hardware por núcleo es para paliar las penalizaciones provocadas por el pipeline en orden y la latencia de la memoria. Es conveniente ejecutar con varios hilos concurrentes en cada núcleo, de forma de que el tiempo ocioso generado por un bloqueo en un hilo pueda aprovecharse con la ejecución de otro hilo del mismo núcleo. De hecho, debido al comportamiento del planificador del núcleo, que no puede ejecutar instrucciones de un mismo hilo en ciclos de reloj contiguos, usar un solo hilo por núcleo solo puede obtener hasta un 50 % del desempeño. Si bien algunos programas funcionan óptimamente con 2 hilos por núcleo, lo usual es que sean necesarios 3 ó 4 hilos por núcleo para optimizar; de esto se desprende que un programa que quiera correr con buen desempeño en Xeon Phi debe tener la posibilidad de escalar a alrededor de 240 hilos concurrentes [28][30].

El tamaño de los operandos de las unidades vectoriales de Xeon Phi, 512 bits, es particularmente grande, y permite hacer 16 operaciones de elementos de 32 bits (por ejemplo, números de punto flotante de precisión simple) y 8 de 64 bits (por ejemplo, números de punto flotante de precisión doble). La capacidad de cómputo de Xeon Phi se debe en gran parte a sus unidades vectoriales, y no usarlas es desaprovechar oportunidades de desempeño. Existen varias formas de vectorizar. La más simple y portable es usando la biblioteca matemática MKL, que ya viene optimizada para usar vectorización [29][30].

Los compiladores que realizan *autovectorización*, en particular el compilador oficial de C/C++ de Intel, son capaces de identificar oportunidades de vectorización en bucles. Para que el compilador lo vectorice automáticamente, un bucle debe cumplir varias condiciones. Principalmente, para ser vectorizado automáticamente, debe ser un bucle de tipo `for`, sin bucles anidados, cuya cantidad de iteraciones se pueda determinar al momento de entrar al bucle. Iteraciones contiguas

del bucle deben acceder a lugares contiguos de memoria y todas las iteraciones deben ser independientes entre sí. En los listados 8 y 9 se pueden ver ejemplos de un bucle que no es candidato a autovectorización y un bucle que sí, respectivamente. El bucle del listado 8 no es vectorizable debido a que el `for` interior, en la línea 13, recorre la matriz en el sentido de las columnas, mientras que son los datos de cada fila los que son contiguos en memoria. Por su parte, el bucle del listado 9 sí es vectorizable, dado que el `for` interior recorre una fila de la matriz, cuyos datos son contiguos. El compilador verifica que estas condiciones se cumplan y luego aplica heurísticas para determinar si corresponde o no vectorizar, según si las heurísticas le sugieren que generará mejor desempeño. En el caso del compilador de Intel, se puede observar si se está vectorizando o no un bucle, y las razones por qué, a través de los reportes de vectorización generados por el compilador, para lo cual es necesario indicar la opción de compilación `-vec-report3` [28][30].

```
1  #define N 10000
2  #define M 1000
3  double** multiplicar_Matriz_por_escalar(double a, double** matriz) {
4    int i;
5    int j;
6    double** resultado = malloc(N * sizeof(double*));
7    for (i = 0; i < tamanio; i++) {
8      resultado[i] = malloc(M * sizeof(double));
9      for (j = 0; j < M; j++) {
10       resultado[i][j] = 0.0;
11     }
12   }
13   for (j = 0; j < M; j++) {
14     for (i = 0; i < N; i++) {
15       resultado[i][j] += a * matriz[i][j];
16     }
17   }
18   return resultado;
19 }
```

Listado 8: Multiplicación de una matriz por un escalar que no es vectorizable.

```
1  #define N 10000
2  #define M 1000
3  double** multiplicar_Matriz_por_escalar(double a, double** matriz) {
4    int i;
5    int j;
6    double** resultado = malloc(N * sizeof(double*));
7    for (i = 0; i < N; i++) {
8      resultado[i] = malloc(M * sizeof(double));
9      for (j = 0; j < M; j++) {
10       resultado[i][j] = 0.0;
11     }
12   }
13   for (i = 0; i < N; i++) {
14     for (j = 0; j < M; j++) {
15       resultado[i][j] += a * matriz[i][j];
16     }
17   }
18   return resultado;
19 }
```

Listado 9: Multiplicación de una matriz por un escalar que sí es vectorizable.

El listado 10 muestra un extracto de un reporte de vectorización. En el reporte se indica que el bucle de la línea 38 del archivo `src/derivada/Fuerza.cpp` pudo vectorizarse. El compilador separó algunas de las iteraciones iniciales y finales del bucle (*peló* el bucle) y vectorizó la parte central. Las partes iniciales y finales del bucle no pudieron vectorizarse. El compilador puede haberse visto forzado a hacer esta modificación del bucle debido a que no puede asegurar que los datos estén alineados con las líneas de caché, ni que la cantidad de elementos sea múltiplo del tamaño de las unidades vectoriales.

El compilador no siempre tiene la capacidad de determinar si se cumplen todas las condiciones para la vectorización. La posibilidad de que los operandos de una función sean punteros aliases puede impedir determinar con seguridad si las iteraciones de un bucle son independientes. También, en algunos casos, la posibilidad de que los datos no estén alineados con las líneas de caché afecta negativamente las heurísticas del compilador para decidir cuándo vectorizar. Para solucionar estas situaciones, el compilador incluye pragmas específicos que se pueden aplicar encima de los bucles para darle pistas sobre vectorización. Puntualmente, el pragma `ivdep` s1rve para indicar que no hay dependencias entre las iteraciones del bucle. El pragma `vector always`

```
LOOP BEGIN at src/derivada/Fuerza.cpp(38,44) inlined into /opt/intel/
    compilers_and_libraries_2017.8.262/linux/tbb/include/tbb/parallel_reduce.h
    (192,22)
<Peeled loop for vectorization>
   remark #15335: peel loop was not vectorized: vectorization possible but seems
       inefficient. Use vector always directive or -vec-threshold0 to override
   remark #25456: Number of Array Refs Scalar Replaced In Loop: 3
   remark #25015: Estimate of max trip count of loop=7
LOOP END

LOOP BEGIN at src/derivada/Fuerza.cpp(38,44) inlined into /opt/intel/
    compilers_and_libraries_2017.8.262/linux/tbb/include/tbb/parallel_reduce.h
    (192,22)
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15462: unmasked indexed (or gather) loads: 3
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 24
   remark #15477: vector cost: 17.750
   remark #15478: estimated potential speedup: 1.240
   remark #15488: --- end vector cost summary ---
   remark #25018: Total number of lines prefetched=2
   remark #25021: Number of initial-value prefetches=4
   remark #25035: Number of pointer data prefetches=2, dist=8
   remark #25141: Using second-level distance 4 for prefetching pointer data
       reference   [ src/derivada/Fuerza.cpp(40,36) ]
LOOP END

LOOP BEGIN at src/derivada/Fuerza.cpp(38,44) inlined into /opt/intel/
    compilers_and_libraries_2017.8.262/linux/tbb/include/tbb/parallel_reduce.h
    (192,22)
<Remainder loop for vectorization>
   remark #15335: remainder loop was not vectorized: vectorization possible but
       seems inefficient. Use vector always directive or -vec-threshold0 to
       override
   remark #25456: Number of Array Refs Scalar Replaced In Loop: 3
LOOP END
```

Listado 10: Extracto de un reporte de vectorización.

le indica al compilador que ignore los resultados de sus heurísticas y opte por la vectorización. El pragma `vector aligned` indica que los datos están alineados con las líneas de caché. Por último, si los otros pragmas no funcionan, el pragma `simd`, incluido en OpenMP, sirve para forzar al compilador para que vectorice. En el momento en que se usa uno de estos pragmas, queda en manos de quien los usa asegurar que se cumplen las condiciones indicadas; de no cumplirse, se puede comprometer la correctitud de los resultados [28][30].

Son frecuentes los casos en que la vectorización guiada por el compilador no funciona o no tiene buen desempeño [39][40]. En estos casos, se puede recurrir a las operaciones intrínsecas de vectorización y a las clases de vectores de C++, disponibles en los archivos `immintrin.h` y `micvec.h`, respectivamente. Estas operaciones son de bajo nivel y específicas de la arquitectura, por lo que usarlas implica sacrificar la portabilidad del programa [28][30].

El uso de estructuras de datos compuestas complica la vectorización. Los tipos de datos compuestos guardan todos sus atributos de forma contigua en memoria. Cuando se opera de forma paralela sobre el mismo atributo de muchos datos que siguen un mismo tipo compuesto, no existe posibilidad de vectorizar las operaciones, dado que los atributos de los elementos a los que se está accediendo no son inmediatamente contiguos en memoria. Para vectorizar en este tipo de situaciones, puede ser necesario modificar las estructuras de datos que se usan. La transformación de arreglos de estructuras a estructuras de arreglos (*array-of-structures to structure-of-arrays, AoS to SoA*) puede ayudar a habilitar la vectorización [5][30]. Un ejemplo de este tipo de transformaciones se ve en los listados 11 y 12. La función `desplazar` implementa un caso en el que un grupo de objetos, cada uno en su posición, quieren ser desplazados cada uno en un desplazamiento distinto. En el listado 11 los valores de las tres coordenadas de cada objeto se guardan contiguamente. Al operar en una misma dimensión sobre todos los elementos, no hay posibilidad de vectorizar, debido a que los operandos no están contiguos en memoria. El listado 12 muestra una transformación del código a estructura de arreglos. En este caso, operar sobre una misma coordenada en todos los objetos implica acceder a lugares de memoria contiguos, lo que permite la vectorización.

Como se mencionó anteriormente, asegurar que los datos estén alineados con las líneas de caché (es decir, que las estructuras de memoria asignadas empiecen en lugares de memoria de dirección múltiplo del tamaño de línea de caché L1) ayuda a mejorar el desempeño de la

```
1  typedef struct Vector {
2    double x;
3    double y;
4    double z;
5  } Vector;
6
7  void desplazar(
8    Vector* objs,
9    Vector* desps,
10   int cant
11 ) {
12   int i;
13   for (i = 0; i < cant; i++) {
14     objs[i].x += desps[i].x;
15     objs[i].y += desps[i].y;
16     objs[i].z += desps[i].z;
17   }
18 }
```

Listado 11: Ejemplo de uso de arreglo de estructuras.

```
typedef struct Vectores {
  double* x;
  double* y;
  double* z;
} Vector;

void desplazar(
  Vectores objs,
  Vectores desps,
  int cant
) {
  int i;
  for (i = 0; i < cant; i++) {
    objs.x[i] += desps.x[i];
    objs.y[i] += desps.y[i];
    objs.z[i] += desps.z[i];
  }
}
```

Listado 12: Ejemplo de transformación del código del listado 11 a estructura de arreglos.

vectorización y, a veces, es determinante para el compilador al momento de decidir si vectorizar o no. Asignar memoria de forma alineada no solo mejora la vectorización, sino que evita que se den casos de *false sharing*, que se da cuando dos núcleos distintos usan datos distintos pero dentro de una misma línea de caché. Debe aplicarse con consideración, debido a que genera que se asigne más memoria de la que se usa. Las operaciones que asignan memoria de forma alineada son de bajo nivel, y el factor de alineación es, por definición, una característica propia de cada arquitectura; en el caso de Xeon Phi, las líneas de caché L1 son de 64 bytes, y el compilador de Intel soporta las directivas `__attribute(align(FACTOR))` y `__declspec(align(FACTOR))`. En general, asignar memoria de forma alineada rompe la portabilidad de la aplicación. Sin embargo, entre los asignadores de memoria de TBB está incluido `tbb::cache_aligned_allocator`, el cual transparenta la asignación de memoria y permite asignar memoria alineada de forma portable [5][28][30][51].

Minimizar la cantidad de accesos a la memoria RAM y aprovechar al máximo la caché son criterios fundamentales a seguir al momento de optimizar el desempeño de un algoritmo. En aplicaciones que trabajan con conjuntos de datos muy grandes, es imposible mantener todos los datos en caché. Para minimizar la cantidad de lecturas, es importante maximizar la cantidad de veces que se usa un dato que es traído a caché antes de que su línea sea reemplazada por otra. El principio de localidad es una clave para conseguirlo [5][28][30]. Una técnica básica aplicable al momento de iterar sobre un conjunto de datos es la *fusión* de bucles. Bucles consecutivos que trabajan sobre los mismos datos pueden fusionarse para evitar duplicar los accesos a memoria. En los listados 13 y 14 se puede ver un ejemplo de fusión entre dos bucles. La función `sumar_Vector_a_Vectores` del listado 13 no tiene un comportamiento de caché óptimo, debido a que se recorre el arreglo `v` en dos bucles separados. La alternativa del listado 14 fusiona los dos bucles, lo que permite reutilizar las lecturas del arreglo `v` y aprovechar mejor la caché. La cantidad de bucles fusionables es arbitraria, pero se debe tener en cuenta el espacio que ocupan en caché los datos que agrega cada bucle además de los reutilizados [5][28].

Una práctica más general es llamada *cache tiling* o *cache blocking*. Consiste en subdividir los conjuntos de datos en subconjuntos (*bloques* o *tiles*) que entren en caché simultáneamente y procesar cada bloque completamente antes de pasar al siguiente bloque. Frecuentemente, el tamaño de los bloques está determinado por un *factor de bloqueo* fijo, que depende del tamaño

```
1  void sumar_Vector_a_Vectores(
2    double* v,
3    double* acc1,
4    double* acc2,
5    int tamanio
6  ) {
7    int i;
8    for (i = 0; i < tamanio; i++) {
9      acc1[i] += v[i];
10   }
11   for (i = 0; i < tamanio; i++) {
12     acc2[i] += v[i];
13   }
14 }
```

Listado 13: Función que suma un mismo vector a dos vectores, con comportamiento de caché subóptimo.

```
1  void sumar_Vector_a_Vectores(
2    double* v,
3    double* acc1,
4    double* acc2,
5    int tamanio
6  ) {
7    int i;
8    for (i = 0; i < tamanio; i++) {
9      acc1[i] += v[i];
10     acc2[i] += v[i];
11   }
12 }
```

Listado 14: Función equivalente a la del listado 13 con mejor comportamiento de caché.

del tipo de dato que se use y el tamaño de la caché (o, más precisamente, el tamaño de las líneas de caché), por lo que su aplicación es dependiente de la plataforma y no es portable. Los algoritmos *cache oblivious* se caracterizan por subdividir los datos de forma recursiva, lo que les sirve para adaptarse a distintos tamaños de caché [5][28][30]. Un ejemplo de la aplicación de cache tiling se ve en los listados 15 y 16. La función `sumar_Vector_a_Matriz` suma un mismo vector a todas las filas de una matriz. La implementación del listado 15 recorre todo el vector para cada fila de la matriz. Esto puede ser ineficiente si la cantidad de columnas es mucho mayor que la cantidad de filas. Para este caso, la implementación del listado 16, que aplica cache tiling con un factor de bloqueo fijo, tiene un mejor aprovechamiento de la caché, dado que lee el vector de a bloques y suma los elementos del bloque a todas las filas antes de pasar al siguiente bloque.

Por defecto, escribir en un lugar de memoria en un programa implica una lectura para traer la línea de memoria a la caché, una escritura en la línea de caché, y luego una posterior escritura en la memoria cuando la línea de caché es reemplazada. Para casos en que solamente se quiere hacer una escritura en memoria, sin necesidad de usar los datos posteriormente, se pueden usar *streaming stores*, que son operaciones de escritura que escriben directamente en memoria sin pasar por la caché. Para indicar que una escritura es una streaming store se puede usar el pragma `vector nontemporal` del compilador de Intel [30].

Además de páginas de 4KB y 64KB, Xeon Phi soporta usar páginas particularmente grandes de 2MB, lo cual puede ser útil configurar si los misses de TLB son un problema [30].

```
1  void sumar_Vector_a_Matriz(
2    double* vector,
3    double** matriz,
4    int n,
5    int m
6  ) {
7    int i;
8    int j;
9    for (j = 0; j < n; j++) {
10     for (i = 0; i < m; i++) {
11       matriz[j][i] = vector[i];
12     }
13   }
14 }
```

Listado 15: Función que suma un vector a todas las filas de una matriz.

```
1  #define FACTOR 64
2  void sumar_Vector_a_Matriz(
3    double* vector,
4    double** matriz,
5    int n,
6    int m
7  ) {
8    int i;
9    int j;
10   int k;
11   for(k = 0; k < m; k += FACTOR) {
12     for (j = 0; j < n; j++) {
13       for (
14         i = k;
15         (i < k + FACTOR) && (i < m);
16         i++
17       ) {
18         matriz[i][j] = vector[j];
19       }
20     }
21   }
22 }
```

Listado 16: Aplicación de cache tiling con factor de bloqueo fijo a la función del listado 15.

2.4.3. Métricas de desempeño

En la optimización guiada por perfilación, las decisiones tomadas en el proceso de mejorar el desempeño de una aplicación se basan en los reportes generados por una herramienta de software, llamada perfilador, que aportan información sobre la ejecución de la aplicación. A partir de los reportes generados, se detectan cuáles son las secciones del código de la aplicación que ocupan la mayor parte del tiempo de ejecución; estas secciones son llamadas *hotspots*, y son potenciales puntos de optimización. Un proceso de optimización guiada por perfilación posible es el siguiente [37][39]:

1. Efectuar un análisis de la aplicación con un perfilador y seleccionar un hotspot.
2. Evaluar la información recabada para el hotspot seleccionado, en busca de ineficiencias optimizables.

3. Si se observan ineficiencias, realizar las optimizaciones pertinentes para mejorarlas.

4. Repetir desde el paso 1 hasta cubrir todos los hotspots o llegar al desempeño deseado.

El perfilador Intel VTune Amplifier XE puede usarse para analizar aplicaciones que corren en Xeon Phi. Cuando se usa para analizar una aplicación, VTune se encarga de monitorear la ejecución de la aplicación en busca de ciertas condiciones del estado del hardware, denominadas eventos. VTune lleva la cuenta de la cantidad de veces que ocurre cada evento y registra en qué secciones del código se dio. A partir de los registros de eventos de un análisis, VTune elabora reportes consistentes de conjuntos de métricas de desempeño que permiten detectar posibles problemas de desempeño, como pueden ser falta de paralelismo, desbalances de carga, desperdicio de la memoria caché u operaciones de larga latencia [35][39]. En esta sección se describen algunas de las métricas relevantes reportadas por VTune.

Varias de las métricas que reporta VTune refieren a tiempos de ejecución. La métrica **Elapsed Time** refiere a la latencia de la ejecución, es decir, el tiempo transcurrido desde que empezó a ejecutar la aplicación hasta que terminó. Una métrica similar, **CPU Time**, indica la acumulación de los tiempos de ejecución de la aplicación en todos los hilos de hardware. VTune tiene la capacidad de diferenciar la porción de CPU Time que se usó para ejecutar código de la aplicación de la porción extra generada por la tecnología de paralelismo usada por la aplicación. La porción correspondiente al código de la aplicación está en la métrica **Effective Time**. La porción restante se divide en dos métricas: por un lado, **Overhead Time** registra el trabajo implicado por la gestión de las tareas paralelas, mientras que, por el otro, **Spin Time** registra el tiempo gastado por hilos esperando en puntos de sincronización. En sus reportes, VTune muestra una advertencia si alguna de estas últimas dos métricas supera un umbral predefinido; sin embargo, la documentación de VTune indica que estos umbrales fueron determinados experimentalmente por arquitectos de Intel, sin otra justificación. En general, solucionar desbalances de carga y favorecer algoritmos que reduzcan la sincronización ayudan a reducir el Spin Time; el Overhead Time puede reducirse aumentando la granularidad de las tareas [35].

Para aplicar conceptos del modelo work-span durante la optimización guiada por perfilación, se pueden buscar equivalentes de los conceptos del modelo dentro de las métricas de tiempos de ejecución reportadas. El trabajo total (T_1) puede medirse con el valor de la métrica Elapsed Time de una ejecución secuencial, mientras que el valor de la métrica Elapsed Time para una ejecución usando P hilos equivale al valor del tiempo de ejecución con P workers (T_P). VTune desglosa la métrica Effective Time según la cantidad de paralelismo presente en cada instante. Divide el valor en cuatro niveles, de menor a mayor paralelismo: **Idle**, **Poor**, **Ok** y **Ideal**. Los umbrales que determinan cada nivel corresponden a la cantidad de hilos de hardware ejecutando simultáneamente y son configurables al momento de consultar el reporte de un análisis en la interfaz de VTune. Dado que el span (T_∞) corresponde al trabajo que no es perfectamente paralelizable, se puede configurar a uno de los niveles mencionados como el nivel deseado de paralelismo y aproximar el valor de T_∞ sumando los valores correspondientes a los niveles inferiores. En el caso de las pruebas de la sección 3.3, se puso el umbral entre los niveles Poor y Ok en 114 hilos (equivalente a dos hilos por núcleo), y se estimó T_∞ como la suma de los valores de los niveles Idle y Poor. Para aproximar la fracción del trabajo total correspondiente al span ($\%_{T_\infty}$), se calcula el porcentaje representado por T_∞ respecto a T_1.

Las métricas de CPI (*clocks per instruction*, ciclos de reloj por instrucción) sirven para visualizar la latencia promedio de las instrucciones. Se pueden medir por núcleo o por hilo de hardware:

- CPI por hilo es la cantidad de instrucciones ejecutadas por el hilo sobre la cantidad de ciclos transcurridos en el núcleo.

- CPI por núcleo es la suma de todas las instrucciones de todos los hilos del núcleo sobre la cantidad de ciclos transcurridos.

Dado que cada núcleo de la arquitectura puede iniciar hasta dos instrucciones por ciclo pero no puede iniciar instrucciones de un mismo hilo en dos ciclos consecutivos, el mínimo teórico de CPI por núcleo obtenible es 0.5, requiriendo haya más de un hilo ejecutando en el núcleo. El mínimo teórico de CPI por hilo es 1 ejecutando con dos hilos por núcleo, 1.5 para tres hilos y 2 para cuatro hilos. Se consideran no deseables valores mayores a 1 de CPI por núcleo y mayores a 4 de CPI por hilo, pero algunas aplicaciones que hacen uso intensivo de la memoria pueden correr óptimamente sin llegar a esos valores. Es importante notar que los CPI por núcleo pueden mejorar mientras los CPI por hilo empeoran; por ejemplo: aumentar la cantidad de hilos por núcleo con los que ejecuta una aplicación puede generar un aumento en el valor de CPI por hilo, pero a la vez puede presentar un mejor desempeño general y disminuir el valor de CPI por núcleo. En particular, las métricas de CPI sirven para comparar dos versiones distintas del código, aunque hay casos en que valores menores no indican mayor eficiencia (por ejemplo, usar vectorización mejora el desempeño pero aumenta los CPI debido a que las instrucciones vectoriales son más lentas pero hacen más trabajo). En general, cualquier optimización afecta el valor de los CPI. En el caso de VTune para Xeon Phi, la métrica **CPI Rate** devuelve el CPI promedio entre todos los hilos de hardware; el CPI por núcleo puede calcularse dividiendo el valor de CPI Rate entre la cantidad hilos por núcleo usados [35][37].

Aprovechar la jerarquía de memoria implica que haya muchas lecturas desde la caché por cada línea que se trae de un nivel más bajo de la jerarquía. En el caso de L1, el umbral que se traza es que el cociente entre la cantidad de operaciones en ese nivel y la cantidad de operaciones aritméticas sea por lo menos igual a la intensidad de vectorización (a grandes rasgos, que haya por lo menos una operación vectorial por operación de lectura o escritura en cache); la métrica con la que VTune reporta este cociente se llama **L1 Compute to Data Access Ratio**, y las métricas sobre vectorización se comentan más adelante en esta sección. En el caso de L2, se espera que su relación con la cantidad de operaciones aritméticas sea por lo menos 100 veces mayor que el valor correspondiente a L1; la métrica que reporta esta relación es **L2 Compute to Data Access Ratio**. Para mejorar el indicador de L1, se puede reducir la cantidad de instrucciones del camino crítico, minimizar los bucles internos (removerles todo lo que no sea necesario como inicialización y condiciones), alinear la memoria, optimizar la vectorización y minimizar el overhead del manejo de tareas. Para L2, se puede probar mejorar la localidad de los datos, para lo que existen múltiples técnicas, algunas descritas en la sección 2.4.2 [37].

Es importante tener buen aprovechamiento de la localidad para evitar que el acceso a memoria domine el tiempo de ejecución. Un acceso a L1 ahorra alrededor de 20 ciclos respecto a un acceso en L2, y un acceso en L2 ahorra por lo menos 250 ciclos respecto a un acceso a memoria o a la cache de otro núcleo, por lo que es útil mantener la localidad en ambos niveles. VTune reporta la tasa de hits de L1 en la métrica **Hit Rate**; se recomienda que sea mayor a 95 %. La tasa de hits L2 no se puede calcular, la métrica **Estimated Latency Impact** calcula la cantidad de ciclos promedio que toma cada miss de L1; se recomienda que sea menor a 145, que es el promedio entre la latencia de L2 y la del acceso a memoria, por lo que un valor menor indicaría que la mayoría de los misses de L1 son servidos por L2. No todas las optimizaciones de un valor afectan el otro (por ejemplo, mejorar el Hit Rate de L1 puede que no tenga impacto en L2). Para optimizar estos valores, se sugiere usar las técnicas tradicionales para mejorar la localidad (tiling, prefetch por software, alineación de los datos y streaming stores). Eventualmente, pueden darse problemas con la asociatividad de caché si se accede a datos con saltos de a 4KB en L1 o 64KB en L2, lo que se puede solucionar usando padding [37].

VTune reporta métricas sobre tasas de miss de TLB; **L1 TLB Miss Ratio** y **L2 TLB Miss Ratio** indican las tasas de miss de TLB L1 y L2, respectivamente. Cada miss de TLB L2

consume por lo menos 100 ciclos y se considera importante evitar misses en este nivel; el umbral recomendado es que el valor de L2 TLB Miss Ratio sea menor al 0.1 %. Los misses de L1 que son hits en L2 se consideran menos graves: un miss de TLB L1 servido por L2 tiene una latencia de 25 ciclos con páginas de 4KB y 8 ciclos con páginas de 8MB; esta tasa se reporta en la métrica **L1 TLB Misses Per L2 TLB Miss**. Si hay muchos misses de L1 servidos por L2 con páginas de 4KB, entonces la aplicación podría beneficiarse de páginas más grandes. En general, mejorar la localidad mejora el uso del TLB, ya que el TLB no es más que otra caché [37].

Idealmente, se debe apuntar a que la intensidad de vectorización (la cantidad de elementos de cada vector que se usan en las operaciones vectoriales) sea igual a la cantidad de elementos que soporta el vector (8 números de punto flotante de precisión doble ó 16 de precisión simple); el valor se reporta en la métrica **Vectorization Intensity**. Se debe tener cuidado de no leer esta métrica para extensiones grandes de código, sino revisarla en las porciones específicas que se quieren analizar, ya que muchas instrucciones pueden afectar la métrica. Para mejorar esta métrica, se recomienda aplicar las técnicas de vectorización mencionadas en la sección 2.4.2 [37].

El ancho de banda de memoria óptimo (es decir, la tasa esperada con la que se lee de y se escribe en la memoria principal) de una aplicación depende del tamaño del conjunto de datos que se usa. VTune reporta el valor del ancho de banda de memoria promedio en la métrica **Average DRAM Bandwidth**. El valor de esta métrica es aproximado, debido a que no se tienen en cuenta las escrituras *streaming stores* y sí se incluyen transferencias entre cachés de distintos núcleos. En las aplicaciones que trabajan sobre pocos datos, en particular si entran completamente en la caché L2, no es esperable que se explote el ancho de banda disponible en el coprocesador. Por otro lado, en el caso de aplicaciones que trabajan sobre cantidades considerablemente más grandes de datos, es esperable explotar un ancho de banda mayor a 80GB/s, con un máximo práctico de 140GB/s. Una aplicación que presenta un ancho de banda menor a 80GB/s puede deberse a una falta de aprovechamiento de la localidad espacial [37].

3. Desarrollo de prueba de concepto

En esta parte del documento se presenta el caso de estudio abordado como prueba de concepto de los conocimientos recopilados en la parte 2. El caso de estudio consiste de la implementación escalable de una aplicación de computación científica real en Xeon Phi usando TBB. En los siguientes capítulos, luego de una descripción del problema resuelto por la aplicación seleccionada y de los detalles de su implementación, se exponen las pruebas realizadas con la aplicación y sus resultados.

Todas las pruebas realizadas en los siguientes capítulos se corrieron sobre el sistema descrito en el apéndice A.

3.1. Problema propuesto: Simulación de red de pesca

En este capítulo se describe la aplicación seleccionada para la prueba de concepto, que consiste de la simulación de una red de pesca. La sección 3.1.1 expone conceptualmente el problema. En la sección 3.1.2 se describe el modelo propuesto para la resolución del problema y en la sección 3.1.3 se detalla el análisis realizado para la implementación de la aplicación que lo resuelve.

3.1.1. Descripción del problema

La simulación de redes de pesca es un problema de la ciencia pesquera solucionado mediante conocimientos de la mecánica de los fluidos computacional. Una simulación de este tipo consiste de calcular el comportamiento de una red al ser usada en un escenario dado. En particular, se calcula la evolución de la forma geómetrica de la red y las fuerzas a las que están sujetos los elementos que la componen durante el transcurso de un intervalo de tiempo. Estas simulaciones son usadas en la industria del equipamiento pesquero para evaluar el desempeño de distintas configuraciones de redes, variando la forma de la red y su despliegue. También permiten evaluar el desempeño de una misma red en distintas condiciones ambientales [53][54][59].

El sistema NaLA (*net-shape and load analysis*, análisis de forma de red y carga), presentado originalmente por Takagi et al. [60], es un modelo para la simulación de redes de pesca. En NaLA, una red de pesca se considera como conformada por tres tipos de componentes: nudos, barras y objetos. Los nudos son las intersecciones entre las cuerdas que conforman la malla de la red; una barra es la porción de cuerda que une a dos nudos; los objetos son elementos externos adheridos a la red, como pueden ser cuerpos del entorno que se enredan en ella. Para calcular el comportamiento de la red, en cada instante de tiempo se calculan las fuerzas ejercidas sobre cada componente; por ejemplo, para un nudo, las fuerzas consisten de las tensiones de las barras, las interacciones con el ambiente (por ejemplo: la fuerza ejercida por la corriente o la flotabilidad) y las interacciones con los objetos cercanos. A partir del cálculo de la fuerza neta sobre cada componente, se obtiene una ecuación diferencial que, de ser resuelta, indica el movimiento del componente y, por lo tanto, el comportamiento de la red [53][54][59]. Varios trabajos validan que NaLA es útil para simular la aplicación de varios tipos de redes de pesca en situaciones reales; puntualmente, Suzuki et al. [53] y Shimizu et al. [54] obtuvieron resultados simulados consistentes con la realidad para redes de tipo cerco danés y trasmallo, respectivamente.

La aplicación elegida como prueba de concepto es un programa, originalmente implementado en MATLAB, que simula redes de pesca cónicas con mallas rómbicas mediante un modelo basado en NaLA y resuelve el sistema de ecuaciones diferenciales correspondiente mediante el método Runge-Kutta de cuarto orden. El trabajo de desarrollo realizado consistió en reimplementar el programa en C/C++ usando TBB como implementación del paralelismo.

3.1.2. Modelo matemático de la solución

La simulación de la red de pesca en la que se basó la prueba de concepto fue descrita originalmente por Suzuki et al. [53] y Shimizu et al [54]. En las publicaciones correspondientes, se presenta el sistema NaLA, que modela a la red de pesca como un conjunto de masas puntuales unidas mediante resortes sin masa, localizando esas masas en el centro de cada nudo y objeto enredado y en el medio de cada barra que une dos nudos. Los nudos y objetos son modelados ambos como esferas, y las barras son modeladas como cilindros con base circular.

La ecuación de la posición de una masa puntual para el punto i de la red es:

$$\sum_{j \in L(i)} [T_{ij}^g + \frac{1}{2}(C_{ij} \cdot F_{ij}^l + W_{ij}^g + B_{ij}^g)]$$

siendo:

- $L(i)$ el conjunto de puntos enlazados con el punto i
- T_{ij} la tensión actuando entre los puntos i y j
- F_{ij} la fuerza de arrastre de la barra entre los puntos i y j
- W_{ij} el peso de la barra
- B_{ij} la flotabilidad de la barra
- los superíndices l y g indicadores de que los vectores se expresan dentro del sistema de coordenadas locales o globales, respectivamente, convertidos mediante la matriz de transformación C_{ij}, determinada por el algoritmo de transformación de inercia descrito por Korte et al [55][56].

El algoritmo que implementa el modelo divide a la red de pesca en tres tipos disjuntos de componentes: nudos, barras (el elemento que une dos nudos) y objetos (elementos enredados). La posición de un componente de la red en un instante de tiempo se puede expresar como una ecuación diferencial de su aceleración, y, por lo tanto, conociendo su masa, de la fuerza neta ejercida sobre el mismo:

$$x_i(t) = x_i(0) + \int_0^t v_i(t)dt = x_i(0) + v_i(0) + \int_0^t a_i(t)dt = x_i(0) + v_i(0) + m_i \int_0^t F_i(t)dt \quad (1)$$

siendo $v_i(t)$, $a_i(t)$ y $F_i(t)$ la velocidad, la aceleración y fuerza neta sobre el componente i para un instante dado, respectivamente, y m_i la masa del componente. En resumen, sabiendo la posición y velocidad iniciales de cada componente de la red y su masa, se puede expresar su posición en el tiempo en función de la integral de la fuerza neta ejercida sobre él. Si se encuentra una expresión para la fuerza neta en cada instante de tiempo, se puede calcular la posición del componente en cada instante.

La fuerza neta ejercida sobre cada componente puede efectivamente aproximarse en función del estado general de los componentes de la red: se puede modelar a partir de las interacciones del componente con los demás componentes de la red y el ambiente (el flujo de agua y la gravedad). Puntualmente según el tipo de elemento se identifican las fuerzas de interacción:

- Los nudos interactúan con las barras que los unen con otros nudos y con los objetos con los que tienen contacto.

- Las barras interactúan con los nudos de sus extremos y con los objetos con los que tienen contacto.
- Los objetos interactúan con los componentes con los que tienen contacto.
- Todos los componentes tienen peso, flotabilidad y arrastre generado por la corriente.

Estos datos determinan, entonces, una función para calcular la fuerza neta sobre cada componente en un instante dado. A partir de la ecuación (1), se puede expresar siguiente sistema de ecuaciones diferenciales:

$$\begin{cases} x_i'(t) = m_i * F_i(t,\, x_i(t)) \\ x_i(t_0) = x_i(0) \end{cases} \tag{2}$$

Este sistema puede resolverse mediante un método Runge–Kutta [53][54].

Puntualmente, para resolver el problema se eligió el método Runge–Kutta clásico, también conocido como el método Runge–Kutta de cuarto orden. El método consiste de, partiendo de una posición inicial, calcular la posición de la red en cada instante de tiempo a partir de la posición en la que se encontraba en un instante anterior. Para hacer esto, dado un instante de tiempo t_n y eligiendo un intervalo de tiempo h lo más corto posible, se calculan cuatro valores denominados incrementos:

$$\begin{cases} k_1 = h * F(t_n,\, x_i(t_n)), \\ k_2 = h * F(t_n + \frac{h}{2},\, x_i(t_n) + \frac{k_1}{2}), \\ k_3 = h * F(t_n + \frac{h}{2},\, x_i(t_n) + \frac{k_2}{2}), \\ k_4 = h * F(t_n + h,\, x_i(t_n) + k_3) \end{cases} \tag{3}$$

Luego, estos valores se combinan para calcular la posición (aproximada) en el instante $t_{n+1} = t_n + h$:

$$x_i(t_{n+1}) = x_i(t_n) + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{4}$$

De esta forma, partiendo desde una posición inicial conocida, se puede calcular el movimiento de la red de pesca en un intervalo de tiempo al dividirlo en instantes consecutivos de distancia h y calculando la posición correspondiente para cada instante.

3.1.3. Análisis de implementación

El diseño de la implementación de la solución se pensó con tres objetivos:

- *Conservación del diseño original:* Se desea que el diseño resultante sea lo más parecido posible al diseño del programa original, para evitar retrabajo conceptual.
- *Escalabilidad:* La plataforma Xeon Phi requiere que el programa sea escalable a cientos de hilos paralelos para obtener un buen rendimiento.
- *Mantenibilidad:* Gran parte del trabajo a realizar con el programa nuevo consiste en modificarlo para analizar el impacto en el rendimiento de distintas variaciones de su implementación en la plataforma Xeon Phi. Es necesario que el programa sea fácilmente entendible y modificable para poder hacer esto de forma efectiva.

Estos objetivos fueron los que determinaron las principales decisiones de diseño.

La primera gran decisión, en pos de conservar el diseño original, fue dividir el programa en dos módulos: por un lado, el cálculo de la función derivada (representada en el modelo por la ecuación (2), equivalente a la función `funcion_derivada` del programa original en MATLAB) y por otro el algoritmo Runge–Kutta para la resolución de la ecuación diferencial a partir de la

función derivada (representado por la función `Runge_kutta_fourth_order` del programa original). A continuación, se describe el diseño de los dos módulos.

El propósito principal del módulo `derivada` es calcular la aceleración de cada componente (nudo, barra u objeto) de la red en un instante de tiempo a partir de las condiciones de la red (el estado de todos los componentes y las fuerzas del ambiente). Este cálculo se realiza calculando todas las fuerzas que ejercen los demás componentes y el ambiente sobre cada componente y sumándolas para calcular la fuerza neta resultante.

Rompiendo parcialmente con el diseño original, pero para favorecer la mantenibilidad del programa, se realizó un análisis orientado a objetos del problema. El primer resultado es el modelo de dominio de la figura 19. Como indica la figura, se identificó una clase por cada tipo de componente (denominada `Nudo`, `Barra` y `Objeto`, respectivamente), las cuales guardan la posición y la velocidad de cada componente y tienen operaciones para calcular las fuerzas de interacción entre sus instancias (por ejemplo, para calcular la fuerza ejercida por una barra sobre un nudo atado a ella) y para calcular el valor de su derivada. Además, se identificó una clase para representar a la red completa, la cual tiene la capacidad de calcular su propia derivada; lo que significa esto es que calcula la derivada de cada uno de sus componentes, como se puede ver en el diagrama de secuencia de la figura 20. Para cada componente, calcular su derivada significa obtener todas las fuerzas que actúan sobre él y, a partir de la fuerza neta resultante, calcular su aceleración. Por ejemplo, el diagrama de secuencia de la figura 21 representa el método que calcula la derivada de un objeto. Se puede observar cómo se obtiene la fuerza ejercida sobre el objeto por todos los demás componentes (nudos, barras y otros objetos), las cuales, sumadas a las fuerzas generadas por el ambiente, determinan su fuerza neta, y a partir de esta fuerza neta se calcula, finalmente, la aceleración.

El módulo `rk4` implementa el algoritmo Runge–Kutta de cuarto orden para resolver la ecuación diferencial del problema. La clase que implementa el algoritmo se llama `Paso` y es utilizada para calcular la simulación completa por la clase `Simulacion`, la cual es externa al módulo. El comportamiento completo de este procedimiento se puede ver en el diagrama de secuencia de la figura 22. Cada `Paso` guarda la informacion sobre el estado de la red en un instante y el tamaño de paso del algoritmo (es decir, la distancia arbitraria entre dos instantes de tiempo de la simulación). A partir de estos dos datos, calcula el valor de los incrementos de la ecuación (3) (obteniendo los valores correspondientes de la función derivada a partir del módulo `derivada`) y

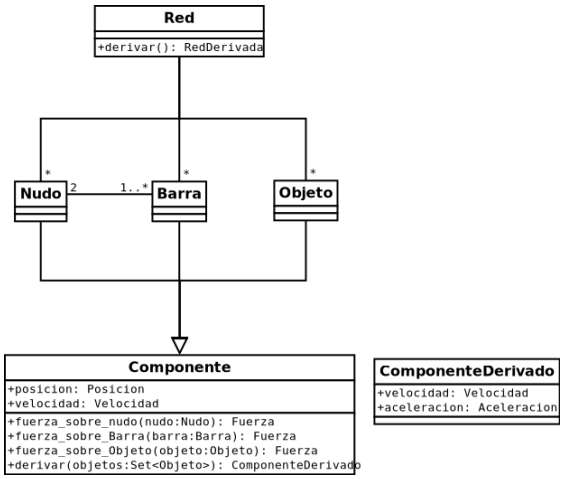


Figura 19: Modelo de dominio del módulo `derivada`.

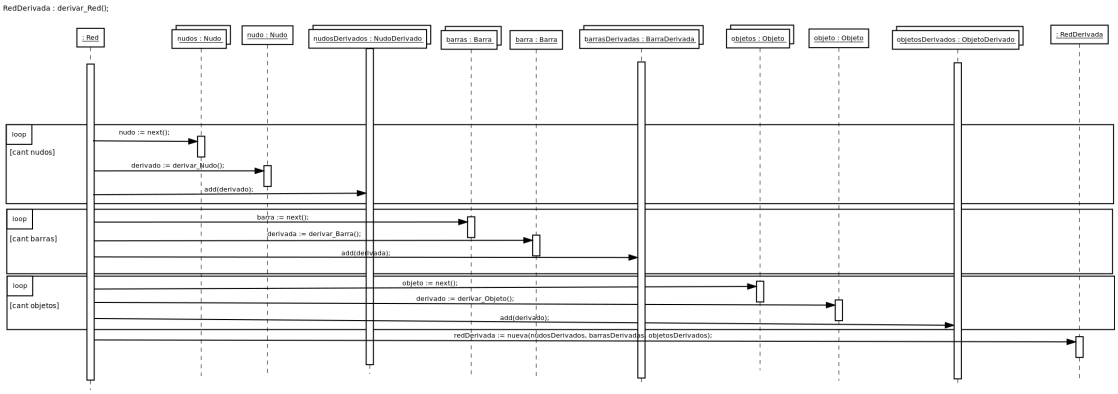


Figura 20: Diagrama de secuencia de la función `derivar_Red`.

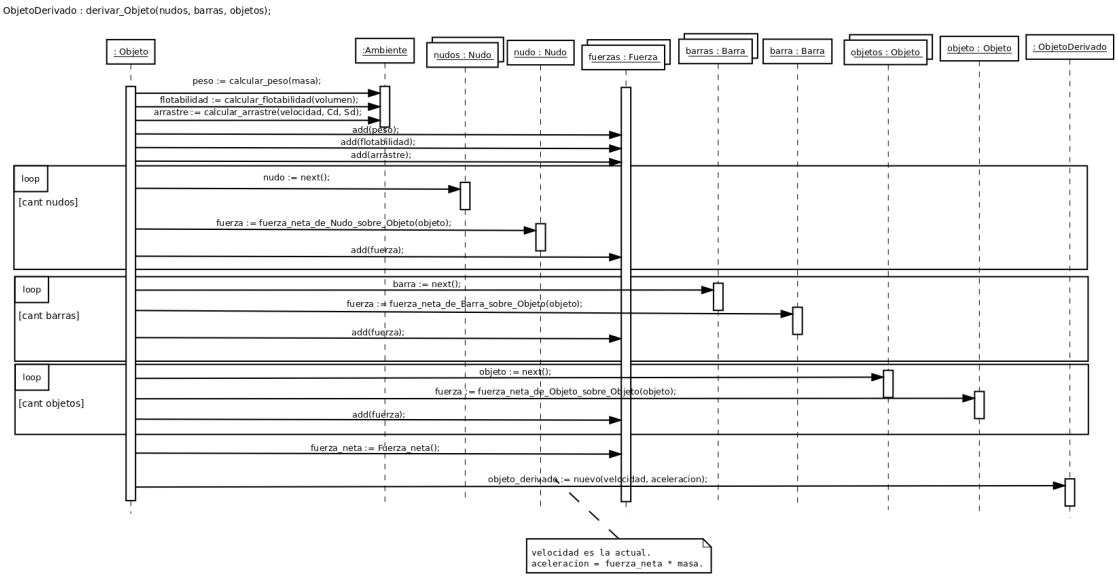


Figura 21: Diagrama de secuencia de la función `derivar_Objeto`.

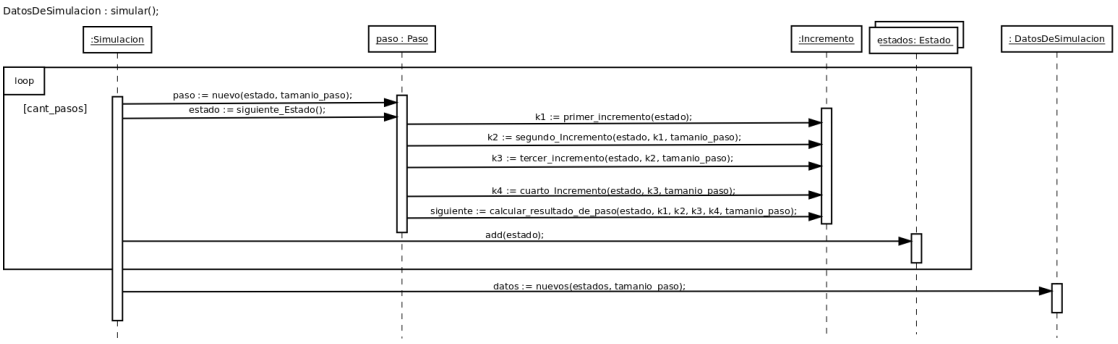


Figura 22: Diagrama de secuencia de la función `simular`.

a partir de los incrementos calcula el estado de la red en el instante siguiente mediante la ecuación (4). Una `Simulacion` se encarga de, a partir de unas condiciones iniciales arbitrarias y dados un tamaño de paso y una cantidad de pasos a simular, como lo implica su nombre, ejecutar la simulación (es decir, obtener la forma de la red en cada instante del intervalo de tiempo a simular). Esto lo hace, partiendo de un paso inicial con las condiciones iniciales, retroalimentando el resultado de cada paso a las condiciones iniciales del paso siguiente, repitiendo el procedimiento una cantidad de veces igual a la cantidad de pasos deseada. Una vez terminado el procedimiento, se cuenta con todos los datos deseados de la simulación.

Un aspecto de la interfaz entre el módulo `rk4` y el módulo `derivada` fue específicamente replicado para procurar mantener el diseño original: el módulo `derivada` cuenta con su propia representación interna del estado de la red, la cual consiste de las clases que representan los componentes y su modelo de datos; por otro lado, el módulo `rk4` es agnóstico de este modelo, y representa a la red con un único arreglo unidimensional que contiene la posición y la velocidad de cada uno de los componentes. La conversión entre ambos modelos es realizada por una función externa a ambos módulos denominada `funcion_derivada`, la cual es utilizada por `rk4` para calcular el valor de la derivada a partir de las condiciones de la red.

Una característica de Xeon Phi es su soporte de los lenguajes y modelos de programación tradicionales de los sistemas de memoria compartida. Entre los lenguajes disponibles, se decidió usar C para la implementación de la versión inicial del problema, dado que es un lenguaje tradicional en el que el equipo tiene experiencia y es usado en múltiples cursos de la carrera de Ingeniería en Computación.

C no es un lenguaje orientado a objetos, por lo que las clases identificadas en la etapa de diseño debieron adaptarse a estructuras permitidas por el lenguaje. A grandes rasgos, lo que se hizo fue definir un tipo de estructura de datos `struct` por cada clase identificada, asignando un miembro dentro de la estructura a cada atributo de la clase. En cuanto a los métodos, por cada método de una clase se definió una función que recibe como parámetro un puntero a la estructura de datos que representa el objeto sobre el cual se estaría invocando el método, además de los parámetros indicados por el diseño original. De esta forma, se puede trabajar con una semántica similar a la de un lenguaje orientado a objetos, sin contar completamente con otras características como el encapsulamiento o la herencia pero simulando los beneficios que brindan.

Además de las clases del dominio del problema de la red de pesca, se implementaron tipos de datos más básicos representando vectores de las magnitudes implicadas en el algoritmo (por ejemplo: `Posicion`, `Velocidad`, `Fuerza`, etc.) y las operaciones básicas en torno a estos tipos (adición, multiplicación por escalar, producto vectorial, etc.). Se tomó la decisión de implementar estos tipos (se podrían haber obtenido de alguna biblioteca) debido a que sus operaciones son potenciales puntos de afinación del rendimiento en Xeon Phi.

Sumado al código del programa que implementa la solución del problema, se elaboró una configuración bastante completa para poder realizar pruebas unitarias sobre las funciones que se implementaron, la cual cuenta con facilidades para establecer valores de retorno y efectos secundarios esperados de las funciones que se prueban. Esto sirvió para que el programa cuente con un cubrimiento casi completo, el cual es útil al momento de modificar el programa para verificar que no se afecte su comportamiento involuntariamente. También se realizó un test de aceptación que ejecuta una simulación completa y compara su resultado con el resultado del programa original.

3.2. Paralelización en Xeon Phi

En este capítulo se comenta el trabajo realizado con el objetivo de paralelizar y optimizar el desempeño del programa simulador de red de pesca en Xeon Phi, lo que se hizo usando Intel TBB

como modelo de programación paralela. Se detallan los cambios realizados al código secuencial del programa para hacerlo paralelo y luego se comentan los resultados de pruebas preliminares realizadas para constatar que se efectivizó la paralelización.

Este capítulo se enfoca en declarar las modificaciones realizadas. Para más información sobre el proceso iterativo de optimización asistida por perfilación que llevó a la determinación de estas modificaciones, se puede consultar el apéndice B.

3.2.1. Modificaciones realizadas

El programa secuencial que se utilizó como punto de partida para paralelizar fue una implementación en C de la simulación de red de pesca descrita en el capítulo 3.1, siguiendo las decisiones de diseño descritas en la sección 3.1.3. Luego de las adaptaciones necesarias para que el código sea compatible con C++, los cambios aplicados consistieron principalmente de emplear las plantillas de función de TBB para indicar las regiones paralelas.

La plantilla `parallel_invoke` se usa en varios puntos del código para indicar que un conjunto de funciones distintas pueden ejecutarse de forma independiente. Uno de los puntos se puede ver en la línea 11 del listado 17. *Derivar* la red implica *derivar* cada uno de sus componentes, es decir, calcular la velocidad y la aceleración a la que se encuentra cada componente. Dado que la derivada de cada componente se puede calcular independientemente de los demás componentes, las funciones `derivar_Nudos`, `derivar_Barras` y `derivar_Objetos` se pueden ejecutar de forma paralela. Los otros lugares donde se aplicó `parallel_invoke` es dentro del cálculo de la derivada de cada componente de la red, como se puede ver en la línea 15 del listado 18 para el caso de los objetos. Cada tipo de fuerza ejercida sobre un componente se puede calcular de forma independiente, por lo que se puede invocar una tarea distinta por cada tipo de fuerza. En el caso del listado 18, se crea una tarea para calcular el peso del objeto, otra para calcular su

```
1  RedDerivada *derivar_Red(Red* red) {
2    NudoDerivado** nudos_derivados = MALLOC_ARRAY(
3      NudoDerivado*, red->cant_nudos
4    );
5    BarraDerivada** barras_derivadas = MALLOC_ARRAY(
6      BarraDerivada*, red->cant_barras
7    );
8    ObjetoDerivado** objetos_derivados = MALLOC_ARRAY(
9      ObjetoDerivado*, red->cant_objetos
10   );
11   tbb::parallel_invoke(
12     [=] { derivar_Nudos(nudos_derivados, red); },
13     [=] { derivar_Barras(barras_derivadas, red); },
14     [=] { derivar_Objetos(objetos_derivados, red); }
15   );
16   return nueva_RedDerivada(
17     nudos_derivados,
18     barras_derivadas,
19     objetos_derivados,
20     red->cant_nudos,
21     red->cant_barras,
22     red->cant_objetos
23   );
24 }
```

Listado 17: Código de la función `derivar_Red` de la versión paralela del simulador de red de pesca.

```
ObjetoDerivado *derivar_Objeto(
        Objeto * objeto,
        Nudo ** nudos,
        int cant_nudos,
        Barra ** barras,
        int cant_barras,
        Objeto ** objetos,
        int cant_objetos
) {
        int i = 0;
        int cant_fuerzas = cant_nudos + cant_barras + cant_objetos + 3;
        Fuerza **fuerzas = MALLOC_ARRAY(Fuerza *, cant_fuerzas);
        Fuerza *neta;
        ObjetoDerivado *derivado;
        tbb::parallel_invoke(
                [=] { fuerzas[0] = calcular_peso(masa); },
                [=] { fuerzas[1] = calcular_flotabilidad(VOLUMEN); },
                [=] { fuerzas[2] = calcular_arrastre(objeto->velocidad, Cd, Sd); },
                [=] {
                        calcular_fuerzas_netas_de_Nudos_sobre_Objeto(
                                objeto, nudos, cant_nudos, &(fuerzas[3])
                        );
                },
                [=] {
                        calcular_fuerzas_netas_de_Barras_sobre_Objeto(
                                objeto,
                                barras,
                                cant_barras,
                                &(fuerzas[cant_nudos + 3])
                        );
                },
                [=] {
                        calcular_fuerzas_netas_de_Objetos_sobre_Objeto(
                                objeto,
                                objetos,
                                cant_objetos,
                                &(fuerzas[cant_nudos + cant_barras + 3])
                        );
                }
        );
        neta = Fuerza_neta(fuerzas, cant_fuerzas);
        derivado = nuevo_ObjetoDerivado(
                objeto->velocidad,
                calcular_Aceleracion(neta, calcular_inercia(masa, vol, Cm))
        );
        borrar_Vectores(fuerzas, cant_fuerzas);
        borrar_Vector(neta);
        return derivado;
}
```

Listado 18: Código de la función `derivar_Objeto` de la versión paralela del simulador de red de pesca.

flotabilidad, otra para su arrastre, y otra por cada tipo de componente con el que interactua (una para los nudos, una para las barras y una para los demás objetos). Los cálculos de las derivadas de los nudos y las barras son análogos al cálculo de las derivadas de los objetos y aplican `parallel_invoke` con el mismo criterio.

`parallel_for` fue usado en casos en que se aplica un mismo procedimiento a los elementos de un arreglo. Los cálculos de la derivada de los distintos tipos de componentes de la red fueron tres casos en los que se aplicó. El listado 19 muestra el ejemplo del cálculo de la derivada de los nudos de la red. En la línea 3, `parallel_for` se usa para indicar que se creen tareas paralelas dividiendo los índices del arreglo en bloques y aplicando la función pasada como parámetro a cada bloque, cuya función es calcular la derivada de los nudos correspondientes al bloque indicado para la tarea. En el listado 20 se puede observar cómo, de manera similar, el cálculo de la fuerzas de interacción de un componente con los demás componentes también se paraleliza mediante

```
1  static void derivar_Nudos(NudoDerivado ** nudos_derivados, Red * red)
2  {
3          tbb::parallel_for(
4                  tbb::blocked_range <int>(
5                          0, red->cant_nudos, FISHNET_CHUNKSIZE_DERIVAR_NUDOS
6                  ),
7                  [=] (const tbb::blocked_range<int> &rango) {
8                          for (int i = rango.begin(); i < rango.end(); i++) {
9                                  nudos_derivados[i] = derivar_Nudo(
10                                         red->nudos[i],
11                                         red->barras,
12                                         red->cant_barras,
13                                         red->objetos,
14                                         red->cant_objetos);
15                         }
16                 }
17         );
18 }
```

Listado 19: Código de la función `derivar_Nudos` de la versión paralela del simulador de red de pesca.

```
1  static void calcular_fuerzas_netas_de_Barras_sobre_Nudo(
2          Nudo * nudo, Barra ** barras, int cant_barras, Fuerza ** fuerzas
3  ) {
4          int i;
5          tbb::parallel_for(
6                  tbb::blocked_range<int>(
7                          0, cant_barras, FISHNET_CHUNKSIZE_BARRAS_SOBRE_NUDO
8                  ),
9                  [=] (const tbb::blocked_range<int> &rango) {
10                         for (int i = rango.begin(); i < rango.end(); i++) {
11                           fuerzas[i] = fuerza_neta_de_Barra_sobre_Nudo(barras[i],
                                 nudo);
12                         }
13                 }
14         );
15 }
```

Listado 20: Código de la función `calcular_fuerzas_netas_de_Barras_sobre_Nudo` de la versión paralela del simulador de red de pesca.

`parallel_for`; el caso de la figura 20 corresponde al cálculo de las fuerzas ejercidas por todas las barras sobre un nudo.

La implementación realizada del algoritmo Runge–Kutta de cuarto orden trabaja con una representación del estado de la red que consiste de un único arreglo largo con elementos numéricos. Las operaciones que se realizan con esta representación también se paralelizaron usando `parallel_for`. Por ejemplo, el listado 21 muestra la implementación de la suma de dos de estos arreglos. Se puede observar que esta operación es candidata para autovectorización, debido a que trabaja con datos contiguos en memoria.

Las operaciones de transformación entre los dos formatos de representación de la red (la estructura `Red` y el arreglo de números) también se paralelizó mediante `parallel_for`, dado que los datos correspondientes a distintos componentes se pueden transformar de forma independiente.

`parallel_reduce` se usa en un solo punto del código, que es en el cálculo de la fuerza neta a partir de un arreglo de fuerzas, operación que es realizada en el cálculo de la derivada de todos los componentes. El listado 22 muestra como se implementó la función `Fuerza_neta`, que realiza la operación comentada. La primera función pasada como parámetro se encarga de sumar las fuerzas del bloque de fuerzas correspondiente a la tarea, y la segunda función se encarga de sumar los resultados de dos bloques y liberar la memoria adicional reservada por esos resultados intermedios.

Por último, un cambio importante para permitir la escalabilidad de la aplicación fue enlazar la librería proxy que sustituye las operaciones de manejo de memoria de forma transparente por alternativas escalables. Para hacer este cambio, se agregaron las banderas de enlace `-ltbbmalloc` y `-ltbbmalloc_proxy`.

3.2.2. Resultados preliminares

Para evaluar preliminarmente la escalabilidad de la aplicación desarrollada, se corrieron pruebas comparando el tiempo transcurrido de ejecución de la versión secuencial implementada en C en comparación con el de la última versión paralelizada. Los datos de prueba ingresados como entrada fueron los correspondientes a una red de pesca cónica con mallas rómbicas de 1.00m de largo y 0.20m de diámetro, que está conformada por 4650 nudos, 9300 barras y 10 objetos. Los datos se generaron usando un script de MATLAB incluido con el programa original. Se realizó una simulación de 10 pasos con tamaño de paso 0.0001s. Dado que no se recomienda cambiar el valor por defecto de la cantidad de hilos invocados por TBB [61], las pruebas se ejecutaron con

```
1  static Numero *sumar_datos(Numero * unos, Numero * otros, int tamanio)
2  {
3          Numero *nuevos = MALLOC_ARRAY(Numero, tamanio);
4          tbb::parallel_for(
5                  tbb::blocked_range<int>(0, tamanio, FISHNET_CHUNKSIZE_DATOS),
6                  [=] (const tbb::blocked_range<int> &rango) {
7                          for (int i = rango.begin(); i < rango.end(); i++) {
8                                  nuevos[i] = unos[i] + otros[i];
9                          }
10                 }
11         );
12         return nuevos;
13 }
```

Listado 21: Código de la función `sumar_datos` de la versión paralela del simulador de red de pesca.

```
1  Fuerza *Fuerza_neta(Fuerza ** fuerzas, int cantidad)
2  {
3          return tbb::parallel_reduce(
4                  tbb::blocked_range<int>(
5                          0, cantidad, FISHNET_CHUNKSIZE_FUERZA_NETA
6                  ),
7                  Fuerza_nula(),
8                  [=] (const tbb::blocked_range<int> &rango, Fuerza * inicial) ->
                       Fuerza * {
9                                  Fuerza * acumulador = nueva_Fuerza(
10                                         (*inicial)[0], (*inicial)[1], (*inicial)[2]
11                                 );
12                                 for (int i = rango.begin(); i < rango.end(); ++i)
                                       {
13                                     (*acumulador)[0] += (*fuerzas[i])[0];
14                                     (*acumulador)[1] += (*fuerzas[i])[1];
15                                     (*acumulador)[2] += (*fuerzas[i])[2];
16                                 }
17                                 borrar_Vector(inicial);
18                                 return acumulador;
19                 },
20                 [=] (Fuerza * una, Fuerza * otra) -> Fuerza * {
21                                 Fuerza * resultado = sumar_Vectores(una, otra);
22                                 borrar_Vector(una);
23                                 borrar_Vector(otra);
24                                 return resultado;
25                 }
26         );
27 }
```

Listado 22: Código de la función `Fuerza_neta` de la versión paralela del simulador de red de pesca.

la cantidad por defecto de hilos, que equivale a uno por cada hilo de hardware y en el sistema en que se corrieron las pruebas es 228.

La tabla de la figura 23 muestra los resultados de las pruebas. Se muestran el tiempo total transcurrido promedio de la versión secuencial (T_1) y de la versión paralela (T_{228}), obtenidos del perfilador VTune Amplifier, para las 5 ejecuciones hechas para cada versión, junto a sus variaciones estándar. El tiempo transcurrido se empezó a contar desde la ejecución del primer paso de la simulación y se dejó de contar luego de la ejecución del último paso, excluyendo el tiempo transcurrido al principio y al final del programa usado para entrada y salida e inicialización de los datos. T_1 tiene un valor de 1825.954s y T_{228} vale 12.943s. Con estos datos, el speedup (S_{228}) obtenido vale 141.076. Este valor de speedup no se acerca a ser lineal respecto al número de hilos. Dos características de Xeon Phi pueden tenerse en cuenta para intepretar mejor este resultado:

Métrica	**Valor**	**Desviación estándar**
T_1(s)	1825.954	93.970
T_{228}(s)	12.943	0.675
S_{228}	141.076	—
T_∞(s)	100.179	5.325
$\%_{T_\infty}$(%)	5.486	0.288

Figura 23: Resultados de las pruebas preliminares.

por un lado, por más que un núcleo soporte hasta 4 hilos de hardware simultáneos, todos los hilos comparten el mismo pipeline, por lo que es esperable que, incluso en un caso ideal, el speedup no se acerque a la cantidad total de hilos. Por otro lado, un núcleo no puede ejecutar instrucciones del mísmo hilo en dos ciclos consecutivos, por lo que en la versión secuencial se alcanza, como mucho, la mitad del desempeño del núcleo en el que ejecuta; esto significa que es esperable que el speedup supere a la cantidad de núcleos en los que se ejecuta, con una cota superior equivalente al doble de la cantidad de núcleos. Sin embargo, el procesador en el que se ejecutaron las pruebas tiene 57 núcleos, por lo que el speedup obtenido es mayor que el doble de la cantidad de núcleos. Esto sugiere que hay factores extra a la muliplicación de los recursos de cómputo que están afectando positivamente al desempeño en la versión paralela; una hipótesis puede ser un mejor aprovechamiento de la memoria debido al tamaño de la caché conjunta en comparación con el de la caché de un único núcleo. Con estas observaciones, los resultados obtenidos son favorables a la hipótesis de que el programa implementado es escalable, hipótesis que se pone a prueba en la siguiente sección.

3.3. Evaluación experimental: pruebas con distintos tamaños de red

En las arquitecturas manycore, optimizar el desempeño de una aplicación requiere que la aplicación sea escalable, lo que implica estar programada para ejecutar aprovechando los recursos de todos los núcleos de la arquitectura. En este capítulo se evalúa la escalabilidad en Xeon Phi de la simulación de red de pesca introducida en el capítulo 3.1 con las modificaciones del capítulo 3.2.

Para evaluar la escalabilidad de la aplicación de la simulación de red de pesca, se corrieron y se analizaron los resultados de pruebas experimentales en Xeon Phi con redes de distintos tamaños. En la sección 3.3.1 se detallan las pruebas experimentales que se realizaron, incluyendo los parámetros de entrada que las determinan y los criterios detrás de su selección. En la sección 3.3.2 se exponen y analizan los resultados de las pruebas.

3.3.1. Descripción de las pruebas

El objetivo del trabajo que se presenta en este capítulo fue el de probar la escalabilidad de la aplicación de simulación de redes de pesca en Xeon Phi. Para eso, se observó el impacto en el uso de los recursos del coprocesador al variar la cantidad de datos a procesar, es decir, el tamaño de la red de la simulación. Como principal medida de la escalabilidad, se estudia el nivel de paralelismo obtenido en las ejecuciones con distintos tamaños de red y distintas cantidades de hilos de ejecución. Al incrementar el tamaño de la red de pesca, se crea más trabajo, el cual los hilos de ejecución tendrán a disposición para ejecutar en forma de tareas.

La forma de la red (y por lo tanto su tamaño) es un parámetro del programa. Se tomó la decisión deliberada de no hacer optimizaciones al programa para adaptarlo a los tamaños de red, es decir, no hacer modificaciones en el código, manteniendo una versión fija del programa para ejecutar todas las pruebas. El propósito de esta decisión es el de estudiar si hay escalamiento transparente en el uso de los recursos al variar la cantidad de trabajo, o si es necesario adaptar la aplicación a cada tamaño de red para optimizar el paralelismo. Con este planteo, la responsabilidad de escalar a los recursos paralelos disponibles está delegada completamente, de forma transparente, a la biblioteca TBB y sus capacidades de asignación de tareas a hilos y balanceo de carga. Por cada tamaño de red elegido se ejecutan pruebas con distintas cantidades de hilos. Xeon Phi acepta hasta 4 hilos por núcleo, por lo que la distintas cantidades de hilos elegidas corresponden a la ejecución secuencial (un hilo en total) y 1, 2, 3 y 4 hilos por núcleo. Al ser TBB una biblioteca, no se puede configurar en el programa una asociación entre hilo y núcleo (en

contraste con el concepto de afinidad de OpenMP); lo que se puede elegir es la cantidad de hilos de ejecución totales, que luego son asignados a los hilos de hardware por el sistema operativo. Dado esto, para ejecutar pruebas utilizando 1, 2, 3 ó 4 hilos por núcleo, se corrieron pruebas indicando 57, 114, 171, y 228 hilos, respectivamente.

El tamaño de la red determina la cantidad de componentes (nudos y barras) que la constituyen. Los datos de entrada de cada prueba se generaron mediante el script de generación de condiciones iniciales que también fue usado en capítulo 3.2.2, el cual genera las condiciones correspondientes a una red cónica de mallas rómbicas en función su largo y diámetro. Las pruebas no buscan representar redes de pesca realistas, sino únicamente tamaños de datos de distinta dimensión con fines experimentales. En la figura 24 se pueden ver los tamaños de red seleccionados. El tamaño de red más chico (XS) se eligió apuntando a no presentar una cantidad de trabajo paralelo suficiente para explotar los recursos del coprocesador. A partir de ese tamaño inicial, las redes de las siguientes pruebas se determinaron siguiendo una secuencia de aumento exponencial en la cantidad de componentes de la red, con el fin de aumentar el trabajo total en la misma proporción. En la figura 25 se puede ver la cantidad de componentes correspondiente a cada tamaño y el factor de aumento de cada tamaño respecto al tamaño inmediatamente menor. Debido a que el script de generación de datos recibe como parámetro las dimensiones geométricas de la red, y no su cantidad de componentes, no se pudo forzar un factor de aumento fijo en la cantidad de componentes de una red respecto a la red inmediatamente menor, y los valores de los aumentos obtenidos terminaron variando entre 1.397 y 2.273. Las pruebas con redes mayores al tamaño XL no pudieron ejecutarse debido a fallas durante la ejecución, potencialmente provocadas por sobrecalentamiento del coprocesador.

3.3.2. Resultados obtenidos y análisis

Los tiempos de ejecución secuencial obtenidos para los distintos tamaños de red se pueden ver en la tabla de la figura 26. Algo que se puede notar preliminarmente es que el aumento del tiempo de ejecución secuencial correspondiente a un tamaño de red respecto al tamaño inmediatamente inferior no fue en la misma proporción que el aumento en la cantidad de componentes de la red: los tiempos de ejecución aumentaron en proporciones mayores. Por ejemplo, el caso más notorio

Etiqueta	**Largo** (m)	**Diámetro** (m)
XS	1.00	0.20
S	1.50	0.25
M	1.75	0.30
L	2.00	0.40
XL	3.00	0.60

Figura 24: Dimensiones de las redes de las pruebas que se ejecutaron.

Etiqueta	**Nudos**	**Barras**	**Objetos**	**Total**	**Crecimiento**
XS	1550	3100	10	4660	—
S	2925	5850	10	8785	1.885
M	4089	8178	10	12277	1.397
L	6200	12400	10	18610	1.515
XL	14100	28200	10	42310	2.273

Figura 25: Cantidades de componentes de las redes de las pruebas que se ejecutaron.

Etiqueta	T_1 (s)	σ_{T_1} (s)	**Crecimiento**
XS	188.897	19.997	—
S	610.042	100.713	3.229
M	1289.594	85.007	2.113
L	3282.550	86.851	2.546
XL	18459.397	868.255	5.621

Figura 26: Tiempos de ejecución secuencial para los distintos tamaños de red.

son los aumentos del tamaño XL respecto a L, donde aumentar la cantidad de componentes en un factor de 2.274 implicó un aumento del tiempo de ejecución de 5.621. Esto sugiere que el trabajo total no se relaciona linealmente con la cantidad de componentes. Sin embargo, sí se puede observar que aumentar la cantidad de componentes de forma exponencial derivó en un aumento exponencial del trabajo, por lo que las pruebas cumplen el propósito buscado.

Los tiempos de ejecución resultantes para ejecución secuencial y con 57, 114, 171 y 228 hilos, junto con su speedup asociado, para las redes de tamaño XS, S, M, L y XL se pueden ver en las tablas de las figuras 27, 28, 29, 30 y 31, respectivamente.

Respecto a los resultados obtenidos para la red XS, se podría observar que la ejecución de 57 hilos presentó un speedup casi ideal, dado que el valor obtenido, 55.920 es muy cercano a

P	T_P (s)	σ_{T_P} (s)	S_P
1	188.897	19.997	—
57	3.378	0.019	55.920
114	2.736	0.037	69.036
171	2.296	0.072	64.563
228	2.885	0.415	65.485

Figura 27: Resultados para las pruebas de tamaño XS.

P	T_P (s)	σ_{T_P} (s)	S_P
1	610.042	100.713	—
57	11.228	0.054	54.334
114	7.408	0.034	82.347
171	7.024	0.113	86.853
228	5.608	0.398	108.784

Figura 28: Resultados para las pruebas de tamaño S.

P	T_P (s)	σ_{T_P} (s)	S_P
1	1289.594	85.007	—
57	22.185	0.116	58.129
114	13.761	0.077	93.712
171	12.519	0.053	103.009
228	9.510	0.421	135.601

Figura 29: Resultados para las pruebas de tamaño M.

P	T_P (s)	σ_{T_P} (s)	S_P
1	3282.550	86.851	—
57	51.502	0.138	63.756
114	30.731	0.104	106.848
171	27.141	0.291	120.980
228	20.293	0.578	161.805

Figura 30: Resultados para las pruebas de tamaño L.

P	T_P (s)	σ_{T_P} (s)	S_P
1	18459.397	868.255	—
57	231.233	31.668	79.830
114	155.962	0.736	118.358
171	140.401	5.056	131.476
228	147.036	16.691	125.543

Figura 31: Resultados para las pruebas de tamaño XL.

la cantidad de hilos, lo que implica un speedup lineal; el tamaño del problema sería suficiente para mantener en uso los hilos asignados. Sin embargo, por los resultados obtenidos en la sección 3.2.2, se sabe que es posible obtener speedups mayores, y cuando se observan los resultados de la red XS para 114, 171 y 228 hilos, se puede apreciar que el speedup se mantiene cercano al valor correspondiente a 57 hilos. Aumentar la capacidad de procesamiento (medida en cantidad de hilos asignados) no está aumentando de forma significativa el speedup obtenido, lo que sugiere que pueden ser necesarios problemas más grandes para obtener mejor desempeño. El mejor speedup, 69.036, se obtuvo con 114 hilos.

A diferencia de las pruebas de tamaño XS, en las pruebas con tamaño S se puede ver que al aumentar la cantidad de hilos hay un decrecimiento considerable en el tiempo de ejecución, por lo que el speedup aumenta junto con la cantidad de hilos. El mayor speedup obtenido fue 108.784 con 228 hilos. Las pruebas de tamaño M siguen la misma tendencia de las pruebas de tamaño S, mejorando el tiempo de ejecución al aumentar la cantidad de hilos, llegando al valor 135.601 con 228 hilos. La tendencia se vuelve a repetir para la red de tamaño L. El máximo speedup obtenido fue para la red de tamaño L con 228 hilos y valió 161.805.

Los resultados de la red XL no siguen la misma tendencia que las redes S a L: en primer lugar, el speedup no aumenta junto con la cantidad de hilos; en segundo, el speedup más alto, obtenido para 171 hilos, es 131.476, considerablemente menor al speedup más alto obtenido en las pruebas anteriores.

Con los resultados mencionados, respecto al impacto de usar distintas cantidades de hilos, se pueden concluir tres cosas: por un lado, si la cantidad de trabajo no es suficiente, no hay una mejora considerable al aumentar la cantidad de hilos. Con hasta un hilo por núcleo se obtiene un speedup lineal, y con 2 ó más hilos la mejora en el tiempo de ejecución no es considerable. Por otro lado, dado que las pruebas con tamaño de red entre S y L mantienen una tendencia a ir disminuyendo los tiempos de ejecución al ir aumentando la cantidad de hilos, problemas de tamaño considerable se benefician de ejecutar con más hilos. Por último, se puede observar que, a partir de cierto tamaño de problema, ejecutar con 3 hilos por núcleo tiene mejor desempeño que ejecutar usando los 4 hilos de cada núcleo.

Otra métrica que sirve para evaluar la escalabilidad de la aplicación es el span. En la tabla de la figura 32 se muestra el span correspondiente a las pruebas de cada tamaño de red, junto con

Etiqueta	T_∞ (s)	σ_{T_∞} (s)	$\%_{T_\infty}$	$\sigma_{\%_{T_\infty}}$
XS	56.495	8.724	29.908	7.553
S	92.803	2.972	15.213	2.722
M	97.696	3.662	7.576	0.500
L	100.521	6.479	3.061	0.200
XL	173.785	9.027	0.941	0.079

Figura 32: Cálculo aproximado del span para los distintos tamaños de red.

la fracción que representa en el trabajo total. Los valores son aproximados, calculados siguiendo el criterio descrito en la sección 2.4.3. Se puede observar que la fracción correspondiente al span en el trabajo total disminuye a medida que aumenta el tamaño de la red. Si se considera a la fracción del span como equivalente al concepto de fracción secuencial de la leyes de Amdahl y Gustafson-Barsis, con estos datos se puede considerar que se cumple una situación similar a la observación originaria de la ley de Gustafson-Barsis, que notaba que el trabajo paralelizable de las aplicaciones paralelas aumentaba con el tamaño del problema mientras que el trabajo secuencial se mantenía estático. En la prueba de tamaño XS se puede ver el efecto de la ley de Amdahl: por más que Xeon Phi tenga muchos recursos disponibles, solo se pueden explotar en los fragmentos paralelos del programa; el tiempo total transcurrido del programa se ve fuertemente impactado por la porción menos paralelizable del programa, la cual es equivalente al span. El trabajo paralelizable del programa es el trabajo contenido dentro de cada paso de Runge–Kutta, y la cantidad de trabajo de este tipo está determinada por el tamaño de la red. El efecto de la ley de Gustafson-Barsis se puede ver con el aumento del tamaño de la red: cuanto más grande es la red, aumenta el trabajo total a la vez que disminuye la fracción ocupada por el span, lo que indica que hay mayor cantidad de trabajo paralelizable, permitiendo una mejor explotación de los recursos paralelos durante el transcurso de la ejecución. La gráfica de la figura 33 muestra esta tendencia: a medida que aumenta el tamaño de la red, el trabajo paralelizable aumenta y el span ocupa una fracción menor del trabajo total; a medida que la fracción del span disminuye, el speedup aumenta. No obstante, el caso del tamaño de red XL, que rompe la tendencia de aumento del speedup junto con el tamaño de la red, sugiere que, si bien es necesario una cantidad de trabajo paralelo considerable para obtener el mejor desempeño de la arquitectura, una cantidad de datos

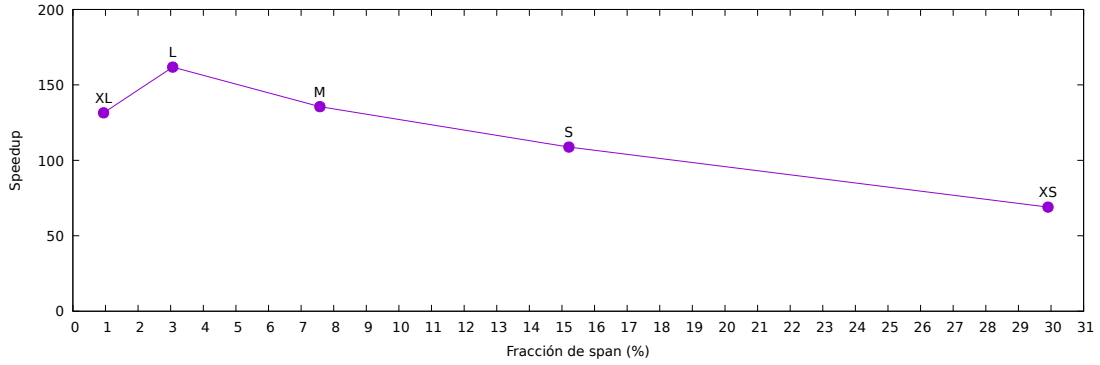


Figura 33: Gráfica del mayor speedup obtenido para las pruebas con distintos tamaño de red en función de la fracción del trabajo total representada por el span.

.

excesiva puede impactar negativamente en el desempeño. Para estudiar mejor las condiciones en las que se da esta disminución del desempeño, se podría experimentar con tamaños intermedios entre L y XL.

La tendencia observada para el speedup indica que el programa desarrollado para la simulación de la red de pesca es escalable, pero que necesita una cantidad de trabajo considerable para aprovechar los recursos de Xeon Phi, y que cantidades de datos excesivas pueden impactar negativamente en el desempeño. Por más que el programa sea escalable, si la cantidad de trabajo no es suficiente, el trabajo que presenta bajo paralelismo ocupa una porción considerable del tiempo de ejecución en relación con el tiempo de ejecución de tareas paralelas, y además, al no haber suficiente trabajo para repartir, gran parte de los recursos se gastan en tiempo ocioso. Sin embargo, la escalabilidad permite que, al aumentar el tamaño del problema, la cantidad de trabajo paralelo agregada se reparta de forma equitativa entre los hilos del coprocesador, tendiendo a un nivel de paralelismo promedio cercano a la cantidad de hilos. De no ser escalable el programa, aumentar el tamaño del problema no generaría trabajo paralelizable suficiente como para asignar a los recursos disponibles, aumentando en cambio el span y el tiempo ocioso de los hilos de hardware.

Es importante destacar que la escalabilidad se obtuvo gracias a las capacidades de TBB para el balanceo de carga. En estas pruebas se evidenció la utilidad de la sobredescomposición: no hubo impactos negativos en el desempeño con los aumentos exponenciales del tamaño del problema entre las pruebas de tamaños XS y L. El trabajo excedente fue particionado y asignado a los hilos de hardware de forma escalable completamente por TBB, corriendo exactamente el mismo programa en todas las pruebas, y sin necesidad de optimizaciones particulares para cada prueba.

3.4. Otras pruebas realizadas

En este capítulo, se describen experimentos realizados sobre la simulación en la red de pesca en Xeon Phi que no derivaron en resultados concluyentes. Se consideró importante comentarlos, de todas formas, para tener documentado el trabajo realizado. Además, se considera que pueden derivar en incógnitas para potencial trabajo futuro.

3.4.1. Ejecución extendida

Se quiso evaluar si había algún impacto en el desempeño al ejecutar una simulación más prolongada. Para esto, se ejecutaron pruebas con los mismos parámetros que las del capítulo 3.2.2 pero aumentando la cantidad de pasos a 100. Los resultados se pueden ver en la tabla de la figura 34. El speedup obtenido para 228 hilos respecto a la ejecución secuencial fue de 168.112, un valor 19.1 % mayor que el speedup obtenido en las pruebas de la sección 3.2.2. Una explicación del aumento del speedup se desprende al observar la fracción correspondiente al span del trabajo total, que se redujo a 0.145 %, en comparación con la sección 3.2.2 donde valió 5.486 %. Que el

Métrica	**Valor**	**Desviación estándar**
T_1(s)	18723.921	41.988
T_{228}(s)	111.378	0.405
S_{228}	168.112	—
T_∞(s)	27.188	4.864
$\%_{T_\infty}$(%)	0.145	0.026

Figura 34: Resultados de las pruebas con una simulación de 100 pasos.

trabajo correspondiente al span no haya aumentado proporcionalmente a la cantidad de pasos es evidencia a favor de que el trabajo menos paralelizable se presenta principalmente en el trabajo de inicialización (por ejemplo, en la creación inicial de hilos de TBB). En función de los resultados, se puede concluir que realizar una simulación larga no tuvo impacto negativo en el desempeño, y, de hecho, permitió que los datos recolectados reflejen mejor el grado de paralelismo que presenta el programa.

Se intentó ejecutar simulaciones con cantidades de pasos mayores, pero no se pudo. Intentar una ejecución más larga provocaba que el nodo se reiniciara o que el coprocesador quedara indisponible hasta reiniciar el nodo. Luego de monitorear los sensores de temperatura en esos intentos de ejecución, se dedujo la hipótesis de que los reinicios fueran causados por sobrecalentamiento del coprocesador. Considerando el trabajo y las dificultades de infraestructura que requeriría mejorar la refrigeración, se desistió de seguir probando simulaciones largas.

3.4.2. Análisis de uso de caché y vectorización

Para hacer evaluaciones adicionales del desempeño de la aplicación, se recolectaron las métricas descritas por Cepeda [37] (presentadas en la sección 2.4.3), disponibilizadas por VTune en los análisis de tipo General Exploration y Memory Access. En la tabla de la figura 35 se pueden ver las métricas recolectadas para la simulación de la sección 3.2.2, junto a los valores óptimos sugeridos. Excepto para las métricas L1 Hit Ratio, L2 TLB Miss Ratio y L1 TLB Misses per L2 TLB Miss, los valores obtenidos no son óptimos, y, por lo tanto, la aplicación no está haciendo un uso eficiente de la arquitectura. El valor correspondiente a la intensidad vectorial, 4.206, se puede explicar si se tiene en cuenta que la mayoría de las operaciones matemáticas que se hacen en el programa son sobre vectores tridimensionales. Que el promedio esté por encima de 3 sugiere que el compilador logra obtener una vectorización mayor en algunas secciones del código. Tanto el valor del impacto estimado de la latencia, que indica que la mayoría de los misses de caché L1 no están siendo servidos por L2, como la tasa de misses de TLB L1, advierten que no se está siguiendo un patrón eficiente de acceso a la memoria. Todas las métricas mencionadas pueden optimizarse mediante técnicas que permitan aprovechar mejor la localidad, como pueden ser caché tiling o la transformación de las estructuras de datos al formato estructura de arreglos. La aplicación de estas técnicas requiere modificaciones generalizadas en el código de la aplicación.

Un valor que llama la atención es el de la intensidad computacional, estimada por la métrica L1 Compute to Data Access Ratio. Dado que la mayoría de las operaciones del programa son sobre vectores tridimensionales, sería esperable que el valor fuera cercano a 3, pero es 0.620. Se sospecha que esto puede deberse a que sea frecuente que la lectura de un vector requiera dos

Métrica	**Valor**	**Desviación estándar**	**Valor óptimo**
L1 Compute to Data Access Ratio	0.620	0.003	> 4.206
L2 Compute to Data Access Ratio	31.835	0.092	> 62.04
L1 Hit Ratio	0.981	0.001	> 0.950
Estimated Latency Impact	380.094	4.424	< 145.000
L1 TLB Miss Ratio	0.018	0,003	< 0.010
L2 TLB Miss Ratio	0.000	0.000	< 0.001
L1 TLB Misses per L2 TLB Miss	107.491	13.572	$\gg 1$
Vectorization Intensity	4.206	0.007	8
Average DRAM Bandwidth (GB/s)	19.018	0.372	> 80

Figura 35: Métricas de desempeño recolectadas para la simulación de la sección 3.2.2.

accesos a la memoria principal, dado que la memoria no se está asignando de forma alineada con las líneas de caché. Para poner a prueba esta hipótesis, se generó una nueva versión de la aplicación usando el asignador de memoria `cache_aligned_allocator` de TBB, que se encarga de que la memoria se asigne de forma alineada. En la tabla 36 se pueden ver las métricas de tiempo de ejecución de las pruebas realizadas corriendo la simulación de la sección 3.2.2 con la versión que usa asignación alineada de memoria; para el valor de T_1 se usó el mismo que el obtenido por la versión secuencial de dicha sección. La tabla de la figura 37 muestra las demás métricas de desempeño recolectadas. El speedup vale 115.731, un valor menor al obtenido en la sección 3.2.2, 141.077. Usar asignación de memoria alineada puede tener el efecto secundario de aumentar la cantidad de bloques de memoria totales ocupados por los datos del programa, lo que deriva en una mayor transferencia de datos desde la memoria principal a la caché. El valor del ancho de banda de memoria promedio utilizado sugiere que este efecto secuendario puede estar ocurriendo, dado que pasó de 19.018GB/s en la versión anterior a 29.025GB/s en la versión actual. Si bien esto hace un mejor aprovechamiento del ancho de banda disponible (aunque el valor aún sea subóptimo), aumenta el tiempo total de espera por lecturas de memoria, lo que puede impactar negativamente en el desempeño. La modificación no tuvo impacto significativo en las demás métricas de desempeño. Se puede concluir que la aplicación no está teniendo un desempeño óptimo, subutilizando las capacidades de la arquitectura, y que el problema no se soluciona usando alineamiento en la asignación de memoria, sino que requiere cambios que mejoren los patrones de acceso a los datos para explotar mejor la localidad y la vectorización.

Métrica	**Valor**	**Desviación estándar**
T_1(s)	1825.954	105.062
T_{228}(s)	15.778	0.159
S_{228}	115.731	—
T_∞(s)	135.053	7.695
$\%_{T_\infty}$(%)	7.396	0.713

Figura 36: Resultados de las pruebas para la simulación de la sección 3.2.2 usando asignación alineada de memoria.

Métrica	**Valor**	**Desviación estándar**	**Valor óptimo**
L1 Compute to Data Access Ratio	0.619	0.002	> 4.201
L2 Compute to Data Access Ratio	31.790	0.105	> 61.88
L1 Hit Ratio	0.981	0.000	> 0.950
Estimated Latency Impact	380.148	4.939	< 145.000
L1 TLB Miss Ratio	0.019	0.002	< 0.010
L2 TLB Miss Ratio	0.000	0.000	< 0.001
L1 TLB Misses per L2 TLB Miss	128.780	21.764	$\gg 1$
Vectorization Intensity	4.201	0.009	8
Average DRAM Bandwidth (GB/s)	29.025	4.378	> 80

Figura 37: Métricas de desempeño recolectadas para la simulación de la sección 3.2.2 usando asignación alineada de memoria.

4. Conclusiones

En el relevamiento del estado del arte respecto a Xeon Phi, se pudieron observar múltiples publicaciones que destacan la potencialidad del coprocesador de obtener altos grados de paralelismo usando modelos de programación paralela tradicionales de las arquitecturas de memoria compartida, lo que ayuda a reducir la curva de aprendizaje a la hora de adoptar Xeon Phi como solución de computación de alto desempeño. A la vez, en varias de las publicaciones relevadas se pudo observar cómo fue necesario que los algoritmos implementados cumplieran ciertos patrones para obtener un desempeño óptimo, incluso llegando al punto de usar operaciones de vectorización específicas de la arquitectura. Los resultados de la prueba de concepto del presente trabajo se condicen con ambas observaciones.

La aplicación de prueba elegida simula una red de pesca, resolviendo una ecuación diferencial con el método Runge–Kutta de orden 4. Respecto a la escalabilidad, el problema de resolver una ecuación diferencial es que se resuelve en pasos. Estos pasos, al tener dependencias entre ellos, no pueden resolverse de forma paralela. Por lo tanto, la solución que se planteó fue paralelizar cada paso y luego ejecutar estos pasos de forma secuencial. Las pruebas realizadas con distintos tamaños de red confirman que la oportunidad de explotar los recursos paralelos del coprocesador está limitada al trabajo disponible en cada paso, lo cual es determinado por el tamaño de la red. Cuanto más grande sea la red, habrá más cantidad de trabajo disponible por paso, con una mayor oportunidad de explotación de los recursos. Estas observaciones se condicen con la ley de Gustafson-Barsis, que liga la posibilidad de escalamiento de un programa al tamaño del problema que resuelve. Las pruebas mostraron que el programa de la red de pesca aprovecha los recursos del coprocesador a medida que el tamaño de la red aumenta, y que, a partir de cierto tamaño de red, el uso de los recursos se satura y el desempeño se estanca; esto se da cuando todos los núcleos de Xeon Phi están ejecutando trabajo de cómputo de forma simultánea y constante, con 2 o más hilos por núcleo. Se puede concluir que, si bien la aplicación de la red de pesca es escalable, no presenta un desempeño óptimo para redes de pesca chicas. Sin embargo, también se puede concluir que el tamaño de red no puede ser arbitrariamente grande, ya que a partir de cierto tamaño el desempeño se empieza a degradar. Dado que las métricas de uso de vectorización y caché mostraron valores subóptimos, se considera probable que existan optimizaciones que permitan obtener desempeños mayores.

Una incógnita que se planteó al inicio de este trabajo fue la de identificar en qué tipo de problemas es adecuado usar Xeon Phi para su resolución. De las fuentes consultadas y de las pruebas realizadas se desprende que el algoritmo que se quiera ejecutar en Xeon Phi de forma óptima debe presentar tres características: escalabilidad, vectorización y localidad de memoria. El algoritmo debe ser escalable a una cantidad de hilos cercana a la cantidad de hilos de hardware del coprocesador. No se descarta que en algunos casos se pueda obtener un desempeño óptimo con dos hilos por núcleo, pero lo normal es que se deban usar 3 ó 4 hilos por núcleo; de hecho, TBB por defecto levanta 4 hilos por núcleo, y no se recomienda modificar la cantidad. Escalar a la cantidad de hilos mencionada implica tener un tamaño de problema acorde para repartir el trabajo necesario entre todos los hilos; sin embargo, la memoria de Xeon Phi es limitada (16GB como máximo), por lo que resolver problemas muy grandes puede requerir cooperación de varios coprocesadores, con el procesador principal u otros nodos mediante MPI. Que un algoritmo sea vectorizable, generalmente, implica que siga un patrón de acceso a los datos, específicamente, que acceda simultáneamente a datos contiguos en memoria y les aplique la misma transformación; presentar localidad tiene implicancias similares. Estas características hacen que los algoritmos aplicables a Xeon Phi, frecuentemente, sean también aplicables a las GPUs. Luego de los experimentos realizados, queda la incógnita sobre la dificultad de adaptar algoritmos para que hagan uso de vectorización y localidad, y si es posible esta adaptación a cualquier algoritmo.

Respecto a la afirmación de que Xeon Phi no requiere pericia específica para el desarrollo de aplicaciones científicas, cosa que lo diferenciaría de otras arquitecturas de alto desempeño, se pudo comprobar que no fueron necesarios conocimientos específicos de la arquitectura para desarrollar un programa escalable: el programa de la red de pesca está implementado íntegramente en C++, usando TBB como modelo de programación paralela, lo que lo hace completamente portable a arquitecturas x86. Por otro lado, obtener un desempeño óptimo sí requiere de cierta pericia, dado que no todos los algoritmos escalables explotan vectorización y localidad, y no hay una forma transparente de explotarlas; la prueba está en que el programa de la red de pesca fue escalable pero no hizo un uso óptimo de la vectorización ni de la memoria caché. Sin embargo, existen múltiples técnicas, frecuentemente comentadas en la bibliografía, que, de aplicarse a los algoritmos, pueden permitir cumplir estas características. Estas técnicas tienen la particularidad de ser aplicables tanto a Xeon Phi como a cualquier arquitectura x86, con la salvedad de que en Xeon Phi son necesarias para obtener un desempeño óptimo, mientras que en los procesadores tradicionales, al tener mayor desempeño secuencial, puede obtenerse un buen desempeño sin la necesidad de aplicarlas. Xeon Phi demuestra la posibilidad de obtener paralelismo altamente escalable a través de arquitecturas y modelos de programación tradicionales. Si bien actualmente está descontinuado, el aumento sostenido del paralelismo en las arquitecturas x86 sugiere que las observaciones aplicadas a Xeon Phi podrían aplicarse a futuras arquitecturas.

VTune demostró ser una herramienta útil para la optimización en Xeon Phi con TBB. No solo sirvió para observar datos cuantitativos del desempeño, sino que también aportó posibles causas para los indicadores negativos que se observaban. Esta información fue usada para tomar decisiones de optimización que mejoraron notoriamente el desempeño. El trabajo realizado se basó fundamentalmente en análisis de tipo Advanced Hotspots y solo se usaron otros tipos de análisis superficialmente. Por lo tanto, es probable que se puedan obtener más indicadores relevantes con otros tipos de análisis, principalmente con el tipo General Exploration, que es particularmente mencionado en la bibliografía.

A. Especificaciones del sistema

En este apéndice se describe el sistema sobre el que se corrieron las pruebas documentadas y las configuraciones que se le realizaron como parte de este trabajo.

La plataforma usada para correr las pruebas fue un nodo que originalmente formaba parte del Clúster FING. El nodo contiene un coprocesador Xeon Phi 3120P (línea Knights Corner), de 57 núcleos (4 hilos de hardware cada uno, dando un total de 228 hilos) a 1.1GHz y con 6GB de memoria RAM. Todas las pruebas realizadas se hicieron en el modo ejecución nativa, corriendo enteramente dentro del coprocesador. El sistema operativo del nodo es CentOS 6. Se instaló la suite Intel Parallel Studio XE 2017, que contiene el compilador de C/C++ de Intel, TBB y VTune Amplifier, y es la última versión de la suite que soporta la línea de coprocesadores Knights Corner. La suite se instaló en el directorio `/opt/intel/parallel_studio_xe_2017`.

Se creó un usuario en el sistema operativo de nombre `proyecto-xeon-phi` y se lo agregó al grupo `clusterusers` y al grupo de administradores `clusteradmins`. En el archivo `.bash_profile` se agregó la siguiente línea para establecer las variables de entorno de Parallel Studio al inicio de cada sesión:

```
source /opt/intel/parallel_studio_xe_2017/psxevars.sh intel64
```

El usuario también existe en el coprocesador. Para crearlo se agregó su entrada correspondiente en el archivo `/var/mpss/mic0/etc/passwd`, el cual es copiado al coprocesador por el servicio MPSS en el arranque. Debido a que VTune requiere que se pueda ingresar al coprocesador sin contraseña, se creó una clave SSH sin contraseña `/.ssh/id_rsa`, y se la agregó al archivo `/var/mpss/mic0/home/proyecto-xeon-phi/.ssh/authorized_keys`.

Para instalar TBB en el coprocesador, se agregó la siguiente línea al archivo `/etc/mpss/mic0.conf`:

```
Overlay Simple /opt/intel/tbb/lib/mic/* /lib64/ On
```

Lo que indica la línea es que, en el arranque del coprocesador, el contenido del directorio `/opt/intel/tbb/lib/mic/` del nodo, donde está instalada TBB en su versión para MIC, se copie al directorio `/lib64/` del coprocesador.

VTune Amplifier requiere instalar drivers de recolección de eventos de hardware para poder hacer análisis sobre Xeon Phi. Para instalarlos, se ejecutó la siguiente línea, como se indica en la documentación de VTune [35]:

```
sudo INSTALL_GID=501 /opt/intel/vtune_amplifier_xe/bin64/k1om/micboot_install.sh
```

El identificador de grupo 501 corresponde al grupo `clusterusers`, dado que es el grupo para el cual se quieren habilitar los drivers de recolección.

Al momento de hacer las configuraciones para este trabajo, el nodo ya estaba configurado para funcionar como servidor NFS. El directorio del repositorio del proyecto está compartido con el coprocesador mediante NFS. Se agregó la siguiente línea al archivo `/etc/exports` para que el servidor compartiera el repositorio con el coprocesador:

```
/state/partition1/proyecto-xeon-phi/proyecto-xeon-phi 172.31.1.1(rw,async,
    no_root_squash)
```

Para que el cambio haciera efecto, se ejecutó:

```
sudo exportfs -a
```

El repositorio se monta en el coprocesador en la ruta `/proyecto-xeon-phi`. Para crear el punto de montaje en el sistema de archivos del coprocesador, se creó el directorio

`/var/mpss/mic0/proyecto-xeon-phi/` en el nodo. Se indicó que el directorio compartido se montara en su punto de montaje agregando una entrada correspondiente en el archivo `/etc/fstab` del coprocesador, lo que implicó agregar la siguiente línea al archivo `/var/mpss/mic0/etc/fstab` del nodo:

```
172.31.1.254:/state/partition1/proyecto-xeon-phi/proyecto-xeon-phi      /proyecto-
    xeon-phi      nfs     defaults        1 1
```

172.31.1.254 corresponde a la IP del sistema operativo del nodo en la red que comparte con el coprocesador.

B. Optimización asistida por VTune Amplifier

El proceso de paralelización del programa secuencial, cuyos resultados se comentan en el capítulo 3.2, en una parte considerable implicó trabajo de optimización, que consistió en analizar datos de desempeño en busca de cuellos de botella y otros fenómenos de la ejecución que puedan modificarse para mejorar el desempeño. El trabajo se realizó siguiendo un proceso iterativo, en el que, en cada paso, se usan reportes de desempeño generados con VTune Amplifier para determinar la optimización a aplicar en este paso. Una vez aplicada la optimización, se vuelve a correr un análisis de VTune para generar los reportes que sirvan para evaluar en el siguiente paso. Algunas de las optimizaciones realizadas siguiendo este procedimiento tuvieron un impacto considerable en el desempeño. En este apéndice, se muestra cómo se usó VTune para determinar las optimizaciones más relevantes.

De los tipos de análisis que puede realizar VTune en Xeon Phi, se eligió el tipo *Advanced Hotspots*. Para visualizar los resultados del análisis, se usó particularmente el punto de vista *Hotspots*, que muestra el tiempo de procesamiento de cada función del programa. De este punto de vista las dos ventanas más usadas fueron *Bottom-up* y *Top-down Tree*. Todos los datos corresponden a versiones no finales del código ejecutando la misma simulación que las pruebas del capítulo 3.2. Los datos mostrados no son rigurosos, por lo que no se plantean para hacer una evaluación experimental, sino para ilustrar el origen de los problemas de desempeño, la localización de los datos relevantes en VTune y el impacto de las optimizaciones en el desempeño.

B.1. Manejo de memoria escalable

Una de las optimizaciones de mayor impacto fue la incorporación de la librería proxy de memoria de TBB. En la figura 38 se puede ver la vista Bottom-Up del reporte correspondiente a una de las primeras versiones paralelas del programa. Lo que se puede observar es que las funciones que más tiempo de CPU consumen son las de manejo de memoria de la biblioteca estándar de C. Además, muestran un uso no óptimo de los recursos paralelos. Estas observaciones sugieren que las funciones de manejo de memoria presentan problemas de escalabilidad, lo que es un problema conocido [51]. Para mejorar la situación, se puede usar la biblioteca proxy de TBB para que se encargue de interceptar todas las llamadas a las funciones de manejo de memoria y reemplazar la implementación original por su propia alternativa escalable.

El tiempo total transcurrido incluido en el mismo reporte del que se extrajo la figura 38 fue 67.599s. Luego de enlazada la biblioteca proxy de TBB, VTune reportó el tiempo total transcurrido con el valor 35.148s. También, la métrica Spin Time bajó de 4733.327s a 1710.772s al usar la librería proxy.

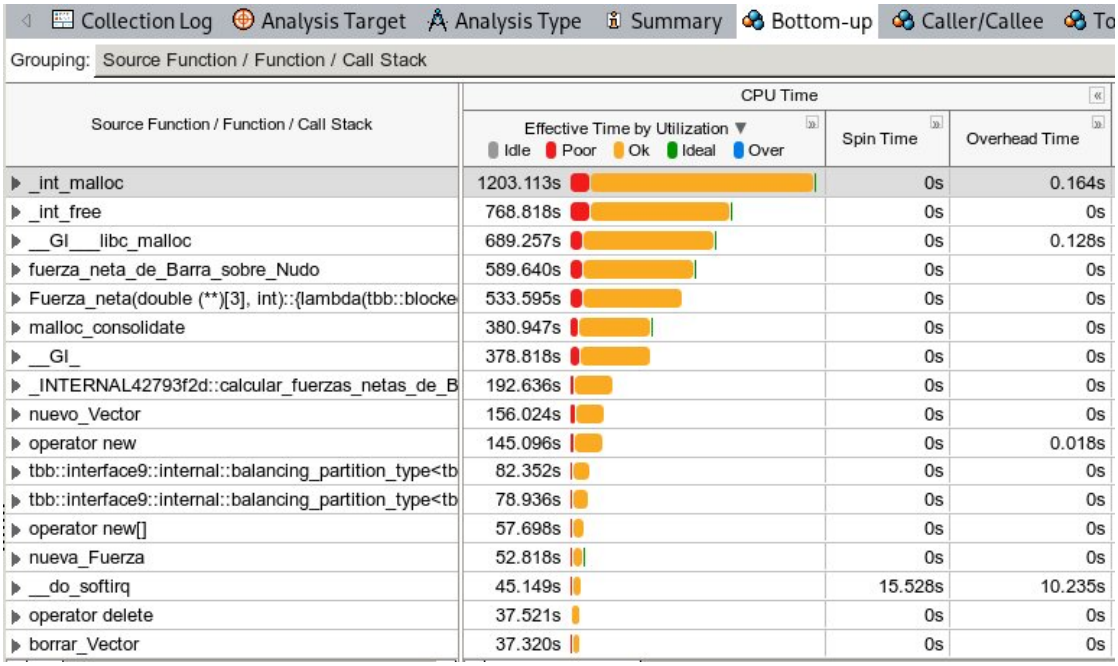


Figura 38: Ventana Bottom-up de la primera versión paralela del programa, ordenada por tiempo de CPU.

B.2. Reducción de Spin Time

Dentro del reporte del análisis realizado a la versión que incorpora la biblioteca proxy de TBB, se puede observar la gráfica de la figura 39, que es una gráfica de línea temporal que indica el estado de los hilos durante el transcurso de la ejecución. En la gráfica del medio, cada uno de los más de doscientos hilos de ejecución tiene una línea horizontal, la cual se pinta de verde en cada instante en que el hilo ejecuta. Se puede observar que hay múltiples franjas negras, las cuales indican zonas en que no se está explotando la mayoría del paralelismo disponible, prácticamente ejecutando secuencialmente.

VTune permite seleccionar una franja arbitraria en la línea temporal y filtrar los datos que se muestran para que solo incluyan los recolectados durante el transcurso de la franja. Para investigar cuáles pueden ser las funciones que están generando cuellos de botella, se seleccionó una de las franjas negras visibles en la gráfica de la figura 39 y se filtraron los datos para esa franja. Entre los datos filtrados se pudo observar lo que se ve en la figura 40, que es que la función

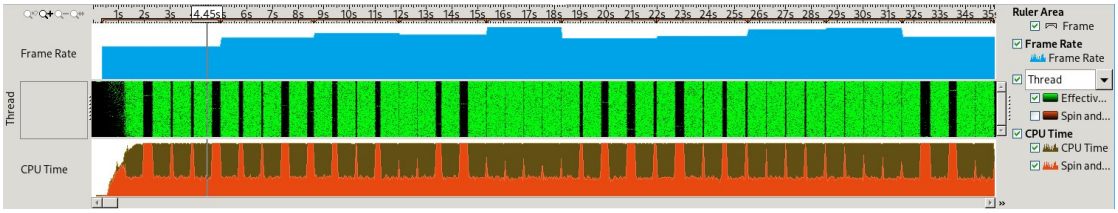
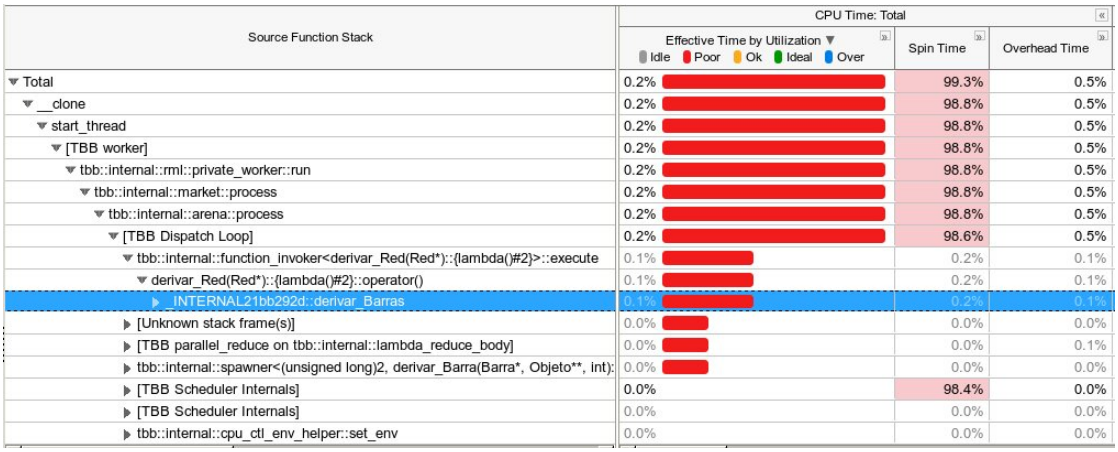


Figura 39: Gráficas de línea temporal del programa luego de agregar la biblioteca proxy de TBB.



Source Function Stack	CPU Time: Total Effective Time by Utilization ▼ Idle Poor Ok Ideal Over	Spin Time	Overhead Time
Total	0.2%	99.3%	0.5%
__clone	0.2%	98.8%	0.5%
start_thread	0.2%	98.8%	0.5%
[TBB worker]	0.2%	98.8%	0.5%
tbb::internal::rml::private_worker::run	0.2%	98.8%	0.5%
tbb::internal::market::process	0.2%	98.8%	0.5%
tbb::internal::arena::process	0.2%	98.8%	0.5%
[TBB Dispatch Loop]	0.2%	98.6%	0.5%
tbb::internal::function_invoker<derivar_Red(Red*)::{lambda()#2}>::execute	0.1%	0.2%	0.1%
derivar_Red(Red*)::{lambda()#2}::operator()	0.1%	0.2%	0.1%
_INTERNAL21bb292d::derivar_Barras	0.1%	0.2%	0.1%
[Unknown stack frame(s)]	0.0%	0.0%	0.0%
[TBB parallel_reduce on tbb::internal::lambda_reduce_body]	0.0%	0.0%	0.1%
tbb::internal::spawner<(unsigned long)2, derivar_Barra(Barra*, Objeto**, int):	0.0%	0.0%	0.0%
[TBB Scheduler Internals]	0.0%	98.4%	0.0%
[TBB Scheduler Internals]	0.0%	0.0%	0.0%
tbb::internal::cpu_ctl_env_helper::set_env	0.0%	0.0%	0.0%

Figura 40: Ventana Top-down Tree del programa luego de agregar la biblioteca proxy de TBB.

que más tiempo está ejecutando dentro de esa franja es la función interna `derivar_Barras`.

Analizando el código de la función `derivar_Barras` se pudo detectar código paralelizable que se estaba pasando por alto. Al aplicar TBB a las secciones paralelizables de la función, se obtuvo una mejora sensible del desempeño, llevando el tiempo transcurrido total a 30.537s, y reduciendo el Spin Time a 645.121s. El impacto también se puede visualizar en la figura 41: las zonas negras se redujeron y la gráfica de ejecución de los hilos se ve más homogénea, lo cual muestra que el paralelismo está siendo mejor explotado.

B.3. Reducción de Overhead Time

El Overhead Time es un efecto secundario del uso de librerías paralelas y, por más que sea esperable y aceptable que ocupe una fracción del tiempo de CPU, es importante minimizarlo. En la figura 42, el tooltip que se muestra dentro de la ventana Summary da una sugerencia para disminuir un tipo particular de Overhead Time: *Scheduling Time*, que es el tiempo que se utiliza en la asignación de tareas a hilos a través de TBB. En particular, VTune sugiere aumentar el tamaño de las tareas que se generan. Esto se puede hacer aumentando el grainsize, parámetro que utilizan las funciones `parallel_for` y `parallel_reduce`, en las regiones que estén generando tareas muy cortas. Otra alternativa es volver a dejar las implementaciones secuenciales de las regiones que generan una sobrecarga mayor al beneficio de ejecutarlas en paralelo.

El trabajo de optimización respecto a la disminución del Scheduling Time se hizo en múltiples iteraciones, guiándose por el aporte del Scheduling Time total que hace cada función en particu-

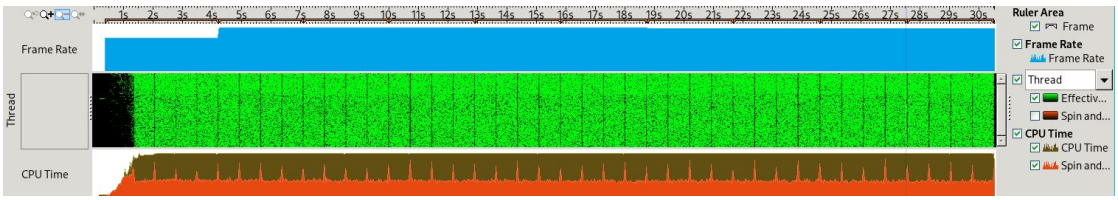


Figura 41: Gráficas de línea temporal del programa luego de paralelizar el código de la función `derivar_Barras`.

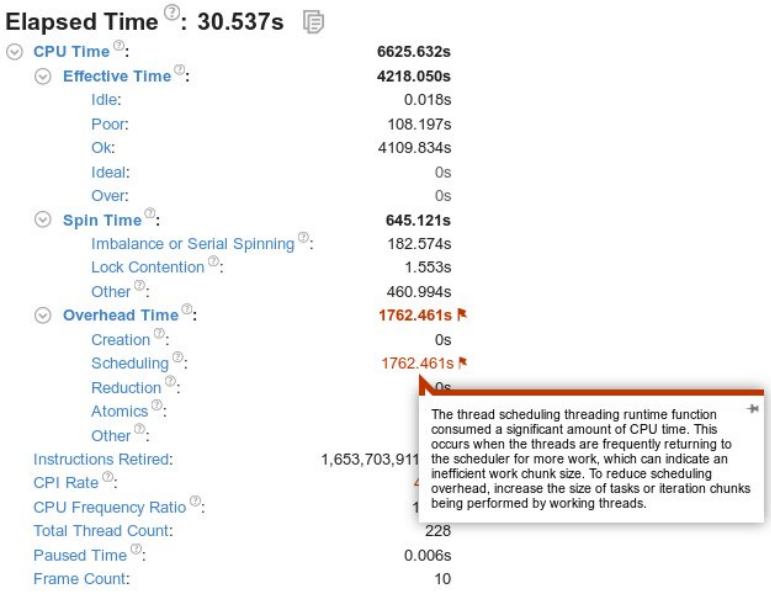


Figura 42: Ventana Summary del reporte una ejecución del programa luego de paralelizar el código de la función `derivar_Barras`.

lar, indicado por VTune. Una vez identificada la función que aporta más al Scheduling Time en una iteración dada, se corrieron pruebas con varios valores de grainsize y con su implementación secuencial. Una vez ejecutadas las pruebas, se seleccionó el valor que optimiza el tiempo total transcurrido del programa. Luego de la optimización de estos valores, el tiempo total transcurrido llegó a 12.246s y el Overhead Time se redujo de 1762.461s a 379.481s.

Referencias

[1] Viktor Eijkhout, Edmond Chow, Robert van de Geijn, *Introduction to High Performance Scientific Computing.* 2016.

[2] Gilbert Strang, *Computational Science and Engineering.* Wellesley-Cambridge Press, 2007.

[3] Benedict J. Leimkuhler, Sebastian Reich, Robert D. Skeel, *Integration Methods for Molecular Dynamics.* In *Mathematical Approaches to Biomolecular Structure and Dynamics*, Volume 82 of the series *The IMA Volumes in Mathematics and its Applications*, pp 161-185, 1996.

[4] David Padua, *Encyclopedia of Parallel Computing.* Springer, 2011.

[5] Michael McCool, Arch D. Robinson, James Reinders, *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann Publishers Inc., 2012.

[6] David Loshin, *High Performance Computing Demystified.* AP Professional, 1994.

[7] James Jeffers, James Reinders, *Intel Xeon Phi Coprocessor High Performance Programming: The Knights Landing Edition.* Morgan Kaufmann Publishers Inc., 2016.

[8] Charles Severance, Kevin Dowd, *High Performance Computing.* Connexions, 2009.

[9] Georg Hager, Gerhard Wellein, *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, 2011.

[10] Barry Wilkinson, Michael Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Pearson Prentice Hall, 2005.

[11] Paul B. Schneck, *Supercomputer Architecture.* Kluwer Academic Publishers, 1987.

[12] John L. Hennessy, David A. Patterson, *Computer Architecture: A Quantitative Approach.* Elsevier, Inc., 2012.

[13] David A. Patterson, John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface.* Elsevier, Inc., 2017.

[14] Andrew S. Tanenbaum, Todd Austin, *Structured Computer Organization.* Pearson Education Inc., 2013.

[15] Bao Li, Pinjing Lu, *The Evolution of Supercomputer Architecture: A Historical Perspective.* In *Volume 592 of the book series Communications in Computer and Information Science (CCIS)*, Springer, 2016.

[16] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, Andy White, *Sourcebook of Parallel Computing.* Morgan Kaufmann Publishers Inc., 2003.

[17] Aad J. van der Steen, *Overview of recent supercomputers.* `http://www.euroben.nl/reports/overview13.pdf` (Consultado el 1 de marzo de 2017.)

[18] *Efficiency, Power, Cores... — TOP500 Supercomputer Sites* `https://www.top500.org/statistics/efficiency-power-cores/` (Consultado el 2 de febrero de 2023.)

[19] *List Statistics — TOP500 Supercomputer Sites* `https://www.top500.org/statistics/list/` (Consultado el 2 de febrero de 2023.)

[20] *November 2022 — TOP500 Supercomputer Sites* https://www.top500.org/lists/2022/11/ (Consultado el 2 de febrero de 2023.)

[21] Ian Foster, *Designing and Building Parallel Programs.* http://www.mcs.anl.gov/~itf/dbpp/ (Consultado el 1 de marzo de 2017.)

[22] Fayez Gebali, *Algorithms and Parallel Computing.* John Wiley & Sons, Inc., 2011.

[23] *Intel® Xeon® Plantinum 8490H Processor.* https://www.intel.com/content/www/us/en/products/sku/231747/intel-xeon-platinum-8490h-processor-112-5m-cache-1-90-ghz/specifications.html (Consultado el 2 de febrero de 2023.)

[24] James Wang, *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210.* NVIDIA Corporation, 2014.

[25] David B. Kirk, Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Elsevier Inc., 2013.

[26] Sparsh Mittal, Jeffrey S. Vetter, *A Survey of Methods For Analyzing and Improving GPU Energy Efficiency.* En *ACM Computing Surveys, Vol. 47, No. 2.* Association for Computing Machinery, 2015.

[27] *November 2016 — TOP500 Supercomputer Sites* https://www.top500.org/green500/lists/2016/11/ (Consultado el 19 de abril de 2017.)

[28] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, *High Performance Computing on the Intel Xeon Phi: How to Fully Exploit MIC Architectures.* Springer International Publishing Switzerland, 2014.

[29] Rezaur Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers.* Apress Media, LLC, 2013.

[30] James Jeffers, James Reinders, *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Publishers Inc., 2013.

[31] *Products formerly Knights Corner.* https://ark.intel.com/content/www/us/en/ark/products/codename/57721/products-formerly-knights-corner.html (Consultado el 19 de setiembre de 2021.)

[32] *Intel® Parallel Studio XE 2017 Release Notes* https://software.intel.com/content/dam/develop/external/us/en/documents/psxe2017-release-notes-en-us-lin-win-743120.pdf (Consultado el 28 de setiembre de 2021.)

[33] *Intel® Parallel Studio XE 2018 Release Notes* https://software.intel.com/content/dam/develop/external/us/en/documents/ipsxe-2018-release-notes-en-743120.pdf (Consultado el 28 de setiembre de 2021.)

[34] *Intel® Parallel Studio XE and Intel® oneAPI Toolkits...* https://software.intel.com/content/www/us/en/develop/articles/intel-parallel-studio-xe-supported-and-unsupported-product-versions.html (Consultado el 28 de setiembre de 2021.)

[35] *Intel® VTune™ Amplifier XE 2017 and Intel® VTune™ Amplifier 2017 for Systems Help.*

[36] *Intel® VTune™ Profiler Release Notes and New Features* https://www.intel.com/content/www/us/en/developer/articles/release-notes/vtune-profiler-release-notes.html (Consultado el 14 de junio de 2022.)

[37] *Optimization and Performance Tuning for Intel® Xeon Phi™ Coprocessors, Part 2: Understanding and Using Hardware Events — Intel® Software* https://web.archive.org/web/20190218150111/https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding (Consultado el 2 de noviembre de 2021.)

[38] Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che, Ana Lucia Varbanescu, *Test-driving Intel Xeon Phi.* En *Proceedings of the 5th ACM/SPEC international conference on Performance engineering (ICPE '14).* Association for Computing Machinery, New York, NY, USA, 2014.

[39] James Reinders, Jim Jeffers, *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches.* Elsevier Inc., 2015.

[40] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, S. A. Jarvis, *Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors.* En *IEEE 27th International Symposium on Parallel and Distributed Processing.* IEEE, 2013.

[41] S. Ali Mirsoleimani, Aske Plaat, Jaap van den Herik, Jos Vermaseren, *Scaling Monte Carlo Tree Search on Intel Xeon Phi.* En *2015 IEEE 21st International Conference on Parallel and Distributed Systems.* IEEE, 2015.

[42] Ashkan Tousimojarad, Wim Vanderbauwhede, *Comparison of Three Popular Parallel Programming Models on the Intel Xeon Phi.* En *Euro-Par 2014 International Workshops. Porto, Portugal, August 25–26, 2014. Revised Selected Papers, Part II.* Springer, 2014.

[43] Evgeny Fiksman, Sania Salahuddin, *STAC-A2 on Intel Architecture: From Scalar Code to Heterogeneous Application* En *Seventh Workshop on High Performance Computational Finance.* IEEE, 2014.

[44] Xiangke Liao, Liquan Xiao, Canqun Yang, Yutong Lu, *MilkyWay-2 supercomputer: system and application.* Higher Education Press, 2014.

[45] Jack Dongarra, *Report on the Tianhe-2A system.* Tech. rep., University of Tennesssee Oak Ridge National Laboratory, 2017. https://www.icl.utk.edu/files/publications/2017/icl-utk-970-2017.pdf (Consultado el 19 de noviembre de 2021.)

[46] *Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 — TOP500.* https://top500.org/system/177999/ (Consultado el 2 de febrero de 2023.)

[47] Jack Dongarra, *Report on the Sunway Taihulight system.* Tech. rep., University of Tennesssee Oak Ridge National Laboratory, 2016. http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf (Consultado el 19 de noviembre de 2021.)

[48] *Changes in Customer Support and Servicing Updates for Select Intel® Processors* https://www.intel.com/content/www/us/en/support/articles/000022396/processors.html (Consultado el 3 de febrero de 2023.)

[49] *OpenMP Application Programming Interface Specification Version 5.2 November 2021.* https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf (Consultado el 20 de diciembre de 2021.)

[50] *OpenMP FAQ - OpenMP.* https://www.openmp.org/about/openmp-faq/ (Consultado el 20 de diciembre de 2021.)

[51] *Intel® Threading Building Blocks Documentation.*

[52] John L. Gustafson, *Reevaluating Amdahl's Law.* Communications of the ACM, 1988.

[53] Katsuya Suzuki, Tsutomu Takagi, *Numerical analysis of dynamic behavior of Danish seining and sea trial verification.* En *Mathematical and Physical Fisheries Science 6*, 2008.

[54] Takashi Shimizu, Tsutomu Takagi, Holger Korte, Tomonori Hiraishi, Katsutaro Yamamoto, *Application of NaLA, a fishing net configuration and loading analysis system, to drift gill nets.* En *Fisheries Research 76*. Elsevier, 2005.

[55] Holger Korte, Tsutomu Takagi, *Inertia transformation in hydromechanics.* En *Mathematical and Physical Fisheries Science 2*, 2004.

[56] Holger Korte, Tsutomu Takagi, *Inertia transformation based motion calculation of fishing nets with Matlab/Simulink.* En *Proceedings of the 4th Internaional Symposium on Automatic Control*, 2005.

[57] *BusyBox.* https://busybox.net/about.html (Consultado el 16 de mayo de 2022.)

[58] Karl Rupp, *Microprocessor Trend Data.* Repositorio en GitHub: https://github.com/karlrupp/microprocessor-trend-data (Consultado el 14 de junio de 2022.)

[59] Tsutomu Takagi, Takashi Shimizu, Katsuya Suzuki, Tomonori Hiraishi, Katsutaro Yamamoto, *Validity and layout of "NaLA": a net configuration and loading analysis system.* En *Fisheries Research 66*. Elsevier, 2004.

[60] Tsutomu Takagi, Katsuya Suzuki, Tomonori Hiraishi, *Development of the numerical simulation method of dynamic fishing net shape.* En *Nippon Suisan Gakkaishi 68*, 2002.

[61] Michael Voss, Rafael Asenjo, James Reinders, *Pro TBB: C++ Parallel Programming with Threading Building Blocks.* Apress Berkeley, CA, 2019.