Universidad de Valladolid

Máster universitario

Ingeniería Informática

Trabajo Fin de Máster

Analizando la escalabilidad de aplicaciones Iterative Stencil Loop en sistemas de supercomputación

Realizado por **DAVID DÍEZ POZA**

✠✠✠

Universidad de Valladolid

27 de junio de 2025

Tutor: ARTURO GONZÁLEZ ESCRIBANO

Resumen

Una clase importante de aplicaciones científicas con un alto potencial de escalabilidad son las aplicaciones ISL (Iterative Stencil Loop). EPSILOD es una herramienta para simplificar el desarrollo y ejecución de aplicaciones ISL en entornos heterogéneos distribuidos. En este trabajo se proponen mejoras y extensiones para una nueva versión de EPSILOD que amplían el rango de aplicaciones que se pueden construir y la eficiencia de los mecanismos de implementación y comunicación para conseguir un alto grado de escalabilidad en sistemas de cómputo de primer nivel. Se presentan resultados experimentales, con hasta 1024 GPUs distribuidas en 256 nodos, que indican que la nueva versión de EPSILOD puede conseguir altos niveles de escalabilidad fuerte y débil en diferentes tipos de escenarios y aplicaciones ISL. Se incluye una comparación experimental con otras herramientas del estado del arte que permiten implementar fácilmente aplicaciones ISL distribuidas: Muesli y Celerity basado en SYCL. Los resultados muestran que EPSILOD permite mejorar sus medidas de rendimiento, especialmente en altos niveles de escalabilidad.

Descriptores

Computación paralela, Aplicaciones stencil, Escalabilidad, Sistemas distribuidos, Sistemas heterogéneos, HPC

Abstract

An important type of scientific applications with high scalability potential are ISL (Iterative Stencil Loop) applications. EPSILOD is a tool designed to simplify the development and execution of ISL applications in distributed heterogeneous environments. This work proposes improvements and extensions for a new version of EPSILOD that expand the range of applications that can be built and enhance the efficiency of implementation and communication mechanisms to achieve a high degree of scalability in top-tier computing systems. Experimental results are presented, with up to 1024 GPUs distributed across 256 nodes, indicating that the new version of EPSILOD can achieve high levels of both strong and weak scalability in different types of scenarios and ISL applications. An experimental comparison is included with other state-of-the-art tools that facilitate the easy implementation of distributed ISL applications: Muesli and Celerity based on SYCL. The results show that EPSILOD improves performance metrics, especially at high levels of scalability.

Keywords

Parallel computing, Stencil applications, Scalability, Distributed systems, Heterogeneous systems, HPC

Índice general

Índice de figuras

Índice de tablas

1: Introducción

1.1. Contexto

En una época en la que predominan los sistemas pre-exaescala y comienzan a estar disponibles los primeros sistemas exaescala es cada vez más necesario aportar soluciones con un alto grado de escalabilidad que sean capaces de aprovechar al máximo este tipo de arquitecturas. Estos sistemas presentan normalmente arquitecturas heterogéneas, que emplean distintos tipos de unidades de cómputo CPU y GPU, estos últimos utilizados para todo tipo de procesamiento de propósito general (GPGPU).

1.2. Motivación

Uno de los retos clásicos en este contexto de HPC (High Performance Computing) es el desarrollo herramientas y modelos de programación eficientes en entornos altamente paralelos. A su vez, se busca mantener un alto grado de abstracción con el objetivo de reducir la complejidad inherente a la programación en estos entornos complejos. Algunas de estas soluciones son más genéricas, mientras que otras están adaptadas a un tipo de problema concreto.

Los ISLs o *Iterative Stencil Loops* son una clase representativa de aplicaciones científicas con un alto potencial de escalabilidad. En ellos, el valor de los elementos del dominio se obtiene repitiendo el mismo cálculo sobre cada elemento del dominio en cada iteración. El cálculo depende de los valores en la iteración anterior de un conjunto de elementos *vecinos*. El criterio de vecindad lo define cada aplicación concreta, creando un patrón de acceso fijo (*stencil*) que introduce dependencias con los datos vecinos. Esta característica implica que paralelizar el cómputo de *stencils* requiere comunicaciones de datos entre unidades de proceso que tienen asignados elementos vecinos en el dominio. El dominio se parte normalmente en bloques contiguos de grano grueso para maximizar la localidad y minimizar las comunicaciones. Sus características de localidad implican que son problemas especialmente adecuados para el uso GPUs en la computación local. Todo esto los convierte en problemas especialmente interesantes de cara a su uso en sistemas HPC.

El grupo de investigación Trasgo, ha propuesto EPSILOD [2] como modelo para el desarrollo de aplicaciones ISL. Está basado en el concepto de esqueletos paralelos. Abstrae los detalles del reparto de trabajo y los patrones de cómputo y comunicación para simplificar la implementación de ISLs eficientes en sistemas heterogéneos distribuidos. EPSILOD se ha convertido en una herramienta muy apropiada para experimentar nuevas técnicas de implementación, despliegue y ejecución de aplicaciones de tipo ISL en sistemas heterogéneos distribuidos de todo tipo de escala. EPSILOD se apoya en dos bibliotecas de funciones subyacentes que conforman su capa de portabilidad entre dispositivos heterogéneos (Controller) y de distribución de carga y comunicaciones en sistemas distribuidos (Hitmap).

1.3. Metodología empleada

La metodología iterativa basada en prototipos experimentales es un enfoque de desarrollo que se utiliza en diversas disciplinas, especialmente en ingeniería de software y diseño de productos. Esta metodología se centra en la creación de prototipos, que son versiones preliminares de un producto, con el fin de explorar ideas y validar conceptos de manera práctica.

Una de las características más destacadas de esta metodología es su naturaleza iterativa. Esto significa que el proceso se lleva a cabo en ciclos repetidos de diseño, implementación y evaluación. En cada ciclo, se crea un prototipo que se prueba y se evalúa, lo que permite a los desarrolladores y diseñadores obtener retroalimentación valiosa. Esta retroalimentación se utiliza para realizar ajustes y mejoras en el prototipo en la siguiente iteración, lo que ayuda a identificar y resolver problemas antes de avanzar a etapas más avanzadas del desarrollo.

Además, la flexibilidad es una ventaja importante de esta metodología. A medida que se avanza en el desarrollo, los equipos pueden adaptarse a cambios en los requisitos o en el entorno del proyecto, lo que les permite responder de manera más efectiva a las necesidades emergentes.

Estas características hacen a esta metodología especialmente adecuada a su uso en entornos de investigación. En estos la incertidumbre suele ser alta y es frecuente que se produzcan cambios de requisitos durante el proceso.

En el caso de este trabajo el ciclo iteretivo supone la creación de códigos de ejemplo y su puesta a punto para el despliegue y ejecución en supercomputador en el que se realizarán los experimentos. De esta experimentación se obtienen medidas de rendimiento o datos de perfilado de la ejecución, que permiten analizar el comportamiento e identificar posibles puntos de mejora.

En el estudio experimental se han obtenido repetidas mediciones de los tiempos de ejecución de las aplicaciones. Esto busca garantizar que los resultados son estadísticamente significativos. Para cada experimento se ejecutan 30 repeticiones, ya que, este es un tamaño de muestra suficientemente grande para que el Teorema Central del Límite se considere

aplicable. Los resultados presentados son la media de los tiempos de ejecución tras haber eliminado los *outliers*, es decir, aquellos resultados por debajo o por encima de la media $\pm 1{,}5 \times IQR$ (rango intercuartílico). Se considera emplear intervalos de confianza si se presentan resultados en los que sea necesario demostrar la diferencia o similitud de dos valores.

1.4. Objetivos

En este trabajo presentamos una serie de cambios de diseño y nuevas técnicas introducidas en EPSILOD y en las bibliotecas de funciones subyacentes que permiten: (1) Mejorar su capacidad para implementar nuevos tipos de aplicaciones ISL; (2) Aumentar su flexibilidad para experimentar con diferentes políticas y mecanismos de partición y mapeo; y (3) Mejorar su rendimiento y escalabilidad. En concreto presentamos los siguientes objetivos:

- Se añadirá soporte para tipos genéricos de datos. Esto permitirá trabajar sobre estructuras de datos cuyo elemento base sea una estructura arbitrariamente compleja, lo que abre las puertas a la experimentación con una mayor gama de aplicaciones.
- Se añadirá un nuevo sistema para escoger en tiempo de ejecución la política de partición del dominio. Este sistema simplifica el proceso de escoger las políticas e introduce nuevas combinaciones para particionar el dominio de una forma más intuitiva. Esto facilitará explorar como influyen diferentes políticas en el rendimiento de las aplicaciones.
- Optimizaremos el modelo de comunicaciones para mejorar el solapamiento entre cómputo y comunicación, lo que mejora la escalabilidad.
- Simplificaremos el mecanismo interno de gestión de colas basado en eventos, reduciendo el sobrecoste de ejecución del modelo en la capa de portabilidad que maneja los dispositivos heterogéneos. Estas mejoras son especialmente notables en las pruebas de escalabilidad fuerte, cuando se distribuyen cargas de trabajo de tamaño fijo entre un gran número de nodos.
- Realizaremos un estudio experimental en un supercomputador pre-exaescala para mostrar las nuevas mejoras de EPSILOD y su escalabilidad utilizando dos aplicaciones de ejemplo, una con dominio de dos dimensiones y otro con dominio de tres dimensiones y un tipo de datos más complejo. Este estudio contiene comparaciones de rendimiento con dos herramientas actuales del estado del arte que permiten implementar fácilmente aplicaciones ISL distribuidas: Muesli y Celerity (basado en SYCL). Este estudio incluye pruebas de escalabilidad débil y fuerte.

1.5. Estructura del documento

El resto del artículo tiene la siguiente estructura. En el capítulo 2 se describen varios conceptos empleados a lo largo del trabajo y las tecnologías y herramientas utilizadas. El capítulo 3 se detalla trabajo relacionado incluyendo otros modelos similares. El capítulo 4 describe las mejoras propuestas. Describimos el estudio experimental en el capítulo 5. Por último, el capítulo 6 trata las conclusiones y el trabajo futuro.

2: Conceptos previos

2.1. High performance computing

La computación de altas prestaciones (HPC) es aquella que utiliza sistemas con una potencia de cómputo muy elevada y que son capaces de resolver problemas a gran escala. Los sistemas HPC típicos como supercomputadores o clústeres consiguen estos niveles de computación agregando unidades de cómputo modulares conocidas como nodos.

El rendimiento de este tipo de sistemas se mide típicamente en operaciones de punto flotante por segundo (FLOPS). El proyecto TOP500 proporciona una lista con los 500 supercomputadores más potentes del mundo, en la que la mayoría tienen un rendimiento que va desde los pocos petaflops hasta alcanzar casi el exaflop. Estos son los conocidos como sistemas pre-exaescala. En los últimos años han ido surgiendo supercomputadores exaescala, que superan la marca del exaflop.

Este tipo de sistemas se utiliza en múltiples campos como, por ejemplo, la investigación científica para realizar simulaciones complejas y modelar fenómenos naturales, o la ingeniería para el diseño y análisis de estructuras, simulaciones de fluidos y estudios de materiales.

En general, se trata de sistemas heterogéneos, ya que cuentan unidades de cómputo con diferentes capacidades. Cada nodo de uno de estos sistemas suele tener una o varias CPUs, que realizan cómputo de propósito general y varios aceleradores hardware.

Los aceleradores hardware están diseñados para realizar un determinado tipo de operaciones de forma más eficiente que las CPUs. Un ejemplo que se usa de manera extendida en los grandes supercomputadores son las GPUs. Estas unidades de procesamiento gráfico se utilizan en este ámbito para cómputo de propósito general debido a su arquitectura SIMD (single instruction multiple data). Esta arquitectura permite aplicar la misma serie de operaciones sobre cada elemento de una estructura de datos, lo que hace de este hardware especialmente adecuado para cierto tipo de problemas que se consideran muy paralelizables.

Para la evaluación de este tipo de sistemas o del software que se ejecuta en ellos es tipo realizar dos tipos de pruebas de escalabilidad: débil y fuerte.

La escalabilidad débil es la capacidad de un sistema para mantener su rendimiento al aumentar el tamaño del problema manteniendo constante la carga en cada unidad de proceso.

Por su parte, la escalabilidad fuerte representa capacidad de un sistema para mejorar el rendimiento al repartir un problema de tamaño fijo entre un número creciente de unidades de proceso.

2.2. Aplicaciones Iterative Stencil Loop

Una aplicación ISL es una solución iterativa a determinados problemas de cómputo numérico. En cada iteración, todas las celdas de una estructura de datos que representa el dominio del problema se actualizan con los valores de la iteración anterior en un conjunto de celdas *vecinas*. Qué celdas se consideran vecinas lo determina el patrón geométrico fijo u operador de *stencil* que define cada aplicación.

Aunque no siempre, en este tipo de problemas es típico aplicar las denominadas condiciones de frontera de Dirichlet. En matemáticas, son unas condiciones impuestas a una ecuación en derivadas parciales, de tal forma que los valores de la solución en la frontera del dominio son fijos. Este tipo de ecuaciones se suelen emplear en simulaciones físicas, por lo que el término se utiliza también en este ámbito para designar esta condición en la frontera del dominio de la simulación.

Se trata de problemas altamente paralelizables debido a la localidad de datos. Así, una implementación paralela parte el espacio de la estructura de datos en bloques de datos contiguos que se asignan a cada proceso o dispositivo. A partir del operador de *stencil* se determina el tamaño y forma de los *bordes* de cada parte. Estos bordes representan las partes de la estructura que son celdas vecinas de alguna de las asignadas a otras partes. Por tanto, será necesario comunicar sus valores actualizados a otros dispositivos. La parte de datos asignada a cada dispositivo se amplía para tener unas zonas denominadas *halos*, donde se recibe la información de los bordes de otras partes que se han de recibir. Estos halos se utilizan en los cálculos de la siguiente iteración.

2.3. Introducción al Método de Red de Boltzmann

Los Métodos de Red de Boltzmann (Latice Boltzmann Method o LBM) son una clase de métodos de dinámica de fluidos computacional (CFD) emplean una red como dominio del problema.

En el contexto de la teoría de grupos, una red en un espacio de coordenadas reales consiste en un conjunto de puntos o celdas tales que la suma o resta por coordenadas de dos de ellos produce otro punto de la red. Además los puntos de la red están separados

por una distancia mínima y cada punto del espacio está a una distancia máxima de un punto de la red.

Los LBM se emplean para resolver problemas de dinámica de fluidos es están basados en el uso de funciones de distribución de partículas. Estas partículas fluyen en direcciones dadas (enlaces de la red) y colisionan en ciertas zonas.

Para describir completamente la dinámica de un sistema es necesario conocer la posición y la velocidad de sus partículas en función del tiempo. Mientras que otros métodos calculan la posición y velocidad de cada partícula, los métodos estadísticos como LBM buscan caracterizar el efecto de las moléculas a mayor escala, bajo la premisa de que no siempre es necesario conocer el estado de cada partícula para describir el efecto general en el sistema. Teniendo en cuenta las elevadas magnitudes del número de partículas en un fluido, esto supone un ahorro computacional significativo.

El espacio se puede discretizar de tal forma que es posible describir el número de partículas que se encuentran en una determinada región dentro de un intervalo de velocidades. Con esta definición del espacio, una función de distribución puede caracterizar el efecto de las moléculas en cada una de estas regiones.

En este contexto, la ecuación de Boltzmann en un sistema sin fuerzas externas es:

$$\frac{\partial f}{\partial t} + c\nabla f = \Omega \tag{2.1}$$

Aquí Ω denota el término de colisión: el ratio de cambio en el número de partículas. Este término simula el efecto de los choques entre partículas, que aumentan o disminuyen su velocidad promocionándolas o degradándolas a distintos intervalos de velocidad. Esto afecta así a su distribución. El término de colisión es una función de f compleja y 2.1 es una ecuación integrodiferencial difícil de resolver, por lo que Bhatnagar, Gross, y Krook definieron un modelo simplificado para este término (2.2).

$$\Omega = \omega(f^{eq} - f) = \frac{1}{\tau}(f^{eq} - f) \tag{2.2}$$

ω representa la frecuencia de colisión y τ el factor de relajación (el tiempo que tarda el sistema en alcanzar el equilibrio). f^{eq} es la función de distribución de equilibrio de las partículas.

La función resultante es válida en direcciones específicas: los ejes de la red. Se puede discretizar y expresar en una dirección específica como:

$$f_i(r + c_i\Delta t, t + \Delta t) = f_i(r, t) + \frac{\Delta t}{\tau}[f_i^{eq}(r, t) - f_i(r, t)] \tag{2.3}$$

La ecuación 2.3 supone la base para implementar simulaciones de dinámica de fluidos mediante LBM. Simplemente cambiando la función de distribución, y añadiendo un término para las fuerzas externas si es necesario, se pueden calcular diferentes magnitudes físicas.

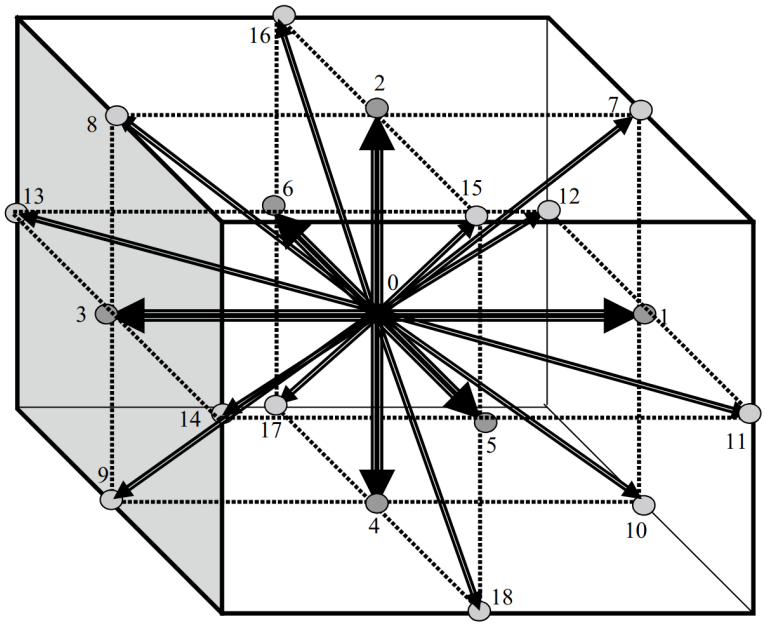


Figura 2.1: Disposición de la red D3Q19 que divide un espacio de 3 dimensiones de forma que cada nodo incluye 19 vectores de velocidad con una partícula central estacionaria.

Para resolver este tipo de problemas, el dominio se tiene que discretizar en una red. Cada punto o celda de la red contiene una serie de partículas ficticias caracterizadas por un vector de velocidad y una función de distribución. Estos vectores de velocidad constituyen los enlaces de la red y representan las distintas direcciones en que fluyen las partículas.

Existen diferentes modelos de red en función del número de dimensiones del dominio y del número de direcciones de flujo en cada celda. Estos modelos se denominan DnQm donde n es el número de dimensiones y m el número de direcciones de las partículas. Por ejemplo, en 2.1 podemos ver la representación de una celda en una red D3Q19, que divide un dominio tridimensional en celdas con 19 partículas ficticias.

2.4. Tecnologías utilizadas

Para llevar a cabo las distintas tareas necesarias, como el uso de aceleradores hardware o la comunicación entre procesos, las aplicaciones y modelos de lenguaje que se van a estudiar se apoyan en algunas bibliotecas de más bajo nivel que se comentan a continuación.

CUDA

CUDA (Compute Unified Device Architecture) es una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA. Permite a los desarrolladores utilizar la potencia de procesamiento de las unidades de procesamiento gráfico (GPU) para realizar cálculos complejos y tareas de computación intensiva, como el procesa-

miento de imágenes, simulaciones físicas y aprendizaje automático. A través de CUDA, los programadores pueden escribir código en lenguajes como C, C++ y Fortran, facilitando la ejecución de algoritmos en paralelo y aprovechando la arquitectura de las GPU para mejorar el rendimiento en comparación con las CPU tradicionales. Esto ha llevado a un avance significativo en diversas áreas, desde la investigación científica hasta el desarrollo de aplicaciones de inteligencia artificial.

HIP

HIP (Heterogeneous-computing Interface for Portability), desarrollada por AMD, es una interfaz de programación y un lenguaje de *kernels* que permite a los desarrolladores escribir código en C++ que puede ejecutarse en diferentes arquitecturas de hardware, como CPU y GPU, de manera portátil. Permite utilizar dispositivos tanto de AMD como de NVIDIA mediante ana API ligera que tene un impacto escaso o nulo en el rendimiento. HIP proporciona una interfaz similar a CUDA, lo que facilita la migración de aplicaciones y algoritmos diseñados originalmente para CUDA a entornos que utilizan hardware de AMD. Al permitir la escritura de código en C++ y ofrecer herramientas para la gestión de memoria y la ejecución de kernels en paralelo, HIP busca simplificar el desarrollo de aplicaciones de alto rendimiento en sistemas heterogéneos, promoviendo la interoperabilidad y la flexibilidad en el uso de diferentes dispositivos de procesamiento.

OpenMP

OpenMP (Open Multi-Processing) es una API que permite la programación paralela en sistemas de memoria compartida. Diseñada para C, C++ y Fortran, OpenMP proporciona un conjunto de directivas, bibliotecas y variables de entorno que facilitan la creación de aplicaciones que pueden ejecutar múltiples hilos de manera paralela. A través de la inclusión de directivas en el código, los desarrolladores pueden especificar qué partes del programa deben ejecutarse en paralelo, lo que permite una fácil escalabilidad y mejora del rendimiento en arquitecturas multi núcleo. OpenMP es ampliamente utilizado en aplicaciones científicas y de ingeniería, donde la paralelización de tareas puede resultar en una significativa reducción del tiempo de ejecución.

MPI

MPI (Message Passing Interface) es un estándar de comunicación diseñado para la programación paralela en sistemas de computación distribuida. Permite que múltiples procesos, que pueden estar ejecutándose en diferentes nodos de un clúster o en una red de computadoras, se comuniquen entre sí mediante el intercambio de mensajes. MPI proporciona un conjunto de funciones y protocolos que facilitan la sincronización, la comunicación y la gestión de datos entre procesos, lo que es esencial para el desarrollo de aplicaciones de alto rendimiento en entornos de supercomputación. Es ampliamente utilizado en áreas como la simulación científica, el modelado y la computación de alto

rendimiento, permitiendo a los desarrolladores escalar sus aplicaciones de manera eficiente en sistemas con múltiples procesadores y nodos.

2.5. Herramientas

Para facilitar el desarrollo de aplicaciones eficientes en entornos HPC, el grupo Trasgo ha desarrollado previamente varias bibliotecas de funciones escritas en C11, compatibles con cualquier compilador moderno de C/C++. Las aplicaciones desarrolladas sobre estas bibliotecas se despliegan y adaptan en entornos distribuidos con configuraciones de hardware diversas tanto homogéneas como heterogéneas. Por tanto, proporcionan una capa de abstracción que simplifica el desarrollo y el despliegue en múltiples tipos de entornos: desde contextos de pequeña escala como sistemas empotrados con hardware heterogéneo, hasta clústeres exaescala con arquitecturas uniformes que incluyen aceleradores.

Hitmap

Hitmap [5] facilita las tareas relacionadas con la partición jerárquica de arrays multidimensionales u otras estructuras de datos con dominios tanto densos como dispersos. Se encarga del particionado de los datos en *tiles* (teselas), del mapeado de los mismos a los procesos y la comunicación y sincronización entre ellos. Implementa un sistema de plug-ins para las políticas de partición y mapeo. Las comunicaciones se expresan en función de los objetos resultantes. Por tanto, el código escrito con Hitmap es independiente de la forma en que se distribuye el trabajo. Permite también agrupar conjuntos de acciones de comunicación en patrones completos que se pueden reutilizar en diferentes puntos del programa. Hitmap está construida sobre MPI. Se ha mostrado con diversas aplicaciones que Hitmap puede ser tan eficiente como la implementación manual en MPI, reduciendo el esfuerzo de programación.

Controller

Controller [14, 20] es una biblioteca de funciones. Implementa el concepto de controlador, una entidad abstracta que se encarga de gestionar, de forma transparente para el programador, la memoria y el lanzamiento de tareas en un dispositivo de cómputo. Un dispositivo puede ser desde un conjunto de núcleos de CPU hasta un acelerador hardware como una GPU o FPGA. Al utilizar unas abstracciones comunes para todos los dispositivos reduce el esfuerzo de programación. Controller implementa diferentes backends o sistemas de ejecución que se encargan de enlazar el código del usuario con las tecnologías específicas para el manejo del dispositivo correspondiente. Por ejemplo, OpenMP para conjuntos de núcleos o procesadores CPU; CUDA, HIP u OpenCL para tarjetas gráficas de NVIDIA y AMD; y OpenCL para FPGAs.

Controller ofrece varias abstracciones. Tiene un mecanismo para definir kernels específicos para diferentes dispositivos con el mismo nombre y enlazarlos en un programa (*fat binary*). El programa escoge la implementación apropiada para cada dispositivo en

el momento de la ejecución. También permite definir kernels genéricos independientes de detalles del modelo de programación. Se definen una única vez y se reutilizan en cualquier dispositivo. Los kernels y operaciones de cómputo se lanzan de forma asíncrona. El sistema analiza las dependencias de datos entre los argumentos de los kernels y se ocupa de forma transparente de mantener la consistencia de memoria entre los diferentes espacios de memoria y de mantener la consistencia secuencial con las sincronizaciones necesarias. Además, ofrece una abstracción para indexar de forma unificada un espacio abstracto de hilos de grano fino que se adapta a la arquitectura del dispositivo, junto con una interfaz para acceder a las estructuras de datos en función de los índices de los hilos que unifica los criterios de coalescencia y explotación de memorias caché tanto en CPUs como en aceleradores.

EPSILOD

EPSILOD [2] es una herramienta para automatizar la creación y ejecución de aplicaciones ISL. Se implementa como un esqueleto paralelo. Un esqueleto paralelo [6] es un constructo de programación en forma de función de segundo orden que abstrae los detalles de un patrón de computación e interacción paralelos. El esqueleto paralelo definido en EPSILOD recibe como argumentos el tamaño del espacio de datos a partir y computar, una especificación del patrón u operador de *stencil* y la función que calcula o actualiza cada celda. EPSILOD utiliza mecanismos tanto de Controller como de Hitmap. Internamente, aplica una partición de datos por bloques y con el resultado de la misma calcula los tamaños y posiciones de los bordes y halos necesarios en cada dispositivo. En cada iteración lanza kernels para calcular de forma separada los valores de cada borde. Cuando se detecta que los valores en un borde ya se han calculado realiza automáticamente las comunicaciones asíncronas necesarias para copiar esos datos en los halos correspondientes de otro dispositivo. De esta forma se maximizan las oportunidades para solapar estas comunicaciones con el cómputo de la parte principal o central de los datos asignados al dispositivo. Los resultados experimentales presentados en la literatura sobre la primera versión de EPSILOD indican que permite obtener una buena escalabilidad fuerte y débil tanto para sistemas homogéneos como heterogéneos, con mejor rendimiento que utilizando las versiones disponibles en aquel momento de herramientas de comunicación abstracta como Celerity [3].

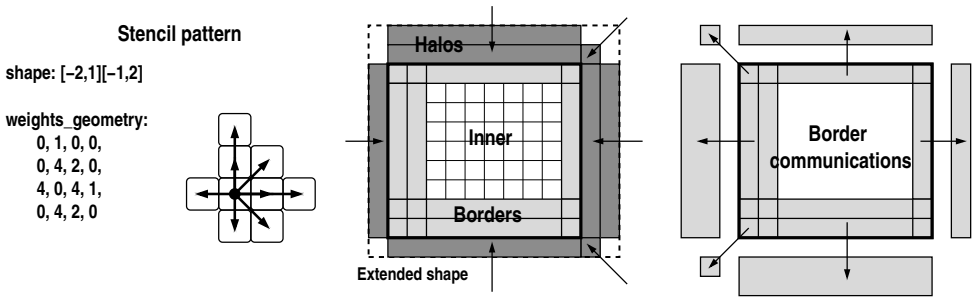


Figura 2.2: Izquierda: Definición de un patrón de estencil 2D asimétrico irregular y su representación gráfica. Centro: Las diferentes partes y la forma extendida calculada para la partición local de un array. Derecha: Diferentes instancias de sub-particiones de los bordes a comunicar a los preocesos vecinos [2]

3: Trabajos relacionados

3.1. Estado del arte

Se pueden encontrar en la literatura diversos modelos o herramientas de programación para desarrollar y ejecutar aplicaciones ISL en sistemas heterogéneos y distribuidos. Estas soluciones presentan abstracciones para ocultar los detalles del manejo de dispositivos heterogéneos y/o aceleradores y los detalles de la distribución y comunicación de datos.

Existen diversos modelos y herramientas de programación que permiten trabajar con códigos ISL en sistemas con GPUs [16, 21, 13, 12, 18, 11, 9]. Algunas están basadas en esqueletos o patrones paralelos. Estos permiten explotar características específicas del patrón de cómputo y comunicación del tipo de aplicación. SkelCL [19] o SkePU3 [4] incluyen un esqueleto *MapOverlap* que puede ser utilizado para implementar aplicaciones ISL. Ambos soportan multi-GPU en este esqueleto, pero no multi-nodo con aceleradores. Lo mismo sucede con el patrón *iterative stencil + reduce* en Fastflow [1]. Otras bibliotecas de esqueletos como Muesli [15] soportan multi-GPU y multi-nodo para aplicaciones ISL, con optimizaciones específicas para dominios complejos [7]. Muesli además da soporte a aplicaciones tipo LBM [10] mediante el uso de estructuras de datos como elementos del dominio. Este tipo de aplicaciones son de especial interés por las oportunidades de optimización que presentan.

Todas estas soluciones no implementan mecanismos que exploten completamente el solapamiento de cómputo y comunicaciones, tanto entre dispositivos y nodo anfitrión como entre nodos. Estas soluciones escalan cuando el ratio entre cómputo y comunicación es lo suficientemente alto como para que las comunicaciones no sean muy significativas. Sin embargo, su coste afecta a la escalabilidad en escenarios comunes en los que el ratio de comunicación con respecto al cómputo crece con el número de dispositivos o nodos, siendo especialmente notable en aplicaciones ISL complejas con alto volumen de comunicación. Además, estos costes de comunicación se revelan ineludiblemente en estudios de escalabilidad fuerte cuando la cantidad de cómputo por nodo se reduce lo suficiente.

Otra solución para implementar aplicaciones ISL es utilizar una herramienta suficientemente abstracta aunque no especifica para esta clase de problemas. Por ejemplo,

Celerity [3] utiliza como capa de portabilidad para dispositivos heterogéneos SYCL [8]. Celerity añade por encima un sistema abstracto para expresar distribuciones de datos y dependencias, que genera internamente las comunicaciones en MPI solapando parte de las transferencias de datos con el cómputo.

Finalmente, EPSILOD [2], la herramienta que nos ocupa en este trabajo, se construye sobre una capa de portabilidad para sistemas heterogéneos denominada Controller [14, 20], integrada con una capa de gestión de comunicaciones en sistemas distribuidos denominada Hitmap [5]. Presenta un esqueleto paralelo con optimizaciones para aplicaciones ISL que aprovechan posibilidades más sofisticadas de solapamiento de cómputo y comunicaciones, lo que facilita la escalabilidad de las soluciones.

4: Mejoras y optimizaciones en EPSILOD

En este capítulo se describen los avances introducidos en EPSILOD con el objetivo de: (1) Mejorar su capacidad para implementar nuevos tipos de aplicaciones ISL; (2) Aumentar su flexibilidad para experimentar con diferentes políticas y mecanismos de partición y mapeo; (3) Mejorar su rendimiento y escalabilidad.

4.1. Datos genéricos

La versión previa de EPSILOD solo soportaba estructuras de datos con elementos de un tipo nativo del lenguaje de programación C. Para cambiar el tipo nativo por otro la biblioteca debía ser recompilada. Con el objetivo de poder implementar una mayor variedad de aplicaciones, se ha adaptado EPSILOD permitiendo el uso de tipos de datos genéricos. Los elementos de las estructuras de datos que representan el dominio del problema se pueden declarar ahora con un tipo de datos arbitrario. Permite tipos de datos definidos por el usuario, incluyendo estructuras complejas. En el fichero de configuración de cada aplicación se especifica el tipo de datos elemental deseado. De forma automática se compila una versión de EPSILOD por cada tipo de datos especificado en alguna aplicación, permitiendo al compilador realizar optimizaciones automáticas específicas para cada tipo. Cada aplicación se enlaza con la versión de EPSILOD correspondiente. En el listado 4.1 se puede ver un ejemplo de cómo declarar y utilizar un tipo de datos definido por el usuario (`cell_t`). Cada aplicación utiliza la macro `EPSILOD_BASE_TYPE` para declarar su tipo de datos. Si no se especifica, el tipo por defecto es `float`.

4.2. Mejoras en la descomposición del dominio

La descomposición del dominio para la paralelización de los problemas de tipo ISL implica la creación de un patrón de comunicación de datos entre los procesos que realizan computación sobre subdominios vecinos. La cantidad de datos en los bordes y halos supone

Listing 4.1: Definición del ejemplo y uso de un tipo de datos definido por el usuario en EPSILOD.

```
// CMakelist
add_epsilod_version( cell_t "gassimulation.h" )
add_app( gassimulation cell_t "gassimulation.h" "-lm" )
----------------------
// Header gassimulation.h
typedef struct {
  float data[19];
} cell_t;
----------------------
// Kernel gassimulation_kernel.c
#include "epsilod.h"

CTRL_KERNEL(updateCell, GENERIC, DEFAULT,
  KHitTile_cell_t matrix,
  const KHitTile_cell_t matrixCopy, ... , {
  ...
});
----------------------
// Main gassimulation.c
#include "epsilod.h"
REGISTER_STENCIL(updateCell, GENERIC, DEFAULT);
...
int main(int argc, char *argv[]) {
  ...
  stencilComputation(...); // EPSILOD skeleton call
  ...
}
```

un sobrecoste de la paralelización de este tipo de algoritmos. Por un lado, se trata de un sobrecoste de memoria, ya que cada proceso tiene que alojar estos datos adicionales. Además, implica un sobrecoste de tiempo de ejecución, ya que esos datos se tienen que intercambiar entre los distintos procesos. Este coste es proporcional al volumen de datos y su disposición en memoria (contiguos o dispersos), por lo que reducir dicha cantidad y optimizar su localización en memoria es de gran importancia.

Una forma de optimizar la cantidad de datos que intervienen en las comunicaciones consiste en una elección adecuada de la política de descomposición del dominio [17]. La cantidad de opciones y su complejidad aumenta con el número de dimensiones del problema. Para un dominio de dos dimensiones, la ecuación 4.4 expresa el tamaño de los halos al partir en una dimensión y la 4.5 al partir por dos dimensiones.

$$S_{h,1d} = 4n \tag{4.4}$$

$$S_{h,2d} = 8\frac{n}{\sqrt{p}} \tag{4.5}$$

Donde $S_{h,1d}$ y $S_{h,2d}$ es el tamaño de los halos, n, el tamaño del lado de la matriz y p, el número de procesos. Estas fórmulas se obtienen al multiplicar el número de comunicaciones que hace cada proceso (envíos y recepciones) por el tamaño de las mismas. A partir de cierto número de procesos, partir en 2 dimensiones resulta en un menor tamaño de los

halos. Esto se puede extrapolar a un mayor número de dimensiones, tanto del dominio como de particionado. Así, en el caso de problemas con tres dimensiones, al partir la matriz en subdominios en una dimensión el volumen de los halos crece notablemente más rápido que si se descompone en dos dimensiones. A partir de las fórmulas 4.4 y 4.5, se ha calculado como evoluciona la proporción entre el número de datos a comunicar y el número de datos que intervienen en el cómputo por cada proceso. Este resultado se muestra en la figura 4.3. Podemos observar que en la partición en 1 dimensión la ratio crece rápidamente hasta alcanzar el 56 %. Partir en 2 dimensiones tiene un crecimiento lento en comparación y partir en el total de las dimensiones genera una ratio constante. Estas 2 presentan ratios de 3.5 % y 1.2 % respectivamente. El hecho de que la proporción entre volumen de datos comunicaciones y cómputo se mantenga constante parece indicar que es el camino a seguir para que las aplicaciones sean escalables en un gran número de nodos. No obstante, esto solo tiene en cuenta el volumen de datos, por lo que es necesario obtener resultados experimentales para ver como influyen en los tiempos de ejecución para cada tipo de partición otros factores como la dispersión de los datos en memoria o las latencias en la transmisión en diferentes escenarios.

La primera versión de EPSILOD no permitía al usuario cambiar la política de partición directamente. Era necesario cambiar en el código de EPSILOD las llamadas a Hitmap para escoger una topología virtual para los procesos y una política de descomposición y partición sobre esa topología, recompilando la herramienta. Para facilitar el experimentar con diferentes políticas se ha propuesto flexibilizar la herramienta introduciendo un parámetro en forma de variable de entorno que permite al usuario escoger la política de partición en el momento del lanzamiento con un único argumento. Se han diseñado una colección de políticas que expresan de forma más sencilla combinaciones de topologías virtuales y distribuciones de datos de las disponibles en el sistema de plug-ins de Hitmap. También se ha construido un nuevo plug-in de Hitmap que mejora el equilibrio de la cantidad de procesos en cada dimensión de una tipología multidimensional.

4.3. Optimización de las comunicaciones

Se propone reducir la complejidad y tiempo de las comunicaciones entre iteraciones de dos formas. La primera consiste en detectar los halos correspondientes a límites del dominio para evitar transferencias innecesarias al dispositivo. En la versión original de EPSILOD se transferían desde el espacio de memoria del proceso anfitrión hacia el dispositivo todos los halos. En EPSILOD los procesos que gestionan los dispositivos se organizan en una topología virtual que puede ser uni-dimensional o multidimensional. Cuántas más dimensiones se usan más procesos tienen algún borde en el límite del dominio. Estos bordes no se comunican y los correspondientes halos no reciben nada.

La segunda optimización está relacionada con el solapamiento del cómputo y las comunicaciones. Durante la experimentación con la versión original de EPSILOD en clústers de supercomputación con dispositivos GPU, se observa que en algunas configuraciones las comunicaciones predominan sobre el cómputo. De esta forma, aparecen períodos en los

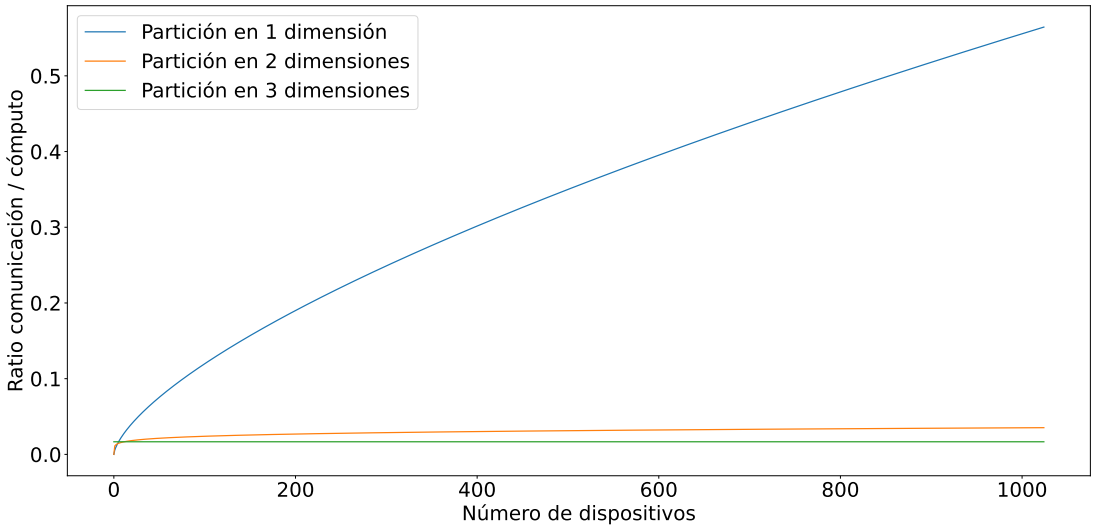
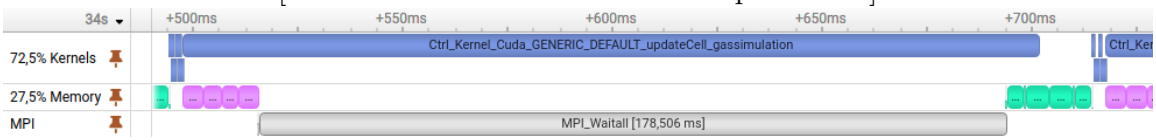


Figura 4.3: Cálculo teórico en un estudio de escalabilidad débil de la proporción, por proceso, entre el volumen de datos en comunicaciones y cómputo, cuando se utilizan particiones de 1, 2 y 3 dimensiones sobre una estructura de datos de tres dimensiones. En este ejemplo cada dispositivo computa 720^3 elementos.

que los dispositivos están ociosos. Además, observamos que el patrón de comunicaciones espera a todas las transferencias de datos entre procesos antes de iniciar las transferencias a la memoria de la GPU correspondiente. Se puede observar este efecto en la imagen del profiler gráfico presentado en la figura 4.3. Para reducir el tiempo empleado en las comunicaciones hemos cambiado el patrón para ir detectando la finalización de cada transmisión asíncrona de datos pertenecientes a un halo, e iniciar la transferencia a la GPU pertinente inmediatamente. Como se puede observar en la figura 4.3, en el caso medio, todas las transferencias de memoria hacia los dispositivos menos la última quedan solapadas con las esperas por otras comunicaciones entre procesos. El tiempo total que tardan las comunicaciones y transferencias con el dispositivo se reduce.

[Patrón de comunicaciones con espera total]



[Patrón de comunicaciones con espera parcial]

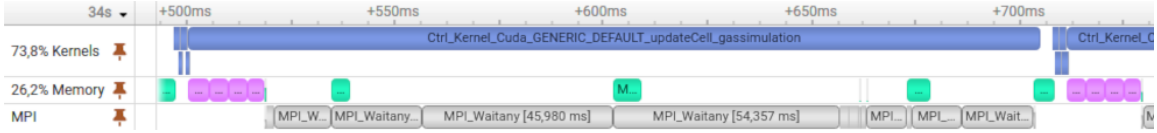


Figura 4.4: Línea temporal de dos sesiones de *profiling* del ejemplo *gas simulation*

4.4. Optimizaciones en la gestión de eventos

Cuando el tamaño de datos con el que trabaja cada proceso es suficientemente grande, las operaciones de bajo nivel, tales como encolar kernels o realizar sincronizaciones con eventos en el sistema de bajo nivel (CUDA, HIP, etc.) quedan ocultas por los tiempos de ejecución de los kernels y/o las comunicaciones. No obstante, cuando la cantidad de datos sobre la que trabaja cada proceso se reduce, como es el caso en las pruebas de escalabilidad fuerte, estas operaciones pueden tomar un papel protagonista y llegar a convertirse en el camino crítico de la ejecución del programa. EPSILOD utiliza la biblioteca Controller como capa de portabilidad para gestionar los dispositivos de cómputo heterogéneos de forma transparente. Controller utiliza un sistema de gestión de eventos genérico para sincronizar la ejecución de cómputo, las transferencias de memoria y las operaciones con el anfitrión y otros dispositivos de cualquier tipo [20]. Para mejorar el rendimiento de EPSILOD en los escenarios en los que los tiempos de la gestión de las operaciones de bajo nivel son relevantes, se proponen dos optimizaciones en los sistemas de gestión de eventos en Controller. Primero, minimizar la cantidad de operaciones con eventos que se utilizan para sincronizar los patrones de lanzamiento de kernels y transferencias de memoria de EPSILOD. Segundo, mover al mismo hilo de ejecución las operaciones de que realizan escrituras (creación, destrucción y grabado) utilizando tecnologías de bajo nivel como CUDA, HIP, etc. Los drivers de estas tecnologías, cuando detectan operaciones de escritura en eventos desde diversos hilos, utilizan automáticamente locks y mutexes para crear regiones críticas y resolver potenciales problemas de concurrencia. Invocar las funciones de gestión de eventos desde un único hilo elimina la necesidad de utilizar esas costosas operaciones de sincronización en el driver, reduciendo el sobrecoste de la gestión de eventos.

5: Estudio experimental

Realizamos un estudio experimental para mostrar el comportamiento de EPSILOD en sistemas de supercomputación de gran escala. El diseño de los experimentos contempla estudios de escalabilidad fuerte y débil. También incluimos comparaciones de rendimiento con otras soluciones del estado del arte que permiten construir aplicaciones ISL en el mismo entorno con alto nivel de abstracción: (1) Muesli [15], una herramienta basada en esqueletos paralelos que incluye uno específico para aplicaciones ISL de hasta 3 dimensiones que utiliza CUDA para trabajar con GPUs de NVIDIA; (2) Celerity [3], un mecanismo abstracto para construir aplicaciones con comunicaciones genéricas en sistemas distribuidos con aceleradores, que utiliza SYCL [8] como capa de portabilidad. Incluimos un experimento de escalabilidad fuerte en comparación la versión anterior de EPSILOD, empleando una aplicación con un stencil básico de tipo 2d4 soportado por esta versión. Diseñamos diversos experimentos para poner de manifiesto la influencia de distintos métodos de partición e implementación en dominios con distinto número de dimensiones.

5.1. Aplicaciones utilizadas

Para probar la escalabilidad de EPSILOD y compararla con otras herramientas del estado del arte como Muesli y Celerity, hemos seleccionado dos aplicaciones que se encuentran en los ejemplos provistos con estas herramientas. Estos ejemplos se encuentran ya desarrollados y han sido verificados en los modelos con los que se han desarrollado originalmente. Por un lado, de Muesli hemos seleccionado uno de sus ejemplos en tres dimensiones denominado *gas simulation*. De Celerity hemos seleccionado el ejemplo que implementa una aplicación ISL en dos dimensiones, denominado *wave simulation*.

Gas simulation es una simulación de fluidos basada en los métodos de red de Boltzmann (LBM) [10]. La implementación de Muesli tiene la particularidad de usar un tipo compuesto como valor de cada celda de un dominio tridimensional. Esto nos permite probar el nuevo soporte de EPSILOD para tipos genéricos de datos. Además, este tipo de datos compuesto, implica comunicaciones más costosas con respecto a otros ejemplos. Por su parte, *Wave simulation* tiene un operador de *stencil* clásico en dos dimensiones con cinco puntas (usa

los valores de la propia celda y de sus cuatro vecinas). En la literatura de EPSILOD se denomina 2d5. Pero tiene la peculiaridad de emplear también su propio valor calculado dos iteraciones atrás.

A la hora de adaptar estas aplicaciones a EPSILOD, el objetivo es hacer una traducción directa de los códigos originales. Hemos puesto especial atención a la traducción de los kernels de cómputo sin introducir ningún tipo de optimización o mejora. En el código de los kernels originales simplemente se han cambiado los accesos a las estructuras de datos para usar las llamadas a Hitmap equivalentes. Se han cambiado los prototipos de los kernels y se han adaptado las funciones de inicialización y salida de resultados.

Gas simulation

Gas simulation trabaja en un dominio de 3 dimensiones con un modelo D3Q19. La implementación de Muesli representa la red como una matriz de celdas. Cada celda está definida como una estructura que contiene un array de 19 elementos.

La ecuación 2.3 se puede resolver en dos pasos: streaming y colisión. La parte izquierda de la ecuación representa el proceso de streaming por el que las partículas se desplazan a lo largo de las distintas direcciones de la red. La parte derecha resuelve el término de colisión. En el listing 5.4 podemos ver el kernel de Muesli que se aplica a cada celda de la red y resuelve esta ecuación en los dos pasos en cada una de las direcciones de la celda.

El kernel de Muesli aplica condiciones de frontera de Dirichlet en las celdas marcadas como `FLAG_KEEP_VELOCITY`. También permite marcar celdas sólidas con las que colisiona el fluido mediante `FLAG_OBSTACLE`.

Como hemos comentado, el objetivo es hacer una traducción directa de los códigos. En este caso, el punto de fricción son las operaciones con vectores. Muesli está escrito en C++ y, para implementar estas operaciones, utiliza la sobrecarga de operadores (listing 5.3). El lenguaje C no presenta esta funcionalidad, así que hemos definido las operaciones con vectores mediante macros (listing 5.2). Están escritas de tal forma que se mantiene la misma asociatividad entre operaciones que en Muesli. Esto evita diferencias en los resultados entre los dos programas, ya que el orden de las operaciones puede alterarlos. El uso de estas macros genera un código más extenso que el uso de los operadores típicos de suma y multiplicación para la suma, escalado y producto escalar de vectores. El kernel del listing 5.5 muestra el uso de estas macros.

Wave simulation

Este ejemplo simula la propagación de ondas en un espacio bidimensional. Para ello resuelve una ecuación diferencial parcial utilizando el método de diferencias finitas. Esto se puede implementar mediante una aplicación ISL.

El valor de la celda (i, j) en el instante de tiempo $n + 1$ se calcula mediante la ecuación de diferencias finitas:

Listing 5.2: Operaciones con vectores de *Gas simulation* en EPSILOD

```
// Escala un vector almacenando el resultado en otra variable
// Equivalente en C++ a: v_out = v * scale;
#define VEC3_SCALE(v_out, v, scale) \
{                                   \
  v_out.x = v.x * scale;            \
  v_out.y = v.y * scale;            \
  v_out.z = v.z * scale;            \
}

// Suma dos vectores almacenando el resultado en el primero
// Equivalente en C++ a: v1 += v2;
#define VEC3_ADD(v1, v2) \
{                        \
  v1.x += v2.x;          \
  v1.y += v2.y;          \
  v1.z += v2.z;          \
}

// Producto escalar de dos vectores
// Equivalente en C++ a la expresion: (v1 * v2)
#define VEC3_DOT(v1, v2) (v1.x * v2.x + v1.y * v2.y + v1.z * v2.z)
```

$$p_{i,j}^{n+1} = 2 * p_{i,j}^{n} - p_{i,j}^{n-1} + \Delta t^2 v^2 (\frac{p_{i+1,j}^{n} - 2p_{i,j}^{n} + p_{i-1,j}^{n}}{\Delta x^2} + \frac{p_{i,j+1}^{n} - 2p_{i,j}^{n} + p_{i,j-1}^{n}}{\Delta x^2}) \quad (5.6)$$

donde p es la presión, v la velocidad de onda en el medio, x e y las coordenadas cartesianas y t el tiempo.

Como podemos ver en la ecuación, para calcular el estado $n+1$ es necesario el estado n para la celda y sus vecinas y el estado $n-1$ de la propia celda. Esto en una aplicación ISL supone disponer del estado de hasta 2 iteraciones atrás.

Como la implementación ISL de EPSILOD emplea 2 matrices de estado del dominio, estas contienen los estados n y $n-1$ hasta que la matriz con el estado $n-1$ es actualizada con los datos del estado $n+1$ durante el cómputo. Al necesitar solo el estado $n-1$ en la propia celda y no en las vecinas, la implementación actual de EPSILOD permite resolver este tipo de problemas sin cambios significativos en la estructura del cómputo ISL.

El único cambio necesario fue el soporte para inicializar independientemente cada una de las matrices de estado para contar con 2 estados iniciales distintos.

5.2. Plataforma experimental

La experimentación se ha realizado en Leonardo, un supercomputador pre-exaescala al que se ha accedido a través del programa EuroHPC JU. Leonardo está localizado en un centro de datos del Tecnopolo de Bologna en Italia y está gestionado por Cineca. Se ha empleado la partición Booster, dedicada al cómputo con GPUs. Esta partición cuenta con

Listing 5.3: Operaciones con vectores de *Gas simulation* en Muesli

```
template<typename T>
struct vec3 {

    T x, y, z;

    MSL_USERFUNC void operator+=(const vec3<T>& other) {
        x += other.x;
        y += other.y;
        z += other.z;
    }

    MSL_USERFUNC friend vec3<T> operator+(vec3<T> v, const vec3<T>& other) {
        v += other;
        return v;
    }

    MSL_USERFUNC void operator*=(const T& val) {
        x *= val;
        y *= val;
        z *= val;
    }

    MSL_USERFUNC friend vec3<T> operator*(vec3<T> v, const T& val) {
        v *= val;
        return v;
    }

    MSL_USERFUNC friend T operator*(const vec3<T>& v1, const vec3<T>& v2) {
        return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
    }
};
```

3456 nodos, cada uno con 4 GPUs NVIDIA Ampere A100 con 64 GB de memoria y una CPU Intel Xeon Platinum "Ice Lake" de 32 núcleos. El sistema de colas permite ejecutar hasta en 256 nodos simultáneamente en esta partición. Las pruebas en Leonardo se han realizado utilizando GCC 12.2, CUDA 12.3, Open MPI 4.1.6 y Celerity 0.6.0 compilado con Clang y basado en AdaptiveCpp 24.10.0.

5.3. Escenarios de escalabilidad

Hemos realizado pruebas de escalabilidad débil y de escalabilidad fuerte. Para las primeras partimos como referencia de la ejecución con un tamaño fijo para un único dispositivo GPU. El dominio de partida es un cuadrado en la aplicación de dos dimensiones y un cubo en la de tres.

Consideramos dos escenarios de escalado: (1) Prisma: El dominio crece solo en la primera dimensión multiplicando el número de filas por el número de GPUs, formando un rectángulo en dos dimensiones o un prisma cuadrado en tres dimensiones; (2) Politopo regular: El dominio crece en todas las dimensiones por igual (un cuadrado o un cubo repartido entre todas las GPUs), manteniendo un número de elementos aproximadamente igual al de la referencia multiplicado por el número de GPUs. La figura 5.5 muestra un

Listing 5.4: Kernel del ejemplo *Gas simulation* en Muesli definido en la función *update*

```
const float FLAG_KEEP_VELOCITY = INFINITY;
const float FLAG_OBSTACLE      = -INFINITY;
const size_t Q                 = 19;

MSL_USERFUNC cell_t update(const PLCube<cell_t> &plCube, int x, int y, int z) {
  cell_t cell = plCube(x, y, z);

  if (cell[0] == FLAG_KEEP_VELOCITY) {
    return cell;
  }

  // Streaming
  for (size_t i = 1; i < Q; i++) {
    int sx = x + (int) offsets[i].x;
    int sy = y + (int) offsets[i].y;
    int sz = z + (int) offsets[i].z;
    cell[i] = plCube(sx, sy, sz)[i];
  }

  // Obstacle collision reorder
  if (cell[0] == FLAG_OBSTACLE) {
    cell_t cell2 = cell;
    for (size_t i = 1; i < Q; i++) {
      cell[i] = cell2[opposite[i]];
    }
    return cell;
  }

  // Particles collision term
  float p = 0;
  vec3f vp {0, 0, 0};
  for (size_t i = 0; i < Q; i++) {
    p += cell[i];
    vp += offsets[i] * cellwidth * cell[i];
  }
  vec3f v = p == 0 ? vp : vp * (1 / p);
  for (size_t i = 0; i < Q; i++) {
    cell[i] = cell[i] + deltaT / tau * (feq(i, p, v) - cell[i]);
  }
  return cell;
}

// Equilibrium distribution function
MSL_USERFUNC inline float feq(size_t i, float p, const vec3f& v) {
  float wi = wis[i];
  float c = cellwidth;
  float dot = offsets[i] * c * v;
  return wi * p * (
    1 + (1 / (c * c)) * (
      3 * dot + (9 / (2 * c * c)) *
      dot * dot - (3.f / 2) * (v * v)
    )
  );
}
```

Listing 5.5: Kernel del ejemplo *Gas simulation* en EPSILOD

```
#define FLAG_KEEP_VELOCITY INFINITY
#define FLAG_OBSTACLE      -INFINITY
#define Q                  19

CTRL_KERNEL(updateCell_gassimulation, GENERIC, DEFAULT,
  KHitTile_cell_t matrix, const KHitTile_cell_t matrixCopy,
  EpsilodCoords global_coords, KHitTile_float stencil,
  float factor, const Epsilod_ext ext_params, {

  cell_t cell = hit(matrixCopy, x, y, z);
  if ( cell.data[0] != FLAG_KEEP_VELOCITY ) {
    // Streaming
    for( int i = 1; i < Q; i++)
      cell.data[i] = hit(matrixCopy,
              ext_params.offsets[i].x,
              ext_params.offsets[i].y,
              ext_params.offsets[i].z).data[i];
    // Obstacle collision reorder
    if ( cell.data[0] == FLAG_OBSTACLE ) {
      cell_t cell_aux;
      for ( int i; i<Q; i++ )
        cell_aux.data[i] = cell.data[ext_params.opposite[i]];
      cell = cell_aux;
    }
    // Particles collision term
    else {
      float p  = 0;
      vec3f vp = {0};
      for (int i = 0; i < Q; i++) {
        p += cell.data[i];
        vec3f scaled;
        VEC3_SCALE(scaled, ext_params.offsets[i], ext_params.cellwidth)
        VEC3_SCALE(scaled, scaled, cell.data[i])
        VEC3_ADD(vp, scaled)
      }
      vec3f v;
      if (p == 0) v = vp;
      else VEC3_SCALE(v, vp, (1 / p))
      for (size_t i = 0; i < Q; i++) {
        // feq function inline
        float wi = wis[i];
        float c  = cellwidth;
        vec3f scaled;
        VEC3_SCALE(scaled, offsets[i], c)
        float dot = VEC3_DOT(scaled, v);
        float feq = wi * p * (
          1 + (1 / (c * c)) * (
            3 * dot + (9 / (2 * c * c)) *
            dot * dot - (3.f / 2) * VEC3_DOT(v, v)
          )
        );

        cell.data[i] += ext_params.deltaT / ext_params.tau *
        (feq - cell.data[i]);
      }
    }
    hit(matrix, x, y, z) = cell
  }
}
```

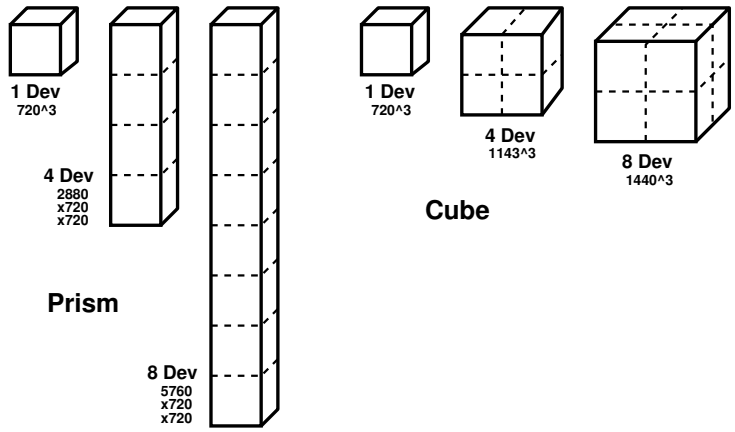


Figura 5.5: Ejemplo de un dominio 3D que crece en los escenarios de prisma y cubo. El ejemplo empieza con un dominio de $720 \times 720 \times 720$ elementos en un dispositivo.

ejemplo de cada escenario en un dominio 3D empezando con un cubo de $720 \times 720 \times 720$ elementos. Para evitar irregularidades en los tamaños de las particiones y el dominio, en los escenarios 2D y 3D experimentamos con números de nodos que son potencia de 2 y 3 respectivamente.

En el escenario de Prisma, para particiones en una sola dimensión, la cantidad de datos a comunicar entre cada par de nodos es independiente del número de GPUs implicadas. La proporción entre comunicaciones y cómputo en cada nodo se mantiene constante. Nos provee de un escenario de control para corroborar que los ejemplos muestran un comportamiento previsible sin sobrecostes ocultos o inesperados que crezcan con el número de dispositivos. En el escenario de politopo regular la matriz mantiene su forma cuadrada o cúbica al escalar. Para particiones en una dimensión esto supone un incremento notable del área de comunicación entre cada par de procesos al escalar. Este incremento es menor en particiones en un mayor número de dimensiones.

Para las pruebas de escalabilidad fuerte utilizamos como referencia la ejecución en un nodo completo con un dominio cuadrado en 2D o cúbico en 3D repartido entre las cuatro GPUs del nodo. Este dominio se reparte con la política seleccionada entre todas las GPUs de los nodos escogidos en cada experimento.

5.4. Tamaños del dominio

Hemos elegido como tamaños de partida los tamaños más grandes posibles de las matrices considerando la memoria disponible de cada GPU y las limitaciones específicas de cada modelo. Hemos buscado que la distribución entre los dispositivos sea uniforme. El número de dispositivos de cómputo también se ha escogido de cara a lograr esta uniformidad.

Para la aplicación *gas simulation*, se han escogido dos posibles tamaños de partida:

- **Caso A:** Tamaño escogido para poder comparar EPSILOD con Muesli. En Muesli el tamaño de dominio global a repartir entre todas las GPUs está restringido por la cantidad de datos que se puede direccionar con un entero con signo de 32 bits. Esto limita el máximo número de nodos a utilizar en la experimentación y/o el tamaño de entrada de la referencia. Además, Muesli impone otra restricción: el tamaño de la matriz en la dimensión z tiene que ser múltiplo del número de procesos (emplea un proceso por nodo) y del número de GPUs por nodo. Esto complica la elección de tamaños de la matriz. En escalabilidad débil ya no puede ser exactamente cúbica para cualquier número de dispositivos si se quiere mantener la uniformidad de la distribución de la matriz entre los elementos de cómputo. En escalabilidad débil el tamaño de partida en una GPU es de 202^3 elementos. Esto supone una ocupación por GPU de unos 1.16 GB de los 64 GB disponibles. En este caso, las limitaciones de Muesli, permiten escalar hasta 256 GPUs en 64 nodos.

  En el caso de escalabilidad fuerte, es posible partir de un tamaño 512^3 en un nodo, con una ocupación de 4.75 GB por GPU. Debido a la restricción del eje z, este tamaño permite escalar hasta 512 GPUs en 128 nodos.

- **Caso B:** Para ver el comportamiento de la aplicación con una mayor ocupación de la GPU con EPSILOD hemos definido otro caso con un mayor tamaño de entrada. En escalabilidad fuerte partimos de un tamaño de 1184^3 en 1 nodo. En estas condiciones, la ocupación de las matrices de cómputo es de unos 52.98 GB de los 64 GB disponibles. Esto es principalmente debido al tamaño adicional que requieren los halos al partir en 1 dimensión. Además es necesario dejar algo de espacio libre para algunas estructuras auxiliares, las estructuras de gestión propias de sistemas subyacentes y los datos de *profiling* si se desea utilizar dicha herramienta. En escalabilidad débil la referencia es una matriz de 720^3 en una GPU. Como hemos comentado previamente, el número de datos a comunicar crece más rápido para particiones en una dimensión. Por ello, y para limitar el tiempo de ejecución de las pruebas y la cuota consumida, hemos limitado el número máximo de nodos a 64 en las pruebas de este caso cuando se parte en una dimensión. En el resto de pruebas de este caso escalamos hasta 1024 GPUs en 256 nodos, el máximo disponible.

En la aplicación *wave simulation* los elementos de las matrices son del tipo nativo *float*. Por tanto, es necesario un número mayor de elementos para conseguir una alta ocupación de las GPUs. En este caso es en EPSILOD donde el tipo de datos empleado para indexar los datos se convierte en un factor limitante, lo que nos lleva a un único caso de experimentación:

- **Caso C:** En escalabilidad fuerte la matriz cuadrada de referencia en un nodo tiene 90112^2 elementos. En escalabilidad débil la matriz cuadrada de referencia en 1 GPU tiene 45056^2 elementos. En ambos casos podemos escalar hasta 1024 GPUs en 256 nodos, el máximo disponible.

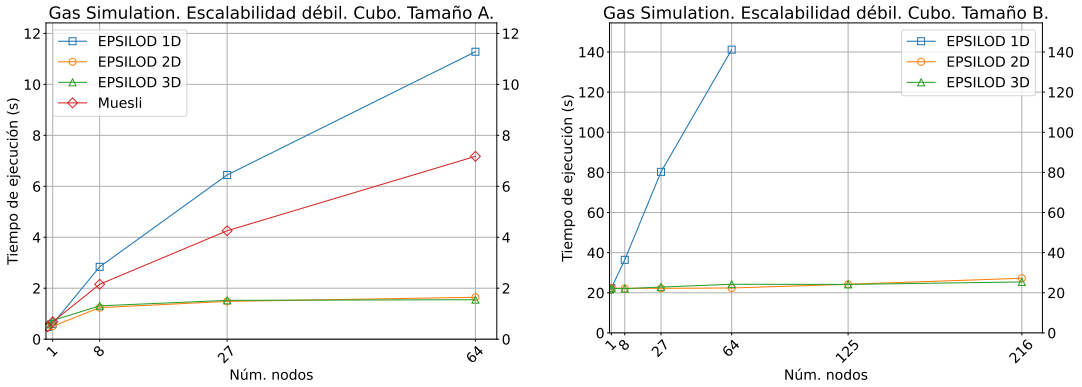


Figura 5.6: Escalabilidad débil. Gas simulation. Escenario politopo regular (cubo). Tamaño Caso A a la izquierda, tamaño Caso B (solo soportado por EPSILOD) a la derecha. En EPSILOD *1D* indica partición solo en la primera dimensión, *2D* indica partición en las dos primeras dimensiones, *3D* indica partición en las tres dimensiones.

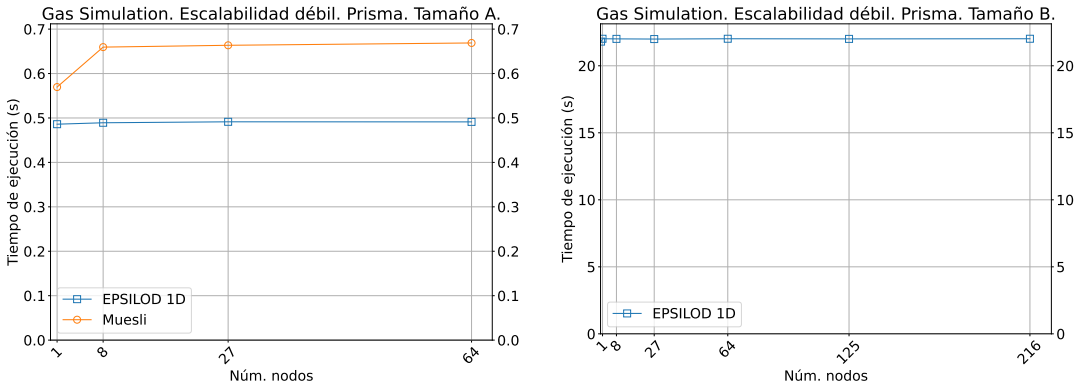


Figura 5.7: Escalabilidad débil. Gas simulation. Escenario prisma. Tamaño Caso A a la izquierda, tamaño Caso B (solo soportado por EPSILOD) a la derecha. En EPSILOD *1D* indica partición en la primera dimensión.

5.5. Resultados de escalabilidad débil

La Fig. 5.6 muestra los resultados de escalabilidad débil para la aplicación *gas simulation* con el escenario en el que el dominio crece como un politopo regular (cubo). En esta aplicación el volumen de comunicación es alto comparado con el cómputo, por lo que las comunicaciones pueden dominar el tiempo de ejecución fácilmente. En Muesli observamos que el tiempo crece rápidamente con la cantidad de nodos. Sus comunicaciones no están solapadas por lo que su efecto es inevitable. Utiliza una partición en una dimensión, pero con un solo proceso por nodo, utilizando comunicaciones más optimizadas entre los dispositivos del mismo nodo. Esto introduce un factor multiplicativo que reduce los tiempos de ejecución comparados con la partición de una dimensión en EPSILOD. Pero

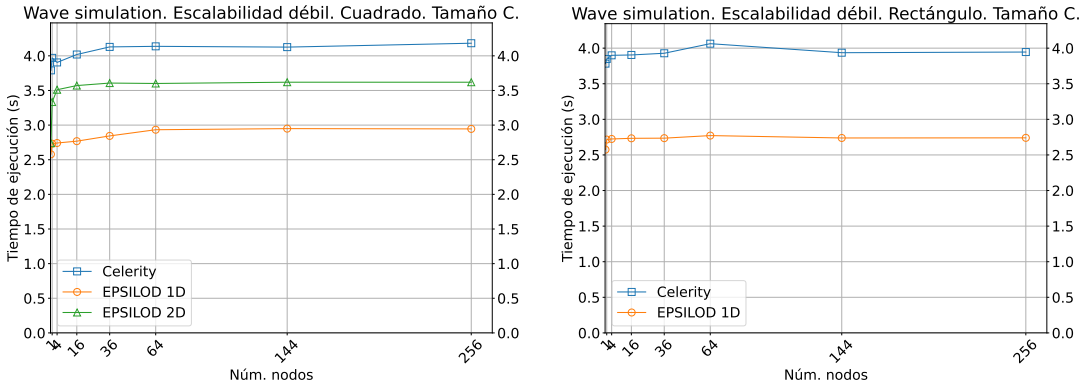


Figura 5.8: Escalabilidad débil. Wave simulation. Escenario politopo regular (cuadrado) a la izquierda. Escenario prisma (rectángulo) a la derecha. En EPSILOD *1D* indica partición solo en la primera dimensión, *2D* indica partición en las dos dimensiones.

no puede evitar la misma tendencia de crecimiento en función del número de nodos. Se observa que en EPSILOD la partición en una dimensión no es adecuada, ya que el volumen de comunicación crece con el número de GPUs. Cuando empleamos particiones en dos y tres dimensiones el crecimiento es mucho más lento. En el Caso B de tamaño grande con una gran ocupación de la memoria de las GPUs con la partición en 2D se observa que hasta 64 nodos las comunicaciones están completamente solapadas y no influyen en el tiempo de ejecución. A partir de ese punto no quedan completamente solapadas y los tiempos de ejecución crecen ligeramente siguiendo la tendencia del crecimiento del volumen de los halos. La partición en 3D muestra tiempos similares o superiores a la 2D hasta 125 nodos. A partir de este punto el tiempo se estabiliza y partir en todas las dimensiones muestra un mejor rendimiento. En el caso del tamaño pequeño soportado por Muesli el cómputo es menor y la tendencia aparece antes. Observamos menos de un 15 % de pérdida de rendimiento en 216 nodos en comparación con el tiempo en una GPU.

En la izquierda de la Fig. 5.8 se muestran los resultados con la aplicación *wave simulation* con el escenario de crecimiento en forma de politopo regular (cuadrado). Aunque en los primeros nodos la introducción de los efectos de la red de interconexión sí incrementan los tiempos, luego se estabilizan, puesto que el volumen de comunicación ya no llega a ser suficiente para hacer crecer los tiempos de forma proporcional. Se observa que Celerity obtiene un peor rendimiento en todos los casos. Un análisis detallado con herramientas de *profiling* muestra que estas pérdidas se deben principalmente a un mayor coste de ejecución de los mismos kernels que se han portado a EPSILOD. Esto se debe a que SYCL y por extensión Celerity utiliza su propio compilador para compilar los kernels para GPU.

En la Fig. 5.7 y en la derecha de la Fig. 5.8 se observa que cuando el dominio crece en forma de prisma en una dimensión, en ambas aplicaciones EPSILOD consigue una alta escalabilidad débil, con pérdidas de menos del 1 % con hasta 864 GPUs en 216 nodos, incluso para el tamaño Caso A en *gas simulation*. En estos casos el volumen de comunicación entre dispositivos vecinos se mantiene independientemente del número de

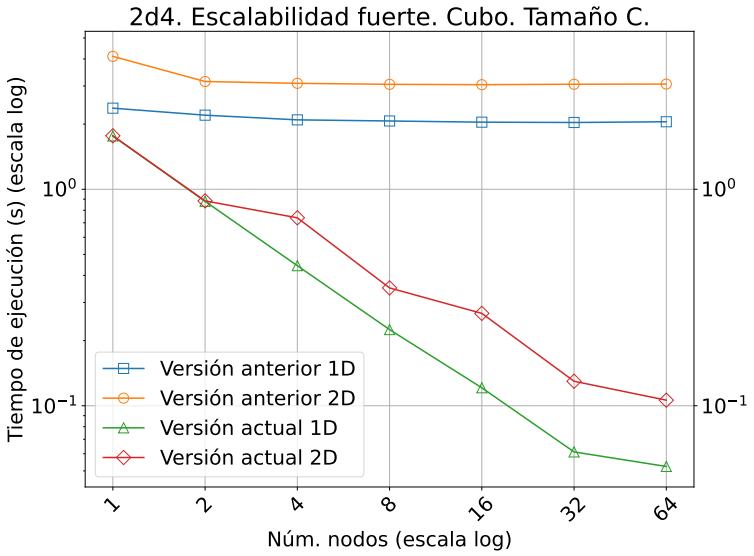


Figura 5.9: Comparación con la versión previa de EPSILOD. Escalabilidad fuerte. Stencil 2d4. Escenario politopo regular (cuadrado). Tamaño Caso C para. *1D* indica partición solo en la primera dimensión, *2D* indica partición en las dos dimensiones.

GPUs. Luego el coste de comunicaciones es constante. Tanto Muesli como Celerity muestran unas tendencias similares aunque con mayores tiempos de ejecución que EPSILOD. Un análisis con herramientas de *profiling* muestra que en Muesli el principal inconveniente es el coste de las comunicaciones que no está solapado con el cómputo, mientras que en Celerity el principal problema es el mayor coste de ejecución de los kernels.

5.6. Resultados de escalabilidad fuerte

En la figura 5.9 observamos los resultados de la comparación con la versión previa de EPSILOD. La versión previa presenta una escalabilidad limitada. La partición en 2 dimensiones escala mejor que la de una, pero se estanca a partir de 8 nodos llegando a un *speedup* de 1.35x. Al partir en 1 dimensión escala hasta 16 nodos pero solo consigue un 1.16x de *speedup*. La nueva versión de EPSILOD presenta mejoría clara en escalabilidad consiguiendo un *speedup* de 28.81x en 32 nodos al partir en 1 dimensión. A partir de ese punto la escalabilidad se reduce y el *speedup* es de 33.63x para 64 nodos. Tanto en la versión previa como en la nueva la partición en 1 dimensión proporciona un resultado mejor en cuanto a tiempo de ejecución, en contradicción con los resultados teóricos respecto al volumen de comunicaciones. Esto nos indica que en este escenario hay otros factores como la dispersión de datos en memoria o el tamaño del dominio que posiblemente tienen una influencia mayor y que habrá que estudiar.

La figura 5.10 muestra los resultados de escalabilidad fuerte para la aplicación *gas simulation*. Como ya vimos anteriormente partir por 2 o 3 dimensiones una matriz de

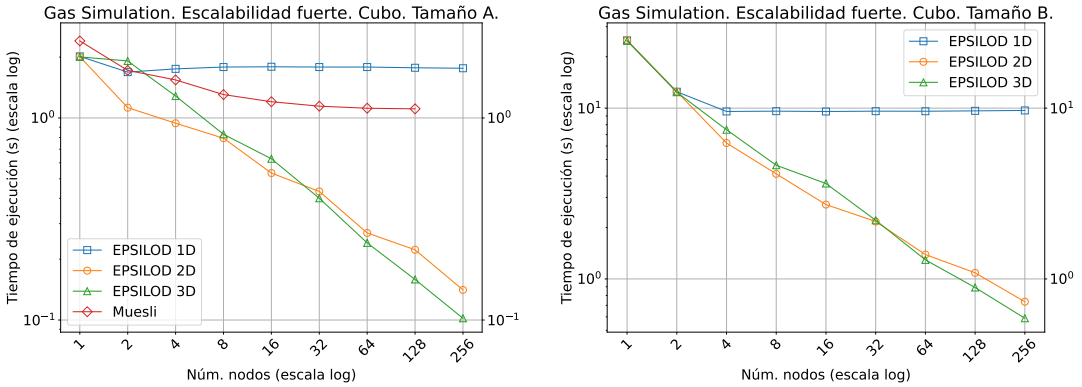


Figura 5.10: Escalabilidad fuerte. Gas simulation. Escenario politopo regular (cubo). Tamaño Caso A a la izquierda, tamaño Caso B (solo soportado por EPSILOD) a la derecha. En EPSILOD *1D* indica partición solo en la primera dimensión, *2D* indica partición en las dos primeras dimensiones, *3D* indica partición en las tres dimensiones.

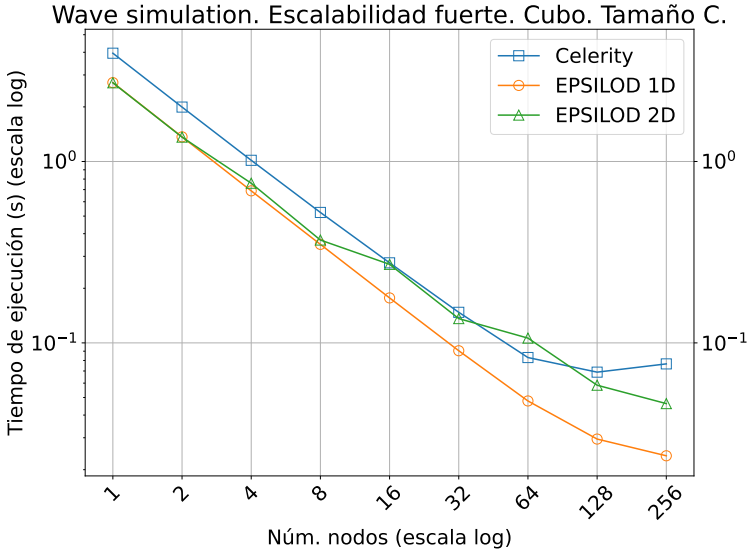


Figura 5.11: Escalabilidad fuerte. Wave simulation. Escenario politopo regular (cuadrado). *1D* indica partición en la primera dimensión. En EPSILOD *1D* indica partición solo en la primera dimensión, *2D* indica partición en las dos dimensiones.

3 resulta en una mejor escalabilidad fuerte que partir por una dimensión. Esta última partición deja de escalar a partir de 2 o 4 nodos dependiendo del tamaño del dominio. En cuanto a la partición en 2 dimensiones, aunque la escalabilidad no es ideal, muestra una tendencia que se mantiene incluso hasta 1024 GPUs en 256 nodos, consiguiendo un *speedup* de 33.90x. La partición en 3 dimensiones muestra un comportamiento peor en los primeros compases, pero mejora significativamente a partir de 32 o 64 nodos. Para el

tamaño B consigue un *speedup* de 42.08x en 256 nodos. En Muesli, la falta de solapamiento de las comunicaciones se revela en cuanto el tamaño del cómputo se reduce al dividirse en un número creciente de GPUs, de forma que no consigue escalar.

La figura 5.11 muestra los resultados de escalabilidad fuerte en *wave simulation*. En EPSILOD la escalabilidad es casi perfecta hasta 256 GPUs en 64 nodos. En 128 y 256 nodos la cantidad de cómputo se reduce de tal forma que las comunicaciones ya no quedan completamente solapadas observándose una reducción de la escalabilidad. En el caso de usar partición en 2 dimensiones los bordes ortogonales a la primera dimensión se componen de elementos no contiguos en memoria. En este caso las operaciones internas para colocar en el buffer y recolocar los datos en sus posiciones en la recepción son más costosas y afectan rápidamente a la escalabilidad. En el caso de la partición en una dimensión la escalabilidad se reduce pero más lentamente, llegando a un 114.07x para 1024 GPUs en 256 nodos. En el caso de Celerity vemos que su sistema de comunicaciones está menos optimizado y a partir de 64 nodos, cuando el coste de las comunicaciones pasa a ser el cuello de botella deja de escalar derivando incluso un incremento en el tiempo de ejecución para 1024 GPUs en 256 nodos.

6: Conclusiones y Líneas de trabajo futuras

Todo proyecto debe incluir las conclusiones que se derivan de su desarrollo. Éstas pueden ser de diferente índole, dependiendo de la tipología del proyecto, pero normalmente van a estar presentes un conjunto de conclusiones relacionadas con los resultados del proyecto y un conjunto de conclusiones técnicas. Además, resulta muy útil realizar un informe crítico indicando cómo se puede mejorar el proyecto, o cómo se puede continuar trabajando en la línea del proyecto realizado.

6.1. Conclusiones

En este trabajo se presentan una serie de cambios de diseño y nuevas técnicas introducidas en EPSILOD, una herramienta para desarrollar y ejecutar aplicaciones ISL (Iterative Stencil Loop) en sistemas heterogéneos distribuidos, para soportar nuevos tipos de aplicaciones, flexibilizar los mecanismos de partición y mejorar su rendimiento y escalabilidad. En concreto se ha introducido un sistema para soportar aplicaciones ISL que utilizan estructuras de datos con elementos de tipos genéricos y arbitrariamente complejos. Se ha añadido un sistema para escoger en tiempo de ejecución la política de partición, con un único argumento que simplifica escoger las combinaciones de topologías y distribuciones de datos. Se ha mejorado el equilibrio de la distribución de procesos en los diferentes ejes de las topologías multidimensionales. Se ha optimizado el modelo de comunicaciones para mejorar el solapamiento entre cómputo y comunicación y mejorar la escalabilidad. Se han simplificado los mecanismos de gestión de eventos en la capa de portabilidad entre dispositivos heterogéneos para eliminar los costes de operación que son especialmente significativos en estudios de escalabilidad fuerte extrema. Se presenta un estudio experimental en un supercomputador pre-exaescala que muestra la escalabilidad de la nueva versión de EPSILOD en escenarios de hasta 1024 GPUs distribuidas en 256 nodos. El estudio utiliza dos aplicaciones de ejemplo de otras herramientas, una con un dominio de dos dimensiones y otra con un dominio de tres dimensiones y un tipo de datos complejo que deriva en un alto coste de comunicaciones. Los resultados muestran un alto

nivel de escalabilidad. Menos de un 1 % de degradación del rendimiento en escalabilidad débil con 1024 GPUs en escenarios de crecimiento del dominio en forma de rectángulo donde las comunicaciones son de coste constante. Menos de un 26 % en escenarios de tres dimensiones con crecimiento en forma de politopo regular (cubo) en el que todas las dimensiones crecen y es conveniente usar políticas de partición en varias dimensiones para contener el crecimiento del volumen de comunicación por dispositivo al escalar. En escalabilidad fuerte utilizamos como referencia el tiempo en un nodo completo. Se muestran resultados para dominios de dos dimensiones con aceleraciones de hasta 56.87x en 64 nodos y 114x en 256 nodos. Y para la aplicación con dominio de 3 dimensiones, tipo de datos complejo y alto volumen de comunicación, se observa una aceleración de 33.58x en 256 nodos. Las comparaciones con otras herramientas del estado del arte que permiten implementar fácilmente aplicaciones ISL distribuidas, como Muesli y Celerity, muestran que EPSILOD presenta mejores propiedades de rendimiento y escalabilidad.

6.2. Trabajo futuro

Como trabajo futuro se plantea completar el estudio experimental con otras técnicas de partición y aplicaciones de tipo *stencil*. También, estudiar nuevas técnicas de optimización de las aplicaciones ISL que reduzcan el ratio de comunicaciones y cómputo, permitiendo un mejor solapamiento y escalabilidad en niveles de escala extremos. Algunas de las técnicas consisten en la transmisión directa de datos entre dispositivos aceleradores del mismo o de distintos nodos. Esto eliminaría mecanismos intermedios como la transferencia de datos al host que son bastante costosos. Los estudios preliminares de este tipo de técnicas parecen indicar una reducción notable en el tiempo de transmisión de datos que resulta prometedora.

Apéndices

Apéndice A

Tablas

A.1. Escalabilidad débil

Politopo regular

Nodos	Tiempo				Speedup			
	Epsilod			Muesli	Epsilod			Muesli
	1D	2D	3D		1D	2D	3D	
0.25	0.48	0.48	0.58	0.49	1.00	1.00	1.00	1.00
1	0.60	0.51	0.73	0.68	0.80	0.93	0.79	0.72
8	2.84	1.24	1.31	2.15	0.17	0.38	0.44	0.23
27	6.44	1.48	1.52	4.25	0.07	0.32	0.38	0.11
64	11.28	1.64	1.55	7.17	0.04	0.29	0.37	0.07

Tabla A.1: Escalabilidad débil. *Gas simulation*. Escenario politopo regular (cubo). Tamaño Caso A. Epsilod y Muesli. Tiempos de ejecución y speedup para las diferentes estrategias de partición

Nodos	Tiempo			Speedup		
	1D	2D	3D	1D	2D	3D
0.25	21.75	21.76	21.70	1.00	1.00	1.00
1	22.38	22.39	22.18	0.97	0.97	0.98
8	36.42	22.22	22.06	0.60	0.98	0.98
27	80.26	22.20	22.85	0.27	0.98	0.95
64	141.17	22.39	24.24	0.15	0.97	0.90
125	-	24.23	24.17	-	0.90	0.90
216	-	27.23	25.40	-	0.80	0.85

Tabla A.2: Escalabilidad débil. *Gas simulation*. Escenario politopo regular (cubo). Tamaño Caso B. Epsilod. Tiempos de ejecución y speedup para las diferentes estrategias de partición

Nodos	Tiempo			Speedup		
	Epsilod		Celerity	Epsilod		Celerity
	1D	2D		1D	2D	
0.25	2.58	2.74	3.79	1.00	1.00	1.00
1	2.73	3.33	3.97	0.94	0.82	0.95
4	2.74	3.51	3.90	0.94	0.78	0.97
16	2.77	3.57	4.02	0.93	0.77	0.94
36	2.84	3.61	4.13	0.91	0.76	0.92
64	2.93	3.60	4.14	0.88	0.76	0.92
144	2.95	3.62	4.13	0.87	0.76	0.92
256	2.95	3.62	4.18	0.88	0.76	0.91

Tabla A.3: Escalabilidad débil. *Wave simulation*. Escenario politopo regular (cuadrado). Tamaño Caso C. Epsilod y Celerity. Tiempos de ejecución y speedup para las diferentes estrategias de partición

Prisma o rectángulo

Nodos	Tiempo		Speedup	
	Epsilod	Muesli	Epsilod	Muesli
1	0.49	0.57	1.00	1.00
8	0.49	0.66	0.99	0.86
27	0.49	0.66	0.99	0.86
64	0.49	0.67	0.99	0.85

Tabla A.4: Escalabilidad débil. *Gas simulation*. Escenario prisma. Tamaño Caso A. Epsilod y Muesli. Tiempos de ejecución y speedup para las diferentes estrategias de partición

Nodos	Tiempo	Speedup 1D
0.25	21.80	1.00
1	22.02	0.99
8	22.01	0.99
27	21.99	0.99
64	22.02	0.99
125	22.01	0.99
216	22.02	0.99

Tabla A.5: Escalabilidad débil. *Gas simulation*. Escenario prisma. Tamaño Caso B. Epsilod. Tiempos de ejecución y speedup para las diferentes estrategias de partición

Nodos	Tiempo		Speedup	
	Epsilod	Celerity	Epsilod	Celerity
0.25	2.57	3.78	1.00	1.00
1	2.72	3.85	0.95	0.98
4	2.72	3.90	0.94	0.97
16	2.73	3.90	0.94	0.97
36	2.74	3.93	0.94	0.96
64	2.77	4.06	0.93	0.93
144	2.74	3.94	0.94	0.96
256	2.74	3.95	0.94	0.96

Tabla A.6: Escalabilidad débil. *Wave simulation*. Escenario prisma. Tamaño Caso C. Epsilod y Celerity. Tiempos de ejecución y speedup para las diferentes estrategias de partición

A.2. Escalabilidad fuerte

Nodos	Tiempo				Speedup			
	Previa		Actual		Previa		Actual	
	1D	2D	1D	2D	1D	2D	1D	2D
1	2.37	4.11	1.76	1.77	1.00	1.00	1.00	1.00
2	2.20	3.15	0.88	0.88	1.08	1.31	2.00	2.00
4	2.09	3.09	0.44	0.74	1.13	1.33	3.97	2.39
8	2.07	3.05	0.22	0.35	1.15	1.35	7.84	5.04
16	2.04	3.04	0.12	0.27	1.16	1.35	14.56	6.61
32	2.03	3.06	0.06	0.13	1.16	1.34	28.81	13.61
64	2.05	3.06	0.05	0.11	1.16	1.34	33.63	16.65

Tabla A.7: Escalabilidad fuerte. Stencil 2d4. Escenario politopo regular (cuadrado). Tamaño Caso C. Comparación con la versión previa de EPSILOD. Tiempos de ejecución y speedup para las diferentes estrategias de partición

Nodos	Tiempo				Speedup			
	Epsilod			Muesli	Epsilod			Muesli
	1D	2D	3D		1D	2D	3D	
1	2.02	2.01	2.01	2.41	1.00	1.00	1.00	1.00
2	1.69	1.12	1.91	1.72	1.20	1.79	1.05	1.40
4	1.75	0.94	1.28	1.54	1.15	2.14	1.56	1.56
8	1.79	0.79	0.83	1.31	1.13	2.53	2.41	1.84
16	1.79	0.53	0.63	1.20	1.13	3.77	3.19	2.00
32	1.79	0.43	0.40	1.14	1.13	4.65	5.01	2.10
64	1.79	0.27	0.24	1.12	1.13	7.45	8.32	2.15
128	1.77	0.22	0.16	1.11	1.14	9.04	12.63	2.17
256	1.76	0.14	0.10	-	1.14	14.27	19.66	-

Tabla A.8: Escalabilidad fuerte. *Gas simulation*. Escenario politopo regular (cubo). Tamaño Caso A. Epsilod y Muesli. Tiempos de ejecución y speedup para las diferentes estrategias de partición

Nodos	Tiempo			Speedup		
	1D	2D	3D	1D	2D	3D
1	24.92	24.93	24.77	1.00	1.00	1.00
2	12.48	12.48	12.39	2.00	2.00	2.00
4	9.56	6.25	7.48	2.61	3.99	3.31
8	9.60	4.12	4.63	2.60	6.05	5.35
16	9.56	2.73	3.62	2.61	9.15	6.84
32	9.60	2.17	2.20	2.60	11.49	11.23
64	9.59	1.39	1.29	2.60	17.95	19.15
128	9.64	1.09	0.89	2.59	22.97	27.80
256	9.69	0.74	0.59	2.57	33.90	42.08

Tabla A.9: Escalabilidad fuerte. *Gas simulation*. Escenario politopo regular (cubo). Tamaño Caso B. Epsilod. Tiempos de ejecución y speedup para las diferentes estrategias de partición

Nodos	Tiempo			Speedup		
	Epsilod		Celerity	Epsilod		Celerity
	1D	2D		1D	2D	
1	2.72	2.72	3.95	1.00	1.00	1.00
2	1.37	1.36	2.00	1.99	1.99	1.98
4	0.69	0.76	1.02	3.95	3.58	3.89
8	0.35	0.37	0.52	7.79	7.36	7.55
16	0.18	0.27	0.28	15.38	10.02	14.31
32	0.09	0.14	0.15	30.07	19.95	26.81
64	0.05	0.11	0.08	56.87	25.56	47.60
128	0.03	0.06	0.07	92.25	46.47	57.31
256	0.02	0.05	0.08	114.07	58.70	51.60

Tabla A.10: Escalabilidad fuerte. *Wave simulation*. Escenario politopo regular (cuadrado). Tamaño Caso C. Epsilod y Celerity. Tiempos de ejecución y speedup para las diferentes estrategias de partición

Apéndice B

Códigos

El código de la nueva versión de EPSILOD junto con las implementaciones de las aplicaciones utilizadas en el estudio experimental está a disposición pública[1].

[1] https://gitlab.com/trasgo-group-valladolid/controllers/-/tree/25_JP_epsilod2?ref_type=tags

Bibliografía

[1] Aldinucci, M., Danelutto, M., Drocco, M., Kilpatrick, P., Misale, C., Peretti Pezzi, G., and Torquati, M. A parallel pattern for iterative stencil + reduce. *The Journal of Supercomputing 74*, 11 (Nov 2018), 5690–5705.

[2] de Castro, M., Santamaria-Valenzuela, I., Torres, Y., Gonzalez-Escribano, A., and Llanos, D. R. Epsilod: efficient parallel skeleton for generic iterative stencil computations in distributed gpus. *The Journal of Supercomputing 79*, 9 (06 2023), 9409–9442.

[3] Distributed and Parallel Systems Group, University of Innsbruck. High-level c++ for accelerator clusters. https://celerity.github.io/.

[4] Ernstsson, A., Ahlqvist, J., Zouzoula, S., and Kessler, C. SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. *International Journal of Parallel Programming 49*, 6 (2021), 846–866.

[5] Gonzalez-Escribano, A., Torres, Y., Fresno, J., and Llanos, D. R. An extensible system for multilevel automatic data partition and mapping. *IEEE Transactions on Parallel and Distributed Systems 25*, 5 (May 2014), 1145–1154.

[6] Gorlatch, S., and Cole, M. *Parallel Skeletons*. Springer US, Boston, MA, 2011, pp. 1417–1422.

[7] Herrmann, N., Dieckmann, J., and Kuchen, H. Optimizing three-dimensional stencil-operations on heterogeneous computing environments. *International Journal of Parallel Programming 52*, 4 (August 2024), 274–297.

[8] Khronos Group. C++ programming for heterogeneous parallel computing. https://www.khronos.org/sycl/.

[9] Kim, H., Hadidi, R., Nai, L., Kim, H., Jayasena, N., E ckert, Y., Kayiran, O., and Loh, G. Coda: Enabling co-location of computation and data for multiple gpu systems. *ACM Trans. Archit. Code Optim. 15*, 3 (Sept. 2018), 32:1–32:23.

[10] Krüger, T., Kusumaatmaja, H., Kuzmin, A., Shardt, O., Silva, G., and Viggen, E. M. The lattice boltzmann method. *Springer International Publishing 10*, 978-3 (2017), 4–15.

[11] Luporini, F., Louboutin, M., Lange, M., Kukreja, N., Witte, P., Hückelheim, J., Yount, C., Kelly, P., Herrmann, F., and Gorman, G. Architecture and performance of devito, a system for automated stencil computation. *ACM Transactions on Mathematical Software 4*, 1 (2020).

[12] Lutz, T., Fensch, C., and Cole, M. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Trans. Archit. Code Optim. 9*, 4 (Jan. 2013), 59:1–59:24.

[13] Maruyama, N., Sato, K., Nomura, T., and Matsuoka, S. Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2011), pp. 1–12.

[14] Moreton–Fernandez, A., Ortega–Arranz, H., and Gonzalez–Escribano, A. Controllers: An abstraction to ease the use of hardware accelerators. *The International Journal of High Performance Computing Applications 32*, 6 (2018), 838–853.

[15] N., H., de Melo Menezes B.A., and H.", K. Stencil calculations with algorithmic skeletons for heterogeneous computing environments. *International Journal of Parallel Programming 50*, 5 (2022), 433–453.

[16] Pereira, A. D., Ramos, L., and Góes, L. F. W. Pskel: A stencil programming framework for cpu-gpu systems. *Concurr. Comput. : Pract. Exper. 27*, 17 (Dec. 2015), 4938–4953.

[17] Quinn, M. J. *Parallel Computing: Theory and Practice*, second ed. McGraw-Hill, New York, NY, 1994.

[18] Shimokawabe, T., Aoki, T., and Onodera, N. A high-productivity framework for multi-gpu computation of mesh-based applications. In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations* (Vienna, Austria, Jan 2014), A. Gr.ºsslinger and H. K.ºstler, Eds., p. 23–30.

[19] Steuwer, M., Haidl, M., Breuer, S., and Gorlatch, S. High-level programming of stencil computations on multi-gpu systems using the skelcl library. *Parallel Processing Letters 24* (2014).

[20] Torres, Y., Andújar, F. J., Gonzalez-Escribano, A., and Llanos, D. R. Supporting efficient overlapping of host-device operations for heterogeneous programming with ctrlevents. *Journal of Parallel and Distributed Computing 179* (2023), 104708.

[21] Viñas, M., Fraguela, B., Andrade, D., and Doallo, R. Facilitating the development of stencil applications using the heterogeneous programming library. *Concurrency Computation 29*, 12 (2017).