Universidad de Valladolid

TRABAJO DE FIN DE MÁSTER
ESCUELA DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN
INGENIERÍA INFORMÁTICA

Reequilibrado automático de carga en supercomputadoras pre-exascala

---

Alumno:
D. Tomás de la Cal Esteban

Tutor:
Dr. Arturo González Escribano

A mi familia, en especial a mi madre Sofía y mi tío Luis Enrique,
A mis amigos, en especial a Hugo y a Monzón.

Agradecimientos

A mi familia, por el apoyo y el cariño que me han dado durante toda la vida.

A mis amigos, por las risas en los buenos momentos y el ánimo en los malos.

A mis compañeros de clase, por haber compartido camino y esfuerzo.

A mis compañeros de trabajo y mis jefes, por la ayuda brindada.

A todos los profesores de la carrera y el máster, por la aportación académica y personal, y a la Universidad.

Y, por supuesto, a mi tutor Arturo, por la orientación, apoyo y comprensión durante el proyecto.

Resumen

La computación paralela es un paradigma de cómputo extremadamente importante basado en el uso de máquinas con múltiples elementos de cómputo y la capacidad de ciertas tareas de subdividirse en trabajos de menor tamaño que se pueden ejecutar de forma simultánea. En el campo de HPC (*High Performance Computing*) se utilizan clústeres con cantidades masivas de equipos interconectados por redes de alto rendimiento. La computación paralela heterogénea es una subclase de la anterior en la que en los nodos de una máquina se utilizan dispositivos de distinta naturaleza adaptados a distintas tareas.

Hoy en día, las grandes supercomputadoras pre-exaescala europeas se construyen con miles de nodos idénticos, que incluyen GPUs (*Graphics Processing Units*). Los nodos con GPUs se manejan con técnicas de programación heterogénea. En un clúster, las GPUs operan simultáneamente con las CPUs que coordinan el trabajo de los procesos o la comunicación entre nodos, siendo necesario combinar diferentes modelos de programación de ambos tipos de dispositivos.

Las aplicaciones ISL (*Iterative Stencil Loop*) representan una clase de soluciones importantes en el campo de la simulación de magnitudes físicas. Están basadas en un proceso iterativo, altamente paralelizable, que requiere gran cantidad de cómputo.

El grupo TRASGO de la universidad de Valladolid ha desarrollado una serie de bibliotecas de funciones que permiten la ejecución de este tipo de aplicaciones en sistemas heterogéneos distribuidos con CPUs y GPUs. Implementan internamente técnicas de *tiling* para separar las estructuras de datos en bloques que se asignan a distintos procesadores y solapan cómputo y comunicación entre dispositivos cuando es posible, mejorando la eficiencia global del programa. El tamaño del bloque asignado a cada proceso es muy relevante, ya que implica una mayor o menor cantidad de trabajo a realizar en cada dispositivo. Si esta cantidad de trabajo no está bien equilibrada, la ejecución se ralentiza, ya que cuando es necesario realizar sincronizaciones o comunicaciones entre procesos, el tiempo de ejecución pasa a ser el del más lento.

En mi trabajo de fin de grado se introdujo en las herramientas del grupo un sistema de equilibrado de carga (ALB, *Automatic Load Balancing*), basado en trabajos previos. Esta herramienta se encarga de analizar continuamente el estado actual de la ejecución y el tiempo que tarda cada proceso en completar una iteración. De forma automática y transparente al usuario redistribuye la cantidad de trabajo, dando más carga a los

dispositivos más rápidos y menos carga a los más lentos, lo que fomenta en un mayor rendimiento global.

El objetivo de este proyecto es seguir desarrollando este mecanismo, teniendo en cuenta algunos de los puntos del trabajo futuro planteados en el trabajo de fin de grado, verificando la utilidad de este mecanismo en ordenadores pre-exascala reales. Se plantea comprobar si en estos sistemas, donde el hardware de cada nodo es completamente idéntico, el estado del sistema o la naturaleza de la carga a ejecutar pueden causar desequilibrios en los tiempos de ejecución que puedan corregirse con el mecanismo ALB propuesto.

Abstract

Parallel computing is an extremely important computing paradigm based on the use of machines with multiple computing elements and the ability of certain task to subdivide into smaller jobs that can be executed simultaneously. In the field of HPC (*High Performance Computing*), clusters with massive amounts of machines interconnected by high-performance networks are used. Heterogeneous parallel computing is a subclass of the former, in which devices of different types adapted to different tasks are used in the nodes of a machine.

Today, large European pre-exascale supercomputers are built with thousands of identical nodes, including GPUs (*Graphics Processing Units*). Nodes with GPUs are managed using heterogeneous programming techniques. In a cluster, GPUs operate simultaneously with CPUs that coordinate the work of processes or communication between nodes, requiring the combination of different programming models for both types of devices.

ISL (*Iterative Stencil Loop*) applications represent an important class of solutions in the field of physical magnitudes simulation. They are based on an iterative, highly parallelizable process that requires a large amount of computing power.

The TRASGO group at the University of Valladolid has developed a series of function libraries that enable the execution of this type of application on heterogeneous distributed systems with CPUs and GPUs. They internally implement *tiling* techniques to separate data structures into blocks that are assigned to different processors and overlap computation and communication between devices whenever possible, improving the overall efficiency of the program. The size of the block assigned to each process is very important, as it implies a greater or lesser amount of work to be done on each device. If this amount of work is not well balanced, execution slows down, since when synchronization or communication between processes is necessary, the execution time becomes that of the slowest process.

In my bachelor's thesis, I introduced an *Automatic Load Balancing* (ALB) mechanism into the group's tools, based on previous works. This tool continuously analyzes the current state of execution and the time it takes for each process to complete an iteration. Automatically and transparently to the user, it redistributes the amount of work, giving more load to the fastest devices and less load to the slowest ones, which promotes greater overall performance.

The objective of this project is to continue developing this mechanism, taking into account some of the points from the future work section raised in the bachelor's thesis, verifying the usefulness of this mechanism in real pre-exascale computers. The aim is to check whether in these systems, where the hardware of each node is completely identical, the state of the system or the nature of the load to be executed can cause imbalances in execution times that can be corrected with the proposed ALB mechanism.

Índice general

Índice de figuras

Índice de tablas

Listados

Capítulo 1

Introducción

En esta sección de la memoria se hará una introducción al proyecto. Dado que el contexto es similar al del Trabajo de Fin de Grado del alumno [9], se reutilizará texto de algunas secciones del mismo, ampliando, resumiendo o adaptando el texto cuando sea necesario.

1.1. Contexto

Si se quiere acelerar un sistema informático, se tienen varias opciones. Una de ellas es aumentar la frecuencia de reloj del núcleo de la CPU, con la consecuencia directa de que se incrementan también tanto el consumo eléctrico como las temperaturas medias de la misma, alcanzándose antes sus límites físicos. Otra opción, más eficiente y escalable, es la de añadir más núcleos de procesamiento y paralelizar las tareas que se van a ejecutar en el sistema. Esta idea comenzó en ordenadores empresariales en la era del socket PGA370 de Intel (procesadores Pentium III y Celeron), donde se empezaron a comercializar placas base que permitían el uso de dos o más procesadores. Posteriormente, se generalizó para el uso doméstico en la gama de los Core 2 de Intel (Duo, Quad y Extreme), que tenían más de un núcleo en el mismo procesador. Desde entonces, la computación paralela y los procesadores multinúcleo han tenido una importancia enorme en el mundo de la informática.

La computación paralela es un paradigma de computación donde se aprovecha la capacidad que tienen ciertas tareas de dividirse en sub-tareas que se pueden ejecutar en más de un procesador de forma simultánea. Mientras que otros enfoques, como el de tiempo compartido, dan turnos a cada proceso para que todas las sub-tareas avancen a la vez, la computación paralela estudia la gestión de los problemas asociados a la utilización de más de un procesador físico ejecutando tareas de forma verdaderamente simultánea.

La computación de alto rendimiento (HPC, *High Performance Computing*) es la evolución natural de estos conceptos. Se trata de la explotación de clústeres con una cantidad masiva de elementos de cómputo interconectados por redes de alto rendimiento configurados para

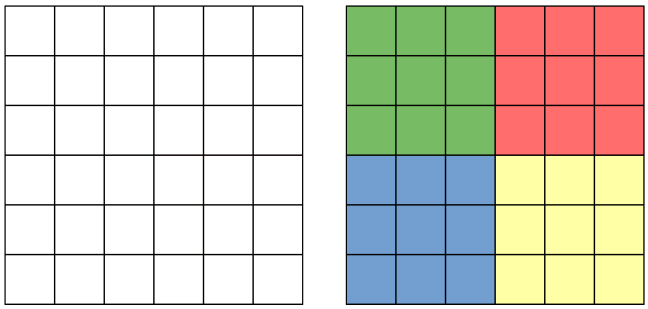

Figura 1.1: Diagrama que representa el concepto de *tiling*. Cada celda de un color distinto se enviaría a un dispositivo de cómputo distinto.

trabajar de forma paralela en la misma tarea [27], aunque engloba la optimización y mejora del rendimiento de cómputo en todo tipo de sistemas. Este tipo de clústeres son los que llenan la lista de los 500 superordenadores más potentes (la lista TOP500 [45]) y muchos centros de datos. También hay ejemplos más moderados en empresas o centros de investigación. Usualmente, los distintos equipos que forman un clúster se denominan los "nodos" del mismo [28].

La computación paralela heterogénea, en contraposición a la homogénea, es aquella en la que en los nodos de un clúster se utilizan dispositivos de cómputo de distinta naturaleza, arquitectura, generación, etc. Tiene la ventaja de que, dado que se pueden utilizar dispositivos adaptados a tareas concretas, son sistemas más flexibles y que se adaptan mejor a trabajos más especializados, pero conllevan un mayor esfuerzo de programación y mantenimiento, debido a las múltiples tecnologías y lenguajes de programación involucrados. Además, es más complejo conseguir implementaciones eficientes que coordinen y exploten los diferentes tipos de dispositivos simultáneamente [4].

Para dar solución a los problemas de la programación, compilación y ejecución de programas en clústeres paralelos heterogéneos han surgido varias herramientas y bibliotecas de funciones que facilitan estas tareas. Algunos ejemplos incluyen el estándar SYCL [34] del que hay varias implementaciones (p.ej. Intel OneApi [29], neoSYCL [33] o hipSYCL, ahora llamado AdaptiveCPP [26]) y bibliotecas construidas sobre él (p.ej. Celerity [16], Kokkos [47], HPX [42], etc.). Estas soluciones, sin embargo, presentan ineficiencias relacionadas con la gestión de las tareas, de la memoria y su comunicación, particularmente al utilizar dispositivos de diferentes tipos a la vez.

El *tiling* [30] (en español este término se traduciría como "teselado") es una transformación sobre un programa utilizada para particionar y distribuir la ejecución del mismo. Se origina en técnicas para dividir el espacio de iteraciones de bucles en bloques o *tiles* de tamaño fijo. En multiprocesadores con memoria distribuida, cada tile se separa y se asigna (*mapea*) en un procesador, detectando dependencias con otros bloques y realizando sincronizaciones y comunicaciones entre ellos cuando es necesario. Una visualización de este proceso se encuentra en la figura 1.1

El grupo de investigación TRASGO, perteneciente al Departamento de Informática de la

Universidad de Valladolid, al Grupo de Investigación Reconocido (GIR) MoBiVAP, y parte de la Unidad de Investigación Consolidada UIC-148 reconocida por la Junta de Castilla y León ha creado una serie de librerías, *Hitmap* [24], *Controllers* [36] y *EPSILOD* [10], que están diseñadas para facilitar la implementación y ejecución de programas en sistemas distribuidos heterogéneos mediante el *tiling*. Estas librerías parten y asignan el espacio de índices de estructuras de datos, por ejemplo matrices multidimensionales, entre varios dispositivos y, posteriormente, realizan cálculos sobre ellas. Simplifican además las tareas de comunicaciones y sincronizaciones entre bloques.

Hitmap es una biblioteca de funciones escrita en el estándar C99 [31], encargada de gestionar la representación y partición de los datos en un entorno distribuido y la comunicación de los mismos entre procesos. Abstrae los detalles de gestión de la memoria y ofrece funciones para la creación, partición, distribución y comunicación de tiles. Soporta el tiling jerárquico de arrays y funciona mediante un esquema SPMD (Single Program Multiple Data) que implica que todos los procesos ejecutan el mismo programa, pero con entradas distintas, pudiendo evolucionar cada uno por separado. La librería permite crear patrones reutilizables de comunicación que explotan las capacidades de bibliotecas estándar de comunicación de datos para HPC que cumplan con el estándar MPI (Message Passing Interface).

Controllers es una biblioteca de funciones desarrollada para el estándar C99 que utiliza llamadas a Hitmap y que implementa un modelo de programación paralelo para dispositivos heterogéneos. Introduce el concepto de "controller", una clase de objetos que el programador utiliza para manejar tanto las comunicaciones como la ejecución de tareas en distintos aceleradores de hardware y/o CPUs multinúcleo, de manera transparente y mediante una interfaz unificada. De esta forma, no es necesario reescribir los programas para aprovechar diferentes combinaciones de dispositivos de cómputo con naturalezas diversas. Controllers selecciona internamente y de forma automática el modelo de programación o tecnología (CUDA, OpenMP, OpenCL, etc.) más apropiado para explotar de forma más eficiente los recursos del acelerador de hardware o dispositivo de cómputo.

Las aplicaciones de tipo ISL (*Iterative Stencil Loop*) son una clase importante de soluciones a problemas de simulación en múltiples campos de la ciencia. Se pueden utilizar para caracterizar a lo largo del tiempo la evolución de magnitudes físicas como puede ser la presión, el calor, la elasticidad, la densidad, etc. En este tipo de programas, se calculan los datos de una iteración tomando los datos de las celdas vecinas en la iteración anterior, multiplicándolos opcionalmente por un factor, sumándolos y dividiéndolos entre un coeficiente. Las celdas que se consideran vecinas vienen dadas por un patrón o *stencil*. Un ejemplo de este tipo de patrones se muestran en la figura 1.2. Este tipo de aplicaciones son extremadamente paralelizables, dado que cada celda se puede calcular, en una iteración, de forma completamente independiente del resto. Además, el cálculo de una celda solo requiere conocer las celdas cercanas, una propiedad denominada "alta localidad de los datos". Esta propiedad permite, en caso de paralelizar una aplicación de este tipo mediante el tiling, reducir los accesos a la memoria durante las sincronizaciones. Este efecto se puede ver en la figura 1.3

Para explotar en sistemas heterogéneos tanto el alto grado de paralelismo como la localidad de datos de las aplicaciones ISL, el grupo TRASGO ha desarrollado una herramienta de dominio específico llamada EPSILOD, que se apoya en el modelo Controllers y en la librería Hitmap. EPSILOD facilita la ejecución de cálculos de tipo ISL en nodos heterogéneos.

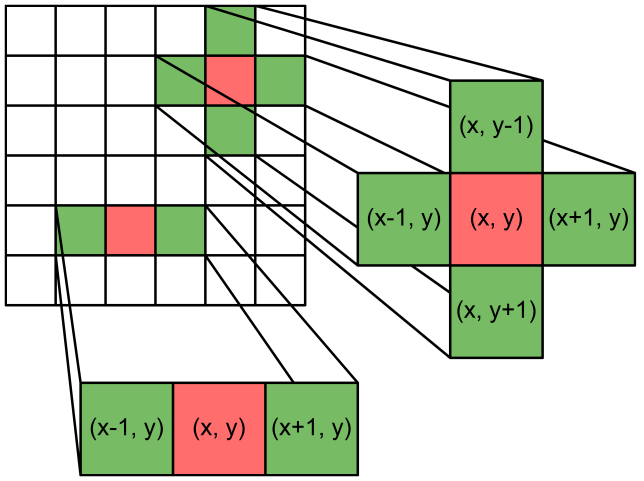

Figura 1.2: Diagrama que representa dos ejemplos de stencils (2d4 y 1dc3) para el uso en aplicaciones ISL.

Trabaja con stencils geométricos de n-dimensiones que describen la vecindad en términos de desplazamientos relativos de los índices de una celda en cualquiera de las n-dimensiones del dominio. Es capaz de manejar stencils de cualquier orden (distancia a las celdas más lejanas), simétricos o asimétricos. Además, ofrece un DSL (Domain Specific Language) para definir un stencil en términos de vecindad y de los pesos asociados para aplicar a los valores de cada vecino. A partir de esta especificación, EPSILOD genera de forma automática los kernels o funciones de cómputo adaptados a distintos dispositivos. Asimismo, calcula automáticamente la partición de datos y se encarga de la sincronización y las comunicaciones.

Se hizo previamente en mi trabajo de fin de grado una modificación a EPSILOD que añade un equilibrador de carga automático. Esta herramienta resuelve uno de los principales problemas de la computación paralela heterogénea y que previamente no se resolvía en las herramientas del grupo, el de la distribución de la carga. Cuando se tienen nodos o dispositivos de distinta naturaleza, sucede que no todos tardan la misma cantidad de tiempo en hacer la misma cantidad de trabajo. Si existen sincronizaciones en el código, por ejemplo, para intercambiar datos entre dispositivos o procesos, el tiempo total que se tarda es el del más lento.

Durante mi TFG adapte una nueva versión de una funcionalidad, denominada ALB (del inglés, *Automatic Load Balancer*), desarrollada para EPSILOD y que permite distribuir la carga de trabajo entre los distintos nodos de computación, de forma dinámica, eficiente y transparente al usuario. Esto es una ventaja importante dado que, como se ha mencionado, los nodos de computación tienen componentes de diferentes tipos y con distinto rendimiento. Se probó además de forma estadísticamente significativa que ofrecía una mejora de rendimiento en comparación a no utilizarse el equilibrador de carga en casos reales.

Este programa se presenta como alternativa al proceso de estudiar la potencia de cada nodo y marcar unos pesos fijos, con la ventaja de que además puede reaccionar a cambios en el estado del sistema. Por ejemplo, si un nodo se calienta más de lo esperado, las frecuencias

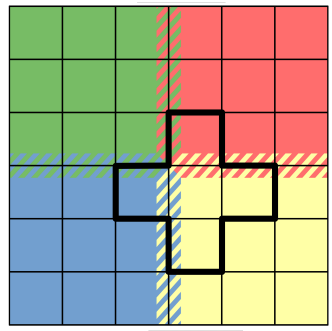

Figura 1.3: Diagrama que representa el efecto de la alta localidad de los datos en un *stencil* 2d4 al aplicar el *tiling*. Para el cálculo de la celda amarilla se necesitan datos del tile rojo y del azul, pero no del verde.

de sus elementos de procesos cambian y tarda más en ejecutar su parte del programa, se puede reequilibrar la carga para que su trabajo se reparta entre otros nodos.

1.2. Motivación

Como se ha dicho, en mi trabajo de fin de grado, se modificaron EPSILOD, Controllers y Hitmap para poder añadir un equilibrador automático de carga (denominado en este trabajo como "ALB", del inglés *"Automatic Load Balancer"*). Se hicieron pruebas preliminares para comprobar que los resultados que daba seguían siendo correctos y que se conseguía una mejora del rendimiento mediante algunos casos de prueba en las máquinas del grupo Trasgo.

En este trabajo quedaron como trabajo futuro algunas modificaciones que harían el programa más flexible y más útil para la investigación en un futuro. Una parte de este TFM es, por tanto, la implementación de esas funcionalidades y, en caso de ser necesario, la realización de experimentación que permita comprobar el comportamiento de la aplicación al añadir estos elementos.

La herramienta tiene también, a nivel teórico, un gran valor en el contexto de la computación paralela en sistemas con nodos homogéneos o de hardware idéntico entre ellos. Dado que es un equilibrador de carga dinámico, puede reaccionar a cambios en el sistema. Los nodos de computación no existen en un vacío y las condiciones ambientales, la refrigeración del sistema, los trabajos que han realizado previamente, etc, pueden afectar a la potencia que puede ofrecer cada nodo. Esperamos que nuestra herramienta sea capaz de detectar estas diferencias y redistribuya la carga de una forma que merezca la pena a largo plazo, aunque pueda funcionar peor en ejecuciones cortas.

Se le ha concedido al grupo acceso a una supercomputadora pre-exaescala europea denominada *Leonardo*. Pre-exaescala en un término que se aplica a las máquinas con capacidad de cómputo agregado en el orden de peta-flops (10^{15} operaciones de punto flotante por segundo). Estas máquinas, por las tecnologías utilizadas, se consideran las precursoras de los futuros

sistemas exa-escala (10^{20} operaciones de punto flotante por segundo). Son las máquinas en las que se están desarrollando y probando los sistemas de programación y ejecución que serán necesarios para explotar de forma eficiente los futuros sistemas de supercomputación masiva. Leonardo es un clúster de cómputo con miles de nodos idénticos que contienen CPUs y GPUs. Actualmente, está catalogada según la clasificación de la lista TOP500 [45] como la décima supercomputadora más potente del mundo y la quinta más potente de Europa. Durante un tiempo fue la segunda supercomputadora más potente de Europa y la sexta del mundo. Está gestionada por CINECA y EuroHPC. Se quiere comprobar en este trabajo si la herramienta ALB tiene también utilidad en el contexto de este tipo de máquinas.

1.3. Objetivos

El objetivo principal de este trabajo es, por una parte, actualizar la herramienta y estudiar algunas facetas de su rendimiento y, por otra parte, analizar en detalle el rendimiento de la aplicación EPSILOD utilizando la función de equilibrado de carga creada en [9] y [41], en supercomputadoras pre-exaescala con nodos idénticos.

Para esto, será necesario modificar el código del programa, tanto para añadir la funcionalidad mencionada, como para garantizar la compatibilidad con los ordenadores a los que se tendrá acceso y para poder ejecutar de una forma más sencilla los diferentes casos de estudio que se plantearán.

Los objetivos específicos del trabajo son, por tanto:

1. Estudiar y comprender los sistemas de compilación y ejecución de la supercomputadora a la que se tendrá acceso.

2. Reescribir parte de la herramienta objetivo para implementar nuevas funcionalidades.

3. Reescribir parte de la herramienta para unificarla con las ramas de desarrollo actuales.

4. Reescribir parte de la herramienta objetivo para garantizar la compatibilidad con las herramientas de desarrollo de estos sistemas per-exaescala y permitir desarrollar la batería de pruebas.

5. Comprender y utilizar los métodos para la evaluación experimental.

6. Diseñar y ejecutar una batería de pruebas con la herramienta que permitan estudiar de forma fiable su comportamiento.

7. Desarrollar habilidades para deducir el origen de los efectos observados, extraer conclusiones y presentar resultados.

1.4. Estructura del documento

El resto de la memoria comienza por una planificación de tiempo, riesgos y costes, junto con el seguimiento de la misma, en el capítulo 2, por un estudio bibliográfico y antecedentes de la aplicación, en el capítulo 3, por una descripción del desarrollo, en el capítulo 4, por un diseño de experimentación, en el capítulo 5, por un análisis de resultados, en el capítulo 6 y, finalmente, por unas conclusiones y líneas de trabajo futuras, en el capítulo 7. Finalmente, y en caso de ser necesario, se añadirán anexos con documentación complementaria al resto del trabajo.

Capítulo 2

Planificación y metodología

2.1. Planificación y seguimiento

2.1.1. Planificación inicial del proyecto

Este proyecto parte del trabajo de fin de grado [9] que se realizó previamente sobre las bibliotecas del grupo TRASGO (Hitmap, Controllers y EPSILOD) para implementar una herramienta de equilibrado de carga. Dicho proyecto no se había terminado cuando se empezó a desarrollar el trabajo de fin de máster. Por tanto, se realiza una planificación alrededor del mismo. Las fases del trabajo se describen con más detalle en los siguientes párrafos. En resumen, se comienza por el estudio del estado del arte, para analizar de que otras formas se ha resuelto el problema del equilibrado de carga. Después, se proponen y desarrollan las modificaciones de las bibliotecas para su mejora y preparación para las pruebas que se harán sobre ellas. A continuación se diseña y realiza el estudio experimental. Finalmente, se lleva a cabo la escritura de la memoria con la presentación de los resultados y las conclusiones. La descomposición temporal de estas fases para el estudiante se encuentra en la tabla 2.1 (dando un total de 150 horas, correspondientes a 6 créditos ECTS) y la misma para el tutor se encuentra en la tabla 2.2.

La primera fase implica la búsqueda y lectura de publicaciones (*papers*) y artículos relevantes escritos sobre el tema del equilibrado de carga en aplicaciones ejecutadas en sistemas distribuidos. Se realizará una síntesis de los mismos que se explicará en la siguiente sección de esta memoria. Además, se repasarán las publicaciones del grupo TRASGO sobre sus herramientas. El coste temporal de esta fase viene dado principalmente porque el alumno nunca ha realizado una revisión bibliográfica de este tipo, aunque se empieza con la ventaja de que ya ha trabajado con las bibliotecas del grupo.

La segunda parte de desarrollo pasa por implementar algunas de las posibles mejoras planteadas en la sección de “trabajo futuro” del TFG, arreglar algunos de los problemas que puedan surgir con las modificaciones realizadas, actualizar las mismas a la última rama

Tarea	Tiempo (horas)
Estudio del estado del arte	30 h
Desarrollo de software	35 h
Experimentación	40 h
Escritura de memoria	45 h

Tabla 2.1: Descomposición de tareas para el estudiante y sus tiempos asociados.

Tarea	Tiempo (horas)
Reuniones de seguimiento	17 h
Revisión de la memoria	5 h
Resolución de dudas	3 h

Tabla 2.2: Descomposición de tareas para el tutor y sus tiempos asociados.

del repositorio del grupo TRASGO y, en general, preparar las bibliotecas para las pruebas en supercomputadores pre-exaescala. No se espera que el tiempo de desarrollo se extienda demasiado, al haber adquirido el alumno experiencia previa con las bibliotecas, pero al tener que realizar varias operaciones de *merge* con otras ramas del repositorio, tampoco se espera que sea trivial.

La siguiente fase será de experimentación, donde se diseñarán una serie de pruebas, donde se utilizará el clúster del grupo TRASGO para la validación, y una supercomputadora pre-exaescala para la experimentación. Hay que tener en cuenta que para realizar la experimentación, el alumno debe además familiarizarse con los sistemas de compilación, ejecución y gestión de colas que utilizan estas máquinas.

Finalmente, se escribirá una memoria donde, utilizando gráficas y tablas creadas con Python y scripts del shell de Linux a partir de las medidas obtenidas, se mostrarán los resultados y se llegarán a unas conclusiones.

Teniendo en cuenta un trabajo de tres horas al día, el diagrama de Gantt que representa estas fases y su distribución temporal se encuentra en la figura 2.1 en el que se obtiene una fecha de fin a mediados de septiembre de 2024. En la sección 2.1.4 se discute el seguimiento del proyecto y se explican las divergencias de tiempo con la planificación que han llevado a la presentación del TFM en septiembre de 2025.

En este diagrama se puede ver una franja de tiempo marcada en rojo, esta es la sección de tiempo reservada para el desarrollo del TFG hasta su conclusión, en un principio, en julio de 2024.

2.1.2. Presupuesto

Como en cualquier otro proyecto de este tipo, se deben estimar los costes que se prevé incurrir a lo largo del mismo. Para esto, se ha hecho una planificación de costes en el caso de que el TFM se realizase en una entidad privada, en lugar de ser un trabajo para la

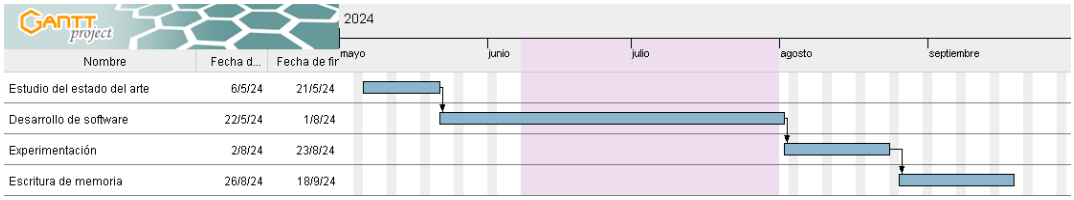

Figura 2.1: Diagrama de Gantt inicial del proyecto del TFM

universidad. Se han tenido en cuenta los costes de las máquinas utilizadas, las horas de uso de la supercomputadora pre-exaescala a la que se tiene acceso y las horas de trabajo tanto del estudiante como del tutor. Los resultados son los siguientes:

Coste de material

Se van a detallar en esta sección los costes estimados devenidos de la compra y uso de equipo informático.

Gastos de amortización del clúster del grupo:

- Gorgon: Esta máquina tuvo un coste original de 20.000€ y contiene hardware donado por la empresa NVIDIA por valor de otros 10.000€ (Que se contará para la estimación como si se hubiera tenido que abonar). Se ha estimado un tiempo de amortización de 4 años, dando un coste de 0,86€ por hora de uso. Teniendo en cuenta las 150 h dedicadas al desarrollo del TFM, tenemos un coste total 128,4€.
- Manticore: Esta máquina tuvo un coste original de 40.000€. Se ha estimado un tiempo de amortización de 4 años, dando un coste de 1,14€ por hora de uso. Teniendo en cuenta las 150 h dedicadas al desarrollo del TFM, tenemos un coste total 171,23€.

Gastos de amortización de la máquina del alumno:

- Máquina del alumno: Esta máquina personal del alumno (un Samsung np300e) tuvo un coste original de alrededor de 500€-600€. Lleva en uso desde 2014, dando una vida útil de algo más de 10 años a fecha de inicio del proyecto. Se ha estimado un tiempo de amortización de 6 años, dando un coste de 0,01€ por hora de uso. Teniendo en cuenta las 150 h dedicadas al desarrollo del TFM, tenemos un coste total 1,5€.

Gastos de amortización de LEONARDO:

- LEONARDO: Esta es la supercomputadora pre-exaescala europea a la que se tiene acceso y se ha utilizado para realizar la experimentación del TFM. Debido a ser el grupo

TRASGO parte de un GIR (*Grupo de Investigación Reconocido*) de la Universidad de Valladolid, una entidad pública de investigación, y a que se le ha concedido un proyecto dentro de la iniciativa europea EuroHPC-JU para el uso del clúster, no fue necesario pagar por las horas de uso. Sin embargo, se va a intentar estimar el coste que tendría el uso de la máquina.

En primer lugar, hay que tener en cuenta que una hora dentro del clúster no equivale a una hora real de ejecución, sino que depende de la cantidad de nodos que se utilicen, de la configuración de los nodos, de la máquina a la que pertenezcan los nodos, etc. Por tanto, se utilizará el comando "*saldo*", propio del clúster, para obtener las "horas" de cómputo globales gastadas.

Por otra parte, el coste de uso de la máquina se negocia por cada entidad privada directamente con CINECA, la organización encargada de LEONARDO. Sin embargo, algunas estimaciones parecen indicar un coste de unos 0,114€ por hora de uso para empresas[46].

Al haberse utilizado un total de 12793 horas de uso durante el proyecto, se habría tenido un coste total de 1458.4€.

Coste de personal

Se pasa a estimar el coste que se ha considerado para las horas de trabajo empleadas tanto por el tutor como el alumno durante el transcurso del proyecto.

Sueldos

- Ingeniero de Software Junior: El sueldo del alumno se ha estimado como el que tendría un trabajador con un puesto junior de Ingeniería de Software en España, que ronda entre los 19.000€ y los 27.000€ [23]. Se ha tomado una cifra media de 23.000€ brutos anuales como sueldo del estudiante. Teniendo en cuenta una jornada laboral de 8 horas al día y tomando 250 días laborables al año, se tiene un salario por hora de 11,5€. Teniendo en cuenta las 150 h que el alumno debe dedicarle al proyecto, se tiene un coste total de 1725€.
- Profesor de la Universidad de Valladolid: El sueldo del tutor se ha estimado como el que tiene un profesor titular de la Universidad de Valladolid, que ronda los 44.000€ [48][32]. Teniendo en cuenta una jornada laboral de 8 horas al día y tomando 250 días laborables al año, se obtiene un salario por hora de 22€. Teniendo en cuenta las 25 h que el tutor debe dedicarle al proyecto, se tiene un coste total de 550€.

Costes de licencias de software

En el proyecto se han utilizado herramientas Open Source o de uso libre en todos los casos, incluyendo el sistema operativo Linux, Visual Studio Code, GanttProject, Latex, compiladores GCC o de NVIDIA de libre distribución, etc. Luego las licencias de software han costado 0€.

Coste total

Teniendo en cuenta el coste estimado en las secciones anteriores, en caso de realizarse el TFM en una empresa, el coste total del proyecto sería de unos 4034.53€. Este coste es algo optimista, al no tenerse en cuenta todos los posibles gastos que se tendrían en una empresa al hacer este tipo de proyecto, pero se considera una buena aproximación a la cifra real.

2.1.3. Planificación de riesgos

Pasamos a detallar la planificación de riesgos del proyecto. Para esto, analizaremos las amenazas que pueden afectarlo y planearemos acciones de mitigación y contingencias para el caso en el que las amenazas se acaben manifestando. En esta sección, haremos la identificación, el análisis, la priorización, la planificación de contingencias y el plan de monitorización de los riesgos planteados para este trabajo.

Cada riesgo quedará definido por las siguientes características:

- Nombre: La forma mediante la cual se denominará el riesgo.
- Descripción: Resumen del riesgo que indicará tanto la vulnerabilidad que explota como la amenaza que supone.
- Categoría: Faceta del proyecto a la que afecta el riesgo.
- Probabilidad: Valor bajo, medio o alto que indica la probabilidad de que la amenaza se materialice.
- Impacto: Valor bajo, medio o alto que indica el impacto que tendría el efecto de la amenaza sobre el proyecto.
- Acciones de mitigación: Acciones que se plantean para reducir la probabilidad de que el riesgo ocurra.
- Acciones correctivas: Acciones orientadas a, una vez una amenaza ha sucedido, reducir su impacto.

Los riesgos planteados para este proyecto son los siguientes:

ID	RSK01
Nombre	Enfermedad del alumno
Descripción	El estudiante puede caer enfermo durante el transcurso del proyecto, evitando que pueda trabajar en el mismo y retrasando la entrega en los plazos establecidos.
Categoría	Personal
Probabilidad	Baja
Impacto	Medio
Acciones de mitigación	1. Evitar situaciones que puedan implicar la enfermedad del alumno
Acciones correctivas	1. Aumentar las horas trabajadas por día 2. Facilitar el teletrabajo u otros sistemas que permitan al estudiante seguir con el proyecto 3. Replanificación del proyecto

Tabla 2.3: Tabla asociada a RSK01: Enfermedad del alumno

ID	RSK02
Nombre	Fallos técnicos
Descripción	Los sistemas necesarios para hacer el TFM (P.E. las máquinas del grupo) podrían fallar, retrasando o impidiendo la realización del mismo.
Categoría	Recursos
Probabilidad	Baja
Impacto	Alto
Acciones de mitigación	1. Utilizar una política de copias de seguridad 2. Monitorizar los sistemas utilizados 3. Disponer de una máquina secundaria
Acciones correctivas	1. Sustituir la máquina afectada 2. Replanificación del proyecto

Tabla 2.4: Tabla asociada a RSK02: Fallos técnicos

ID	RSK03
Nombre	Sustitución del tutor
Descripción	Por enfermedad u otras circunstancias, podría ser necesario que el tutor del TFM sea sustituido. Esto podría requerir que el trabajo se realizara de forma distinta o con otro enfoque o podría ser más difícil trabajar con él.
Categoría	Personal
Probabilidad	Media
Impacto	Bajo
Acciones de mitigación	1. Tratar de reducir la dependencia del tutor
Acciones correctivas	1. Replanificación del proyecto

Tabla 2.5: Tabla asociada a RSK03: Sustitución del tutor

ID	RSK04
Nombre	Cambios en los requisitos
Descripción	Es posible que surjan cambios necesarios en los requisitos del proyecto y que, por tanto, sea necesario replantear el desarrollo, dando lugar a retrasos.
Categoría	Planificación
Probabilidad	Baja
Impacto	Medio
Acciones de mitigación	1. Realizar un modelado de requisitos lo más detallado posible 2. Utilizar algún tipo de metodología ágil
Acciones correctivas	1. Aumentar las horas trabajadas por día 3. Replanificación del proyecto

Tabla 2.6: Tabla asociada a RSK04: Cambios en los requisitos

ID	RSK05
Nombre	Dificultad del desarrollo
Descripción	Por desconocimiento, falta de formación u otros motivos, el desarrollo puede resultar más difícil de lo inicialmente esperado, provocando retrasos en el trabajo.
Categoría	Personal
Probabilidad	Baja
Impacto	Medio
Acciones de mitigación	1. Dedicar el tiempo necesario antes del desarrollo al aprendizaje de las librerías y herramientas del grupo
Acciones correctivas	1. Aumentar las horas trabajadas por día 2. Replanificación del proyecto

Tabla 2.7: Tabla asociada a RSK05: Dificultad del desarrollo

ID	RSK06
Nombre	Errores en la planificación
Descripción	Se pueden cometer errores, por inexperiencia, durante la etapa de planificación del proyecto, lo cual puede evitar que se lleguen a los hitos esperados en las fechas esperadas.
Categoría	Planificación
Probabilidad	Baja
Impacto	Alto
Acciones de mitigación	1. Comprobar con el tutor que la planificación se hace de la forma correcta de forma regular
Acciones correctivas	1. Aumentar las horas trabajadas por día 2. Replanificación del proyecto

Tabla 2.8: Tabla asociada a RSK06: Errores en la planificación

ID	RSK07
Nombre	Desarrollo incorrecto del software
Descripción	Es posible que el desarrollo que se haga sea incorrecto y que, por tanto, haya que volver a realizarlo, retrasando la entrega.
Categoría	Planificación
Probabilidad	Baja
Impacto	Medio
Acciones de mitigación	1. Comprobar con el tutor que el desarrollo se hace de la forma correcta de forma regular
Acciones correctivas	1. Replanificación del proyecto

Tabla 2.9: Tabla asociada a RSK07: Desarrollo incorrecto del software

ID	RSK08
Nombre	Incorrecta estimación del tiempo para el TFG
Descripción	Dadas las circunstancias concretas del alumno y el solapamiento del TFG con el TFM, puede suceder que una mala planificación del TFG retrase la entrega del TFM.
Categoría	Planificación
Probabilidad	Media
Impacto	Medio
Acciones de mitigación	1. Comprobar con el tutor que la planificación se hace de la forma correcta de forma regular
Acciones correctivas	1. Replanificación del proyecto

Tabla 2.10: Tabla asociada a RSK08: Incorrecta estimación del tiempo para el TFG

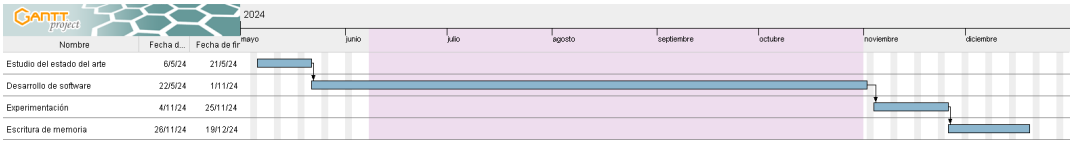

Figura 2.2: Diagrama de Gantt tras la manifestación del riesgo "Incorrecta estimación del tiempo para el TFG"

ID	RSK09
Nombre	Cambios en la situación laboral
Descripción	El estudiante está trabajando en un laboratorio de investigación ajeno al TFM de forma simultánea a sus estudios. Un cambio en el entorno laboral o en los requisitos del trabajo podrían retrasar el TFM.
Categoría	Planificación
Probabilidad	Baja
Impacto	Medio
Acciones de mitigación	1. Buscar condiciones especiales que permitan realizar el TFM
Acciones correctivas	1. Replanificación del proyecto

Tabla 2.11: Tabla asociada a RSK09: Cambios en la situación laboral

2.1.4. Seguimiento del proyecto

En este proyecto se ha manifestado un riesgo que ha obligado a la replanificación del mismo, este se trata del riesgo *RSK08* (Incorrecta estimación del tiempo para el TFG) indicado en la tabla 2.10. Inicialmente, se esperaba terminar dicho trabajo en julio de 2024. Sin embargo, su desarrollo, por acuerdo entre el tutor y el alumno, se extendió hasta octubre de 2024. Dadas estas circunstancias, fue necesario replanificar el desarrollo, que se muestra en el diagrama de Gantt de la figura 2.2. Esta planificación nos da una fecha de fin de proyecto en diciembre de 2024.

El trabajo, tal como se indica, se terminó en su mayoría en diciembre de 2024. Por motivos personales, tanto del alumno como del tutor, se retrasó la fecha de entrega y presentación del Trabajo de Fin de Grado hasta la convocatoria ordinaria de junio de 2025. Dado que es necesario disponer del título de grado para defender el proyecto, se esperó hasta la convocatoria extraordinaria de septiembre de 2025 para la entrega del Trabajo de Fin de Máster.

En el tiempo entre la finalización del trabajo y su entrega se hicieron algunas modificaciones menores. En concreto, se arreglaron algunos errores que se fueron detectando durante su uso y se solucionaron problemas puntuales. Se realizaron algunas nuevas pruebas y medidas experimentales. También se realizaron algunas revisiones posteriores de la memoria. Finalmente, se escribió y publicó un artículo de congreso en las XXXV Jornadas de Paralelismo

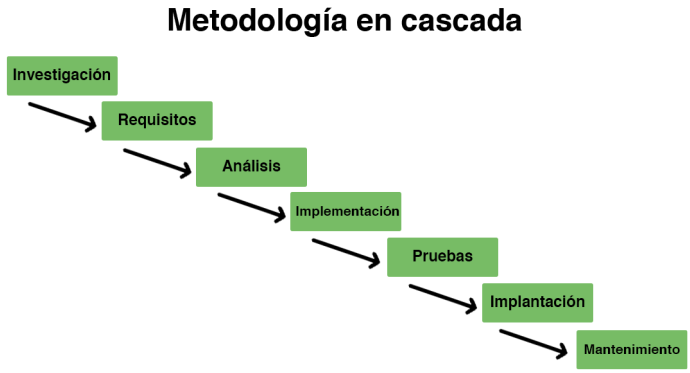

Figura 2.3: Diagrama que representa la metodología en cascada

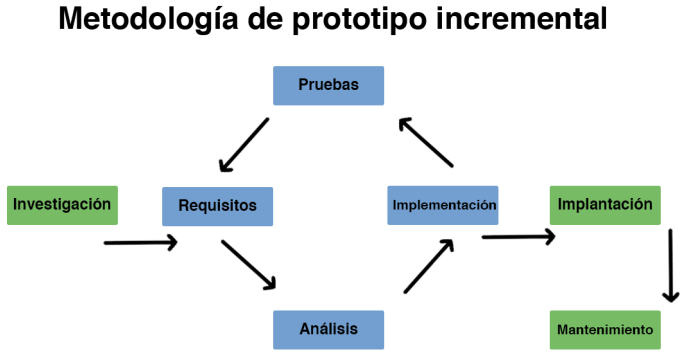

Figura 2.4: Diagrama que representa la metodología de prototipo incremental

(JP2025) a partir del TFG del alumno y que incluye parte de material relacionado con el TFM.

2.2. Metodología

La metodología que se utilizará durante este proyecto es la de prototipo incremental [18][17], como se indica en la figura 2.4. Esta es una adaptación de la ingeniería de software en cascada [19], mostrado en la figura 2.3, donde al principio del proyecto se planifican de forma secuencial y lineal todas las tareas que se tienen que realizar durante el mismo. Esta metodología es demasiado rígida para el proyecto planteado, ya que hay requisitos que pueden cambiar a lo largo del mismo a medida que se vaya probando la herramienta en los ordenadores pre-exaescala.

La metodología de prototipo incremental, sin embargo, se adapta perfectamente al trabajo a realizar, donde en cada incremento se van a diseñar, añadir y probar distintas funcionalidades al programa, tal como se plantearon en el trabajo futuro del TFG. Esto permite un desarrollo mucho más flexible donde los requisitos pueden variar durante el proyecto.

Por otra parte, otra metodología que se podría haber utilizado es la de SCRUM [20], un *framework* ágil para el desarrollo de software donde se realizan "*sprints*" con una duración fija en los que se marcan una serie de objetivos para el desarrollo de un software. Tiene múltiples ventajas, pero no se ha utilizado en este proyecto dado que no es necesaria la flexibilidad que ofrece (el desarrollo software a realizar es menor y los requisitos no son tan variables) y se ha considerado oportuno un sistema más sencillo.

Una vez realizado el desarrollo, se llevará a cabo una fase de experimentación. Se obtendrán resultados y se intentará explicarlos.

Capítulo 3

Estado del arte

En esta sección de la memoria se mostrará tanto el resultado de la revisión bibliográfica que se ha realizado, como los antecedentes al trabajo que se va a realizar. Al igual que sucede en la introducción, algunas partes se reutilizarán del TFG, expandiendo, resumiendo o adaptando al nuevo contexto según sea necesario.

3.1. Trabajo relacionado

En el ámbito de la computación paralela heterogénea, a la hora de ejecutar un trabajo, el equilibrio de la carga es un aspecto crítico (véase [7], [40], [1]). Cuando se producen comunicaciones o sincronizaciones, las partes que se procesan más lento se convierten en el camino crítico, retrasando la ejecución y aumentando el tiempo global de la aplicación. Por tanto, un equilibrado de carga efectivo no solo maximiza el uso de los recursos disponibles, sino que también minimiza el tiempo de ejecución total. Estas diferencias son más notables en el caso de la computación paralela heterogénea, ya que las diferencias de capacidad de cómputo entre los distintos dispositivos de procesamiento es mucho más grande. Dado que esta es una tendencia importante en el mundo de [6] HPC (High Performance Computing), es un escenario al que se debe prestar atención. Una primera clasificación de los métodos de distribución de la carga los puede separar entre métodos estáticos y métodos dinámicos [50] [21].

3.1.1. Métodos estáticos

Estos métodos son todos aquellos en los que la distribución de la carga se determina antes de la ejecución [21], haciendo suposiciones o tomando medidas sobre el problema a resolver y/o en las máquinas sobre las que se ejecuta el programa. Son más sencillos de implementar que los métodos dinámicos, pero también son menos flexibles y adaptables. Algunos ejemplos de estos métodos pueden ser:

- Distribución homogénea: Se divide el problema en tantas partes como dispositivos, todos del mismo tamaño. En sistemas heterogéneos esta distribución no es la más apropiada [7].
- Distribución basada en pesos: Se divide el problema en tantas partes como dispositivos, cada una con un tamaño acorde a la capacidad de procesamiento estimada de cada uno (posiblemente medida previamente con un benchmark adecuado). Este método se describe en, por ejemplo, [5]. Véase por ejemplo [7], [43]. Tienen la desventaja de que, aun equilibrando mejor en situaciones de heterogeneidad, no pueden reaccionar a cambios en el sistema.

3.1.2. Métodos dinámicos

Los métodos de equilibrado de carga dinámico permiten ajustar la distribución de tareas en tiempo real, respondiendo a las variaciones en la carga de trabajo y al comportamiento del sistema. Estos métodos son más complejos y pueden incurrir en sobrecostes, pero ofrecen una mayor flexibilidad y eficiencia [21]. Especialmente en entornos cambiantes, donde los dispositivos utilizan sistemas de adaptación dinámica de la frecuencia en función del uso o calentamiento de los mismos.

- Migración de tareas: En este método, tareas que están tardando más de lo esperado pueden ser transferidas de un dispositivo sobrecargado a uno menos ocupado. Esto se puede implementar mediante algoritmos de asignación que monitorizan continuamente el estado de cada dispositivo y toman decisiones en tiempo real. Véase por ejemplo [40], [12].
- Algoritmos de equilibrado de carga en tiempo real: Estos algoritmos utilizan información sobre el tiempo de ejecución de las tareas y la carga actual de cada dispositivo para redistribuirlas en función de su comportamiento. Utilizan heurísticas para decidir la asignación de tareas de manera más eficiente, considerando factores como el tiempo de espera, el tiempo de ejecución estimado y el estado actual de cada dispositivo. Véase por ejemplo [2], [8].
- División en subtareas: Se divide el problema en subtareas, usualmente del tamaño más pequeño posible. Se reparten estas tareas a cada dispositivo cuando este está libre. Se asume que un dispositivo hará una cantidad de trabajo reducida. Ver por ejemplo [37], [13], [44]. Tiene la desventaja de que, aun siendo más flexible que otras opciones anteriores, la sobrecarga sobre el tiempo total de ejecución puede ser demasiado alta. Suele ser considerado como método dinámico, ya que la distribución se hace en tiempo de ejecución, pero se puede considerar estático (es considerado estático en [39]) o híbrido, al hacerse la división en subtareas de forma previa.

Nuestra propuesta se encuentra en esta clase. Es una mejora a los sistemas ya existentes mostrados, ya que realiza reequilibrados de particiones de grano grueso de forma progresiva, utilizando una heurística introducida como parámetro para decidir cuando es más conveniente efectuar los reequilibrados. Además, es completamente transparente.

3.2. Antecedentes

Tal como se ha mencionado previamente, el grupo TRASGO de la Universidad de Valladolid ha desarrollado varias bibliotecas de funciones que, a partir del concepto de particiones no solapadas de arrays (tiling [30]), tienen como función distribuir un trabajo entre varios nodos y/o dispositivos de cómputo homogéneos o heterogéneos. Estas herramientas permiten un gran control sobre la partición y distribución sin modificar el resto del código que ejecuta las partes de cómputo o realiza las comunicaciones. Estas bibliotecas de funciones están escritas en C11, y son compatible con cualquier compilador de C o C++ moderno.

3.2.1. Hitmap

Hitmap [24] es una biblioteca diseñada para facilitar la gestión del *tiling* en programas paralelos en entornos distribuidos. Su función principal es encargarse del particionado de datos, la asignación de estos a los procesadores disponibles, y la gestión de la comunicación y sincronización entre procesadores. Para ello, Hitmap ofrece una interfaz abstracta y un sistema de plug-ins que permite escribir código independiente de las funciones de mapeo subyacentes. Además, soporta el *tiling* jerárquico de arrays multidimensionales lo que posibilita la explotación del paralelismo tanto de datos como de tareas mediante un modelo SPMD (*Single Program, Multiple Data*), donde cada procesador ejecuta el mismo programa pero con diferentes datos. Finalmente, Hitmap facilita la creación de patrones de comunicación reutilizables que aprovechan internamente las capacidades de MPI. Mediante diversas aplicaciones y benchmarks paralelos se ha demostrado que los programas desarrollados con Hitmap alcanzan un nivel de eficiencia comparable al de implementaciones ad hoc que utilizan MPI directamente, pero con una reducción significativa del esfuerzo de programación al ocultar al usuario las complejidades asociadas con el uso de MPI.

Las demás librerías del grupo TRASGO, Controllers y EPSILOD, están construidas encima de Hitmap, y utilizan extensamente sus abstracciones para la coordinación y el movimiento de los datos y para ocultar los detalles sobre la disposición en memoria de los mismos.

Uso de Hitmap

Para utilizar y modificar Hitmap es necesario describir sus estructuras de datos (arrays o, de forma más genérica, tiles), qué representan y cuál es la notación mediante la que se representan y utilizan. Se definen los siguientes conceptos:

- *Signature*: Se podría traducir como *signatura*. Se trata de una tupla de tres enteros que representan un subespacio de índices de un array en una dimensión. Esta notación es similar a la utilizada en la selección de índices de matrices en lenguajes como Fortran90 o MATLAB. La cardinalidad de una signatura es el número de índices distintos en un dominio.

$$s \in Signature = (inicio : final : paso)$$

$$Card(s \in Signature) = [(s.final - s.principio)/s.paso]$$

- *Shape*: Se podría traducir como *forma*. Se trata de una n-tupla de signatures. Representa una selección de un subespacio de índices de un array en un dominio multidimensional. La cardinalidad del shape es el número de combinaciones distintas de índices en el dominio.

$$h \in Shape = (S_0, S_1, S_2, ..., S_{n-1})$$

$$Card(h \in Shape) = \prod_{i=0}^{n-1} Card(S_i)$$

- *Tile*: Se podría traducir como *tesela*. Se trata de un array de n dimensiones. Su dominio viene definido por un shape y tiene datos de un tipo determinado en su creación. Gestiona el espacio de memoria y almacenamiento de los datos.

$$Tile_{h \in Shape} : (S_0 \cdot S_1 \cdot S_2 \cdot ... \cdot S_{n-1}) \rightarrow < type >$$

Hitmap ofrece funciones que se pueden ordenar en tres tipos. Funciones de tiling, de mapeado y de comunicaciones.

- Funciones de tiling: Tanto los arrays como los tiles en Hitmap están implementados mediante un tipo abstracto de datos, *HitTile*. Se puede derivar un tile a partir de otro especificando un subdominio, que es un subconjunto de índices del tile padre. Un ejemplo de esto se ve en la figura 3.1.

  Además, se tienen funciones para asignar espacio en memoria a un tile y manejar la superposición de tiles. Esto resulta muy útil para crear buffers y sub-buffers en algoritmos de stencil paralelizados.

- Funciones de mapeado: Estas funciones se encargan de forma automática del particionado y la distribución de los datos en los distintos procesadores a los que se tiene acceso. Las funciones se separan a su vez entre las que construyen una topología (*topology*) virtual de procesos para organizarlos en *grids* o mallas de varias dimensiones, y las que controlan el mapeo, la disposición de las partes o tiles de un array sobre la topología de procesos (*layout*). La combinación de funciones de topology y layout permiten escoger diferentes formas de partir y distribuir los datos con solo cambiar los nombres de las funciones o políticas que se usan como parámetros, sin necesidad de cambiar el resto del código que se adapta a la topología y layout escogidos.

- Funciones de comunicaciones: Hitmap define una abstracción para la comunicación de partes concretas dentro de estructuras jerárquicas entre procesos virtuales. La información necesaria para enviar los datos entre procesadores físicos se guarda en un tipo de datos denominado *Comm*. Los objetos Comm se crean recibiendo como parámetro la

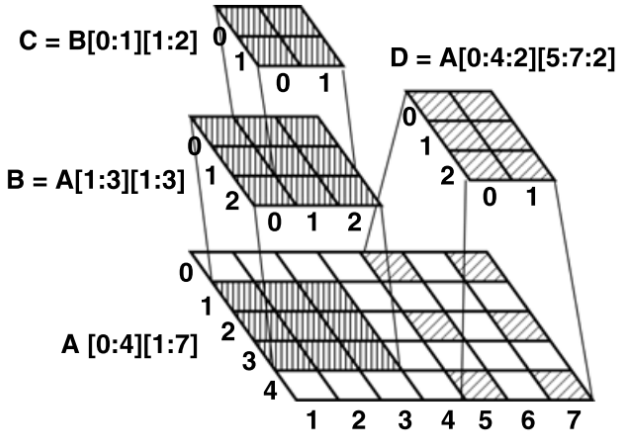

Figura 3.1: Ejemplo de la creación de tiles a partir de un array original, extraído de [24]

topología y layout. Las funciones de comunicación utilizadas internamente se adaptan a la topología y layout de forma transparente. Un conjunto de objetos de este tipo se pueden agrupar en un objeto *Pattern*. Esto permite generar combinaciones o estructuras de comunicación más complejas. Estos patrones de comunicación se pueden reutilizar a lo largo del programa.

Diseño de Hitmap

Hitmap es una biblioteca de funciones escrita en C99 para maximizar su compatibilidad con compiladores y herramientas específicos de dispositivos o aceleradores hardware y para aprovecharse de la experiencia de los desarrolladores de Hitmap en las optimizaciones de los compiladores de C. Aunque está escrita en C, se ha diseñado con un enfoque orientado a objetos, utilizando estructuras de datos para almacenar los campos de cada clase, y un conjunto de funciones que trabajan con cada estructura para implementar sus métodos. El diagrama de clases de la librería se muestra en la figura 3.2. El desarrollo de una interfaz orientada a objetos en C++ de esta biblioteca sería razonablemente sencilla.

3.2.2. Controllers

Controllers es una biblioteca de funciones escrita en C99 que implementa el concepto de *controller*, presentado en [36]. Controller es una entidad abstracta que se encarga de manejar, de forma transparente para el programador, el lanzamiento de tareas tanto en aceleradores de hardware como en CPUs multicore utilizando en ambos casos las mismas abstracciones y metodología. Esto reduce el esfuerzo de programación, siendo la alternativa la combinación de varios modelos de programación para los diferentes aceleradores utilizados. Este concepto es una modificación y ampliación de *communicators*, una entidad abstracta similar definida en [35].

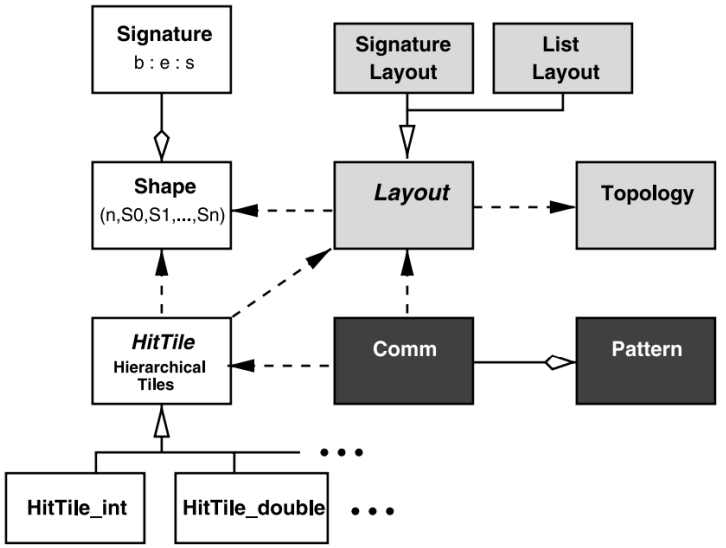

Figura 3.2: Diagrama de clases UML de Hitmap, extraído de [24]

Para esto, controllers implementa diferentes *backends* que se encargan de compilar y enlazar el código del usuario con las tecnologías específicas para el manejo de cada tipo de acelerador hardware o procesador de CPU. Ejemplos de estas tecnologías pueden ser CUDA u OpenCL para tarjetas gráficas de NVIDIA y AMD respectivamente, OpenMP para conjuntos de procesadores de CPU, o la biblioteca COI de intel para aceleradores Xeon Phi.

El modelo controller tiene varias propiedades importantes. En primer lugar, tiene un mecanismo para definir kernels comunes, de tal forma que sean reutilizables para distintos tipos de aceleradores hardware e incluso un subconjunto de procesadores de una CPU multicore. Además, tiene un sistema de optimización que encuentra los mejores valores para diversos parámetros necesarios en el lanzamiento de los kernels. También tiene un sistema transparente de manejo de la memoria y, finalmente, una abstracción para unificar la forma de indexar los datos en distintos tipos de dispositivos.

Arquitectura y explicación del modelo Controller

El modelo controller, cuya arquitectura se muestra en la figura 3.3, introduce una forma sencilla de coordinar la ejecución de una serie de kernels, que se definen como funciones. Un controller se asocia a un acelerador hardware (dispositivo o *device*) y gestiona una cola de ejecución de *kernels* o bloques de código, que se encuentra en la máquina *host*. Los controllers manejan automáticamente dos conceptos importantes al programar para aceleradores hardware:

- Control de kernels: Esto incluye tanto la selección de parámetros específicos de confi-

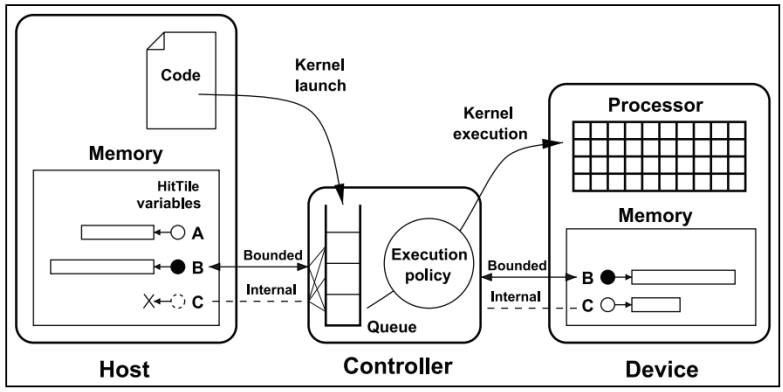

Figura 3.3: Arquitectura original del modelo Controller, extraído de [36]

guración para un kernel como el lanzamiento en el dispositivo asociado al controller del kernel mediante un sistema de colas. Los datos se transmiten a la memoria del acelerador hardware de forma automática cuando se necesita. Las principales contribuciones que el modelo controller añade a la gestión de kernels son:

- Definición y lanzamiento de kernels: El modelo define un sistema genérico para declarar kernels, estos se definen como una tupla ($< nombre >, < tipoDispositivo >, < listaParametros >, < código >$), donde en *tipoDispositivo* es una lista de símbolos que indican los dispositivos donde se lanzará el kernel. Mediante este sistema se pueden definir varios kernels con el mismo nombre, pero optimizados para distintas arquitecturas.
- Caracterización de kernels para ejecución: Controllers considera de forma automática la caracterización del código del kernel para optimizar los parámetros de lanzamiento como, por ejemplo, las geometrías de los bloques de hilos.

- Manejo de datos: Los aceleradores hardware suelen tener su propio espacio de memoria, obligando a transferir al dispositivo los datos de entrada del kernel que se vaya a ejecutar y, al finalizar la ejecución, a recuperar los resultados. Programar estas transferencias de forma manual es engorroso y puede dar lugar a errores en el código. Además, es difícil prever cuando se pueden hacer transferencias asíncronas de datos, o cuando deberían mantenerse en el acelerador hardware, dado que esto depende del orden de lanzamiento de los kernels. El modelo controllers abstrae estos conceptos y le evita al programador gestionarlos. Las características principales del modelo con respecto al manejo de datos son:

  - Transferencias de datos: Al crear un controller, este se asocia automáticamente con un dispositivo acelerador hardware o con un subconjunto de núcleos de una CPU. Este controller maneja, de forma transparente para el programador, el espacio de memoria del acelerador. El controller decide cuando y como se realizan transferencias de datos, según los kernels utilizados.
  - Variables ligadas: Las variables ligadas son aquellas que están definidas en el host, pero tienen una imagen reflejada en el espacio de memoria del dispositivo. Una

vez se liga una variable, el controlador gestiona la coherencia de los datos entre la imagen de memoria en el host y en el acelerador, realizando los movimientos de datos necesarios para que los kernels que los usan y el host estén siempre sincronizados.

- Variables internas: Las variables internas tienen espacio de memoria reservado solo en el dispositivo. Permiten mantener datos que se reutilizan por varios kernels, actuando como variables temporales en el dispositivo y sin necesidad de sincronización con el host. Estas variables nunca implican una transferencia de memoria.
- Accesos de memoria con un espacio de índices uniformes: En lenguajes de programación como C, las estructuras de datos indexadas se almacenan en memoria en orden *row-major*. En esta disposición el primer índice indica filas, el segundo columnas, etc., y los datos quedan ordenados primero por filas, luego por columnas, etc. Sin embargo, en algunos dispositivos como las GPUs, los índices de los threads tienen ordenaciones diferentes: *colum-major* o híbridas. Controller utiliza la abstracción HitTile de Hitmap para representar las estructuras de datos y crear un interfaz homogéneo para la ordenación de índices, de forma que se puede asumir un orden *row-major* independientemente del dispositivo. Esto permite reutilizar el código de una forma orgánica y simple.

3.2.3. EPSILOD

EPSILOD es un *framework* que facilita programar y ejecutar aplicaciones ISL en todo tipo de sistemas heterogéneos. Está construido en C99, al igual que controllers e hitmap. Esto lo hace compatible con cualquier compilador de C o C++ moderno. EPSILOD está basado en el concepto de esqueleto paralelo (*parallel skeleton*), concepto definido en [25] como un constructo de programación o una función en una biblioteca que abstrae un patrón de computación e interacción paralela. EPSILOD implementa un esqueleto paralelo para aplicaciones stencil en forma de una función de alto orden (recibe como parámetro otra función). La función recibida como parámetro se utiliza para calcular la actualización de una celda del dominio en función de sus celdas vecinas. EPSILOD provee mecanismos para generar automáticamente esta función a partir de especificaciones más abstractas. El esqueleto paralelo resuelve un número determinado de iteraciones del stencil, aplicándolo a todos los elementos del dominio n-dimensional deseado, iteración tras iteración.

EPSILOD utiliza por dentro mecanismos tanto de controllers como de hitmap para iniciar de forma asíncrona comunicaciones cuando se detecta que se ha terminado de actualizar una parte de una estructura de datos que reside en un dispositivo, pero es necesaria en la siguiente iteración en otro. Continúa actualizando el resto de las celdas del dominio lanzando tareas en el dispositivo y sincroniza la ejecución y la comunicación. De esta forma se maximizan las oportunidades para solapar los cálculos con los movimientos de datos, reduciendo el impacto del uso de múltiples dispositivos.

Los resultados experimentales que se han tomado sobre EPSILOD indican que es más rápido que herramientas de su misma clase como Celerity e indican una buena escalabilidad fuerte y débil tanto para sistemas homogéneos como heterogéneos.

Uso de EPSILOD

En esta sección se describe la interfaz de EPSILOD, los conceptos necesarios para su uso, y una introducción a su funcionamiento interno.

Descripción de un stencil La operación de un stencil viene dada por la descripción del mismo, la cual está constituida por:

- Un Shape, expresado como un objeto HitShape, que indique la distancia al vecino más lejano en cada dimensión.
- Un patrón, expresado como un array de pesos/coeficientes para cada elemento del shape, donde los ceros se ignoran y los valores positivos o negativos se multiplican por los valores de los vecinos correspondientes.
- Un factor por el cual se divide la suma de los valores obtenidos de los vecinos y sus pesos.

Un ejemplo de una descripción, en concreto de una estrella de cuatro puntas en dos dimensiones, se muestra en el listado 3.1. Este stencil sirve para calcular una aproximación para la distribución de calor, en un espacio multi-dimensional, en el punto de equilibrio. Implementa un método iterativo de Jacobi para resolver la ecuación de Poisson en un espacio discretizado con elementos finitos.

```
1 HitShape shp_jacobi2d4 = hitShape((-1,1), (-1,1));
2
3 float patt_jacobi2d4[] = {
4 0,1,0,
5 1,0,1,
6 0,1,0};
7
8 float factor_jacobi2d4 = 4;
```

Listado 3.1: Ejemplo de descripción de un stencil de una estrella de cuatro puntas en dos dimensiones. Extraído de [10].

Kernel genérico EPSILOD incluye un kernel genérico que funciona para cualquier descripción de un stencil en una, dos o tres dimensiones. Funciona iterando sobre todos los elementos del patrón del stencil. En cada elemento diferente de cero, se accede al vecino correspondiente, se multiplica por el peso indicado y se suma al resultado.

Kernel Generation Tool Al igual que existe un kernel genérico, el usuario también puede utilizar kernels con código más optimizado. EPSILOD provee además una herramienta para generar kernels específicos a partir de la definición de un stencil. Esta herramienta se llama Kernel Generation Tool o KGT.

El KGT genera un kernel que evita accesos innecesarios a elementos que no tienen peso en el cálculo final. El KGT está implementado en Bash+Awk, toma como entrada un archivo en código C que contenga una descripción de un stencil con el formato indicado previamente y devuelve el código de un kernel directamente utilizable en el modelo Controller, que contiene las expresiones de acceso mínimas necesarias.

En el listado 3.2 se muestra la implementación del kernel descrito en el listado 3.1, como ejemplo de un kernel generado por el KGT.

```
1  CTRL_KERNEL(jacobi24, GENERIC, DEFAULT,
2          KHitTile_float matrix, KHitTile_float matrixCopy, {
3      int x = thread_id_x;
4      int y = thread_id_y;
5
6      hit(matrix, x, y) = (hit(matrixCopy, x - 1, y) +
7                   hit(matrixCopy, x, y - 1) +
8                   hit(matrixCopy, x, y + 1) +
9                   hit(matrixCopy, x + 1, y)) / 4;
10 });
```

Listado 3.2: Ejemplo de implementación generada por el KGT del stencil descrito en el listado 3.1. Extraído de [10].

Prototipo de función del skeleton En el listado 3.3 se muestra el prototipo de la función esqueleto de EPSILOD, y la definición de los tipos para las funciones de inicialización y salida que define el usuario para su aplicación concreta. Los parámetros de la función esqueleto son los siguientes:

1. El tamaño del dominio en cada dimensión, indicado con un array de enteros.
2. La descripción del stencil:
   a) El objeto HitShape describiendo la forma del stencil.
   b) El patrón del stencil, descrito como un array de pesos para cada elemento del stencil en el shape.
   c) El factor por el que se divide la suma pesada de cada celda. Este argumento solo se utiliza con el kernel genérico y se ignora con kernels generados con KGT, que ya lo incluyen en sus expresiones optimizadas.
3. El número de iteraciones.
4. El nombre de la función stencil a utilizar, o un puntero NULL para usar el kernel genérico.
5. Un puntero a una función de inicialización para los valores del array inicial, como se muestra en la línea 2 del listado 3.3.
6. Un puntero a una función que escriba los resultados, como se muestra en la línea 3 del listado 3.3.

```
/* Type definitions included in header file */
typedef void (*initDataFunction)(HitTile_float, int, int[], int[]);
typedef void (*outputDataFunction)(HitTile_float, int, int[], int[]);

/* Parallel skeleton prototype included in header file */
void stencilComputation( int sizes[], HitShape stencilShape,
    float stencilPattern[], float factor, int numIterations,
    stencilFunction f_updateCell,
    initDataFunction f_init, outputDataFunction f_output);
```

Listado 3.3: Sección de código del archivo header de EPSILOD, contiene el prototipo de la función. Extraído de [10].

Registrar nuevos kernels Para poder utilizar un kernel dentro de la función skeleton, es necesario registrarlo. Esta operación permite a Controller identificarlo, enlazarlo y utilizarlo en los diferentes dispositivos heterogéneos. El registro se realiza mediante la macro en C "*REGISTER_STENCIL*". Esta macro recibe el nombre del kernel a registrar como argumento. Por dentro, la macro se expande en los prototipos necesarios para las funciones del modelo controllers y en una función envoltorio que lanza el kernel. El nombre del kernel se convierte, de forma transparente, en un puntero a la función envoltorio, que lanza la misma en el dispositivo mediante el framework de controllers. De esta forma se puede utilizar el nombre de la función en el esqueleto independientemente de sus implementaciones heterogéneas. Este proceso se puede observar en el listado 3.4.

```
#include "Ctrl.h"

/* Declare/Register kernel before using it */
REGISTER_STENCIL( jacobi2d4 );

/* Main program: stencil execution */
int main( int argc. char *argv[]){
    ...
    stencilComputation( sizes, shpStencil, stencilPattern, ignored_int,
        numIter, jacobi2d4, initData, outputData );
    ...
}
```

Listado 3.4: Sección de código que muestra el registro y la ejecución de un kernel. Extraído de [10].

3.2.4. Automatic Load Balancing (ALB)

La primera vez que se definió el concepto de un ALB (equilibrador automático de carga, *Automatic Load Balancing*) en las herramientas del grupo TRASGO fue en un trabajo de fin de grado [41] en el que se implementaron y probaron sobre Hitmap una serie de funciones para, a partir de una distribución de datos, generar otra donde se da más trabajo a los dispositivos de cómputo más rápidos y menos a los más lentos. No permitía el uso en sistemas heterogéneos de aceleradores como GPUs u otros de los dispositivos que se pueden utilizar en Controllers o EPSILOD, tampoco soportaba el concepto de espacios de memoria separados por dispositivo.

El mecanismo toma medidas del tiempo que se tarda en ejecutar una iteración, las comunica entre los procesos, calcula los nuevos pesos (dando más peso a los dispositivos más rápidos), genera una nueva distribución de datos y redistribuye los valores de la asignación antigua a la que acaba de generarse. En este trabajo se propusieron además distintas fórmulas para el cálculo de pesos y diferentes medias móviles. Toda esta funcionalidad se introdujo en una clase llamada HitAvg.

El mecanismo solo se implementó en una aplicación ISL, la de un stencil 2d4 para el cálculo de la difusión de calor por el método de Jacobi en dos dimensiones, introduciendo ALB. Se tomaron medidas en este ejemplo en máquinas multinúcleo y se detectaron las ventajas y las desventajas preliminares. Este mecanismo se ha probado en un artículo de congreso a posteriori [11].

Posteriormente, en mi trabajo de fin de grado [9], se tomó este mecanismo, solo existente en Hitmap, y se introdujo en Controllers, para poder utilizarse en EPSILOD. Para esto, fue necesario hacer un trabajo de reingeniería de software donde se hicieron diagramas (adaptados de UML para el uso en C) que representaban el estado original de las aplicaciones. Posteriormente, se hizo un nuevo diseño donde se añadió la funcionalidad del ALB y se implementó en las bibliotecas. Además, fue necesario arreglar algunos problemas con la implementación original en Hitmap de las funciones de ALB.

Esta aproximación contaba con una heurística planteada para elegir el mejor momento en el que realizar un equilibrado de carga, teniendo en cuenta el estado actual del sistema. Una vez decidido el momento de reequlibrado, en medio de la ejecución de un ISL mediante EPSILOD y de forma completamente transparente al usuario, utiliza funciones de Controllers (que a su vez utilizan funciones de Hitmap) para, de forma eficiente, construir nuevas estructuras de datos con una distribución de carga en los procesadores relativa a sus tiempos de ejecución.

Se realizaron pruebas y estudios estadísticos sobre este mecanismo y se llegó a la conclusión de que, con un alto grado de confianza, existían casos reales en los que la herramienta hacía un trabajo de reequilibrado que reducía los tiempos de ejecución medios de iteración, reduciendo así el tiempo total de ejecución. También se encontraron casos (en GPUs) donde se reducían los tiempos de iteración, pero, al no realizarse suficientes iteraciones, no se mejoraba el tiempo total de ejecución, pero se demostró que con aplicaciones lo suficientemente largas se podría conseguir una mejora.

En este trabajo original, sin embargo, quedaron algunos puntos como trabajo futuro que van a realizarse en este proyecto. Uno de ellos es el de la creación de un sistema para la definición y selección de heurísticas, para poder permitir la comparación entre algoritmos de decisión del momento de reequilibrado. Otro punto de trabajo futuro que se va a realizar es el de probar las herramientas en superordenadores pre-exascale.

Capítulo 4

Desarrollo

El desarrollo de este trabajo parte, como se ha mencionado previamente, de un trabajo de fin de grado realizado por el alumno. En este trabajo se desarrolló e implementó un equilibrador de carga en las librerías del grupo TRASGO, Hitmap, Controllers y EPSILOD, y se probó tanto su correcto funcionamiento como su rendimiento.

Estas bibliotecas, en especial EPSILOD, están sometidas constantemente a cambios, dado que se trata de herramientas diseñadas para la investigación. Por este motivo, de forma paralela y ya durante el trabajo del TFG, se fueron realizando cambios sobre ellas que consiguen un mejor rendimiento o añaden una mayor funcionalidad, como el posible uso con dominios de problema de más de tres dimensiones.

Una de las tareas de este trabajo fue, por tanto, realizar un merge (término proveniente de Git [22], que indica la unión de dos bases de código distintas) con las ramas en desarrollo de las bibliotecas para tener un mejor rendimiento durante las pruebas y hacerlas funcionar en la supecomputadora a utilizar, asegurando además que el trabajo realizado en el TFG y el TFM se mantiene actualizado y en uso a lo largo del tiempo.

Por otra parte, un compañero del máster ha realizado como TFM [15] la modificación de estas bibliotecas para poder añadir una serie de casos de ejemplo existentes en otras herramientas de programación creadas para resolver el mismo problema que EPSILOD. Hubo que realizar un merge con esta rama para, de esta forma, tener acceso a estos ejemplos, que ejecutan cargas de distintos tipos a las que se tenían previamente.

Posteriormente, se realizaron cambios sobre el equilibrador de carga que se había desarrollado para implementar algunas de las proposiciones de trabajo futuro que se mencionaron en el TFG.

4.1. Merge con ramas actualizadas

En esta sección se van a detallar los dos merge que se realizaron con las dos ramas más actualizadas del repositorio del grupo TRASGO, permitiendo hacer las pruebas con algunas optimizaciones que se han ido introduciendo y permitiendo que el trabajo del ALB se siga desarrollando en el grupo sin que esto cause problemas o sea necesario hacer una refactorización del código.

4.1.1. multi_ctrl

El primer merge que se realizó fue para actualizar la rama de redistribución automática de carga y que hubiera paridad entre ella y la rama de desarrollo actual. Esta rama, cuyo nombre es *multi_ctrl* [3], fue desarrollada originalmente para permitir el uso de más de un tipo de dispositivo en la ejecución de un stencil. Posteriormente, se ha seguido utilizando para, por ejemplo, la ejecución de stencils de más de tres dimensiones. Esta rama contiene una gran cantidad de cambios que han tenido que implementarse en la última versión de la rama *auto_redis*, sobre la que se está trabajando.

La mayor parte de los cambios de la última versión de la rama *multi_ctrl* se encuentran en paquetes que no se modificaron en la implementación del ALB, por ejemplo en la mayor parte del código relativo a cada acelerador de hardware o en las definiciones de los kernels. Sin embargo, fueron necesarios algunos cambios menores en el código del ALB. Por ejemplo, fue necesario modificar la forma en la que se llamaban a las funciones de controllers (para alojar la memoria, hacer subselecciones, etc).

Originalmente, en el TFG se pidió el desarrollo de una función que, mediante operaciones de los dispositivos hardware utilizados durante la ejecución, obtenía el tiempo que habían tardado en realizar la última operación solicitada. Mediante esta función, se podía calcular el tiempo que tardaba cada nodo en ejecutar su parte del trabajo, sin tener en cuenta comunicaciones.

En la versión de *multi_ctrl* del código de los dispositivos, esta función no existía. Además, de una versión a otra cambió la forma en la que se definen en el código los eventos de los aceleradores hardware (de ser específicos a cada acelerador, a ser genéricos) dentro de Controllers, de tal forma que la adaptación de la función no era directa. Por tanto, se intentó utilizar el tiempo de iteración de cada nodo, incluyendo comunicaciones.

Esto tiene una justificación, ya que las comunicaciones no dejan de ser una parte dinámica del sistema que puede encontrarse desequilibrada. Es decir, teóricamente podríamos encontrarnos un sistema donde, por ejemplo, por una avería o por tener conexiones de red de distinto tipo, un conjunto de nodos con velocidades de ejecución similares tardan tiempos muy dispares en terminar su parte del trabajo, al llegar a una sincronización. Tener en cuenta este efecto podría resultar en una mejora del rendimiento al reequilibrar.

Por desgracia, unas pruebas preliminares indicaban que el rendimiento era notablemente peor, más aún en el caso de Leonardo, ya que los tiempos de comunicación, que no presentan

grandes diferencias, eclipsaban el tiempo de ejecución. Es por esto que se pidió que se volviera a reescribir la función.

Otro pequeño cambio que se realizó fue el de realizar un merge de los cambios realizados en Hitmap en el TFG sobre la rama *master*. Se aprovechó para arreglar algunos detalles del código de Hitmap (p. ej. algún print que se encontraba fuera de lugar). Posteriormente, se actualizó la rama de hitmap de la que dependen Controllers y EPSILOD de *auto_redis* a *master*.

4.1.2. muesli_2024

Como se ha mencionado previamente, un compañero del máster estaba desarrollando su TFM [15] en la misma franja de tiempo que yo. Su trabajo era el de implementar una serie de ejemplos en EPSILOD, que se encuentran en los ejemplos que acompañan a otras herramientas de programación heterogénea, como SYCL [34] o Celerity [16]. De esta forma es posible compararlas fielmente, midiendo el rendimiento del mismo problema en la misma supercomputadora, con diferentes herramientas de programación. Para esto tuvo que hacer una serie de cambios sobre EPSILOD, por ejemplo los tipos de datos que pueden usarse en los tiles, permitiendo el uso de tipos de datos complejos.

Para poder utilizar tipos de datos complejos, en la nueva versión de EPSILOD se compila una versión distinta para cada uno de los tipos que se van a utilizar y, posteriormente, se enlazan todas estas con la aplicación final. Esto se realiza dentro del archivo CMake tal como se ve en el listado B.1 del anexo B.1. Como se ve, esta última sección compila cada archivo (p.ej. “epsilod_types” o “test_gassimulation_types”) en su propia librería. Estas librerías se enlazan con cada ejecutable mediante el código del listado B.2 del anexo B.1.

Estos archivos contienen algunos tipos de datos y macros necesarios para EPSILOD, además de un tipo de datos definible por el usuario. Uno de los macros, necesario para poder comprobar si un tipo de datos ha sido compilado y enlazado, está mal planteado en la versión original, dado que solo permite comprobar si HitTile_float está definido y no sirve para más tipos de datos. Para arreglar esto, se pasó del listado 4.1 al listado 4.3. Fue necesario arreglar este problema, ya que dentro de ALB se utiliza HitTile_double para guardar los datos de tiempo de iteración y, por tanto, si no se ha definido previamente, se debe definir.

```
1 ...
2
3 /* For conditional compilation based on the value of EPSILOD_BASE_TYPE, float
      or any other one */
4 #define EPSILOD_BASE_TYPE_float 1
5 #define EPSILOD_IS_FLOAT(type)  EPSILOD_IS_FLOAT2(type)
6 #define EPSILOD_IS_FLOAT2(type) EPSILOD_BASE_TYPE_##type
7
8 ...
```

Listado 4.1: Sección de código que muestra la macro definida por David Díez Poza en [15] para la comprobación de si el tipo de datos HitTile_float está definido.

```
1 ...
2
```

```
/* HitTile types of base type and float for stencil weights */
#if !EPSILOD_IS_FLOAT(EPSILOD_BASE_TYPE)
Ctrl_NewType(float);
#endif
Ctrl_NewType(EPSILOD_BASE_TYPE);

...
```

Listado 4.2: Sección de código que muestra el uso de la macro definida en 4.1

```
...

/* For conditional compilation based on the value of EPSILOD_BASE_TYPE, float
    or any other one */
#define EPSILOD_BASE_TYPE_FLOAT_float 1
#define EPSILOD_IS_FLOAT(type)        EPSILOD_IS_FLOAT2(type)
#define EPSILOD_IS_FLOAT2(type)       EPSILOD_BASE_TYPE_FLOAT_##type

#define EPSILOD_BASE_TYPE_DOUBLE_double 1
#define EPSILOD_IS_DOUBLE(type)         EPSILOD_IS_DOUBLE2(type)
#define EPSILOD_IS_DOUBLE2(type)        EPSILOD_BASE_TYPE_DOUBLE_##type

...
```

Listado 4.3: Sección de código que muestra la macro del listado 4.1 modificada para permitir comprobar más tipos de datos.

Por otra parte, tanto en mi trabajo de fin de grado como en el proyecto de muesli_2024 se definieron tipos de datos estructurados para facilitar el paso de parámetros a funciones de EPSILOD. En este último caso permitió además simplificar la refactorización del bucle principal de EPSILOD. Fue necesario hacer una serie de modificaciones a los tipos de datos para unificar ambas funcionalidades. El cambio principal fue la modificación de los tipos TileData y TileCommon de mi trabajo de fin de grado, que se renombraron a EpsilodTiles y EpsilodCommon y que se adaptaron para cumplir las fuciones del tipo de dato EpsilodTiles de muesli_2024. Estos tipos de datos se definen en el listado 4.4.

```
...

typedef struct EpsilodTiles {
  HitTile(EPSILOD_BASE_TYPE) * mat;
  HitTile(EPSILOD_BASE_TYPE) * inner_local;
  HitTile(EPSILOD_BASE_TYPE) * io_tile;
  HitTile(EPSILOD_BASE_TYPE) * tileBorderIn;
  HitTile(EPSILOD_BASE_TYPE) * tileBorderOut;
  HitTile(EPSILOD_BASE_TYPE) (*tileBorderOutDev)[2];
  HitPattern neighSync;
} EpsilodTiles;

typedef struct EpsilodCommon {
  int  num_tiles_total;
  int  numBorders;
  int *borderInActive;
  int *indexCommBorder;
  int  indexCommBorderCount;
  int  dims;
} EpsilodCommon;

```

```
22 ...
```

Listado 4.4: Sección de código que muestra la definición de EpsilodTiles y EpsilodCommon dentro de epsilod_structs.

En el programa se generan estos tipos de datos justo antes de llamar al bucle principal y se guardan sobre ellos los punteros de los datos generados durante la incialización. Se ha modificado esta funcionalidad para que los datos se generen directamente sobre los punteros de EpsilodTiles y EpsilodCommon, de la misma forma que se organizó en mi TFG.

Otro cambio de muesli_2024 es la posibilidad de ejecutar las inicializaciones de EPSILOD en los dispositivos aceleradores de hardware, en lugar de en el dispositivo host. Esta funcionalidad se ha añadido a las funciones de ALB. Por otra parte, en muesli_2024 se ha reestructurado completamente el bucle de EPSILOD para que las operaciones de la computación del stencil se separen cada una en su propia función (p.ej. doStep, que llama a swap, compute y a doComms). Se ha tenido que añadir la funcionalidad de ALB, que previamente se contenía en el bucle principal, al salir de la función "doStep".

Finalmente, se ha hecho un esfuerzo por arreglar todos los "*warnings*" que se mostraban al compilar esta nueva versión tras el merge. Para esto se hicieron cambios menores como, por ejemplo, cambiar la forma en la que se definían ciertas variables para no realizar "cast" innecesarios o peligrosos entre tipos de datos.

4.2. Modificaciones

En esta sección se van a detallar las modificaciones que se han realizado sobre el mecanismo de ALB que se desarrolló durante el trabajo de final de grado.

4.2.1. Cambio de estructura

En el trabajo de fin de grado se realizaron una serie de diagramas que mostraban el diseño de la herramienta que se desarrolló. Estos diagramas, al haberse realizado para un programa escrito en C99, se hicieron mediante una serie de modificaciones sobre el estándar UML, debido a que C no dispone de abstracciones como el concepto de clase o la existencia de un espacio de nombres.

Se utilizarán diagramas de actividades sin modificaciones para mostrar el flujo del código. Se utilizarán diagramas de clase donde las interfaces representan archivos ".h", las clases que las implementan representan archivos ".c" que implementan el ".h" y una interfaz que hereda de otra representa un archivo ".h" que incluye a otro. Finalmente, se utilizan diagramas de paquetes para mostrar la distribución lógica del código que, sin embargo, no se basa en un espacio de nombres. Cada paquete representa un archivo ".h" que incluye todos los relacionados con la funcionalidad asignada.

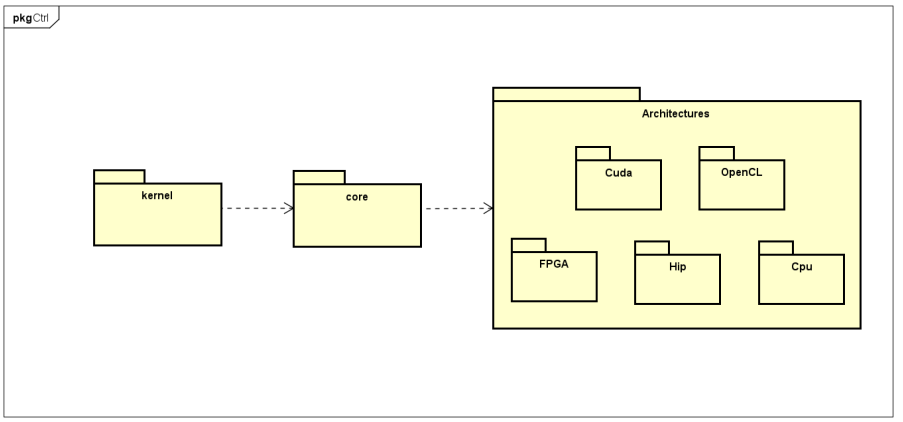

Figura 4.1: Estructura de paquetes de Controllers, extraído de [38] y modificado en [9]

En primer lugar, vamos a detallar algunos cambios que se realizaron en la estructura del código del equilibrador de carga, dado que se replanteó la forma en la que se iba a desarrollar de cara al futuro. En su versión original, las "clases" encargadas del trabajo de reequilibrado se encontraban dentro de la biblioteca Controllers (cuyo diagrama de paquetes se muestra en la figura 4.1). Estas clases relacionadas con el mecanismo ALB se encontraban distribuidas dentro del paquete "Core" según el diagrama de la figura 4.2.

Sin embargo, para este trabajo se decidió que se movería toda esta funcionalidad dentro de EPSILOD, dado que se consideró que, por la forma en la que se va a desarrollar el programa en adelante, está demasiado acoplado a EPSILOD como para mantenerse en Controllers. Se realizará una nueva versión dentro de Controllers en algún futuro, más general y que permita su uso con aplicaciones más diversas. Para empezar el rediseño, en primer lugar es necesario hacer un diagrama de clases de EPSILOD, que es el que se muestra en la figura 4.3.

En la nueva versión, todas las clases del equilibrador de carga son propias de EPSILOD. La distribución de clases de EPSILOD, con los cambios necesarios marcados con otro color, se encuentra en la figura 4.4. En primer lugar, es conveniente mover parte de las estructuras y las funciones de EPSILOD a una nueva clase, "epsilod_structs", que también contendrá algunas de las funciones previamente contenidas en ALB, por ejemplo la reserva de memoria de las EpsilodTiles. Un diagrama detallado de clases de estas dos se encuentra en la figura 4.5.

El flujo del programa se mantendrá igual a la versión sin modificar.

4.2.2. Trabajo futuro del TFG

La propuesta principal del trabajo futuro del TFG era la de la implementación de un sistema que permitiera definir varias heurísticas y la posibilidad de elegir entre ellas. Para esto, se modificó la función principal de reequilibrado, que decide automáticamente cuando

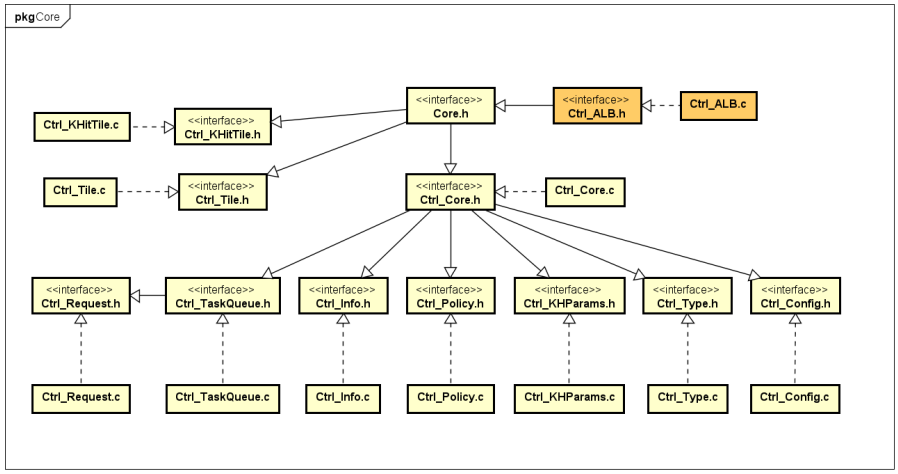

Figura 4.2: Modificación del diagrama de clases del paquete Core que muestra una variación para añadir una clase conteniendo las clases de ALB. Extraído de [9]

merece la pena reequilibrar la carga, para que llamase en varios puntos clave a otras funciones. El flujo original de la función de reequilibrado en Controllers se muestra en la figura 4.6.

Estas funciones se le mandan a la función de reequilibrado como punteros, que se ejecutan cuando la función de reequilibrado se ejecuta por primera vez, para comprobar si es necesario hacer un reequilibrado, cuando se hace un reequilibrado y en la última iteración del stencil. Esto se muestra en el diagrama de flujo de la figura 4.7.

Las funciones a las que llama se definen siguiendo la definición planteada en el listado 4.5, y se deben enviar punteros a funciones con estos esquemas al llamar a EPSILOD_ALB.

```
1  ...
2
3  // heur_initFunction will be used when first calling ALB
4  typedef void *(*heur_initFunction)();
5  // heur_checkFunction will be used every iteration to check if ALB should be
       performed
6  typedef bool (*heur_checkFunction)(void *internalState, int currentIter, int
       currentALB);
7  // heur_redisFunction will be called everytime an ALB is performed
8  typedef void (*heur_redisFunction)(void *internalState, int currentIter, int
       currentALBIter, double lastRedisSeconds, HitTile_double timesAllTimes,
       HitTile_double timesAvgTimes, HitTile_double timesRedisTimes);
9  // heur_endFunction will be called on last iter that ALB will be called
10 typedef void (*heur_endFunction)(void *internalState);
11
12 ...
```

Listado 4.5: Sección de código que muestra la definición las funciones a utilizar en EPSILOD_ALB.

Para separar las heurísticas del resto de ALB, se creó una nueva clase para definir las diferentes funciones que se deben implementar y, opcionalmente, una estructura de datos que

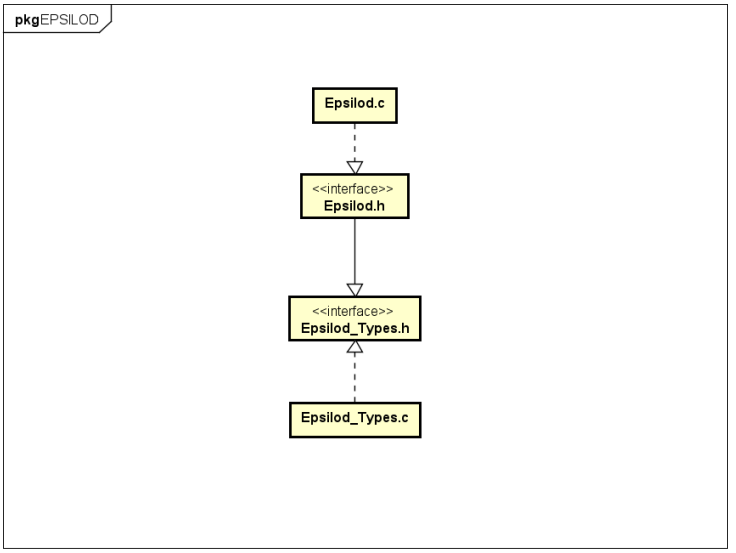

Figura 4.3: Diagrama de clases de EPSILOD

se inicializará y se conservará entre iteraciones. El diagrama final de clases de EPSILOD, con este sistema de heurísticas añadido, se muestra en la figura 4.8.

4.2.3. Mejoras del código

Se realizaron una serie de cambios sobre el equilibrador de carga para facilitar y hacer más eficiente la experimentación con él. En primer lugar, se han añadido algunas variables más a los datos que muestra el programa, una vez este ha finalizado. Algunas de estas son, por ejemplo, los valores internos de las funciones de las heurísticas.

Estas variables se imprimen por pantalla con un formato especial y, al ejecutar, se redirigen a un archivo junto con el resto del output estándar. Al tener un formato especial, es fácil filtrar estas variables mediante scripts. Se ha modificado la impresión de estas variables para, en lugar de imprimirse cuando se generan, causando un retraso en la ejecución, guardarlas en un buffer que, al terminar la ejecución, se imprime en su totalidad.

Finalmente, se cambió el código para introducir toda la recogida de variables y la impresión de estas por pantalla en una sección de código con compilación condicional. De esta forma, se puede activar y desactivar el entorno de experimentación según se necesite o no.

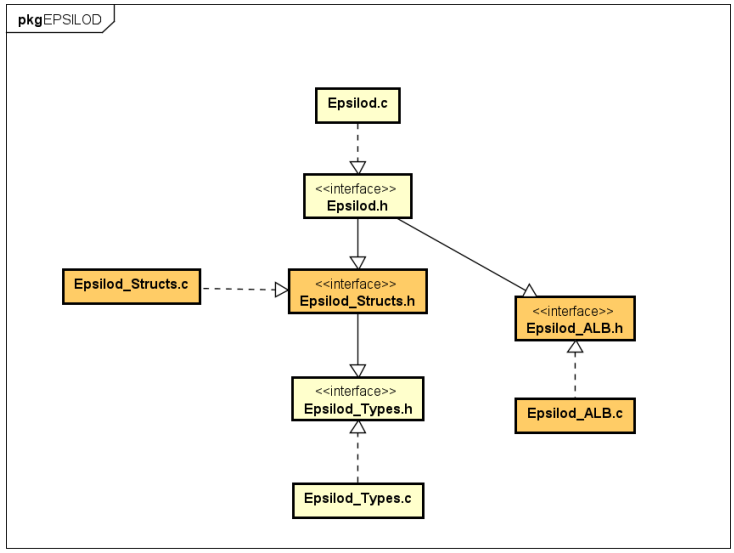

Figura 4.4: Diagrama de clases de EPSILOD modificado a partir de la figura 4.3 con los cambios necesarios para implementar las funciones de ALB

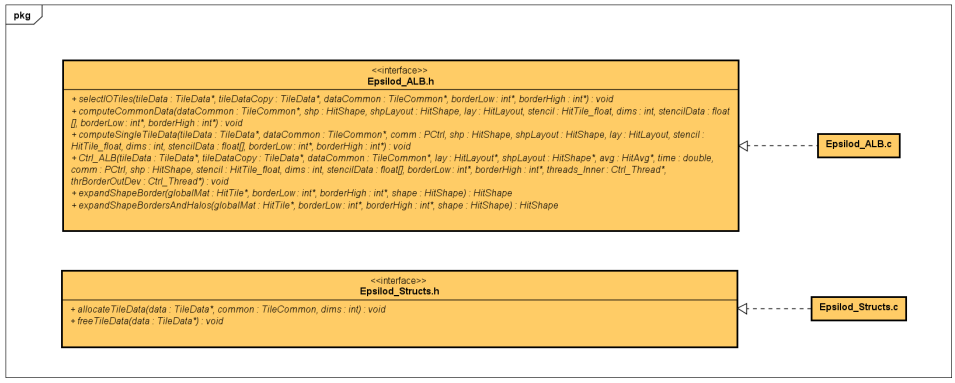

Figura 4.5: Diagrama de clases detallado de epsilod_alb y epsilod_structs

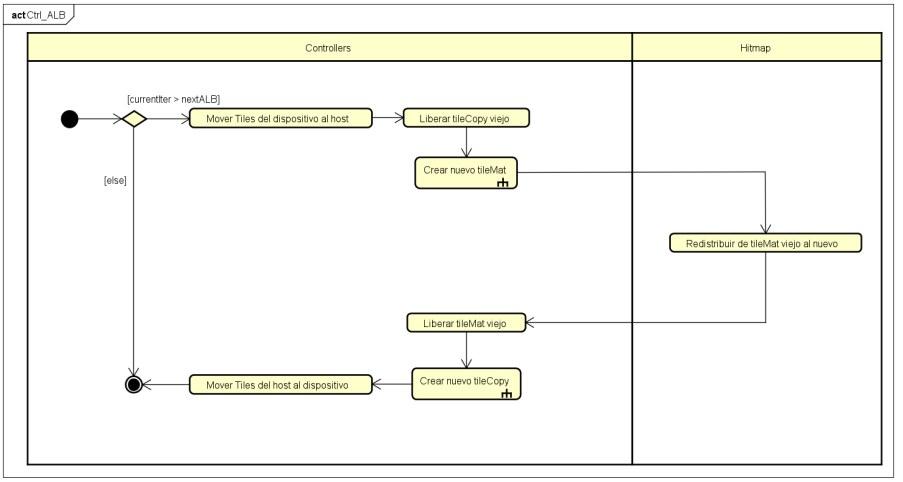

Figura 4.6: Simplificación de la función Ctrl_ALB, utilizada posteriormente en EPSILOD. Extraída de [9]

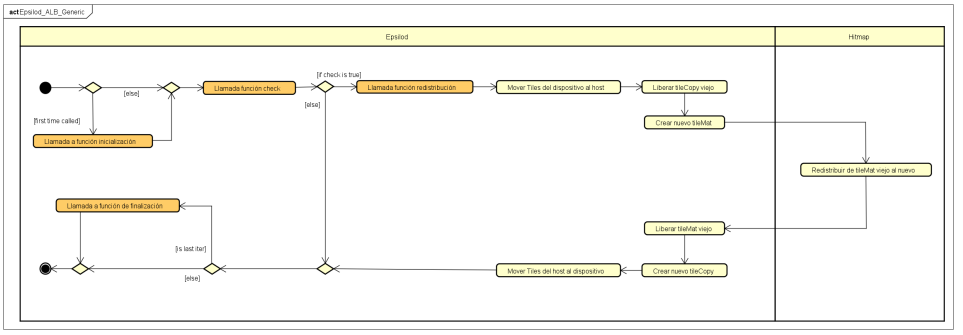

Figura 4.7: Simplificación de la función Epsilod_ALB, modificada de 4.6

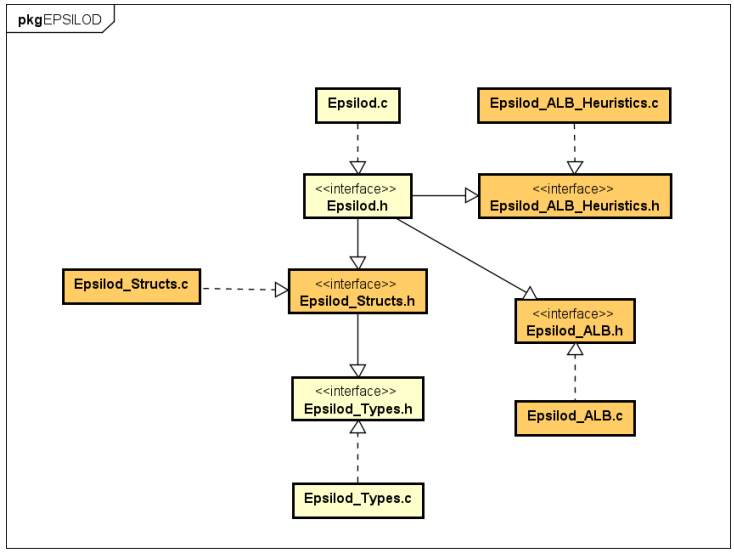

Figura 4.8: Diagrama de clases de EPSILOD modificado a partir de la figura 4.4 con los cambios necesarios para implementar los punteros a funciones de la heurística

Capítulo 5

Experimentación

5.1. Objetivos de la experimentación

Como se ha mencionado previamente, la idea del proyecto y, por tanto, de la experimentación, es doble. Por una parte, se pretende evolucionar el mecanismo de equilibrado para facilitar su uso y, por otra, se quiere probar la herramienta desarrollada en entornos distintos a los planteados originalmente en el mismo. Para esto, se han hecho modificaciones que permiten probar distintas heurísticas de decisión del momento de reequilibrado y se ha añadido un mejor soporte para realizar las pruebas en supecomputadoras pre-exaescala.

Para estas pruebas, se va a utilizar una supercomputadora europea denominada Leonardo. Es una máquina pre-exaescala, dentro del Tier 0 (el nivel más alto de categorización de infraestructuras de supercomputación). Está reconocida en el ranking TOP500 como la décima supercomputadora más potente del mundo y la quinta más potente de Europa. Se le ha dado acceso a ella al grupo TRASGO a través de la iniciativa EuroHPC.

El primer objetivo será el de definir algunas heurísticas de decisión y compararlas con el objetivo de estudiar si hay diferencias significativas entre ellas y, en caso de haberlas, cuantificarlas y descubrir los motivos que las causan.

Por otra parte, queremos estudiar el comportamiento de nuestro equilibrador de carga en máquinas con cantidades masivas de nodos idénticos. Por encima de todo, queremos comprobar si estos nodos se comportan como sistemas cerrados homogéneos o si, por el contrario, puede haber diferencias de rendimiento derivadas del estado del sistema, la temperatura en el interior de los mismos, la reducción o aumento de las frecuencias de reloj, la velocidad o congestión de las conexiones de red, etc.

Para conseguir estos objetivos, hemos diseñado una serie de experimentos que se detallarán en las siguientes secciones de este capítulo.

5.2. Definición de heurísticas

En esta primera sección vamos a definir las heurísticas que se han definido para compararlas. Estas son "*constIters*", "*doubleIters*" y "*nextALB*".

La primera heurística, cuyo pseudocódigo se puede observar en el listado 5.1, es la de constIters. En esta heurística se realiza un reequilibrado cada vez que se ejecuta un número constante de iteraciones (en las pruebas 30). Sirve de caso "base" de ejemplo a partir del cual desarrollar el resto de heurísticas y para compararlas con ellas, ya que es la más sencilla de implementar. El código real con las funciones definidas se muestra en el listado B.3 del anexo B.2.

```
func epsilod_alb:
    if first iter:
        itercounter = 0
    end

    if itercounter == 30 :
        compute weights using times
        redistribute tiles
        itercounter = 0
    end else:
        save times
        communicate values of borders to halos
        itercounter++
    end
end
```

Listado 5.1: Pseudocódigo que muestra la función de reequilibrado, con la heurística consIters configurada.

La siguiente heurística es la de doubleIters y su pseudocódigo se muestra en el listado 5.2. Esta heurística se basa en que, a medida que pasa el tiempo y se van haciendo reequilibrados, la distribución del trabajo tenderá a acercarse cada vez más a un punto ideal en el que la carga está perfectamente equilibrada. Por tanto, los reequilibrados deberían ser más importantes al principio de la ejecución, y menos al final. Cada vez que se haga un reequilibrado, el siguiente se hará en el doble del número de iteraciones que el anterior (Un ejemplo de una secuencia de este tipo podría ser: 30, 60, 120, 240, 480, etc). El código real con las funciones definidas se muestra en el listado B.4 del anexo B.2.

```
func epsilod_alb:
    if first iter:
        itercounter = 0
        nextalbcounter = 30
    end

    if itercounter == nextalbcounter:
        compute weights using times
        redistribute tiles
        nextalbcounter = nextalbcounter * 2
    end else:
        save times
        communicate values of borders to halos
        itercounter++
```

```
    end
end
```

Listado 5.2: Pseudocódigo que muestra la función de reequilibrado, con la heurística doubleIters configurada.

La heurística final, cuyo pseudocódigo se muestra en el listado 5.3, es la misma que se planteó originalmente en el Trabajo de Fin de Grado. La idea de la misma es estimar, a partir de los tiempos de ejecución por cada nodo, en cuantas iteraciones merecerá la pena hacer el siguiente reequilibrado. El código real con las funciones definidas se muestra en el listado B.5 del anexo B.2.

```
func epsilod_alb:
    if first iter:
        itercounter = 0
        nextalbcounter = 30
    end

    if itercounter == nextalbcounter:
        compute weights using times
        redistribute tiles
        nextalbcounter = redistime  / (slowesttime - avgtime)
    end else:
        save times
        communicate values of borders to halos
        itercounter++
    end
end
```

Listado 5.3: Pseudocódigo que muestra la función de reequilibrado, con la heurística nextALB configurada.

Para esto tomamos medidas del tiempo de ejecución por nodo de las últimas 30 iteraciones, para tener medidas fiables del mismo mediante el teorema central del límite. Podemos considerar que, en caso de que todos los nodos estuvieran equilibrados, el tiempo total de ejecución sería el tiempo medio de todos los nodos. Si restamos del tiempo del nodo más lento, es decir, el tiempo que se tarda actualmente (por culpa de las sincronizaciones) este valor, obtenemos la posible mejora de tiempo que conseguiríamos al reequilibrar. Si dividimos el tiempo que se tarda en reequilibrar entre esta posible mejora, obtenemos en cuantas iteraciones merece la pena realizar el próximo reequilibrado. Realizamos un mínimo de dos reequilibrados para tomar un valor fiable del tiempo que se tarda en reequilibrar,

Tiene la ventaja de que, al ser una heurística dinámica, puede reaccionar mejor a cambios en el sistema. La fórmula utilizada es la siguiente:

$$IteracionesSinReequilibrio = \frac{TiempoReequilibrio}{PeorTiempoIteracion - TiempoMedioIteracion}$$

5.3. Diseño de la experimentación

En esta sección vamos a detallar el diseño de la experimentación que se ha realizado para cumplir con los objetivos planteados en las secciones anteriores. Para esto, vamos a caracterizar Leonardo, la máquina en la que se han realizado las pruebas, y posteriormente vamos a plantear los casos de prueba que se van a ejecutar en la misma.

5.3.1. Máquinas de experimentación

Como se ha mencionado en varias ocasiones, la máquina que se va a utilizar durante las pruebas es Leonardo [14], una supecomputadora europea pre-exascale, tier 0, financiada con fondos europeos y del ministerio italiano de las universidades y la investigación. Se encuentra gestionado por CINECA y EuroHPC. En la actualidad está catalogada según la clasificación de la lista TOP500 [45]como la décima supercomputadora más potente del mundo y la quinta más potente de Europa.

Es una máquina con 4992 nodos de computación con refrigeración líquida, distribuidos en 155 racks. Alcanza los 250 petaflops y tiene 2800 terabytes de memoria RAM y 110 petabytes de memoria en disco.

Todos los nodos de la partición booster, que utilizamos para las pruebas y de los que hay un total de 3456, son idénticos y tienen cada uno la siguiente configuración:

- **P**rocesador: Intel Xeon Platinum 8358 CPU de 2.60 GHz, con 32 núcleos
- **G**ráficas: 4 x NVidia Ampere A100 customizadas con 64 GB de memoria de vídeo HBM2e y NVLink 3.0
- **M**emoria RAM: 8 x 64 GB de memoria DDR4 a 3200 MHz (para un total de 512 GB)
- **R**ed: 2 x Tarjetas con puertos duales HDR100 (400 Gbps por nodo)

Cabe destacar que existen otras máquinas dentro del clúster de CINECA y EuroHPC, como pueden ser Galileo100 (que no se ha utilizado en este trabajo) o el futuro superordenador cuántico que se construirá próximamente.

5.3.2. Diseños de experimentos

A nivel general, vamos a utilizar dos stencils y una simulación tridimensional de la dispersión de un gas. Los stencils a utilizar van a ser los siguientes:

- 2d4: Stencil de dos dimensiones, compacto y de 4 celdas (salvando la central) mostrado en la figura 5.1. Se ha escogido por ser compacto y, por tanto, de los stencils más simples y por ser un benchmark habitual en artículos y publicaciones relacionadas.

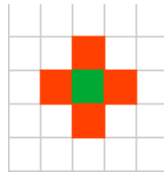

Figura 5.1: Representación del stencil 2d4, la celda verde indica el centro, las rojas las adyacentes utilizadas en el cálculo.

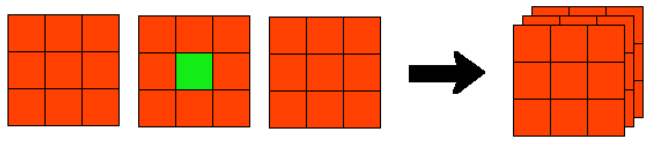

Figura 5.2: Representación del stencil 3d27, la celda verde indica el centro, las rojas las adyacentes utilizadas en el cálculo.

- 3d27: Stencil de tres dimensiones, compacto y de 27 celdas (Incluyendo la central) mostrado en la figura 5.2. Se ha escogido porque es el stencil de 3D más sencillo que incluye además esquinas, y tiene una relación entre comunicación y cómputo más elevada que los de 2D
- GasSimulation: Simulación tridimensional de la dispersión de un gas. Creada en [15]. Es computacionalmente mucho más pesada que 3d27, entre otros motivos, ya que el tipo de datos utilizado en la simulación es bastante más complejo.

En todas las pruebas vamos a utilizar todas las GPUs de cada nodo seleccionado y vamos a elegir los tamaños de matriz para maximizar la ocupación en memoria de las GPUs, con la particularidad de que se quiere tener un número de celdas en la primera dimensión que se pueda partir perfectamente entre el número de dispositivos seleccionados. Esto nos ha dado un tamaño de 46240x42640 celdas en dos dimensiones y de 1184x1184x1184 celdas en tres dimensiones.

En los stencils básicos (los dos primeros), se divide la cardinalidad de la primera dimensión menos dos, debido a que se utilizan las celdas de los bordes para definir valores constantes que simulan condiciones de contorno de Direlecht (aislando el dominio de la simulación con aportaciones de energía constantes en el contorno). Por tanto, en 2d4 y 3d27 los tamaños seleccionados para la experimentación con EPSILOD serán 46242x42642 celdas y 1186x1186x1186 celdas respectivamente.

Comparación de heurísticas

Para esta comparación vamos a tomar los dos stencils, que se consideran significativos para el estudio de la herramienta, y ejecutarlos con una cantidad razonable de 1000 iteraciones

en un único nodo, es decir, utilizando 4 GPUs. El tamaño de las matrices, como se ha dicho, se planteará para ocupar al máximo las tarjetas gráficas. Estos criterios para seleccionar las configuraciones son, según la experiencia del tutor, los adecuados para generar tiempos de ejecución experimentales suficientemente estables para un análisis estadístico concluyente.

Los stencils que se van a utilizar son los planteados previamente, el 2d4 y el 3d27. Se van a realizar 3 repeticiones de cada experimento, ya que no se quiere consumir demasiado saldo de la máquina de pruebas, que está compartida con más miembros del grupo. Se espera evaluar cuanto tiempo se pierde en el tiempo total de ejecución entre las diferentes aproximaciones. Para esto, se van a tabular los valores estadísticos más relevantes de los datos obtenidos por cada heurística, en comparación a la versión sin ALB.

Realizar esta comparación en nodos homogéneos tiene la ventaja de que, ya que se entiende que la redistribución de carga mantendrá un equilibrio similar a partir la matriz en tamaños iguales para cada nodo, las diferencias en los resultados de tiempo total están causados de forma exclusiva por la redistribución en sí.

Experimentación sobre nodos idénticos

Se van a estudiar los tiempos medios por iteración de los dos stencils y la simulación de gases. Se van a realizar 1000 iteraciones en los stencils y, debido a su alto coste computacional, solo 200 iteraciones en GasSimulation. No se van a hacer repeticiones, ya que solo se quieren estudiar los tiempos medios por iteración. Se va a utilizar tan solo la heurística nextALB.

Para este estudio, se van a mantener los tamaños de matriz, pero se van a incrementar el número de nodos, haciendo un estudio similar al que se haría para comprobar la escalabilidad fuerte del sistema (sin modificar el tamaño de la entrada cuando aumentan los nodos). Los números de nodos que se van a probar son 1, 2, 4 y 8, dando un número de GPUs de 4, 8, 16 y 32 respectivamente. Se sobreentiende, teniendo en cuenta los resultados del TFG [9], que no se va a conseguir mejorar el tiempo total de ejecución de estas aplicaciones. Por tanto, se van a tabular los estadísticos de los tiempos por iteración medias en comparación a una versión sin ALB y se van a graficar los tiempos de cada iteración de la ejecución en comparación a una versión sin ALB.

5.4. Análisis y presentación de resultados

Todas las gráficas van a mostrarse truncadas en el eje y, ya que los tiempos de redistribución son tan altos en comparación a las iteraciones normales que, si no se truncasen, los efectos que se producen en los tiempos de cada iteración de cómputo serían imposibles de observar. Los tiempos de los reequilibrados que no se pueden ver en las gráficas se detallarán aparte. Se mostrarán por pares, utilizando y sin utilizar el mecanismo de ALB. En estos casos ambas gráficas se truncarán en el mismo rango para ser comparables. Se van a marcar mediante líneas horizontales dos puntos, en rojo el tiempo medio sin utilizar el ALB, en verde utilizándolo.

En tablas se van a mostrar datos estadísticos calculados con Python [49] mediante las fórmulas del apéndice C.1. Las estadísticas que se van a calcular son la media, la desviación típica, un intervalo de confianza del 95 % de la media, la mediana, el percentil 99 y un intervalo de confianza del 95 % de la diferencia de las medias.

Capítulo 6

Resultados y discusión

En esta sección vamos a analizar los resultados de la experimentación planteada, de la forma que se describió en el apartado 5.4. Vamos a mostrar las gráficas y las tablas generadas y estudiaremos la información que nos indican los datos.

Cabe destacar, antes de empezar con la experimentación real, que se hicieron algunas pruebas preliminares, en las que se comprobó la corrección del mecanismo y de los tamaños de matrices mediante el perfilador Nsight Systems, desarrollado por NVidia. Un ejemplo de los resultados de un profiler se muestran en la figura 6.1. En esta herramienta gráfica se puede observar en una línea temporal las diferentes actividades de cómputo, comunicación, llamadas al sistema, etc. Esto ayuda a identificar el origen de los efectos observados y la interpretación de los resultados.

6.1. Comparación de heurísticas

Vamos a analizar en primer lugar los resultados de la experimentación para comparar las distintas heurísticas planteadas. Para esto, como se mencionó previamente, se van a tabular ciertos estadísticos calculados sobre el tiempo total de ejecución de los programas.

6.1.1. Stencil 2d4

Empezando por el stencil 2d4, en el primer caso con la heurística constIters mostrado en la tabla 6.1, tenemos que se tardan, como mínimo, 115 segundos más en ejecutarse la aplicación al utilizar ALB que sin utilizarse. El programa se ejecuta un 439.73 % más lento al utilizar esta heurística.

En el siguiente caso, con la heurística doubleIters mostrado en la tabla 6.2, tenemos que se tardan, como mínimo, 21 segundos más en ejecutarse la aplicación al utilizar ALB que sin

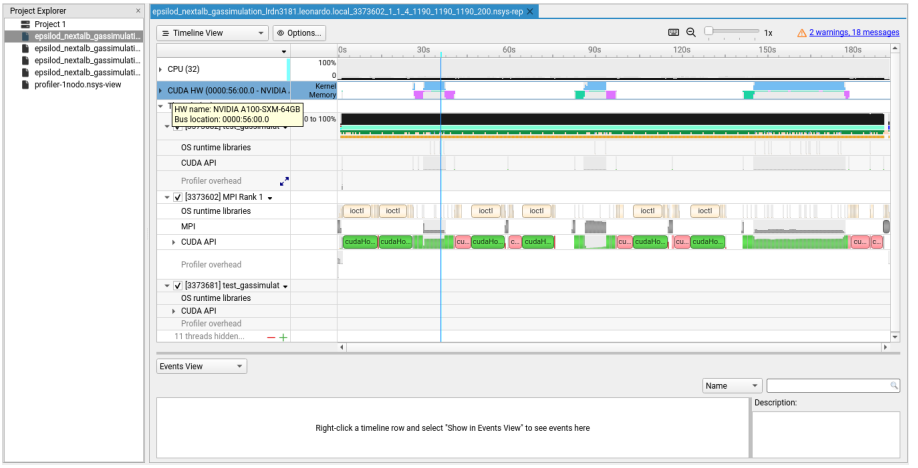

Figura 6.1: Ejemplo de uso del profiler desarrollado por NVidia, Nsight Systems

	Con ALB	**Sin ALB**
Media	142.3519 s.	26.2619 s.
Desviación estándar	0.2700 s.	0.0152 s.
Intervalo 95 %	(141.5304,143.1735) s.	(26.2155,26.3083) s.
Mediana	142.2592 s.	26.2601 s.
Percentil 99	142.7099 s.	26.2810 s.
Diferencia	(115.4814,116.6987) s.	

Tabla 6.1: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de ejecución total en segundos para el kernel 2d4 con la heurística constIters

utilizarse. El programa se ejecuta tan solo un 80.77 % más lento al utilizar esta heurística. Al reducirse con esta heurística la cantidad de reequilibrados a realizar, se tarda menos en realizar la ejecución.

Finalmente, en el último caso, con la heurística nextALB mostrado en la tabla 6.3, tenemos que se tardan, como mínimo, 6.9 segundos más en ejecutarse la aplicación al utilizar ALB que sin utilizarse. El programa se ejecuta solamente un 26.21 % más lento al utilizar esta heurística. Se puede decir con un alto grado de confianza que la propiedad de nextALB de reaccionar al sistema real evita que realice reequilibrados de más y, con ello, evita gastar demasiado tiempo en ellos.

6.1.2. Stencil 3d27

En el stencil 3d27 podemos observar los mismos efectos que en el caso del 2d4. En el primer caso, de la heurística constIters mostrado en la tabla 6.4 tenemos que se tardan, como mínimo, 88.96 segundos más en ejecutar el programa al usar el ALB, es decir, el programa

	Con ALB	**Sin ALB**
Media	47.6681 s.	26.2619 s.
Desviación estándar	0.1506 s.	0.0152 s.
Intervalo 95 %	(47.2100,48.1263) s.	(26.2155,26.3083) s.
Mediana	47.6435 s.	26.2601 s.
Percentil 99	47.8592 s.	26.2810 s.
Diferencia	(21.0656,21.7469) s.	

Tabla 6.2: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de ejecución total en segundos para el kernel 2d4 con la heurística doubleIters

	Con ALB	**Sin ALB**
Media	33.2538 s.	26.2619 s.
Desviación estándar	0.0456 s.	0.0152 s.
Intervalo 95 %	(33.1150,33.3927) s.	(26.2155,26.3083) s.
Mediana	33.2662 s.	26.2601 s.
Percentil 99	33.3018 s.	26.2810 s.
Diferencia	(6.8837,7.1002) s.	

Tabla 6.3: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de ejecución total en segundos para el kernel 2d4 con la heurística nextALB

es un 137.89 % más lento.

El siguiente caso, el de la heurística doubleIters mostrado en la tabla 6.5, es bastante mejor. Se tiene un aumento de, como mínimo, 16.45 segundos en el tiempo de ejecución, es decir, un aumento del 25.5 %.

Finalmente, una vez más el mejor caso es el de la heurística nextALB mostrado en la tabla 6.6, es bastante mejor. Se tiene un aumento de, como mínimo, 5 segundos en el tiempo de ejecución, es decir, un aumento de solamente el 7.94 %.

En general, al igual que en el caso anterior, se puede deducir que la heurística planteada originalmente [9] tiene un funcionamiento más que razonable en comparación a otras alternativas más simples.

6.2. Experimentación sobre nodos idénticos

En esta sección estudiaremos los datos derivados del segundo caso de experimentación planteado. Como se ha mencionado, se usaran tanto gráficas como tablas que representarán el tiempo invertido en realizar cada iteración.

	Con ALB	**Sin ALB**
Media	154.6736 s.	64.5177 s.
Desviación estándar	0.5299 s.	0.0168 s.
Intervalo 95 %	(153.0615,156.2856) s.	(64.4668,64.5687) s.
Mediana	154.8433 s.	64.5236 s.
Percentil 99	155.2132 s.	64.5345 s.
Diferencia	(88.9629,91.3488) s.	

Tabla 6.4: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de ejecución total en segundos para el kernel 3d27 con la heurística constIters

	Con ALB	**Sin ALB**
Media	81.1242 s.	64.5177 s.
Desviación estándar	0.0654 s.	0.0168 s.
Intervalo 95 %	(80.9253,81.3231) s.	(64.4668,64.5687) s.
Mediana	81.0828 s.	64.5236 s.
Percentil 99	81.2138 s.	64.5345 s.
Diferencia	(16.4546,16.7583) s.	

Tabla 6.5: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de ejecución total en segundos para el kernel 3d27 con la heurística doubleIters

6.2.1. Stencil 2d4

El primer caso de estudio es el del stencil 2d4 con 4 gráficas, mostrado en la tabla 6.7 y en las figuras 6.2 y 6.3. Cuando se ejecuta este caso con un único nodo, es decir, con 4 GPUs, no se tiene una diferencia estadísticamente significativa del tiempo de iteración entre la versión que utiliza el ALB y la que no. Esto quiere decir que, por lo menos en este stencil, la carga no se encuentra desequilibrada.

El siguiente caso, con 8 gráficas, se muestra en la tabla 6.8 y en las figuras 6.4 y 6.5. Tenemos la misma situación que en el anterior, donde no se detecta una diferencia estadísticamente significativa ni en las gráficas ni en los estadísticos.

El penúltimo caso para este stencil, con 16 gráficas, se muestra en la tabla 6.9 y en las figuras 6.6 y 6.7. Se detecta una ligera mejora en los tiempos de iteración tanto en los datos estadísticos como en las gráficas, alrededor de una diezmilésima de segundo. Esto parece indicar el efecto que planteábamos al principio del trabajo. Se tiene un desequilibrio leve, que parece estar causado por las condiciones actuales del sistema, y nuestro mecanismo es capaz de equilibrar la carga para solventarlo.

El último caso, con 32 gráficas, se muestra en la tabla 6.10 y en las figuras 6.8 y 6.9. Se detecta una ligera mejora en los tiempos de iteración en las gráficas, no siendo así en los datos estadísticos. Al haber aumentado el número de nodos, la mejora que se puede obtener al reequilibrar la carga de un nodo anómalo se va diluyendo entre el resto. Por esto, a un

	Con ALB	**Sin ALB**
Media	69.8484 s.	64.5177 s.
Desviación estándar	0.0908 s.	0.0168 s.
Intervalo 95 %	(69.5720,70.1248) s.	(64.4668,64.5687) s.
Mediana	69.7957 s.	64.5236 s.
Percentil 99	69.9726 s.	64.5345 s.
Diferencia	(5.1227,5.5385) s.	

Tabla 6.6: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de ejecución total en segundos para el kernel 3d27 con la heurística nextALB

	Con ALB	**Sin ALB**
Media	0.0063 s.	0.0062 s.
Desviación estándar	0.0019 s.	0.0001 s.
Intervalo 95 %	(0.0062,0.0063) s.	(0.0062,0.0062) s.
Mediana	0.0062 s.	0.0062 s.
Percentil 99	0.0063 s.	0.0062 s.
Diferencia	(-0.0000,0.0001) s.	

Tabla 6.7: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 2d4 con 4 GPUs

nivel de confianza del 95 %, no se ven efectos estadísticamente significativos.

A modo de resumen, en este stencil se ha podido comprobar que, dependiendo de los nodos con los que se ejecuten las pruebas de entre los más de 3000 de Leonardo, hay posibilidades de que encontremos alguno que ejecuta el programa levemente más lento que el resto. Nuestro mecanismo puede ayudar en estos casos y si la ejecución es lo suficientemente larga, aunque si se tienen demasiados nodos ejecutando el programa este efecto puede verse diluido.

6.2.2. Stencil 3d27

El siguiente stencil a probar es el 3d27, empezando por el caso de un solo nodo, mostrado en la tabla 6.11 y en las figuras 6.10 y 6.11. En este primer caso no se tienen diferencias estadísticamente significativas en los datos ni en las gráficas.

Continuando por el caso de dos nodos, mostrado en la tabla 6.12 y en las figuras 6.12 y 6.13, obtenemos los mismos resultados.

En el caso de utilizar cuatro nodos, mostrado en la tabla 6.13 y en las figuras 6.14 y 6.15, seguimos sin ver diferencias estadísticamente significativas.

Y en el caso final, utilizando ocho nodos, mostrado en la tabla 6.14 y en las figuras 6.16 y 6.17, vemos un leve empeoramiento del rendimiento, alrededor de una milésima de segundo.

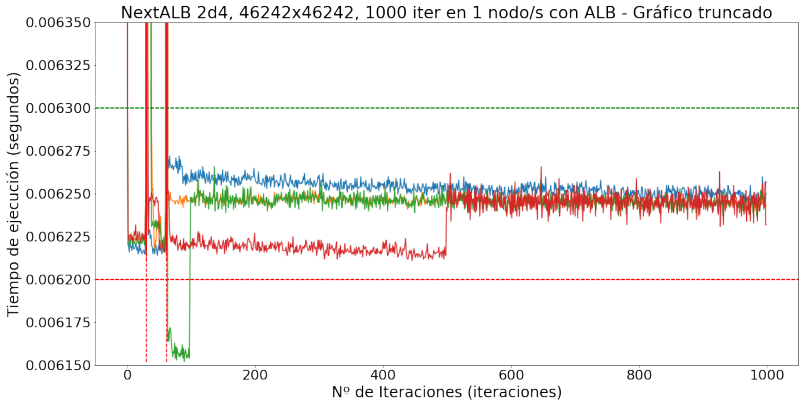

Figura 6.2: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 4 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0032 s.	0.0032 s.
Desviación estándar	0.0009 s.	0.0002 s.
Intervalo 95 %	(0.0032,0.0032) s.	(0.0032,0.0032) s.
Mediana	0.0031 s.	0.0031 s.
Percentil 99	0.0033 s.	0.0035 s.
Diferencia	(0.0000,0.0000) s.	

Tabla 6.8: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 2d4 con 8 GPUs

Como conclusión de este stencil, en contraposición al anterior, podemos sacar que el tipo de carga que se ejecuta en los nodos afecta claramente a las posibles mejoras de rendimiento que el ALB puede proporcionar en sistemas homogéneos.

6.2.3. Gas Simulation

Finalmente, tenemos el caso de la simulación de la dispersión de un gas. Precisamente por como está construido este ejemplo, con tipos de datos complejos, quizás esto implique una mayor asimetría en la ejecución, ya que observamos una mayor variabilidad. Como en estas pruebas se observa una mejora de rendimiento en las gráficas, pero no en todos los estadísticos, se intentará estudiar también el rendimiento tomando el tiempo que se ha tardado en realizar la última operación, en lugar del tiempo por iteración. De esta forma, reducimos el efecto de las comunicaciones.

El primer caso, en un solo nodo, se muestra en las tablas 6.15 y 6.16 y en las figuras 6.18,

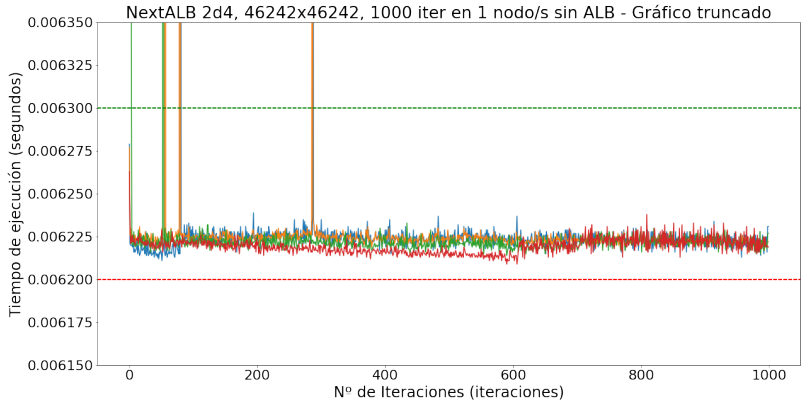

Figura 6.3: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 4 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0016 s.	0.0018 s.
Desviación estándar	0.0008 s.	0.0001 s.
Intervalo 95 %	(0.0016,0.0017) s.	(0.0018,0.0018) s.
Mediana	0.0016 s.	0.0018 s.
Percentil 99	0.0018 s.	0.0018 s.
Diferencia	(-0.0001,-0.0001) s.	

Tabla 6.9: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 2d4 con 16 GPUs

6.19 y 6.20. Encontramos en las gráficas una reducción aparente del tiempo por iteración, pero no observamos una reducción estadísticamente significativa del mismo. Sin embargo, al estudiar el tiempo de iteración sin las comunicaciones tenemos una mejora de, como mínimo, 0.0032 segundos, un 1.40 % con respecto al tiempo sin ALB.

El siguiente caso, con dos nodos, se muestra en las tablas 6.17 y 6.18 y en las figuras 6.21, 6.22 y 6.23. Tenemos el mismo efecto que en el caso anterior. Las gráficas indican una mejora que no se ve en los tiempos por iteración normales, sino que empeoran. Sin embargo, al estudiar el tiempo de iteración sin las comunicaciones tenemos una mejora de, como mínimo, 0.0012 segundos, un 1.05 % con respecto al tiempo sin ALB.

El penúltimo caso, con cuatro nodos, se muestra en las tablas 6.19 y 6.20 y en las figuras 6.24, 6.25 y 6.26. Tenemos un efecto similar. Las gráficas con tiempos de iteración esta vez no mejoran, si lo hace la que no tiene en cuenta las comunicaciones, una mejora que no se ve en los tiempos por iteración normales, sino que empeoran. Al estudiar el tiempo de iteración sin las comunicaciones tenemos una mejora de, como mínimo, 0.0005 segundos, un 0.87 % con respecto al tiempo sin ALB.

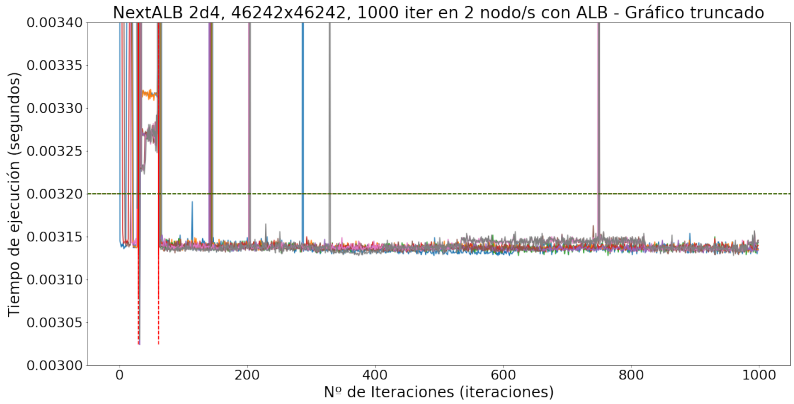

Figura 6.4: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 8 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0009 s.	0.0009 s.
Desviación estándar	0.0006 s.	0.0001 s.
Intervalo 95 %	(0.0009,0.0009) s.	(0.0008,0.0009) s.
Mediana	0.0009 s.	0.0008 s.
Percentil 99	0.0010 s.	0.0009 s.
Diferencia	(0.0000,0.0001) s.	

Tabla 6.10: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 2d4 con 32 GPUs

El último caso, con ocho nodos, se muestra en las tablas 6.21 y 6.22 y en las figuras 6.27, 6.28 y 6.29. Tenemos el mismo efecto que en el caso anterior. Solo las gráficas y los datos estadísticos sin comunicaciones muestran una mejora estadísticamente significativa. Al estudiar el tiempo de iteración sin las comunicaciones tenemos una mejora de, como mínimo, 0.0002 segundos, un 0.73 % con respecto al tiempo sin ALB.

Estos resultados corroboran los planteados en el anterior stencil, donde vemos que la carga a ejecutar y sus características influyen en el rendimiento que se puede obtener al hacer un equilibrado. Por otra parte, hemos visto como las comunicaciones pueden eliminar este efecto en los resultados finales, sobre todo al aumentar la cantidad de nodos. Y finalmente, hemos vuelto a comprobar como el hecho de aumentar la cantidad de nodos diluye las mejoras posibles del equilibrado.

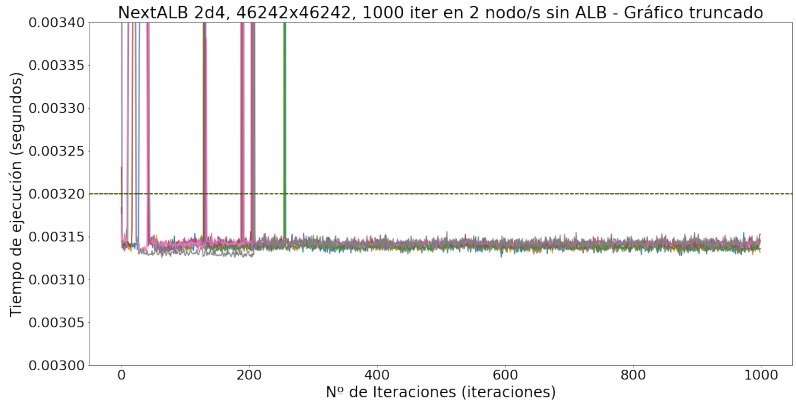

Figura 6.5: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 8 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0154 s.	0.0152 s.
Desviación estándar	0.0031 s.	0.0001 s.
Intervalo 95 %	(0.0153,0.0155) s.	(0.0152,0.0152) s.
Mediana	0.0153 s.	0.0152 s.
Percentil 99	0.0156 s.	0.0154 s.
Diferencia	(0.0001,0.0003) s.	

Tabla 6.11: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 3d27 con 4 GPUs

6.3. Nodo anómalo

Hasta ahora hemos observado la existencia de leves diferencias entre los nodos de un clúster, causadas por las condiciones actuales del sistema. Estas diferencias son, en un principio, menores, pero pueden llegar a ser particularmente notables en algunas situaciones. En general es difícil encontrar estas situaciones y es aún más difícil repetir el experimento, ya que por defecto el sistema de colas de la máquina asigna los nodos de forma dinámica a los usuarios en cada ejecución. Por tanto, es complejo solicitar una nueva ejecución en los mismos nodos y encontrarlos libres. Durante la ejecución de las pruebas finales del apartado anterior, se encontró uno de estos casos especialmente notorio.

En el experimento que discutimos, se hicieron pruebas de los stencils y la simulación de gases con 4 nodos. En uno de ellos la ejecución era particularmente lenta. Se repitieron de nuevo estas pruebas con éxito en los mismos nodos para la subsección previa y se mantuvieron los siguientes datos para su estudio posterior. Como veremos, en este caso el sistema de equilibrio de carga adapta los tamaños adecuadamente para minimizar el impacto de un

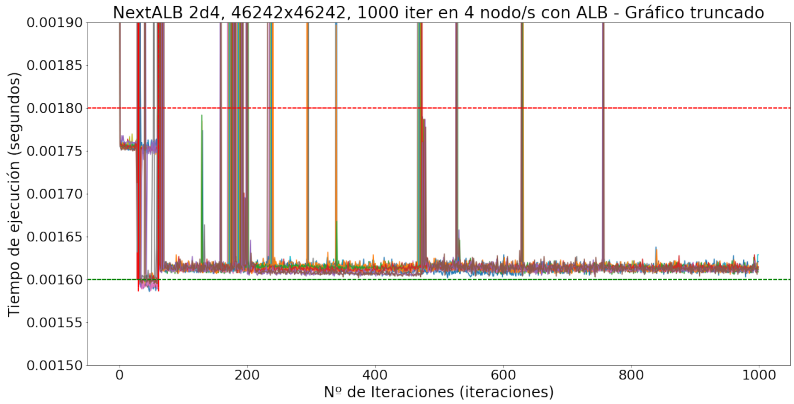

Figura 6.6: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 16 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0077 s.	0.0077 s.
Desviación estándar	0.0015 s.	0.0002 s.
Intervalo 95 %	(0.0077,0.0077) s.	(0.0077,0.0077) s.
Mediana	0.0076 s.	0.0076 s.
Percentil 99	0.0080 s.	0.0086 s.
Diferencia	(-0.0000,0.0000) s.	

Tabla 6.12: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 3d27 con 8 GPUs

dispositivo que ocasionalmente ofrece un rendimiento claramente inferior al resto, obteniendo una mejora notable del rendimiento en comparación a la práctica habitual cuando se trabaja con nodos idénticos, donde se suele asumir rendimientos similares y utilizar un reparto equitativo de la carga.

6.3.1. Stencil 2d4

El primer caso, del stencil 2d4, se muestra en la tabla 6.23 y las figuras 6.30 y 6.31. Se ven en las gráficas y en los estadísticos una mejora estadísticamente significativa de los tiempos de iteración. Se tiene una mejora del 56.7 % respecto al caso sin ALB, un ahorro de un mínimo de 0.0034 segundos. Sin embargo, esta mejora no consigue obtener los tiempos de iteración que se observan en la tabla 6.9, ya que uno de los dispositivos está teniendo un rendimiento inferior al esperado.

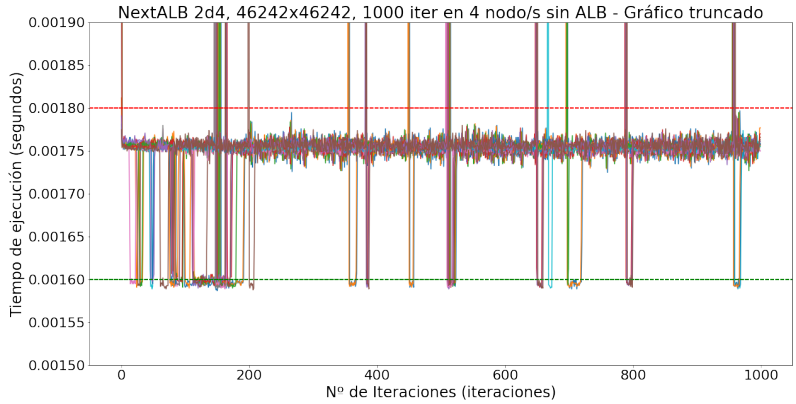

Figura 6.7: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 16 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0047 s.	0.0047 s.
Desviación estándar	0.0007 s.	0.0003 s.
Intervalo 95 %	(0.0047,0.0048) s.	(0.0047,0.0048) s.
Mediana	0.0047 s.	0.0047 s.
Percentil 99	0.0059 s.	0.0058 s.
Diferencia	(-0.0000,0.0000) s.	

Tabla 6.13: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 3d27 con 16 GPUs

6.3.2. Stencil 3d27

El segundo caso es el del stencil 3d27, que se muestra en la tabla 6.24 y las figuras 6.32 y 6.33. Al igual que en caso anterior, se ven en las gráficas y en las tablas una mejora estadísticamente significativa de los tiempos de iteración. Se tiene una mejora del 61.94 % respecto al caso sin ALB, un ahorro mínimo de 0.0096 segundos. No se consiguen alcanzar los tiempos que se tienen sin el nodo anómalo mostrado en la tabla 6.13.

6.3.3. Gas Simulation

En el último caso de la simulación de la dispersión de un gas, que se muestra en la tabla 6.25 y las figuras 6.34 y 6.35, se ven en las gráficas y en las tablas una mejora estadísticamente significativa de los tiempos de iteración, al igual que en los casos anteriores. Se tiene una mejora del 40.86 % respecto al caso sin ALB, un ahorro mínimo de 0.0802 segundos. Siguen sin igualarse los tiempos sin el nodo anómalo de la tabla 6.19.

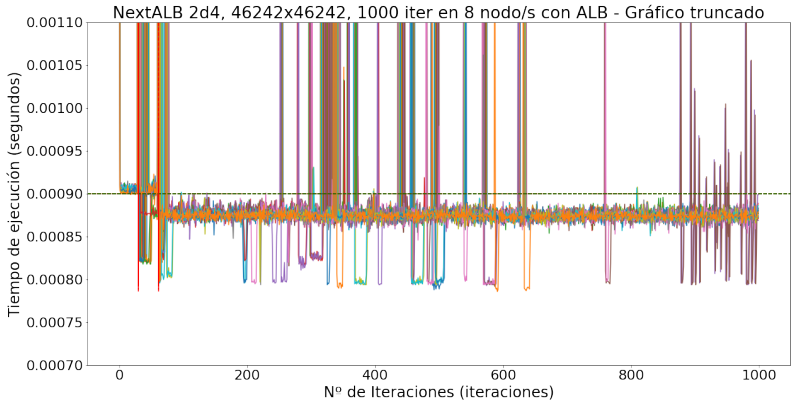

Figura 6.8: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 32 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0050 s.	0.0049 s.
Desviación estándar	0.0004 s.	0.0003 s.
Intervalo 95 %	(0.0050,0.0050) s.	(0.0049,0.0049) s.
Mediana	0.0050 s.	0.0049 s.
Percentil 99	0.0061 s.	0.0059 s.
Diferencia	(0.0001,0.0001) s.	

Tabla 6.14: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 3d27 con 32 GPUs

6.4. Tiempos de redistribución

Se observó durante la experimentación que los tiempos de redistribución eran consistentes para cada tipo de carga y cantidad de nodos utilizados. Por este motivo, el análisis de estos se encuentra en su propia sección. Merece la pena mencionar que estos tiempos de redistribución se pueden mejorar con optimizaciones en el código, algunas de las cuales ya se están realizando sobre EPSILOD.

Estos datos se muestran en la tabla 6.26 y de ellos se pueden sacar algunas conclusiones. En primer lugar, en todos los casos el tiempo para redistribuir es significativamente mayor que el tiempo medio de iteración, de ahí la importancia que se le debe dar a la optimización de esta operación. En segundo lugar, al aumentar los nodos y tener cada nodo menor parte proporcional de la carga, se tiene una disminución de la cantidad de datos que se deben redistribuir y, por tanto, del tiempo que se tarda en hacerlo. En tercer lugar, el tipo de carga es extremadamente importante, ya que se ve que, por ejemplo, la simulación de la dispersión del gas gasta mucho más tiempo en redistribuir, probablemente por su patrón de

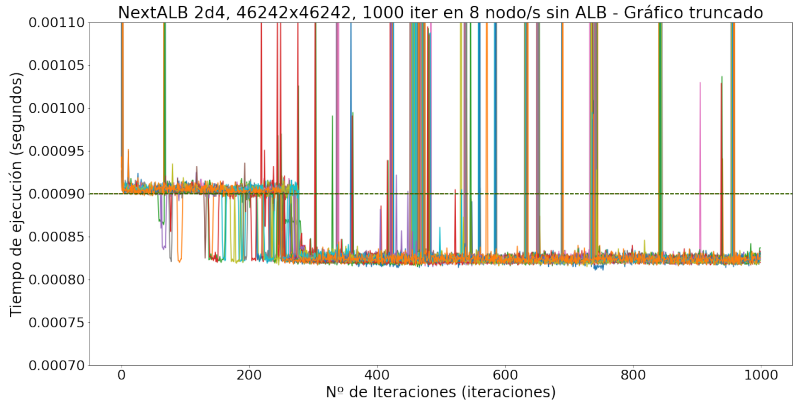

Figura 6.9: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 32 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.2344 s.	0.2309 s.
Desviación estándar	0.1207 s.	0.0004 s.
Intervalo 95 %	(0.2253,0.2436) s.	(0.2309,0.2309) s.
Mediana	0.2276 s.	0.2309 s.
Percentil 99	0.4359 s.	0.2313 s.
Diferencia	(-0.0056,0.0127) s.	

Tabla 6.15: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para la simulación de la dispersión de un gas en 4 GPUs

comunicación y por su uso de tipos de datos complejos.

6.5. Resumen y análisis de resultados

En esta sección se van a hacer un resumen de los resultados encontrados y se intentarán sacar unas conclusiones que concuerden con los mismos.

En primer lugar, tenemos el análisis de heurísticas, donde hemos podido comprobar que, en comparación a otras de las planteadas que no miden el estado actual del sistema, la que se diseñó en mi trabajo de fin de grado tiene un funcionamiento razonable. Por otra parte se ha comprobado correctamente el funcionamiento del sistema desarrollado para seleccionar heurísticas, lo que permite el desarrollo de más funciones en el futuro.

Por otra parte, tenemos la experimentación sobre nodos idénticos. En el caso de los stencils hemos comprobado que, en este tipo de nodos, es complejo mejorar el rendimiento dado que

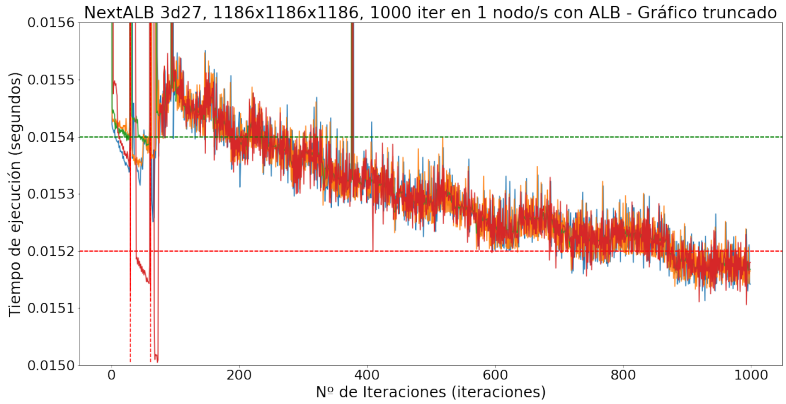

Figura 6.10: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 4 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.2260 s.	0.2293 s.
Desviación estándar	0.0014 s.	0.0004 s.
Intervalo 95 %	(0.2259,0.2261) s.	(0.2293,0.2294) s.
Mediana	0.2258 s.	0.2293 s.
Percentil 99	0.2286 s.	0.2296 s.
Diferencia	(-0.0034,-0.0032) s.	

Tabla 6.16: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos sin las comunicaciones para la simulación de la dispersión de un gas en 4 GPUs

las gráficas son extremadamente estables. Sin embargo, podemos encontrar combinaciones de nodos donde alguna de las tarjetas gráficas puede ir particularmente lenta y, reequilibrando estos casos, podemos obtener una mejora extremadamente ligera, en nuestras pruebas de alrededor de una diezmilésima de segundo. Hay que tener en cuenta de que estamos hablando de mejoras de tiempos minúsculas que, sin embargo, a lo largo de ejecuciones de miles o millones de iteraciones, pueden tener sentido.

En el caso de la simulación de la dispersión de los gases, hemos conseguido una mejora razonable y consistente del rendimiento al hacer reequilibrados. Esta mejora es estadísticamente significativa, y parece indicar al menos en parte que en este tipo de carga, con tipos de datos complejos y un kernel más complejo, se puede obtener un mayor rendimiento haciendo reequilibrados puntuales. Una comprobación con los profilers de NVidia Nsight demuestran que esta mejora es real, aunque merece la pena hacer ciertos apuntes.

Por ejemplo, en ocasiones nos encontramos con que después de un reequilibrado, los tiempos de iteración (sin comunicaciones) en las gráficas siguen pareciendo estar desequilibrados. Esto puede estar causado porque el ALB en esta iteración solo puede equilibrar datos en un

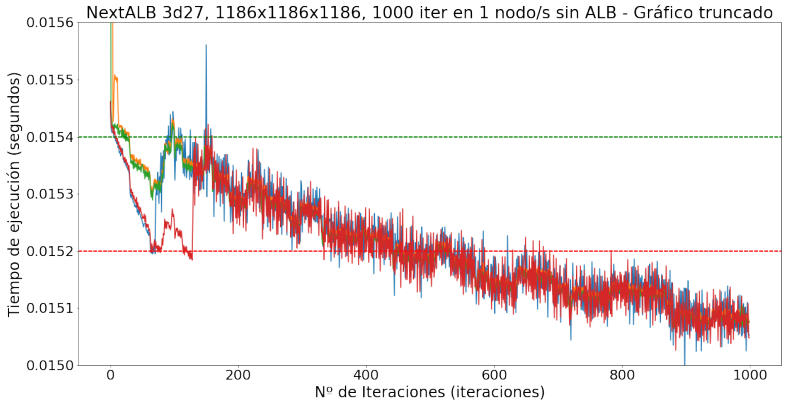

Figura 6.11: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 4 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.1178 s.	0.1155 s.
Desviación estándar	0.0333 s.	0.0001 s.
Intervalo 95 %	(0.1160,0.1196) s.	(0.1155,0.1155) s.
Mediana	0.1147 s.	0.1155 s.
Percentil 99	0.2197 s.	0.1157 s.
Diferencia	(0.0005,0.0041) s.	

Tabla 6.17: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para la simulación de la dispersión de un gas en 8 GPUs

grano demasiado grueso como para dejar la carga perfectamente colocada. Otro detalle es el de que, con el tiempo, los tiempos de iteración vuelven a su estado original. Esto parece indicar, por la forma de la gráfica, una bajada de la frecuencia de reloj de las GPUs. Aunque desconocemos los motivos que la causan, podría deberse a una subida de la temperatura en el nodo. Una mejor heurística podría detectar estos cambios en el sistema.

Finalmente, merece la pena plantear que es posible que una parte de la mejora que obtenemos esta basada, al menos en parte, en una mejor disposición de la memoria en la GPU, aunque no hemos utilizado por ahora las herramientas que podrían darnos esta información a partir de los profilers.

El último caso, en el que encontramos un nodo anómalo, es probablemente el mejor caso de uso para la herramienta en sistemas con nodos idénticos. Tenemos que uno de los nodos funciona de forma significativamente más lenta y, tras hacer un reequilibrado, conseguimos acercar los tiempos de iteración a los originales, con nodos en buen estado. Esto es importante, ya que permite hacer ejecuciones más resilientes a fallos. En caso de realizarse pruebas con miles o millones de iteraciones, se podría tener un grave problema si se tiene un nodo anómalo

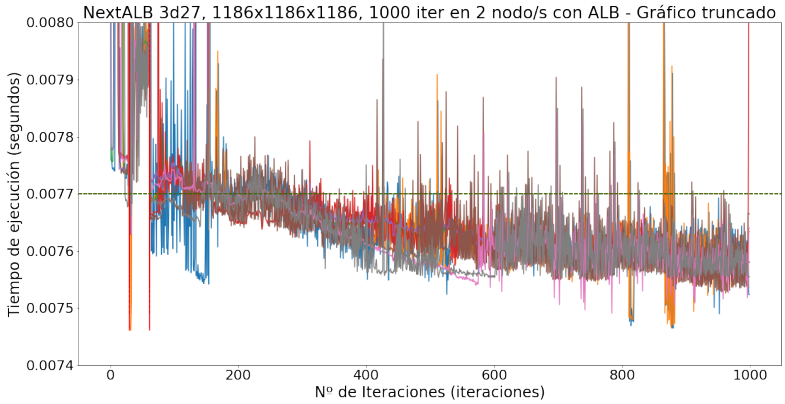

Figura 6.12: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 8 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.1127 s.	0.1139 s.
Desviación estándar	0.0010 s.	0.0001 s.
Intervalo 95 %	(0.1127,0.1128) s.	(0.1139,0.1139) s.
Mediana	0.1127 s.	0.1139 s.
Percentil 99	0.1141 s.	0.1141 s.
Diferencia	(-0.0013,-0.0012) s.	

Tabla 6.18: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos sin las comunicaciones para la simulación de la dispersión de un gas en 8 GPUs

como el que hemos encontrado, ya que tardaría significativamente más en ejecutar la prueba, con el gasto de saldo que esto conlleva. Sin embargo, con nuestro mecanismo de ALB, este problema no sería tan importante.

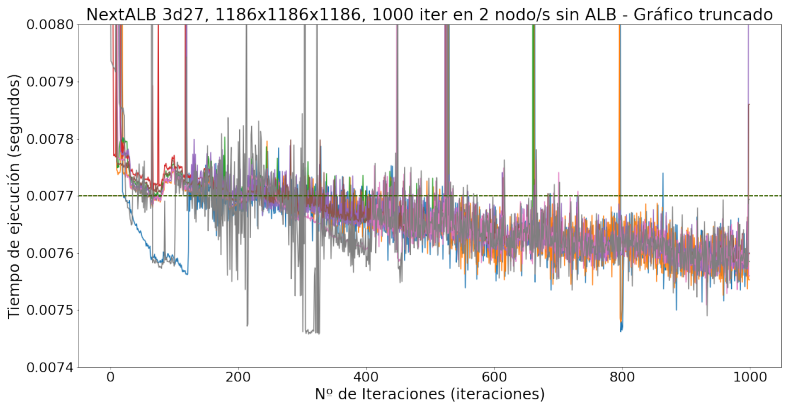

Figura 6.13: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 8 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0924 s.	0.0911 s.
Desviación estándar	0.0167 s.	0.0031 s.
Intervalo 95 %	(0.0918,0.0930) s.	(0.0910,0.0912) s.
Mediana	0.0913 s.	0.0912 s.
Percentil 99	0.1347 s.	0.0980 s.
Diferencia	(0.0007,0.0020) s.	

Tabla 6.19: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para la simulación de la dispersión de un gas en 16 GPUs

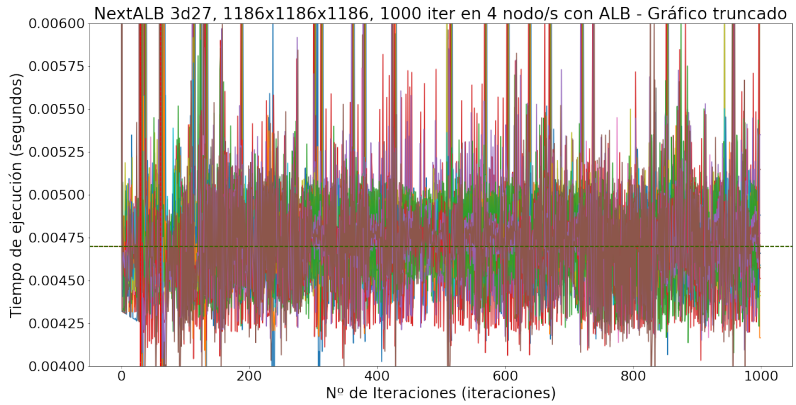

Figura 6.14: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 16 GPUs (truncados)

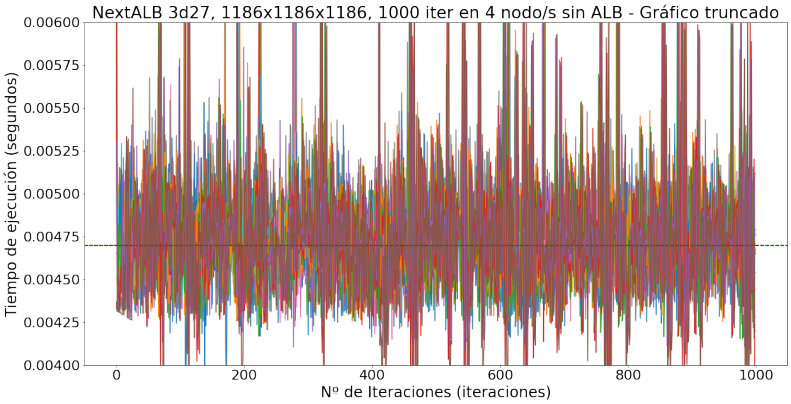

Figura 6.15: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 16 GPUs (truncados)

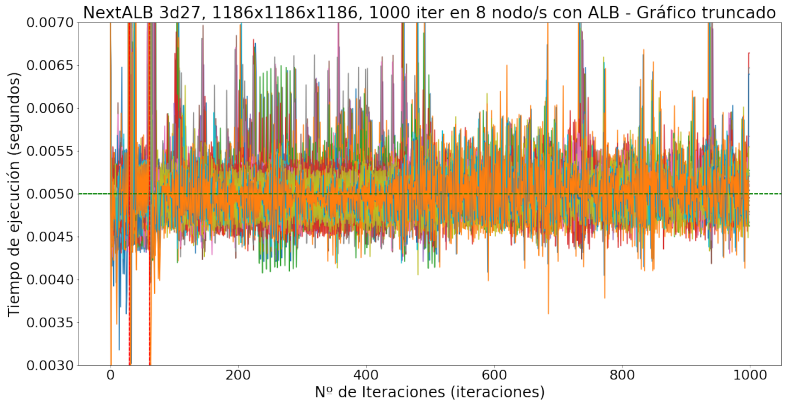

Figura 6.16: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 32 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0564 s.	0.0569 s.
Desviación estándar	0.0006 s.	0.0019 s.
Intervalo 95 %	(0.0563,0.0564) s.	(0.0568,0.0570) s.
Mediana	0.0564 s.	0.0562 s.
Percentil 99	0.0572 s.	0.0619 s.
Diferencia	(-0.0006,-0.0005) s.	

Tabla 6.20: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos sin las comunicaciones para la simulación de la dispersión de un gas en 16 GPUs

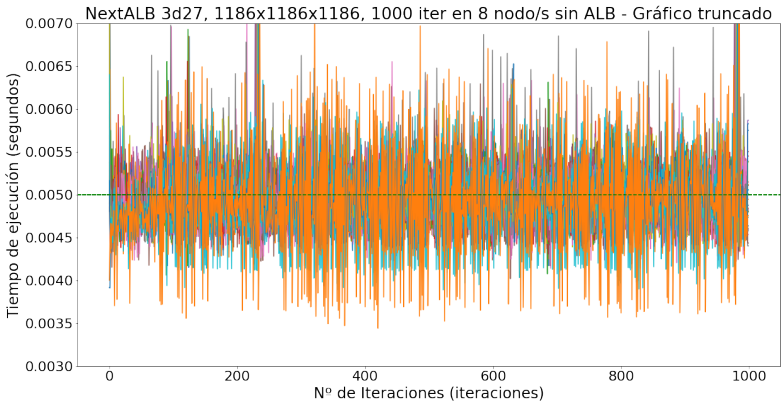

Figura 6.17: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 32 GPUs (truncados)

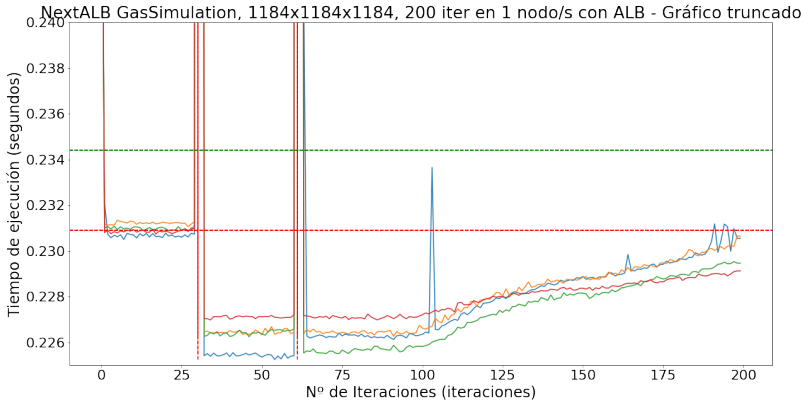

Figura 6.18: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 4 GPUs (truncados)

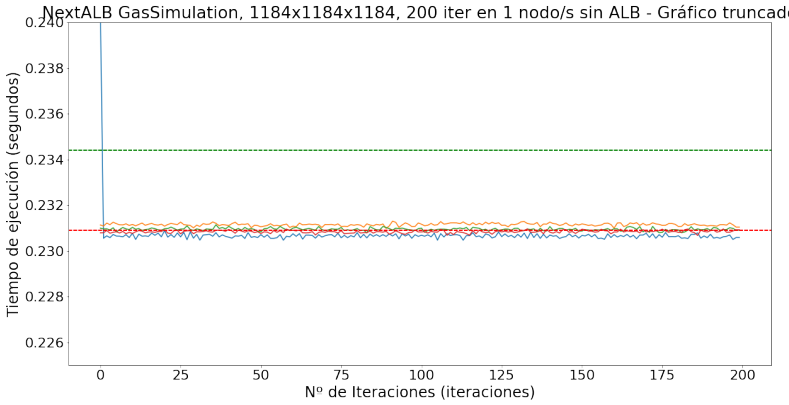

Figura 6.19: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 4 GPUs (truncados)

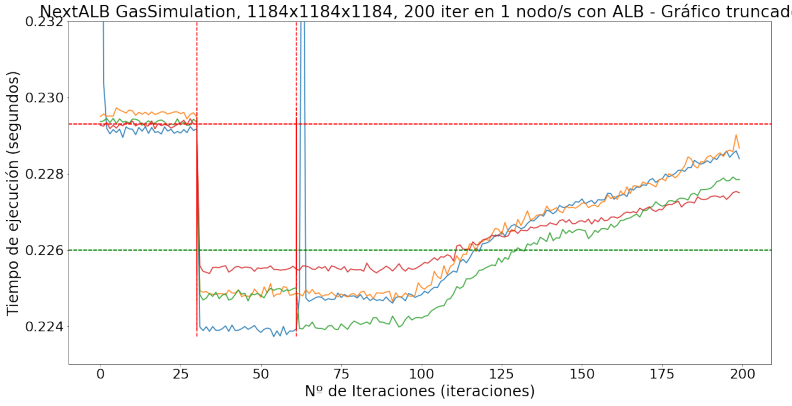

Figura 6.20: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración sin las comunicaciones en segundos/Iteración) en la simulación de la dispersión de un gas y 4 GPUs (truncados)

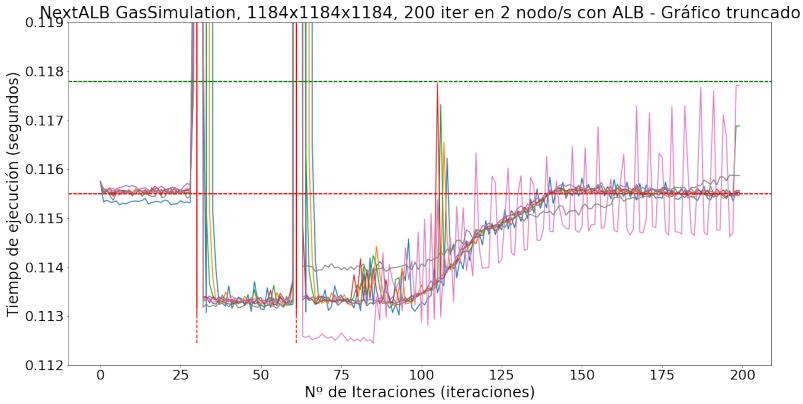

Figura 6.21: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 8 GPUs (truncados)

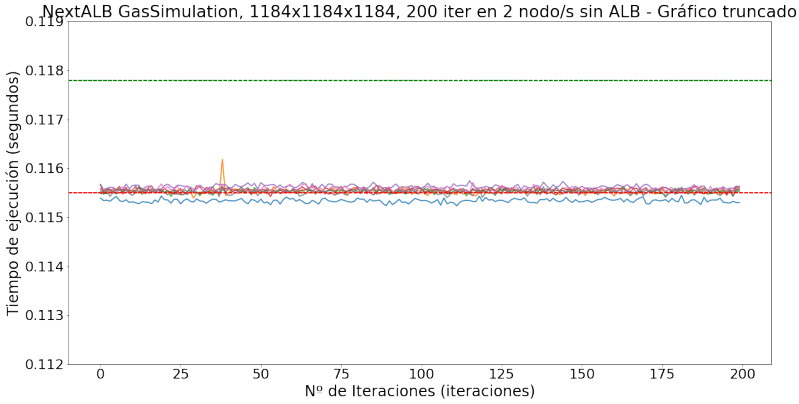

Figura 6.22: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 8 GPUs (truncados)

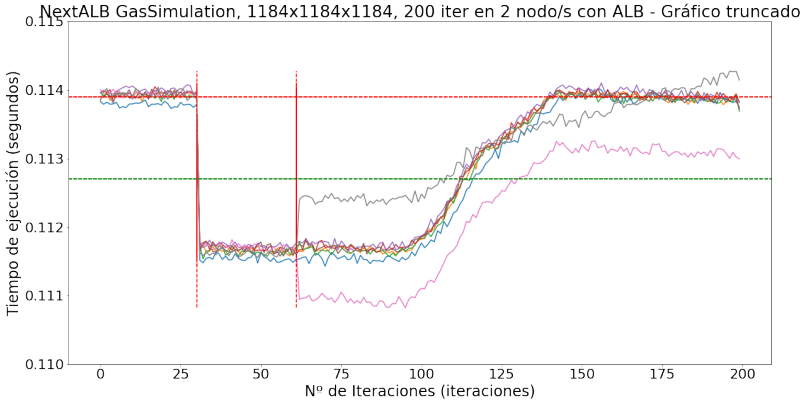

Figura 6.23: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración sin las comunicaciones en segundos/Iteración) en la simulación de la dispersión de un gas y 8 GPUs (truncados)

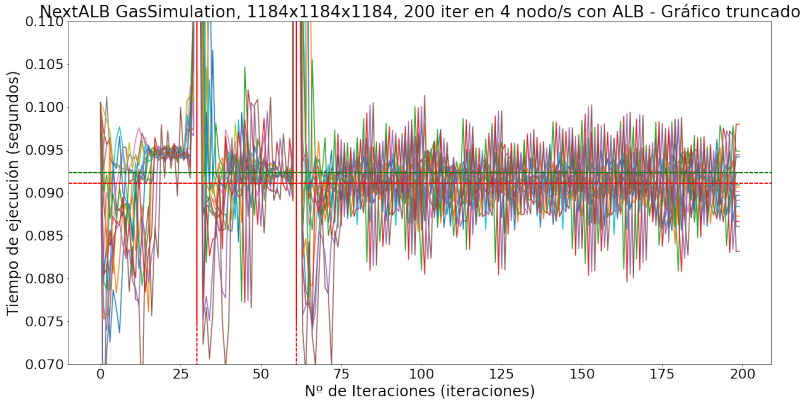

Figura 6.24: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 16 GPUs (truncados)

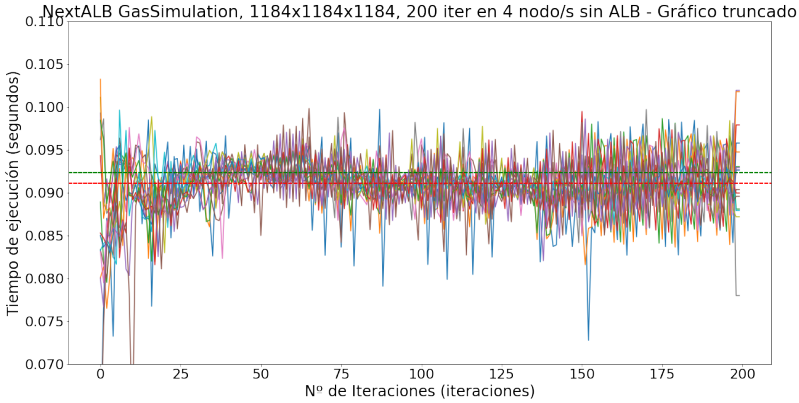

Figura 6.25: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 16 GPUs (truncados)

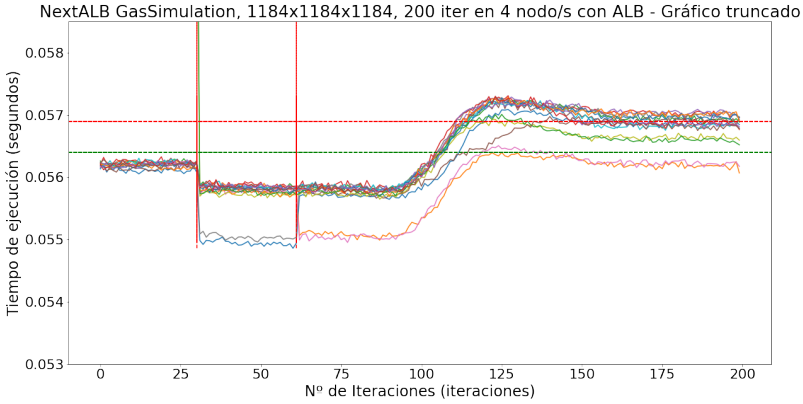

Figura 6.26: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración sin las comunicaciones en segundos/Iteración) en la simulación de la dispersión de un gas y 16 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0947 s.	0.0934 s.
Desviación estándar	0.0076 s.	0.0044 s.
Intervalo 95 %	(0.0945,0.0949) s.	(0.0933,0.0935) s.
Mediana	0.0944 s.	0.0936 s.
Percentil 99	0.1158 s.	0.1030 s.
Diferencia	(0.0011,0.0016) s.	

Tabla 6.21: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para la simulación de la dispersión de un gas en 32 GPUs

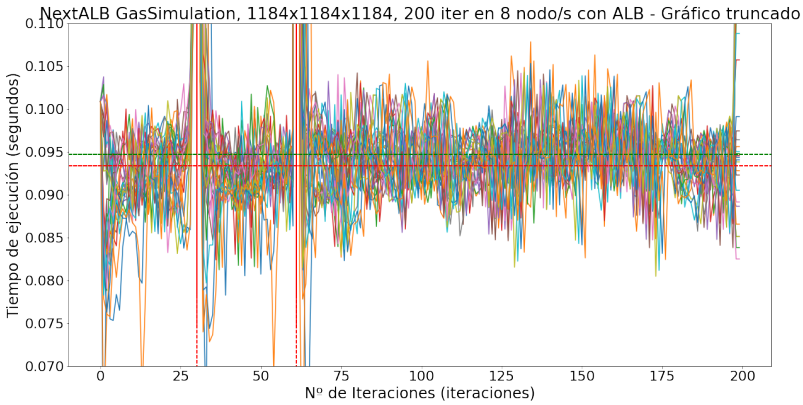

Figura 6.27: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 32 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0272 s.	0.0273 s.
Desviación estándar	0.0005 s.	0.0001 s.
Intervalo 95 %	(0.0271,0.0272) s.	(0.0273,0.0273) s.
Mediana	0.0273 s.	0.0273 s.
Percentil 99	0.0283 s.	0.0275 s.
Diferencia	(-0.0002,-0.0002) s.	

Tabla 6.22: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos sin las comunicaciones para la simulación de la dispersión de un gas en 32 GPUs

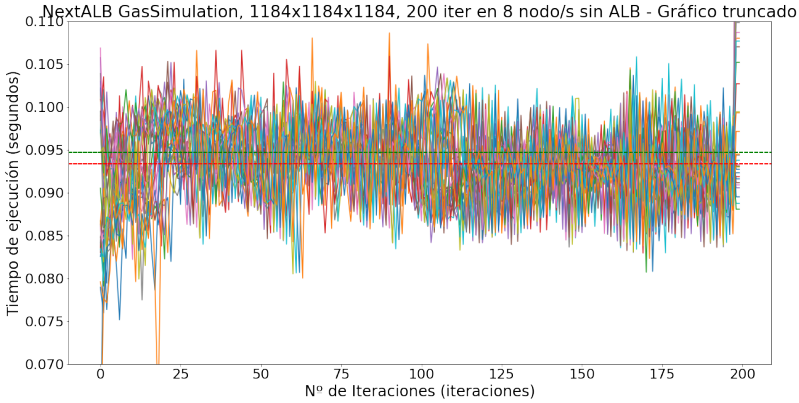

Figura 6.28: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 32 GPUs (truncados)

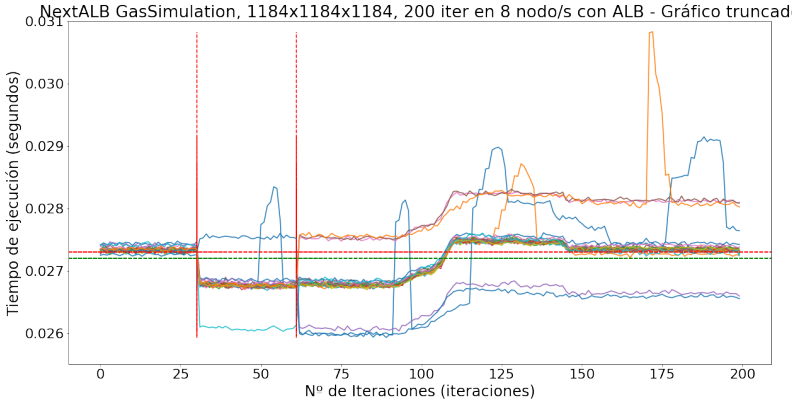

Figura 6.29: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración sin las comunicaciones en segundos/Iteración) en la simulación de la dispersión de un gas y 32 GPUs (truncados)

	Con ALB	**Sin ALB**
Media	0.0024 s.	0.0060
Desviación estándar	0.0096 s.	0.0003
Intervalo 95 %	(0.0022,0.0025) s.	(0.0060,0.0060) s.
Mediana	0.0020 s.	0.0060
Percentil 99	0.0022 s.	0.0060
Diferencia	(-0.0037,-0.0034) s.	

Tabla 6.23: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 2d4 con 16 GPUs con el nodo anómalo

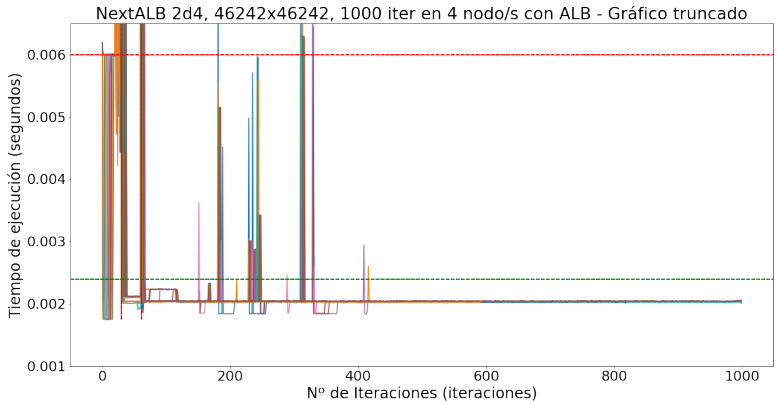

Figura 6.30: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 16 GPUs con el nodo anómalo (truncados)

	Con ALB	**Sin ALB**
Media	0.0058 s.	0.0155 s.
Desviación estándar	0.0080 s.	0.0008 s.
Intervalo 95 %	(0.0057,0.0060) s.	(0.0155,0.0156)
Mediana	0.0056 s.	0.0157 s.
Percentil 99	0.0062 s.	0.0161 s.
Diferencia	(-0.0098,-0.0096) s.	

Tabla 6.24: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para el kernel 3d27 con 16 GPUs con el nodo anómalo

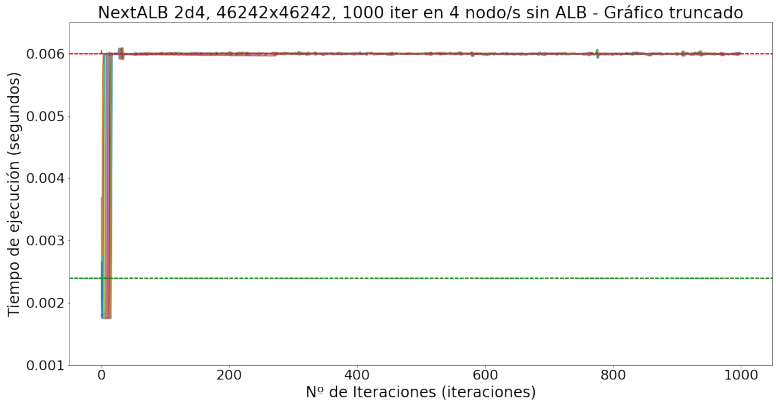

Figura 6.31: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 2d4 y 16 GPUs con el nodo anómalo (truncados)

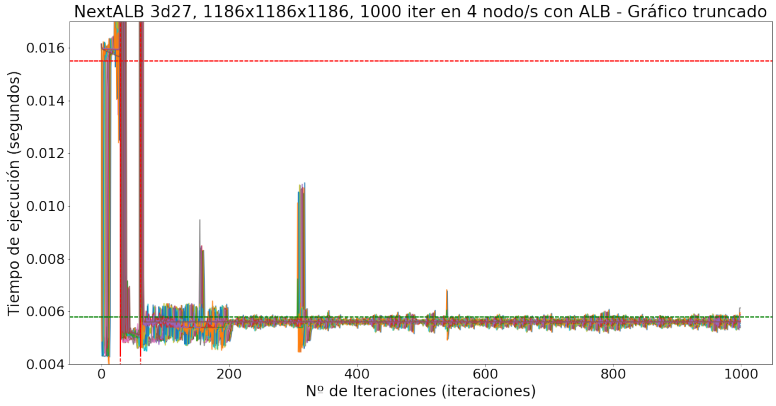

Figura 6.32: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 16 GPUs con el nodo anómalo (truncados)

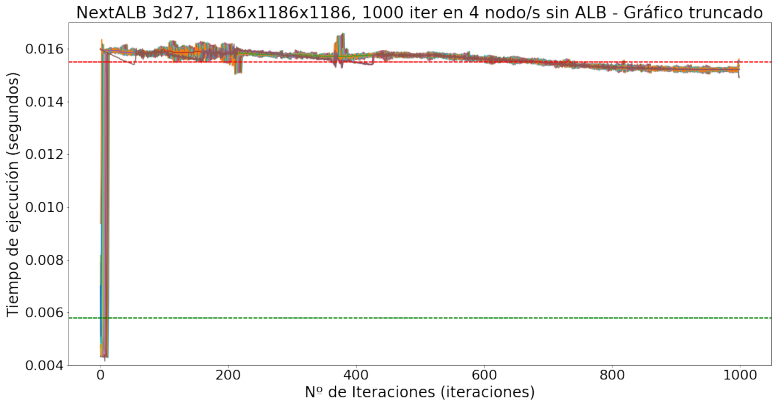

Figura 6.33: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) con kernel 3d27 y 16 GPUs con el nodo anómalo (truncados)

	Con ALB	**Sin ALB**
Media	0.1070 s.	0.1963 s.
Desviación estándar	0.2383 s.	0.0201 s.
Intervalo 95 %	(0.0980,0.1161) s.	(0.1956,0.1970)
Mediana	0.0905 s.	0.1999 s.
Percentil 99	0.1498 s.	0.2031 s.
Diferencia	(-0.0983,-0.0802) s.	

Tabla 6.25: Datos estadísticos (media, desviación estándar, intervalo al 95 % de confianza, mediana, percentil 99 y diferencia entre medias) para los datos del tiempo de iteración en segundos para la simulación de la dispersión de un gas en 16 GPUs con el nodo anómalo

	1 nodo	**2 nodos**	**4 nodos**	**8 nodos**
Stencil 2d4	3.4992 s.	1.7833 s.	0.8570 s.	0.4324 s.
Stencil 3d27	2.7215 s.	1.3685 s.	0.7091 s.	0.3660 s.
GasSimulation	46.5924 s.	23.2962 s.	12.0341 s.	6.0983 s.

Tabla 6.26: Media en segundos de los tiempos de redistribución para los casos y nodos probados. En cada columna se prueba una cantidad de nodos, en cada fila un caso.

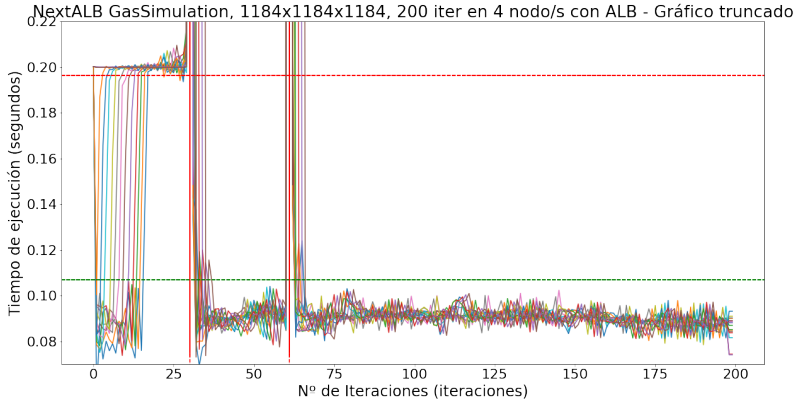

Figura 6.34: Resultados de rendimiento de la ejecución del programa con ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 16 GPUs con el nodo anómalo (truncados)

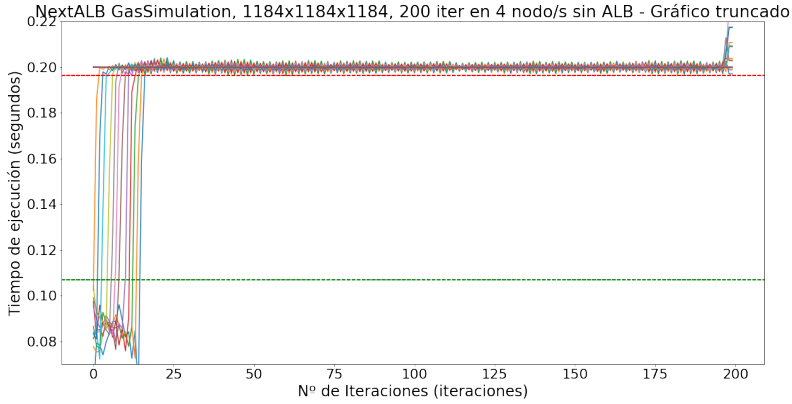

Figura 6.35: Resultados de rendimiento de la ejecución del programa sin ALB (Tiempo de iteración en segundos/Iteración) en la simulación de la dispersión de un gas y 16 GPUs con el nodo anómalo (truncados)

Capítulo 7

Conclusiones

Vamos a acabar este trabajo de fin de máster con algunos párrafos de conclusiones, describiendo los resultados obtenidos y el estado en el que se deja el proyecto. Finalmente, se añadirán algunas líneas de trabajo futuro y la valoración personal del alumno.

7.1. Conclusiones

Se considera que se han cumplido todos los objetivos que se plantearon al principio del trabajo. La actualización del sistema implementado en Hitmap, Controllers y EPSILOD integrándolo en la rama de desarrollo actual, la implementación de mejoras planteadas en el trabajo futuro del TFG y la prueba del sistema en supercomputadoras pre-exaescala.

Se han desarrollado unos cambios sobre las bibliotecas del grupo TRASGO que preparan al mecanismo para su futuro desarrollo, trabajando con varios compañeros del laboratorio.

Se ha completado el desarrollo planteado y se ha implementado un sistema extremadamente flexible para la creación y ejecución de heurísticas para la selección del momento de reequilibrado.

Se ha realizado una experimentación que ha dado información muy valiosa sobre la herramienta y su uso en sistemas con nodos idénticos. Por una parte, se ha comprobado que la heurística planteada en el trabajo original tiene un buen funcionamiento en comparación a versiones más sencillas y menos dinámicas. Por otra, se ha demostrado que existen casos en sistemas con nodos teóricamente idénticos donde, o bien por el estado del sistema, o bien por particularidades del trabajo a ejecutar, el mecanismo de equilibrado de carga da buenos resultados.

Estos resultados indican que, en general, se puede obtener una mejora leve en los tiempos de iteración que, sin embargo, a lo largo de una ejecución real, con cantidades masivas de

iteraciones, puede ser rentable. Por otro lado, se puede utilizar el mecanismo de ALB para minimizar los efectos nocivos que puede tener un nodo anormalmente lento en la ejecución.

7.2. Trabajo futuro

Quedan como trabajo futuro de este trabajo de fin de máster múltiples líneas que se pueden seguir desarrollando. Algunas de ellas son:

- Se puede utilizar el mecanismo para detectar nodos particularmente lentos y las causas de que estos se comporten de esta manera. Esta información puede ser muy valiosa para los gestores de los clústeres.
- Dado que en este estudio no se consigue reducir el tiempo total de ejecución de forma fiable por culpa de los tiempos de redistribución, se puede hacer un trabajo de optimización de esta función. Por ejemplo, el tiempo de reequilibrados se puede reducir drásticamente usando un pool de memoria en vez de liberar y reservar cada vez. Este trabajo ya está planteado en el desarrollo de EPSILOD.
- Merece la pena seguir estudiando el efecto del estado actual del sistema en estas pequeñas variaciones en la velocidad de ejecución.
- Se deben probar más heurísticas que sean capaces de, en mejor medida, detectar los desequilibrios del sistema. Quizás alguna basada en la teoría estadística, donde se hagan pruebas de hipótesis con un cierto grado de confianza de que merece la pena reequilibrar.
- Queda pendiente, también del TFG, realizar más pruebas mezclando dispositivos de diferentes naturalezas (CPUs, GPUs, etc.) en una misma ejecución.

7.3. Valoración personal

Finalmente, y a nivel personal, el valor de este trabajo es incalculable. He podido seguir trabajando con los compañeros del grupo TRASGO y con compañeros del máster en problemas de vanguardia. He estudiado el funcionamiento de supercomputadoras extremadamente potentes a nivel europeo y mundial y trabajar con ellas de primera mano. He participado y ayudado en la investigación y en escritura de artículos que avanzan el conocimiento sobre la informática.

Siendo un estudiante de ingeniería de software, esta línea de investigación es extremadamente útil para terminar de cerrar mis estudios tanto de grado como de máster, dándome nuevas perspectivas sobre la computación.

En conclusión, he tenido la oportunidad de aprender sobre tecnologías y trabajar con máquinas que, de otro modo, no habría tenido la oportunidad. He avanzado en mi desarrollo como ingeniero informático y como persona, y por ello estoy extremadamente agradecido.

Bibliografía

[1] Doaa M. Abdelkader y Fatma Omara. «Dynamic task scheduling algorithm with load balancing for heterogeneous computing system». En: *Egyptian Informatics Journal* 13.2 (2012). Accessed: 2024-10-12, págs. 135-145. ISSN: 1110-8665. DOI: https://doi.org/10.1016/j.eij.2012.04.001. URL: https://www.sciencedirect.com/science/article/pii/S1110866512000175.

[2] Alejandro Acosta, Vicente Blanco y Francisco Almeida. «Dynamic load balancing on heterogeneous multi-GPU systems». En: *Computers & Electrical Engineering* 39.8 (2013). Accessed: 2024-10-12, págs. 2591-2602. ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2013.08.004. URL: https://www.sciencedirect.com/science/article/pii/S0045790613002152.

[3] Sergio Alonso Pascual, Yuri Torres y Arturo Gonzalez-Escribano. *Sistema de ejecución y coordinación de CPUs en el modelo Controller de programación paralela heterogénea.* Accessed: 2024-10-20. URL: http://uvadoc.uva.es/handle/10324/43962.

[4] Amar Shan. *¿Heterogeneous Processing: a Strategy for Augmenting Moore's Law.* https://www.linuxjournal.com/article/8368. Accessed: 2024-06-29.

[5] Gerassimos Barlas. «Chapter 8 - Load balancing». En: *Multicore and GPU Programming.* Ed. por Gerassimos Barlas. Accessed: 2024-10-12. Boston: Morgan Kaufmann, 2015, págs. 575-628. ISBN: 978-0-12-417137-4. DOI: https://doi.org/10.1016/B978-0-12-417137-4.00008-3. URL: https://www.sciencedirect.com/science/article/pii/B9780124171374000083.

[6] Gordon Bell. «Foreword». En: *High Performance Computing.* Ed. por Thomas Sterling, Matthew Anderson y Maciej Brodowicz. Accessed: 2024-10-12. Boston: Morgan Kaufmann, 2018, págs. xix-xx. ISBN: 978-0-12-420158-3. DOI: https://doi.org/10.1016/B978-0-12-420158-3.06001-9. URL: https://www.sciencedirect.com/science/article/pii/B9780124201583060019.

[7] Christopher A. Bohn y Gary B. Lamont. «Load balancing for heterogeneous clusters of PCs». En: *Future Generation Computer Systems* 18.3 (2002). Accessed: 2024-10-12, págs. 389-400. ISSN: 0167-739X. DOI: https://doi.org/10.1016/S0167-739X(01)00058-9. URL: https://www.sciencedirect.com/science/article/pii/S0167739X01000589.

[8] Michael Boyer et al. «Load balancing in a changing world: dealing with heterogeneity and performance variability». En: *Proceedings of the ACM International Conference on Computing Frontiers*. CF '13. Accessed: 2024-10-12. Ischia, Italy: Association for Computing Machinery, 2013. ISBN: 9781450320535. DOI: 10.1145/2482767.2482794. URL: https://doi.org/10.1145/2482767.2482794.

[9] Tomás de la Cal Esteban y Arturo Gonzalez-Escribano. *Redistribución automática de carga para ejecución paralela heterogénea de aplicaciones Iterative Stencil Loop*.

[10] Manuel de Castro et al. «EPSILOD: efficient parallel skeleton for generic iterative stencil computations in distributed GPUs». En: *The Journal of Supercomputing* 79.9 (jun. de 2023). Accessed: 2024-06-29, págs. 9409-9442. ISSN: 1573-0484. DOI: 10.1007/s11227-022-05040-y. URL: https://doi.org/10.1007/s11227-022-05040-y.

[11] Manuel de Castro Caballero et al. *Exploiting highly heterogenous systems with stencil applications*.

[12] Daniel Cederman y Philippas Tsigas. «On Dynamic Load Balancing on Graphics Processors». En: *Graphics Hardware*. Ed. por David Luebke y John Owens. Accessed: 2024-10-12. The Eurographics Association, 2008. ISBN: 978-3-905674-09-5. DOI: /10.2312/EGGH/EGGH08/057-064.

[13] Long Chen et al. «Dynamic load balancing on single- and multi-GPU systems». En: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. Accessed: 2024-10-12. 2010, págs. 1-12. DOI: 10.1109/IPDPS.2010.5470413.

[14] CINECA. *Leonardo Pre-exascale Supercomputer*. Accessed: 2024-10-20. URL: https://www.hpc.cineca.it/systems/hardware/leonardo/.

[15] David Díez Poza y Arturo Gonzalez-Escribano. *Analizando la escalabilidad de aplicaciones Iterative Stencil Loop en sistemas de supercomputación*.

[16] Distributed and Parallel Systems Group, University of Innsbruck. *High-level C++ for Accelerator Clusters*. https://celerity.github.io/. Accessed: 2024-06-29.

[17] GeeksForGeeks. *Difference between Prototype Model and Incremental Model*. https://www.geeksforgeeks.org/software-engineering/difference-between-prototype-model-and-incremental-model/. Accessed: 2024-09-25.

[18] GeeksForGeeks. *Incremental Process Model - Software Engineering*. https://www.geeksforgeeks.org/software-engineering/software-engineering-incremental-process-model/. Accessed: 2024-09-25.

[19] GeeksForGeeks. *Waterfall Model - Software Engineering*. https://www.geeksforgeeks.org/software-engineering/waterfall-model/. Accessed: 2024-09-25.

[20] GeeksForGeeks. *What is Scrum in Software Development?* https://www.geeksforgeeks.org/software-engineering/scrum-software-development/. Accessed: 2024-09-25.

[21] Xiaozhong Geng, Gaochao Xu y Yuan Zhang. «Dynamic Load Balancing Scheduling Model Based on Multi-core Processor». En: *2010 Fifth International Conference on Frontier of Computer Science and Technology*. Accessed: 2024-10-12. 2010, págs. 398-403. DOI: 10.1109/FCST.2010.54.

[22] Git. *Git*. https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F. Accessed: 2024-07-05.

[23] GLASSDOOR. *Sueldos para el puesto de Ingeniero De Software Junior en España*. https://www.glassdoor.es/Sueldos/ingeniero-de-software-junior-sueldo-SRCH_KO0,28.htm. Accessed: 2024-06-29.

[24] Arturo Gonzalez-Escribano et al. «An Extensible System for Multilevel Automatic Data Partition and Mapping». En: *IEEE Transactions on Parallel and Distributed Systems* 25.5 (2014). Accessed: 2024-06-29, págs. 1145-1154. DOI: 10.1109/TPDS.2013.83.

[25] Sergei Gorlatch y Murray Cole. «Parallel Skeletons». En: *Encyclopedia of Parallel Computing*. Ed. por David Padua. Accessed: 2024-07-05. Boston, MA: Springer US, 2011, págs. 1417-1422. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_24. URL: https://doi.org/10.1007/978-0-387-09766-4_24.

[26] Heidelberg University. *AdaptiveCpp (formerly known as hipSYCL / Open SYCL)*. https://github.com/AdaptiveCpp/AdaptiveCpp. Accessed: 2024-06-29.

[27] IBM. *¿Qué es la computación de alto rendimiento (HPC)?* https://www.ibm.com/es-es/topics/hpc. Accessed: 2024-06-29.

[28] insideHPC. *What is high performance computing?* https://insidehpc.com/hpc-basic-training/what-is-hpc/. Accessed: 2024-06-29.

[29] Intel. *Quick Guide to SYCL Implementations*. https://www.intel.com/content/www/us/en/developer/articles/technical/quick-guide-to-sycl-implementations.html. Accessed: 2024-06-29.

[30] François Irigoin. «Tiling». En: *Encyclopedia of Parallel Computing*. Ed. por David Padua. Accessed: 2024-06-29. Boston, MA: Springer US, 2011, págs. 2040-2049. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_511. URL: https://doi.org/10.1007/978-0-387-09766-4_511.

[31] ISO. *ISO C Standard 1999*. Inf. téc. Accessed: 2024-06-29. 1999. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[32] Jobted. *Sueldo del Profesor de Universidad en España*. https://www.jobted.es/salario/profesor-universidad. Accessed: 2024-06-29.

[33] Yinan Ke, Mulya Agung e Hiroyuki Takizawa. «neoSYCL: a SYCL implementation for SX-Aurora TSUBASA». En: *The International Conference on High Performance Computing in Asia-Pacific Region*. HPCAsia '21. Accessed: 2024-06-29. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, págs. 50-57. ISBN: 9781450388429. DOI: 10.1145/3432261.3432268. URL: https://doi.org/10.1145/3432261.3432268.

[34] Khronos Group. *C++ Programming for Heterogeneous Parallel Computing*. https://www.khronos.org/sycl/. Accessed: 2024-06-29.

[35] Alejandro Alonso Mayo, Hector Ortega–Arranz y Arturo Gonzalez–Escribano. «Communicators: an abstraction to ease the use of hardware accelerators». En: *"HiPEAC 2016 Workshop on High-Level Parallel Programming for GPUs (HLPGPU)"* (2016). Accessed: 2024-07-05. URL: https://uvadoc.uva.es/handle/10324/29123.

[36] Ana Moreton–Fernandez, Hector Ortega–Arranz y Arturo Gonzalez–Escribano. «Controllers: An abstraction to ease the use of hardware accelerators». En: *The International Journal of High Performance Computing Applications* 32.6 (2018). Accessed: 2024-06-29, págs. 838-853. DOI: 10.1177/1094342017702962. eprint: https://doi.org/10.1177/1094342017702962. URL: https://doi.org/10.1177/1094342017702962.

[37] Borja Pérez et al. «Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems». En: *Journal of Parallel and Distributed Computing* 157 (2021). Accessed: 2024-10-12, págs. 30-42. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2021.06.003. URL: https://www.sciencedirect.com/science/article/pii/S0743731521001325.

[38] Javier Pericacho Ávila y Arturo Gonzalez-Escribano. *Integración de HIP/ROCm en un modelo de programación paralela heterogénea*. Accessed: 2024-07-05. URL: https://uvadoc.uva.es/handle/10324/63027.

[39] Hendra Rahmawan y Yudi Satria Gondokaryono. «The simulation of static load balancing algorithms». En: *2009 International Conference on Electrical Engineering and Informatics*. Vol. 02. Accessed: 2024-10-12. 2009, págs. 640-645. DOI: 10.1109/ICEEI.2009.5254739.

[40] Primož Rus, Boris Štok y Nikolaj Mole. «Parallel computing with load balancing on heterogeneous distributed systems». En: *Advances in Engineering Software* 34.4 (2003). Accessed: 2024-10-12, págs. 185-201. ISSN: 0965-9978. DOI: https://doi.org/10.1016/S0965-9978(02)00141-2. URL: https://www.sciencedirect.com/science/article/pii/S0965997802001412.

[41] María Sánchez Girón, Yuri Torres y Arturo Gonzalez-Escribano. *Distribución dinámica de carga y redistribuciones de datos en aplicaciones paralelas*. Accessed: 2024-06-29. URL: https://uvadoc.uva.es/handle/10324/44397.

[42] STE——AR GROUP. *HPX Documentation*. https://hpx-docs.stellar-group.org/latest/html/index.html. Accessed: 2024-06-29.

[43] Asser N. Tantawi y Don Towsley. «Optimal static load balancing in distributed computer systems». En: *J. ACM* 32.2 (abr. de 1985). Accessed: 2024-10-12, págs. 445-465. ISSN: 0004-5411. DOI: 10.1145/3149.3156. URL: https://doi.org/10.1145/3149.3156.

[44] Peter Thoman et al. «A taxonomy of task-based parallel programming technologies for high-performance computing». En: *The Journal of Supercomputing* 74.4 (ene. de 2018). Accessed: 2024-10-12, págs. 1422-1434. ISSN: 1573-0484. DOI: 10.1007/s11227-018-2238-4. URL: http://dx.doi.org/10.1007/s11227-018-2238-4.

[45] TOP500. *TOP500 List*. https://www.top500.org/. Accessed: 2024-06-29.

[46] Università degli Studi di Milano. *INDACO and CINECA prices*. https://www.indaco.unimi.it/index.php/prezzi-risorse-on-demand/. Accessed: 2024-09-23.

[47] US Department of Energies Exascale Project. *Kokkos Abstract*. https://kokkos.org/about/abstract/. Accessed: 2024-06-29.

[48] UVa. *Retribuciones del Profesorado Universitario, Funcionarios de Carrera*. https://transparencia.uva.es/_documentos/Retrib-31-enero.pdf. Accessed: 2024-06-29.

[49] Guido Van Rossum y Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[50] Belabbas Yagoubi y Yahya Slimani. «Dynamic Load Balancing Strategy for Grid Computing». En: *Transactions on Engineering, Computing and Technology* 13 (ene. de 2006). Accessed: 2024-10-12.

Apéndice A

Contenidos del fichero ZIP

En este apéndice se explican los contenidos del archivo .zip en el que se han comprimido los entregables del trabajo de fin de máster. Estos son:

- Controllers: contiene el código fuente descargado del gitlab del grupo Trasgo de la librería Controllers, incluyendo Hitmap y EPSILOD.
- Results: contiene los resultados de la experimentación realizada sobre Leonardo.
- Scripts: contiene los scripts mediante los que se han filtrado los datos, generado las gráficas y calculado los datos estadísticos de cada medida.

Apéndice B

Secciones de código de EPSILOD

Este apéndice contiene algunos extractos de algunas de las funciones de las librerías Hitmap, Controllers y EPSILOD. En concreto, contienen aquellas que son mencionadas durante el resto del texto. El resto de las mismas se encuentran en el código fuente de las librerías.

B.1. Código y cambios provenientes de muesli_2024

```
...

# Function to compile diferent versions of epsilod for each base type
function(add_epsilod_version type include)
  message(STATUS "Generating Epsilod Kernels for ${type}")
  if(SUPPORT_CUDA)
    add_library(epsilod_kernels_${type} OBJECT src/epsilod_kernels.cu)
  elseif(SUPPORT_HIP)
    add_library(epsilod_kernels_${type} OBJECT src/epsilod_kernels.cpp)
  else(SUPPORT_CUDA)
    add_library(epsilod_kernels_${type} OBJECT src/epsilod_kernels.c)
  endif(SUPPORT_CUDA)
  target_compile_definitions(epsilod_kernels_${type} PRIVATE EPSILOD_BASE_TYPE
    =${type} EPSILOD_TYPES_INCLUDE=\"${include}\")

  message(STATUS "Generating Epsilod for ${type}")
  add_library(epsilod_${type} OBJECT src/epsilod_structs.c src/epsilod.c src/
    epsilod_alb.c src/epsilod_alb_heuristics.c)
  target_include_directories(epsilod_${type} PRIVATE src/)
  target_compile_definitions(epsilod_${type} PRIVATE EPSILOD_BASE_TYPE=${type}
     EPSILOD_TYPES_INCLUDE=\"${include}\")
endfunction()

# Epsilod versions for different types
add_epsilod_version( float "epsilod_types.h" )
```

```
add_epsilod_version( double "epsilod_types.h" )
add_epsilod_version( cell_t "test_gassimulation_types.h" )

...
```

Listado B.1: Sección de código que muestra la compilación de EPSILOD para cada uno de los tipos de datos que se van a utilizar. Desarrollado por David Díez Poza en [15]

```
...

# Tests, examples, benchmarks, and applications
add_app( test_gassimulation cell_t "test_gassimulation_types.h" "" )
add_app( test_gaussian ${EXAMPLES_BASE_TYPE} "epsilod_types.h" "" )
add_app( test_poisson ${EXAMPLES_BASE_TYPE} "test_poisson_types.h" "" )
add_app( test_laplace ${EXAMPLES_BASE_TYPE} "test_laplace_types.h" "" )
add_app( test_wavesim ${EXAMPLES_BASE_TYPE} "test_common.h" "" )
add_app( test_parallelStencilSkeleton float "epsilod_types.h" "" )

...
```

Listado B.2: Sección de código que muestra el linkado de los ejecutables de EPSILOD para cada uno de los tipos de datos que se van a utilizar. Desarrollado por David Díez Poza en [15]

B.2. Heurísticas

```

/*
 *   heur_initFunction_constIters: The init function of the ConstIters
    strategy, in which we rebalance after a constant
 *   number of iterations
 */
int *heur_initFunction_constIters() {
  return malloc(sizeof(int));
}

/*
 *   heur_checkFunction_constIters: The check function of the ConstIters
    strategy, in which we rebalance after a constant
 *   number of iterations
 */
bool heur_checkFunction_constIters(void *internalState, int currentIter, int
    currentALB) {
  return true;
}

/*
 *   heur_redisFunction_constIters: The redis function of the ConstIters
    strategy, in which we rebalance after a constant
 *   number of iterations
 */
void heur_redisFunction_constIters(void *internalState, int currentIter, int
    currentALBIter, double lastRedisSeconds, HitTile_double allTimes,
    HitTile_double avgTimes, HitTile_double redisTimes) {
  double suma    = 0;
```

```
24   double sumaAvg = 0;
25   double peor    = 0;
26
27   for (int k = 0; k < hit_NProcs; k++) {
28     suma += hit_tileElemAt(allTimes, 1, k);
29     sumaAvg += hit_tileElemAt(avgTimes, 1, k);
30     if (hit_tileElemAt(avgTimes, 1, k) > peor) {
31       peor = hit_tileElemAt(avgTimes, 1, k);
32     }
33   }
34   double media = sumaAvg / hit_NProcs;
35   #ifdef _EPS_ALB_EXP_MODE_
36   fprintf(stream, "&2& \"constiters\",%d,%d,%d,%lf,%lf\n", hit_Rank,
       currentIter, currentALBIter, peor, media);
37   fflush(stream);
38   #endif //_EPS_ALB_EXP_MODE_
39 }
40
41 /*
42  *   heur_endFunction_constIters: The ending function of the ConstIters
       strategy, in which we rebalance after a constant
43  *   number of iterations
44  */
45 void heur_endFunction_constIters(void *internalState) {
46   free((int *)internalState);
47 }
```

Listado B.3: Sección de código que muestra la definición de las funciones de la heurística constIters

```
1
2  /*
3   *   heur_initFunction_doubleIters: The init function of the DoubleIters
       strategy, in which we rebalance after an amount of iterations that doubles
        each time
4   */
5  InternalState_DoubleIters *heur_initFunction_doubleIters() {
6    InternalState_DoubleIters *internalState = (InternalState_DoubleIters *)
       malloc(sizeof(InternalState_DoubleIters));
7    internalState->nextALB                  = 0;
8    return internalState;
9  }
10
11 /*
12  *   heur_checkFunction_doubleIters: The check function of the DoubleIters
       strategy, in which we rebalance after an amount of iterations that doubles
        each time
13  */
14 bool heur_checkFunction_doubleIters(void *internalState, int currentIter, int
       currentALB) {
15   InternalState_DoubleIters *internalStateInner = (InternalState_DoubleIters
       *)internalState;
16   return currentIter >= (internalStateInner->nextALB);
17 }
18
19 /*
20  *   heur_redisFunction_doubleIters: The redis function of the DoubleIters
       strategy, in which we rebalance after an amount of iterations that doubles
        each time
```

```
21  */
22 void heur_redisFunction_doubleIters(void *internalState, int currentIter, int
       currentALBIter, double lastRedisSeconds, HitTile_double allTimes,
       HitTile_double avgTimes, HitTile_double redisTimes) {
23   InternalState_DoubleIters *internalStateInner = (InternalState_DoubleIters
       *)internalState;
24   internalStateInner->nextALB                   = currentIter * 2;
25
26   double suma    = 0;
27   double sumaAvg = 0;
28   double peor    = 0;
29
30   for (int k = 0; k < hit_NProcs; k++) {
31     suma += hit_tileElemAt(allTimes, 1, k);
32     sumaAvg += hit_tileElemAt(avgTimes, 1, k);
33     if (hit_tileElemAt(avgTimes, 1, k) > peor) {
34       peor = hit_tileElemAt(avgTimes, 1, k);
35     }
36   }
37   double media = sumaAvg / hit_NProcs;
38   #ifdef _EPS_ALB_EXP_MODE_
39   fprintf(stream, "&2& \"doubleiters\",%d,%d,%d,%lf,%lf,%d\n", hit_Rank,
       currentIter, currentALBIter, peor, media, internalStateInner->nextALB);
40   fflush(stream);
41   #endif //_EPS_ALB_EXP_MODE_
42 }
43
44 /*
45  *   heur_endFunction_doubleIters: The ending function of the DoubleIters
       strategy, in which we rebalance after an amount of iterations that doubles
        each time
46  */
47 void heur_endFunction_doubleIters(void *internalState) {
48   InternalState_DoubleIters *internalStateInner = (InternalState_DoubleIters
       *)internalState;
49   free(internalStateInner);
50 }
```

Listado B.4: Sección de código que muestra la definición de las funciones de la heurística doubleIters

```
1
2  /*
3   *   heur_initFunction_nextALB: The init function of the NextALB strategy, in
       which we try to estimate in which iteration will
4   *   a new ALB be needed
5   */
6  InternalState_NextALB *heur_initFunction_nextALB() {
7    InternalState_NextALB *internalState = (InternalState_NextALB *)malloc(
       sizeof(InternalState_NextALB));
8    internalState->nextALB               = 0;
9    internalState->mediaRedisTime        = 0;
10   return internalState;
11 }
12
13 /*
14  *   heur_checkFunction_nextALB: The check function of the NextALB strategy,
       in which we try to estimate in which iteration will
15  *   a new ALB be needed
```

```
16  */
17 bool heur_checkFunction_nextALB(void *internalState, int currentIter, int
       currentALB) {
18   InternalState_NextALB *internalStateInner = (InternalState_NextALB *)
       internalState;
19   return currentIter >= (internalStateInner->nextALB);
20 }
21
22 /*
23  *   heur_redisFunction_nextALB: The redis function of the NextALB strategy,
       in which we try to estimate in which iteration will
24  *   a new ALB be needed
25  */
26 void heur_redisFunction_nextALB(void *internalState, int currentIter, int
       currentALBIter, double lastRedisSeconds, HitTile_double allTimes,
       HitTile_double avgTimes, HitTile_double redisTimes) {
27   InternalState_NextALB *internalStateInner = (InternalState_NextALB *)
       internalState;
28   double                  suma               = 0;
29   double                  sumaAvg            = 0;
30   double                  peor               = 0;
31   double                  redisPeor          = 0;
32
33   for (int k = 0; k < hit_NProcs; k++) {
34     suma += hit_tileElemAt(allTimes, 1, k);
35     sumaAvg += hit_tileElemAt(avgTimes, 1, k);
36     if (hit_tileElemAt(avgTimes, 1, k) > peor) {
37       peor = hit_tileElemAt(avgTimes, 1, k);
38     }
39   }
40   double media = sumaAvg / hit_NProcs;
41
42   int iters = 0;
43
44   if (lastRedisSeconds == -1) {
45     iters = 0;
46   } else if ((peor - media) != 0.0) {
47     for (int k = 0; k < hit_NProcs; k++) {
48       if (hit_tileElemAt(redisTimes, 1, k) > redisPeor) {
49         redisPeor = hit_tileElemAt(redisTimes, 1, k);
50       }
51     }
52     internalStateInner->mediaRedisTime = (((internalStateInner->mediaRedisTime
       ) * (currentALBIter - 1)) + redisPeor) / (currentALBIter);
53     iters                              = (internalStateInner->mediaRedisTime)
       / (peor - media);
54   }
55
56   internalStateInner->nextALB = currentIter + iters;
57   #ifdef _EPS_ALB_EXP_MODE_
58   fprintf(stream, "&2& \"nextalb\",%d,%d,%d,%lf,%lf,%lf,%d,%d\n", hit_Rank,
       currentIter, currentALBIter, internalStateInner->mediaRedisTime, peor,
       media, iters, internalStateInner->nextALB);
59   fflush(stream);
60   #endif //_EPS_ALB_EXP_MODE_
61 }
62
63 /*
64  *   heur_endFunction_nextALB: The ending function of the NextALB strategy, in
```

```
     which we try to estimate in which iteration will
 *    a new ALB be needed
 */
void heur_endFunction_nextALB(void *internalState) {
  InternalState_NextALB *internalStateInner = (InternalState_NextALB *)
    internalState;
  free(internalStateInner);
}
```

Listado B.5: Sección de código que muestra la definición de las funciones de la heurística nextALB

Apéndice C

Secciones de código del script de Python

Este apéndice contiene algunos extractos de algunas de las funciones de los scripts de Python utilizados para el cálculo estadístico. El resto de las mismas utilizadas para, entre otras funciones, graficar los datos se encuentran en el código fuente del trabajo.

C.1. Scripts de cálculo de estadísticos

```
    ...
    def manMedia(data):
        #Convertimos los datos en un dataframe
        data = pd.Series(data=data)
        return float(data.sum()/data.count())
    ...
```

Listado C.1: Sección de código python del cálculo de la media.

```
    ...
    def manVar(data):
        data = pd.Series(data=data)
        med = manMedia(data)
        data = data-med
        data = data**2
        return float(data.sum()/data.count())
    ...
```

Listado C.2: Sección de código python del cálculo de la varianza.

```
    ...
    def manCVar(data):
        data = pd.Series(data=data)
        med = manMedia(data)
```

```
        data = data-med
        data = data**2
        return float(data.sum()/(data.count()-1))
        ...
```

Listado C.3: Sección de código python del cálculo de la cuasivarianza.

```
    ...
    def manDesv(data):
        data = pd.Series(data=data)
        return float(pow(manVar(data),1/2))
    ...
```

Listado C.4: Sección de código python del cálculo de la desviación estándar.

```
    ...
    def manIntervaloMediaCalcStudent(data,dataLength):
        student = scipy.stats.t.ppf(1-0.025, dataLength-1)
        data = pd.Series(data=data)
        return "("+"{:.4f}".format(manMedia(data)-student*(manDesv(data)/(pow(
    data.count()-1,1/2))))+","+"{:.4f}".format(manMedia(data)+student*(manDesv
    (data)/(pow(data.count()-1,1/2))))+")"
    ...
```

Listado C.5: Sección de código python del cálculo del intervalo de confianza de la media.

```
    ...
    def fCalc(data1, data2):
        data1 = pd.Series(data=data1)
        data2 = pd.Series(data=data2)
        numer = (manCVar(data1)/data1.count())+(manCVar(data2)/data2.count())
        numer = pow(numer,2)
        frac1 = pow(manCVar(data1)/data1.count(),2)/(data1.count()+1)
        frac2 = pow(manCVar(data2)/data2.count(),2)/(data2.count()+1)
        return float((numer/(frac1+frac2))-2)
    ...
```

Listado C.6: Sección de código python del cálculo del valor f.

```
    ...
    def manDiffMed(data1, data2, f):
        data1 = pd.Series(data=data1)
        data2 = pd.Series(data=data2)
        mult = manCVar(data1)/data1.count() + manCVar(data2)/data2.count()
        mult = pow(mult,1/2)
        med = manMedia(data1) - manMedia(data2)
        return "("+("{:.4f}".format(med-f*mult))+","+("{:.4f}".format(med+f*
    mult))+")"
    ...
```

Listado C.7: Sección de código python del cálculo del valor del intervalo de confianza de la diferencia entre las medias.