Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática
(Mención Ingeniería de Software)

Abstracciones para el cálculo automático de comunicaciones en computación paralela distribuida

Autor:
Rubén Carrasco Fernández

Tutores:
Arturo González Escribano
Yuri Torres de la Sierra

Resumen

La computación paralela es una rama de la informática que busca extraer el paralelismo inherente de las aplicaciones dividiendo el cómputo en subtareas que se ejecutarán a la vez, mejorando así el rendimiento. Su uso ha ganado importancia en los últimos años debido al aumento de las capacidades de la tecnología. Presenta aplicaciones en diversos campos, entre los que destacan las simulaciones científicas.

El tiling es una técnica utilizada en la computación paralela que busca, mediante la reescritura de bucles o el reparto de estructuras de datos, dividir el cómputo en partes más pequeñas, cuya unión constituya el resultado final. Un ejemplo de programa que se beneficia del tiling son los stencils. Estos son algoritmos iterativos en los cuales cada valor de una matriz se calcula a partir de sus elementos adyacentes en la matriz de la iteración anterior, utilizando operaciones matemáticas como la media aritmética o ponderada.

La biblioteca Hitmap implementa la técnica del tiling ofreciendo funciones para crear, manipular y comunicar estructuras de datos de forma eficiente. Sin embargo, la definición de estas comunicaciones es tediosa, incluyendo una gran cantidad de parámetros que deben ser determinados por el programador, suponiendo una fuente de errores. Por ello, en este trabajo se propone una extensión para la biblioteca Hitmap que permita calcular las comunicaciones de manera automática.

Abstract

Parallel computing is a field of computer science that seeks to extract the inherent parallelism of applications by dividing the computation in subtasks that will be run simultaneously, thus improving the performance. It has grown in importance over the last few years thanks to the capabilities of the new technologies. It has applications in different fields, especially in scientific simulations.

Tiling is a technique used in parallel computing that uses loop rewriting or data structure distribution to divide the computation in smaller parts, whose union makes the final result. An example of program that benefits from tiling is a stencil. Those are iterative algorithms in which each value of a matrix its computed based on its adjacent elements in the matrix of the previous iteration, using mathematical operations such as the weighted average.

The Hitmap library implements tiling offering functions to create, manipulate and communicate data structures efficiently. However, the definition of this communications is tedious, including many parameters that must be determined by the programmer, resulting in error prone code. For this reason, this project presents an extension for the Hitmap library that provides the automatic creation of communications.

Índice

Lista de figuras

Lista de tablas

Lista de algoritmos

Lista de códigos

Capítulo 1

Introducción

En este capítulo se tratarán los siguientes puntos:

- El contexto en el que se enmarca el proyecto.
- El estado del arte y las soluciones similares a la presentada en este trabajo.
- La motivación que lleva al desarrollo de este proyecto
- Los objetivos propuestos para el trabajo.
- La estructura de este documento.

1.1 Contexto

Esta sección sitúa el proyecto en su contexto, explicando qué es la computación paralela, la disciplina de la informática con la que se relaciona este proyecto. También proporcionará una definición para los sistemas heterogéneos y la técnica de partición de datos llamada tiling.

1.1.1 Computación paralela

La computación paralela [1] es una rama de la informática que busca extraer el paralelismo inherente de cada aplicación para lograr maximizar el rendimiento aprovechando al máximo las capacidades del sistema.

Desde sus inicios y hasta la actualidad, a pesar de los avances en la tecnología, la programación siempre ha sido principalmente secuencial, imitando el comportamiento interno de los procesadores así como nuestra forma de pensar. Esto se traduce en que la mayoría de algoritmos y programas se entienden como una lista de instrucciones que deben ser ejecutadas de manera ordenada, iniciando cada una cuando termina la anterior.

Si bien esta estructura secuencial facilita el diseño de algoritmos, supone malgastar gran parte de los recursos de las máquinas actuales, que disponen de un gran número de núcleos. Según las estadísticas [2], en mayo de 2022 solo un 11.05% de los procesadores utilizados en ordenadores personales tenían menos de 4 núcleos (es decir, 2 o 1). Este estudio muestra que los números de núcleos más habituales son 4 (30.58%), 8 (25.13%) y 6 (22.99%). Los procesadores mononúcleo están prácticamente extintos (en informática doméstica), con un 0.16%.

Es por este motivo que surge la programación paralela. Esta visión de la programación divide los problemas en tareas más pequeñas, las cuales pueden realizarse a la vez. Sin embargo, la implementación física dista de la visión teórica, ya que no siempre es posible asignar cada una de estas tareas paralelas a un procesador físico para que se ejecute concurrentemente. Las implementaciones más habituales del paralelismo se consiguen con procesos e hilos, que se diferencian en que en los hilos comparten espacio de direcciones y memoria, por lo que pueden acceder a los mismos datos [3].

Hay que destacar que el proceso de división en tareas paralelas no es tan sencillo como podría parecer, ya que es habitual que surjan dependencias en esta división del trabajo, es decir, que un proceso necesite un resultado que ha sido calculado por otro. Nace así la necesidad de las comunicaciones, intercambios de datos entre procesos de forma que todos dispongan de las entradas que necesitan para sus cálculos. Estas comunicaciones pueden implementarse de diversas formas, donde destaca la memoria compartida y el paso de mensajes [4].

En el primer caso, se supone la existencia de una región de memoria que puede ser accedida por todos los procesos, tanto para realizar lecturas como escrituras. Esto se traduce en que todos trabajan sobre el mismo conjunto de datos.

En el segundo cada proceso tiene sus propios datos (solo aquellos que le han sido asignados), por lo que si quiere conocer algún valor perteneciente a otro proceso deberá iniciar una comunicación, es decir, realizar un par envío-recepción.

La programación paralela tiene multitud de aplicaciones en la actualidad, entre las que destacan las relacionadas con ciencia o ingeniería, donde se necesita realizar simulaciones de escenarios que requieren un elevado número de cálculos. En concreto, algunos usos de esta tecnología son la predicción y modelado de fenómenos como terremotos o el clima, ciencias como la física, química o astrofísica y aplicaciones industriales como investigación de materiales o aeronáutica.

Muchas de estas aplicaciones se implementan por medio de los denominados stencils. Estos se definen como matrices cuyo valor se calcula de manera iterativa, obteniendo cada elemento a partir de los elementos contiguos a él en la iteración anterior.

1.1.2 Sistemas heterogéneos

Los sistemas heterogéneos son, como su nombre indica, aquellos constituidos por máquinas de diversas características hardware. Para poder aprovechar la potencia de este tipo de sistemas es necesario realizar un reparto en tareas más pequeñas entre las distintas unidades de cómputo para reducir el tiempo de ejecución de dicha aplicación de forma proporcional al número de tareas creadas. Esta división da lugar a un trabajo más especializado de forma que cada tarea se adapte a las características de la máquina que le sea asignada. Por tanto, los sistemas heterogéneos constituyen una rama de la computación paralela.

La lista TOP 500 [5], que recoge los ordenadores más potentes del mundo cada 6 meses, muestra la tendencia al alza de estos sistemas. La mayoría de ellos contienen nodos que usan una arquitectura heterogénea internamente, al emplear tanto CPUs como GPUs para los cálculos.

En dicha lista se muestran también computadores con un nivel más alto de heterogeneidad, ya que contienen nodos con tecnologías diferentes cuyo uso conjunto permite aumentar su puntuación en el ranking. Perlmutter, que ocupa el puesto 5 en la lista, destaca sobre los demás. Como la mayoría de sus compañeros de ranking, utiliza nodos heterogéneos con CPUs y GPUs, pero además

dispone de distintos tipos de nodos como se muestra en la figura 1.1, cada uno con un propósito específico. Además, explota una de las ventajas de las arquitecturas heterogéneas, la capacidad de expansión, ya que parte de sus nodos serán instalados en el futuro [6].

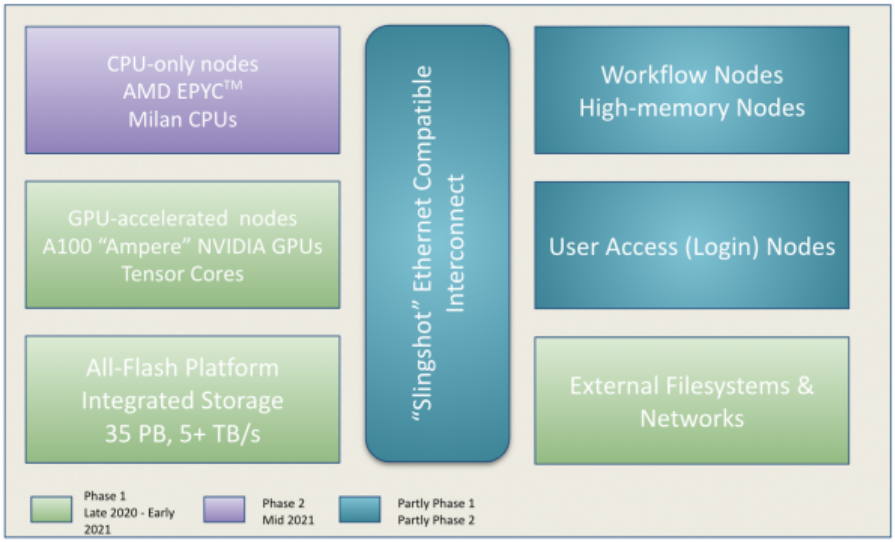


Figura 1.1: Tipos de nodos del Perlmutter (extraído de [6])

Uno de los motivos más importantes por los que se usa esta técnica es que permite escalar la potencia más allá de los límites de la tecnología, es decir, si el procesador del ordenador que se está utilizando para ejecutar un programa no es suficientemente potente no es necesario buscar un procesador con el doble de capacidades (que es posible que no exista). En su lugar basta con conectar dos máquinas y repartir el cómputo entre ellas (aunque teniendo en cuenta que la relación entre el tiempo de ejecución y el número de máquinas no es estrictamente lineal).

Otra ventaja importante es la flexibilidad que aportan los sistemas heterogéneos para su construcción, ya que se pueden emplear distintas máquinas con diversas capacidades, edades e incluso tecnologías, siempre y cuando exista un soporte para ellas en el software. No es necesario por tanto disponer de máquinas clónicas, y es fácilmente expansible en el futuro.

Aunque este trabajo se centra en la división de tareas idénticas en una alto número de procesos para mejorar sus tiempos de ejecución, los sistemas heterogéneos también son una buena elección para aquellas tareas que necesitan distintos tipos de computación. Si son suficientemente complejas y variadas, es imposible encontrar la plataforma perfecta (hardware, compilador, sistema operativo...), por ello se opta por dividir la tarea en subtareas diferentes, centrándose cada una de ellas en un tipo de programación (tareas vectoriales, de operaciones con números decimales, de acceso a datos...). Esta división permite asignar a cada tarea la máquina que consiga el menor tiempo de ejecución posible [7].

En estos sistemas destaca la necesidad de comunicaciones, ya que no solo se compartirán datos entre tareas, sino que se hará también entre máquinas. Por tanto, las comunicaciones deben ser optimizadas y conllevan un importante esfuerzo por parte del programador. Es por eso que este tipo de entornos son los principales casos de uso de la tecnología de redistribución de estructuras de datos que se mostrará en este trabajo.

1.1.3 Tiling

El tiling [8] es una técnica utilizada en la programación de aplicaciones paralelas, consistente en dividir el cómputo en fragmentos más pequeños. El término proviene de la palabra inglesa *tile*, que significa baldosa, estableciendo una metáfora con cómo la unión de baldosas cubre una gran superficie.

Esta técnica surge como un requisito de la programación paralela. En la sección anterior se explica cómo se dividen el trabajo en subtareas, pero se ignora un detalle importante, con qué datos trabajará cada una de estas. Aparece así la necesidad de dividir los datos de forma que cada fracción pueda ser asignada a un proceso de la forma más independiente posible, es decir, minimizando las dependencias entre tareas para maximizar el paralelismo.

El tiling no es solo interesante por ser el paso previo a la división de tareas, sino que también presenta otras ventajas. Por ejemplo, al estar trabajando sobre zonas de memoria más reducidas, la localidad espacial y temporal puede verse beneficiada, mejorando los tiempos de ejecución al utilizar eficientemente la caché y evitando costosos accesos a memoria [9].

Hitmap [10] es una biblioteca que implementa funciones para crear, manipular y comunicar estructuras de datos de forma eficiente. Para ello, lleva a cabo una implementación del tiling algo diferente a la habitual. Generalmente, el tiling se suele lograr reescribiendo los bucles que constituyen el código (ya sea manual o automáticamente) para conseguir que se adapten mejor al comportamiento de la memoria. Por su parte, Hitmap propone el uso de la estructura de datos denominada *tile*, la cual puede ser repartida de forma automática entre varios procesos. Este reparto de carga hace que los bucles realicen un menor número de iteraciones, consiguiendo así el comportamiento de acceso a memoria deseado. Este proceso se lleva a cabo en tiempo de ejecución, frente a la reescritura de bucles, que requiere regenerar el código y debe ser soportada por parte del compilador.

El tiling facilita la implementación de los stencils, debido al volumen de las matrices que manejan. Al dividirse las estructuras de datos entre varios procesos, cada uno de ellos puede trabajar sobre una región distinta de la matriz, maximizando así el paralelismo.

1.2 Estado del arte

En esta sección se hará un breve recorrido por algunas soluciones ya existentes para el problema de la distribución de datos y cálculo automático de comunicaciones. Además, se evaluarán las principales diferencias con la solución mostrada en este trabajo.

Celerity [11] es el proyecto más similar a Hitmap en cuanto a objetivos perseguidos, ya que también busca automatizar el reparto y acceso a datos para eliminar esa responsabilidad del programador, consiguiendo códigos más mantenibles, legibles y optimizados. Sin embargo, existen algunas diferencias en su enfoque, ya que está más orientado a clústers de GPUs, usando MPI solo para intercambiar mensajes entre nodos, mientras que Hitmap puede ser utilizado tanto en CPU como GPU y está implementado únicamente en MPI. Además su API está escrita en C++ 14 en lugar de C99, haciendo que no sea utilizable en situaciones en las que no se dispone de un compilador de C++ moderno.

La principal diferencia entre Hitmap y Celerity es su nivel de abstracción. Hitmap solo contiene estructuras propias que representan conceptos fácilmente comprensibles por el programador, como

tiles o shapes y nunca se utiliza MPI de manera directa. Por su lado, Celerity no oculta su implementación en SYCL (un nivel de abstracción multiplataforma que permite escribir código para procesadores heterogéneos en C++ estándar[12]), de hecho, mantiene una estructura similar al código nativo de esta tecnología además de utilizar sus conceptos como colas y grupos de comandos. A pesar de estas diferencias, su aproximación al reparto de datos es muy similar a los shapes utilizados por Hitmap, aunque ofreciendo menor flexibilidad.

Existen también lenguajes de programación orientados a paralelismo de datos que ofrecen soporte para el tiling o reparto de datos como HPF (High Performance Fortran) [13]. Este ofrece constructores que permiten la distribución de datos, pero lo hace en tiempo de compilación, a diferencia de Hitmap que ofrece gran flexibilidad en tiempo de ejecución.

Revisando el estado del arte, se puede afirmar que no existe una solución que implemente estructuras de datos que se puedan distribuir de forma automática entre varios procesos, calculando automáticamente las comunicaciones necesarias. Además, las soluciones encontradas carecen del nivel de abstracción que se busca en este proyecto, donde se pretende ocultar los detalles subyacentes.

1.3 Motivación

A pesar de las funcionalidades que ofrece Hitmap para crear estructuras de datos, distribuirlas entre procesos y realizar intercambios de información, sus funciones de creación de comunicaciones tienen una interfaz compleja. El programador debe proporcionar manualmente diversos parámetros, como los procesos emisor y receptor o las regiones de las matrices involucradas.

Estas funciones son, además, propensas a fallos, ya que es fácil que el programador confunda algún índice en alguna de las llamadas, sobre todo al aumentar el número de dimensiones de las estructuras o complicar los patrones de comunicación. Este hecho motiva el desarrollo de una extensión para Hitmap, que sea capaz de determinar de manera automática las comunicaciones a realizar en un programa paralelo.

Como se ha visto en el análisis del estado del arte, existen tecnologías que han implementado comunicaciones automáticas similares a las que se buscan en este proyecto. Sin embargo, dado que ninguna de estas soluciones presenta todas las características que tiene Hitmap en otros campos como el reparto de estructuras de datos entre procesadores me veo motivado a desarrollar una extensión para Hitmap que permita automatizar la definición de los patrones de comunicación.

Se decide además utilizar stencils como ejemplo durante el proyecto, debido a la importancia que estos tienen en los diversos proyectos que están siendo desarrollados por el grupo de investigación Trasgo [14] utilizando la biblioteca Hitmap.

1.4 Objetivos

En esta sección se describen los objetivos del proyecto. Para su consecución, será necesario también alcanzar los objetivos secundarios que se mencionan.

1.4.1 Objetivos del proyecto

1. Desarrollar una extensión para la biblioteca Hitmap que permita crear patrones de comunicación de manera automática dado un tile y su reparto entre los distintos procesos.

2. Implementar un conjunto de programas de ejemplo de tipo stencil para probar y demostrar el funcionamiento de la extensión creada.

1.4.2 Objetivos secundarios

Durante el desarrollo del proyecto, deberán completarse los siguientes objetivos:

1. Estudiar la herramienta Hitmap.

2. Estudiar los stencils y su implementación paralela.

3. Diseñar distintas propuestas para la extensión de la biblioteca.

4. Implementar las propuestas para poder llevar a cabo la validación de los diseños.

5. Extraer resultados

6. Obtener conclusiones y validar la extensión.

1.5 Estructura del trabajo

Este documento se estructura de la siguiente forma:

- El capítulo 2 muestra la planificación del proyecto así como su presupuesto.
- El capítulo 3 presenta el estudio teórico realizado sobre la biblioteca Hitmap y los stencils.
- El capítulo 4 muestra una propuesta de solución para la creación automática de comunicaciones.
- El capítulo 5 describe diferentes opciones de implementación para la solución propuesta en el capítulo anterior.
- El capítulo 6 detalla los stencils que han sido desarrollados durante el proyecto, así como las optimizaciones realizadas sobre ellos.
- El capítulo 7 describe las pruebas de corrección y rendimiento llevadas a cabo y expone sus resultados.
- El capítulo 8 presenta las conclusiones del proyecto, así como las líneas de trabajo futuras.

Capítulo 2

Planificación

En este capítulo se tratarán los siguientes puntos:

- La metodología utilizada en el desarrollo del proyecto.
- Las tareas que componen el trabajo, así como su planificación.
- El camino crítico formado por estas tareas.
- El análisis de riesgos junto con los planes de contingencia y mitigación.
- El análisis de los costes estimados del proyecto.

2.1 Metodología

2.1.1 Definición

En el desarrollo del proyecto se usará el *Software Development Life Cycle* de entrega incremental [15]. Esta metodología se basa en dividir el sistema en pequeños componentes que se irán entregando en secuencia, aportando cada uno de ellos algún valor. La figura 2.1 muestra un esquema de funcionamiento de esta aproximación a la gestión del proyecto.

Las principales ventajas que ofrece esta metodología son:

- Permite obtener feedback desde las etapas tempranas del desarrollo.
- Al ser incrementos cortos es más difícil que aparezcan cambios en los requisitos.
- Se obtienen beneficios mucho antes, no hay que esperar al final del proyecto.
- Los proyectos más pequeños son más fáciles de gestionar.
- Ayuda a prevenir el "gold plating", es decir, la inclusión de características que no son realmente necesarias. Esto se debe a que el cliente es consciente de que puede añadir esas características en incrementos posteriores, por lo que no las exige desde el principio.

En cuanto a las desventajas, las principales son:

- Es posible que los últimos incrementos requieran hacer cambios sobre los primeros.

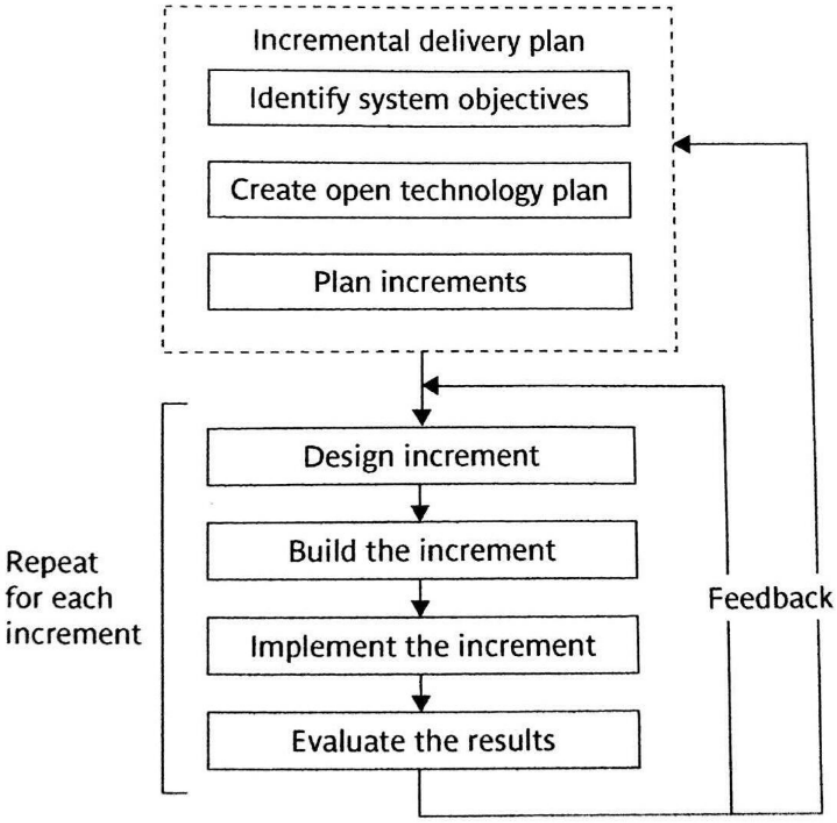


Figura 2.1: Funcionamiento de la metodología incremental (Extraída de [15])

- Puede que algunos desarrolladores sean más productivos trabajando solo en un proyecto de mayor tamaño en lugar de en pequeños sub-proyectos.

En esta metodología, se distinguen dos fases. Primero se identifican los objetivos globales del proyecto, es decir, se define qué se pretende conseguir pero sin necesidad de entrar en detalles. También pueden definirse métricas de calidad que se aplicarán a lo largo del proyecto.

La segunda fase es planear los incrementos. Cada uno de ellos debe cumplir una serie de características:

- Cada incremento debe contener entre un 1 y 5 % del proyecto y durar alrededor de un mes. Estas restricciones serán flexibilizadas en este proyecto debido a su corta extensión.
- Los incrementos deben proporcionar algún valor o beneficio.
- Algunos incrementos pueden depender de otros.

2.1.2 Justificación

Uno de los motivos por el que se elige esta metodología es que los incrementos se adaptan a las necesidades de un proyecto de investigación, en el que los requisitos no pueden definirse de manera precisa desde el inicio, sino que es probable que cambien en base a los resultados que se obtienen.

El enfoque iterativo es beneficioso para este proyecto ya que, debido a su naturaleza, es imposible establecer un diseño inicial fijo que sea implementado posteriormente, sino que será necesario

realizar varias propuestas de solución, implementarlas y probarlas para comprobar que rindan de la manera esperada.

Otra ventaja de este modelo es que permite obtener resultados desde la primera iteración. En este proyecto esto es importante ya que, al tener una versión preliminar, se pueden empezar a implementar los códigos que se usarán para las pruebas desde el inicio, de forma que todas las versiones puedan ser testadas para asegurar la corrección de los resultados, evitando así descubrir los fallos demasiado tarde lo que llevaría a repetir parte del trabajo.

Por tanto, dado que la metodología de entrega incremental presenta características como el enfoque iterativo, la obtención temprana de resultados y la no dependencia de requisitos iniciales bien definidos, se considera conveniente realizar este proyecto utilizando esta metodología.

2.2 Tareas del proyecto

En esta sección se explica cómo se ha dividido el proyecto en tareas y cómo se ha planificado el desarrollo de estas.

Este proyecto tiene asignada una carga de trabajo de 12 ECTS. Sabiendo que cada ECTS equivale a 25 horas, el trabajo tendrá una duración total de 300 horas. Se decide dedicar 20 horas semanales (es decir 4 horas diarias de lunes a viernes), por lo que la planificación se dividirá en 15 semanas.

La figura 2.2 muestra el diagrama de Gantt del proyecto. En el se observan dos ejes, el vertical, que indica el identificador de cada tarea, y el horizontal, que muestra una escala temporal para la duración de las tareas. También contiene flechas que representan las dependencias entre las actividades. Además, se muestran en color rojo aquellas tareas pertenecientes al camino crítico. La tabla 2.1 contiene el identificador y la descripción de cada una de las tareas que componen el diagrama de Gantt.

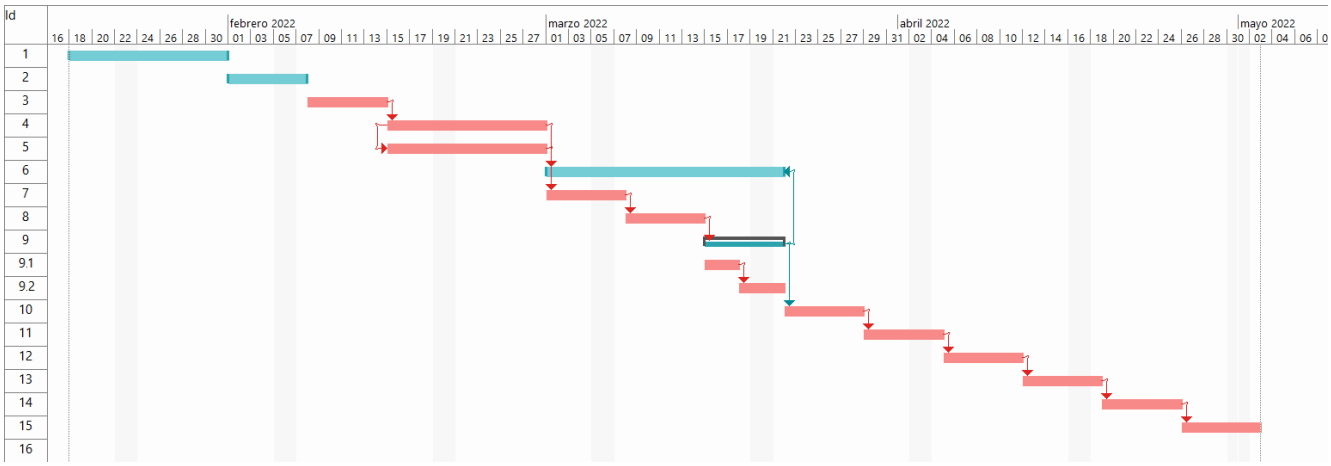


Figura 2.2: Diagrama de Gantt del proyecto

El diagrama de Gantt tiene una estructura prácticamente secuencial debido a la metodología iterativa utilizada. Cada semana (5 días laborables) se lleva a cabo una tarea que, en la mayoría de casos, supone el desarrollo de uno de los incrementos del proyecto. El siguiente no comenzará hasta que no haya terminado el anterior.

Entre las tareas se encuentran varios tipos. Las dos primeras son tareas de preparación, en las cuales se realiza el estudio previo necesario para adquirir conocimientos sobre las técnicas que se

ID	Descripción de la tarea
1	Familiarización con Hitmap
2	Familiarización con los stencils
3	Implementación del cálculo de comunicaciones utilizando Allgather
4	Desarrollo de stencils de ejemplo 2d
5	Desarrollo de stencils de ejemplo 3d
6	Ejecución de test de corrección
7	Implementación del cálculo de comunicaciones utilizando views
8	Implementación del cálculo de comunicaciones utilizando domains
9	Desarrollo de los tiles distribuidos
9.1	Diseño del tipo de datos *DistribTile*
9.2	Implementación de la función de bounding box
10	Implementación de stencils con tiling
11	Implementación de stencils con halos de cardinalidad variable
12	Implementación de los códigos de referencia para pruebas
13	Diseño de las pruebas finales
14	Ejecución de las pruebas finales
15	Análisis de resultados y extracción de conclusiones

Tabla 2.1: Tareas del diagrama de Gantt

utilizarán después en el proyecto. La tarea 3 constituye el primer incremento, que proporciona una primera versión funcional del cálculo automático de comunicaciones. Las tareas 4 y 5 consisten en desarrollar stencils de ejemplo en 2d y 3d respectivamente. Se llevan a cabo lo antes posible (después de acabar la primera versión del cálculo de comunicaciones) para así disponer durante el resto del proyecto de códigos con los que realizar pruebas de los diferentes incrementos implementados. Estas dos tareas se desarrollan en paralelo debido a la gran similitud que hay entre ellas, por lo que el trabajo en una puede beneficiar a la otra.

Una vez se dispone de los stencils, se puede iniciar la tarea 6, consistente en la ejecución de dichos stencils para comprobar que los resultados obtenido son correctos. Esta se prologa durante tres semanas debido a su carácter transversal, ya que, durante el desarrollo de los tres siguientes incrementos se realizan pruebas de forma regular para garantizar que los resultados son correctos en todo momento.

Las tareas 7, 8 y 9 suponen tres incrementos que mejoran el algoritmo de cálculo de comunicaciones hasta llegar a su versión final. La tarea 9, consistente en la creación de los "tiles distribuidos" se subdivide en dos, ya que es necesario tanto diseñar la nueva estructura de datos que se añade en Hitmap como realizar cambios en el algoritmo implementado en tareas anteriores.

Tras haber perfeccionado la creación de patrones de comunicación, se desarrollarán las tareas 10 y 11, que implementarán técnicas más avanzadas de optimización en los stencils, demostrando así las capacidades de la extensión creada para Hitmap.

Por último, se realizarán tareas consistentes en el diseño de las pruebas finales, la realización de los códigos de referencia que se utilizarán en estas, su ejecución y la extracción de conclusiones para cerrar el proyecto.

2.3 Camino crítico

El camino crítico de un proyecto es la ruta más larga de actividades que deben completarse a tiempo para completar el proyecto. Es decir, está formado por aquellas tareas que no pueden durar más de lo planificado ya que si lo hicieran retrasarían el final del proyecto.

En este proyecto los incrementos se realizan de forma secuencial, iniciando cada uno cuando ha terminado el anterior (con la excepción de las actividades 4 y 5 como se explica en la sección anterior). Además, existen dependencias entre la mayoría de ellos: no se puede desarrollar una versión del cálculo de comunicaciones hasta que no se dispone de la anterior, los stencils no se pueden implementar hasta que se disponga de una versión preliminar de la extensión desarrollada, etc. Por ello, casi todas las actividades están en el camino crítico. Las únicas tareas que no lo están son las de estudio previo y familiarización con la tecnología, ya que no tienen restricciones. Esto significa que, en caso de que se retrasaran, podrían continuar desarrollándose cuando ya hayan iniciado las siguientes actividades.

Un retraso en la actividad 6 también afectaría a la fecha de finalización del proyecto, sin embargo no está en el camino crítico porque no constituye el camino más largo, como sí lo hacen la 7, 8 y 9.

2.4 Análisis de riesgos

Un riesgo es "un evento o condición que, si ocurre, tiene un efecto positivo o negativo en los objetivos del proyecto" [16]. En esta sección se expondrán los principales riesgos que se han encontrado para la realización del proyecto.

Para cuantificar la importancia de los riesgos se usará la matriz probabilidad/impacto [17]. Esta tiene en cuenta los dos factores más importantes en el análisis de riesgos, la probabilidad y el impacto, y los promedia para conocer su magnitud. Esto implica que para que un riesgo tenga una importancia alta, tanto su impacto como probabilidad deben ser considerables, mientras que riesgos con un impacto medio pierden importancia si su probabilidad no es alta. La matriz que se utilizará para priorizar los riesgos en este proyecto se muestra en la tabla 2.2. La valoración de la criticidad y el impacto de los riesgos ha sido consensuada con los dos tutores del proyecto.

PROB/IMP	BAJO	MEDIO	ALTO
BAJA	BAJO	BAJO	MEDIO
MEDIA	BAJO	MEDIO	ALTO
ALTA	MEDIO	ALTO	ALTO

Tabla 2.2: Matriz probabilidad/impacto

Como se ha explicado en la sección anterior (2.3), las tareas del camino crítico son aquellas cuyo retraso afecta a la fecha de finalización del proyecto. Debido a la importancia de estas tareas, es necesario centrar el análisis de los riesgos en ella, por lo que, para cada riesgo, se indicará a qué tareas del camino crítico afecta.

Para cada uno de los riesgos se describe su plan de mitigación y de contingencia. Las acciones de mitigación son aquellas destinadas a disminuir el impacto o la probabilidad de un riesgo, es decir, aquellas que se llevan a cabo antes de que este ocurra. Por otro lado, las acciones de contingencia

son aquellas que se realizarán solo si el riesgo se materializa y por tanto es necesario afrontar sus consecuencias.

Parte de los riesgos seleccionados provienen de la lista "Top 10 riesgos en del desarrollo de software" de Boehm [8]. Se han escogido los más relevantes para el proyecto y se han añadido otros específicos para este desarrollo.

Riesgo 1 - Planificación poco realista	
Descripción	Debido a la falta de experiencia en la estimación de tiempo, es posible que una tarea no se complete dentro del plazo, lo que podría retrasar el camino crítico.
Probabilidad	Media
Impacto	Medio
Tareas a las que afecta	Todas
Mitigación	Se añade un margen en las tareas para poder compensar posibles estimaciones demasiado optimistas
Contingencia	Aumentar el tiempo y recursos empleados en el proyecto para cumplir los plazos

Tabla 2.3: Riesgo 1

Riesgo 2 - "Gold plating"	
Descripción	Añadir más funcionalidades de las necesarias (gold plating) puede provocar que se dedique menos recursos a las tareas realmente importantes, haciendo que no se cumplan los objetivos o los plazos del proyecto
Probabilidad	Baja
Impacto	Medio
Tareas a las que afecta	Todas
Mitigación	Utilizar una metodología de entrega incremental que ayuda a disminuir el gold plating
Contingencia	Alargar el proyecto para poder hacer frente al tiempo malgastado y desarrollar las características realmente importantes

Tabla 2.4: Riesgo 2

Riesgo 3 - Cambios tardíos en la especificación del proyecto	
Descripción	Debido a una mala especificación inicial del proyecto, puede que surja un cambio durante el desarrollo, alterando la planificación de tiempo y costes del resto del proyecto
Probabilidad	Baja
Impacto	Medio
Tareas a las que afecta	3, 4, 5, 7, 8, 9, 10, 11, 12
Mitigación	Usar metodología incremental, en la cual un cambio en la especificación solo afecta al trabajo realizado en un incremento
Contingencia	Eliminar características del proyecto para poder incluir los cambios necesarios

Tabla 2.5: Riesgo 3

Riesgo 4 - Problemas de rendimiento del software	
Descripción	Debido a un mal diseño o implementación, es posible que los tiempos de ejecución sean demasiado altos, haciendo que el software resultante no sea utilizable
Probabilidad	Baja
Impacto	Alto
Tareas a las que afecta	3, 7, 8, 14
Mitigación	Realizar pruebas de rendimiento en todos los incrementos para asegurar que se cumple el objetivo de rendimiento
Contingencia	Retrasar el fin del proyecto para poder revisar el incremento que causó el problema, ya que no se puede seguir construyendo sobre él

Tabla 2.6: Riesgo 4

Riesgo 5 - Problemas en las máquinas de pruebas	
Descripción	Un error de software o hardware en alguna de las máquinas del clúster puede provocar fallos en su funcionamiento, lo que retrasaría el proyecto al no permitir realizar pruebas en ellas
Probabilidad	Baja
Impacto	Alto
Tareas a las que afecta	14
Mitigación	No dejar las pruebas para el final del proyecto ni de los incrementos, de forma que se disponga de más margen para su realización
Contingencia	Recurrir a otra máquina en la que ejecutar las pruebas, como un ordenador portátil

Tabla 2.7: Riesgo 5

Riesgo 6 - Pérdida del trabajo realizado	
Descripción	Debido a un fallo en el ordenador utilizado durante el desarrollo, pueden perderse los archivos del proyecto, causando la pérdida del progreso en las tareas realizadas
Probabilidad	Baja
Impacto	Alto
Tareas a las que afecta	Todas
Mitigación	Utilizar herramientas de control de versiones como git, soluciones en la nube para la escritura de la memoria como Overleaf y hacer copias de seguridad periódicas
Contingencia	Ajustar el alcance del proyecto para compensar el trabajo extra que se va a hacer para volver al estado previo a la pérdida

Tabla 2.8: Riesgo 6

Riesgo 7 - Falta de familiarización con la tecnología	
Descripción	Debido a la falta de experiencia con las tecnologías utilizadas en el proyecto, es posible que surjan dudas o problemas, que provoquen que se retrasen las tareas
Probabilidad	Media
Impacto	Medio
Tareas a las que afecta	3, 4, 5, 7, 8, 9, 10, 11, 12
Mitigación	Crear tareas de estudio de las tecnologías, reservando así tiempo del proyecto a ganar la experiencia necesaria
Contingencia	Añadir a otra persona al proyecto capaz de resolver dudas o solucionar problemas de las tecnologías utilizadas

Tabla 2.9: Riesgo 7

Riesgo 8 - Errores en software externo	
Descripción	Debido a un fallo en el software externo al proyecto (como la biblioteca Hitmap), pueden aparecer errores en las características implicadas bloqueando las tareas
Probabilidad	Baja
Impacto	Alto
Tareas a las que afecta	3, 4, 5, 7, 8, 9, 10, 11, 12
Mitigación	Dedicar tiempo a verificar que las partes de la biblioteca que se van a utilizar funcionan correctamente antes de empezar con su expansión
Contingencia	Recurrir a ayuda externa que solucione los problemas en el software ajeno al proyecto

Tabla 2.10: Riesgo 8

2.5 Análisis de costes

En esta sección se detallarán los costes de desarrollo del proyecto. Estos se pueden desglosar en tres categorías: coste de personal, coste del hardware y licencias de software.

Los costes del personal se dividen en dos tipos:

- Sueldo del desarrollador: el salario medio de un programador junior en España es de 18.886€ al año [18]. Suponiendo una jornada de 8 horas diarias y unos 250 días laborales por año, el sueldo por hora resultante es de 9.5€.
- Sueldo de los tutores: suponiendo un salario anual de 27.000€ y el mismo número de horas que en el caso anterior, el coste por hora resultante es de 13.5€.

El desarrollador trabaja durante todas las horas del proyecto (300), mientras que los dos tutores realizan reuniones semanales de 1 hora, haciendo un total de 15 horas cada uno. La tabla 2.11 muestra los costes totales de personal.

El coste de hardware está compuesto por dos elementos:

- Ordenador personal: ordenador portátil utilizado durante el desarrollo del proyecto. Se trata de un Asus TUF con 16GB de RAM, un AMD Ryzen 7 4800H y una tarjeta gráfica Nvidia

Persona	Salario por hora (€)	Total horas	Total(€)
Alumno	9,5	300	2850
Tutor 1	13,5	15	202,5
Tutor 2	13,5	15	202,5
Total			3255

Tabla 2.11: Coste de personal del proyecto

GeForce GTX 1650 Ti. Su coste es de unos 900€ y su vida útil estimada de unos 5 años. Por tanto, su coste por hora es de 0.02€.

- Clúster del grupo Trasgo: para realizar algunas pruebas de rendimiento se utilizan las máquinas del clúster del grupo de investigación Trasgo. Sin embargo, es muy difícil calcular cuál ha sido la amortización de dichas máquinas respecto a este proyecto, ya que solo se han desarrollado pruebas puntuales en ellas, el resto del trabajo se ha llevado a cabo en local. Además, es complicado saber cuál es la vida media de los equipos, ya que estos son actualizados cada año. Por tanto, se omitirán estas máquinas del análisis de costes.

La tabla 2.12 muestra el coste total del hardware, teniendo en cuenta que el ordenador personal se utiliza durante las 300 horas que dura el proyecto.

Equipo	Coste por hora (€)	Total horas	Total (€)
Portátil	0.02	300	6
Total			6

Tabla 2.12: Coste de hardware del proyecto

El coste del software de este proyecto es de 0€ ya que para el desarrollo solo se utiliza como editor de texto Visual Studio Code, que es gratuito, y como compilador gcc, software de GNU, por lo que es de uso libre.

La tabla 2.13 muestra la suma de todos los costes del proyecto.

Coste	Importe (€)
Personal	3255
Hardware	6
Software	0
Total	3261

Tabla 2.13: Coste de total del proyecto

Capítulo 3

Estudio previo y familiarización con Hitmap

En este capítulo se tratarán los siguientes puntos:

- La definición de los stencils, junto con una descripción de su implementación tanto secuencial como paralela.
- La biblioteca Hitmap, describiendo su funcionamiento y definiendo sus conceptos más importantes, que serán utilizados en el resto del documento.

3.1 Caso de estudio: stencils

3.1.1 Definición

Un stencil es un tipo de algoritmo iterativo que trabaja sobre matrices de n dimensiones. En cada iteración, se obtiene una nueva matriz a partir de la de la iteración anterior. Este proceso se lleva a cabo hasta que se llegue a un límite de iteraciones preestablecido o los valores hayan convergido (es decir, el mayor cambio realizado esté por debajo de un umbral de convergencia) [19].

Para cada stencil, se define un "patrón", una serie de posiciones relativas al elemento que está siendo calculado, en las cuales se encuentran los datos que servirán de entrada a una operación matemática (generalmente una media aritmética o ponderada).

Este tipo de problemas son muy utilizados en tareas como simulaciones de naturaleza científica, procesado de imágenes o resolución de ecuaciones diferenciales parciales [20].

Existen diversos tipos de stencils, que se pueden clasificar en base a distintos criterios:

- Dimensiones: pueden operar sobre arrays de una dimensión, matrices bidimensionales, o incluso estructuras en 3 o más dimensiones.
- Número de elementos accedidos: este número no está limitado por el número de dimensiones (en una matriz 2D un elemento tiene 8 elementos adyacentes contando las diagonales mientras que en una 3D tiene 26). Esto se debe a que los elementos leídos no tienen que ser estrictamente adyacentes, sino que se puede avanzar dos o más posiciones en una dirección para llegar al dato leído.

- Compacto o no compacto: los stencils no compactos son aquellos en los que elementos no adyacentes intervienen en el cómputo.
- Simetría: los stencils no son necesariamente simétricos, lo que puede generar patrones especiales en los que solo se acceda a datos en una dirección.

En este trabajo se utilizará la nomenclatura XdY, done X es el número de dimensiones e Y el número de elementos accedidos. Por ejemplo, un stencil *2d4*, es un stencil bidimensional en el que se accede a cuatro vecinos.

3.1.2 Implementación secuencial

El código 3.1 muestra la implementación secuencial de un stencil 2d4, en el cual cada elemento se calcula como la media de sus vecinos superior, inferior, izquierdo y derecho.

Este problema se puede resolver de forma simple con código secuencial. Basta con plasmar la fórmula matemática del stencil ($x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4$) y operar con los índices de los arrays para acceder al elemento adecuado. La única complicación destacable es tener en cuenta los extremos, por ejemplo, un elemento de la primera columna no podrá acceder a su vecino de la izquierda al no existir.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
 double mat[ROWS][COLS];
 double copy[ROWS][COLS];

 int i,j;

 // Inicializacion
 for ( i=0; i<ROWS; i++ )
  for ( j=0; j<COLS; j++ )
   mat[i][j] = 0.0;

 //Valores iniciales para que el resultado no sea 0
 for ( i=0; i<ROWS; i++ ) {
  mat[i][0] = 3;
  mat[i][COLS-1] = 4;
 }
 for ( j=0; j<COLS; j++ ) {
  mat[0][j] = 1;
  mat[ROWS-1][j] = 2;
 }

 int iteracion;
 for ( iteracion=0; iteracion<ITERACIONES; iteracion++ ) {
```

```
27   // Guardar valores anteriores en una copia
28   for ( i=0; i<ROWS; i++ )
29     for ( j=0; j<COLS; j++ )
30       copy[i][j] = mat[i][j];
31
32   for ( i=1; i<ROWS-1; i++ )
33     for ( j=1; j<COLS-1; j++ )
34       mat[i][j] = ( copy[i-1][j] + copy[i+1][j] + copy[i][j-1] + copy[i
             ][j+1] ) / 4;
35   }
36
37   return 0;
38 }
```

Código 3.1: Implementación secuencial de un stencil 2d4 en C

3.1.3 Implementación paralela

En la versión paralelizada, se mantienen los problemas antes mencionados (como el manejo de índices de los arrays), pero aparecen otros nuevos, como es el caso de la necesidad de comunicaciones.

Al ser un código paralelizado, la estructura de datos (matriz) sobre la que se trabaja ha sido distribuida entre varios procesos. Esto se traduce en que en algunas posiciones no se puede acceder a todos los vecinos, pero sin embargo estos valores han de ser calculados. Por tanto, es necesaria la existencia de comunicaciones: cada proceso dará a conocer a sus vecinos los valores de sus bordes de forma que éstos puedan realizar sus cálculos. Además, este proceso también recibirá datos de los procesos adyacentes, ya que las comunicaciones suelen ser bidireccionales salvo en casos específicos.

En la figura 3.1 se muestran las comunicaciones llevadas a cabo por un proceso para realizar los cálculos, tanto los envíos (línea punteada) como las recepciones de datos (línea continua)

Halos

En el ejemplo de la figura 3.1 se muestra una zona sombreada en cada una de las matrices. Estas zonas, llamadas halos, son el lugar en el que se almacenan los datos recibidos de los procesos vecinos.

Los halos se suelen colocar de tal forma que los valores recibidos mantengan su posición con respecto a los datos locales. Esto es, si un proceso recibe la última columna de su vecino izquierdo, el halo se colocará a la izquierda de la primera columna de la matriz local.

Al duplicar los datos frontera y mantener su lugar en la matriz global, el algoritmo podrá leerlos como si se trataran de propios, por lo que no es necesario distinguir su procedencia durante la parte de cómputo secuencial del programa.

En la figura 3.2 se muestra un ejemplo de halos en el cual un proceso (el central) almacena una copia de la fila frontera de sus cuatro vecinos. Los valores copiados que constituyen el halo se muestran en colores.

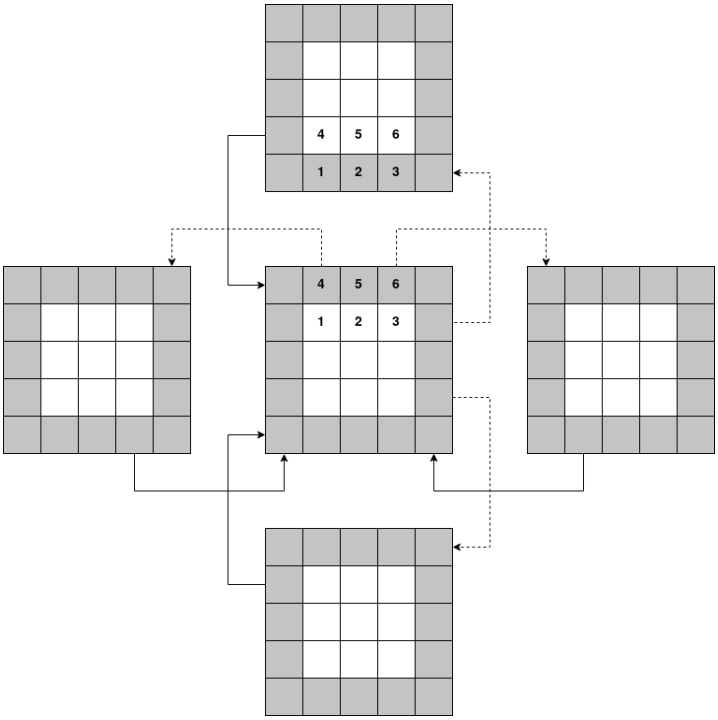


Figura 3.1: Esquema de intercambio de datos en un stencil paralelo

En este ejemplo es discutible si las esquinas de la matriz forman parte del halo. En un código real esto dependerá de si el cómputo utiliza los datos de las diagonales, aunque en ese caso ese dato se leería de los procesos que se encuentran en la diagonal del proceso central, y de la forma en la que se haya creado el halo (si solo se soportan matrices cuadradas sin "salientes" es imposible no incluir esas esquinas aunque no vayan a ser leídas nunca).

3.2 Libería Hitmap

3.2.1 Introducción

Hitmap es una biblioteca para C99 que pretende facilitar la conversión de algoritmos secuenciales a programas paralelos ocultando y automatizando algunos de los detalles más complejos relativos a la implementación, ofreciendo una abstracción sobre MPI que permite al desarrollador centrarse en únicamente en el algoritmo que está implementando.

El concepto principal en el que se basa Hitmap es la partición automática de datos, es decir, la biblioteca crea una topología virtual, la asocia a los procesadores disponibles y determina qué fracción de los datos corresponde a cada proceso [10].

Hitmap ha sido diseñado ofreciendo posibilidades de expansión mediante un sistema de plug-ins. Esto permite separar el código de la biblioteca de la implementación de estructuras de datos complejas o de sofisticados mecanismos de distribución de datos. Esta polivalencia permite a los desarrolladores implementar una gran cantidad de algoritmos con Hitmap, no solo aquellos que utilicen estructuras de datos más tradicionales (como matrices de dos o tres dimensiones).

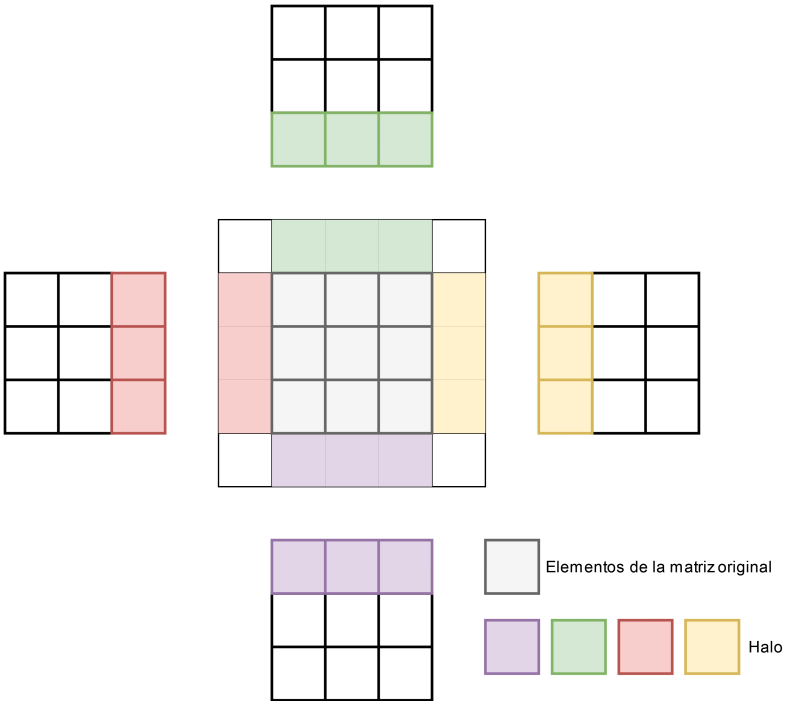


Figura 3.2: Halos de un proceso con copias de datos de sus cuatro vecinos

3.2.2 Definiciones y términos

Para comprender el funcionamiento de Hitmap es necesario familiarizarse con algunos conceptos que se expondrán en esta sección:

- Signature: un signature es una tupla formada por tres elementos: inicio, fin y stride (salto). Representa un subconjunto de los índices de un array de una dimensión. Esta basado en la sintaxis de arrays en Fortran 90 [21].

  Un signature se puede expresar de la siguiente forma:

  $$S \in Signature = (begin : end : stride)$$

  Esto significa que el signature contiene todos aquellos índices comprendidos entre *begin* y *end* (incluidos), avanzando en saltos de *stride* elementos desde *begin*.

  El cardinal de un signautre es, como en cualquier conjunto, su número de elementos diferentes.

- Shape: un shape es una n-tupla de signatures, lo cual permite representar índices en arrays de más de una dimensión. Una vez más, el cardinal es el número de índices diferentes del shape.

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

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

- Tiles: los tiles son la forma en la que Hitmap representa los arrays n-dimensionales. Su dominio está definido por un shape y contienen elementos de un determinado tipo.

3.2.3 Tiling jerárquico

Una de las características más potentes de Hitmap es la capacidad de crear tiles jerárquicos. Esto significa que, dado un tile, puede obtenerse un tile subconjunto del mismo. Este proceso se conoce como selección, y tiene como entrada un shape, que señala los elementos del tile original que serán seleccionados.

Un ejemplo de selección en tiles puede verse en la figura 3.3. En ella, se crea un tile A, cuyos índices van de 0 a 4 en la primera dimensión y de 1 a 7 en la segunda (Hitmap soporta cualquier rango de índices, no tienen que empezar en 0 e incluso pueden ser negativos). De estos elementos, se seleccionan aquellos con coordenadas entre 1 y 3 en ambas dimensiones, dando lugar al tile B. La figura también muestra las coordenadas locales del tile B, ya que se pueden utilizar estas o las del tile global (por ejemplo, el elemento (0,0) corresponde al elemento (1,1) en el tile padre). El tile D muestra el uso del stride en un shape, seleccionando elementos alternos, que serán vistos como contiguos utilizando coordenadas locales a D.

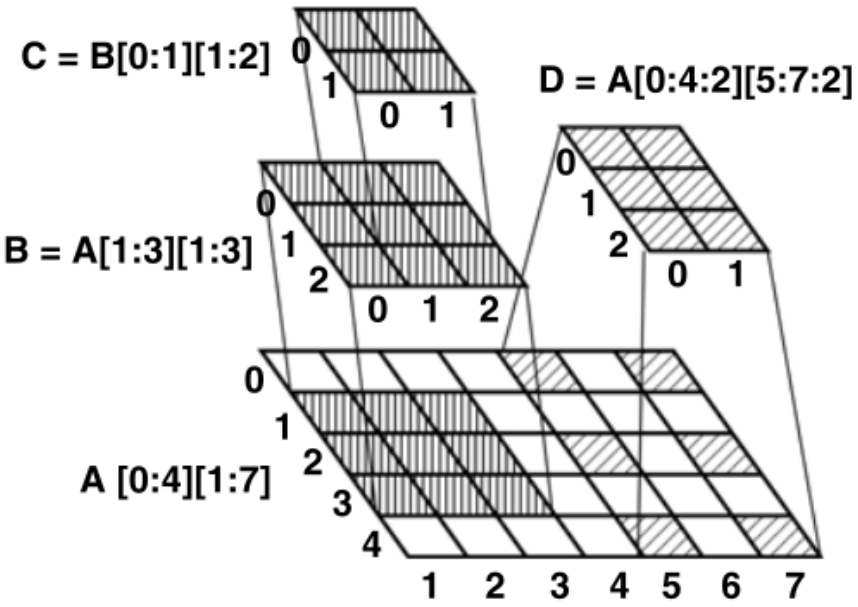


Figura 3.3: Obtención de tiles a partir del original (tomada de [10])

Una peculiaridad de los tiles es que no tienen una correspondencia directa con los arrays tradicionales ya que la reserva de memoria no es automática. Esto quiere decir que, volviendo al ejemplo de la figura 3.3, el tile A puede no tener memoria asignada. En su lugar, puede ser el tile B quien reserve este espacio en memoria cuando sea necesario (a decisión del programador). Además, el tile C podría acceder a los elementos del tile B de forma transparente, como si tuviera su propia memoria pero usando en realidad la de B.

También existe la posibilidad de que C también reserve su propia memoria "duplicando" una parte del tile. Hitmap permite actualizar los valores de C con los del padre, es decir, copiar los valores de B a su correspondiente posición en C.

La existencia de tiles sin memoria reservada permite que cada proceso reserve solo el espacio que necesite para su subtile, permitiendo así trabajar con tiles de gran tamaño sin agotar la memoria del sistema.

Existen también funciones que permiten modificar el shape de un tile para, por ejemplo, extenderlo o desplazarlo. Esto es especialmente útil para implementar los ya mencionados halos.

3.2.4 Layouts y topologías

Como se ha mencionado en la introducción, Hitmap realiza el reparto de datos de manera automática, alejando así las dificultades que esto implica del programador. Esto se consigue gracias a los layouts y las topologías.

Una topología es una organización virtual de los procesadores físicos. Es decir, permiten que el programador vea, por ejemplo, un grid cuadrado de procesadores aunque estos en realidad se encuentren en diferentes máquinas de diferentes características. Es posible que incluso un procesador potente se divida en varios en la topología virtual.

A pesar de la conveniencia de las topologías, que abstraen los detalles de la implementación física de las máquinas, estas son inútiles por sí mismas por lo que deben ser usadas conjuntamente con los layouts, funciones que permiten distribuir los elementos de un tile entre los distintos procesos reflejando la topología virtual antes definida. Para ello, asigna a cada uno un shape (subconjunto de los índices del tile original).

La forma en la que se hace este reparto, tanto de la topología como el layout, es altamente configurable por el programador. En el código es suficiente con especificar el nombre de la topología y del layout que se usará, y el sistema de plug-ins hará el resto. Esto permite probar de manera sencilla diferentes combinaciones de topologías y layouts, para encontrar aquella que mejor se adapte al problema, mejorando así los tiempos de ejecución.

Para comprender mejor estos conceptos, se presenta a continuación un ejemplo. En este se utilizarán dos topologías, 1D y 2D. Estas crearán dos distribuciones virtuales en base a los procesadores físicos, un array lineal y una matriz bidimensional respectivamente. En cuanto al layout, se usará el *Blocks*, el cual da lugar a particiones de datos formadas por bandas casi iguales (con una fila de diferencia como máximo).

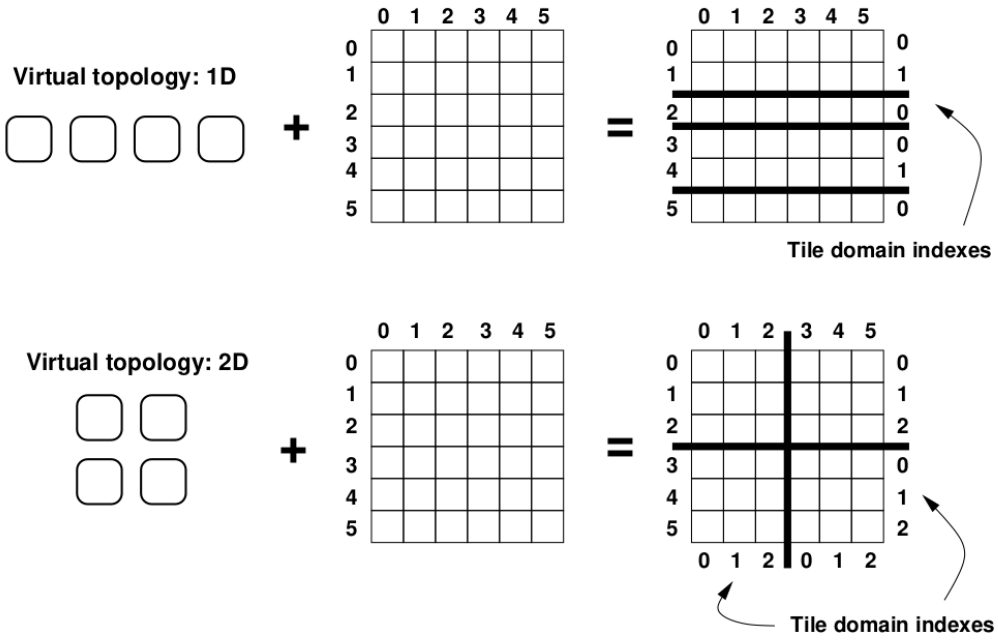


Figura 3.4: Ejemplo de uso conjunto de topologías y layouts (tomada de [10])

Este ejemplo se muestra en la figura 3.4. Esta compuesta por dos casos. En el mostrado en la parte superior se utiliza la topología lineal, por lo que los cuatro procesadores físicos se disponen

en fila. Al ser una topología de una dimensión, el layout repartirá también los datos en una única dimensión, es decir, no partirá las filas. La matriz tiene 6 filas, por lo que no puede repartirlas de forma exacta entre cuatro procesadores. Así que, como se ha explicado, el layout *Blocks* creará dos divisiones de dos filas y dos de una, sumando las 6 filas totales y de forma que la diferencia máxima entre dos divisiones es de una sola fila.

En el ejemplo inferior se emplea una topología 2D, dando lugar a una matriz de 2x2 procesadores. En este caso, la topología también actuará en dos dimensiones, dando lugar a cuadrados en lugar de filas. A diferencia del ejemplo anterior, este sí permite realizar una división exacta de la matriz, por lo que todas las particiones tienen el mismo número de elementos.

3.2.5 Comunicaciones

Hitmap ofrece una gran variedad de funciones que permiten intercambiar datos entre diferentes procesos. La implementación de estas comunicaciones se simplifica gracias a la existencia de shapes (que permiten seleccionar aquellas partes del tile que se enviarán o marcar la zona de recepción) y a las topologías, que permiten determinar de manera sencilla los vecinos de un proceso en cualquier dirección.

Existe un tipo de datos llamado *Comm* que almacena toda la información necesaria para llevar a cabo la comunicación. Como es frecuente que una misma comunicación se repita a lo largo de varias iteraciones, Hitmap cuenta con los *Pattern*, que definen un patrón de comunicación recurrente para facilitar su reutilización.

Estas funciones tienen la ventaja de ocultar al programador las tareas de gestión de comunicaciones que, mal implementadas, pueden llevar a deadlocks o condiciones de carrera, de forma que solo tenga que centrarse en definir el flujo que deben seguir los datos. Sin embargo, como se muestra en el código 3.2, la función que añade un par envio-recepción a un patrón necesita un gran número de parámetros, como los identificadores de los dos procesos involucrados en el intercambio, los tiles de envío y recepción o los shapes que seleccionan los datos concretos de ambos tiles que intervendrán en la comunicación. Por este motivo, es necesario aumentar la automatización y evitar este código propenso a errores.

```
1 hit_patternAdd( &patron,
2     hit_comSendRecvSelectTag(lay, sendRank, tileSend, shapeSend,
        modeSelectSend, recvRank, tileRecv, shapeRecv, modeSelectRecv,
        double, tag));
```

Código 3.2: Creación de una nueva comunicación

Capítulo 4

Propuesta de solución

En este capítulo se tratarán los siguientes puntos:

- El análisis de la propuesta de solución.
- El algoritmo utilizado para calcular las comunicaciones de forma automática.
- El diseño de la propuesta de solución.

4.1 Análisis de la propuesta de solución

Como se ha mencionado en el capítulo de introducción (sección 1.5), Hitmap ofrece diversas funcionalidades para la implementación de programas paralelos, pero carece de mecanismos que permitan automatizar los intercambios de datos entre procesos. Por ello se decide añadir los tiles distribuidos como una extensión.

La figura 4.1 muestra el diagrama de clases de Hitmap al comienzo del proyecto. Este representa los conceptos vistos en la sección 3.2. La clase de más importancia es HitTile, que representa una estructura de datos abstracta que será especializada dependiendo del tipo de datos que albergue (HitTile_int, HitTile_dobule...). Los HitTiles almacenan un Shape que contiene todos sus posibles índices. Estos shapes están compuestos por Signatures, conjuntos de índices en una de las dimensiones.

El diagrama también muestra la clase Layout (con dos de sus implementaciones concretas, SignatureLayout y ListLayout), encargada de calcular shapes a partir de la información leída de la clase Topology.

Por último, la clase Comm representa una comunicación realizada entre procesos intercambiando HitTiles. La agregación de comunicaciones recibe el nombre de Pattern.

En este proyecto se añadirá una nueva clase a la arquitectura de Hitmap, los tiles distribuidos (HitDistribTile). Estos serán una especialización de los HitTiles genéricos de Hitmap. Se diferenciarán de estos en que en su contenido incluirán la información necesaria para calcular las comunicaciones de forma automática. La figura 4.2 muestra la modificación que se realizará sobre el diagrama original. Las clases señaladas en azul son aquellas que han sido añadidas en este proyecto.

El uso de la herencia en HitDistribTile permite que los tiles distribuidos puedan ser utilizados en todas las demás funciones de Hitmap. Por tanto, sigue manteniendo funcionalidades como el acceso y modificación de datos o la reserva de memoria.

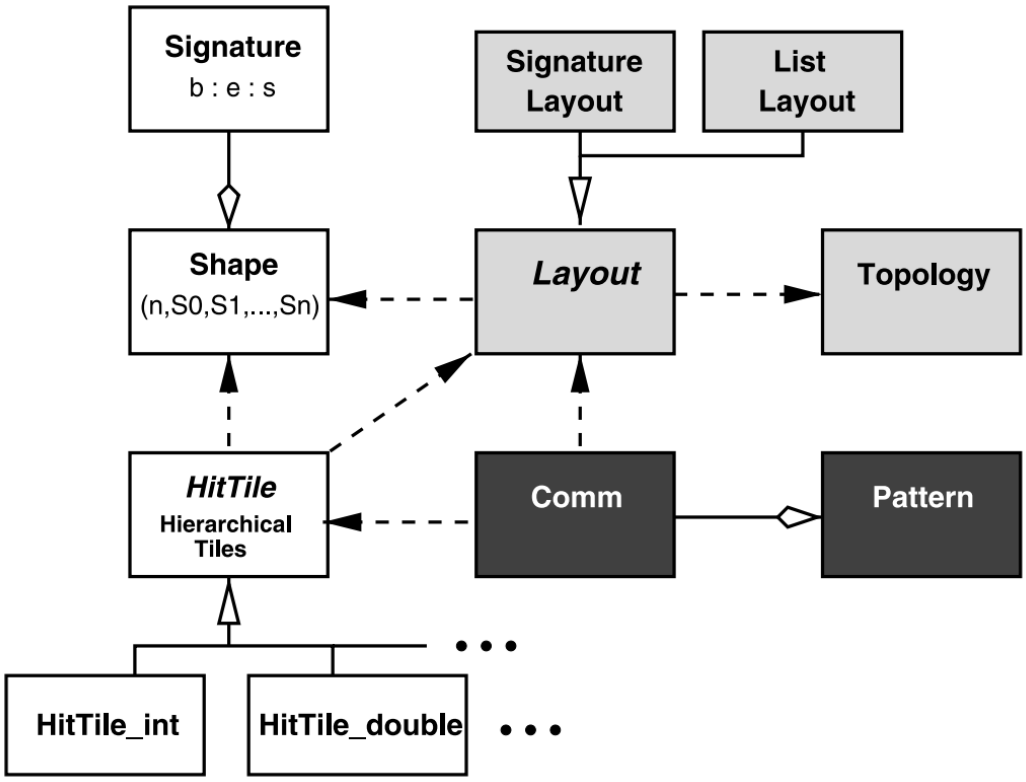


Figura 4.1: Diagrama de clases de Hitmap (extraído de [10])

También se añadirá una nueva función a la clase Pattern, llamada redistribute, la cual tomará como parámetros dos tiles distribuidos, uno de envío y otro de recepción, y retornará el patrón de comunicación generado automáticamente para transferir datos de un tile a otro. La figura 4.3 muestra esta nueva función. Se omiten el resto de contenidos de la clase para simplificar el diagrama.

4.2 Solución teórica

En esta sección se explicará cómo funciona la función redistribute, lo cual definirá también de qué datos se debe disponer en los tiles distribuidos, lo cual será importante en su diseño.

Para poder calcular las comunicaciones entre dos procesos se utilizará un algoritmo basado en intersecciones de shapes. El primer paso es disponer los tiles de todos los vecinos de forma ordenada, es decir, de forma que entre todos reconstruyan la matriz original. Al hacer esto, los halos de cada proceso se superpondrán a partes del tile de sus vecinos. El cálculo de esta intersección entre dos procesos determinará la región del tile que debe enviar el vecino para que se reciban los datos necesarios.

La figura 4.4 muestra una representación gráfica de este procedimiento. En ella, se observan varios tiles (numerados del uno al nueve) que representan los datos correspondientes a cada uno de los procesos. En rojo está representado el halo del tile 5 (que será el sujeto de este ejemplo). Como se puede comprobar, este tile tiene intersección con los datos de los procesos 2, 4, 6 y 8, lo que se representa con los rectángulos azules. Esto quiere decir, que los ya mencionados vecinos deberán enviar estas regiones azules al proceso 5, que las almacenará en su halo.

El halo también tiene intersección con los otros cuatro vecinos, pero este ejemplo, por simpli-

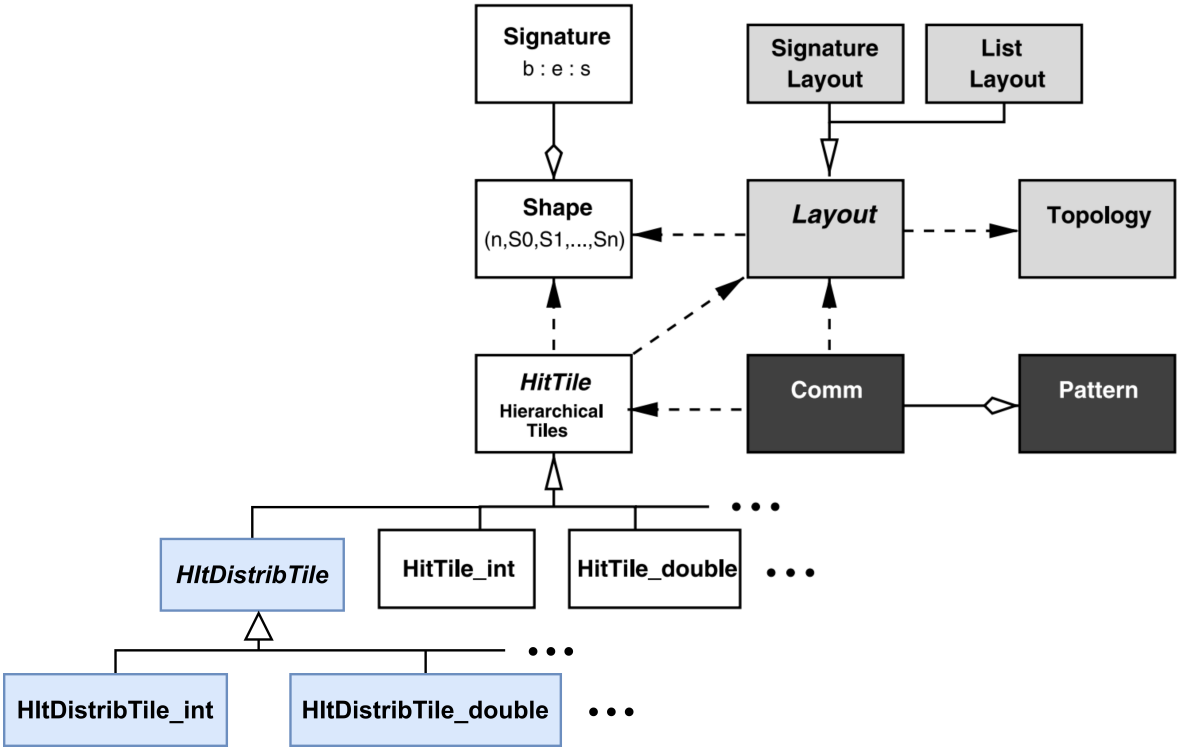


Figura 4.2: Diagrama de clases de Hitmap incluyendo los tiles distribuidos

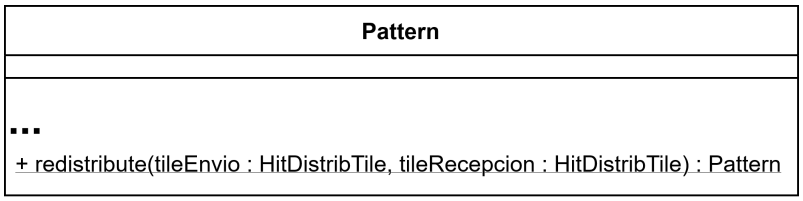


Figura 4.3: Función añadida a la clase Pattern

cidad, ignora las comunicaciones con las diagonales. Dichas comunicaciones serán eliminadas en las versiones más optimizadas en aquellos casos en los que no sean necesarias.

El algoritmo 1 proporciona una descripción del procedimiento de cálculo de intersecciones. Además hace hincapié en las dos fases que componen el calculo, ya que cada proceso tendrá que determinar no solo qué datos debe enviar a sus vecinos, sino también qué debe recibir de estos.

Para la primera fase, se recorren todos los vecinos obteniendo sus tiles de envío y recepción. Después se calcula la intersección entre los datos que se tienen en local (tile de envío local) y los datos que necesita el vecino (su tile de recepción con halos). Para optimizar, se comprueba que la intersección no es vacía y que los datos que se van a enviar (es decir, la recién creada intersección) no estén ya en el tile destino (líneas 6 y 8). Si se cumple la condición, se añade al patrón un envío al vecino, de los datos de la intersección, procedentes del tile de envío (línea 9)

La recepción sigue una lógica similar: se hace la intersección entre los datos que se necesitan y los que tiene el otro proceso (tile de recepción local con halo y de envío del vecino respectivamente) y se comprueba que no se disponga ya de esos datos. Tras eso se añade al patrón una recepción desde ese vecino, en el tile de recepción, concretamente en la zona marcada por la intersección.

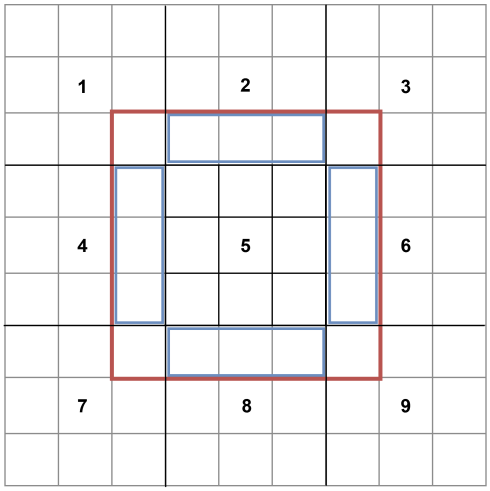


Figura 4.4: Intersección del halo de un proceso con sus vecinos

Algoritmo 1 Cálculo automático de comunicaciones entre procesos

```
 1: procedure REDISTRIBUTE(tileEnv, tileRec)
 2:     for vecino ← 1 to numProcesos do                    ▷ Calculo de envíos
 3:         tileVecinoEnv ← getTileVecino(vecino, envio)
 4:         tileVecinoRec ← getTileVecino(vecino, recepcion)
 5:         interseccion ← getInterseccion(tileEnv, tileVecinoRec)
 6:         yaPresentes ← getInterseccion(interseccion, tileVecinoEnv)
 7:         if interseccion ≠ NULL then
 8:             if yaPresentes = NULL then
 9:                 incluirEnPatron(patron, envio(vecino, tileEnv, interseccion))
10:             end if
11:         end if
12:     end for
13:     for vecino ← 1 to numProcesos do                    ▷ Calculo de recepciones
14:         tileVecinoEnv ← getTileVecino(vecino, envio)
15:         interseccion ← getInterseccion(tileRec, tileVecinoEnv)
16:         yaPresentes ← getInterseccion(interseccion, tileEnv)
17:         if interseccion ≠ NULL then
18:             if yaPresentes = NULL then
19:                 incluirEnPatron(patron, recepcion(vecino, tileRecv, interseccion))
20:             end if
21:         end if
22:     end for
23:     return patron
24: end procedure
```

4.3 Diseño

En esta sección se explican las decisiones de diseño que han sido tomadas para implementar la solución vista en la sección anterior y solucionar algunos problemas encontrados.

La primera decisión de diseño es cómo implementar la herencia vista entre HitTile y HitDistribTile en el lenguaje de programación C, que no es orientado a objetos. Esta herencia es imprescindible ya que los HitDistribTiles deben poder ser pasados como parámetro a aquellas funciones que requieren un HitTile, ya que si no carecerían de utilidad.

El resto de la biblioteca utiliza los *struct* [22] de C para simular objetos. Sin embargo, estos no permiten implementar herencia ni polimorfismo (Al menos sin hacer uso de conversiones de tipos). Por ello se decide crear una estructura de extensión, llamada HitTileDistExt que estará formado por los datos extra que contiene un tile distribuido pero no uno convencional. Por su parte, el struct HitTile contendrá un campo de tipo puntero, de forma que, si se quiere utilizar un tile distribuido, se creará una instancia del struct de extensión y se asigna el puntero del struct HitTile a dicha extensión. Por otro lado, si se quiere usar un tile convencional, ese puntero será null.

```
typedef struct HitTile {
    ...
 void   *distributed;   /**< Extension for distributed tiles  */
} HitTile ;

typedef struct {
    ...
} HitTileDistExt;
```

Código 4.1: Implemenación de polimorfismo mediante structs

Una vez resuelto el problema del polimorfismo, es necesario determinar qué campos debe contener este struct HitTileDistExt para que la función redistribute tenga la información necesaria para calcular las comunicaciones.

Una opción contemplada es incluir una lista con los shapes de los tiles de todos los procesos, es decir, que el struct almacene dos arrays de shapes, uno para los tiles de envío y otro para los de recepción, cada uno de ellos con tantos elementos como procesos haya. Sin embargo, la obtención de estos shapes requeriría comunicaciones potencialmente costosas como se estudiará en la sección 5.1.

Ante la inviabilidad de esta solución se opta por un diseño alternativo. En él solo se almacenará la información necesaria para calcular los shapes de cada proceso, y en lugar de guardar los propios shapes. Por tanto se decide añadir dos campos:

- Layout: el layout es el encargado de distribuir un tile global entre los distintos procesos, por tanto, este conoce cuál es el shape asignado a cualquiera de ellos.
- View: como se explicará en la sección 5.2, un view contiene todas las transformaciones que se han hecho sobre el shape asignado por un layout para crear los halos.

Combinando estas dos informaciones es posible saber qué shape tenía originalmente un proceso y cómo fue modificado por lo que se puede calcular su shape final.

Se añade también un campo de tipo *Domain*, que se explicará en la sección 5.3 que se usa como caché, reduciendo la cantidad de cálculos que se deben hacer en caso de que se llame varias veces a la función redistribute con el mismo tile.

El struct HitTileDistExt resultante, adaptado a las limitaciones de C se muestra en la figura 4.5.

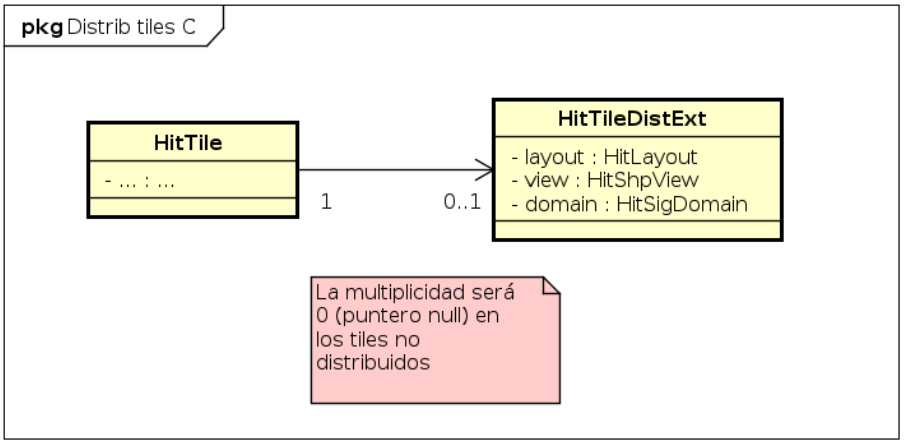


Figura 4.5: Struct HitTileDistExt

Capítulo 5

Implementación de la extensión de Hitmap

En este capítulo se tratarán los siguientes puntos:

- La versión 1 de la implementación, que usa la función MPI_Allgather.
- La versión 2 de la implementación, que utiliza los Views de Hitmap.
- La versión 3 de la implementación, que usa Domains.
- La versión 4 de la implementación, que da lugar a las nuevas estructuras de datos añadidas a Hitmap
- La versión 5 de la implementación, que implementa optimizaciones sobre las versiones anteriores.

5.1 Versión 1: Uso de MPI_Allgather

La primera propuesta de solución es la más trivial y se basa en el uso de comunicaciones para obtener los datos necesarios. Si un proceso no tiene forma de conocer qué parte de la matriz le corresponde a su vecino ambos pueden intercambiar información sobre sus shapes de forma que puedan calcular las intersecciones necesarias. Por tanto, en esta versión cada proceso enviará a todos los demás (y recibirá de ellos) información sobre el shape tanto de su tile de envío como el de recepción.

Para implementar esta solución se utiliza un método "Allgather" [23] proporcionado por MPI. Antes de llamar a este método, todos preparan un array con tantos espacios como procesos haya en total. Cada uno de ellos inicializará la posición del array correspondiente a su número de proceso o "rank". Después, todos enviarán el dato que han inicializado y recibirán los de los demás, colocándolos de nuevo en la posición correspondiente al rank. Estas recepciones conseguirán completar el array, de forma que todos los participantes en la comunicación terminen con una copia idéntica, la cual contendrá la información de qué parte de la matriz corresponde a cada proceso. La figura 5.1 muestra como tres procesos intercambian datos para constituir el array global.

En la función redistribute, el MPI_Allgather se utiliza para intercambiar dos shapes: el de envío, es decir, los datos propios del proceso (en este caso, la matriz sin halo) y el de recepción, que son los datos que espera obtener el proceso (matriz junto con sus halos). La figura 5.2 muestra

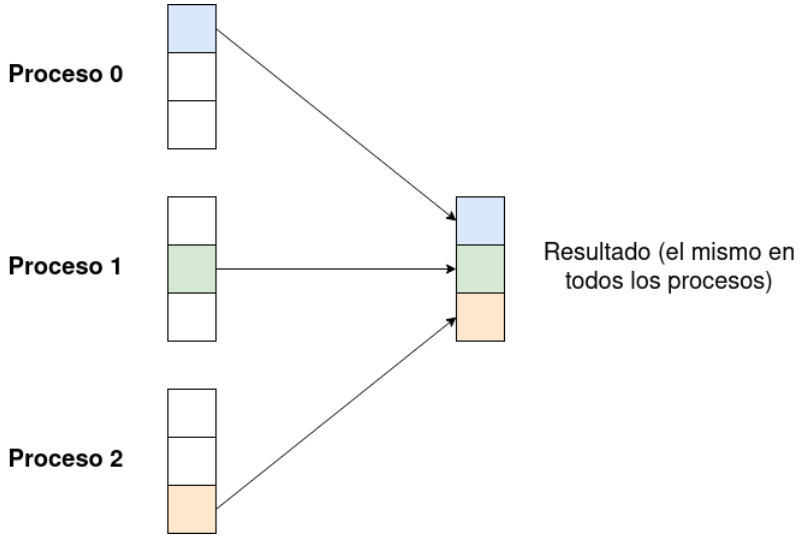


Figura 5.1: Esquema de funcionamiento de la función Allgather

la estructura del array resultado del AllGather (para matrices bidimensionales), teniendo en cuenta que cada shape está formado por dos dimensiones con sus respectivas signatures.

Una vez obtenida esta información y utilizando las funciones que aporta Hitmap para conocer los vecinos de un proceso se puede calcular el patrón de comunicación necesario. El algoritmo 2 muestra una versión simplificada de dicha función.

El algoritmo 2 es un código que ya es implementable, que permite superar la limitación de no conocer los datos asignados a un vecino. Sin embargo, aunque este método consigue el objetivo de ocultar al usuario el cálculo de las comunicaciones necesarias, tiene un inconveniente: es necesario realizar una comunicación entre todos los procesos. Este intercambio de datos puede ser costoso en escenarios en los que intervenga un gran número de procesos.

5.2 Versión 2: Eliminación de las comunicaciones con Views

Esta versión tiene como objetivo solucionar el problema de la versión 1, eliminando las comunicaciones. Para ello, es necesario encontrar una forma de calcular el shape de los procesos vecinos, que son la entrada para el proceso de intersecciones y creación del patrón.

Hitmap proporciona herramientas para solucionar este problema, pero, como se verá posteriormente, estas no son suficientes. Concretamente Hitmap dispone de la función hit_layShapeOther, que toma como parámetros el rank del proceso vecino (sus coordenadas dentro de la topología) y el layout, ambos datos conocidos por todos los procesos. El resultado de la función es el shape que representa al tile que ha sido asignado al proceso vecino por el layout en su creación.

Podría parecer que estos datos son suficientes para calcular las intersecciones, sin embargo, existe un matiz que no es contemplado por la función hit_layShapeOther: los halos. Para crear los halos se hacen modificaciones al shape proporcionado por el layout, por lo que es imposible conocer la forma final de los mismos.

Surge así la necesidad de una estructura de datos que permita almacenar las transformaciones realizadas al shape retornado por el layout, de forma que estas puedan aplicarse posteriormente al shape de un proceso vecino para conocer los datos que posee su tile realmente. Esta técnica funciona ya que el halo de todos los procesos es idéntico, es decir, todos los tiles se expanden

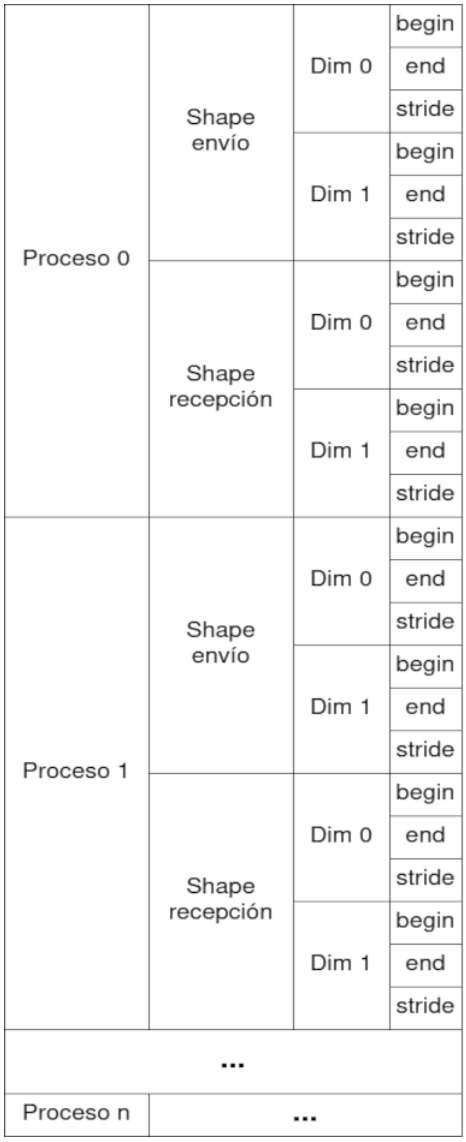


Figura 5.2: Resultado de la llamada a Allgather

el mismo número de elementos en las mismas direcciones, por lo que conocer las modificaciones realizadas localmente son suficientes para conocer las de los vecinos.

La estructura que recoge esta información recibe el nombre de View (HitShpView) y está formada por una lista enlazada de transformaciones. Cada una de las transformaciones está definida por los siguientes elementos:

- Dimensión: dimensión del tile sobre la que se llevará a cabo la operación. Existe una constante que permite especificar que la acción se realizará sobre todas las dimensiones.
- Offset: cantidad de elementos que se desplazará o escalará (dependiendo de la acción) el shape.
- Acción: operación que se va a llevar a cabo. Existen varias acciones:
  - HIT_SHAPE_BEGIN: Mueve la coordenada inicial de la dimensión especificada el número de posiciones indicado por el offset
  - HIT_SHAPE_END: similar a la anterior, pero en esta ocasión con la coordenada final

Algoritmo 2 Cálculo de comunicaciones con AllGather

```
1: procedure REDISTRIBUTE(layout, tileEnv, tileRec)
2:     if estoyActivo(layout) ≠ true then          ▷ Solo hacer cálculos con procesos activos
3:         return
4:     end if
5:
6:     ▷ Inicializar elementos del array correspondientes al proceso
7:     tileShapes[myRank][0] ← getShapeDeTile(tileEnv)
8:     tileShapes[myRank][1] ← getShapeDeTile(tileRec)
9:
10:    allGahter(tileShapes)                                   ▷ Comunicar shapes
11:
12:    for vecino ← 1 to numProcesos do                        ▷ Calculo de envíos
13:        tileVecinoEnv ← tileShapes[vecino][0]
14:        tileVecinoRec ← tileShapes[vecino][1]
15:        interseccion ← getInterseccion(tileEnv, tileVecinoRec)
16:        yaPresentes ← getInterseccion(interseccion, tileVecinoEnv)
17:        if interseccion ≠ NULL then
18:            if yaPresentes = NULL then
19:                incluirEnPatron(patron, envio(vecino, tileEnv, interseccion))
20:            end if
21:        end if
22:    end for
23:
24:    for vecino ← 1 to numProcesos do                        ▷ Calculo de recepciones
25:        tileVecinoEnv ← tileShapes[vecino][0]
26:        interseccion ← getInterseccion(tileRec, tileVecinoEnv)
27:        yaPresentes ← getInterseccion(interseccion, tileEnv)
28:        if interseccion ≠ NULL then
29:            if yaPresentes = NULL then
30:                incluirEnPatron(patron, recepcion(vecino, tileRecv, interseccion))
31:            end if
32:        end if
33:    end for
34:    return patron
35: end procedure
```

- HIT_SHAPE_MOVE: mueve la coordenada inicial y final o, lo que es lo mismo, mueve el shape la cantidad especificada.
- HIT_SHAPE_STRETCH: decrementa la coordenada inicial y aumenta la final. Esto "estira" el shape por ambos extremos la cantidad indicada de elementos.
- HIT_SHAPE_FIRST / HIT_SHAPE_LAST: selecciona los n primeros / últimos elementos del shape, siendo n el valor del offset

Esta lista de transformaciones se realiza secuencialmente sobre un tile (el resultado de una transformación es la entrada de la siguiente). Esto permite crear fácilmente halos en un tile, ya que basta con estirar el shape las unidades deseadas en todas las dimensiones.

Como se puede ver en el ejemplo de código 5.1, el constructor de views toma como parámetro una lista de tuplas de la forma (dimensión, acción, offset). Esta sintaxis de tuplas se consigue

gracias al uso de macros de C.

```
HitShpView view=hitShpView((HIT_SHAPE_ALLDIMS, HIT_SHAPE_STRETCH, 1));
```

Código 5.1: Ejemplo de uso de un View

Hay que destacar que cada proceso crea una instancia de un view, pero todos ellos serán iguales (todos construyen sus halos de manera idéntica). Una vez se tiene el shape original de cada proceso (como se mencionó anteriormente) y las transformaciones que se han realizado sobre este view, es posible reconstruir los elementos que tiene exactamente cada proceso aplicando dichas transformaciones. Tras realizar estos cálculos, el proceso de creación del patrón mediante intersecciones es igual al del apartado anterior (algoritmo 2).

Aunque se haya eliminado la comunicación de shapes entre procesos, este método aún es optimizable, especialmente en casos específicos. En concreto, vamos a suponer un stencil en el que se acceda a los vecinos adyacentes pero no diagonales. Los elementos de los vecinos a los que un proceso debe acceder se muestran en la figura 5.3.

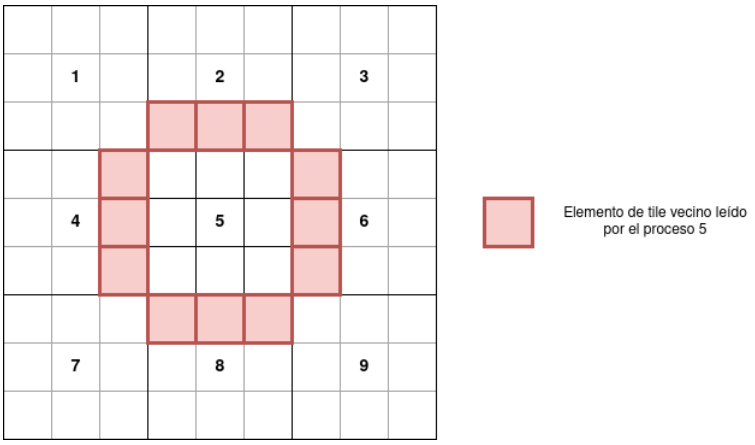


Figura 5.3: Elementos de tiles vecinos leídos en un stencil básico

Sin embargo, al hacer la intersección del tile y su halo con los tiles de los demás procesos se generan comunicaciones para obtener los elementos de las esquinas aunque, como se ha visto, estos no vayan a ser utilizados. Estas comunicaciones se muestran en la figura 5.4.

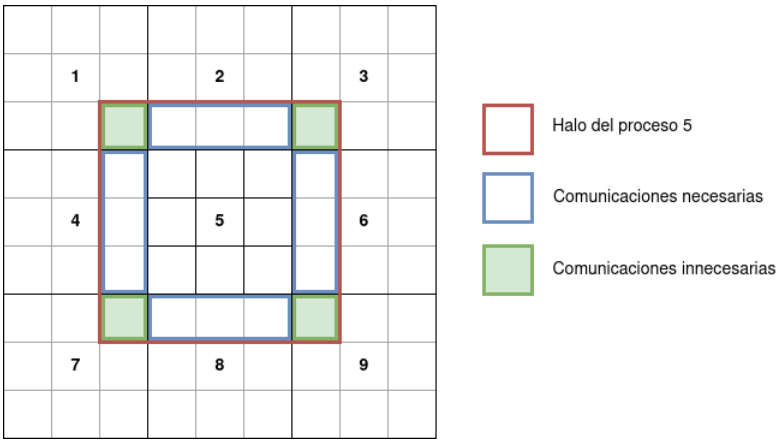


Figura 5.4: Comunicaciones reales entre procesos, incluyendo las innecesarias

Esta forma de utilizar los views, al estar limitada a los shapes rectangulares, no permite representar de manera eficiente comunicaciones complejas, por lo que es necesario implementar una nueva forma de utilizarlos.

5.3 Versión 3: Domains, shapes no rectangulares

Esta propuesta pretende solucionar el problema de la versión 2 y adaptarse a cualquier patrón de comunicación. Para ello, es necesario adoptar un nuevo enfoque respecto a los views. Como se ha explicado en la solución anterior, los views se trataban como una sucesión de transformaciones que se aplicaban secuencialmente a un shape. Para conseguir mayor flexibilidad, en esta nueva versión, los views se interpretarán como transformaciones que se hacen cada una de ellas sobre shape inicial. Después, se realizará la unión de todos los shapes resultado, formando así un Domain (que no tiene que ser rectangular como sí lo es un shape).

En las siguientes figuras se muestran las dos opciones de uso de los views: construcción de un shape (Figura 5.5) y de un domain (Figura 5.6). El view que se utilizará de ejemplo está formado por dos transformaciones: estirar el shape en al dimensión 0 y hacer lo mismo en la 1:

```
( (0, HIT_SHAPE_STRETCH, 1), (1, HIT_SHAPE_STRETCH, 1) ).
```

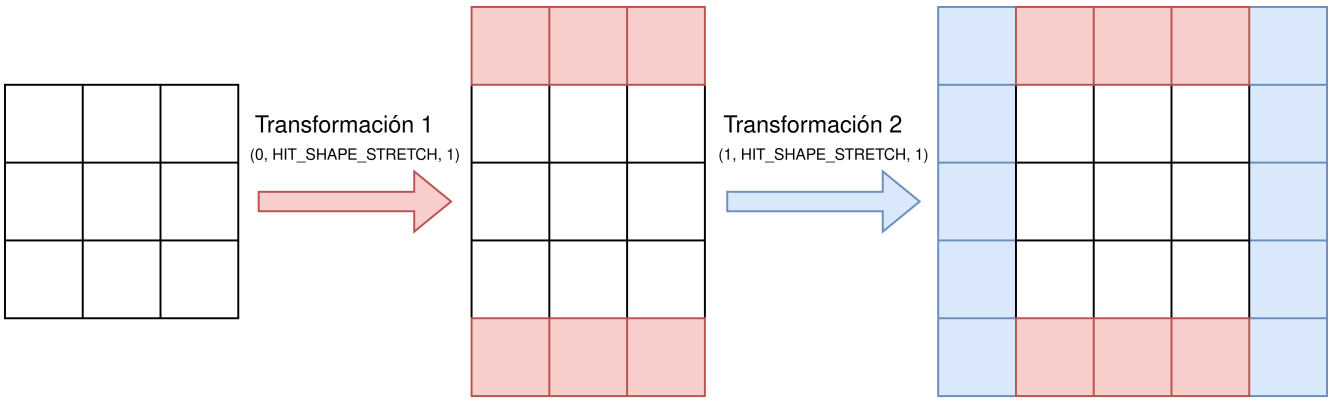


Figura 5.5: Aplicación secuencial de un view para construir un shape

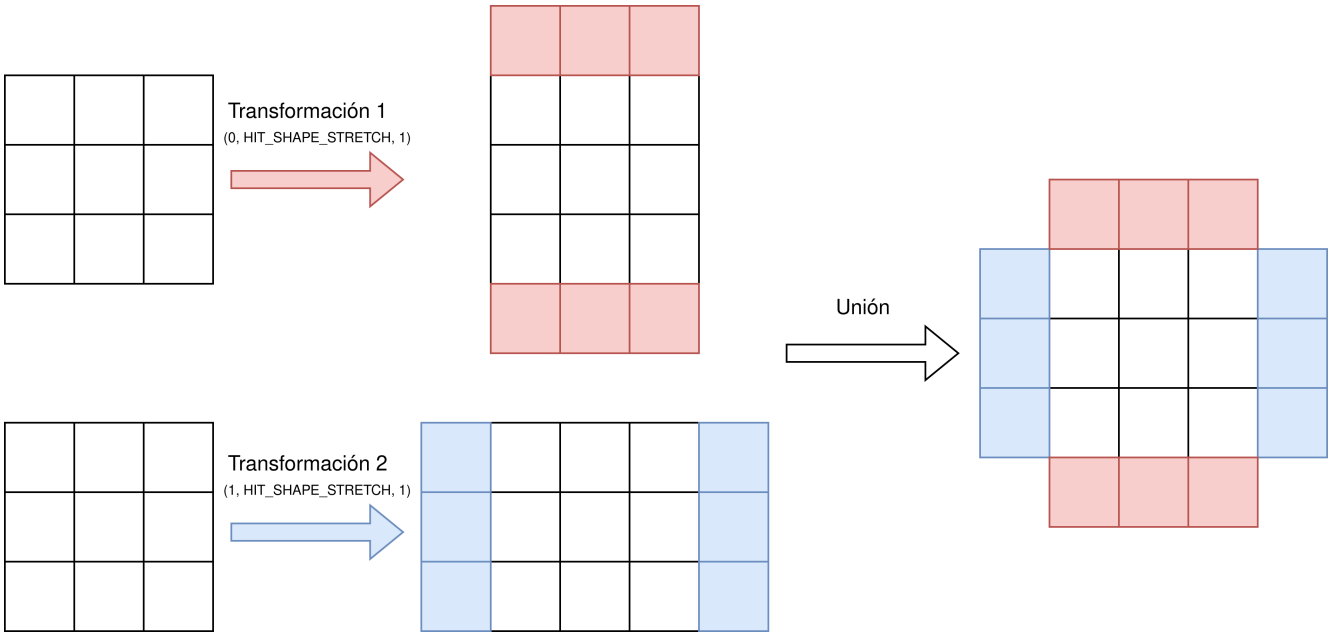


Figura 5.6: Unión de transformaciones de un view para formar un domain

Al tener los domains mayor detalle que los shapes, la intersección con los vecinos solo incluye en las comunicaciones aquellos elementos que son estrictamente necesarios. Esto no solo disminuye

la cantidad de datos enviados en una comunicación (reduciendo potenciales cuellos de botella), sino que además disminuye el número de comunicaciones establecidas, ya que, en el ejemplo visto (Figura 5.3), solo será necesario intercambiar datos con cuatro procesos (en lugar de con ocho).

Una particularidad de este método es que la intersección del domain local con los domains de los vecinos no da como resultado un shape (una forma rectangular) como sí ocurría en los ejemplos anteriores, sino que se obtiene un nuevo domain (que puede tener cualquier forma). Las comunicaciones en Hitmap están preparadas para funcionar con shapes rectangulares, por lo que es necesaria una adaptación. Para solucionar el problema, Hitmap permite obtener los shapes rectangulares que constituyen el domain. Esta operación es trivial, ya que el domain internamente es una colección de shapes. Sin embargo, donde reside la verdadera complejidad es en el funcionamiento interno de Hitmap, que es capaz de determinar el número mínimo de shapes capaz de formar el domain completamente sin repetir ningún elemento, haciendo así más eficientes las comunicaciones.

Además de la ya vista adaptación necesaria para utilizar domains en las comunicaciones, también surge la necesidad de adaptar los tiles para el uso de estos domains. Hay que recordar que tanto los shapes como los domains sirven para identificar a ciertos elementos de un tile, pero es el tile el que realmente contiene la memoria donde se almacenan los datos. Aunque los domains tengan formas extrañas, estos tiles tienen que estar formados por una región de memoria rectangular. En esta versión, es responsabilidad del programador crear un tile rectangular suficientemente grande para que pueda albergar el domain, es decir, los elementos del domain deben ser un subconjunto de los elementos del tile como se muestra en la figura 5.7.

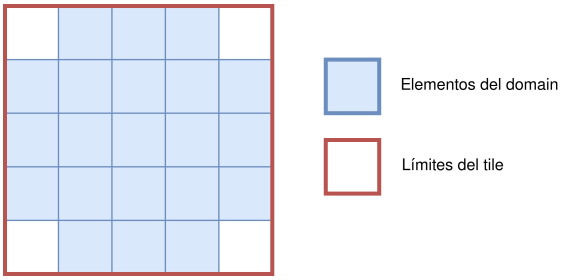


Figura 5.7: Elementos de un domain junto con el tile en el que están almacenados

5.4 Versión 4: Tiles distribuidos

Esta propuesta de solución va más allá de las anteriores, ya que no se limita solo a implementar la función redistribute, sino que cambia la forma en la que se construyen los tiles. Por tanto, se modifica la estructura fundamental de Hitmap (HitTile).

El objetivo de estas modificaciones es esconder aún más todos los conceptos añadidos en las versiones anteriores, haciendo el proceso prácticamente transparente al usuario, que solo sabrá que ha creado un tile que se distribuirá automáticamente (incluyendo la comunicaciones) entre todos los procesos.

Por tanto, ahora la estructura tile almacenará datos como el layout (ya no será gestionado por el programador en la aplicación que esté usando Hitmap), el view utilizado para generar el tile a partir del shape original retornado por el layout (por ejemplo añadiendo halos) y el domain

formado al aplicar el view (podría ser calculado en cualquier momento al tener el layout y el view, pero se almacena a modo de caché).

Otra de las ventajas que proporcionan los tiles distribuidos es que ya no es responsabilidad del programador determinar el tamaño necesario del tile de forma que el domain creado quepa, es decir, ya no hay que crear a mano un tile suficientemente grande para dejar espacio a los halos. Esta tarea se realiza automáticamente al construir el tile. Tras crear el domain, se llama a una función que calcula su bounding box, es decir, la "caja" más pequeña capaz de albergar todos los datos [24]. En el caso de Hitmap, esto se traduce en el shape mínimo capaz de contener el domain.

Aunque existen algoritmos complejos que permiten calcular el bounding box de una nube de puntos en tiempo lineal [25], en este caso se usará un algoritmo mucho más sencillo, ya que solo se está trabajando con un conjunto pequeño de rectángulos y el código se usa solo en la inicialización, por lo que no es necesario llevar a cabo una gran optimización en esta parte. Además, los algoritmos más optimizados solo permiten calcular el bounding box en dos dimensiones o aproximarlo en tres, pero en Hitmap se puede trabajar con cualquier número de dimensiones.

El algoritmo 3 muestra el proceso seguido: para cada dimensión se recorren los shapes que constituyen el domain. Para cada uno de estos, se lee su signature, estructura que contiene el valor mínimo y máximo de esa dimensión. Estos valores se compararán con el valor mínimo y máximo que se tenían ya almacenados, obteniendo así los valores más extremos en cada dimensión, lo cual formará el shape resultado.

Algoritmo 3 Cálculo del bounding box de un domain

```
1:  procedure BOUNDINGBOX(domain)
2:      result ← HIT_SHAPE_NULL
3:      for dim ← 0 to numDims do
4:          ▷ Inicializar máximo y mínimo con el primer elemento
5:          signature ← getSignature(getElementoDomain(domain, 0), dim)
6:          min ← signature.begin
7:          max ← signature.end
8:          for shape ← 1 to getTamDomain(domain) do
9:              signature ← getSignature(getElementoDomain(domain, shape), dim)
10:             if signature.end > max then
11:                 max ← signature.end
12:             end if
13:             if signature.begin < min then
14:                 min ← signature.begin
15:             end if
16:         end for
17:         ▷ Incluir dimensión en el resultado
18:         resultado.signature[dim] ← signature(min, max)
19:     end for
20:     return resultado
21: end procedure
```

Al contener el tile distribuido toda la información que se requiere para el cálculo de las comunicaciones, la llamada a la función redistribute pierde algunos parámetros. Como se ve en el código 5.2 ahora solo es necesario pasar el tile de envío y el de recepción, ya que el layout, domain y view están contenidos en el propio tile. Esto hace además que el código sea más comprensible para el programador, ya que la parte donde se definen los halos (es decir, creación del view) está

relacionada con la creación de los tiles, y no tiene que ser "arrastrada" hasta la zona donde se crean las comunicaciones.

```
1 HitPattern patron = hit_patternRedistributeDis(&tileEnvio, &
    tileRecepcion, HIT_DOUBLE);
```

Código 5.2: Llamada a la nueva función redistribute

5.5 Versión 5: Comunicaciones en el mismo proceso

Como se observó en las pruebas de caja negra PCN-2d4-1d y PCN-2d4-2d, todos los procesos añadían una comunicación extra a su patrón, en la cuál se enviaban datos a sí mismos. Ante este resultado, se decide comprobar el impacto que tienen estas comunicaciones en los tiempos de ejecución. Como se puede observar en la sección 7.2.1, estos aumentan notablemente, por lo que se decide tomar medidas para solucionar este problema.

Observando el algoritmo 1 se puede encontrar la causa. Para calcular las comunicaciones, se recorren todos los procesos, se obtienen sus shapes, y se realiza la intersección con el shape del proceso actual. Sin embargo, no se comprueba que el proceso vecino sea el mismo que el proceso actual. Si se diera ese caso, el resultado de la intersección sería el tile completo ya que los dos tiles de entrada son iguales.

Por tanto, no solo se está haciendo una comunicación innecesaria (el proceso se envía datos que ya tiene), sino que además el número de elementos que la componen es extremadamente grande (todo el tile).

La primera solución que se pensó para este problema fue la más trivial, añadir una condición que asegure que el proceso vecino no sea igual al proceso que está realizando el cálculo. Esto, obviamente, resuelve el problema del tiempo de ejecución, pero no es suficiente.

Un análisis más detallado de las suposiciones que se han hecho en la implementación de esta solución revela que esta no es correcta. Antes se ha dicho que la intersección se realizará con dos tiles iguales, pero, esto es solo parcialmente cierto.

En el caso de los stencils solo se trabaja con una matriz, por lo que los datos serán enviados y recibidos en el mismo tile. Sin embargo, los tiles distribuidos son un mecanismo potente que pretende adaptarse a cualquier problema. Por tanto hay que contemplar aquellos problemas que trabajen con dos matrices A y B, en los cuales cada proceso envía datos de la matriz A a la B de todos los vecinos. En ese caso puede ser deseable que se envíen los datos propios de A a B, por lo que la comunicación consigo mismo sí es necesaria.

Como se puede comprobar, dependiendo del escenario concreto esta comunicación puede ser necesaria o no, por lo que la solución más correcta es pasar un flag a la función redistribute que, si es 1, permita a los procesos añadir comunicaciones con ellos mismo.

Capítulo 6

Implementación de los caso de uso (stencils)

En este capítulo se tratarán los siguientes puntos:

- Los tipos de stencil considerados durante el proyecto.
- Las optimizaciones realizadas sobre los stencils.

6.1 Tipos de stencil considerados

Como se comentó en la sección 3.1, un stencil es un algoritmo iterativo, en el cual se calculan los elementos de una matriz a partir de sus valores en la iteración anterior. En esta sección se detallarán los stencils que han sido desarrollados como parte del proyecto y que constituyen el conjunto de pruebas utilizado para validar el código tanto en corrección como rendimiento.

En concreto, los stencils implementados son:

- 2d9 compacto: este stencil es uno de los ejemplos más sencillos que se pueden construir en 2d. Cada elemento de la matriz se calcula como media de todos los vecinos (incluyendo las diagonales) y él mismo. La figura 6.1 representa gráficamente todos los elementos leídos para llevar a cabo el cálculo.

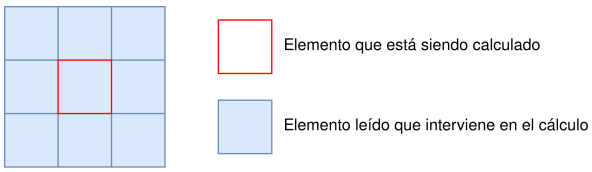


Figura 6.1: Elementos leídos en un stencil 2d9 compacto

- 2d9 no compacto: como el nombre indica, se trata una vez más de un stencil 2d en el que se calcula la media de nueve elementos. Sin embargo, se trata de un stencil no compacto, por lo que los vecinos ya no son siempre adyacentes. En concreto, para cada elemento se accede a los dos vecinos superiores, inferiores a la derecha y a la izquierda. Una vez más, el noveno elemento es aquel que se encuentra en la misma posición que el dato que se está calculando. La figura 6.2 muestra el comportamiento de este stencil.

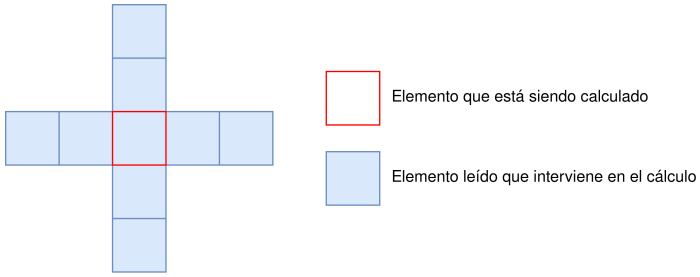


Figura 6.2: Elementos leídos en un stencil 2d9 no compacto

- 2d5 no compacto asimétrico: en este stencil 2d de 5 elementos, al igual que en el caso anterior, no se accede solo a los vecinos adyacentes al ser no compacto. Además presenta la novedad con respecto a los anteriores de ser un patrón de acceso asimétrico. En este caso en particular, se accede a los dos datos superiores, los dos izquierdos y el que se encuentra en la diagonal que ambos forman. La figura 6.3 muestra el comportamiento de este stencil.

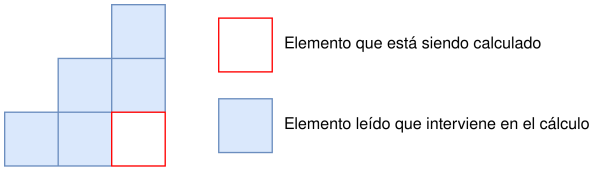


Figura 6.3: Elementos leídos en un stencil 2d5 no compacto asimétrico

Este stencil tiene una particularidad en su esquema de comunicaciones: se genera un efecto de pipeline. Esto significa que la información fluye en una única dirección (hacia la diagonal inferior derecha) en contraposición a los ejemplos anteriores, donde las comunicaciones se realizan en pares (por ejemplo un proceso envía su primera fila hacia arriba y su vecino superior le envía datos hacia abajo)

- 3d27: este stencil es equivalente al 2d9 en su sencillez. Sin embargo, incorpora un nuevo elemento, añade una nueva dimensión a las comunicaciones (pasan de ser bidimensionales a tridimensionales). En la siguiente figura se muestran los elementos que intervienen en el cálculo de la media (que incluye al elemento que está siendo calculado, aunque no se pueda representar en la ilustración). La figura 6.4 muestra el comportamiento de este stencil.

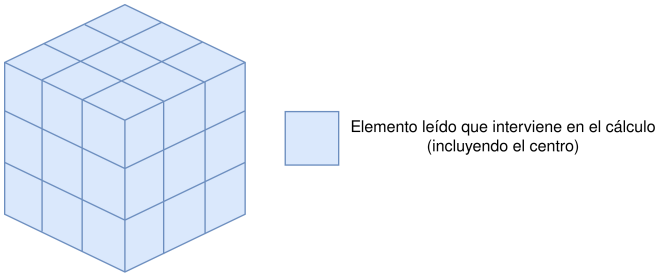


Figura 6.4: Elementos leídos en un stencil 3d27

- 3d33: al igual que el caso anterior, se trata de un stencil tridimensional. La peculiaridad es que en este caso no solo se accede a los 26 elementos adyacentes y el centro, sino que también se leen los elementos a dos posiciones de distancia del centro en las 6 direcciones. La figura 6.5 muestra el comportamiento de este stencil.

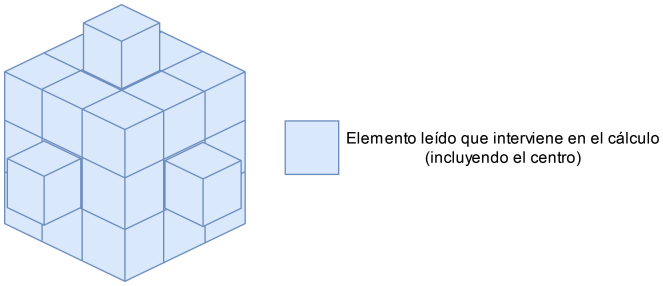


Figura 6.5: Elementos leídos en un stencil 3d33

6.2 Implementación en Hitmap

En esta sección se muestra, por medio del código 6.1, el esquema de la implementación de un stencil haciendo uso de Hitmap. El objetivo es demostrar que se puede desarrollar un gran número de stencils modificando unas pocas líneas del código de Hitmap.

Las dos primeras líneas de este código son necesarias para la inicialización del programa y creación de los tipos de datos. Las líneas 3 y 4 crean la matriz global compartida por todos los procesos, mientras que la 5 crea el tile local, indicando el nombre del layout y la topología que se usará para el reparto. En su último parámetro se determina el view que constituirá los halos del tile (en este caso se extiende el shape en una unidad en todas las dimensiones).

Hitmap permite realizar el cómputo solo en procesos activos (línea 6). Para cada uno de ellos, se calcula el patrón de comunicaciones (únicamente se necesita la línea 7) y se entra en el bucle donde se realizan los cálculos e intercambian los resultados (utilizando hit_patternDo). Las últimas líneas liberan estructuras en memoria y finalizan el programa.

```
1  hitNewType( double );
2  hit_comInit( &argc, &argv );
3  HitTile_double matrix;
4  hit_tileDomain( &matrix, double, 2, rows, columns );
5  HitTile_double tileMat = hitDistribTile(double, matrix,
       Array2DComplete, Blocks, hitShpView((HIT_SHAPE_ALLDIMS,
       HIT_SHAPE_STRETCH, 1)));
6  if ( hit_layImActive( hit_tileLayout(tileMat) ) ) {
7      HitPattern neighSync = hit_patternRedistributeDis(&tileMat, &
           tileMat, HIT_DOUBLE, 0);
8
9      for (int loopIndex = 0; loopIndex < numIter; loopIndex++) {
10         updateCell(...);
11         hit_patternDo( neighSync );
12     }
13
14     hit_tileFree( tileMat );
15     hit_patternFree( &neighSync );
16 }
```

```
17 hit_comFinalize();
```

Código 6.1: Ejemplo de implementación de un stencil

Si se quisiera implementar un stencil diferente, que acceda a diferentes datos o incluso con distinto número de dimensiones los cambios en el código serían mínimos. Exceptuando las modificaciones que se deban realizar en la ficticia función updateCell, utilizada en este ejemplo para representar el algoritmo implementado y que por tanto es ajena a Hitmap, solo sería necesario cambiar la creación del tile indicando el nuevo número de dimensiones y el nuevo view, que permita generar el halo necesario para ese stencil.

6.3 Optimizaciones implementadas

En esta sección se muestran algunas mejoras realizadas a la implementación básica de los stencils. El principal objetivo es mostrar la flexibilidad que ofrece Hitmap en el desarrollo de stencils, ya que es posible añadir modificaciones e implementar técnicas avanzadas modificando solo unas pocas líneas del código encargado de distribuir y comunicar las estructuras de datos. Por ese motivo, se corregirán algunos inconvenientes que presentan las versiones básicas y se implementarán algunas técnicas conocidas en el mundo de la programación paralela, todo ello sin apenas cambios en el código (salvo, obviamente, en la parte referente al algoritmo).

6.3.1 Eliminación de copias redundantes

Como ya se ha explicado, en un stencil los valores de la matriz se calculan haciendo la media de determinados valores obtenidos en la iteración anterior. Sin embargo esto plantea un problema, a medida que se va avanzando en el cálculo, los valores de la iteración anterior se van sobrescribiendo, perdiendo así los datos de entrada de futuras operaciones.

Esto se puede ilustrar con un stencil sencillo, como es el 2d4 (un elemento se calcula como la media de sus cuatro vecinos). En este ejemplo para calcular cada dato es necesario acceder a su vecino izquierdo pero, al estar recorriendo los elementos de las filas, el cálculo anterior sobrescribió el valor que tenía este vecino izquierdo, lo cual hace que no se pueda calcular el elemento actual.

Este problema tiene una solución trivial: crear dos matrices, una de lectura que almacene una copia de los datos y otra de escritura donde se guarden los resultados. Antes de cada iteración será necesario copiar los valores de la matriz de resultado a la matriz de copia. Este método era el utilizado en los stencils que formaban parte de los ejemplos de Hitmap y que se han utilizado como referencia. Esta solución se puede ver implementada en el código 6.2

```
1 int stage;
2 for ( stage=0; stage<STAGES; stage++ ) {
3  // Actualizar copia
4  for ( i=0; i<ROWS; i++ )
5   for ( j=0; j<COLS; j++ )
6    copy[i][j] = mat[i][j];
7
8  // Calculos
9  for ( i=1; i<ROWS-1; i++ )
```

```
10   for ( j=1; j<COLS-1; j++ )
11    mat[i][j] = ( copy[i-1][j] + copy[i+1][j] + copy[i][j-1] + copy[i][
         j+1] ) / 4;
12 }
```

Código 6.2: Copia de datos antes del cómputo

Aunque esta solución funciona, tiene un claro inconveniente, ya que es necesario hacer una copia de toda la matriz en cada una de las iteraciones del programa. Sin embargo, existe una solución que también garantiza la obtención de los resultados correctos pero sin necesidad de copias.

Esta consiste en intercambiar los punteros de las matrices de lectura y escritura en lugar de hacer una copia de datos. Así, llamando a ambas matrices A y B, la primera iteración leería los valores de A y guardaría sus resultados en B. La siguiente iteración invertiría los papeles, leyendo de B (resultados de la iteración anterior, que es lo mismo que habría en la matriz de lectura) y escribiendo su resultado en A, sobrescribiendo unos datos que ya no se van a utilizar más, ya que eran la entrada de la iteración anterior. Realizar este proceso repetidamente permite obtener resultados correctos en el stencil sin realizar ninguna copia. Un ejemplo de este método puede observarse en el código 6.3.

```
1  int stage;
2  for ( stage=0; stage<STAGES; stage++ ) {
3   //Intercambio de punteros
4   tmp = copy;
5   copy = mat;
6   mat = tmp;
7
8   // Calculos
9   for ( i=1; i<ROWS-1; i++ )
10   for ( j=1; j<COLS-1; j++ )
11    mat[i][j] = ( copy[i-1][j] + copy[i+1][j] + copy[i][j-1] + copy[i][
         j+1] ) / 4;
12 }
```

Código 6.3: Intercambio de punteros para evitar la copia de datos

A pesar de su conveniencia, este método presenta algunas particularidades que deben ser tenidas en cuenta. La primera de ellas es que ambas matrices tienen que ser inicializadas escribiendo en ella sus valores por defecto. En versiones anteriores a esta optimización era suficiente con inicializar una de las matrices, cuyos datos serían volcados durante la copia a la otra, por lo que no se leería ningún valor sin inicializar. Sin embargo, en este nuevo procedimiento es necesario inicializar ambas, ya que posiciones como los bordes no serán escritas nunca durante el cómputo, por lo que mantendrán su valor inicial (que sería basura de no ser inicializado).

También hay que tener en cuenta otro matiz, en este caso relativo a la implementación del stencil en Hitmap. Las comunicaciones se expresan utilizando estructuras denominadas patrones de comunicación. Estos contienen una referencia al tile del que se leen los datos a enviar y al tile de recepción. Sin embargo, al estar intercambiando los punteros a las matrices, estos tiles cambiarán cada iteración. Volviendo de nuevo a la nomenclatura antes usada (matrices A y B), en la primera

iteración los resultados se escribirían en B, por lo que los procesos deberán intercambiar los valores del tile B. En la siguiente iteración los datos se escriben en A, pero, como el patrón almacena una referencia a la matriz B, las comunicaciones se efectuarían sobre una matriz errónea.

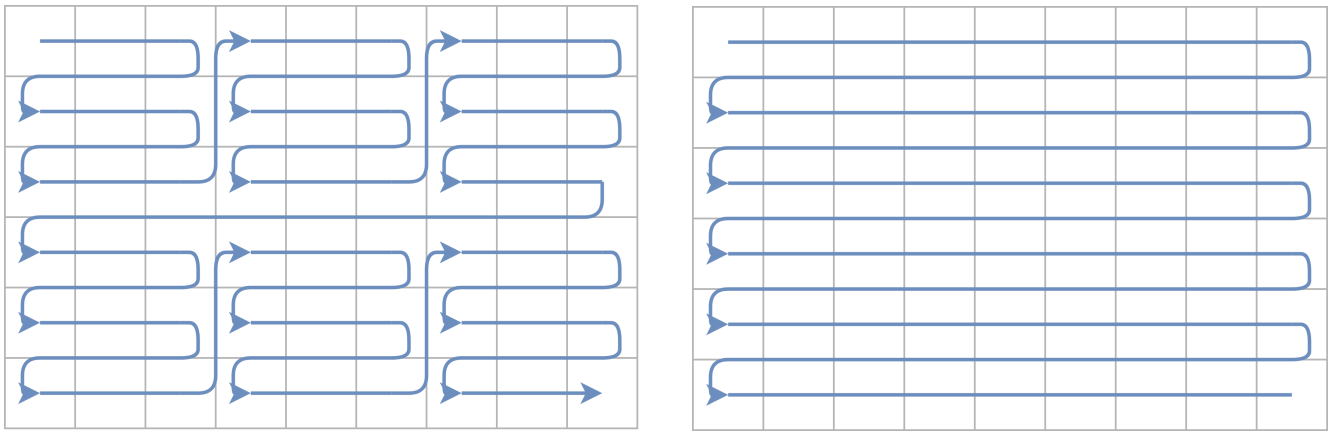
La solución a este problema es sencilla, y pasa una vez más por el intercambio de estructuras. En concreto se crean dos patrones de comunicación, uno en el que se intercambien datos entre las matrices A y otro para las B. Así, en la primera iteración se usará el patrón B, pero en la segunda se intercambiarán los patrones para que se utilice el A, leyendo así los datos de la matriz donde han sido escritos, la A.

6.3.2 Reducción del tamaño del tile

Como se explica en la sección 1.1.3, Hitmap está diseñado para dar soporte al tiling, dividiendo las estructuras de datos de forma automática entre los procesadores disponibles. Sin embargo, es posible aplicar esta técnica en más de un nivel. En concreto, en este ejemplo se utilizará también la aproximación más convencional al tiling reescribiendo los bucles.

Para implementar esta técnica no se necesitará cambiar nada en el código de Hitmap, únicamente la estructura de los bucles for que recorren los elementos. El rango de cada uno de estos bucles se dividirá en partes iguales de tamaño configurable en tiempo de compilación (para favorecer las posibles optimizaciones del compilador). Esto genera un patrón de acceso como el mostrado en la figura 6.6a.

En esta versión, al conocer el tamaño de las filas y columnas con las que se trabajan (por haber fijado el tamaño del tile), el comportamiento de las cachés es uniforme y predecible, evitando así los problemas que se manifestaron en las pruebas (sección 7.2.3).



(a) Recorrido de los datos de una matriz con reescritura de bucles

(b) Recorrido de los datos de una matriz sin reescritura de bucles

Figura 6.6: Comparación del recorrido de datos con y sin tiling

6.3.3 Halos con cardinalidad variable

En las aplicaciones paralelas el intercambio de datos entre procesos o nodos constituye gran parte del tiempo total de ejecución. Por ese motivo, en esta nueva implementación de los stencils se intentará reducir el número de comunicaciones realizando más cálculos por proceso.

El objetivo es reducir el número de comunicaciones, de forma que los procesos no tengan que intercambiar datos con los vecinos al final de cada iteración, sino que puedan realizar una serie

de iteraciones seguidas antes de necesitar comunicar. Esto implica que en algunas iteraciones no se dispondrá de los valores actualizados de los vecinos, por lo que los procesos deberán duplicar cálculos para obtener por ellos mismos los valores de los vecinos. Por tanto, los halos deben tener mayor tamaño para poder albergar los datos de estos cálculos extra.

La figura 6.7 muestra todos los datos de los que se debe disponer para calcular el valor de un elemento en el borde de la matriz tras 3 iteraciones. Las flechas indican los datos que se usan como operandos para calcular el valor señalado por la cabeza. Aquellos datos marcados en gris corresponden al halo de la matriz. Como se puede comprobar, para el calcular los bordes de la matriz cada proceso no necesitará un solo elemento como ocurría antes, sino que necesitará un halo con tantas posiciones como iteraciones se quieran calcular antes de hacer una comunicación.

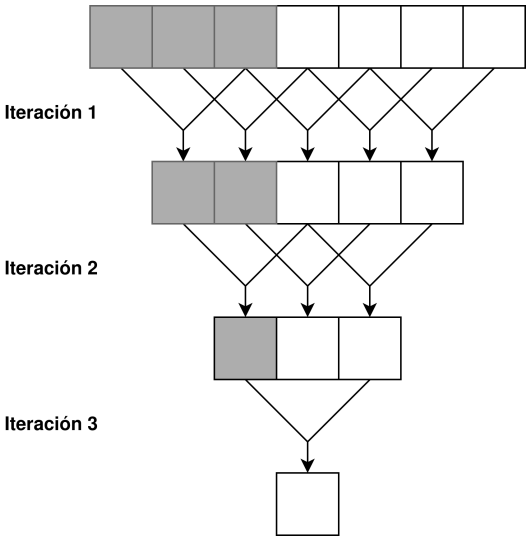


Figura 6.7: Datos necesarios para calcular un elemento tras 3 iteraciones

Para el cálculo de la primera iteración, se leerá el primer elemento del halo, pero, en la segunda, este ya no será necesario, por lo que no hace falta actualizar su valor en la primera iteración. Lo mismo ocurre con el segundo elemento del halo, será utilizado en las iteraciones 1 y 2 pero no en la 3, por lo que no necesita ser actualizado en la iteración 2, es suficiente con hacerlo en la 1. Por tanto, en cada iteración se dejará de utilizar un elemento de cada extremo, reduciendo así el número de cálculos necesarios. La figura 6.8 muestra como se va reduciendo la región de datos calculados. Las líneas negras más gruesas delimitan los tiles de cada proceso y la región marcada en azul identifica los elementos que serán calculados en esa iteración. Todos aquellos datos marcados en azul pero fuera del tile central son cálculos repetidos, ya que pertenecen a otro proceso, que es quien debería calcularlos. Sin embargo, esta redundancia en los cálculos se ve compensada por la reducción de comunicaciones, ya que, en este ejemplo, los procesos solo intercambiarán datos cada 4 iteraciones.

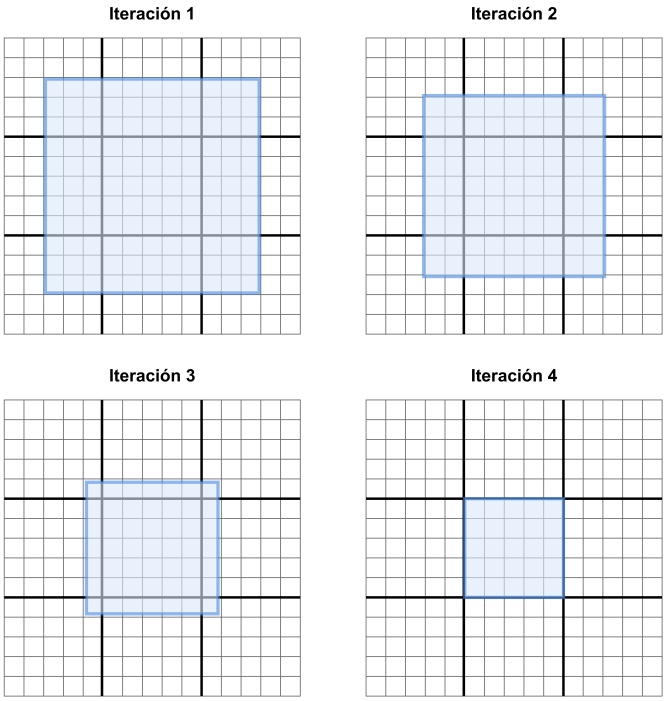


Figura 6.8: Evolución de los datos calculados en 4 iteraciones

Capítulo 7

Pruebas

En este capítulo se tratarán los siguientes puntos:

- Las pruebas de corrección, que incluyen test unitarios y comparación de resultados con códigos de referencia
- Los pruebas de rendimiento, que comparan los tiempos de ejecución de diferentes versiones del programa.
- Las pruebas de comparación con códigos de referencia, que evalúan el rendimiento del código nuevo respecto al que existía antes de iniciar el proyecto.

7.1 Pruebas de corrección

7.1.1 Test unitarios

A continuación se muestra una parte de los tests unitarios realizados durante el proyecto. Algunos de ellos, debido al gran número de líneas de su salida, han sido omitidos en este documento. En cada prueba de caja blanca o caja negra se indicará su nombre, descripción, valores esperados (se mostrarán ordenados para mejorar su legibilidad, pero cualquier orden es admitido), valores obtenidos siempre que sean diferentes de los esperados (con las diferencias señaladas en negrita), resultado y descripción de la solución al error si procede.

Caja negra

Los tests de caja negra son aquellos que han sido diseñados ignorando los detalles de implementación de las funciones y considerando solo su descripción e interfaz (parámetros de entrada y salida).

En concreto, se han simulado los halos generados por distintos tipos de stencils y se llamado a la función redistribute para que calcule el patrón de comunicaciones correspondiente. Después, se imprime los resultados de esta función, es decir, cada proceso muestra de qué vecinos recibe datos y a quiénes envía. Se han omitido los stencils tridimensionales debido a que tienen un número mayor de layouts posibles y muchos más intercambios entre procesos.

PCN-2d4-1d	
Nombre	Comunicaciones en stencil 2d4 con layout 1d
Descripción	Se crea una matriz con halos como los que utiliza un stencil 2d4. Después, se calculan las comunicaciones necesarias. Para cada proceso, se imprime su identificador (entre corchetes) y de qué proceso envía o recibe datos.
Valores esperados (ordenados)	[0] receives from 1 [0] sends to 1 [1] receives from 0 [1] receives from 2 [1] sends to 0 [1] sends to 2 [2] receives from 1 [2] receives from 3 [2] sends to 1 [2] sends to 3 [3] receives from 2 [3] receives from 4 [3] sends to 2 [3] sends to 4 [4] receives from 3 [4] receives from 5 [4] sends to 3 [4] sends to 5 [5] receives from 4 [5] receives from 6 [5] sends to 4 [5] sends to 6 [6] receives from 5 [6] receives from 7 [6] sends to 5 [6] sends to 7 [7] receives from 6 [7] receives from 8 [7] sends to 6 [7] sends to 8 [8] receives from 7 [8] sends to 7
Valores obtenidos	**[0] receives from 0** [0] receives from 1 **[0] sends to 0** [0] sends to 1 [1] receives from 0 **[1] receives from 1** [1] receives from 2 [1] sends to 0 **[1] sends to 1** [1] sends to 2 [2] receives from 1 **[2] receives from 2** [2] receives from 3 [2] sends to 1 **[2] sends to 2** [2] sends to 3 [3] receives from 2 **[3] receives from 3** [3] receives from 4 [3] sends to 2 **[3] sends to 3** [3] sends to 4 [4] receives from 3 **[4] receives from 4** [4] receives from 5 [4] sends to 3 **[4] sends to 4** [4] sends to 5 [5] receives from 4 **[5] receives from 5** [5] receives from 6 [5] sends to 4 **[5] sends to 5** [5] sends to 6 [6] receives from 5 **[6] receives from 6** [6] receives from 7 [6] sends to 5 **[6] sends to 6** [6] sends to 7 [7] receives from 6 **[7] receives from 7** [7] receives from 8 [7] sends to 6 **[7] sends to 7** [7] sends to 8 [8] receives from 7 **[8] receives from 8** [8] sends to 7 **[8] sends to 8**
Resultado	Incorrecto

Descripción de la solución	Hacer que las comunicaciones dentro de un proceso sean configurables como se ve en la sección 5.5. Solo las pruebas del stencil 2d4 se realizaron sin aplicar este cambio.

Tabla 7.1: Prueba unitaria PCN-2d4-1d

PCN-2d4-2d	
Nombre	Comunicaciones en stencil 2d4 con layout 2d
Descripción	Se crea una matriz con halos como los que utiliza un stencil 2d4. Después, se calculan las comunicaciones necesarias. Para cada proceso, se imprime su identificador (entre corchetes) y de qué proceso envía o recibe datos.

Valores esperados (ordenados)	`[0] receives from 1`
	`[0] receives from 3`
	`[0] sends to 1`
	`[0] sends to 3`
	`[1] receives from 0`
	`[1] receives from 2`
	`[1] receives from 4`
	`[1] sends to 0`
	`[1] sends to 2`
	`[1] sends to 4`
	`[2] receives from 1`
	`[2] receives from 5`
	`[2] sends to 1`
	`[2] sends to 5`
	`[3] receives from 0`
	`[3] receives from 4`
	`[3] receives from 6`
	`[3] sends to 0`
	`[3] sends to 4`
	`[3] sends to 6`
	`[4] receives from 1`
	`[4] receives from 3`
	`[4] receives from 5`
	`[4] receives from 7`
	`[4] sends to 1`
	`[4] sends to 3`
	`[4] sends to 5`
	`[4] sends to 7`
	`[5] receives from 2`
	`[5] receives from 4`
	`[5] receives from 8`
	`[5] sends to 2`
	`[5] sends to 4`
	`[5] sends to 8`
	`[6] receives from 3`
	`[6] receives from 7`
	`[6] sends to 3`
	`[6] sends to 7`
	`[7] receives from 4`
	`[7] receives from 6`
	`[7] receives from 8`
	`[7] sends to 4`
	`[7] sends to 6`
	`[7] sends to 8`
	`[8] receives from 5`
	`[8] receives from 7`
	`[8] sends to 5`
	`[8] sends to 7`

Valores obtenidos	**[0] receives from 0** [0] receives from 1 [0] receives from 3 **[0] sends to 0** [0] sends to 1 [0] sends to 3 [1] receives from 0 **[1] receives from 1** [1] receives from 2 [1] receives from 4 [1] sends to 0 **[1] sends to 1** [1] sends to 2 [1] sends to 4 [2] receives from 1 **[2] receives from 2** [2] receives from 5 [2] sends to 1 **[2] sends to 2** [2] sends to 5 [3] receives from 0 **[3] receives from 3** [3] receives from 4 [3] receives from 6 [3] sends to 0 **[3] sends to 3** [3] sends to 4 [3] sends to 6 [4] receives from 1 [4] receives from 3 **[4] receives from 4** [4] receives from 5 [4] receives from 7 [4] sends to 1 [4] sends to 3 **[4] sends to 4** [4] sends to 5 [4] sends to 7 [5] receives from 2 [5] receives from 4 **[5] receives from 5** [5] receives from 8 [5] sends to 2 [5] sends to 4 **[5] sends to 5** [5] sends to 8 [6] receives from 3 **[6] receives from 6** [6] receives from 7 [6] sends to 3 **[6] sends to 6** [6] sends to 7 [7] receives from 4 [7] receives from 6 **[7] receives from 7** [7] receives from 8 [7] sends to 4 [7] sends to 6 **[7] sends to 7** [7] sends to 8 [8] receives from 5 [8] receives from 7 **[8] receives from 8** [8] sends to 5 [8] sends to 7 **[8] sends to 8**
Resultado	Incorrecto
Descripción de la solución	Hacer que las comunicaciones dentro de un proceso sean configurables como se ve en la sección 5.5. Solo las pruebas del stencil 2d4 se realizaron sin aplicar este cambio.

Tabla 7.2: Prueba unitaria PCN-2d4-2d

PCN-2d9Comp-1d	
Nombre	Comunicaciones en stencil 2d9 compacto con layout 1d
Descripción	Se crea una matriz con halos como los que utiliza un stencil 2d9 compacto. Después, se calculan las comunicaciones necesarias. Para cada proceso, se imprime su identificador (entre corchetes) y de qué proceso envía o recibe datos.

Valores esperados (ordenados)	`[0] receives from 1`
	`[0] sends to 1`
	`[1] receives from 0`
	`[1] receives from 2`
	`[1] sends to 0`
	`[1] sends to 2`
	`[2] receives from 1`
	`[2] receives from 3`
	`[2] sends to 1`
	`[2] sends to 3`
	`[3] receives from 2`
	`[3] receives from 4`
	`[3] sends to 2`
	`[3] sends to 4`
	`[4] receives from 3`
	`[4] receives from 5`
	`[4] sends to 3`
	`[4] sends to 5`
	`[5] receives from 4`
	`[5] receives from 6`
	`[5] sends to 4`
	`[5] sends to 6`
	`[6] receives from 5`
	`[6] receives from 7`
	`[6] sends to 5`
	`[6] sends to 7`
	`[7] receives from 6`
	`[7] receives from 8`
	`[7] sends to 6`
	`[7] sends to 8`
	`[8] receives from 7`
	`[8] sends to 7`

Resultado	Correcto

Tabla 7.3: Prueba unitaria PCN-2d9Comp-1d

PCN-2d9Comp-2d	
Nombre	Comunicaciones en stencil 2d9 compacto con layout 2d
Descripción	Se crea una matriz con halos como los que utiliza un stencil 2d9 compacto. Después, se calculan las comunicaciones necesarias. Para cada proceso, se imprime su identificador (entre corchetes) y de qué proceso envía o recibe datos.

Valores esperados (ordenados)	`[0] receives from 1`
	`[0] receives from 3`
	`[0] receives from 4`
	`[0] sends to 1`
	`[0] sends to 3`
	`[0] sends to 4`
	`[1] receives from 0`
	`[1] receives from 2`
	`[1] receives from 3`
	`[1] receives from 4`
	`[1] receives from 5`
	`[1] sends to 0`
	`[1] sends to 2`
	`[1] sends to 3`
	`[1] sends to 4`
	`[1] sends to 5`
	`[2] receives from 1`
	`[2] receives from 4`
	`[2] receives from 5`
	`[2] sends to 1`
	`[2] sends to 4`
	`[2] sends to 5`
	`[3] receives from 0`
	`[3] receives from 1`
	`[3] receives from 4`
	`[3] receives from 6`
	`[3] receives from 7`
	`[3] sends to 0`
	`[3] sends to 1`
	`[3] sends to 4`
	`[3] sends to 6`
	`[3] sends to 7`
	`[4] receives from 0`
	`[4] receives from 1`
	`[4] receives from 2`
	`[4] receives from 3`
	`[4] receives from 5`
	`[4] receives from 6`
	`[4] receives from 7`
	`[4] receives from 8`
	`[4] sends to 0`
	`[4] sends to 1`
	`[4] sends to 2`
	`[4] sends to 3`
	`[4] sends to 5`
	`[4] sends to 6`
	`[4] sends to 7`
	`[4] sends to 8`
	`[5] receives from 1`
	`[5] receives from 2`
	`[5] receives from 4`
	`[5] receives from 7`
	`[5] receives from 8`
	`[5] sends to 1`
	`[5] sends to 2`
	`[5] sends to 4`
	`[5] sends to 7`
	`[5] sends to 8`
	`[6] receives from 3`
	`[6] receives from 4`
	`[6] receives from 7`
	`[6] sends to 3`
	`[6] sends to 4`
	`[6] sends to 7`
	`[7] receives from 3`
	`[7] receives from 4`
	`[7] receives from 5`
	`[7] receives from 6`
	`[7] receives from 8`
	`[7] sends to 3`
	`[7] sends to 4`
	`[7] sends to 5`
	`[7] sends to 6`
	`[7] sends to 8`
	`[8] receives from 4`
	`[8] receives from 5`
	`[8] receives from 7`
	`[8] sends to 4`
	`[8] sends to 5`
	`[8] sends to 7`

Resultado	Correcto

Tabla 7.4: Prueba unitaria PCN-2d9Comp-2d

PCN-2d9NoComp-1d	
Nombre	Comunicaciones en stencil 2d9 no compacto con layout 1d
Descripción	Se crea una matriz con halos como los que utiliza un stencil 2d9 no compacto. Después, se calculan las comunicaciones necesarias. Para cada proceso, se imprime su identificador (entre corchetes) y de qué proceso envía o recibe datos.

Valores esperados (ordenados)	`[0] receives from 1`
	`[0] sends to 1`
	`[1] receives from 0`
	`[1] receives from 2`
	`[1] sends to 0`
	`[1] sends to 2`
	`[2] receives from 1`
	`[2] receives from 3`
	`[2] sends to 1`
	`[2] sends to 3`
	`[3] receives from 2`
	`[3] receives from 4`
	`[3] sends to 2`
	`[3] sends to 4`
	`[4] receives from 3`
	`[4] receives from 5`
	`[4] sends to 3`
	`[4] sends to 5`
	`[5] receives from 4`
	`[5] receives from 6`
	`[5] sends to 4`
	`[5] sends to 6`
	`[6] receives from 5`
	`[6] receives from 7`
	`[6] sends to 5`
	`[6] sends to 7`
	`[7] receives from 6`
	`[7] receives from 8`
	`[7] sends to 6`
	`[7] sends to 8`
	`[8] receives from 7`
	`[8] sends to 7`

Resultado	Correcto

Tabla 7.5: Prueba unitaria PCN-2d9NoComp-1d

PCN-2d9NoComp-2d	
Nombre	Comunicaciones en stencil 2d9 no compacto con layout 2d
Descripción	Se crea una matriz con halos como los que utiliza un stencil 2d9 no compacto. Después, se calculan las comunicaciones necesarias. Para cada proceso, se imprime su identificador (entre corchetes) y de qué proceso envía o recibe datos.
Valores esperados (ordenados)	[0] receives from 1 [0] receives from 3 [0] receives from 4 [0] sends to 1 [0] sends to 3 [0] sends to 4 [1] receives from 0 [1] receives from 2 [1] receives from 3 [1] receives from 4 [1] receives from 5 [1] sends to 0 [1] sends to 2 [1] sends to 3 [1] sends to 4 [1] sends to 5 [2] receives from 1 [2] receives from 4 [2] receives from 5 [2] sends to 1 [2] sends to 4 [2] sends to 5 [3] receives from 0 [3] receives from 1 [3] receives from 4 [3] receives from 6 [3] receives from 7 [3] sends to 0 [3] sends to 1 [3] sends to 4 [3] sends to 6 [3] sends to 7 [4] receives from 0 [4] receives from 1 [4] receives from 2 [4] receives from 3 [4] receives from 5 [4] receives from 6 [4] receives from 7 [4] receives from 8 [4] sends to 0 [4] sends to 1 [4] sends to 2 [4] sends to 3 [4] sends to 5 [4] sends to 6 [4] sends to 7 [4] sends to 8 [5] receives from 1 [5] receives from 2 [5] receives from 4 [5] receives from 7 [5] receives from 8 [5] sends to 1 [5] sends to 2 [5] sends to 4 [5] sends to 7 [5] sends to 8 [6] receives from 3 [6] receives from 4 [6] receives from 7 [6] sends to 3 [6] sends to 4 [6] sends to 7 [7] receives from 3 [7] receives from 4 [7] receives from 5 [7] receives from 6 [7] receives from 8 [7] sends to 3 [7] sends to 4 [7] sends to 5 [7] sends to 6 [7] sends to 8 [8] receives from 4 [8] receives from 5 [8] receives from 7 [8] sends to 4 [8] sends to 5 [8] sends to 7
Resultado	Correcto

Tabla 7.6: Prueba unitaria PCN-2d9NoComp-2d

PCN-2d5NoComp-1d	
Nombre	Comunicaciones en stencil 2d5 no compacto con layout 1d
Descripción	Se crea una matriz con halos como los que utiliza un stencil 2d5 no compacto. Después, se calculan las comunicaciones necesarias. Para cada proceso, se imprime su identificador (entre corchetes) y de qué proceso envía o recibe datos.

Valores esperados (ordenados)	`[0] sends to 1` `[4] sends to 5`
	`[1] receives from 0` `[5] receives from 4`
	`[1] sends to 2` `[5] sends to 6`
	`[2] receives from 1` `[6] receives from 5`
	`[2] sends to 3` `[6] sends to 7`
	`[3] receives from 2` `[7] receives from 6`
	`[3] sends to 4` `[7] sends to 8`
	`[4] receives from 3` `[8] receives from 7`

Resultado	Correcto

Tabla 7.7: Prueba unitaria PCN-2d5NoComp-1d

PCN-2d5NoComp-2d	
Nombre	Comunicaciones en stencil 2d5 no compacto con layout 2d
Descripción	Se crea una matriz con halos como los que utiliza un stencil 2d5 no compacto. Después, se calculan las comunicaciones necesarias. Para cada proceso, se imprime su identificador (entre corchetes) y de qué proceso envía o recibe datos.

Valores esperados (ordenados)	`[0] sends to 1` `[4] sends to 5`
	`[0] sends to 3` `[4] sends to 7`
	`[0] sends to 4` `[4] sends to 8`
	`[1] receives from 0` `[5] receives from 1`
	`[1] sends to 2` `[5] receives from 2`
	`[1] sends to 4` `[5] receives from 4`
	`[1] sends to 5` `[5] sends to 8`
	`[2] receives from 1` `[6] receives from 3`
	`[2] sends to 5` `[6] sends to 7`
	`[3] receives from 0` `[7] receives from 3`
	`[3] sends to 4` `[7] receives from 4`
	`[3] sends to 6` `[7] receives from 6`
	`[3] sends to 7` `[7] sends to 8`
	`[4] receives from 0` `[8] receives from 4`
	`[4] receives from 1` `[8] receives from 5`
	`[4] receives from 3` `[8] receives from 7`

Resultado	Correcto

Tabla 7.8: Prueba unitaria PCN-2d5NoComp-2d

Caja blanca

Las pruebas de caja blanca son aquellas que se desarrollan mirando la implementación interna del código, no solo su interfaz pública. En este caso, los tests se diseñan basándose en la función redistribute, creando casos de prueba para cada una de las comprobaciones que esta realiza, buscando así recorrer todos los flujos posibles del programa estudiando sus casos extremos en busca de fallos.

PCB-redistribute-noInterseccion	
Nombre	Redistribución para dos tiles sin intersección
Descripción	Se intenta calcular las comunicaciones entre dos procesos, cuyos tiles no tienen ninguna intersección.
Valores esperados	`HIT_PATTERN_NULL`
Resultado	Correcto

Tabla 7.9: Prueba unitaria PCB-redistribute-noInterseccion

PCB-redistribute-procInactivo	
Nombre	Redistribución para un proceso inactivo
Descripción	Se elige un número de procesos (13) tal que sea imposible repartir el tile entre ellos, por lo que quedan procesos inactivos. Estos procesos inactivos llamarán a la función redistribute, la cual no debería asignarles ningún patrón de comunicación, ya que no tienen datos que intercambiar.
Valores esperados	`HIT_PATTERN_NULL`
Resultado	Correcto

Tabla 7.10: Prueba unitaria PCB-redistribute-procInactivo

PCB-redistribute-intersecEnvVacia	
Nombre	Redistribución con intersección vacía para el tile de envío
Descripción	Se crea un tile que hace que la intersección utilizada para calcular los envíos sea vacía. Para ello, se ejecuta la prueba con dos procesos, a los que se llamará 0 y 1. Se creará un tile que tendrá halo únicamente arriba. El reparto realizado por Hitmap asignará al proceso 0 la mitad superior del tile y al 1 la inferior. Después, se llamará a la función redistribute en el proceso 0. Esta deber retornar un patrón con un envío (ya que los datos del proceso 0 coinciden con el halo del proceso de debajo, el 1) y ninguna recepción (ya que el halo del proceso 0 no tiene intersección con ningún proceso, ya que no hay ninguno encima).
Valores esperados	`[0] sends to 1`
Resultado	Correcto

Tabla 7.11: Prueba unitaria PCB-redistribute-intersecEnvVacia

PCB-redistribute-intersecRecVacia	
Nombre	Redistribución con intersección vacía para el tile de recepción
Descripción	Se crea un tile que hace que la intersección utilizada para calcular las recepciones sea vacía. Se repite el escenario de la prueba anterior, pero en este caso, es el proceso 1 quien llama a la función redistribute. Esta deber retornar un patrón con una recepción (ya que el halo del proceso 1 tiene intersección con su vecino superior, el proceso 0) y ningún envío (ya que, al no haber un vecino inferior, no hay ningún tile cuyo halo tenga intersección con los datos del proceso 1).
Valores esperados	`[1] receives from 0`
Resultado	Correcto

Tabla 7.12: Prueba unitaria PCB-redistribute-intersecRecVacia

7.1.2 Comparación con referencia secuencial

El objetivo de estas pruebas es comprobar que las funciones incluidas en Hitmap para dar soporte a los tiles distribuidos y la redistribución automática funcionan correctamente. Además,

garantizan que los stencils implementados no tienen ningún error de programación.

Para poder ejecutar este tipo de pruebas es necesario contar con unos resultados de referencia con los que comparar los obtenidos por el programa. En el caso de de los stencil esto no es un problema, ya que, a pesar de que su implementación paralela es compleja, su versión secuencial es sencilla, compuesta únicamente por bucles anidados y algunas comprobaciones para asegurar que ningún iterador se salga de los límites del array. Por tanto, el primer paso para realizar las pruebas de corrección es desarrollar un stencil secuencial equivalente al implementado en Hitmap.

Además, para poder comparar los resultados fácilmente, es necesario que ambas versiones (secuencial y paralela) almacenen su salida en un fichero. Una vez más, en el código secuencial esta es un tarea sencilla que puede ser realizada con una serie de bucles anidados. Sin embargo, en su alternativa paralela, aparece una complicación debido a la concurrencia, que genera problemas al escribir todos los procesos sobre el mismo fichero. Esta situación empeora por el hecho de que los distintos números que componen el resultado deben estar ordenados acorde a su posición en la matriz global. Por suerte, este problema ya está resuelto por Hitmap, que incluye una función denominada hit_tileTextFileWrite. Es necesario destacar que en la versión paralela también hay que tener en cuenta los halos: filas y columnas extras añadidas para almacenar resultados intermedios, pero que deben ser eliminadas de la salida final.

Una vez se dispone del código a probar adaptado para los tests y de su versión de referencia, el siguiente requisito es definir los valores que constituirán las entradas del programa. En concreto, se definen los siguientes valores para cada uno de los parámetros:

- Número de procesos: 1, 4, 5, 9, 10, 13, 16 y 25
- Tamaño de la matriz: 10 y 100 elementos (por dimensión). No se utilizan tamaños más grandes ya que ralentizan la parte secuencial de algunas pruebas y porque el objetivo de estos tests es verificar la corrección, no suponer una prueba de carga.
- Iteraciones: 1, 10 y 100. El caso de una iteración es destacable ya que en él no se producen comunicaciones entre procesos, por lo que ayuda a detectar problemas en la parte secuencial del programa.

Para simplificar la tarea de realizar las pruebas se crea un script de bash encargado de llevarlas acabo. Su funcionamiento es el siguiente: primero comprueba que los ejecutables de ambas versiones del programa han sido generados. Si es así, mediante varios bucles anidados, genera todas las posibles pruebas combinando número de procesos, tamaños e iteraciones. Para cada combinación ejecuta tanto el programa secuencial como el paralelo y comprueba que se haya ejecutado sin errores y generando un fichero de salida. Cuando se cumplen ambas condiciones, invoca el comando diff para verificar que ambos ficheros de salida son iguales. El script mostrará por pantalla para cada prueba realizada si el resultado ha sido correcto (OK) o si se ha detectado un fallo en los resultados.

7.2 Pruebas de rendimiento

El objetivo de estas pruebas es evaluar el rendimiento de los stencils generados con los tiles distribuidos así como su escalabilidad. Estos tests no se limitan a medir el tiempo que tarda

un programa en ejecutarse dada una determinada entrada, sino que buscan comparar diferentes versiones de un mismo stencil para verificar que no ha aparecido en ninguna de ellas un cuello de botella que disminuya el rendimiento. En algunas pruebas las diferencias entre las versiones se limitarán únicamente a las opciones de configuración.

Este tipo de pruebas someten a los stencils a altas cargas de trabajo. Además, para poder probar su escalabilidad con el paralelismo, requieren de un gran número de procesos concurrentes (y por tanto, una mayor cantidad de núcleos en los procesadores). Es por esto que estas pruebas no pueden ser realizadas en un ordenador doméstico. Por ello se utilizará el clúster del grupo Trasgo.

Este clúster dispone de varias máquinas, pero se utilizará principalmente una llamada *manticore*. Esta cuenta con dos CPUs Intel Xeon Scalable Platinum, cada una con 24 cores, que pueden ser 48 haciendo uso de la tecnología Hyperthreading [26]. Esto hace un total de 96 procesos, cifra que nunca se pasará a la hora de ejeuctar las pruebas, ya que crear más procesos de los soportados por el procesador no es eficiente y produce resultados impredecibles al no estar ejecutándose realmente en paralelo.

En aquellas pruebas en las que se quiera probar el comportamiento de los programas con dos máquinas conectadas por red (en lugar de hacer las comunicaciones solo dentro de una misma máquina) se utilizará *medusa*, que dispone de dos CPUs de Intel con 8 cores cada una, o 16 haciendo uso del Hyperthreading. Por tanto, esta máquina puede albergar hasta 32 procesos.

En cuanto al compilador, se utiliza la versión 10.3.0 de gcc haciendo uso de los flags de optimización. Por otro lado, la implementación de MPI utilizada es MPICH en su versión 3.4.2.

Estas pruebas no se ejecutarán una única vez, ya que, a pesar de que la máquina del clúster es un entorno suficientemente estable y estanco (solo se ejecuta una tarea a la vez, por lo que se disponen de todos los recursos de la máquina), pueden aparecer alteraciones debidas a, por ejemplo, variaciones en la temperatura del hardware o en los procesos ejecutados por el sistema operativo. Por tanto, se decide lanzar cada programa 3 veces y calcular su media para minimizar la influencia de estos posibles valores extremos.

Estas ejecuciones no serán consecutivas, sino en series. Es decir, se realizarán tres ejecuciones secuencialmente (cada una comenzará al finalizar la anterior). En cada iteración se ejecutarán una vez todas las combinaciones de stencils, número de procesos y topologías. Esto se hace para que, si se produce una variación en el sistema causada por un factor externo, esta afecte a varios stencils pero en una sola de sus iteraciones, haciendo más fácil detectar el motivo del fallo. Si no se hiciera así, la influencia externa podría detectarse únicamente en uno de los stencils, lo que llevaría a pensar que el error está en el código del mismo y no en un factor no controlado por la prueba.

Para cada una de las ejecuciones se recogen dos datos: el reloj secuencial y el total. El reloj total mide el tiempo de ejecución del programa, mientras que el secuencial se centra solo en aquellas partes dedicadas al cómputo, ignorando el tiempo consumido por las comunicaciones.

Para simplificar la ejecución de los tests se crea un script que ejecuta todas las combinaciones antes mencionadas y recoge los resultados en un archivo csv, lo que permite importar fácilmente los datos en otros softwares para su análisis.

Existe además otro script que facilita la compilación. Su tarea es iterar entre distintos flags del compilador, de forma que el mismo programa pueda ser compilado con distintas opciones.

Además, el script se encarga de mover el ejecutable resultante cambiándole el nombre por uno más identificativo para poder distinguir las distintas versiones del programa.

A continuación se describirán las pruebas que se han realizado a lo largo del proyecto.

7.2.1 Pruebas de comunicaciones dentro del mismo proceso

Como ya se ha visto en los test unitarios, en el stencil 2d4 aparecen comunicaciones no deseadas de todos los procesos con ellos mismos (tabla 7.1). Para confirmar que estos intercambios de datos son realmente problemáticos y afectan a los tiempos de ejecución se diseña esta prueba. En ella se ejecutará el stencil 2d4 junto con un código de referencia, incluido como ejemplo en la biblioteca Hitmap, que implementa el mismo algoritmo pero especificando manualmente las comunicaciones sin hacer uso de los tiles distribuidos, por lo que no incluye la comunicación extra. A la hora de analizar estos resultados, es muy importante tener en cuenta que la versión con tiles distribuidos implementa la optimización secuencial hablada en la sección 6.3.1, por lo que sus tiempos de ejecución deberían ser menores que los de la referencia, que no incluye dicha mejora.

La figura 7.1 muestra los resultados de la prueba. Como se puede comprobar, los tiempos obtenidos triplican a los de la referencia, por lo que es evidente que la comunicación redundante es problemática.

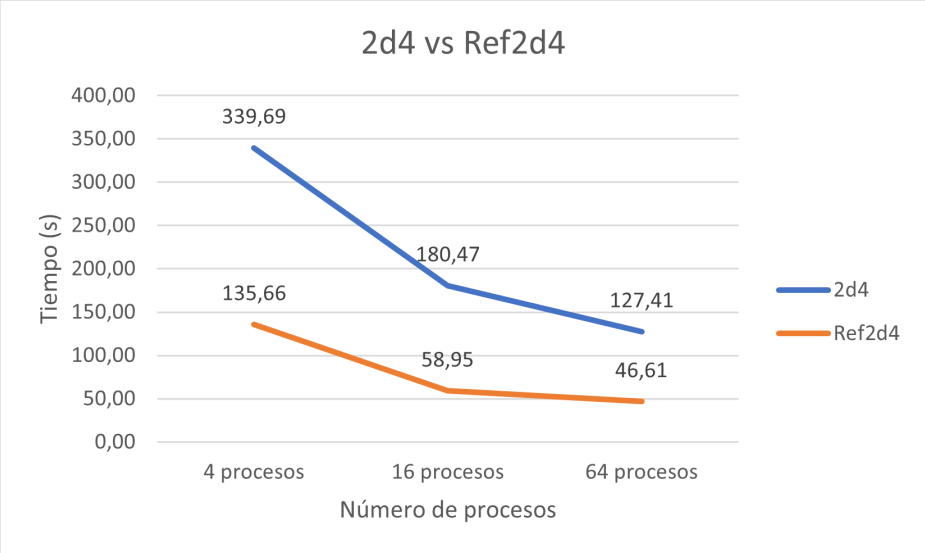


Figura 7.1: Diferencia de tiempo entre el nuevo código y la referencia

Realizando un análisis más detallado de los resultados, desglosando el tiempo total de ejecución y el secuencial (que no incluye las comunicaciones), se hace aún más evidente la magnitud del problema. Como se muestra en la tabla 7.13, las comunicaciones han pasado de tardar, como máximo, 1.5 segundos a llegar hasta los 280 segundos en algunos casos.

		4 procesos				16 procesos				64 procesos			
		Iter. 1	Iter. 2	Iter. 3	Media	Iter. 1	Iter. 2	Iter. 3	Media	Iter. 1	Iter. 2	Iter. 3	Media
2d4	**Tiempo total (s)**	**338,25**	**333,45**	**347,38**	**339,69**	**174,36**	**175,88**	**191,19**	**180,47**	**127,45**	**127,33**	**127,45**	**127,41**
	Tiempo secuencial (s)	65,41	65,25	65,25	65,30	34,39	31,14	39,33	34,96	32,61	32,89	31,98	32,49
	Comunicaciones (s)	272,84	268,20	282,14	274,39	139,96	144,73	151,86	145,52	94,84	94,44	95,47	94,92
Ref2d4	**Tiempo total (s)**	**135,65**	**135,15**	**136,18**	**135,66**	**58,95**	**58,95**	**58,94**	**58,95**	**46,66**	**46,57**	**46,61**	**46,61**
	Tiempo secuencial (s)	134,11	133,60	134,59	134,10	58,49	58,49	58,47	58,49	46,26	45,78	46,20	46,08
	Comunicaciones (s)	1,54	1,55	1,59	1,56	0,46	0,46	0,46	0,46	0,40	0,79	0,40	0,53

Tabla 7.13: Desglose del tiempo de ejecución en cálculo y comunicación

Tras eliminar estas comunicaciones como se explica en la sección 5.5 se repite la misma prueba. Como se puede observar en la tabla 7.14 los tiempos de comunicaciones vuelven a valores normales, por lo que se ha resuelto el problema.

		4 procesos				16 procesos				64 procesos			
		Iter. 1	Iter. 2	Iter. 3	Media	Iter. 1	Iter. 2	Iter. 3	Media	Iter. 1	Iter. 2	Iter. 3	Media
2d4	**Tiempo total (s)**	**68,31**	**68,16**	**68,29**	**68,26**	**34,83**	**34,84**	**34,85**	**34,84**	**26,35**	**26,35**	**26,40**	**26,37**
	Tiempo secuencial (s)	65,16	65,25	65,35	65,25	33,90	33,94	33,80	33,88	25,60	25,60	25,65	25,62
	Comunicaciones (s)	3,15	2,91	2,94	3,00	0,93	0,90	1,05	0,96	0,75	0,75	0,75	0,75
Ref2d4	**Tiempo total (s)**	**135,65**	**135,52**	**135,46**	**135,54**	**59,06**	**59,10**	**59,09**	**59,08**	**46,65**	**46,66**	**46,65**	**46,66**
	Tiempo secuencial (s)	134,02	133,96	133,88	133,95	58,60	58,64	58,63	58,62	46,25	46,27	46,25	46,26
	Comunicaciones (s)	1,63	1,56	1,58	1,59	0,46	0,46	0,46	0,46	0,40	0,40	0,40	0,40

Tabla 7.14: Desglose del tiempo de ejecución en cálculo y comunicación tras la corrección

Además, comparando de nuevo el código con la referencia (figura 7.2), se puede comprobar que esta vez sí se obtiene el resultado esperado, es decir, el nuevo código tiene un tiempo de ejecución menor que el de la referencia (teniendo en cuenta, como se dijo antes, que este primero consta de una optimización secuencial que no tiene el segundo).

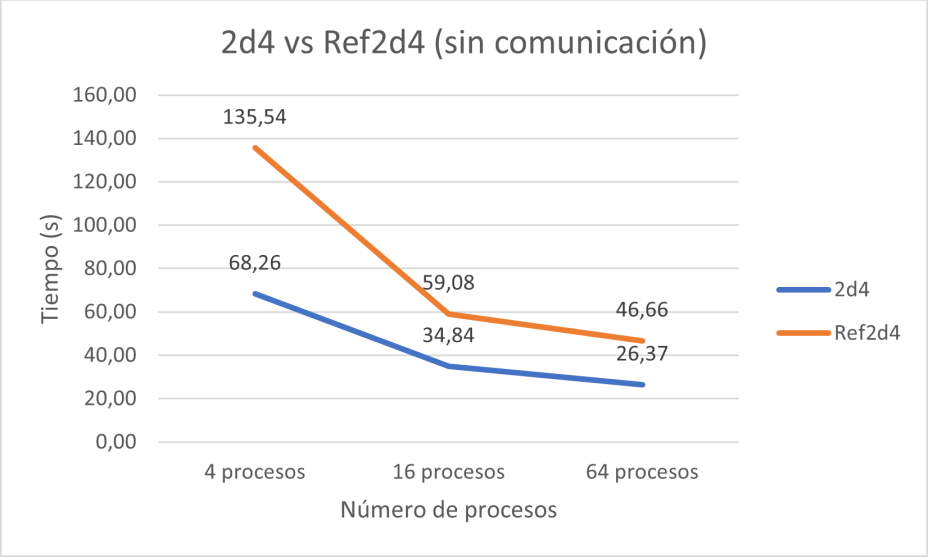


Figura 7.2: Diferencia de tiempo entre el nuevo código y la referencia tras la corrección

7.2.2 Pruebas de comparación de topologías

A la hora de crear un tile distribuido, es posible elegir su layout y topología, es decir, la estructura que forman los procesos y la forma de repartir las estructuras de datos entre estos procesos. En estas pruebas se pretende estudiar el impacto que tienen las diversas topologías en los tiempos de ejecución por lo que, para cada stencil, se comparará su tiempo de ejecución en cada una de las topologáias posibles.

Para ello se utilizan tres topologías: Array1D, Array2D y Array3D. Como su nombre indica, estas topologías distribuyen los procesos generando estructuras de una, dos y tres dimensiones respectivamente. En otras palabras, dan lugar a una red lineal, cuadrada o cúbica. No todas las topologías serán utilizadas en todos los stencils: el reparto en tres dimensiones solo se utilizará con stencils 3D, garantizando así que la distribución de los procesos se adapte a las necesidades del problema.

En cuanto al layout, en todas las pruebas se utilizará el *Blocks*. Como se explicó en la sección 3.2.4, este divide los datos en bandas casi iguales, con un máximo de una fila de diferencia. Es importante recordar que la acción del layout está condicionada al reparto de los procesadores realizado por la topología, es decir, si la topología es 2D el layout repartirá equitativamente las filas y columnas entre todos los procesos. Si la topología es 1D, el layout tan solo repartirá las filas, asignando columnas enteras a cada proceso.

Una vez elaborados los códigos que implementan los stencils con todas las topologías es necesario definir los parámetros y condiciones de la prueba:

- Número de procesos: se elige de tal forma que se adapte perfectamente a las necesidades del reparto realizado por la topología. Es decir, en los stencils 2D (que admiten topologías 1D y 2D) las pruebas se realizarán con 4, 16 y 64 procesos, que son cuadrados perfectos. Por otro lado, los stencils que admiten topologías 3D utilizarán números de raíz cúbica exacta como 8 y 64.

  Es importante destacar que el número de procesos nunca sobrepasa la cantidad de hilos concurrentes que puede manejar la máquina, garantizando así que los cálculos sean realmente paralelos a nivel de hardware. El no hacer esto supondría la obtención de unos datos falsos que no podrían ser utilizados, ya que mostrarían una limitación en la escalabilidad provocada por el hardware y no el software.

- Número de iteraciones: las pruebas se realizarán con un total de 100 iteraciones. Este número es suficientemente alto para generar una carga considerable y mitigar el impacto de la inicialización del programa (donde se consume tiempo preparando las estructuras de datos e inicializando memoria).

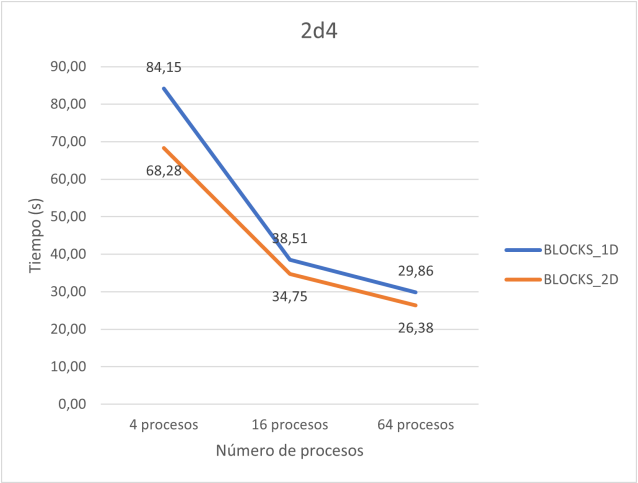
- Tamaño: el tamaño de las matrices variará con el stencil específico, ya que cada uno tiene una carga diferente. El tamaño escogido es aquel que hace que el programa tarde aproximadamente 60 segundos en ejecutarse (con el número de procesos más bajo entre los posibles)

Las figuras 7.3 y 7.4 muestran los tiempos de ejecución de las pruebas en 2D y 3D respectivamente.

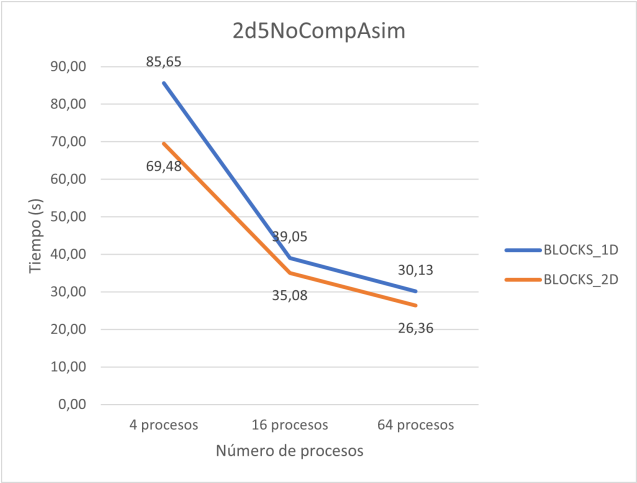
Como se ha mencionado anteriormente, a la hora de analizar los datos recogidos en estas prueba lo importante no son los tiempos de ejecución, sino su evolución en base a las condiciones de la prueba. Se pueden apreciar dos tendencias en estas evoluciones que se analizarán a continuación.

La primera de estas tendencias es la esperable en un programa paralelo: el tiempo de ejecución se reduce al aumentar el número de procesos. Este fenómeno se conoce como escalabilidad. El hecho de que se manifieste en los resultados obtenidos (tanto en 2D como 3D) implica que la implementación de los stencils es la correcta, ya que se ha repartido correctamente la carga entre los procesos y no se ha producido ningún cuello de botella en lugares como, por ejemplo, las comunicaciones.

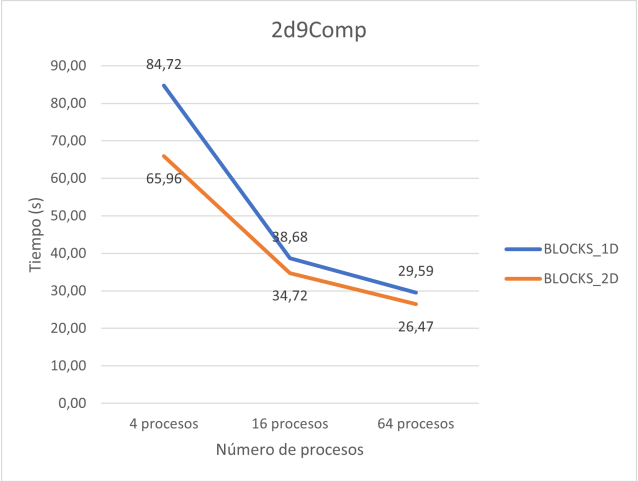
Es también llamativo que la mejora que se produce en los stencils bidimensionales al pasar de 4 a 16 procesos es mayor que la que se produce entre 16 y 64. Esta estabilización en los resultados se debe a que la carga asignada a cada proceso es cada vez más y más pequeña, haciendo que vaya adquiriendo más importancia el tiempo empleado en tareas independientes de la carga.



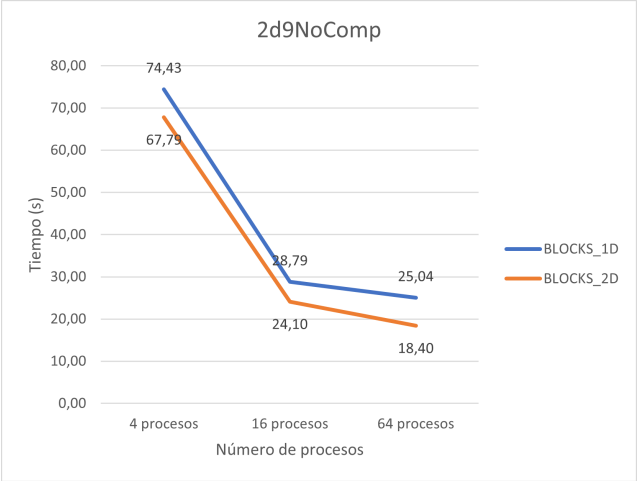
(a) 2d4



(b) 2d5 no compacto asimétrico



(c) 2d9 compacto



(d) 2d9 no compacto

Figura 7.3: Tiempos de ejecución para los stencils 2d

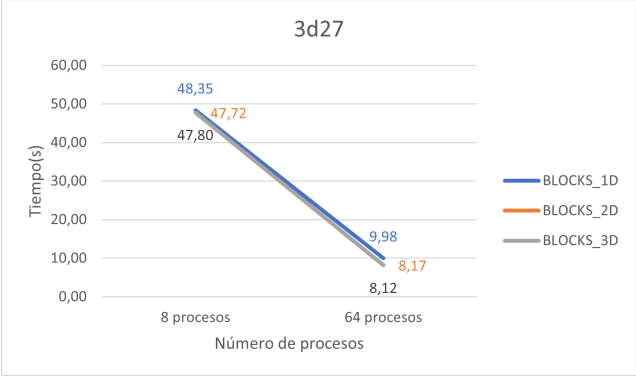
La segunda de las tendencias aporta datos más interesantes e inesperados. Se puede observar que el tiempo de ejecución de los stencils bidimensionales disminuye al utilizar topologías 2D. Como se muestra en la tabla 7.15, el tiempo se reduce entre un 10 y un 30 por ciento aproximadamente.

	2d4			2d5 no compacto asimétrico			2d9 compacto			2d9 no compacto		
	4 proc.	16 proc.	64 proc.	4 proc.	16 proc.	64 proc.	4 proc.	16 proc.	64 proc.	4 proc.	16 proc.	64 proc.
Tiempo 1D (s)	84,15	38,51	29,86	85,65	39,05	30,13	84,72	38,68	29,59	74,43	28,79	25,04
Tiempo 2D (s)	68,28	34,75	26,38	69,48	35,08	26,36	65,96	34,72	26,47	67,79	24,10	18,40
Porcentaje de reducción	18,86%	9,76%	11,65%	18,88%	10,17%	12,51%	22,15%	10,23%	10,55%	8,92%	16,29%	26,50%

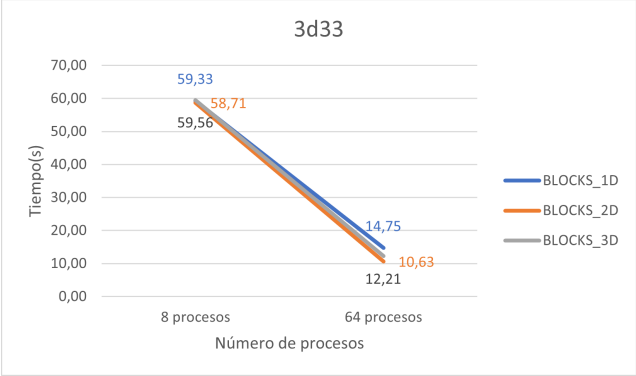
Tabla 7.15: Reducción de tiempo por la topología en stencils 2D (en porcentaje)

En los stencils 3D esta mejora con el cambio de topología desaparece con un número bajo de procesos, en este caso, 8.

En un análisis preliminar, podría concluirse que estas diferencias de comportamiento entre las distintas versiones se deben a que una topología 2D se "adapta" mejor a un stencil bidimensional, reduciendo así los tiempos de ejecución. Sin embargo, un examen del funcionamiento del código del programa y del reparto de los tiles entre procesos desmiente esta idea inicial, ya que la carga asociada a cada proceso es aproximadamente la misma en ambas topologías.



(a) 3d27



(b) 3d33

Figura 7.4: Tiempos de ejecución para los stencils 3d

	8 proc.	64 proc.	8 proc.	64 proc.
Tiempo 1D (s)	48,35	9,98	59,33	14,75
Tiempo 2D (s)	47,72	8,17	58,71	10,63
Tiempo 3D (s)	47,80	8,12	59,56	12,21
Porcentaje de reducción 2D	1,31%	18,09%	1,04%	27,91%
Porcentaje de reducción 3D	1,15%	18,61%	-0,39%	17,21%

Tabla 7.16: Reducción de tiempo por la topología en stencils 3D (en porcentaje)

Para comprender que no existe una diferencia de carga se puede utilizar el siguiente ejemplo: un tile de 20x20 elementos repartido entre 4 procesos. Si se usa una topología 1D, la división solo se realizará en una dimensión, por tanto, a cada proceso le corresponderían 5 filas de 20 elementos cada una (un total de 100 elementos). En cambio, en una topología bidimensional, el reparto se hace en dos dimensiones, por lo que la matriz se dividiría en cuadrados de 10x10 elementos, de nuevo haciendo un total de 100 datos a procesar. Esto descarta la idea de que la diferencia de tiempo pueda deberse a un reparto de la carga distinto en ambas versiones. Aunque este razonamiento se haya hecho solo para stencils bidimensionales, es sencillo extrapolarlo a cualquier número de dimensiones.

La segunda hipótesis que se planteó para explicar este comportamiento fue que la diferencia se encontrara en las comunicaciones, ya que cada una de las topologías genera diferentes patrones de comunicación.

En una topología lineal (1D) cada proceso puede comunicarse como máximo con dos vecinos, mientras que en una 2D pueden producirse hasta cuatro comunicaciones (ignorando diagonales): arriba, abajo, izquierda y derecha. Pero aunque aumente el número de comunicaciones, la cantidad de datos intercambiada es similar. Volviendo al ejemplo anterior de la matriz de 20x20 elementos, en la topología 1D cada proceso tendría que enviar al vecino de arriba (o abajo) una fila entera (la primera o última), es decir, 20 elementos. Esto hace un máximo de 40 elementos intercambiados por un proceso. En la topología 2D los bordes del cuadrado tienen 10 elementos por lo que, suponiendo que un proceso se encuentre en el centro de una cuadrícula y por tanto

pueda intercambiar datos con 4 vecinos, el máximo de datos intercambiados es 40 de nuevo.

Es cierto que el hecho de que el número de elementos sea el mismo en ambos patrones de comunicación no implica necesariamente que los tiempos de ejecución sean iguales. En concreto, podría ser más eficiente hacer menos comunicaciones pero con más datos, ya que es necesario llevar a cabo menos tareas de inicialización y control de las comunicaciones. A pesar de esto, esta segunda hipótesis también es descartada ya que no puede ser la causante de diferencias tan grandes de tiempo. Como se explicó al principio de esta sección, en las pruebas se recogen tanto el reloj secuencial (tiempo que tarda el programa en hacer cálculos, excluyendo inicializaciones y comunicaciones) y reloj total (tiempo total de ejecución). Si se analiza las diferencias de tiempo entre topologías, estas se encuentran en el tiempo secuencial, por tanto las comunicaciones no han podido tener ninguna influencia. Además, el tiempo invertido en el intercambio de datos es prácticamente despreciable en comparación con el tiempo empleado en realizar cálculos, sobre todo en este entorno en el que todos los procesos están en la misma máquina y no es necesario realizar costosas comunicaciones de red.

La última hipótesis contemplada es que esta diferencia de tiempos se deba al comportamiento de las cachés en cada una de las topologías. Esta hipótesis será evaluada (y confirmada) en la siguiente sección.

7.2.3 Pruebas de memorias caché

Como se explica en la sección anterior, el motivo por el que una carga de trabajo prácticamente idéntica pueda tener un tiempo de ejecución hasta un 20% más alto puede encontrarse en el patrón de acceso a datos y cómo éste aprovecha las memorias caché.

Para explicar esta hipótesis se recurre de nuevo al ejemplo de un tile de 20x20 elementos. Estos son divididos en 4 rectángulos de 5x20 en una topología 1D y en 4 cuadrados de 10x10 en una topología 2D. En esta ocasión se añade al escenario una caché de 32 elementos por proceso. Para simplificar, se supone que el stencil es el único código que está siendo ejecutado y que por tanto ningún elemento externo influye en el uso de la caché, y que se leen datos individuales de memoria y no líneas de caché. Además se utilizará un stencil 2d4, es decir, que solo accede a 4 vecinos para evitar un elevado número de lecturas de memoria.

Durante el cálculo del stencil la matriz se recorrerá por filas, avanzando posteriormente por las columnas dentro de cada fila. Estas iteraciones comenzarán en el segundo elemento, ya que el primero no tiene vecino anterior para hacer el cálculo, y terminarán en el penúltimo por el mismo motivo.

En el caso de la topología en dos dimensiones, cada una de las filas medirá 10 elementos. La figura 7.5 muestra los siguientes comportamientos:

1. Para calcular el primer elemento (1,1) se accede al propio elemento y sus vecinos superior (0,1), inferior (2,1), izquierdo (1,0) y derecho (1,2). Todos estos accesos suponen fallos de caché (figura 7.5a).

2. En el siguiente elemento (1,2) se accederá a los datos siguiendo el mismo patrón, pero en este caso el vecino izquierdo (1,1) así como el propio elemento ya están presentes en la caché por la iteración anterior, por tanto se producen solo 3 fallos de caché, en lugar de los cinco de la primera iteración (figura 7.5b).

3. Al acabar de recorrer la primera fila, la caché tendrá almacenados 26 elementos (8 de las filas superior e inferior y los 10 de la fila que se está recorriendo)

4. Al empezar a recorrer la segunda fila la caché no estará vacía. En concreto, aún se mantienen en ella la segunda fila (que antes era la fila que estaba siendo recorrida pero ahora es la fila de los vecinos superiores) y la segunda (que ahora es la fila que se recorre, pero antes fue la de los vecinos inferiores). Esto hace que en el primer elemento de la fila solo haya dos fallos de caché: el vecino inferior (la fila 3 aún no se ha recorrido) y el primer elemento de la fila (como los cálculos empiezan en el segundo elemento de la fila para no salirse de la matriz, los vecinos inferiores de la iteración anterior empezaron también en el segundo elemento, por lo que nunca se leyó el primero) Esto se muestra en la figura7.5c.

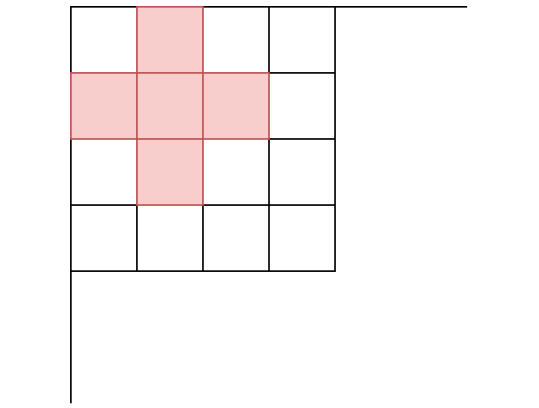
5. Como ocurrió con la primera fila, a partir de su segundo elemento los vecinos izquierdos dejan de ser fallos de caché, por lo que 4 de las 5 lecturas (80%) son respondidas por la caché y por tanto mucho más rápidas (figura 7.5d).

6. Cuando se lea el séptimo elemento de la fila, la lectura del vecino inferior supondrá un fallo de caché como en los casos anteriores. Sin embargo, en esta ocasión la caché no tendrá más espacio, por lo que tendrá que desalojar un elemento. Suponiendo un patrón de desalojo LRU (leído menos recientemente) [27], se eliminará un elemento de la fila 0, que ya no va a ser utilizada de nuevo, por lo que no evitará futuros aciertos de caché. Este comportamiento se repetirá cíclicamente durante toda la ejecución, dando lugar a un uso de la caché muy eficiente y por tanto rápido.

Al pasar de una topología 2D a 1D las filas son más largas (la división se hace en filas, no en cuadrados), por lo que el patrón de accesos a memoria y el uso de caché cambia.

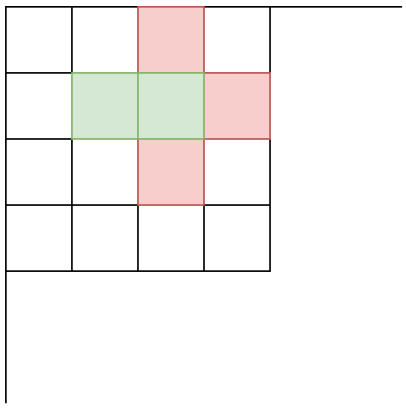
Las iteraciones correspondientes a la primera fila serían iguales al ejemplo 2D, por lo que lo visto en las figuras 7.5a y 7.5b sigue siendo válido. Sin embargo, al duplicar la longitud de las filas, el número de elementos leídos (y por tanto guardados en caché) al finalizar el recorrido de la primera fila es mayor, exactamente 56, lo que excede el tamaño de la caché (32) provocando desalojos. De nuevo suponiendo una política LRU, los elementos que se sobrescribirían de la caché serían los correspondientes a las primeras columnas, que son las primeras que fueron leídas. Esto hará que al empezar a recorrer la segunda fila ya no se produzcan aciertos de caché en el vecino superior y los elementos de la propia fila como en las figuras 7.5c y 7.5d, por lo que se repetirá el patrón de fallos de caché de la primera fila (2 aciertos en cada elemento) frente a los 4 aciertos que se lograban con la topología 2D, aumentando así el tiempo invertido en accesos a memoria.

Lo explicado hasta ahora en esta sección es una hipótesis que, si bien tiene fundamentos teóricos, se basa en muchas suposiciones (tamaños de tiles y cachés, uso exclusivo del sistema, lectura de elementos individuales en lugar de líneas de caché, etc.). Por tanto, para poder comprobar que esto es lo que está ocurriendo realmente es necesario diseñar un nuevo conjunto de pruebas.



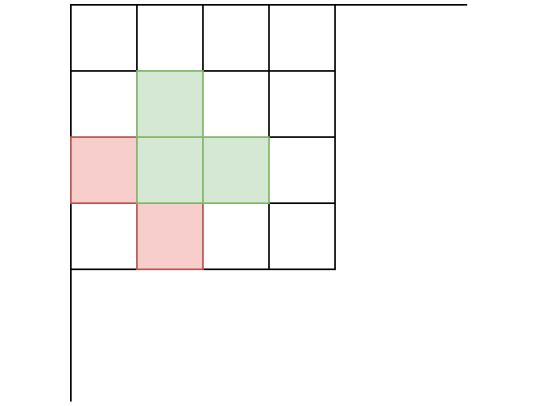
0,1	1,0	1,1	1,2	2,1			

(a) Iteración 1



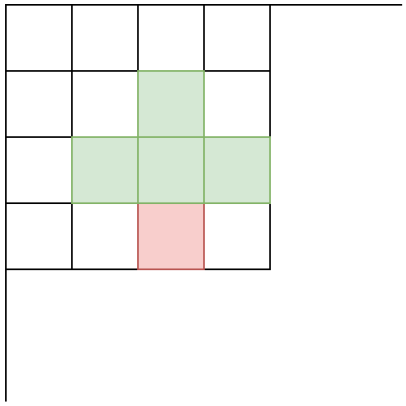
0,1	1,0	1,1	1,2	2,1	0,2	1,3	2,2

(b) Iteración 2



0,1	1,0	1,1	1,2	2,1	0,2	1,3	2,2
0,3	1,4	2,3	0,4	1,5	2,4	0,5	1,6
2,5	0,6	1,7	2,6	0,7	1,8	2,7	0,8
1,9	2,8	2,0	3,1				

(c) Iteración 1 de la segunda fila



0,1	1,0	1,1	1,2	2,1	0,2	1,3	2,2
0,3	1,4	2,3	0,4	1,5	2,4	0,5	1,6
2,5	0,6	1,7	2,6	0,7	1,8	2,7	0,8
1,9	2,8	2,0	3,1	3,2			

(d) Iteración 2 de la segunda fila

Figura 7.5: Comportamiento de la caché en topologías 2D

La prueba se realizará con el stencil 2d9 compacto, ejecutado con 4 procesos. Sus dos versiones (topología 1D y 2D) serán sometidas a dos tipos de carga diferente:

- Matriz de 30000 elementos y 100 iteraciones: estas condiciones son las mismas que en la prueba de topologías (sección 7.2.2), por lo que estos resultados se usarán como datos de control o referencia.
- Matriz de 50000 elementos y 20 iteraciones: la disminución del número de iteraciones se hace por conveniencia para evitar que los tiempos del test se disparen, pero no influyen en el

experimento, ya que solo se van a comparar entre sí las distintas topologías para una misma carga de trabajo. En cuanto al tamaño, es mayor que en la prueba anterior, lo que hará que las filas asignadas a cada proceso (incluso en la topología 2D) sean más largas, provocando así desalojos de caché.

Si la hipótesis es correcta, en la segunda prueba, que usa matrices más grandes, las cachés no serán capaces de almacenar suficientes elementos como para que se pueda producir la reutilización de datos entre iteraciones. Esto se debería traducir en que la proporción de fallos de caché sea la misma en ambas topologías.

Tras ejecutar la prueba, como se puede ver en la figura 7.6 la hipótesis se confirma. Con una matriz suficientemente pequeña (30000 elementos) el cambio de topología (y por tanto de patrón de uso de la caché) se traduce en una bajada de 85 segundos a 66 (un 38% más rápido). En cambio, con la matriz más grande, la caché no tiene espacio suficiente, por lo que los tiempos son prácticamente idénticos entre ambas topologías: 52,54 segundos en 1D y 52.03 en 2D.

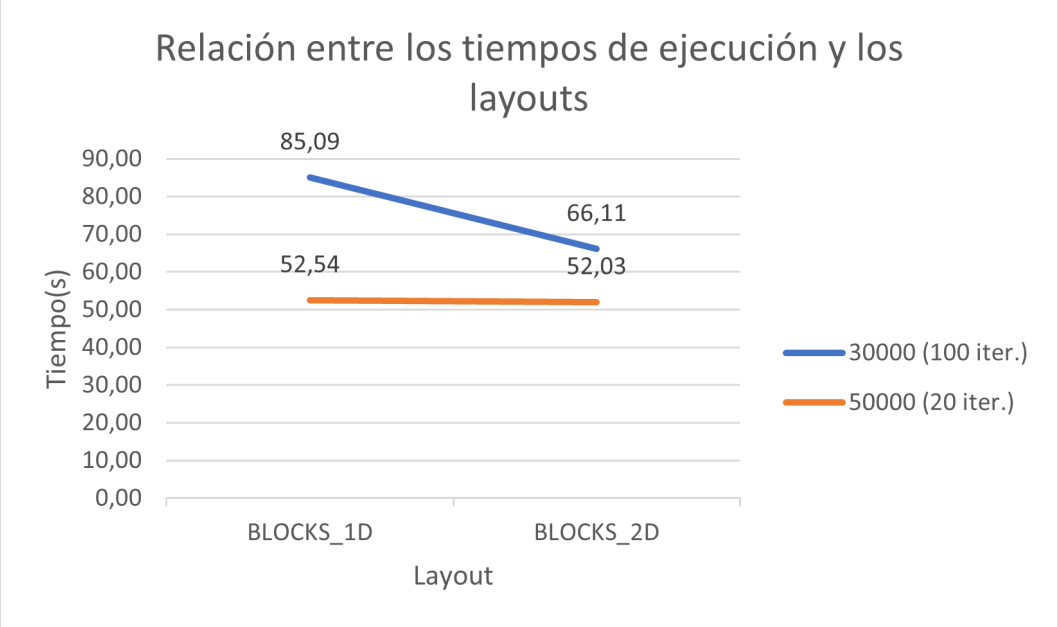


Figura 7.6: Efecto de las cachés en un stencil 2D en función de la topología

La confirmación de esta hipótesis plantea un problema: las cachés alteran notablemente los tiempos de ejecución lo cuál complica la extracción de conclusiones de los experimentos. Para poner solución a este problema se debe cambiar la forma en la que se accede a los datos, garantizando que las cachés se comporten de forma similar independientemente de la topología y del tamaño de la matriz. Para ello se utiliza tiling en varios niveles, reduciendo el tamaño de los tiles utilizados para el cómputo de forma que el comportamiento de la memoria sea el mismo independientemente del tamaño, como se explica en la sección 6.3.2

7.2.4 Pruebas de reducción del tamaño del tile

Para confirmar que los problemas mencionados en la sección anterior se han resuelto se diseña una nueva prueba. En este caso se ejecutará una versión del stencil 2d4 en la cual la matriz asignada a cada proceso a sido subdividida en tiles más pequeños de tamaño fijo (32 elementos). El objetivo de esta prueba es comprobar que este nuevo patrón de acceso a los datos (en tiles

más pequeños) uniformiza los comportamientos de la memoria caché, haciendo que los tiempos no varíen entre la topología 1D y la 2D.

La figura 7.7 muestra los resultados de la prueba. Los tiempos de ejecución entre ambas versiones son muy similares, por lo que se puede confirmar que, al haber cambiado la estructura de los bucles del programa, se ha logrado que ambos layouts se comporten de forma prácticamente idéntica.

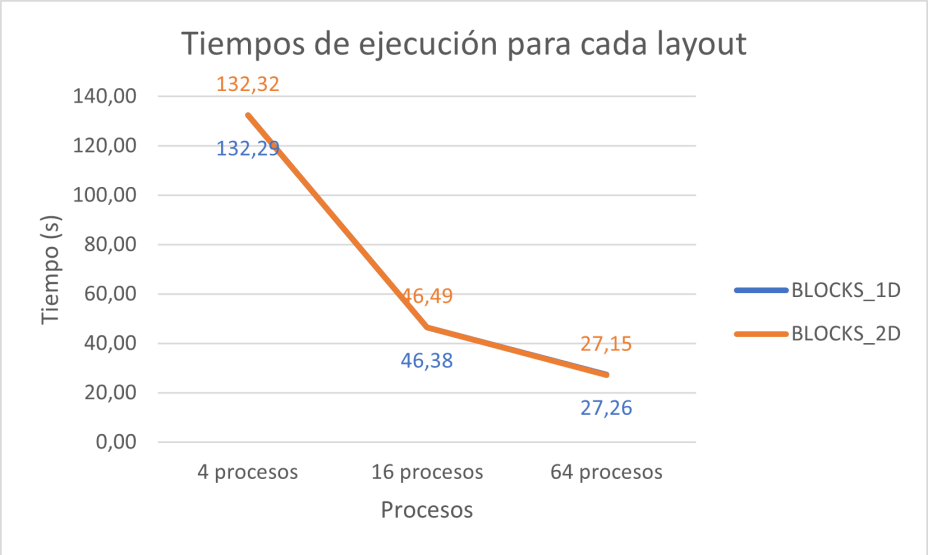


Figura 7.7: Tiempo de ejecución del stencil 2d4 con layout 1D y 2D

7.3 Comparación con códigos de referencia

La prueba más importante que se debe hacer es comprobar que los tiempos de ejecución no han variado entre la versión del stencil de la que se disponía antes de empezar el proyecto y la que utiliza todas las características de los tiles distribuidos. De hecho, los tiempos de ejecución deberían ser prácticamente idénticos entre las dos versiones, ya que su funcionamiento subyacente es exactamente el mismo (el objetivo de la función redistribute y los tiles distribuidos es crear el mismo patrón de comunicaciones que se creaba a mano anteriormente o incluso más optimizado).

A la hora de preparar esta prueba surge un inconveniente: no es posible comparar la versión original del stencil con la versión que se ha elaborado utilizando tiles distribuidos, ya que su código secuencial no es el mismo. Concretamente, la versión original utilizaba dos matrices, una de lectura (copia) y otra para almacenar los resultados. Antes de cada iteración, los datos de la matriz principal tenían que ser copiados a la matriz "copia". En cambio, en la nueva versión distribuida se añade una optimización secuencial (sección 6.3.1), ya que, en lugar de copiar los datos, se intercambian los punteros de las dos matrices, lo cual da lugar a los mismos resultados pero en un tiempo menor. Esta optimización produciría importantes diferencias de tiempo entre ambas versiones, lo cual puede enmascarar una mejora o empeoramiento en la versión con tiles distribuidos.

Por tanto, el primer paso es crear un código de referencia que use todas las nuevas características de los tiles distribuidos pero que elimine la optimización secuencial, es decir, que realice las copias entre ambas matrices en cada iteración. A lo largo de las siguientes pruebas, este código será

denominado Ref2d4Dis (código de referencia del stencil 2d4 con tiles distribuidos). Al código original, en el que el patrón de comunicación se crea manualmente se le llamará Ref2d4.

La prueba se realizará en las mismas condiciones que en la vista en la sección 7.2.2: matrices de 30000 elementos, 100 iteraciones y tres ejecuciones con 4, 16 y 64 procesos. También se calculará la media de tres iteraciones para mitigar la posible influencia de factores externos. En este caso no existirán varias topologías, el código de referencia distribuido (Ref2d4Dis) ha sido creado con una topología 2D para imitar al código original.

La figura 7.8 muestra los resultados de la prueba. Analizándolos se puede observar un resultado inesperado. Hay diferencias notables de tiempo entre dos versiones que deberían ser iguales. Además, esta diferencia no es constante, si no que fluctúa: con 4 procesos es más rápido el código con tiles distribuidos, pero con 16 y 64 el comportamiento se invierte.

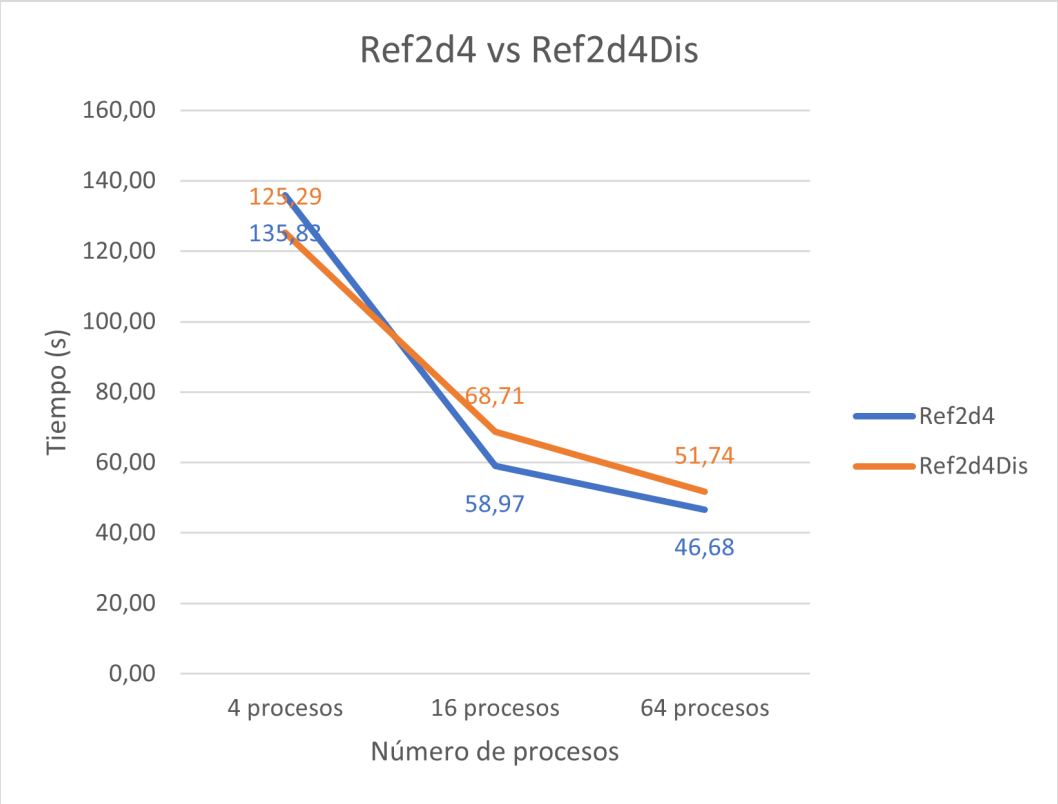


Figura 7.8: Diferencia de tiempos entre el código original y su versión con tiles distribuidos

Antes de empezar a buscar motivos de esta diferencia de tiempo en el código interno de Hitmap se lleva a cabo un último análisis en las referencias, buscando diferencias en su implementación que hubieran pasado desapercibidas hasta el momento. En este proceso se encuentra una diferencia: la forma en la que se trasladan los datos al tile de "copia".

En la referencia Ref2d4Dis las copias de la matriz se llevan a cabo utilizando el método más sencillo, dos bucles for anidados (código 7.1). Sin embargo, el código original utiliza un mecanismo más sofisticado (y transparente para el programador). En esta implementación el tile de copia se crea como un subconjunto del tile principal (aunque tenga el mismo número de elementos y su propia memoria asignada). Esto permite llamar a la función de Hitmap *hit_tileUpdateFromAncestor* que copia los datos del tile padre (matriz principal) al hijo (matriz de copia), como se ve en el código 7.2. En su implementación la función *hit_tileUpdateFromAncestor* utiliza un *memcpy* en lugar de los bucles anidados empleados en la otra referencia, lo cual puede

explicar la diferencia de comportamiento con la otra versión en base al número de procesos (y por tanto tamaño de los datos a copiar).

```
for ( i=0; i<hit_tileDimCard( tileMat , 0 ); i++ )
    for ( j=0; j<hit_tileDimCard( tileMat , 1 ); j++ )
        hit_tileElemAt(tileCopy , 2, i , j) = hit_tileElemAt(tileMat , 2,
            i , j);
```

Código (7.1) Copia de matrices con bucles anidados

```
//Inicializacion
hit_tileSelect( &tileCopy , &tileMat , HIT_SHAPE_WHOLE );

//Bucle de computo
hit_tileUpdateFromAncestor( &tileCopy );
```

Código (7.2) Actualización de matriz con los valores del padre

Figura 7.9: Versiones de la copia entre matrices

Para confirmar que este es el problema causante de la diferencia de tiempos (y para solucionarlo en caso de que así sea) se crea una nueva referencia a la que se llamará RefFor2d4 (referencia que utiliza bucles for para la copia) frente a la referencia anterior, que se renombra a RefAncestor2d4 (ya que utiliza la función *hit_ tileUpdateFromAncestor*).

De nuevo se realiza la prueba mencionada al principio de esta sección, con los mismos parámetros pero con las tres referencias (distribuida, copia con for y actualización desde el ancestro). La figura 7.10 muestra los resultados obtenidos.

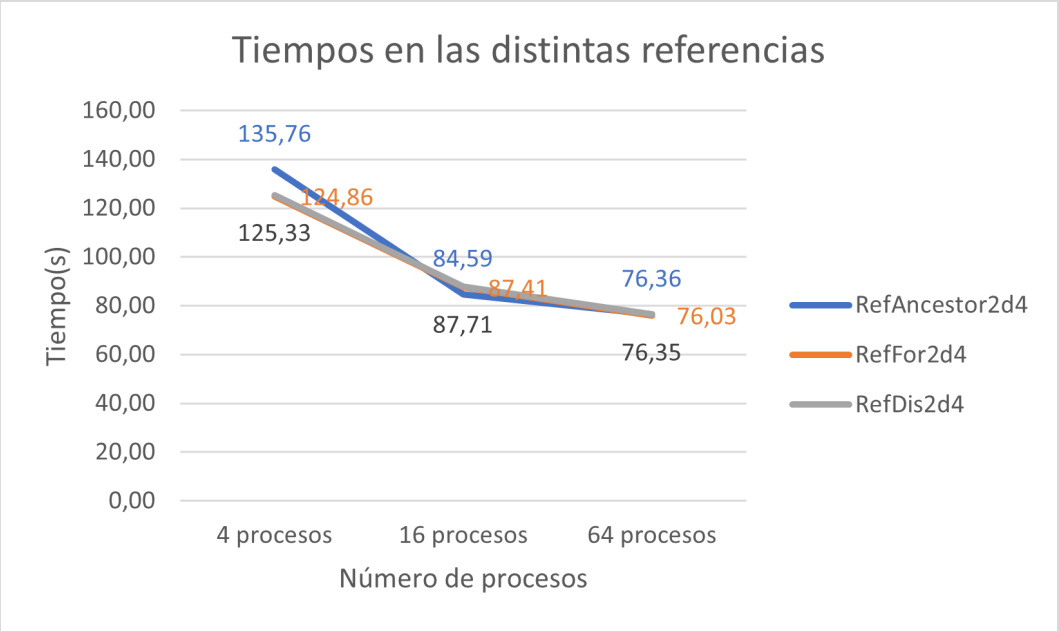


Figura 7.10: Diferencia de tiempos entre el código original (con las dos versiones de copia) y el distribuido

Como se puede comprobar la versión distribuida y la referencia que utiliza los bucles for para la copia tienen tiempos prácticamente idénticos, lo que confirma que esa sección del código era la causante de la diferencia de resultados observada.

Además, el haber logrado tiempos tan parejos entre una referencia y la implementación con tiles distribuidos confirma que la nueva versión del stencil no ha añadido ningún cuello de botella ni incrementado el tiempo de ejecución.

7.3.1 Comparación con MPI en máquinas conectadas por red

En esta prueba no solo se comparará la implementación de un stencil en Hitmap con tiles distribuidos y la que usa patrones de comunicación creados a mano, sino que además se añadirá una versión implementada en MPI. El objetivo es evaluar tanto el rendimiento como la complejidad del código necesario para implementar un stencil.

Comparación de tiempos de ejecución

El objetivo de esta prueba es comprobar que, a pesar de las abstracciones introducidas, el código desarrollado tiene un rendimiento similar a aquel optimizado manualmente.

Para que estas pruebas sean más representativas, se ejecutarán en dos máquinas simultáneamente, haciendo que se produzcan comunicaciones reales entre nodos. Por tanto, los tiempos de ejecución serán más variables debido a la inestabilidad de la red.

Esta prueba se ejecutará en un entorno heterogéneo (como los explicados en la sección 1.1.2), por lo que es necesario cambiar los parámetros de entrada con respecto a otras pruebas para compensar las diferencias de rendimiento. En este caso, el tamaño de las matrices será de 10000 elementos, más pequeño que en pruebas pasadas, y el número de ejecuciones, 1000, será mayor. Con esto se pretende conseguir que las iteraciones sean más cortas, de forma que no surjan diferencias de tiempo de ejecución significativas entre los procesos al final de cada iteración, lo que se traduciría en elevados tiempos de espera por parte de los procesos rápidos debido a que tendrían que esperar a que la máquina lenta finalice su trabajo.

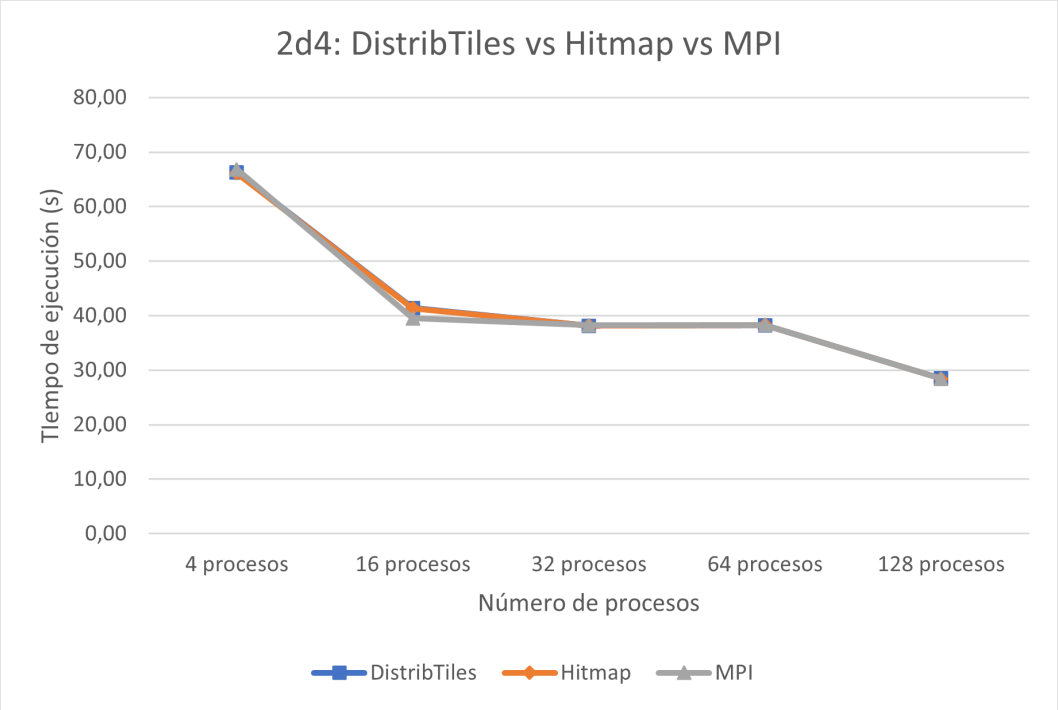
Las pruebas se lanzarán con 4, 16, 32, 64 y 128 procesos, evitando así superar el máximo de procesos concurrentes soportado por ambas máquinas.

De nuevo se repite el procedimiento de ejecutar varias veces la misma prueba y hacer la media para mitigar la influencia de factores externos. Sin embargo, al ejecutarse esta prueba en red, lo que se traduce en mayor variabilidad, se realizarán 5 iteraciones, frente a las 3 que se hacían anteriormente.

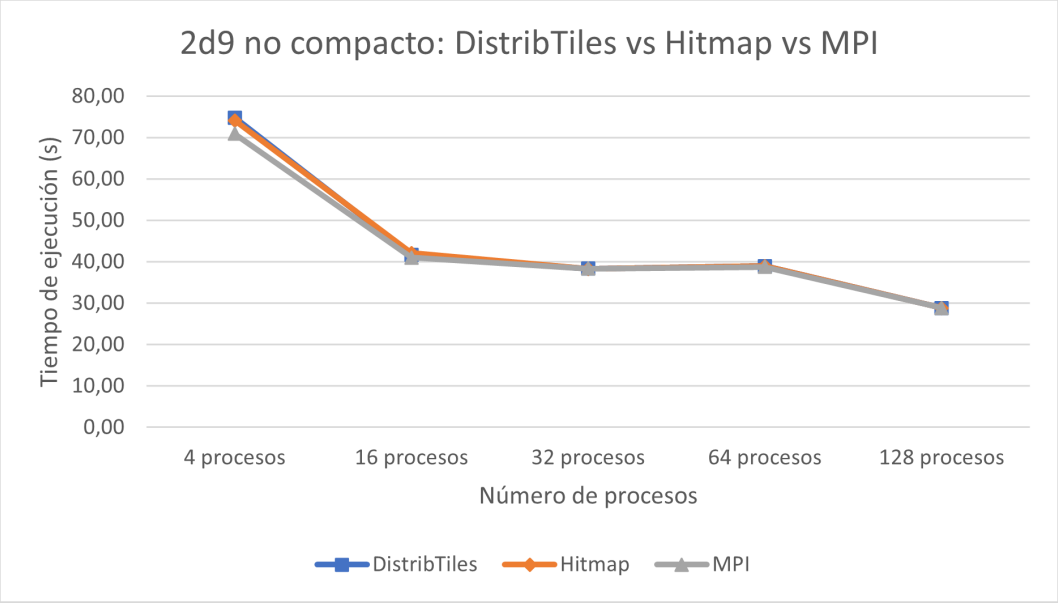
Estas pruebas no se realizan sobre todos los stencils, ya que elaborar códigos de referencia en MPI es una tarea muy costosa. Por ello se utilizan el stencil 2d4 y el 2d9 no compacto, que presentan diferencias entre ellos como el número de operandos o el tamaño de sus halos.

Para realizar estas pruebas ha sido necesario hacer pequeñas modificaciones en los códigos para asegurar que todos tienen una carga de trabajo prácticamente idéntica. Para ello, se ha eliminado la optimización en las copias mencionada en la sección 6.3.1 ya que esta no está implementada en la versión en MPI. Por otro lado, se han eliminado todas las operaciones de escritura de resultados en ficheros, ya que en esta prueba solo se evalúan los tiempos de ejecución, no la corrección.

La figura 7.11 muestra los tiempos de ejecución del stencil 2d4 (7.11a) y el 2d9 no compacto (7.11b) en sus versiones implementadas con MPI, Hitmap y Hitmap con tiles distribuidos. Las tablas 7.17 y 7.18 muestran la media de los tiempos de ejecución, ya que no pueden ser añadidos al gráfico debido a la proximidad de las líneas.



(a) Tiempos de ejecución del stencil 2d4 para MPI, Hitmap y DistribTiles



(b) Tiempos de ejecución del stencil 2d9 no compacto para MPI, Hitmap y DistribTiles

Figura 7.11: Comparación de la implementación con MPI, Hitmap y DistribTiles de un stencil

	4 procesos	16 procesos	32 procesos	64 procesos	128 procesos
DistribTiles	66,23	41,43	38,10	38,24	28,47
Hitmap	66,14	41,34	38,16	38,28	28,45
MPI	66,81	39,52	38,23	38,27	28,42

Tabla 7.17: Tiempos de ejecución del gráfico 7.11a

Antes de empezar a analizar estos datos, es necesario calcular también la desviación estándar que presentan, ya que en esta ocasión existe una mayor variabilidad en los tiempos que en pruebas anteriores al utilizar dos máquinas conectadas por red. Se usará la desviación estándar relativa

	4 procesos	16 procesos	32 procesos	64 procesos	128 procesos
DIstribTiles	74,81	41,72	38,35	38,97	28,82
Hitmap	74,12	42,11	38,29	38,94	28,83
MPI	70,91	40,99	38,33	38,78	28,86

Tabla 7.18: Tiempos de ejecución del gráfico 7.11b

(en tanto por ciento) de forma que se puedan comparar los valores para diferentes números de procesos. Es decir, al dividir la desviación estándar por la media, se consigue que los valores no dependan del tiempo total de ejecución, que disminuye al aumentar los procesos.

Las tablas 7.19 (2d4) y 7.20 (2d9 no compacto) muestran las desviaciones estándar en porcentaje de ambas ejecuciones. Como se puede comprobar, el valor máximo es 2.01% para el stencil 2d4 y 1.14 % para el 2d9 no compacto. El resto de valores son menores de un 1 %, por lo que se puede aceptar estos datos y continuar con su análisis.

	4 procesos	16 procesos	32 procesos	64 procesos	128 procesos
DistribTiles	0,30%	0,82%	0,12%	0,31%	0,27%
Hitmap	0,61%	0,40%	0,20%	0,27%	0,38%
MPI	0,69%	2,01%	0,11%	0,21%	0,37%

Tabla 7.19: Desviación estándar relativa (porcentaje) de los tiempos de ejecución del stencil 2d4

	4 procesos	16 procesos	32 procesos	64 procesos	128 procesos
DistribTiles	0,58%	0,63%	0,03%	0,10%	0,04%
Hitmap	1,08%	0,43%	0,05%	0,12%	0,12%
MPI	1,14%	0,92%	0,10%	0,13%	0,14%

Tabla 7.20: Desviación estándar relativa (porcentaje) de los tiempos de ejecución del stencil 2d9 no compacto

Para comprobar si la inclusión de los tiles distribuidos en Hitmap ha provocado la aparición de algún cuello de botella, se calcula la diferencia de tiempo de las dos versiones de Hitmap (con y sin tiles distribuidos) con respecto a la implementación en MPI. Las tablas 7.21 y 7.22 muestran el resultado de restar el tiempo de ejecución de Hitmap menos el de MPI (es decir, valores negativos implican mejoría respecto a la referencia).

	4 procesos	16 procesos	32 procesos	64 procesos	128 procesos
Hitmap	-0,67	1,81	-0,07	0,01	0,03
DistribTiles	-0,57	1,90	-0,13	-0,03	0,05

Tabla 7.21: Diferencia de tiempo del stencil 2d4 respecto a su implementación en MPI

	4 procesos	16 procesos	32 procesos	64 procesos	128 procesos
Hitmap	3,20	1,13	-0,04	0,17	-0,03
DistribTiles	3,90	0,74	0,02	0,20	-0,04

Tabla 7.22: Diferencia de tiempo del stencil 2d9 no compacto respecto a su implementación en MPI

En el stencil 2d4, las diferencias son despreciables con la excepción de la ejecución con 16 procesos, donde el tiempo ha aumentado casi 2 segundos respecto a la versión de MPI. Sin embargo, el tiempo de ejecución de las dos versiones de Hitmap es muy similar en ese caso (9 centésimas de diferencia), por lo que la causa de esa diferencia de tiempo no se encuentra en la nueva extensión desarrollada, sino que estaba ya en la biblioteca antes de iniciar el proyecto.

Por otro lado, el stencil 2d9 no compacto presenta unas diferencias mayores en sus ejecuciones con 4 y 16 procesos (casi 4 y 2 segundos respectivamente). Aunque estas diferencias sean significativas, hay que destacar que se dan en las dos ejecuciones con mayor desviación estándar relativa de la prueba (rondando el 1% frente al 0.1% de las demás), por tanto son datos que no aportan una gran confianza. Además, y como ocurría en el stencil 2d4, estas diferencias se encontraban ya en la versión de Hitmap sin tiles distribuidos, por lo que el retardo incluido por la extensión desarrollada es, a lo sumo, 0.7 segundos.

Análisis del código fuente

Los resultados anteriores muestran que las nuevas características añadidas a Hitmap no introducen ningún cuello de botella, sin embargo, es necesario también analizar la verdadera utilidad de la extensión desarrollada y comprobar que realmente reduce la complejidad del código.

El primer análisis consistirá en la comparación directa del código necesario para construir el patrón de comunicación de un stencil 2d9 no compacto. Se incluye también el código implementado en MPI aunque no exista en él el concepto de patrón de comunicación, ya que los intercambios de datos son gestionados manualmente por el programador. Los códigos que se mostrarán a continuación son ejemplos de la estructura que tiene cada una de las implementaciones, el objetivo no es que sean comprendidos en detalle.

El código 7.3 muestra la implementación del las comunicaciones en MPI. Las 6 primeras líneas crean un nuevo tipo de datos en MPI. Esto es necesario ya que los datos de las columnas no son contiguos en memoria, por lo que deben ser leídos "a saltos". El resto del código se divide en dos partes, envío y recepción. En cada una de ellas se gestionan los cuatro intercambios posibles, siendo necesario realizar la comprobación de que existe un vecino en esa dirección así como verificar que no se han producido errores durante la comunicación. Además, se usan funciones de envío y recepción asíncronas para garantizar que no se produce ningún bloqueo mutuo, por lo que es necesario sincronizar todos los procesos al salir de esta sección del código (línea 44).

```
1  //Creacion de un tipo de datos para las columnas
2  MPI_Datatype verticalBorderType;
3  ok = MPI_Type_hvector( size[0], 2, (int)sizeof(double)*size[1],
       MPI_DOUBLE, &verticalBorderType );
4  mpiTestError( ok, "Creating the columns type" );
5  ok = MPI_Type_commit( &verticalBorderType );
6  mpiTestError( ok, "Commiting the columns type" );
7
8  //Envios de datos
9  if ( myCoords[0] != 0 ) {
10     ok = MPI_Isend( &element(matrix, 2, 0), size[1]*2, MPI_DOUBLE,
           rankUp, 0, MPI_COMM_WORLD, &request[0]);
```

```
11      mpiTestError( ok, "Send Up" );
12  }
13  if ( myCoords[0] != procs[0]-1 ) {
14      ok = MPI_Isend( &element(matrix, size[0]-4, 0), size[1]*2,
            MPI_DOUBLE, rankDown, 1, MPI_COMM_WORLD, &request[1]);
15      mpiTestError( ok, "Send Down" );
16  }
17  if ( myCoords[1] != 0 ) {
18      ok = MPI_Isend( &element(matrix, 0, 2), 1, verticalBorderType,
            rankLeft, 2, MPI_COMM_WORLD, &request[2]);
19      mpiTestError( ok, "Send Left" );
20  }
21  if ( myCoords[1] != procs[1]-1 ) {
22      ok = MPI_Isend( &element(matrix, 0, size[1]-4), 1,
            verticalBorderType, rankRight, 3, MPI_COMM_WORLD, &request[3]);
23      mpiTestError( ok, "Send Right" );
24  }
25
26  //Recepcion de datos
27  if ( myCoords[0] != procs[0]-1 ) {
28      ok = MPI_Irecv( &element(matrixCopy, size[0]-2, 0), size[1]*2,
            MPI_DOUBLE, rankDown, 0, MPI_COMM_WORLD, &request[4]);
29      mpiTestError( ok, "Recv from Down" );
30  }
31  if ( myCoords[0] != 0 ) {
32      ok = MPI_Irecv( &element(matrixCopy, 0, 0), size[1]*2, MPI_DOUBLE,
             rankUp, 1, MPI_COMM_WORLD, &request[5]);
33      mpiTestError( ok, "Recv from Up" );
34  }
35  if ( myCoords[1] != procs[1]-1 ) {
36      ok = MPI_Irecv( &element(matrixCopy, 0, size[1]-2), 1,
            verticalBorderType, rankRight, 2, MPI_COMM_WORLD, &request[6]);
37      mpiTestError( ok, "Recv from Right" );
38  }
39  if ( myCoords[1] != 0 ) {
40      ok = MPI_Irecv( &element(matrixCopy, 0, 0), 1, verticalBorderType,
             rankLeft, 3, MPI_COMM_WORLD, &request[7]);
41      mpiTestError( ok, "Recv from Left" );
42  }
43
44  MPI_Waitall( 8, request, statusV );
```

Código 7.3: Implementación de las comunicaciones de un stencil en MPI

La siguiente implementación a estudiar es la realizada en Hitmap sin tiles distribuidos, que se

muestra en el código 7.4. En este caso, el código del extracto mostrado se ubica en la inicialización del programa, haciendo que la parte relativa al cómputo sea más limpia. Sin embargo, el número de líneas también es muy alto, aunque solo haya 5 sentencias. El motivo es que los parámetros pasados a la función son tan largos e incluyen tantos datos, que deben ser separados para mejorar su legibilidad. Sin entrar en detalles específicos de la implementación del stencil, los parámetros pasados a la función son: layout, identificador del vecino al que se envían los datos, tile de envío, shape que selecciona un subconjunto del tile de envío, constante que indica el tipo de coordenadas a utilizar, identificador del vecino del que se reciben datos, tile de recepción, shape que señala en qué región del tile se recibirán los datos, constante que indica el tipo de coordenadas a utilizar, tipo de datos e identificador de la comunicación (usado internamente por MPI). No solo es un número elevado de parámetros, sino que además la obtención de cada uno de ellos supone la la llamada anidada a varias funciones de Hitmap.

```
1  HitPattern neighSync = hit_pattern( HIT_PAT_UNORDERED );
2  hit_patternAdd( &neighSync,
3      hit_comSendRecvSelectTag( matLayout,
4          hit_layNeighbor( matLayout, 0, -1 ) , &tileMat,
5          hit_shape( 2, hit_sig( 2, 3, 1), hit_sig( 2, hit_tileDimCard(
               tileMat,1)-3, 1) ),
6          HIT_COM_TILECOORDS,
7          hit_layNeighbor( matLayout, 0, +1 ) , &tileMat,
8          hit_shape( 2, hit_sig(hit_tileDimCard(tileMat,0)-2,
               hit_tileDimCard(tileMat,0)-1, 1), hit_sig( 2,
               hit_tileDimCard(tileMat,1)-3, 1) ),
9          HIT_COM_TILECOORDS,
10         HIT_DOUBLE, TAG_UP )
11         );
12 hit_patternAdd( &neighSync,
13     hit_comSendRecvSelectTag( matLayout,
14         hit_layNeighbor( matLayout, 0, +1 ) , &tileMat,
15         hit_shape( 2, hit_sig( hit_tileDimCard(tileMat,0)-4,
               hit_tileDimCard(tileMat,0)-3, 1 ), hit_sig( 2,
               hit_tileDimCard(tileMat,1)-3, 1) ),
16         HIT_COM_TILECOORDS,
17         hit_layNeighbor( matLayout, 0, -1 ) , &tileMat,
18         hit_shape( 2, hit_sig(0, 1, 1), hit_sig( 2, hit_tileDimCard(
               tileMat,1)-3, 1) ),
19         HIT_COM_TILECOORDS,
20         HIT_DOUBLE, TAG_DOWN )
21         );
22 hit_patternAdd( &neighSync,
23     hit_comSendRecvSelectTag( matLayout,
24         hit_layNeighbor( matLayout, 1, -1 ) , &tileMat,
25         hit_shape( 2, hit_sig( 2, hit_tileDimCard(tileMat,0)-3, 1),
```

```
            hit_sig(2, 3, 1) ),
        HIT_COM_TILECOORDS,
        hit_layNeighbor( matLayout, 1, +1 ) , &tileMat,
        hit_shape( 2, hit_sig( 2, hit_tileDimCard(tileMat,0)-3, 1),
            hit_sig( hit_tileDimCard(tileMat,1)-2, hit_tileDimCard(
            tileMat,1)-1, 1 ) ),
        HIT_COM_TILECOORDS,
        HIT_DOUBLE, TAG_LEFT )
        );
hit_patternAdd( &neighSync,
    hit_comSendRecvSelectTag( matLayout,
        hit_layNeighbor( matLayout, 1, +1 ) , &tileMat,
        hit_shape( 2, hit_sig( 2, hit_tileDimCard(tileMat,0)-3, 1),
            hit_sig( hit_tileDimCard(tileMat,1)-4, hit_tileDimCard(
            tileMat,1)-3, 1 ) ),
        HIT_COM_TILECOORDS,
        hit_layNeighbor( matLayout, 1, -1 ) , &tileMat,
        hit_shape( 2, hit_sig( 2, hit_tileDimCard(tileMat,0)-3, 1),
            hit_sig(0, 1, 1) ),
        HIT_COM_TILECOORDS,
        HIT_DOUBLE, TAG_RIGHT )
        );
```

Código 7.4: Implementación de las comunicaciones de un stencil en Hitmap (sin tiles distribuidos)

Por último, el código 7.5 muestra la creación del patrón de comunicaciones haciendo uso de los tiles distribuidos. Como se observa, está formado por una única línea, que llama a una función cuyos parámetros son el tile de envío, el tile de recepción, el tipo de datos y un flag que indica si se deben permitir las comunicaciones dentro de un mismo proceso. La llamada a la función realiza internamente los cálculos necesarios de forma que el patrón obtenido sea idéntico al creado manualmente en el código de Hitmap (código 7.4).

```
HitPattern neighSync = hit_patternRedistributeDis(&tileMat, &tileMat,
    HIT_DOUBLE, 0);
```

Código 7.5: Implementación de las comunicaciones de un stencil en Hitmap (con tiles distribuidos)

Una vez vistos los códigos, se llevará a cabo un análisis estático de los mismos para analizar cuantitativamente la disminución de la complejidad. De nuevo se compararán las tres versiones de los stencil (MPI, Hitmap y tiles distribuidos). Para ello se utilizará el software Lizard [28], un script de Python que cuenta el número de líneas de código (LOC), la complejidad ciclomática (CCN) y el número de tokens. La tablas 7.23 y 7.24 muestran los resultados de este análisis.

Los datos muestran una mejoría en todas las métricas tanto en el stencil 2d4 como el 2d9 no compacto. El número de líneas de código necesarias para implementar estos stencils con tiles distribuidos es la mitad que en su versión en MPI. Resulta llamativa la disminución de líneas de código entre las dos versiones de Hitmap, teniendo en cuenta que sus códigos son idénticos a excepción de las líneas de creación del patrón de comunicación. Como se ha visto anteriormente

Versión	NLOC	CCN	Núm. tokens
MPI	205	48	2245
Hitmap	160	13	1438
DIstribTiles	117	13	1079

Tabla 7.23: Resultado del análisis estático del stencil 2d4

Versión	NLOC	CCN	Núm. tokens
MPI	221	49	2450
Hitmap	171	15	1773
DistribTiles	122	15	1270

Tabla 7.24: Resultado del análisis estático del stencil 2d9 no compacto

en el código 7.4, esto se debe a que cada uno de los parámetros de las funciones de creación del patrón de comunicación se escriben en una línea distinta, lo que aumenta la cuenta obtenida por el software de análisis utilizado.

El número de tokens es una métrica más fiable, ya que no influye la disposición de las líneas de código, por lo que no es necesario sacrificar la legibilidad para obtener valores más bajos. En este caso no solo existe una diferencia entre la versión en MPI y la de Hitmap, sino que se produce otra bajada importante entre Hitmap y los tiles distribuidos. Esto se debe a que la extensión desarrollada elimina las ya mencionadas operaciones con índices de tiles que se utilizaban para definir las comunicaciones, que contenían un elevado número de tokens.

Por otro lado, la complejidad ciclomática se reduce bastante al pasar de la versión en MPI a cualquiera de las de Hitmap. Esto se debe al elevado número de ifs que se necesitan en la primera para controlar las comunicaciones, tarea que es asumida por Hitmap en los otros dos casos. Las dos versiones de Hitmap no presentan diferencia ya que su código solo difiere en las líneas de creación del patrón de comunicación, que no cuentan con ningún condicional o bucle, por lo que no aumentan la complejidad.

Capítulo 8

Conclusiones

En este capítulo se tratarán los siguientes puntos:

- Las conclusiones del proyecto.
- Las líneas de trabajo futuro.
- La valoración personal del proyecto.

8.1 Objetivos cumplidos

En este trabajo se ha desarrollado una extensión para Hitmap que permite automatizar la creación de los patrones de comunicación necesarios para intercambiar datos entre procesos. Esto se ha logrado incluyendo una nueva estructura de datos, llamada tile distribuido. Además se han desarrollado diversos stencils para demostrar las capacidades de la extensión implementada.

Se han cumplido por tanto todos los objetivos indicados en las sección 1.4:

1. Se ha conseguido implementar un mecanismo capaz de calcular las comunicaciones necesarias en una aplicación paralela llamando únicamente a una función, la cual solo toma como parámetros los dos tiles que intervendrán en el intercambio de datos. Se ha eliminado por tanto la necesidad de realizar cualquier operación con índices de matrices o identificadores de procesos.

2. Se ha demostrado el uso de la nueva extensión implementando los siguientes stencils: 2d4, 2d9 compacto, 2d9 no compacto, 2d5 no compacto, 3d27 y 3d33. Se han desarrollado además versiones más avanzadas de algunos stencils, que mejoran el comportamiento de la memoria caché reduciendo el tamaño del tile o reducen las necesidades de comunicación aumentando la cardinalidad de los halos.

Como se ha demostrado en la sección 7.3.1, el desarrollo de esta extensión no ha supuesto la aparición de ningún cuello de botella en ninguno de los stencils desarrollados, por tanto, los tiempos de ejecución se han mantenido estables. Por otro lado, se ha conseguido una simplificación notable del código que implementa un stencil, tanto en complejidad ciclomática como en número de líneas y tokens. El resultado es, por tanto, un código más legible y menos propenso a errores, por lo que es más fácil de desarrollar y mantener.

Se han conseguido además objetivos que no estaban contemplados en el alcance del proyecto. En concreto, se ha mejorado la flexibilidad y la capacidad del código desarrollado para manejar casos extremos sin producir errores. Un ejemplo se encontró durante el desarrollo de las pruebas descritas en la sección 7.3.1, donde las comunicaciones creadas a mano fallaban en aquellos casos en los que a cada proceso se les asignaba una región más pequeña que su halo. Por su parte, el método basado en intersecciones era capaz de crear un patrón de comunicación adaptado a este caso, obteniendo así los resultados esperados.

Este trabajo ha dado lugar a un artículo para las XXXII Jornadas de Paralelismo que se celebrarán del 21 al 23 de septiembre en Alicante [29].

8.2 Trabajo futuro

Este trabajo ha demostrado que los tiles distribuidos son una potente herramienta para implementar stencils. Sin embargo, los ejemplos utilizados durante el proyecto son demasiado sencillos y carecen de utilidad. Por tanto, una posibilidad de trabajo futuro es la inclusión de los tiles distribuidos en aplicaciones reales que hagan uso de los stencils como alguno de los proyectos desarrollados por el grupo de investigación Trasgo.

Esta extensión de Hitmap también incluye una característica que no ha sido estudiada ni aprovechada a lo largo del proyecto, las comunicaciones ya no se definen de forma estática en el código, si no que se calculan en tiempo real. Esto hace posible cambiar el reparto inicial de las matrices entre los procesos en tiempo de ejecución, actualizando el patrón de comunicación para reflejar estos cambios. Un ejemplo en el que tendría utilidad esta técnica es el reparto de carga, ya que sería posible asignar más datos a aquellos procesos que se estén ejecutando en máquinas más rápidas, todo ello de forma automática y en tiempo real. Antes era necesario llevar a cabo pruebas previas a la ejecución para calcular la velocidad de cada nodo y adaptar el reparto a mano en el código.

8.3 Valoración personal

Este trabajo de fin de grado ha sido una oportunidad para conocer el trabajo que se realiza dentro de un grupo de investigación como Trasgo, lo que me ha permitido complementar los conocimientos técnicos sobre informática con la metodología utilizada en la investigación.

Haber trabajado añadiendo funciones a una biblioteca me ha hecho valorar la importancia de la documentación y el buen diseño de interfaces, no solo en la parte pública accedida por el código cliente sino también la interna. Esta documentación ha simplificado el proceso de desarrollo de este proyecto, al facilitar la comprensión del código existente.

Durante este proyecto he podido poner en práctica lo aprendido en diversas asignaturas, especialmente computación paralela al estar trabajando con aplicaciones paralelas y concretamente con MPI. Todas los códigos se han realizado en C en un entrono Linux, por lo que la asignatura de Fundamentos de Sistemas Operativos también ha sido útil. Para comprender los resultados de algunas pruebas he empleado los conocimientos adquiridos en las asignaturas relacionadas con la arquitectura de computadores así como los sistemas operativos, que enseñan conceptos como los procesos, hilos, concurrencia, memorias caché, etc. Por último, también he utilizado lo aprendido

en estadística para analizar los resultados de las pruebas y planificación y gestión de proyectos para organizar este trabajo.

Bibliografía

[1] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engi- neers*. CRC Press, 2010.

[2] P. Software, *Hardware survey - cpu cores*, May 2022. [Online]. Available: https://www.pcbenchmarks.net/number-of-cpu-cores.html Último acceso 01/20/2022.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*, Aug. 2018. [Online]. Available: https://pages.cs.wisc.edu/%5C%7Eremzi/OSTEP/ Último acceso 01/20/2022.

[4] Z. J. Czech, *Introduction to parallel computing / Zbigniew J. Czech, Silesia University of Technology.* eng. Cambridge: Cambridge University Press, 2016, ISBN: 9781316795835.

[5] TOP500.org, *June 2021 | top 500*, Jun. 2021. [Online]. Available: https://www.top500.org/lists/top500/2021/06/ Último acceso 01/20/2022.

[6] NERSC, *Perlmutter*, Mar. 2022. [Online]. Available: https://www.nersc.gov/systems/perlmutter/ Último acceso 01/20/2022.

[7] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Heterogeneous computing," *ECE Technical Reports*, p. 206, 1994.

[8] B. Boehm, "Software risk management: Principles and practices," *IEEE Software*, vol. 8, no. 1, pp. 32–41, 1991. DOI: 10.1109/52.62930.

[9] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '89, Reno, Nevada, USA: Association for Computing Machinery, 1989, pp. 655–664, ISBN: 0897913418. DOI: 10.1145/76263.76337. [Online]. Available: https://doi.org/10.1145/76263.76337.

[10] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. R. Llanos, "An extensible system for multilevel automatic data partition and mapping," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1145–1154, 2014. DOI: 10.1109/TPDS.2013.83.

[11] P. Thoman, P. Salzmann, B. Cosenza, and T. Fahringer, "Celerity: High-level c++ for accelerator clusters," in *Euro-Par 2019: Parallel Processing*, R. Yahyapour, Ed., Cham: Springer International Publishing, 2019, pp. 291–303, ISBN: 978-3-030-29400-7.

[12] K. group, *Sycl overview - the khronos group*, 2022. [Online]. Available: https://www.khronos.org/sycl/ Último acceso 01/29/2022.

[13] D. Loveman, "High performance fortran," *IEEE Parallel Distributed Technology: Systems Applications*, vol. 1, no. 1, pp. 25–42, 1993. DOI: 10.1109/88.219857.

[14] Trasgo, *Grupo trasgo – trasgo research group (universidad de valladolid)*. [Online]. Available: https://trasgo.infor.uva.es/ Último acceso 06/11/2022.

[15] R. Hughes and M. Cotterell, *Software project management*, English. McGraw-Hill, May 2009, ISBN: 9780077122799.

[16] PMI, Ed., *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*, 5th ed. Newtown Square, PA: Project Management Institute, 2013, ISBN: 978-1-935589-67-9.

[17] T. U. of Denmark, *Impact and probability in risk assessment*, Nov. 2018. [Online]. Available: http://apppm.man.dtu.dk/index.php/Impact_and_Probability_in_Risk_Assessment Último acceso 01/20/2022.

[18] glassdoor, *Sueldo de programador junior*. [Online]. Available: https://www.glassdoor.es/Sueldos/programador-junior-sueldo-SRCH_KO0,18.htm Último acceso 01/22/2022.

[19] M. Koraei, O. Fatemi, and M. Jahre, "Dcmi: A scalable strategy for accelerating iterative stencil loops on fpgas," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–24, 2019.

[20] G. Roth, G. Roth, J. Mellor-crummey, *et al.*, "Compiling stencils in high performance fortran," in *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM*, ACM Press, 1997, pp. 1–20.

[21] T. Q. U. of Belfast, *Introduction to fortran 90, arrays*. [Online]. Available: https://www.adt.unipd.it/corsi/Bianco/www.pcc.qub.ac.uk/tec/courses/f90/stu-notes/F90_notesMIF_5.html Último acceso 01/19/2022.

[22] cppreference, *Struct declaration - cppreference.com*, Mar. 2022. [Online]. Available: https://en.cppreference.com/w/c/language/struct Último acceso 02/08/2022.

[23] MPICH, *Mpi_ allgather*. [Online]. Available: https://www.mpich.org/static/docs/v3.2/www3/MPI_Allgather.html Último acceso 01/31/2022.

[24] Wikipedia, *Minimum bounding box*, Dec. 2021. [Online]. Available: https://en.wikipedia.org/wiki/Minimum_bounding_box Último acceso 03/15/2022.

[25] G. T. Toussaint, "Solving geometric problems with the rotating calipers," in *Proc. IEEE Melecon*, vol. 83, 1983, A10.

[26] Intel, *¿qué es hyper-threading? - intel*. [Online]. Available: https://www.intel.es/content/www/es/es/gaming/resources/hyper-threading.html Último acceso 04/15/2022.

[27] Wikipedia, *Cache replacement policies*, Apr. 2022. [Online]. Available: https://en.wikipedia.org/wiki/Cache_replacement_policies Último acceso 03/20/2022.

[28] T. yin, *Github - terryyin/lizard: A simple code complexity analyser without caring about the c/c++ header files or java imports, supports most of the popular languages.* Apr. 2022. [Online]. Available: https://github.com/terryyin/lizard Último acceso 05/19/2022.

[29] sarteco, *Jornadas sarteco | sociedad de arquitectura y tecnología de computadores*. [Online]. Available: www.jornadassarteco.org Último acceso 06/13/2022.