Universidad de Valladolid

E.T.S Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática, Mención en Ingeniería de Software

Sistema de equilibrado de carga en computación heterogénea

Autor:

D. Fernando Alonso Pastor

Tutores:

Dr. Yuri Torres de la Sierra
Dr. Arturo González Escribano

Dedicado a mis padres, hermano, familia, profesores y a mis compañeros del Grupo de Investigación.

Resumen

Existen supercomputadores y sistemas de alto rendimiento que disponen de unidades funcionales o nodos que poseen diferentes capacidades de cómputo. Estos sistemas, conocidos como heterogéneos, cuentan con dispositivos de características y fabricantes muy diversos. Por tanto, al no disponer todos sus nodos de los mismos recursos computacionales, el reparto de la carga de trabajo en dichos sistemas es una tarea complicada.

El tipo de distribuciones de datos más comúnmente utilizado es el reparto equitativo entre todos los procesos. Sin embargo, para los sistemas heterogéneos es necesario una política de reparto más sofisticada. Este trabajo propone la integración de un sistema de reparto de la carga computacional por pesos en una herramienta programación paralela distribuida denominada Hitmap y desarrollada por el Grupo de Investigación Trasgo de la Universidad de Valladolid. Para utilizar dicho sistema de reparto de carga, se indica el porcentaje de carga total que se desea asignar a cada proceso. De esta forma, se puede explotar, en la medida de lo posible, la capacidad de cómputo de cada uno de los dispositivos que componen el sistema. Además, se propone como trabajo futuro un sistema automático de equilibrado de la carga computacional en tiempo de ejecución, del que se presenta el diseño.

Por medio de las pruebas realizadas, se comprueba que el sistema de particionado que se ha implementado se comporta correctamente en las situaciones planteadas y se integra correctamente con la herramienta de programación paralela, Hitmap. Nuestro estudio experimental muestra que la propuesta se adapta perfectamente al contexto del modelo de programación escogido, consiguiendo una mejora de rendimiento significativa comparado con una distribución equitativa de la carga de trabajo.

Abstract

There are supercomputers and high performance systems that contains a wide variety of functional units or nodes that have different computational capacities. These systems are known as heterogeneous systems. The load distribution in heterogeneous systems is a complicated task because its nodes doesn't have the same computational resources.

The most common type of data distributions is the equal division of the data across all the system. Nevertheless, in Heterogeneous clusters, a more sophisticated policy of data distribution is needed. This work comes up with the integration of a new type partitioning in Hitmap, a tool developed by Trasgo Research Group. To utilize said tool, the percentage of total load which may be assigned to each process must be indicated. Thus, the computational capacity of each device forming the system can be exploited as far as possible. Moreover, it is proposed as a future work an automatic system of balancing the computational load in execution time, whose design is presented.

By means of the performed tests, it is proven that the implemented partitioning system works as expected in the given situations, and it integrates properly with the parallel programming tool, Hitmap. Our experimental study shows that the proposal adapts perfectly to the context of the chosen programming model, accomplishing a significative performance improvement compared to an equitative distribution of the workload.

Contenidos

Lista de Figuras

Lista de Tablas

Lista de Ejemplos de Código

Capítulo 1

Introducción

El presente Trabajo de Fin de Grado forma parte de un proyecto de investigación desarrollado por el grupo de Investigación Trasgo. En esta sección se realiza una introducción del mismo.

1.1. Computación de Altas Prestaciones

Durante las primeras décadas en las que se introdujeron los ordenadores digitales, la computación era secuencial, esto es, los procesadores o CPUs ejecutaban una sola instrucción a cada momento. Además, las computadoras contaban con una sola CPU. Sin embargo, tenían una arquitectura muy similar a las actuales, ya que, al igual que estas, seguían el modelo de Von-Neumann [1]. Así mismo, contaban, además de con la unidad de procesamiento principal, con uno o varios módulos de memoria, una tarjeta de vídeo y, algunos otros dispositivos secundarios de entrada/salida, red, etc. Todos estos componentes se construían sobre una placa principal o placa base que se encargaba de conectar todos los circuitos electrónicos. Todo ello, sigue realizándose de la misma manera a día de hoy, la diferencia, reside, en el aumento de la capacidad de cómputo y la velocidad de los componentes.

Para conseguir una mayor potencia de cómputo en estas máquinas, normalmente, se aumentaba la velocidad de reloj de los procesadores, obteniendo así, una ejecución de las instrucciones más rápida. Cada vez se fabricaban procesadores con velocidades de reloj más elevadas, sin embargo, se sabía que el límite físico de la velocidad de reloj se llegaría a alcanzar en algún momento, debido a que los transistores por los que están formados los circuitos del procesador, tienen su propio retardo de cambio de estado [2]. Por ello, en algún momento, se alcanzaría la velocidad máxima del procesador, o, expresado de forma más completa, se alcanzaría la velocidad máxima de un procesador digital elaborado a base de circuitos de semiconductores. Esta afirmación se formalizó como la ley de Moore en 1965 [3].

En este contexto, surge la computación paralela, que se define como "una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que problemas grandes, a menudo se pueden dividir en problemas más pequeños, que luego son resueltos simultáneamente" [4]. Hasta ese momento, la única forma de obtener más capacidad de cómputo era mediante el aumento de la frecuencia de reloj en las CPU. Fue entonces, cuando se decidió replicar las unidades de procesamiento de la máquina para obtener así un mayor rendimiento en tiempo de ejecución. La dificultad que conlleva adoptar este tipo de solución, es que los programas tienen que ser específicamente codificados para utilizar

más de una unidad de procesamiento . Por ello, la aceleración en la ejecución del programa siempre va a depender de lo "paralelizable" que sea el problema a resolver y de la forma en que se resuelva dicho problema. Aún así, la solución, no era tan sencilla ni efectiva como el aumento de la frecuencia de reloj, principalmente, porque la tarea de paralelización, no siempre es posible, y, en caso de que sea así, no se mejora el tiempo de ejecución con la misma facilidad, ni en la misma proporción. Más aún, en un principio no existían librerías o frameworks que ayudasen al programador en la codificación con múltiples procesadores.

Uno de los primeros sistemas de clasificación de arquitecturas de computadoras en base a la paralelización de la arquitectura fue el conocido como Taxonomía de Flynn [5] y se define como "clasificación los programas y computadoras atendiendo a si están operando con uno o varios conjuntos de instrucciones y si dichas instrucciones se utilizan en una o varias series de datos" [6] . Se distinguen cuatro categorías principales, como se puede ver en la siguiente tabla. En la categoría de SISD, estarían todos los computadores secuenciales que no explotan ningún tipo de paralelismo. SIMD, se refiere a las arquitecturas que son capaces de ejecutar una sola instrucción al mismo tiempo pero, pueden ejecutarla en múltiples conjuntos simultáneamente. MISD ejecuta múltiples instrucciones sobre los mismos en un mismo periodo de tiempo. MISD es muy poco común y es la menos utilizada. En el modelo MIMD, se tienen unidades funcionales independientes que ejecutan instrucciones diferentes sobre datos, también diferentes. Cunado hemos explicado en el párrafo anterior la replicación de unidades funcionales, nos referíamos a una arquitectura de este último tipo, MIMD, ya que cada unidad funcional es independiente en cuanto a instrucciones que ejecuta y datos sobre los que se ejecutan dichas instrucciones.

	Una instrucción	Múltiples instrucciones
Un Dato	SISD	MISD
Múltiples Datos	SIMD	MIMD

Tabla 1.1: Taxonomía de Flynn [5]

Con la replicación de unidades funcionales, se consigue aumentar la capacidad de cómputo de las máquinas. Sin embargo, las CPUs con múltiples núcleos no existían todavía y, en una misma placa base de una computadora era físicamente imposible ubicar más de dos o, quizá cuatro procesadores. Por ello, surgen los clústers de computadoras [7]. Los *clústers* son conjuntos de computadoras conectadas a través de una red de alta velocidad que funcionan como un único equipo. Cada uno de estas máquinas está constituida por diversas unidades de cómputo. Atendiendo a la similitud entre los nodos o computadoras individuales de un clúster de cómputo podemos dividir dichos clústers en homogéneos y heterogéneos. Gracias a los clústers de computadoras, se consigue una escalabilidad del número de procesadores muy alta, llegando a miles o millones de procesadores o núcleos.

En la década actual, con la llegada de los procesadores multi-core o many-core, se ha conseguido llegar a tener hasta 112 unidades funcionales o núcleos dentro de un mismo procesador [8]. Además de procesadores principales más potentes y con varias unidades funcionales en su interior, también han surgido distintos tipos de aceleradores hardware como las Field-programmable gate arrays (FPGAs) [9], el Intel Xeon Phi

(Hasta 72 núcleos) [10] o incluso las tarjetas gráficas o GPUs que se pueden utilizar como co-procesadores, ya que, algunas de las más modernas cuentan con hasta 5120 núcleos, como es el caso de la NVIDIA Tesla V100 [11]. A pesar de que las GPUs contienen una cantidad de núcleos muy grande, las aplicaciones son limitadas debido al tipo de arquitectura. Además, los núcleos de las GPUs son extremadamente simples, por ello pueden contener tantos en una sola tarjeta y no sirven para cualquier tipo de operación.

La Computación de Alto Rendimiento (HPC) [12] surge del diseño y desarrollo de nuevas computadoras que ofrecen cada vez más potencia de cálculo. Haciendo uso de la computación paralela, concurrente y distribuida tiene como objetivo mejorar el rendimiento de los programas paralelos ejecutados en supercomputadoras. La computación de altas prestaciones, permite explotar al máximo las nuevas tecnologías en computación. Para utilizar varios núcleos de la CPU o varios nodos de un clúster, en un programa, debemos adaptarlo, de forma que realice los cálculos o tareas de forma paralela. Lo mismo ocurre cuando queremos utilizar algún tipo de co-procesador, como por ejemplo, una Tarjeta Gráfica (GPU). En este último caso, deberemos adaptar la aplicación a alguno de los lenguajes o frameworks de programación para GPU. Existen múltiples disciplinas científicas y tecnológicas que han explotado las técnicas de HPC para la resolución de problemas. Entre ellas destacan la biología molecular, la física de partículas, la meteorología, la inteligencia artificial, etc.

La computación de alto rendimiento se basa en la aplicación de ciertas técnicas de computación a ciertos problemas pensados para ser resueltos, generalmente en una supercomputadora. Dichas supercomputadoras consisten, comúnmente, en un clúster de ordenadores muy extenso, que cuentan con algunos de las CPUs y co-procesadores más rápidos del mercado. Existen ciertos sistemas de supercomputación que cuentan con equipos o nodos muy diferentes a lo largo del clúster. Dichos sistemas, los denominamos, heterogéneos. La dificultad añadida que supone tener un supercomputador heterogéneo, es que no todos los nodos poseen la misma capacidad de computación y, por tanto, una distribución de la carga equitativa, no explotaría las capacidades de cómputo de dicho supercomputador.

Las supercomputadoras más potentes del mundo se encuentran listadas en el TOP-500 [13], un proyecto de dedicado al análisis de los equipos con mayores prestaciones pertenecientes al entorno de la supercomputación. En los primeros puestos, figuran equipos con millones de cores y una capacidad de procesamiento muy grande. De hecho, el uso de un supercomputador como *Sunway TaihuLight* durante una hora, equivaldría al consumo eléctrico de una vivienda en España durante más de un año y medio, si contamos con que el gasto medio en España de electricidad al año es de $9992kW * h$ y el consumo del supercomputador *Sunway TaihuLight* durante una hora de uso es de $15371kW * h$. Gracias a esta información, podemos ver incluso el impacto que podría llegar a tener el uso de estos computadores de forma prolongada. Por ello, es necesario aprovechar al máximo los recursos de cómputo, utilizando las técnicas necesarias para repartir adecuadamente los problemas, reduciendo así el tiempo de ejecución y el consumo de energía.

Para desarrollar aplicaciones que se ejecutan en estos dispositivos de computación son necesarios modelos de programación específicos.

1.2. Modelos de Computación Paralela

Con el fin de poder explotar, en la medida de lo posible, las capacidades de multiprocesamiento de todos estos dispositivos multi-core, many-core y, por tanto, todos aquellos supercomputadores que se construyen a base de estos dispositivos, la computación paralela cuenta con, principalmente, dos técnicas de resolución de problemas de forma paralela: el paralelismo de datos y el paralelismo de tareas. El paralelismo de datos, por un lado, se centra en la ejecución de las mismas instrucciones en todas las unidades de procesamiento, cada una, con un conjunto de datos distinto. Por otro lado, el paralelismo de tareas, consiste en repartir la ejecución de los diferentes cálculos de un programa entre las distintas unidades de procesamiento, de tal forma que puedan ser ejecutados de forma simultánea. El desarrollo de este trabajo, se centra en el paralelismo de datos, como modelo de computación paralela. En el trabajo [14] se explica detalladamente el paralelismo de datos.

Para explotar las capacidades de paralelismo de cada dispositivo o conjunto de dispositivos que cuenta con múltiples unidades de procesamiento, existen determinado modelos de programación ligados al tipo del propio dispositivo o a la forma en la que se quiere conseguir paralelismo. Por ejemplo, en una sola máquina, si queremos explotar las capacidades de multiprocesamiento de la CPU, podremos usar OpenMP y paralelizar ciertas partes de un programa. Sin embargo, si queremos utilizar un clúster de ordenadores conectados a través de la red, deberemos emplear otro modelo de programación como MPI, que nos permita paralelizar programas entre diferentes núcleos, ya estén en una misma máquina o en varias máquinas conectadas entre sí a través de la red, como es el caso de un clúster. Otro caso completamente distinto es que quisiésemos utilizar una GPU para resolver un determinado problema, en dicha situación, como además las GPUs funcionan de forma diferente, tendremos que utilizar CUDA u OpenCL.

Por otra parte, para explotar la capacidad de computación de un conjunto de nodos en un clúster, es necesario hacer uso de herramientas y entornos que incluyan mecanismos de particionado de datos. En entornos heterogéneos la distribución de la carga es una tarea más complicada, ya que, para obtener un rendimiento óptimo, es necesario realizar particiones irregulares entre los diferentes nodos [15].

1.2.1. MPI

MPI se define como “MPI ("Message Passing Interface", Interfaz de Paso de Mensajes) es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.” [16], [17]. Existen implementaciones de MPI para varios lenguajes de programación, incluso existen varias de ellas para cada lenguaje. Entre estos lenguajes, destacan: C, C++, Fortran o Python. Con MPI, tenemos un enfoque SPMD (Single Program, Multiple Data), es decir, el programa que se ejecuta en todos los procesadores es el mismo, y, con funciones explícitas de comunicación, los procesadores envían y reciben los datos que necesitan unos de otros. En el Código 1.1 se muestra un ejemplo de un sencillo programa con MPI en C [18].

```
1 int main(int argc, char** argv) {
```

```
2  // Initialize the MPI environment
3  MPI_Init(NULL, NULL);
4
5  // Get the number of processes
6  int world_size;
7  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
8
9  // Get the rank of the process
10 int world_rank;
11 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
12
13 // Get the name of the processor
14 char processor_name[MPI_MAX_PROCESSOR_NAME];
15 int name_len;
16 MPI_Get_processor_name(processor_name, &name_len);
17
18 // Print off a hello world message
19 printf("Hello world from processor %s, rank %d out of %d processors\n",
20 processor_name, world_rank, world_size);
21
22 // Finalize the MPI environment.
23 MPI_Finalize();
24 }
```

Código 1.1: Ejemplo de MPI

Todos los procesadores ejecutan el mismo código, pero cada uno, obtiene un nombre de procesador diferente y un rank o id de procesador distinto. De esta manera, se puede ejecutar código distinto para un solo procesador, estableciendo una condición: si soy el procesador 0, hacer esto. Como ya se ha comentado, existen también, una serie de funciones que sirven para comunicar datos entre procesadores, para hacer broadcast de un procesador a todos, reducciones,etc.

1.2.2. Otros Modelos

Existen muchas otras APIs o modelos de programación paralela, además de los explicados, como pueden ser C++AMP, OpenMP, OpenHMPP, OpenACC, Pthreads, UPC, TBB o modelos específicos para otros tipos de dispositivo o aceleradores hardware como CUDA u OpenCL. Sin embargo, hemos introducido el más importante para el desarrollo de este trabajo: Memoria Distribuida (MPI)

1.3. Hitmap

Hitmap es una librería eficiente pensada para particionado de datos (tiling) jerárquico y mapeado de arrays y estrucuras dispersas [19] desarrollada por el Grupo de Investigación Trasgo [20]. Todo el sistema de comunicaciones de Hitmap está basado en MPI. Está diseñada para usarse con un efoque SPMD (Single Process, Multiple Data) y simplificar la programación (o computación) paralela, permitiendo la creación, manipulación, distribución y comunicación de tiles y jerarquías de tiles [21]. La librería Hitmap enmascara todas las operaciones explícitas de particionado, reparto y comunicación de datos a través de una interfaz,

sencilla de utilizar, en lenguaje C. Podemos elegir el método de reparto de datos a través de los *Layouts*, la topología de los procesadores, a través de las *Topologies* o, las comunicaciones que se desean realizar entre procesadores, a través de los *Communication Patterns*. Podríamos decir que Hitmap. nos añade una capa más de abstracción en la programación paralela, al permitirnos el reparto de tiles, similares a arrays n-dimensionales, entre distintos procesos de forma transparente.

1.3.1. Conceptos básicos

Es necesario introducir los siguientes conceptos clave y notaciones sobre Hitmap:

Signature: Una *signature* S se define como una terna que representa un subespacio de indices sobre un array unidimensional. Una signature sigue la notación de Fortran90 or MATLAB de selección de índices de un array. La cardinalidad de una signature es el número de índices diferentes del dominio [21].

$$S \in \text{Signature} = (\text{begin} : \text{end} : \text{stride})$$
$$\text{Card}(s \in \text{Signature}) = \lfloor (s.end - s.begin + 1)/s.stride \rfloor$$

Shape: Una *shape* h es una n-pla de signatures. Representa una selección de un subespacio de los índices de un array con dominio multidimensional. La cardinalidad de una shape es el número de diferentes combinaciones de índices del dominio [21].

$$h \in \text{Shape} = (S_0, S_1, S_2, \ldots, S_{n-1})$$
$$\text{Card}(h \in \text{Shape}) = \prod_{i=0}^{n-1} \text{Card}(S_i)$$

Tile: Un *tile* es un array n-dimensional. Su dominio es definido por un shape, y tiene un tiene un número de elementos de un tipo (*type*) dado [21].

$$\text{Tile}_{h \in \text{Shape}} : (S_0 \times S_1 \times S_2 \times \ldots S_{n-1}) \rightarrow \texttt{<type>}$$

1.3.2. Arquitectura

En la Figura 1.1, se muestra la arquitectura de la librería Hitmap.

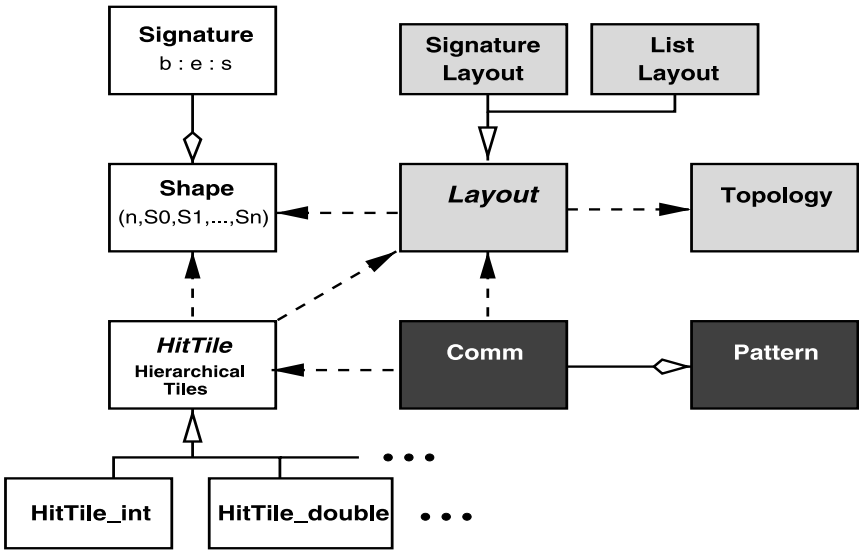

Figura 1.1: Arquitectura de Hitmap [21]

Como se puede apreciar en la Figura 1.1, existen tres conjuntos de funcionalidades distinguibles en Hitmap:

- ***Tiling:*** Parte encargada de la definición y manipulación de arrays y tiles. Los HitTiles, que podemos clasificar de forma general como arrays multidimensionales, son la estructura de datos principal en la que se fundamenta Hitmap. Los HitTiles pueden ser de cualquier tipo básico e incluso de nuevos tipos definidos por el usuario y están definidos por un Shape. Dichos Shapes, incluyen una Signature por cada dimensión que tiene el Tile. De esta forma, un HitTile, bidimensional que contuviese, por ejemplo, 10 posiciones en cada dimensión, contaría con un Shape, de dos elementos, que llevaría la asignatura $0 : 9 : 1$ asociada tanto en S_0 con en S_1.

- ***Mapeo:*** En segunda instancia, tenemos la funcionalidad de mapeo de la librería, que, se encarga de la distribución y disposición de datos. A través de la Topology, se elige una una distribución de los procesadores determinada. Algunos ejemplos de Topologies son: plana (1 dimensión), rectangular o cuadrada (2 dimensiones). De esta forma, los procesos, pueden intercambiar datos en cualquiera de las direcciones que permita la Topology. Los Layouts, en cambio, nos permiten elegir una distribución de la carga determinada, según el propio Layout, a través de los procesos y de forma automática. Para poder utilizar un Layout sobre un HitTile, se necesita establecer una Topology. Mediante los Layouts, los procesos pueden conocer que parte del Tile es la "suya". Existen también varios Layouts diferentes, dependiendo de como se quiera repartir la carga. Algunos ejemplos son: plug_lay_Blocks, que reparte el Tile por bloques de forma equitativa entre todos los procesadores o plug_lay_Cyclic, que reparte el Tile de forma cíclica y equitativa entre todos los procesos.

- ***Comunicación:*** Por último, la librería también incluye toda una parte de comunicaciones. Mediante la información que proporciona el Layout y la Topology (incluida en el Layout), las funciones de comunicación de Hitmap permiten el intercambio de Tiles o elementos de Tiles entre procesos, de una

forma sencilla. Además, en HitMap, existen los Communication Patterns, que permiten establecer un patrón o secuencia de comunicaciones básicas de forma automática. El Comm. Pattern, almacena la secuencia de comunicaciones que se desea realizar, y, simplemente, permite ejecutar dicha secuencia de comunicaciones en un punto del programa, de una forma sencilla.

1.3.3. Layouts

En la librería Hitmap, el particionado y mapeado de datos, como ya se ha comentado, se realiza mediante los Layouts. Debido a que uno del temas principales de este trabajo serán los Layouts, creemos que es importante realizar una introducción más amplia.

Existen dos tipos Layouts en Hitmap: *Signature Layouts* y *List Layouts* [21]. Los primeros se utilizan, generalmente, para realizar particiones regulares, asignando una Signature a cada proceso. En cambio, los List Layouts, sirven para realizar particiones irregulares por medio de otros algoritmos, son menos eficientes y carecen de cierta funcionalidad necesaria en determinadas situaciones. Algunos de los *Signature Layouts* más utilizados de Hitmap son los siguientes:

- **plug_layBlocks** Particiona cada dimensión en fragmentos contiguos de tamaño similar. Por ejemplo, si el Shape tiene una dimensión, dividirá el número de elementos del Shape entre el número de procesos definidos en el Layout, de forma que, cada uno tenga un número similar de elementos. En la Figura 1.2 se muestra este reparto por bloques en una dimensión (primera fila a la izquierda: **BLOCK**), y, en dos dimensiones (segunda fila, tercera columna: **BLOCK, BLOCK**). En [22] se puede encontrar información más detallada acerca del reparto por bloques.

- **plug_layDimBlocks** Al igual que el anterior, fragmenta el Shape del HitTile en partes de igual tamaño, sin embargo, solo se realiza en una dimensión, la que se especifique en los argumentos. En la Figura 1.2 se muestra este reparto por bloques, partiendo por filas, dimensión 0 (segunda fila, primera columna: **BLOCK,***), y, partiendo por columnas, dimensión 1 (segunda fila, segunda columna: ***, BLOCK**)

- **plug_layBlocksF** Similar a **plug_layBlocks**, con la diferencia de que, si no se puede entregar un número igual de elementos a todos los procesos, los primeros procesos obtendrán un mayor número.

- **plug_layBlocksL** Similar a **plug_layBlocks**, con la diferencia de que, si no se puede entregar un número igual de elementos a todos los procesos, los últimos procesos obtendrán un mayor número.

- **plug_layCyclic** Distribuye los elementos de forma cíclica entre los procesos, esto es, el primer elemento del HitTile se le entrega al procesador 0, el segundo elemento al procesador 1, y así sucesivamente, hasta repartir todo los elementos. Este realiza repartos multidimensionales, es decir, reparte todas las dimensiones del Tile entre todos los procesos. En la Figura 1.2 se muestra esta partición cíclica, en una dimensión (primera fila a la derecha: **CYCLIC**), y, en dos dimensiones (tercera fila, tercera columna: **CYCLIC, CYCLIC**). En el caso de las particiones cíclicas, no existe en Hitmap, un Layout que reparta cíclicamente por una sola dimensión, ya que, dicha partición, no se utiliza tan frecuentemente. En futuro, podrá ser desarrollada si es necesario. [22] se puede encontrar información más detallada acerca de las particiones cíclicas.

- **plug_layInLeader** Toda la carga se entrega al proceso principal.
- **plug_layBlocksBalance** Un porcentaje de la carga elegido por el usuario se le entrega a la dimensión seleccionada, y, el resto se reparte por bloques de forma equitativa.

Además de estos Layouts que hemos explicado, existen también otros que realizan diferentes particiones, menos comunes o con menor interés para este trabajo: **plug_layMetis**, **plug_layBitmap**, **plug_laySparseRows**, **plug_laySparseBitmapRows**,**plug_layIndependentLB**.

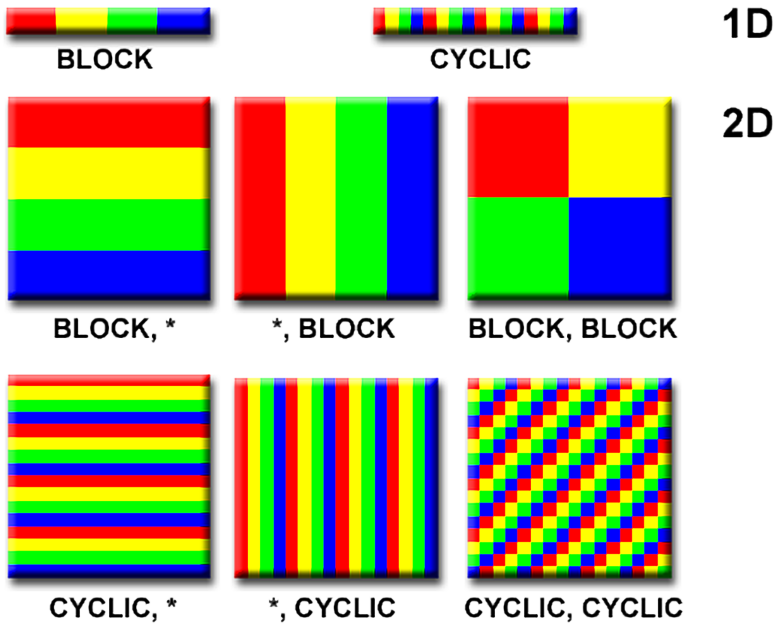

Figura 1.2: Particiones de bloques y cíclicas [23]

En cuanto a los *List Layouts*, existen dos Layouts, que, resultan poco eficientes de forma general y está diseñado, cada uno, para un problema concreto. Ambos funcionan solamente en HitTiles de una dimensión y están adaptados al problema para el que se diseñó cada uno. Estos son **plug_layIndependentLB** y **plug_layContiguous**.

1.4. Trabajo Relacionado

Aumentar la localidad de los datos respectos a los elementos de cómputo que los utilizan permite reducir los costes de comunicaciones o de acceso a los datos. Las técnicas orientadas a ello se han convertido en un elemento clave para la paralelización efectiva de aplicaciones. Especialmente en el contexto de los actuales sistemas paralelos, cada vez más heterogéneos, más complejos en sus jerarquías de memoria, con espacios de datos separados y/o distribuidos [24].

En modelos de programación clásicos, como HPF (High-Performance-Fortran) [25], [26] ya se considera la idea de expresar afinidades entre elementos de estructuras de datos y de proceso, o elementos de estructuras de datos que deben ser mapeadas en el mismo elemento de proceso. La mayor parte de las

técnicas propuestas en este sentido se han orientado a un análisis en tiempo de compilación.

Un objetivo perseguido por diversas propuestas de lenguajes o modelos de programación paralela ha sido introducir abstracciones que separen las especificaciones de los algoritmos de las técnicas de partición y distribución de las estructuras de datos. De esta forma, el programador puede probar diversas estrategias de partición sin necesidad de reorganizar el código por completo. Esto es muy evidente, por ejemplo, en los modelos de programación de tipo PGAS, como Chapel [27], STAPL [28] o DASH [29].

En cuanto a las técnicas de partición y distribución especificamente orientadas a sistemas heterogéneos, ya en [30], se muestran algunas técnicas que permiten realizar una asignación de carga a cada nodo de una forma eficiente, aunque no se diseñe una política o mecanismo de particionado.

En el año 2005, Dongarra et.al [31] se focalizan en la computación heterogénea y los clústers, describiendo el trabajo futuro por hacer. En [32] se realiza un estudio del estado del arte sobre la computación heterogénea, centrándose en el uso de algunos co-procesadores como GPUs, FPGAs junto con los procesadores tradicionales para la resolución de problemas en sistemas heterogéneos. Más modernamente, el trabajo de Unat et.al [24] clasifica las técnicas actuales sobre localidad de datos en general, mostrando las tendencias actuales a utilizar sistemas dinámicos de equilibrado de carga, pero basados en tareas de grano fino o medio, con el consiguiente sobrecoste en el sistema de ejecución.

Estos trabajos carecen de modularidad dentro del sistema de programación paralela, que permite aumentar la funcionalidad sin necesidad de reestructurar el código para utilizarla. Además, debido a que no permite aplicar la partición en tiempo de ejecución, es necesario recompilar la aplicación para realizar cambios.

1.5. Motivación

En los sistemas de supercomputación heterogéneos cada nodo cuenta con capacidades de procesamiento muy diferentes. Uno de los grandes retos de los supercomputadores heterogéneos es la realización de un reparto de carga óptimo entre los nodos para hacer un uso más eficiente de los recursos hardware que componen el clúster. La librería Hitmap carece de dicho mecanismo de reparto de carga, que logre un aprovechamiento óptimo de los recursos en sistemas heterogéneos.

Este trabajo se centra en el diseño de un plug-in para Hitmap, a través de la creación de un Layout, que permita el reparto por pesos de la carga computacional en diferentes procesadores en tiempo de ejecución. Por medio de este sistema, se buscará aumentar la velocidad de ejecución de todos aquellas aplicaciones compatibles con este sistema de reparto de carga. Algunas de ellas, al estar programadas usando la librería Hitmap, no es necesario codificarlas de nuevo, sino que, simplemente con realizar un cambio de Layout en el código, sería suficiente. Nuestra propuesta se presenta como una técnica sencilla de reparto de datos en grano grueso, apropiado para sistemas distribuidos o con alto coste de transferencia de datos.

De forma adicional, se diseñará un mecanismo de autoajuste de la carga en tiempo de ejecución, que, utilizando el Layout de Hitmap, realice un equilibrado de la carga automático entre los diferentes procesos.

Debido al corto espacio de tiempo del que se dispone en un Tabajo de Fin de Grado, solamente será posible diseñar dicho sistema, dejando como trabajo futuro su implementación.

1.6. Objetivos

A continuación, se muestra la lista de objetivos que se pretenden conseguir mediante la realización del presente trabajo:

- Estudiar el uso de la librería Hitmap para la resolución de problemas de programación y computación paralela y de alto rendimiento.
- Comprender el funcionamiento de la librería Hitmap, así como, ser capaz de añadir e integrar funcionalidad en dicha herramienta.
- Realizar el estudio del estado del arte de sistemas de equilibrio de carga en computación heterogénea y homogénea.
- Construir un prototipo de Layout para Hitmap, que permita el reparto de carga selectivo por pesos entre procesadores.
- Elaborar los test y pruebas necesarios para probar el funcionamiento de todas las funciones desarrolladas para la creación del nuevo Layout.
- Realizar una experimentación para probar la mejora de este sistema con un ajuste manual, respecto a un reparto equitativo .
- Diseñar un sistema que sea capaz de autoajustar los pesos de cada nodo y procesador en tiempo de ejecución, sirviéndose del Layout creado previamente.

1.7. Metodología utilizada

La metodología empleada para la realización del trabajo se basa en la investigación y desarrollo del software necesario para la resolución del problema planteado, en este caso, un sistema de equilibrado de carga para computación heterogénea integrado en la herramienta Hitmap.

En primer lugar, se realizará un estudio del estado del arte para comprobar si el ámbito de investigación del trabajo estaba cubierto por algún otro artículo, o, si es posible emplear alguna de las investigaciones ya realizadas para abordar parte del problema a resolver.

En segundo lugar, se realizará un estudio de los conocimientos que sean necesarios para la realización del trabajo. Dicho estudio, se basará principalmente en la herramienta Hitmap, en concreto, en la comprensión del uso de la misma, así como, su funcionamiento. Sin embargo, también se necesitará estudiar otros conocimiento auxiliares de computación paralela, supercomputadores, computación heterogénea, etc.

En tercer lugar, se procederá a una búsqueda iterativa de la solución para obtener un sistema de equilibrado de carga para sistemas heterogéneos. Por tanto, se desarrollarán versiones incrementales de la solución

final. En cada iteración, se diseñará e implementará, en primer lugar, el software necesario, basado en el desarrollo teórico realizado, para cubrir los objetivos de dicha iteración. Seguidamente, se realizarán los test unitarios y de integración que han sido necesarios para la verificación y validación del software de cada iteración. Además, también se realizarán experimentaciones, tras el desarrollo de algunas partes, para mostrar la aceleración que supone, en los problemas elegidos, un buen equilibrado de carga en sistemas heterogéneos.

Por último, se extraerán las conclusiones pertinentes de la realización del trabajo, así como, el trabajo futuro que plantea este sistema de equilibrado de carga, las posibles mejoras, aplicaciones, etc.

1.8. Estructura del documento

El capítulo 2 se presentará la propuesta, mientras que el capítulo 3 expondrá la implementación de dicha propuesta. Por otra parte, el capítulo 4 probará que el desarrollo es funcional mediante la realización de los tests unitarios y de integración que sean necesarios. En el capítulo 5 se mostrará el desarrollo experimental y los resultados de la experimentación, tratando de probar que el propósito del trabajo se ha cumplido. Por último en el capítulo 6 se aportarán las conclusiones obtenidas mediante la realización del TFG y el trabajo futuro que se podría realizar partiendo de las ideas, prototipos y sistemas desarrollados,

Capítulo 2

Propuesta

En este capítulo se introducen las ideas propuestas para el desarrollo de un sistema de reparto de carga en computación heterogénea. Se diseñará un mecanismo de particionado de los datos que permita ajustar la carga de los diferentes procesos, de manera no equitativa, en la ejecución de una aplicación. Este sistema se integrará en la librería Hitmap para permitir distribuciones de carga por pesos entre los diferentes procesos y, de esta forma, favorecer la explotación de los entornos heterogéneos.

De forma adicional, se procederá al diseño de un sistema que realice un ajuste automático de la carga en tiempo de ejecución, utilizando, para ello, el Layout de la librería Hitmap de particionado de carga por pesos propuesto.

2.1. Sistema de particionado de la carga por pesos

Realizar una distribución correcta de la carga en un sistema heterogéneo no es una tarea sencilla de abordar. En general, las técnicas de particionado y mapeado de datos más comunes, se apoyan, habitualmente, en una distribución equitativa de los datos por nodo, es decir, para todos los nodos, la carga es similar. En un supercomputador heterogéneo, un particionado de datos equitativo no explotaría toda la capacidad de cómputo del sistema, ya que, pueden existir nodos con menor capacidad de cómputo que otros, o, nodos en otras subredes donde las comunicaciones sean más lentas. Este tipo de máquinas puede provocar que la ejecución del programa se extienda más de lo necesario. Tampoco sería una solución válida y eficiente no utilizar este tipo de nodos, ya que, aunque sean los más lentos, pueden ayudar a acelerar la ejecución del programa, aunque, para ello, hay que realizar una distribución de la carga más sofisticada que un reparto equitativo.

Nosotros planteamos un reparto por pesos de la carga a lo largo de los diferentes nodos, es decir, a cada proceso se le otorga una fracción de la carga total, igual al peso elegido para dicho proceso. De esta manera, a los procesos de los nodos con mayor capacidad de cómputo se les entrega una mayor carga computacional, otorgándoles un mayor peso, mientras que, a los procesos de otros nodos con menor capacidad se les entrega menor carga, otorgándoles un menor peso. Por medio de este sistema de reparto de carga, se consigue, explotar los recursos de un sistema heterogéneo, consiguiendo una mayor capacidad de cómputo, ya que, todos los nodos del sistema, incluso los más lentos, pueden ejecutar parte de la aplicación sin resultar un cuello de botella para aquellos nodos con mejores prestaciones. De hecho, se

consigue una mayor velocidad de ejecución, en determinadas situaciones, utilizando todos los nodos debido a que, mientras los nodos con mejores prestaciones realizan la mayoría del trabajo, la parte restante puede ser completada simultáneamente, por aquellos con capacidades de cómputo reducidas.

Para realizar el desarrollo de un sistema de particionado de la carga por pesos, se diseña un plug-in para la librería Hitmap que lo permita. Esta decisión de integrar dicho sistema de particionado con la librería, se ha debido, principalmente, a las siguientes razones:

- En primer lugar, **la portabilidad**. Dado que el sistema se desarrolla dentro de una librería de particionado de datos, cualquier aplicación programada con dicha librería, puede utilizar el reparto de carga por pesos en cualquier sistema. De este modo, se permite la posibilidad de ejecutar aplicaciones compatibles con dicho reparto de la carga en un entorno heterogéneo, explotando así sus recursos de forma eficiente. Por tanto, no es necesario que el programador cambie el código de la aplicación para utilizar el nuevo particionado de carga para sistemas heterogéneos, si ya hace uso de la librería Hitmap.

- En segundo lugar, **la facilidad de integración**. La librería Hitmap ya posee toda la funcionalidad necesaria para el particionado de datos jerárquico y el mapeado de arrays, es decir, Hitmap nos provee de la funcionalidad de particionado, reparto de carga, comunicaciones, mecanismos de sincronización, etc. de una forma transparente. En nuestro caso, simplemente, con añadir un nuevo Layout y realizar algunos otros ajustes necesarios sería suficiente para integrar una nueva forma de reparto. Además, los Layouts de Hitmap están diseñados para poder ampliarse sin necesidad de modificar otras partes de la librería, por lo que nos facilita bastante la tarea de programación.

- En tercer lugar, **la compatibilidad con otros proyectos del Grupo de Investigación Trasgo**. El HitTile, de la librería Hitmap, se utiliza como estructura de datos base en la librería Controller [33], también desarrollada por el Grupo de Investigación Trasgo. Por ello, se puede utilizar, también en Controller, la distribución balanceada de carga sin tener que desarrollar específicamente el software necesario para implantarla en dicha librería. Por medio de Controller, el particionado de carga se podrá llevar también a aceleradores hardware tales como GPUs, FPGAs Xeon Phi, etc.

- Por último, **la futura utilidad del trabajo desarrollado en la comunidad científica**. Existen multitud de propuestas útiles, en las distintas áreas de cocimiento, que nunca han llegado a ser ampliamente utilizadas. Aunque la solución más sencilla podría haber sido crear una librería sobre MPI que simplemente realizase la distribución de carga por pesos, sin ocuparse de comunicaciones, estructuras de datos, etc., no dejaría de ser un prototipo que habría que adaptar e integrar en otra parte y posiblemente volver a codificar. Esto ya genera un desarrollo más costoso que posiblemente llevaría a la no utilización y, finalmente, al olvido del trabajo realizado. Gracias a la librería Hitmap, que propone una abstracción de MPI utilizable sin necesidad de readaptaciones ni cambios, con diversas aplicaciones y benchmarks que lo confirman, se facilita la utilización del sistema de reparto de carga en otras aplicaciones.

Dado que el plug-in de particionado de la carga por pesos, se integrará en Hitmap, a través de un Layout, es importante asegurar que la librería Hitmap no dispone de un Layout de reparto por pesos que realmente funcione, de forma general, en sistemas heterogéneos. Dado que los Layouts que reparten

por pesos (ver Sección 1.3.3) no son Layouts genéricos que podamos utilizar fuera del ámbito para el que fueron diseñados ni son Layouts compatibles con HitTiles de múltiples dimensiones, no nos resultan útiles para realizar un reparto de carga en sistemas heterogéneos que sirva para todos los tipos de problemas que se puedan resolver con particiones balanceadas según unos determinados pesos.

El plug-in de particionado por pesos para Hitmap permitirá indicar, a través de unos pesos, el volumen de la carga que se envía a cada proceso del clúster heterogéneo en el que se ejecuta una aplicación. En la Figura 2.1 se muestra la idea de la distribución de la carga por pesos en el Layout. A los procesos de los nodos con menos capacidad se les reparte la menor carga computacional, en este caso 0.1 y 0.2, dejando el $0.7(0.4 + 0.3)$ o 70 % de la carga para los nodos con mayor capacidad de cómputo.

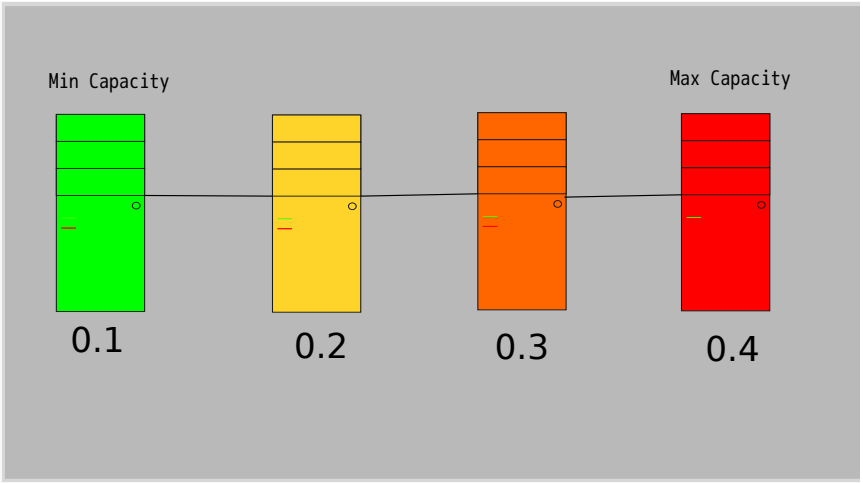

Figura 2.1: Ejemplo de una distribución de la carga por pesos en un clúster heterogéneo ideal C_1, con nodos N_1, N_2, N_3 y N_4, siendo N_1 el nodo más a la izquierda y N_4 el nodo de la derecha

En la distribución de la Figura 2.1, se muestra un clúster heterogéneo C_1. Suponemos que los procesos del primer nodo N_1 tardan t en ejecutar la aplicación A_1 con el 100 % de la carga, por tanto:

Siendo $T(A, E)$ = Tiempo de ejecución de la aplicación A con el 100 % de la carga en un entorno E

$T(A_1, N_1) = t$

$T(A_1, N_2) = \frac{t}{2}$

$T(A_1, N_3) = \frac{t}{3}$

$T(A_1, N_4) = \frac{t}{4}$

Damos por hecho, también, en esta situación ideal que:

$T(A, E) = x \implies T(n * A, E) = n * x,$ para n=1, la carga es del 100 %

Por ello, al ejecutar una fracción de carga de una aplicación, se obtiene un tiempo proporcional a dicha fracción respecto del tiempo de ejecución de la aplicación completa.
Suponiendo una ejecución paralela en todos los nodos, si la carga se reparte como se indica en la Figura 2.1, el tiempo total de ejecución es el máximo del tiempo de ejecución individual de cada nodo:

$$T(A_1, C_1) = max(T(0.1 * A_1, N_1), T(0.2 * A_1, N_2), T(0.3 * A_1, N_3), T(0.4 * A_1, N_4)) = 0.1 * t$$

Partiendo de una distribución de carga, teóricamente, conseguimos el máximo rendimiento, ya que, al ser todos los tiempos de ejecución son iguales, todos los procesos realizan el máximo trabajo posible. Si, por el contrario, hubiésemos distribuido la carga de forma equitativa en el clúster heterogéneo, hubiésemos obtenido:

$$T(A_1, C_1) = max(T(0.25 * A_1, N_1), T(0.25 * A_1, N_2), T(0.25 * A_1, N_3), T(0.25 * A_1, N_4)) = 0.25 * t$$

Un tiempo de ejecución de más del doble y, además, una muy baja explotación del clúster. Por ejemplo, los procesos de N_4 permanecerían en reposo $0.25t - 0.0625t = 0.1875t$, un 75 % del tiempo total de ejecución, lo cual, nos llevaría a la posible desconexión de los nodos más lentos, cuando, como se puede ver en el ejemplo, aceleran la ejecución del problema, si se les otorga la carga adecuada.

Aunque en el ejemplo de la Figura 2.1 se haya utilizado una distribución por pesos a nivel nodo, en realidad, la aproximación que se propone para este trabajo es utilizar una distribución por pesos, de grano más fino, esto es, a nivel proceso. De esta forma, cuando se diseñe e implemente el autoajuste de los pesos, en la segunda parte del trabajo, se podrá realizar con una mayor precisión.

En realidad, vamos a proponer el diseño de dos Layouts de reparto por pesos. El primero, recibirá el nombre **plug_layDimBlocksWeighted** y será el principal. El segundo de ellos, será el **plug_layBlocksWeightedToSelectedDim**. Ambos son Signature Layout (ver Sección 1.3.3), ya que es la única forma de conseguir un Layout eficiente y genérico, que podamos utilizar para una distribución por pesos de cualquier HitTile.

2.1.1. Layout plug_layDimBlocksWeighted

Por medio de este Layout, se podrán realizar repartos en HitTiles de una y varias dimensiones, por pesos. En el caso de utilizarlo con un HitTile de una sola dimensión, simplemente, debemos pasar el peso de cada proceso en un array. La posición del array, indicará el número o identificador del proceso, y, el contenido el propio peso. Por ejemplo, si se van a utilizar 8 procesos en el lanzamiento de una aplicación, el array de pesos debería ser similar al siguiente:

$$\text{Pesos} \leftarrow [p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7]$$

Dado que tenemos en cuenta

$$\sum_{i=0}^{Nprocs-1} p_i \stackrel{?}{=} 1$$

La escala que se utilice para los pesos de los procesos es indiferente, ya que, el número de elementos de cada proceso está definida por:

$\frac{p_i}{\sum_{i=0}^{Nprocs-1} p_i} * card(S)$, siendo S el Shape del HitTile (ver Sección 1.3.1 para los conceptos de HitMap).

A cada proceso, se le entrega una parte del Tile que corresponde con el peso otorgado. En la Figura 2.2 se muestra un ejemplo de la distribución de la carga por pesos en un Tile de 10 elementos $(0 : 9 : 1)$, de una sola dimensión, y, entre tres procesos distintos. A Proc 1 se le otorgó un peso de 0.5 del Tile, a Proc 2 se le proporcionó un peso de 0.2 y a Proc 3, 0.3

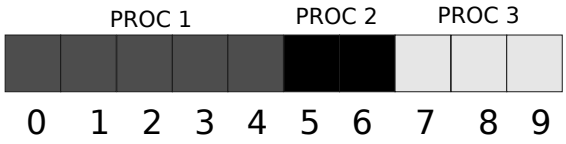

Figura 2.2: Tile distribuido por pesos entre tres procesadores

Como se puede observar, cuando solamente existe una dimensión el reparto del Shape del HitTile es sencillo de comprender, la única dimensión que se reparte es la que existe, ya que, tampoco existen más posibilidades.

Sin embargo, cuando se realizan repartos en un HitTile de varias dimensiones, existen más opciones para realizar las particiones. La estructura se puede dividir por una sola dimensión, por varias o por todas. El Layout por pesos que se ha diseñado soportará todas las formas de reparto en HitTiles multidimensionales. A continuación, se mostrará más detalladamente cada tipo de partición que se puede realizar en estructuras multidimensionales por pesos.

En la Figura 2.3 podemos observar un reparto por pesos en una sola dimensión en una estructura bidimensional más comúnmente conocida como matriz. Los pesos que se han otorgado a cada proceso, han sido, en ambos casos, $[0.3, 0.1, 0.4, 0.2]$.

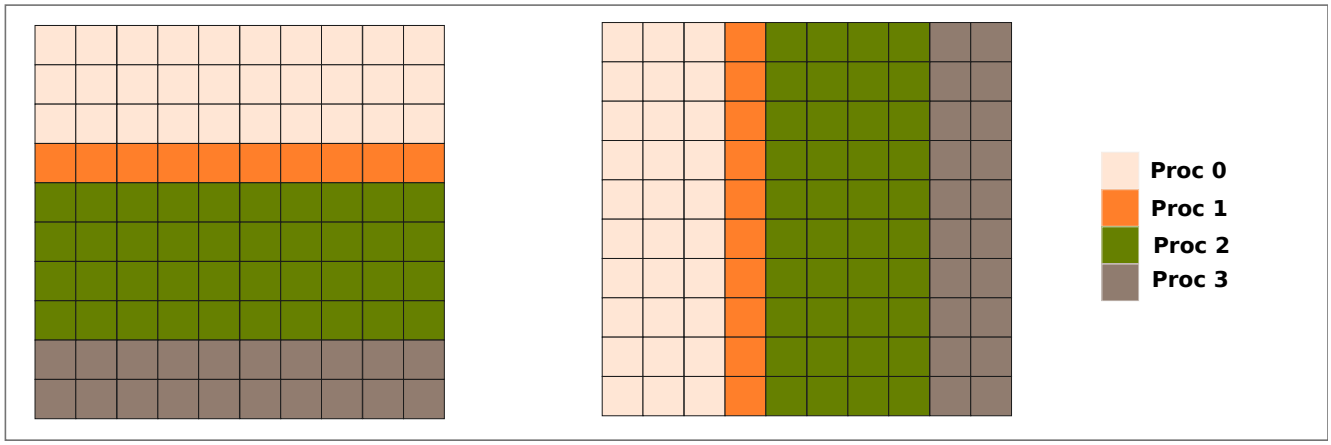

Figura 2.3: Reparto por pesos en una estructura de dos dimensiones por filas (izquierda) y por columnas (derecha)

Al realizar una reparto en una sola dimensión, los elementos de dicha dimensión se distribuyen según los pesos indicados y los elementos de las dimensiones restantes, se entregan todos a cada proceso. En el caso de la Figura 2.3, el reparto de Shapes es como se muestra a continuación.

En la matriz de la izquierda (por filas), se reparte el Shape de la primera dimensión:

- *Proceso 0:* S_{P0} = (0:2:1, 0:9:1)
- *Proceso 1:* S_{P1} = (3:3:1, 0:9:1)
- *Proceso 2:* S_{P2} = (4:7:1, 0:9:1)
- *Proceso 3:* S_{P3} = (8:9:1, 0:9:1)

En la matriz de la derecha (por columnas), se reparte el Shape de la segunda dimensión:

- *Proceso 0:* S_{P0} = (0:9:1, 0:2:1)
- *Proceso 1:* S_{P1} = (0:9:1, 3:3:1)
- *Proceso 2:* S_{P2} = (0:9:1, 4:7:1)
- *Proceso 3:* S_{P3} = (0:9:1, 8:9:1)

En la Figura 2.4 se muestra un reparto en múltiples dimensiones de forma jerárquica, es decir, el particionado de la dimensión 1 se realiza sobre cada uno de los fragmentos en los que se dividió la dimensión 0. Dicho particionado en múltiples dimensiones es el que soportará el Layout layDimBlocksWeighted que estamos diseñando y desarrollando. La distribución de pesos en la dimensión 0, viene dada, al igual que en la Figura 2.3, por el array siguiente $[0.3, 0.1, 0.4, 0.2]$. Dentro de cada uno de los subtiles que se forman al dividir la dimensión 0, las columnas se reparten mediante una distribución diferente de pesos. En el caso de la primera partición de la dimensión 0, los pesos para las particiones de la dimensión 1 serían $[0.4, 0.4, 0.2]$, en cambio, en el caso del cuarto fragmento de la dimensión 0, la distribución de pesos utilizada para la columna es totalmente diferente $[0.6, 0.2, 0.2]$

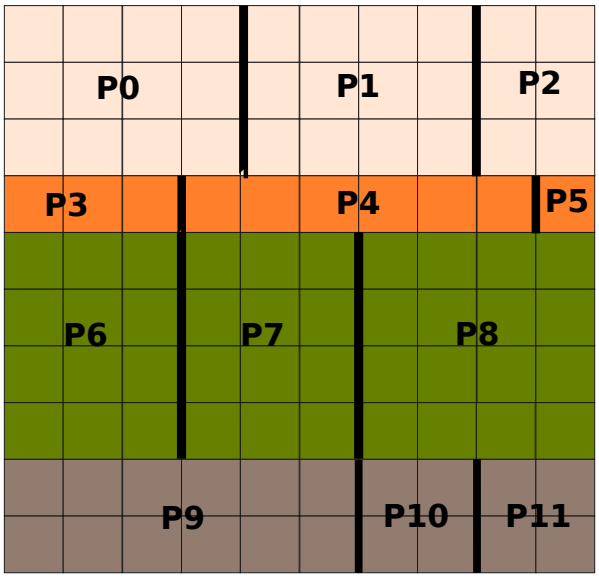

Figura 2.4: Reparto por pesos en una estructura de dos dimensiones de forma jerárquica

En el ejemplo de la Figura 2.4 se muestra el trozo que cada proceso obtendría. Seguidamente, mostraremos los Shapes que obtendrá cada proceso, dada la matriz 10×10 y los 12 procesos del ejemplo.

- *Proceso 0:* $S_{P0} = (0{:}2{:}1,\ 0{:}3{:}1)$
- *Proceso 1:* $S_{P1} = (0{:}2{:}1,\ 4{:}7{:}1)$
- *Proceso 2:* $S_{P2} = (0{:}2{:}1,\ 8{:}9{:}1)$
- *Proceso 3:* $S_{P3} = (3{:}3{:}1,\ 0{:}2{:}1)$
- *Proceso 4:* $S_{P4} = (3{:}3{:}1,\ 3{:}8{:}1)$
- *Proceso 5:* $S_{P5} = (3{:}3{:}1,\ 9{:}9{:}1)$
- *Proceso 6:* $S_{P6} = (4{:}7{:}1,\ 0{:}2{:}1)$
- *Proceso 7:* $S_{P7} = (4{:}7{:}1,\ 3{:}5{:}1)$
- *Proceso 8:* $S_{P8} = (4{:}7{:}1,\ 6{:}9{:}1)$
- *Proceso 9:* $S_{P9} = (8{:}9{:}1,\ 0{:}5{:}1)$
- *Proceso 10:* $S_{P10} = (8{:}9{:}1,\ 6{:}7{:}1)$
- *Proceso 11:* $S_{P11} = (8{:}9{:}1,\ 8{:}9{:}1)$

Como se ha comentado anteriormente, el reparto por varias dimensiones que realizará el Layout es jerárquico. Por ello, no se permite realizar repartos irregulares, como el que se muestra en la Figura 2.5, pues tampoco son útiles para ninguna de las situaciones que hemos contemplado. Como se puede observar

en la Figura 2.5, se muestra un tipo de reparto por pesos en múltiples dimensiones de forma irregular, la parte de la matriz que obtiene cada proceso viene representada en un color diferente. Como se puede apreciar, no se realiza un reparto de una dimensión y después las sucesivas, sino que se reparto toda la estructura de forma irregular y al mismo tiempo. Además de ser más complicado de gestionar, no nos aporta nada poder realizar este tipo de particiones por bloques de forma irregular, ya que, no representa ningún tipo de reparto que explote las capacidades de las máquinas de una mejor forma ni provoca el ahorro de comunicaciones entre máquinas, ya que, los bloques no son siempre contiguos en memoria. Por ello, no es de interés.

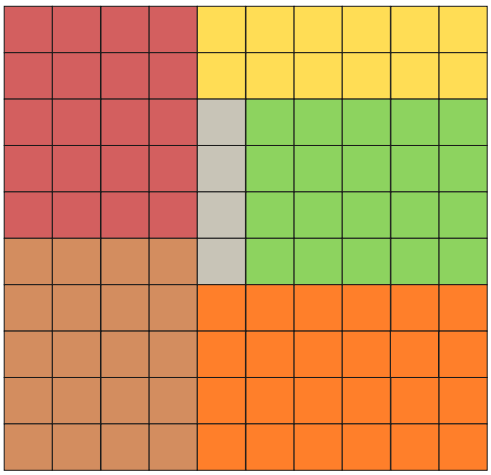

Figura 2.5: Reparto irregular por pesos en una estructura de dos dimensiones

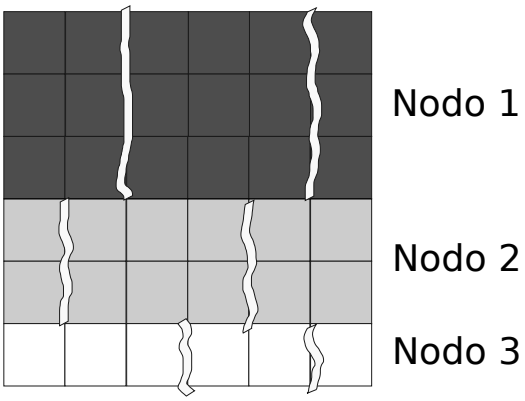

Figura 2.6: Tile multidimensional distribuido por pesos entre nodos

Por el contrario, el modelo de repartición jerárquico par varias dimensiones sí resulta útil, sobretodo, para uno de los casos más comunes, como es, el reparto entre diferentes nodos de una estructura, y, dentro del nodo el reparto entre los diferentes dispositivos de procesamiento. Para obtener un reparto jerárquico, es necesario realizar una primera partición por nodo con unos pesos determinados, es decir, por ejemplo,

un primer reparto por la dimensión 0. Después, dentro de cada uno de los SubTiles que se entregan a cada nodo, se necesita hacer un segundo reparto entre los procesos o dispositivos de procesamiento auxiliares que contiene el nodo, es decir, se realiza un reparto con diferentes pesos de la segunda dimensión en cada uno de los fragmentos que se han originado en el primer reparto. Un ejemplo de ello, se expone en la Figura 2.6. A cada nodo se le entregaría las filas correspondientes, y, dentro de cada nodo, se realizaría una distribución a nivel de proceso, o, incluso en otros dispositivos de procesamiento tales como aceleradores hardware.

Por otra parte, hemos mostrado un ejemplo de particionado en estructuras multidimensionales por una dimensión, en la Figura 2.3 y por todas las dimensiones en la Figura 2.4. Para explicar un particionado por pesos solo en algunas dimensiones, simplemente utilizaríamos una distribución como la de la Figura 2.4, pero, añadiendo al HitTile una dimensión más. De esta forma, obtendríamos un cubo de $10 \times 10 \times 10$ con la distribución de la Figura 2.4 en la dimensiones 0 y 1, y, todo el Shape en la dimensión 2. Así, los Shapes del reparto de la Figura 2.4, suponiendo el mismo número de procesos, serían los mismos que en este caso, solo que, se añadiría (0:9:1) en la tercera dimensión de cada uno.

Para soportar un reparto por una sola dimensión, como los mostrados en la Figura 2.3, la función de creación del Layout contará con un parámetro extra, que será, la dimensión por la que se desea realizar la partición. A la función de creación de Layout se le pasará también como argumento el array de los pesos, que, será aplicado en la dimensión correspondiente. Por otro lado, si se quiere realizar un reparto en múltiples dimensiones, la forma que resulta más sencilla y escalable a n dimensiones, es aplicar el Layout que particiona por una dimensión, tantas veces como dimensiones se quiera repartir por pesos. De esta forma, por ejemplo, en una matriz, se puede realizar un primer reparto por filas, y, dentro de cada nodo, al que se le ha otorgado un Tile diferente, se puede realizar un segundo reparto (las columnas de las filas correspondientes), aplicando un Layout con unos pesos diferentes en una dimensión diferente, a cada subTile de cada nodo, obteniendo así una partición jerárquica y extensible a n dimensiones. En resumen, con un solo Layout se puede realizar todo tipo de particiones regulares, por pesos, unidimensionales y multidimensionales. Para realizar particiones por una sola dimensión basta con aplicar una vez el Layout al Tile original. Por el contrario, para particiones por múltiples dimensiones, se aplica un primer Layout al Tile original y, se siguen aplicando sucesivos Layouts a los subtiles del tile original, a los subtiles de los subtiles y así sucesivamente hasta conseguir particionar todas las dimensiones deseadas.

La interfaz que poseerá la función de creación del Layout **plug_layDimBlocksWeighted** será la siguiente:

```
1 /* plug_layDimBlocksWeighted */
2 typedef struct
3 {
4 int num_procs;
5 float *ratios;
6 } Load_ratios;
7
8 HitLayout hit_layout( plug_layDimBlocksWeighted, HitTopology topo,
      HitShape shape, int restrictDim, Load_ratios weights);
```

Código 2.1: Interfaz del Layout plug_layDimBlocksWeighted

En el Código 2.1 se muestra la interfaz del Layout. A través de Load_ratios, se le indica el número de procesos totales, junto con el peso de cada proceso. Cada posición del array, indica el número de proceso, y, el contenido de la posición, el peso que se le da a dicho proceso. El parámetro restrictDim, indica la dimensión a la que se va a aplicar el Layout por pesos. Dichos pesos, no han de sumar 1 necesariamente. El ratio de carga de cada procesador será siempre $\frac{weights[proc]}{\sum_{i=0}^{procs-1} weights[i]}$, por lo que, se puede usar la escala que se desee. En cuanto al número de procesos, solo se activarán aquellos que se pase como parámetro a Load_ratios, es decir, si se pasamos 4 procesos y el programa se lanza con 8 procesos, estarán activados $[0, 3]$ y desactivados. $[4, 7]$. Si, por el contrario, el número de procesos que se le pasa al Layout es mayor que el número de procesos con los que ha lanzado el programa, se ignorarán los pesos que no pertenezcan a ningún proceso real.

2.1.2. Layout plug_layBlocksWeightedToSelectedDim

En determinadas ocasiones es útil disponer de un Layout que realice una partición por pesos en una dimensión pero reparta las demás dimensiones de forma equitativa. Por ejemplo, en el caso de la diferencia de capacidades de procesamiento entre nodos, necesitamos más carga computacional en unos nodos que en otros pero la carga de los procesos de un mismo nodos puede ser igual para todos los procesos de dicho nodo si solo se trabaja con CPUs en lugar de hacerlo con aceleradores hardware, ya que, todos los núcleos de una CPU, generalmente, tienen la misma capacidad computacional.

Para ello, disponemos del Layout **plug_layBlocksWeightedToSelectedDim** que permite un reparto por pesos en la dimensión seleccionada y, un reparto por bloques equitativo en las dimensiones restantes. Para HitTiles de una sola dimensión, el comportamiento sería el mismo que con el Layout anterior, ya que, se realiza un reparto por pesos en la única dimensión. Este Layout, está pensado sobretodo para realizar repartos multidimensionales sencillos y rápidos en Topologies 2D. Mediante dichas Topologies, se permite realizar repartos jerárquicos, como los ya explicados, debido a que, a cada "fila de procesos" se le puede asignar el mismo SubTile al realizar la primera partición de la primera dimensión. Seguidamente, dentro de de cada "fila de procesos", se permite asignar un diferente Subtile (partiendo por columnas) a cada proceso individual, por pesos, aplicando de nuevo otro Layout. Ver Sección 1.3.2 y [21] para más información acerca de las Topologies.

Por ejemplo, si tenemos una Topology 2D de 4×2 procesadores y, un HitTile de 10×10 elementos y, seleccionamos la dimensión 0 (filas) para el reparto balanceado, con los pesos $[0.3, 0.3, 0.2, 0.2]$ para cada proceso, obtendremos los siguientes elementos:

$$Procesos\ [P_{00}, P_{01}] \rightarrow 0.3 * 1/2 * 100 = 15$$

$$Procesos\ [P_{10}, P_{11}] \rightarrow 0.3 * 1/2 * 100 = 15$$

$$Procesos\ [P_{20}, P_{21}] \rightarrow 0.2 * 1/2 * 100 = 10$$

$$Procesos\ [P_{30}, P_{31}] \rightarrow 0.2 * 1/2 * 100 = 10$$

En la Tabla 2.1.2 podemos ver el Shape exacto que obtendría cada proceso, en el ejemplo:

	0	1
0	(0:2:1,0:4:1)	(0:2:1,5:9:1)
1	(3:5:1,0:4:1)	(3:5:1,5:9:1)
2	(6:7:1,0:4:1)	(6:7:1,5:9:1)
3	(8:9:1,0:4:1)	(8:9:1,5:9:1)

Tabla 2.1: Shape de cada proceso de una Topology 2D de 4×2 procesadores y, un HitTile de 10×10 elementos

En cuanto a la interfaz de este Layout, es similar a la anterior, y los campos necesarios también son iguales. A través de Load_ratios, se le indica el número de procesos totales, junto con el peso de cada proceso en la dimensión seleccionada. El parámetro selectedDimToWeight, indica la dimensión a la que se va a aplicar el Layout por pesos. Dicha interfaz se muestra en el Código 2.2

```
1 /* plug_layDimBlocksWeighted */
2 typedef struct
3 {
4 int num_procs;
5 float *ratios;
6 } Load_ratios;
7
8 HitLayout hit_layout( plug_layBlocksWeightedToSelectedDim, HitTopology
      topo, HitShape shape, int selectedDimToWeight, Load_ratios weights);
```

Código 2.2: Interfaz del Layout plug_layBlocksWeightedToSelectedDim

2.2. Sistema automático de equilibrado de la carga

Como parte adicional del trabajo, se diseña un sistema automático de equilibrado de la carga, es una parte adicional de este trabajo y con ello, se propone ua línea de investigación futura. Se necesita el Layout por pesos diseñado en la primera parte la construcción de un prototipo de sistema automático de equilibrado de la carga. Se dejará como trabajo futuro la implementación de dicho sistema y una futura integración en la librería Controller [33].

El sistema se diseñará para problemas iterativos, es decir, problemas que se resuelven mediante la repetición de un bucle principal un cierto número de iteraciones. Esto se debe a que el equilibrado de carga se realizará cada ciertas iteraciones para garantizar que la explotación de los recursos del sistema heterogéneo es la mejor posible.

El algoritmo que se utilizará para la creación del sistema de ajuste de la carga en tiempo de ejecución se describe a continuación de la siguiente forma:

1. Se parte de una distribución inicial de la carga por pesos que se realizará de forma manual, teniendo en cuenta las capacidades de procesamiento de cada uno de los procesos, dependiendo del nodo al que pertenezcan. Cuanto mejor sea la aproximación manual, menos tiempo se tardará en alcanzar un estado en el que cada proceso obtenga la carga adecuada para lograr una explotación del sistema heterogéneo.

2. En cada una de las iteraciones de la aplicación, cada proceso procederá a realizar una medición del tiempo que tarda en completar la ejecución de la iteración.

3. Cada cierto número de iteraciones (n), todos los procesos, realizarán la media de las últimas n iteraciones ejecutadas. Todos los procesos activos intercambiarán dicha medida y calcularán una nueva distribución de pesos basada en el inverso de los tiempo de ejecución. De tal manera que, los procesos más lentos obtendrán una carga más baja de procesamiento que los más rápidos. Finalmente se redistribuirá el HitTile de nuevo con un Layout basado en los pesos calculados mediante el inverso de los tiempos de ejecución.

Para intercambiarse todos los tiempos medios de ejecución entre los procesos de Hitmap, se utilizará un Tile que almacene en cada posición la media del tiempo de ejecución del proceso que cuyo identificador sea igual a dicha posición del Tile. Para ello se utilizará la operación **All_gather** de hitmap que es similar a la que proporciona MPI en el estándar [34] [17]. En la Figura 2.7 se muestra un esquema de la operación **All_gather**,

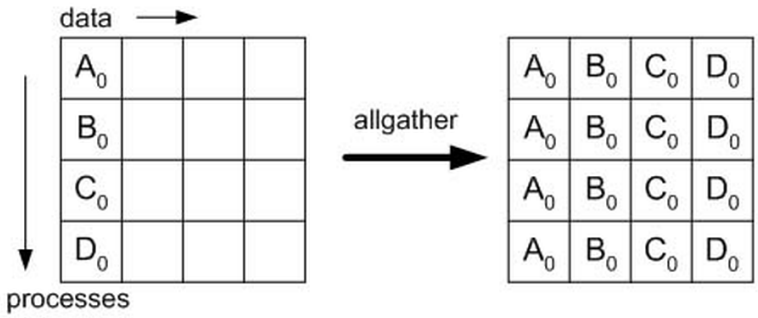

Figura 2.7: Operación **All_gather** de MPI, extraída de [34]

Por medio de este algoritmo se conseguirá redistribuir la carga en tiempo de ejecución de forma automatizada y haciendo uso del particionado por pesos. Debido a que dicha redistribución es costosa, si se parte de una buena distribución de carga se necesitarán menos iteraciones para que converja el equilibrio de la carga. Incluso en un futuro, se podría realizar un cálculo previo a la ejecución teniendo en cuenta las capacidades computacionales de cada uno de los procesos. Sin embargo, debido a que influyen más factores en la velocidad de la ejecución de la aplicación, a parte de las propia capacidad computacional del nodo, como por ejemplo, aquellos relacionados con la red, seguirá siendo necesario realizar ajustes en tiempo de ejecución para balancear la carga entre los nodos.

No obstante y, en contraste con lo que se ha afirmado en el párrafo anterior, este sistema permite no realizar ningún tipo de ajuste antes de comenzar la ejecución de la aplicación y automatizar todo el proceso,

aunque sea algo menos eficiente. Por ello, se puede comenzar con un reparto equitativo e ir realizando ajustes automáticos, si no es posible o no se quiere, realizar un primer ajuste de la carga. De esta forma, si carecemos de la información necesaria para realizar un primer ajuste de la carga, por medio de este algoritmo de ajuste, se conseguirá una explotación del entorno heterogéneo sin conocer las capacidades computacionales del sistema.

En cuanto a la métrica utilizada para realizar equilibrados de carga, consideramos que el tiempo de ejecución es la más sencilla y fiable, ya que, es una media de n iteraciones. Además, en una misma medida se concentran varios factores diferentes que inciden en el rendimiento de un sistema heterogéneo como pueden ser la red, la propia capacidad de cómputo o la ejecución simultánea de otra tareas, como se indica en [15]. Para una primera aproximación, es un buen indicador de la velocidad de ejecución en cada nodo.

Por otra parte, gracias a que Hitmap facilita la redistribución del HitTile original usando otro Layout, es bastante simple realizar una nueva distribución de la carga teniendo un nuevo Layout. Al realizarse cada cierto número de iteraciones, tampoco compromete la eficiencia ganada al realizar un reparto de carga que explote la capacidad del entorno heterogéneo.

2.3. Conclusiones del capítulo

En este capítulo se ha presentado la propuesta del sistema de equilibrado de carga para computación heterogénea:

Por un lado un sistema de particionado de carga para computación heterogénea, que incluye el diseño de dos Layouts integrados en Hitmap, que permiten el reparto por pesos entre los diferentes procesos. Gracias a este sistema de particionado integrado con Hitmap, se permite que la ejecución de las aplicaciones desarrolladas con dicha librería, exploten la capacidad de los sistemas heterogéneos. Este sistema de particionado de carga requiere que se ajusten los pesos que cada proceso debe tener, de forma manual.

Por otro lado, el diseño de un sistema automático de ajuste de la carga, que, sirviéndose del Layout creado y de la librería Hitmap, ajusta los pesos de cada proceso de forma automática en aplicaciones iterativas, permitiendo así, una explotación de un sistema heterogéneo desconocido y, por tanto, la portabilidad de la aplicación a cualquier sistema heterogéneo.

Capítulo 3

Implementación

En este capítulo se detalla el desarrollo y la implementación de la propuesta mostrada en el capítulo anterior (ver Capítulo 2) del plug-in para la librería Hitmap. Los conceptos básicos de Hitmap quedan descritos en el Sección 1.3.1.

El sistema automático de equilibrado de la carga no será posible implementarlo debido al límite temporal del un Trabajo Fin de Grado. Se establecerá como trabajo futuro.

3.1. Creación de un nuevo Layout

En primer lugar, un Layout de Hitmap, se encargan del particionado y mapeado de los datos. Para comprender mejor el funcionamiento de los Layouts, se muestra en el Código 3.1, un ejemplo de un Tile distribuido mediante un Layout, en la librería Hitmap.

```
1  #include <hitmap.h>
2  //Create the new type
3  hit_tileNewType( double );
4
5  int main(int argc, char *argv[])
6  {
7    hit_comInit( &argc, &argv );
8
9    //Create topology
10   HitTopology topo = hit_topology(plug_topPlain);
11
12   //Create Tile of double of 10 elements
13   HitTile_double tile, mySubTile;
14   hit_tileDomain(&tile, double, 1, 10);
15
16   //Extract the shape
17   HitShape shape = hit_tileShape( tile );
18
19   //Compute partition
```

```
20    HitLayout theLayout = hit_layout( plug_layBlocks, topo, shape );
21
22    /* COMPUTATION */
23
24    //Select my shape given by the Layout
25    HitShape myShape = hit_layShape(layout);
26
27    //Create and allocate the subtile with myShape
28    hit_tileSelect( &mySubTile, &tile, myShape );
29    hit_tileAlloc( &mySubTile);
30
31    //COMPUTE THE DATA IN MY SUBTILE
32    int i;
33    for ( i=0; i<hit_tileDimCard( mySubTile, 0 ); i++ )
34    {
35      hit_tileElemAt( mySubTile, 1, i ) = hit_Rank * hit_tileDimCard(
          mySubTile, 0 ) + i ;
36    }
37
38    /* EXPORT TO A FILE  (Omitted) */
39
40    //Free resources
41    hit_tileFree( tile );
42    hit_layFree( theLayout );
43    hit_topFree( topo );
44    hit_comFinalize();
45    return 0;
46  }
```

Código 3.1: Ejemplo de uso de Layouts en la librería Hitmap

Como se puede observar en el Código 3.1, tras crear la Topology y los Tiles, se construye un Layout a partir de la Topology y el Shape del HitTile que se va a particionar. La función *hit_layout* nos proporciona el Layout de tipo HitLayout. Seguidamente, a través de la función *hit_layShape*, cada proceso obtiene el Shape de su parte del Tile. Seleccionando, mediante la función *hit_tileSelect*, cada uno de los procesos, su Shape del Tile original, obtiene, cada uno, su respectivo subTile. De esta forma, cada uno de los procesos trabaja sobre su parte del Tile, y, el Layout se encarga de entregar a cada proceso su parte correspondiente del Tile original. En este caso, la parte de computación de la aplicación es muy sencilla, ya que, se trata de un ejemplo. Simplemente, cada proceso calcula, para cada elemento del Tile, el índice que representa en el Tile original y lo guarda en dicha posición. La parte de escritura del Tile en un fichero la hemos omitido, ya que, tampoco es de interés para el propósito de este trabajo. Por último, se liberan todos los recursos que ha utilizado la aplicación y se finaliza.

Como se ha visto en el Código 3.1, para crear nuevos Layouts se utiliza la función *hit_layout*. Para crear un Layout, se necesita, el nombre del Layout y la Topology, como mínimo. El resto de los parámetros dependen del propio Layout que se elija, aunque, en la mayor parte de los Layouts, suele ser necesario indicar también el Shape del Tile que se va a repartir. En el Código 3.2 se muestra, en primer lugar, el

constructor de los Layouts, realmente es una macro de C [35], no una función. Dicha macro es sustituida por la rutina de creación de cada Layout, en función del nombre que se le entregue a la macro como primer argumento. Más abajo, en el Código 3.2, encontramos el prototipo de la función de creación del Layout Blocks (ver Sección 1.3.3). Por tanto, para desarrollar un nuevo Layout en Hitmap, lo primero que necesitamos es la función de creación del Layout, para que esta, sea reconocida por la macro, y, se pueda utilizar el Layout en la librería Hitmap.

```
1  /* 4. LAYOUT CONSTRUCTORS AND PREDEFINED LAYOUT FUNCTIONS */
2  /**
3  * Hit layout constructor macro
4  */
5  #define hit_layout( name, topo, ...)  hit_layout_##name( topo,
       __VA_ARGS__ )
6
7  /* 4.1 LAYOUT SIGNATURE CREATION FUNCTIONS */
8  /* 4.1.1 BLOCKS */
9  /**
10 * Hit Layout Blocks constructor function
11 * This layout leave the inactive processor at the end of the dimension
12 * when there are more processor than data.
13 * @param topo The topology
14 * @param shape The global shape
15 */
16 HitLayout hit_layout_plug_layBlocks(HitTopology topo, HitShape shape);
```

Código 3.2: Constructor de Layouts, extraído de hitmap/include/hit_layout.h

Un HitLayout, es un tipo de datos (un Struct de C, realmente) que almacena toda la información necesaria para realizar el particionado de la carga en la librería. Dentro de la función que construye el Layout, realmente, se necesita rellenar la estructura HitLayout con toda la información necesaria para que se puedan utilizar, con dicho Layout, todas las funciones y propias de los Layouts en la librería Hitmap. Realmente, en Hitmap, existe una función que facilita el trabajo de creación de la estructura HitLayout, el *hit_layout_wrapper*. Por medio de dicha función, se simplifica bastante el trabajo de creación del Layout, ya que, realiza todo el trabajo de creación de la estructura pasándole por los argumentos, ciertas parámetros necesarios para realizar los repartos de la carga. *hit_layout_wrapper* devuelve un Layout que es prácticamente ya utilizable. La única información que hay que proveer a mayores es aquella referente al grupo de procesos y el líder. En el Código 3.3 se muestra la cabecera de la función *hit_layout_wrapper*.

```
1
2  /* 10.7. INTERNAL WRAPPER: DO LAYOUT FOR ONE OR ALL DIMENSIONS */
3  HitLayout hit_layout_wrapper( HitTopology topo,
4  HitShape shape,
5  HitLayoutSignatureFunction signatureGenericF,
```

```
6  HitLayoutSignatureInvFunction signatureInvGenericF,
7  HitLayoutSignatureFunction signatureRestrictedF,
8  HitLayoutSignatureInvFunction signatureInvRestrictedF,
9  HitLayoutRanksFunction ranksGenericF,
10 HitLayoutRanksFunction ranksRestrictedF,
11 HitLayoutSignatureMaxCardFunction maxCardGenericF,
12 HitLayoutSignatureMaxCardFunction maxCardRestrictedF,
13 HitLayoutSignatureMinCardFunction minCardGenericF,
14 HitLayoutSignatureMinCardFunction minCardRestrictedF,
15 HitLayoutSignatureNumActivesFunction activesGenericF,
16 HitLayoutSignatureNumActivesFunction activesRestrictedF,
17 float* extraParameter,
18 int restrictToDim
19 );
```

Código 3.3: Prototipo de la función hit_layout_wrapper , extraído de hitmap/src/hit_layout.c

A parte del Shape, restrictedDim y un parámetro extra, los argumentos restantes son funciones. Dichas funciones se pueden dividir en dos clases distintas: las Generic y las Restricted. Si el Layout realiza la misma partición en todas las dimensiones, entonces, las funciones de tipo Restricted no son necesarias, pues e utilizan solo las Generic. En cambio, si el Layout realiza un reparto diferente en una dimensión que en las demás, entonces, debemos pasar las funciones de dicho reparto como Restricted , y, las funciones que realizan los repartos en las dimensiones restantes, como Generic. En los Layouts que vamos a implementar, necesitamos utilizar la dimensión restringida en ambos, ya que, el reparto por pesos solo se hará en una dimensión. Sin embargo, en las dimensiones restantes, se copiará el contenido en todos los procesos, en el primer Layout, o, se realizará un reparto por bloques tradicional, en el segundo Layout (ver Capítulo 2).

Por otro lado, las funciones que se le pasan a *hit_layout_wrapper* son de seis tipos diferentes, ya sean Generic o Restricted. Con dicho conjunto de seis funciones, se puede construir un nuevo Layout a través del wrapper, ya que, por medio de ellas, el wrapper obtiene toda la funcionalidad necesaria para crear un nuevo sistema particionado de datos. A continuación, se detalla cada uno de los tipos de funciones mencionados.

- *HitLayoutSignatureFunction*: Tipo de función que calcula la Signature que le corresponde al proceso (porcId) en una dimensión. En la línea 1 de Código 3.4, se muestra la definición del tipo de la función. **procId** indica el identificador del proceso en la dimensión en la que se aplica la función; **procsCard** el número total de procesos en dicha dimensión; *blocksCard* el número total de bloques en dicha dimensión; *extraParam* el parámetro extra que se le pasó al wrapper; *input* la Signature completa de la dimensión y, *res*, la Signature que devuelve la función. La dimensión de los procesos solamente tiene sentido si la Topology de los procesadores tiene más de una dimensión, en ese caso, la primera dimensión del Tile se enviará a los procesadores de la primera dimensión, la segunda a los segundos y así sucesivamente. En caso contrario, tanto procId como procsCard serán iguales para todas las dimensiones del Tile.

- *HitLayoutSignatureInvFunction*: Tipo de función que calcula el inverso de la Signature, es decir, devuelve el número de proceso correspondiente, dado un índice de la Signature original. Los cinco primeros parámetros son los mismo que en el anterior tipo de función. En este caso, el **procId** no

suele realizar ninguna función, ya que, se calcula el procId del parámetro **ind** de la Signature original. El sexto parámetro **ind**, indica, como ya se ha comentado, el índice de la Signature que se va a calcular a que proceso pertenece. El número de proceso calculado se devolverá en el retorno de la función.Esta función se encuentra en la línea 9 de Código 3.4.

- *HitLayoutRanksFunction*: En la línea 17 de Código 3.4, se presenta este tipo de función. Transforma el id del proceso dado, del identificador de la Topology al identificador de los procesos activos y viceversa. Prácticamente todos los parámetros que tiene se han explicado en los tipos anteriores excepto *topoActiveMode* que indica el modo de funcionamiento de la función. Devuelve el identificador al que se ha transformado, dado el identificador origen en **procId**.

- *HitLayoutSignatureMaxCardFunction*: Este tipo de función, devuelve el máximo número de bloques que ha asignado a alguno de los procesos. Se muestra en la línea 26 de Código 3.4 y todos los parámetros que se le pasan, son los mismos que los de las otras funciones, ya explicadas.

- *HitLayoutSignatureMinCardFunction*: Este tipo de función, devuelve el mínimo número de bloques que ha asignado a alguno de los procesos. Se muestra en la línea 31 de Código 3.4 y todos los parámetros que se le pasan, son los mismos que los de las otras funciones, ya explicadas.

- *HitLayoutSignatureNumActivesFunction*: Este tipo de función, devuelve el número de procesos activos que hay en la dimensión. Se muestra en la línea 36 de Código 3.4 y todos los parámetros que se le pasan, son los mismos que los de las otras funciones, ya explicadas.

```
1  // HitLayoutSignatureFunction
2  typedef int (*HitLayoutSignatureFunction)(  int procId,
3  int procsCard,
4  int blocksCard,
5  float* extraParam,
6  HitSig input,
7  HitSig *res );
8
9  // HitLayoutSignatureInvFunction
10 typedef int (*HitLayoutSignatureInvFunction)( int procId,
11 int procsCard,
12 int blocksCard,
13 float* extraParam,
14 HitSig input,
15 int ind );
16
17 // HitLayoutRanksFunction
18 #define HIT_LAY_RANKS_ACTIVE_TO_TOPO  0
19 #define HIT_LAY_RANKS_TOPO_TO_ACTIVE  1
20 typedef int (*HitLayoutRanksFunction)(    char topoActiveMode,
21 int procId,
22 int procsCard,
23 int blocksCard,
```

```
24 float* extraParam );
25
26 // HitLayoutSignatureMaxCardFunction
27 typedef int (*HitLayoutSignatureMaxCardFunction)( int procsCard,
28 int blocksCard,
29 float* extraParameter );
30
31 // HitLayoutSignatureMinCardFunction
32 typedef int (*HitLayoutSignatureMinCardFunction)( int procsCard,
33 int blocksCard,
34 float* extraParameter );
35
36 // HitLayoutSignatureNumActivesFunction
37 typedef int (*HitLayoutSignatureNumActivesFunction)(  int procsCard,
38 int blocksCard,
39 float* extraParameter );
```

Código 3.4: Tipos de funciones del wrapper, extraído de hitmap/include/hit_layout.h

Por medio del *hit_layout_wrapper*, se consigue estandarizar el proceso de creación de nuevos Layouts. Sin embargo, la funcionalidad hay que diseñarla e implementarla igualmente, ya que, la forma de calcular las partes del Shape que le tocan a cada proceso es a través de estas funciones. Aún así, es más portable y sencillo implementar nuevas funciones que rellenar un struct a mano, por ello, el trabajo se simplifica.

En resumen, para añadir un nuevo Layout en la librería Hitmap, es necesario, fundamentalmente, lo siguiente:

- Diseñar e implementar una función principal que sirva como constructor para el Layout, que llame a *hit_layout_wrapper*, rellene la información correspondiente al grupo y al líder, y, realice todas las operaciones auxiliares necesarias para realizar el reparto.
- Diseñar e implementar todas las funciones necesarias para llamar a la función *hit_layout_wrapper*. Puede que algunas de las funciones que se utilicen estén ya implementadas para otros Layouts y se puedan reutilizar.

3.2. Layout plug_layDimBlocksWeighted

En primer lugar, para la creación de este nuevo Layout, se ha diseñado la interfaz para la función que hace de "constructor" del Layout. Dicha interfaz ya fue mostrada en el Capítulo 2, Sección 2.1.1. En este caso se mostrará directamente el prototipo de la función constructor del Layout. Creemos que es importante volverlo a mostrar aquí, ya que, se explicará y se comentará sobre ella en este capítulo.

```
1 /* plug_layDimBlocksWeighted */
2 typedef struct
3 {
```

```
4 int num_procs;
5 float *ratios;
6 } Load_ratios;
7
8 HitLayout hit_layout_plug_layDimBlocksWeighted(HitTopology topo,
      HitShape shape, int restrictDim, Load_ratios l_ratios)
```

Código 3.5: Constructor del Layout plug_layDimBlocksWeighted

Como se muestra en el Código 3.5, como parámetros del Layout, se necesita a parte de la Topolgy, el Shape, la dimensión restringida o seleccionada y los pesos. El Shape es fundamental, ya que, se repartirá por pesos entre los procesos. El parámetro restrictedDim, indica en que dimensión se aplicará el Layout, en caso de ser un Tile multidimensional (ver Sección 2.1.1 para una explicación más detallada). Por último, la estructura Load_ratios contiene el array de los pesos (ratios), indexado por el identificador del proceso y un parámetro, num_procs, para indicar el número de procesos totales que se pasará en el array. Por medio de estos argumentos, tenemos toda la información necesaria para realizar las particiones de los datos, por tanto, se puede repartir el Shape entre los procesos teniendo un cuenta el peso que cada proceso debe tener sobre el Shape original.

El siguiente paso que se debe seguir para construir el Layout es implementar la función que mostramos en el Código 3.5. Como se expuso en la subsección anterior (Sección 3.1), para crear la estructura final del Layout no se realiza de forma manual, sino que se dispone de un wrapper que devuelve prácticamente el Layout final, invocándole con ciertas funciones como argumentos. En el Código 3.5 se mostraba la cabecera de *hit_layout_wrapper*. Como ya se explicó, en el wrapper, existen una lista de funciones Generic y otras Restricted. Las Restricted se aplicaban en una dimensión, la seleccionada, y las Generic en todas las demás. En nuestro caso, queremos que cada vez que se aplique el Layout, lo haga solamente una dimensión, por ello, necesitaremos pasar las funciones del reparto por pesos como Restricted, y, pasar como Generic, funciones copia que otorguen a todos los procesos, todos los elementos de las dimensiones restantes.

Los seis tipos básicos de funciones, también se explicaron en la subsección anterior (Sección 3.1) y se pueden visualizar en Código 3.4. Con dicho conjunto de seis funciones, se puede construir un nuevo Layout, ya que, realizan todos los cálculos necesarios para repartir la carga. Nosotros debemos construir las seis funciones Restricted, que repartirán por pesos y las seis funciones Generic que simplemente copiarán todos los elementos en todos los procesadores, en las restantes dimensiones. Las funciones de copia están ya implementadas, debido a que, ya existen otros Layouts como *plug_layDimBlocks* que reparten por una sola dimensión y en las demás copian.

Antes de realizar la llamada al wrapper, es necesario decidir cómo se le indicará a las funciones que realizan los repartos por pesos, cuáles son dichos pesos. Para ello, existe un *extraParam* que una vez pasado al wrapper, todas las funciones de reparto, obtienen dicho parámetro como uno de sus argumentos. En el *extraParam* almacenaremos un puntero a una estructura de tipo *Load_ratios* que contendrá dicha información. Es importante recalcar que para facilitar el cálculo de la Signature de cada proceso, los pesos que se almacenarán en dicha estructura no serán los mismos que los que se pasan como argumento a la función "constructor". Por el contrario, en este caso, el array de pesos, almacenará el ratio de ocupación

de la estructura que ha sido utilizada por los procesos que preceden al actual, es decir, la suma de ratios hasta el actual. De esta forma, si se llama a la función constructor con los pesos $[0.3, 0.3, 0.3, 0.1]$, en *extraParam* se almacenará $[0, 0.3, 0.6, 0.9, 1]$. Así, conseguiremos una mayor velocidad al calcular dónde empieza y dónde acaba la Signature correspondiente a cada proceso.

Comenzaremos mostrando los detalles de implementación más importantes de las seis funciones Restricted necesarias para realizar el reparto por pesos en la dimensión seleccionada. Seguidamente, mostraremos las funciones Generic. Por último, presentaremos la implementación final de la función constructor.

HitLayoutSignatureFunction signatureRestrictedF

La función recibe el nombre de *hit_layout_plug_layDimBlocksWeighted_Sig* y se encuentra implementada en hitmap/src/hit_layout.c. Recibe como parámetros los pesos, el número de bloques, el de procesadores y la Signature completa, y calcula la Signature que le corresponde al proceso *procId* con el que se la invoca. Dado un proceso procId, la Signature (begin:end:stride) que le a corresponde a dicho proceso se calcula mediante:

$$begin = \left\lfloor \frac{weights[procId]}{totalWeigh} * blocksCard \right\rfloor * input.stride + input.begin$$

$$end = \left(\left\lfloor \frac{weights[procId+1]}{totalWeigh} * blocksCard \right\rfloor - 1 \right) * input.stride + input.begin$$

$$stride = input.stride$$

Siendo *input* la Signature completa a repartir, *weights* el array de pesos, que contiene en cada posición, el peso acumulado de la suma de los pesos de los procesos anteriores, *totalWeigh* la suma de todos los pesos y *procId* el proceso del que se calcula la Signature.

De esta forma, todos los procesos obtienen una Signature con $card(Signature)$ proporcional al peso elegido al construir el Layout, ya que weights[i+1] - weights[i] = oldWeights[i], siendo *oldWeights*, los pesos que el usuario ha introducido como parámetros en la construcción del Layout. Por ello, cada proceso obtendrá el porcentaje de carga que se le introdujo. Si el peso de la carga multiplicado por el total de elementos no es número real, entonces, se ajustarán los restos, dotando con un elemento más a alguno de los procesos de forma completamente automática, simplemente utilizando las fórmulas expuestas. Cuando el proceso siguiente obtenga un número real como end, a ese proceso, se le otorgará un elemento más. En caso de que todos los begin y end intermedios sean decimales, el proceso que obtendrá el resto de más será el último y el proceso que lo perderá será el primero, todos los del medio se irán compensando; los decimales del begin que se ganan con los del end que se pierden. Se multiplica por el stride y se suma el begin de la Signature original tanto en begin como en end para conseguir una porción de la Signature con el mismo stride y begin que el original, ya que, si el Shpae inicial comienza en x, habrá que sumar x a todo, o, si va de y en y elementos, habrá que multiplicar por y para conseguir el fragmento de la la

Signature con el mismo stride y begin que el original.

HitLayoutSignatureInvFunction signatureInvRestrictedF

En este caso la función recibe el nombre de *hit_layout_plug_layDimBlocksWeighted_SigInv* y también se encuentra implementada en hitmap/src/hit_layout.c. Por medio de dicha función, dado un índice de la Signature, se obtiene el proceso al que hace referencia. Debido a que la complejidad de esta función es mayor, se mostrará la implementación de dicha función, a continuación, en el Código 3.6 y se explicará el algoritmo que se ha utilizado.

```
1  int hit_layout_plug_layDimBlocksWeighted_SigInv(int procId, int
       procsCard, int blocksCard,float *extraParameter, HitSig input, int
       ind)
2  {
3    HIT_NOT_USED(procId);
4    HIT_NOT_USED(procsCard);
5
6    /* CHECK: THE INDEX SHOULD BE IN THE INPUT DOMAIN */
7    if (!hit_sigIn(input, ind))
8      return HIT_RANK_NULL;
9
10   /* Load ratios */
11   Load_ratios *load_ratios = (Load_ratios *)extraParameter;
12
13   float *weights = load_ratios->ratios;
14
15   /* 1. CALCULATE WEIGHT */
16   int tileInd = (ind - input.begin) / input.stride;
17   float weightAprox = (float)tileInd * weights[load_ratios->num_procs]
       / (float)blocksCard;
18
19   /* 2. FIND PROC: BIN-LIKE SEARCH IN ARRAY OF WEIGHTS (SORTED)*/
20   int l = 0, r = load_ratios->num_procs - 1, c;
21   int proc = 0;
22
23   while (l <= r)
24   {
25     c = (l + r) / 2;
26     proc = c;
27
28     if (weightAprox >= weights[c] && weightAprox < weights[c + 1])
29       break;
30
31     if (weightAprox < weights[c])
32       r = c - 1;
33     else
```

```
34          l = c + 1;
35      }
36
37      /* SKIP PROCS WITH 0 BLOCKS */
38      int i = proc + 1;
39      while (i < load_ratios->num_procs)
40      {
41
42        int begin =(int)(weights[i] / weights[load_ratios->num_procs] * (
              float)blocksCard);
43        int end = ((int)(weights[i + 1] / weights[load_ratios->num_procs] *
              (float)blocksCard)) - 1;
44
45        if (end - begin >= 0)
46          break;
47        i++;
48      }
49
50      /* 4. RETURN PROC */
51      /* IF PRECISION WAS LOST AND THE INDEX IS AT THE BEGINNING OF THE
           NEXT PROC(NOT EMPTY) */
52      if ((int)(weights[i]/ weights[load_ratios->num_procs] * (float)
           blocksCard ) == tileInd)
53        return i;
54      else
55        return proc;
56  }
```

Código 3.6: hit_layout_plug_layDimBlocksWeighted_SigInv, extraído de hitmap/src/hit_layout.c

Como se puede observar en los comentarios explicativos de la función en el Código 3.6, la función se divide en tres partes, principalmente. Primero, se calcula el peso que le correspondería a dicho índice; En segundo lugar, se busca a que proceso pertenece; Por último, se devuelve el proceso al que pertenece finalmente el índice.

La primera parte consiste en calcular la fracción del bloque en la que se halla el índice que se ha proporcionado a la función, ya que, en realidad, solo necesitamos el peso p para después realizar una búsqueda en el array de pesos.

$$ind = \left\lfloor \frac{p}{totalWeight} * blocksCard \right\rfloor * input.stride + input.begin$$

$$\frac{ind - input.begin}{input.stride} \approx \frac{p}{totalWeight} * blocksCard$$

$$p \approx \frac{ind - input.begin}{input.stride * blocksCard} * totalWeight \tag{3.1}$$

Tras averiguar p, es necesario saber que proceso (i) tiene la Signature cuyo peso de inicio (o peso con el que se calcula el inicio) es mayor o igual que p y cuyo peso de fin es menor que p, es decir:

$$weights[i] \leq p < weights[i+1]$$

Para conocer el valor de i, se ha implementado un algoritmo de búsqueda basado en la búsqueda binaria (ver [36], sección 6.2.1) en el array de los pesos, ya que, dicho array, al contener los pesos acumulados, realmente tiene los pesos ordenados y es bastante más eficiente, encontrar entre que dos pesos acumulados, se halla p. Podemos encontrar la búsqueda del proceso entre la línea 19 y la línea 35 del Código 3.6. Cuando encuentra entre que dos pares inicio-fin está el peso p, termina. En el caso de que $p > maxCard$ o existiese algún error, se detendría la búsqueda cuando el pivote izquierdo es mayor que el derecho.

En la última parte de la función, debido a que p no se puede calcular de forma exacta, pues al aplicar la función *floor* sobre los índices del Tile, se pierde la precisión decimal, es necesario realizar ciertas comprobaciones y ajustes para comprobar que realmente se ha calculado correctamente a qué proceso pertenece el índice dado. Cuando se obtiene p mediante la Ecuación 3.1, puede ocurrir que el p sea menor que el que debería ser si a *ind* se le aplicó la función $floor$ para obtener el índice como un número entero. En ese caso, si *ind* es el extremo inferior de la Signature de un proceso i, al calcular p mediante la Ecuación 3.1, p resulta en un peso menor al del extremo inferior de i y, por tanto, cuando se realiza la búsqueda del proceso correspondiente en el array de pesos, se obtiene $i-1$ como proceso (si tiene 1 elemento o más, en caso contrario, se obtendrá $i-(n+1)$ siendo n el número de procesos vacío) cuando en realidad es i el proceso al que pertenece *ind*.

Sabemos que se obtiene el primer proceso anterior en tener un bloque o más al que se debería obtener, ya que, al eliminar la parte decimal de *ind*, puede dar como resultado, utilizando la Ecuación 3.1, un p_i menor al que tiene el proceso i pero nunca podrá ser menor que el que posee $i-1$ como primer peso, teniendo en cuenta que $i-1$ tiene un bloque o más. Para que $i-1$ tenga un bloque o más, necesariamente el begin de la Signature de $i-1$ debe ser por lo menos una unidad menor que el begin de la Signature de i. Esto provoca que aunque se pierda precisión con el begin de i al aplicar *floor*, siempre deberá ser estrictamente mayor que el begin de $i-1$, pues $i-1$ tiene como mínimo un bloque, por ello, ya que, $\lfloor S_i.begin \rfloor > S_{i-1}.begin \geq \lfloor S_{i-1}.begin \rfloor$, aunque el p_i sea menor por la pérdida de precisión, siempre deberá ser mayor que p_{i-1}. Cuando se sustituye en la Ecuación 3.1, al cumplirse $\lfloor S_i.begin \rfloor > S_{i-1}.begin \geq \lfloor S_{i-1}.begin \rfloor$, se obtiene que $p_i > p_{i-1}$ y, por tanto solo se puede saltar al elemento del procesador con un bloque o más, anterior, por la pérdida de precisión.

Por ello en línea 37 del Código 3.6 se salta a los procesos siguientes con 0 bloques, ya que, si ocurre el caso del extremo inferior, que acabamos de mencionar el *ind*, pertenecería al primer proceso con 1 bloque o más, después del proceso actual. Finalmente, se comprueba en la línea 52 de Código 3.6 si al aplicar la fórmula para obtener el primer elemento de la Signature del siguiente proceso con 1 bloque, se obtiene exactamente *ind*. En ese caso habrá ocurrido lo que acabamos de explicar y, por tanto, se devuelve el proceso correspondiente. En caso contrario, se devuelve el proceso encontrado.

HitLayoutRanksFunction ranksRestrictedF

Se encuentra definida e implementada en hitmap/src/hit_layout.c y recibe el nombre de *hit_layout_plug_layDimBlocksWeighted_ranks* Utilizando esta función, se pueden obtener dos funcionalidades. Por un lado, pasar del identificador del proceso original en la Topology al identificador de los procesos activos. Por otro lado realizar la conversión contraria. Para indicar el modo en el que se utiliza la función, en el primer parámetro se indica la constante correspondiente.

Los procesos de la Topology son todos los procesos con los que se ha ejecutado la aplicación. Para realizar algunas operaciones como las comunicaciones colectivas, es deseable tener un segundo tipo de identificador que salte a los procesos inactivos y no los incluya. De esta forma la numeración de procesos activos en sencilla; el primer proceso activo en el orden de la Topology, obtendrá el identificador 0, el segundo proceso activo el 1 y así sucesivamente. Así, si todos los procesadores están activos, ambas formas de numeración de procesos coincidirán.

La conversión de procesos de la Topology a activos se realiza, como se muestra en el fragmento del Código 3.7 perteneciente a la función original.

```
1  int i = 0, num_proc = -1, begin, end;
2  for (i = 0; i <= procId; i++)
3  {
4    begin = (int) (weights[i] / weights[load_ratios->num_procs] * (float)
         blocksCard);
5    end = ((int)(weights[i + 1] / weights[load_ratios->num_procs] * (
         float)blocksCard)) - 1;
6
7    if (end - begin >= 0)
8      num_proc++;
9  }
10
11 if (end - begin < 0)
12   return HIT_RANK_NULL;
13 else
14   return num_proc;
```

Código 3.7: Conversión de procesos de la Topology a activo

Se recorren todos los procesos de la Topology hasta el número de proceso que se indicado en el parámetro *procId*, aumentando el contador *num_proc*, cuando se encuentra un proceso activo de la Topology. De esta forma, si todos los procesos hasta el indicado están activos, el contador se incrementará en todas las iteraciones y se obtendrá el mismo número que se introdujo. En caso de que el proceso de la Topology que se ha indicado, no esté activo, se comprueba en la última condición y se devuelve NULL.

Por otro lado, la conversión de procesos activos de la Topology a activos, se realiza como se muestra en

el fragmento del Código 3.8, perteneciente, también a la fución original implementada.

```
1  int i = 0, num_proc = -1, begin, end;
2  while (num_proc < procId && i < load_ratios->num_procs)
3  {
4    begin = (int) (weights[i] / weights[load_ratios->num_procs] * (float)
       blocksCard);
5    end = ((int)(weights[i + 1] / weights[load_ratios->num_procs] * (
       float)blocksCard)) - 1;
6
7    if (end - begin >= 0)
8    num_proc++;
9    i++;
10 }
11
12 i--; /* Restamos uno porque queremos obtener el identificador del ú
       ltimo proceso de la topology que se comprobó antes de salir y como
       se suma uno al final para la siguiente iteración, deshacemos dicha
       suma. */
13
14 if (num_proc == procId)
15   return i;
16 else
17   return HIT_RANK_NULL;
18 }
```

Código 3.8: Conversión de procesos de activos a la Topology

Se recorre la lista de procesos de la Topology o bien, al completo, si no se encuentra el proceso de la Topology , o bien, hasta que se encuentra el proceso de la Topology. Para cada proceso de la Topology. se comprueba si es activo y, si es así, se suma uno a *num_proc*. Cuando *num_proc* es igual al proceso a convertir, hemos encontrado en i, el último número de proceso de la Topology para llegar a *num_proc*, es decir, el proceso de la Topology al que pertenece. En la línea 12 se explica por qué se resta 1 a i y, por último se devuelve el proceso de la Topology si es que se ha encontrado y, en caso contrario, se devuelve NULL.

HitLayoutSignatureMaxCardFunction maxCardRestrictedF

La función recibe el nombre de *hit_layout_plug_layDimBlocksWeighted_maxCard* y se encuentra implementada en hitmap/src/hit_layout.c. La función tiene que devolver el mayor número de bloques que le ha asignado a los procesos. En este caso, como se trata de una Layout por pesos, aquellos con mayores pesos son los que se llevan mayor número de bloques. La implementación resulta sencilla, simplemente, recorriendo todos los procesos, se selecciona aquel que tiene una mayor carga y, se devuelve dicha carga. Hemos decidido mostrar una simplificación de la implementación de la función en el Código 3.9 para

mostrar un pequeño detalle de eficiencia que se ha tenido en cuenta.

```
1  int hit_layout_plug_layDimBlocksWeighted_maxCard(int procsCard, int
       blocksCard, float *extraParameter)
2  {
3    HIT_NOT_USED(procsCard);
4  \textbf{hit\_layout\_plug\_layCopy\_Sig}
5    /* Load ratios */
6    Load_ratios *load_ratios = (Load_ratios *)extraParameter;
7
8    float threshold = load_ratios->ratios[load_ratios->num_procs] /
         blocksCard;
9    int i, proc = 0;
10   float max = 0;
11
12   /* Find the maximum ratio (max Card) */
13   for (i = 0; i < load_ratios->num_procs; i++)
14   {
15     if (load_ratios->ratios[i + 1] - load_ratios->ratios[i] + threshold
           >= max)
16     {
17       // SIMPLIFICADO
18       if(numBloques(i) > numBloques(proc))
19       {
20         max = load_ratios->ratios[i + 1] - load_ratios->ratios[i];
21         proc = i;
22       }
23     }
24   }
25   // SIMPLIFICADO
26   return numBloques(proc);
27 }
```

Código 3.9: Implementación simplificada de *hit_layout_plug_layDimBlocksWeighted_maxCard*, extraído de hitmap/src/hit_layout.c

En la función, se busca el proceso con mayor peso, para evitar calcular el número de bloques que tiene cada uno y seleccionar el más grande. La ventaja de realizar la búsqueda de ese modo es que ahorramos todo el cálculo de los *card* de bloques de todos los procesos. El problema reside en que no siempre el proceso que tiene el peso mayor es el que más carga tiene. Puede darse el caso de que

$\exists p_i \in P \mid p_i < p_{max}$, siendo P el vector de los pesos y p_{max} el mayor de los pesos del vector

para el que se cumpla que

$p_{max} * blocksCard - p_i * blocksCard < 1$

en el que p_i ha recibido uno de los restos, al no resultar todas la operaciones $p * blockCard$ en un número entero de bloques y p_{max} no ha recibido ningún resto y, por tanto, tener más bloques con p_i que con p_{max} , siendo p_{max} mayor. Para tratar todos estos casos utilizamos un umbral (*threshold*) para comprobar también los pesos que no son mayores que el actual pero en los que se cumple $p_{max} * blocksCard - p_n * blocksCard < 1$ y por un ajuste de restos, sí podría resultar mayor el número de bloques de un proceso que tiene algo menos de peso, siempre y cuando la diferencia de pesos sea menor que el umbral establecido, esto es, el peso que representa 1 bloque. De esta forma solo comprobamos el número de bloques de aquellos procesos que sí podrían tener un *card* mayor que el actual pero no de todos, ganando así en eficiencia. En el Código 3.9 está simplificada la comparación del número de bloques para facilitar la explicación.

HitLayoutSignatureMaxCardFunction minCardRestrictedF

La función recibe el nombre de *hit_layout_plug_layDimBlocksWeighted_minCard* y se encuentra implementada en hitmap/src/hit_layout.c. La función tiene que devolver el menor número de bloques que le ha asignado a los procesos. La implementación para esta función es similar a la que se ha relazado para la función anterior (*hit_layout_plug_layDimBlocksWeighted_maxCard*), solo que hay que buscar el número de bloques inferior. El algoritmo es el mismo, solo que en este caso, buscamos el mínimo y, por tanto, tenemos en cuenta lo mismo que en el caso anterior, que pueden existir procesos con algo más de peso pero menos carga, y, por ello, hacemos también uso del umbral para evitar calcular el número de bloques en todos los casos, asegurando que obtenemos el menor número de bloques. Una simplificación de la implementación se muestra en el Código 3.10.

```
1  int hit_layout_plug_layDimBlocksWeighted_maxCard(int procsCard, int
       blocksCard, float *extraParameter)
2  {
3    HIT_NOT_USED(procsCard);
4
5    /* Load ratios */
6    Load_ratios *load_ratios = (Load_ratios *)extraParameter;
7
8    if (load_ratios->num_procs <= 0)
9      return 0;
10
11   float threshold = load_ratios->ratios[load_ratios->num_procs] /
         blocksCard;
12   int i, proc = 0;
13   float min = load_ratios->ratios[i + 1] - load_ratios->ratios[i];
14
15   /* Find the maximum ratio (max Card) */
16   for (i = 1; i < load_ratios->num_procs; i++)
17   {
18     if (load_ratios->ratios[i + 1] - load_ratios->ratios[i] - threshold
```

```
            <= min)
19      {
20        // SIMPLIFICADO
21        if(numBloques(i) < numBloques(proc))
22        {
23          min = load_ratios->ratios[i + 1] - load_ratios->ratios[i];
24          proc = i;
25        }
26      }
27    }
28    // SIMPLIFICADO
29    return numBloques(proc);
30  }
```

Código 3.10: Implementación simplificada de *hit_layout_plug_layDimBlocksWeighted_minCard*, extraído de hitmap/src/hit_layout.c

Como se puede apreciar, el código es prácticamente idéntico al de la función anterior, ya que, realmente, el algoritmo que se utiliza es el mismo.

HitLayoutSignatureNumActivesFunction activesRestrictedF

Esta función se encuentra definida e implementada en hitmap/src/hit_layout.c y recibe el nombre de *hit_layout_plug_layDimBlocksWeighted_numActives*. La función devuelve el número de procesos activos. La implementación de esta función es sencilla, simplemente recorre todos los procesos comprobando si están activos o no. Para que estén activos deben tener al menos 1 bloque del Tile, es decir, el $card(S)$, siendo S la Signature local del Proceso, debe ser mayor o igual que 1. Para calcular el $card(S)$, simplemente recurrimos a realizar la siguiente operación $end - begin + 1$ (Ecuación 3.2), siendo *begin* y *end*, el inicio y fin del bloque de cada proceso, indexado de $(0..n-1)$, con Stride 1. No necesitamos utilizar la Signature original, ya que, el número de bloques va a ser el mismo en realidad, por ello, simplificamos el cálculo y optimizamos la función.

$$begin = \left\lfloor \frac{weights[procId]}{totalWeigh} * blocksCard \right\rfloor$$

$$end = \left\lfloor \frac{weights[procId+1]}{totalWeigh} * blocksCard \right\rfloor - 1$$

$$\left\lfloor \frac{weights[procId+1]}{totalWeigh} * blocksCard \right\rfloor - \left\lfloor \frac{weights[procId]}{totalWeigh} * blocksCard \right\rfloor \tag{3.2}$$

La fórmula final se muestra en la Ecuación 3.2. Si el resultado es mayor que 0, entonces el proceso está activo, en caso contrario, no está desactivado.

Funciones Generic

Las funciones Generic que se deben proporcionar al wrapper son de los seis mismos tipos que se han mostrado en el caso de las funciones Restricted. Estas últimas sirven para repartir la Signature en la dimensión seleccionada, mientras que, las Generic se utilizan para repartir la Signature en las dimensiones restantes.

Como se ha comentado al inicio de la sección, en las dimensiones restantes del Shape, se realizará una copia exacta de la Signature para todos los procesos. Por ejemplo, si tenemos una estructura de 3 dimensiones con el Shape h = (0:4:1,0:4:1,0:4:1) y se selecciona la dimensión 0 para realizar el reparto por pesos entre 3 procesadores con los siguientes pesos $[0.2, 0.4, 0.4]$, las Signatures de las dimensiones 1 y 2 se copiarán del original en los Shapes de todos los procesos y la dimensión 0 se repartirá por pesos, obteniendo como resultado los Shapes (0:0:1,0:4:1,0:4:1), (1:2:1,0:4:1,0:4:1), (3:4:1,0:4:1,0:4:1), los proceos 0,1 y 2, respectivamente.

Debido a que ya existen otros Layouts que realizan copias en otras dimensiones, utilizaremos las funciones de copia que se utilizaron en otros Layouts, como Generic, para este Layout. Hemos extraído dichas funciones de copia del Layout *plug_layDimBlocks*, definido en hitmap/src/hit_layout.c que realiza un reparto por bloques. A continuación, detallaremos qué funciones exactamente hemos utilizado como como Generic. Todas las funciones que indicaremos se encuentran definidas e implementadas en hitmap/src/hit_layout.c

- *HitLayoutSignatureFunction signatureGenericF* → *hit_layout_plug_layCopy_Sig*: Por medio de esta función todos los procesos obtienen una copia de la Signature en la dimensión aplicada. En este caso es sencillamente lo que se requiere.

- *HitLayoutSignatureInvFunction signatureInvGenericF* → *hit_layout_plug_layCopy_SigInv*: En este caso se devuelve el argumento procId en lugar de calcularlo a través de la Signature, ya que, todos los procesos tienen la misma Signature.

- *HitLayoutRanksFunction ranksGenericF*
  → *hit_layout_plug_layRegularContiguos_ranks* Esta función no realiza, realmente, la conversión entre identificadores, ya que, considera que todos los procesos están activos según $min(numBlocks, numProcs-1)$. Sin embargo, si se indica un identificador mayor que el máximo que de procesadores, devuelve NULL. Además, si existe un menor número de bloques que de procesos, considera solo como activos, si hay n bloques, los n primeros procesos, por tanto, se devuelve NULL si se indica un id de proceso mayor que el $min(numBlocks, numProcs - 1)$. Este último comportamiento podría no resultar útil en algunos casos concretos, por ello, en un futuro es posible que se elimine. De todas formas, esta función solo se utiliza para Topologies de más de una dimensión, ya que, se refiere a los procesos de las dimensiones Generic.

- *HitLayoutSignatureMaxCardFunction maxCardGenericF*
  → *hit_layout_plug_layCopy_maxCard*: El máximo cardinal siempre será igual que el cardinal de la Sinature de dicha dimensión, ya que, todos las Signatures de todos los procesos serán iguales a ella, por tanto, se devuelve siempre el cardinal de la Signature original.

- *HitLayoutSignatureMinCardFunction minCardGenericF*
  → *hit_layout_plug_layCopy_minCard*: Al igual que ocurría en el caso anterior, el mínimo cardinal siempre será igual que el cardinal de la Sinature de dicha dimensión, ya que, todos las Signatures de todos los procesos serán iguales a ella, por tanto, se devuelve siempre el cardinal de la Signature original

- *HitLayoutSignatureNumActivesFunction activesGenericF*
  → *hit_layout_plug_layRegular_numActives*: Al igual que sucedía en la función que hemos elegido como parámetro de *ranksGenericF*, para calcular el número de procesos activos se utiliza $min(numBlocks, numProcs - 1)$, dejando a todos lo procesos activos a menos que no existan bloques suficientes. Como ocurría también en el caso de la función de ranks, en estos momentos, este funcionamiento es correcto, aunque puede que en futuro los procesos activos sean siempre todos o se necesite algún otro tipo de política.

Función constructor del Layout: hit_layout_plug_layDimBlocksWeighted

Tras obtener todos los argumentos necesarios para llamar al wrapper y obtener la estructura HitLayout, simplemente resta implementar la función necesaria para poder construir el Layout y utilizarlo a través de Hitmap. En el Código 3.11 se muestra la implementación completa de la función.

```
1  HitLayout hit_layout_plug_layDimBlocksWeighted(HitTopology topo,
       HitShape shape, int restrictDim, Load_ratios l_ratios)
2  {
3    int i = 0;
4    HitLayout res;
5    HitRanks group;
6
7    // Struct for the actual store of the ratios
8    Load_ratios *current_ld = (Load_ratios *)malloc(sizeof(Load_ratios));
9
10   // Compute the actual number of procs in the restricted dim
11   current_ld->num_procs = topo.card[restrictDim] < l_ratios.num_procs ?
         topo.card[restrictDim] : l_ratios.num_procs;
12   current_ld->ratios = (float *)malloc(sizeof(float) * (unsigned long)(
       current_ld->num_procs + 1));
13
14   current_ld->ratios[0] = 0;
15
16   // Compute the accumulate ratios
17   for (i = 0; i < current_ld->num_procs; i++)
18   {
19     current_ld->ratios[i + 1] = current_ld->ratios[i] + l_ratios.ratios
         [i];
20   }
21
```

```
22    /* WRAPPER CALL */
23    res = hit_layout_wrapper(
24      topo,
25      shape,
26      &(hit_layout_plug_layCopy_Sig),
27      &(hit_layout_plug_layCopy_SigInv),
28      &(hit_layout_plug_layDimBlocksWeighted_Sig),
29      &(hit_layout_plug_layDimBlocksWeighted_SigInv),
30      &(hit_layout_plug_layRegularContiguos_ranks),
31      &(hit_layout_plug_layDimBlocksWeighted_ranks),
32      &(hit_layout_plug_layCopy_maxCard),
33      &(hit_layout_plug_layDimBlocksWeighted_maxCard),
34      &(hit_layout_plug_layCopy_minCard),
35      &(hit_layout_plug_layDimBlocksWeighted_minCard),
36      &(hit_layout_plug_layRegular_numActives),
37      &(hit_layout_plug_layDimBlocksWeighted_numActives),
38      (float *)current_ld,
39      restrictDim);
40
41    res.type = HIT_LAYOUT_WEIGHTED;
42
43    if (!res.active)
44    {
45      return res;
46    }
47
48    /* MY GROUP */
49    int dim;
50    for (dim = 0; dim < hit_shapeDims(shape); dim++)
51    {
52      if (res.active)
53      {
54        group.rank[dim] = topo.self.rank[dim];
55      }
56      else
57      {
58        group.rank[dim] = -1;
59      }
60    }
61    res.group = hit_layActiveRanksId(res, group);
62
63    /* MY LEADER */
64    res.leaderRanks = group;
65    res.leader = 1;
66
67    /* RETURN LAYOUT */
68    return res;
69  }
```

Código 3.11: Implementación de la función constructor del Layout *hit_layout_plug_layDimBlocksWeighted*, extraído de hitmap/src/hit_layout.c

En primer lugar, se crea otra estructura interna para almacenar los pesos acumulados, que se calculan en la línea número 16 del Código 3.11. El número de procesos activos máximo se calculará como el mínimo de los procesos de la Topology y del número de procesos indicado en la estructura load_ratios. Si hay más procesos en load_ratios que en la Topology, se utilizarán solo los pesos que haya hasta el último proceso y se normalizarán. Por el contrario, si hay menos procesos en load_ratios que en la Topology, los que sobren se desactivarán.

Más adelante, se realiza la llamada a *hit_layout_wrapper*. Se dispone ya de todos los parámetros necesarios para dicha llamada, la topology, el shape, las doce funciones, el parámetro extra (en este caso será la estructura que almacena los pesos) y, finalmente, la dimensión restringida. Por último, se rellena el grupo de activos y se selecciona el líder, que es algo que no se realiza automáticamente el wrapper, y se devuelve la estructura de tipo HitLayout, y, por tanto, el Layout creado.

Finalmente, tras implementar la función constructor, restaría realizar ciertas tareas como modificar el hit_layFree para que libere los recursos necesarios en caso de utilizar este Layout, escribir la función constructor del Layout en hitmap/include/hit_layout.h y la estructura de pesos, crear la documentación necesaria y algunos otros ajustes que no consideramos que sea necesario detallar, ya que, son detalles internos del funcionamiento de la librería que no son relevantes para el trabajo.

3.3. Layout plug_layBlocksWeightedToSelectedDim

Con este Layout se pretende conseguir un funcionamiento similar al anterior, es decir, un Layout que reparte por pesos en una función Restricted. Sin embargo, en este caso, el Layout no ha de servir para todo tipo de particiones regulares por pesos, es decir, particiones de una o varias dimensiones, sino que, solamente se necesita que se realice una partición por pesos en una sola dimensión y una partición por bloques "tradicional" las dimensiones restantes. Ver Capítulo 2 para consultar el alcance de cada uno de los Layouts propuestos y sus utilidades.

La interfaz de este Layout es similar a la anterior, ya que, tiene los mismos parámetros exactamente. No la detallaremos ahora porque se mostrará en el Código 3.12 un poco más adelante.

Por un lado, las funciones Restricted del wrapper, serán las mismas que en el Layout *plug_layDimBlocksWeighted*, puesto que, se necesita realizar una partición por pesos en esa dimensión. La explicación detallada de la implementación de dichas funciones ya fue realizada en la subsección anterior (Consultar Sección 3.2)

Por otro lado, las funciones Generic del wrapper, deberán realizar un reparto por pesos en cada una de las dimensiones restantes. Debido a que ya existen Layouts en Hitmap que realizan este tipo de reparto (ver Sección 1.3.3) no es necesario que re-implementemos nada. En este caso, utilizaremos las funciones

Generic del Layout *plug_layBlocks* que realizan justamente lo que necesitamos, todas ellas se encuentran definidas e implementadas en hitmap/src/hit_layout.c, estas son:

- *HitLayoutSignatureFunction signatureGenericF* → *hit_layout_plug_layBlocks_Sig*: Realiza un reparto por bloques equitativo y ajusta los restos de forma automática.
- *HitLayoutSignatureInvFunction signatureInvGenericF* → *hit_layout_plug_layBlocks_SigInv*: Realiza el inverso de la anterior. Dado un índice de una Signature, devuelve el proceso al que pertenece.
- *HitLayoutRanksFunction ranksGenericF* → *hit_layout_plug_layRegularContiguos_ranks*: Esta función ya se utilizó en el Layout anterior para las dimensiones copia, por tanto funciona como ya se explicó (ver Sección 3.2). Simplemente transforma de id de Topology a id de activos.
- *HitLayoutSignatureMaxCardFunction maxCardGenericF* → *hit_layout_plug_layRegular_maxCard*: Calcula el máximo número de bloques repartido a los procesos.
- *HitLayoutSignatureMinCardFunction minCardGenericF* → *hit_layout_plug_layRegular_minCard*: Calcula el mínimo número de bloques repartido a los procesos.
- *HitLayoutSignatureNumActivesFunction activesGenericF* → *hit_layout_plug_layRegular_numActives*: Esta función ya se utilizó en el Layout anterior para las dimensiones copia, por tanto funciona como ya se explicó (ver Sección 3.2). Simplemente calcula el número de procesos activos.

A continuación, en Código 3.12 se mostrará la implementación de la función constructor del Layout.

```
1  HitLayout hit_layout_plug_layBlocksWeightedToSelectedDim(HitTopology
       topo, HitShape shape, int restrictDim, Load_ratios l_ratios)
2  {
3    int i = 0;
4    HitLayout res;
5    HitRanks group;
6
7    // Struct for the actual store of the ratios
8    Load_ratios *current_ld = (Load_ratios *)malloc(sizeof(Load_ratios));
9
10   // Compute the actual number of procs in the restricted dim
11   current_ld->num_procs = topo.card[restrictDim] < l_ratios.num_procs ?
         topo.card[restrictDim] : l_ratios.num_procs;
12   current_ld->ratios = (float *)malloc(sizeof(float) * (unsigned long)(
       current_ld->num_procs + 1));
13
14   current_ld->ratios[0] = 0;
15
```

```
16    // Compute the accumulate ratios
17    for (i = 0; i < current_ld->num_procs; i++)
18    {
19      current_ld->ratios[i + 1] = current_ld->ratios[i] + l_ratios.ratios
          [i];
20    }
21
22    /* ACTIVE */
23    res = hit_layout_wrapper(
24      topo,
25      shape,
26      &(hit_layout_plug_layBlocks_Sig),
27      &(hit_layout_plug_layBlocks_SigInv),
28      &(hit_layout_plug_layDimBlocksWeighted_Sig),
29      &(hit_layout_plug_layDimBlocksWeighted_SigInv),
30      &(hit_layout_plug_layRegularContiguos_ranks),
31      &(hit_layout_plug_layDimBlocksWeighted_ranks),
32      &(hit_layout_plug_layRegular_maxCard),
33      &(hit_layout_plug_layDimBlocksWeighted_maxCard),
34      &(hit_layout_plug_layRegular_minCard),
35      &(hit_layout_plug_layDimBlocksWeighted_minCard),
36      &(hit_layout_plug_layRegular_numActives),
37      &(hit_layout_plug_layDimBlocksWeighted_numActives),
38      (float *)current_ld,
39      restrictDim);
40
41    res.type = HIT_LAYOUT_BLOCKS;
42
43    if (!res.active)
44    {
45      return res;
46    }
47
48    /* MY GROUP */
49    int dim;
50    for (dim = 0; dim < hit_shapeDims(shape); dim++)
51    {
52      if (res.active)
53      {
54        group.rank[dim] = topo.self.rank[dim];
55      }
56      else
57      {
58        group.rank[dim] = -1;
59      }
60    }
61    res.group = hit_layActiveRanksId(res, group);
62
```

```
63    /* MY LEADER */
64    res.leaderRanks = group;
65    res.leader = 1;
66
67    return res;
68  }
```

Código 3.12: Implementación de la función constructor del Layout *hit_layout_plug_layBlocksWeightedToSelectedDim*, extraído de hitmap/src/hit_layout.c

Como se puede observar, la función es exactamente igual a la del Layout mostrado en Sección 3.2 simplemente cambiado algunas de las funciones Generic con las que se llama al wrapper para obtener un reparto por bloques en las dimensiones restantes en lugar de la copia de los Shapes como se obtenía en el Layout anterior.

3.4. Conclusiones del capítulo

En este capítulo se ha mostrado la implementación de los Layouts que fueron propuestos en el Capítulo 2. Gracias a la utilización de Hitmap, el esfuerzo de desarrollo del segundo Layout ha sido mínimo, ya que, implementado únicamente una función constructor nueva ha sido suficiente para tener otro Layout funcional. La parte más compleja de la implementación ha sido el desarrollo de las funciones del wrapper y lograr un reparto por pesos prácticamente sin estructuras de datos adicionales, únicamente los propios pesos, con un ajuste automático de restos y con un $O(1)$ para la operación más utilizada, esto es, consultar la Signature perteneciente a un proceso concreto.

Capítulo 4

Tests

En este capítulo se muestran los tests realizadas para probar el correcto funcionamiento del código implementado para el desarrollo del plug-in para Hitmap de reparto por pesos. Debido a que no existe una implementación para el sistema automático de reparto de la carga mostrado en el Capítulo 2, no es posible realizar un conjunto de pruebas y tests.

El desarrollo del capítulo se divide en tres partes. En primer lugar, describiremos los tipos de tests y pruebas que se han realizado y las herramientas utilizadas para ello. En segundo lugar, describiremos las pruebas de caja negra del Layout, que en este caso, son pruebas del Layout completo con ejemplos reales. Por último, mostraremos las pruebas de caja blanca, que, incluyen, todos los tests unitarios de las funciones que se han desarrollado para la implementación de los Layouts y otros tests y otras pruebas adicionales para ciertos casos en los que no es posible realizar tests automatizados con un framework o herramienta debido al paralelismo de la aplicación, el lenguaje de programación, etc.

Para realizar las pruebas de caja negra de los Layouts desarrollados, se han utilizado ciertos problemas que han sido previamente programados de forma secuencial y probados con otros Layouts de Hitmap para comprobar la corrección de los resultados para diferentes tamaños de entrada. Dichos problemas se han ejecutado y comparado los resultados correctos mediante Scripts de bash localizados en hitmap/examples.

En cuanto a las pruebas de caja blanca, se realizarán los tests unitarios de las funciones desarrolladas para particionar los datos, es decir, las funciones que se han desarrollado para *hit_layout_wrapper*, principalmente. Gracias a que estas funciones se pueden probar de forma secuencial, se utilizará el framework de pruebas Check (Ver página web de Check) para realizar tests unitarios, ya que, permite de una forma sencilla, la realización de dichos tests en lenguaje C. En este caso, debido a que C no es un lenguaje orientado a objetos, los tests unitarios se realizarán en torno a la función, en lugar de realizarlos en torno a las clases. Por otro lado, se realizaran pruebas adicionales de integración de todas las funciones y el Layout completo de forma "manual", debido a la imposibilidad de hacerlo de forma automatizada, ya que, hay cierta funcionalidad que solamente se puede probar ejecutando en paralelo la aplicación.

4.1. Pruebas de Caja Negra

Las pruebas de caja negra, en este caso, son los tests de la implementación completa de los Layouts, utilizando aplicaciones reales que se ejecutan en paralelo. Para ello, estas aplicaciones o ejemplos, han sido previamente programados de forma secuencial y probados con otros Layouts de Hitmap para comprobar la corrección de los resultados para diferentes tamaños de entrada. De hecho, estos ejemplos se han utilizado para realizar el estudio experimental del Capítulo 5 con otros tamaños de entrada y así, probar las mejoras del uso de este plug-in de reparto respecto a las particiones originales.

Los ejemplos están ubicados en *hitmap/examples/nombre_de_aplicación* y se dispone de versiones secuenciales de todos ellos y de scripts que ejecutan una verificación de los resultados de la aplicación de Hitmap (check) para diversos tamaños de entrada.

Se han realizado pruebas de caja negra de ambos Layouts con los siguientes ejemplos elegidos, estos son:

- **Stencil 2D Jacobi**: Los aplicaciones de tipo Stencil son ampliamente utilizadas para la resolución numérica de ciertas ecuaciones en derivadas parciales como por ejemplo la Ecuación de convección-difusión en 2D (ver [37]) o la ecuación de Laplace. En nuestro caso, es buen ejemplo para probar el correcto funcionamiento del Layout con estructuras multidimensionales. Se han probado ambos Layouts con este ejemplo
- **Stencil 1D Jacobi**: Es una simplificación del ejemplo anterior que se programó en un inicio para realizar pruebas en Tiles de una sola dimensión. Se han probado ambos Layouts con este ejemplo
- **Cannon Matrix Multiplication**: Multplicación de matrices con un algoritmo paralelo [38]. Realmente no está pensado para una distribución irregular de las matrices pero funciona correctamente con el Layout *plug_layBlocksWeightedToSelectedDim*, que es con el que se probó.
- **SmithWaterman**: Algoritmo que realiza un alineamiento de secuencias de proteínas [39] . Se ha utilizado para realizar pruebas con el Layout *plug_layDimBlocksWeighted* en estructuras unidimensionales, utilizando otro tipo de problema diferente al Stencil.

Por medio de los scripts de verificación de los diferentes ejemplos, se ha comprobado el correcto funcionamiento de los Layouts en diferentes casos con distintas estructuras de datos y con problemas muy diferentes. Ejecutando el Check de cada ejemplo, automáticamente se realizan diversas ejecuciones comprobando que todos los resultados son correctos. Como ya se ha comentado, en el estudio experimental, nos serviremos de los resultados obtenidos en la ejecución de estos ejemplo para demostrar la versatilidad y escaliabilidad de los Layouts por pesos, así como la mejora obtenida en tiempo de ejecución, utilizándolos.

4.2. Pruebas de Caja Blanca

Las pruebas de caja blanca que se realizarán se pueden dividir en dos tipos. Por un lado las pruebas unitarias de las nuevas funciones implementadas y, por otro lado, otras pruebas y tests, como los de integración, que deben realizarse manualmente, ya que, se necesita una ejecución en paralelo para su correcto funcionamiento y no se disponen de los frameworks y herramientas adecuados para este tipo de

pruebas.

4.2.1. Tests unitarios de las funciones implementadas

Se realizarán tests unitarios con el framework *Check* de todas las funciones nuevas que se han implementado y que ha sido posible verificar su funcionamiento con una ejecución secuencial. Esto incluye todas las funciones que se desarrollaron para el particionado de datos y que había que pasarlas al wrapper como argumentos para la construcción del Layout. Sin embargo, no se incluyen las funciones constructoras del Layout, ya que, es necesario verificar su funcionamiento con ejecuciones paralelas con varios procesos. A continuación se muestra la lista de funciones de las que se han implementado tests unitarios.

- *hit_layout_plug_layDimBlocksWeighted_Sig*
- *hit_layout_plug_layDimBlocksWeighted_SigInv*
- *hit_layout_plug_layDimBlocksWeighted_ranks*
- *hit_layout_plug_layDimBlocksWeighted_maxCard*
- *hit_layout_plug_layDimBlocksWeighted_minCard*
- *hit_layout_plug_layDimBlocksWeighted_numActives*

Todos los tests unitarios realizados se encuentran en hitmap/tests/layout-imp.test.c. Invocando a make, se compilan los tests y se pueden ejecutar, mostrando el resultando de todos ellos. A continuación, se detallan los tests unitarios realizados para cada una de estas funciones. Es recomendable ver el Capítulo 3 para comprender que debe realizar cada función.

Tests unitarios de *hit_layout_plug_layDimBlocksWeighted_Sig*

1. *hit_layout_plug_layDimBlocksWeighted_SigTestValid*: Se comprueba que la función se comporta correctamente y realiza el reparto de Signatures entre los diferentes procesos, dado un caso cualquiera.
2. *hit_layout_plug_layDimBlocksWeighted_SigTestValidMoreProcesses*: Se comprueba que la rutina realiza correctamente su función con un mayor número de procesos que en el caso anterior y con un Shape con Stride diferente de 1.
3. *hit_layout_plug_layDimBlocksWeighted_SigTestWithProcsNonActivated*: Se comprueba que la Signature se reparte correctamente y sin huecos entre todos los procesos a pesar de que existan procesos desactivados por no disponer de suficiente carga.
4. *hit_layout_plug_layDimBlocksWeighted_SigTestProcIdGreaterThanMax*: Se comprueba que cuando el procId es mayor que el máximo de procesos, el resultado que devuelva la función sea una Signature nula y que el proceso está desactivado.
5. *hit_layout_plug_layDimBlocksWeighted_SigTestProcIdLowerThanMin*: Igual que en el caso anterior pero, en este caso ProcId es menor que el mínimo. También, el comportamiento de la función ha de ser el mismo, devolver una Signature NULL.

6. *hit_layout_plug_layDimBlocksWeighted_SigTestBlocksCardLessOrEqualTo0*: En este caso también se comprueba que el funcionamiento sea correcto cuando el número de bloques es 0, ya que, es un caso posible. En este caso deberá proporcionar siempre una Signature nula y por tanto devolver 0, ya que todos los procesos permanecerán desactivados.

Como se puede observar no se han realizados tests de verificación de todos los parámetros de la función, ya que, el usuario no llama directamente a estas funciones sino que son utilizadas por otros partes de la librería que nunca podrían indicar, por ejemplo, que el número de procesos es negativo y ya realizan todas las verificaciones oportunas en las funciones expuestas al usuario. Por ello, probamos todos los casos esperados y no todos los posibles.

Tests unitarios de *hit_layout_plug_layDimBlocksWeighted_SigInv*

1. *hit_layout_plug_layDimBlocksWeighted_SigInvTestValid*: Se comprueba que la función se comporta correctamente y retorna el proceso correcto para todos los índices de la Signature, dado un caso cualquiera de uso de la función, siendo todos los parámetros válidos y todos los datos correctos.

2. *hit_layout_plug_layDimBlocksWeighted_SigInvTestWithProcsNonActivated*: Se comprueba que la función se comporta correctamente y retorna el proceso correcto para todos los índices de la Signature, dado un caso en el que existen procesos desactivados.

3. *hit_layout_plug_layDimBlocksWeighted_SigInvTestIndexGreaterThanLimitOfSignature*: Se comprueba que la función retorna HIT_RANK_NULL cuando el índice de la signatura que se proporciona a la función es mayor que el mayor de la Signature original.

4. *hit_layout_plug_layDimBlocksWeighted_SigInvTestIndexLowerThanLimitOfSignature*: Se comprueba que la función retorna HIT_RANK_NULL cuando el índice de la signatura que se proporciona a la función es menor que el menor de la Signature original.

5. *hit_layout_plug_layDimBlocksWeighted_SigInvTestBlocksCardLessOrEqualTo0*: En este caso se comprueba que el funcionamiento sea correcto cuando el número de bloques es 0, ya que, es un caso posible. En este caso se deberá retornar siempre HIT_RANK_NULL, ya que todos los procesos permanecerán desactivados.

6. *hit_layout_plug_layDimBlocksWeighted_SigInvTestLimitIndex*: Se comprueba que la función retorna correctamente el procesador al que pertenece un índice de la signatura que es el primer elemento de alguno de los procesos y que además, al realizar la función *floor* en el reparto de las Signatures se pierde precisión decimal y al deshacer la operación para encontrar el proceso al que pertenecen, con la precisión perdida, el primer resultado es que el proceso debería pertenecer al proceso anterior. En este caso se prueba la corrección que realiza la función para solucionar este pequeño error de precisión.

7. *hit_layout_plug_layDimBlocksWeighted_SigInvTestLimitIndexWithNonActivedProcessesBefore*: Se comprueba lo mismo que en la función anterior, el índice límite de un proceso, sin embargo, en este caso hay procesos desactivados entre el proceso que la función, en un primer lugar, calcula, y, el proceso real al que pertenece el índice.

Al igual que sucedía con la función anterior, no se han realizados tests de verificación de todos los parámetros de la función, ya que,las funciones son utilizadas por otros partes de la librería que nunca podrían indicar ciertos parámetros de forma errónea, por lo cual, no verificamos dichos casos.

Tests unitarios de *hit_layout_plug_layDimBlocksWeighted_ranks*

1. *hit_layout_plug_layDimBlocksWeighted_ranksTestValid*: Se comprueba que la función se comporta correctamente y retorna el processId del proceso activo o de la Topology en cualquiera de los dos modos de funcionamiento con los parámetros válidos y todos los datos correctos.

2. *hit_layout_plug_layDimBlocksWeighted_ranksTestValidWithNonActivatedProcs*: Se comprueba que la función se comporta correctamente y retorna el processId del proceso activo o de la Topology en cualquiera de los dos modos de funcionamiento con los parámetros válidos y todos los datos correctos cuando existen procesos desactivados.

3. *hit_layout_plug_layDimBlocksWeighted_ranksTestTopotoActNonActiveProcess*: Se comprueba que la función devuelve HIT_RANK_NULL cuando se intenta convertir de un proceso no activo de la Topology a los procesos activos,

4. *hit_layout_plug_layDimBlocksWeighted_ranksTestActToTopoNonActiveProcess*: Se comprueba que la función devuelve HIT_RANK_NULL cuando se intenta convertir de un proceso activo que sobrepasa el identificador máximo a un proceso de la Topology.

5. *hit_layout_plug_layDimBlocksWeighted_ranksTestEmptySignature)*: En este caso se comprueba que el funcionamiento sea correcto cuando el número de bloques es 0, ya que, es un caso posible. En este caso se deberá retornar siempre HIT_RANK_NULL, ya que todos los procesos permanecerán desactivados.

Tests unitarios de *hit_layout_plug_layDimBlocksWeighted_maxCard*

1. *hit_layout_plug_layDimBlocksWeighted_maxCardTestValid*: Se comprueba que la función se comporta correctamente y retorna el máximo *Card* o número de bloques que ha sido entregado a un proceso.

2. *hit_layout_plug_layDimBlocksWeighted_maxCardTestSameMaxWeightsButDifferentCards*: Se comprueba que la función se comporta correctamente y retorna el máximo *Card* que ha sido entregado a un proceso si existen dos procesos con el mismo pesos pero diferente *card*, debido al ajuste de los restos.

3. *hit_layout_plug_layDimBlocksWeighted_maxCardTestMinorWeightWithMaxCard*:Se comprueba que la función se comporta correctamente y retorna el máximo *Card* que ha sido entregado a un proceso si existe un proceso con algo más de peso que otro(siendo este el máximo) pero debido al ajuste de los resto tiene menor *card* él. En el Capítulo 3 se explicó esta situación.

4. *hit_layout_plug_layDimBlocksWeighted_maxCardTestBlocksCardEmptyOrLessThan0*: En este caso se comprueba que el funcionamiento sea correcto cuando el número de bloques es 0, ya que, es un caso posible. En este caso se deberá retornar siempre 0, ya que todos los procesos permanecerán desactivados.

Tests unitarios de *hit_layout_plug_layDimBlocksWeighted_minCard*

1. *hit_layout_plug_layDimBlocksWeighted_minCardTestValid*: Se comprueba que la función se comporta correctamente y retorna el mínimo *Card* o número de bloques que ha sido entregado a un proceso.

2. *hit_layout_plug_layDimBlocksWeighted_minCardTestSameMinWeightsButDifferentCards*: Se comprueba que la función se comporta correctamente y retorna el mínimo *Card* que ha sido entregado a un proceso si existen dos procesos con el mismo pesos pero diferente *card*, debido al ajuste de los restos.

3. *hit_layout_plug_layDimBlocksWeighted_minCardTestMinorWeightWithMaxCard*:Se comprueba que la función se comporta correctamente y retorna el mínimo *Card* que ha sido entregado a un proceso si existe un proceso con algo más de peso(siendo este el mínimo) que otro pero debido al ajuste de los resto tiene menor *card* él. En el Capítulo 3 se explicó esta situación.

4. *hit_layout_plug_layDimBlocksWeighted_minCardTestBlocksCardEmptyOrLessThan0*: En este caso se comprueba que el funcionamiento sea correcto cuando el número de bloques es 0, ya que, es un caso posible. En este caso se deberá retornar siempre 0, ya que todos los procesos permanecerán desactivados.

Tests unitarios de *hit_layout_plug_layDimBlocksWeighted_numActives*

1. *hit_layout_plug_layDimBlocksWeighted_numActivesTestValid*: Se comprueba que la función se comporta correctamente y retorna el número de procesos activos, en un caso cualquiera que contiene procesos activados y desactivados de forma aleatoria y con los parámetros válidos.

2. *hit_layout_plug_layDimBlocksWeighted_numActivesTestAllZeros*: Se comprueba que la función se comporta correctamente y devuelve 0 procesadores activos cuando todos los pesos son 0 y ningún proceso tiene partes del Shape.

3. *hit_layout_plug_layDimBlocksWeighted_numActivesTestBlocksCardEmptyOrLessThan0*: En este caso se comprueba que el funcionamiento sea correcto cuando el número de bloques es 0, ya que, es un caso posible. En este caso se deberá retornar siempre 0, ya que todos los procesos permanecerán desactivados.

4.2.2. Otros tests y pruebas

Las funciones constructoras de los Layouts ha sido imposible probarlas de forma secuencial, ya que, necesitaban ser lanzadas con MPI para construir correctamente la Topology, por ejemplo. Además, tampoco se ha podido probar el correcto funcionamiento de todas las "partes" que componen el Layout dentro de la librería, es decir, no se ha podido probar la integración del código en la librería. Por ello, se han realizado otras pruebas manuales para garantizar, en la medida de lo posible, el correcto funcionamiento de ambos Layouts.

Se ha diseñado e implementado un ejemplo básico en el que se crea un Layout por pesos y un HitTile de una sola dimensión. El proceso líder rellena todo el HitTile una primera vez y se lo envía a los demás procesos, cada proceso rellena su parte de un HitTile de una dimensión con números y se lo mandan todos al líder, que finalmente lo imprime. El código fuente del programa que se ha utilizado para realizar dichas pruebas reside en hitmap/tests/parallelTests.c. Por medio de este ejemplo se han realizado las siguientes pruebas **en los dos Layouts implementados**, aunque, para realizar algunas pruebas, se ha añadido código adicional:

1. Sin realizar ninguna modificación, se prueba que el ejemplo funciona con ambos Layouts. Se intenta probar que los Layouts funcionan correctamente con el ejemplo descrito, el proceso líder recibe todos los elementos de todos los procesos y los imprime correctamente.
2. Para diferentes tamaños del HitTile, se imprimen, además del Tile final, los trozos que cada proceso tiene, comprobando que todos los procesos han recibido la parte correspondiente del Shape del HitTile.
3. Para diferentes pesos entre los procesos se imprimen, además del Tile final, los trozos que cada proceso tiene, comprobando que todos los procesos han recibido la parte correspondiente del Shape del HitTile.
4. Se prueba con pesos entre los procesos que sumen más y menos que 1 y se comprueba que los trozos de todos los procesos son los correctos.
5. Se prueba con pesos de cero en uno y varios procesos y se comprueba que los trozos de todos los procesos son los correctos.
6. Se lanza la aplicación con n procesos a través de *mpiexec*, pero se le pasa a Load_ratios un número menor de procesos. Automáticamente todos los procesos con id mayor que el máximo de Load_ratios deben quedar desactivados. Se comprueba que esto sucede así.
7. Se lanza la aplicación con n procesos a través de *mpiexec*, pero se le pasa a Load_ratios un número mayor de procesos. Automáticamente el Layout debe ajustar el número de procesos en la estructura Load_ratios al número de procesos real, ya que este es menor. En este momento, el peso de cada proceso será su peso original divido entre la suma de los primeros n pesos activos y los pesos restantes, se inutilizarán. Se comprueba que esto sucede así.
8. Se comprueba que los procesos activos en los Layouts creados, se indican como activos mediante la función hit_layImActive que proporciona Hitmap.
9. A través de la ejecución y funcionamiento del ejemplo, se comprueba que las comunicaciones funcionan correctamente utilizando cualquiera de los dos Layouts.

4.3. Conclusiones del capítulo

Mediante las pruebas de caja negra, hemos podido verificar que los Layouts funcionan correctamente con aplicaciones completas y, de hecho, se ha podido garantizar su utilización para ejemplos muy diferentes. Utilizando las pruebas de caja blanca, hemos podido depurar, en primera instancia, todas las funciones que fueron implementadas para realizar los cálculos de las particiones y, en segunda instancia, la integración del Layout en la librería y su correcto funcionamiento ejecutándose en paralelo con un ejemplo real. La realización de pruebas ha resultado ser una tarea imprescindible aunque complicada debido al tipo de aplicación, lenguaje de programación, paralelismo, etc.

Capítulo 5

Estudio experimental

En este capítulo se realiza un estudio experimental para comprobar la aceleración obtenida en el clúster heterogéneo del Grupo de Investigación Trasgo, utilizando un reparto de carga balanceado según la capacidad computacional del nodo, por medio de los Layouts de Hitmap que se han desarrollado.

5.1. Descripción del entorno de experimentación

Para realizar la experimentación, vamos a emplear tres máquinas diferentes del clúster del Grupo Trasgo, que es heterogéneo. A pesar de que existen más nodos, estos tres, son bastante diferentes entre sí y cada uno dispone de una capacidad de cómputo diferente. Estas máquinas son Manticore, Heracles e Hydra. Sus especificaciones son las siguientes:

Maticore:

- CPU: 2 x Intel Xenon Platinum 8160 @ 2.10GHz (96 cores)
- Memoria: 256 GB DDR4
- Coprocesaores: NVIDIA TESLA V100

Heracles:

- CPU: AMD Opteron 6376 @ 2.3 GHz (64 cores)
- Memoria: 250 GB
- Coprocesaores: _

Hydra:

- CPU: 2 x Xenon E5-2690v3 @ 1.9 GHz (12 cores)
- Memoria: 64GB
- Coprocesaores: 4 x NVIDIA GTX TITAN BLACK 2880

A continuación, se muestran las tres experimentaciones que se han realizado para probar el correcto funcionamiento de los Layouts. Uno de los principales motivos por los que se han elegido estos ejemplos es que ya existía una implementación en Hitmap de los mismos.

5.2. Stencil 2D Jacobi

Los aplicaciones de tipo Stencil son ampliamente utilizadas para la resolución numérica de ciertas ecuaciones en derivadas parciales como por ejemplo la Ecuación de convección-difusión en 2D (ver [37]) o la ecuación de Laplace. Ya utilizamos este ejemplo en el Capítulo 4 para realizar pruebas del funcionamiento de los Layouts. La aplicación se encuentra codificada en hitmap/examples/Stencil_Jacobi2D.

Para esta primera experimentación, utilizaremos las tres máquinas que se han descrito en la primera sección del capítulo. En cada una, emplearemos 12 procesos , ya que, es el máximo de una de estas máquinas, y, para esta experimentación, usaremos el mismo número de procesadores en todos los nodos (en este caso 3). La máquina que dispone de más capacidad de cómputo es Manticore, depués, Heracles y, por último, Hydra. Otorgaremos más carga computacional a los nodos con mejor capacidad de procesamiento. Por ello, la distribución de carga será la siguiente:

15 % de la carga se dispondrá en Hydra.

38 % de la carga se ubicará en Heracles.

47 % de la carga se situará en Manticore.

En cada máquina, los 12 procesos tendrán el mismo ratio de carga.

Se utilizará para probar la velocidad de ejecución en diferentes entradas, el Stencil de Jacobi en 2D [40] , que ya ha sido analizado, resuelto y probado en Hitmap [21]. Mostraremos la mejora en tiempo de ejecución al utilizar una distribución en bloques por pesos frente a usar un reparto equitativo por bloques entre los tres nodos elegidos del clúster. Utilizaremos, por una parte, el *Layout plug_layDimBlocksWeighted*, para medir el tiempo de ejecución cuando se reparte la matriz por filas de forma balanceada, según la carga indicada, entre todos los procesos. Por otro lado, utilizaremos *lay_DimBlocks* ,Layout de reparto por bloques y por dimensión de forma equitativa, para medir el tiempo de ejecución de un reparto por filas de la matriz, equitativo, entre todos los procesos.

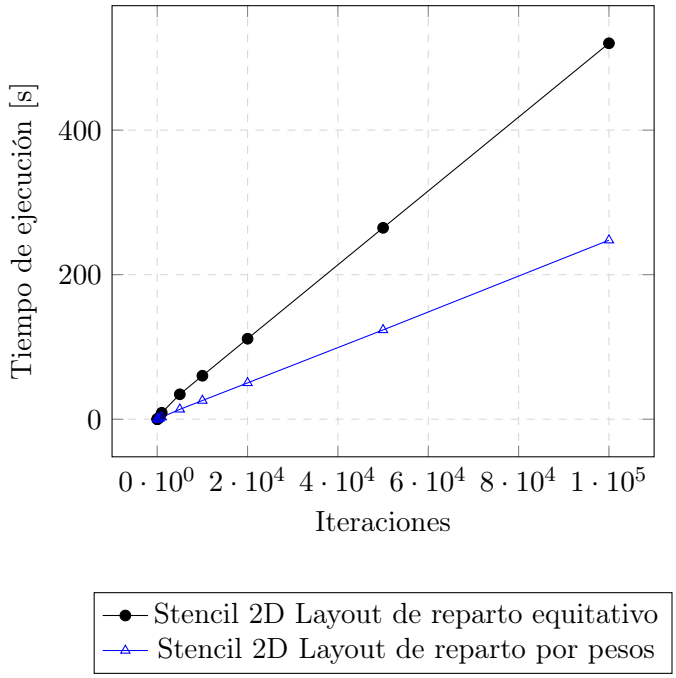

Figura 5.1: Tiempo de ejecución por iteraciones, tamaño de la matriz fija: 3000×3000, Stencil2D

Hemos decido medir el tiempo de ejecución empleado por el ejemplo del Stencil de Jacobi en 2D, con ambos Layouts (equitativo y balanceado por pesos). Por una parte, a medida que aumentan las iteraciones, con un tamaño de matriz fijo , Figura 5.1. Por otro lado, con un número de iteraciones fijo (1000), a medida que aumenta el tamaño de la matriz.

En la Figura 5.1 se muestra el resultado de una serie de ejecuciones con un tamaño de matriz fija cuadrada, 3000×3000, lo suficientemente grande como para que los 36 procesos utilizados, tengan la suficiente capacidad de cómputo en cada iteración. Se puede observar como la mejora al realizar un reparto de la carga balanceado según las capacidades computacionales de cada nodo, es cada vez mayor. El tiempo por iteración es más corto,ya que, la carga está mejor distribuida. Por tanto, la mejoría es más notable cuantas más iteraciones pasan, puesto que, cada iteración, el ejemplo con el Layout por pesos, "gana tiempo", con respecto al ejemplo con una distribución de carga normal. De hecho, en el último punto, 100000 iteraciones, se alcanza la aceleración máxima, de, un 110 % con 520 segundos en el caso del reparto equitativo y 247 segundos en el caso de un reparto por pesos.

En la Figura 5.2 se muestra el resultado de una serie de ejecuciones con un número de iteraciones fijo de 1000, y un tamaño n, variable, de la matriz $(n \times n)$. En la Figura 5.2 podemos fácilmente apreciar, que, la mejora del un Layout por pesos, se aprecia en matrices generalmente grandes, que es, dónde un número grande de procesos, como 36, tienen una carga por iteración elevada. Sin embargo, en matrices pequeñas, prácticamente no existe mejora, incluso, el resultado podría empeorar. Esto es debido a que cuando la carga por iteración es pequeña, con un reparto no equitativo, podríamos dejar algunos nodos desactivados o con muy poca carga computacional comparada con otros, resultando en una espera de los nodos menos potentes a los nodos más potentes. La mejora máxima, se alcanza con la matriz de 20000×20000, una aceleración del 70 % con 154 segundos en el caso del reparto equitativo y 250 segundos en el caso de un reparto por pesos.

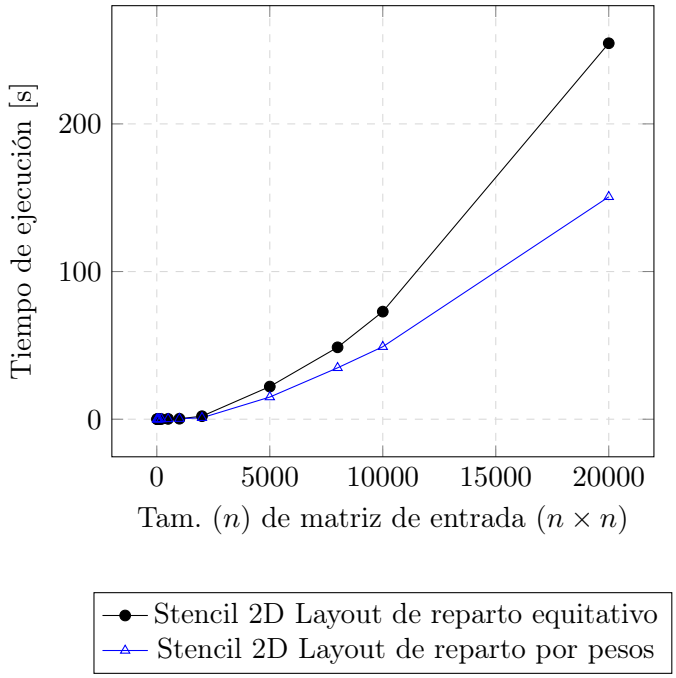

Figura 5.2: Tiempo de ejecución por tamaño de la matriz, iteraciones fijas: 1000, Stencil2D

En general, podemos observar que, en problemas grandes el rendimiento puede llegar a ser incluso el doble, como se ha visto en la Figura 5.1. En estos casos, en los que hay muchas iteraciones, el tiempo de ejecución es cada vez más bajo, utilizando un reparto por pesos en comparación con un reparto equitativo, debido a que, en cada iteración el reparto por pesos gana x segundos de ventaja y, por tanto, cuantas más iteraciones, mayor será la diferencia. Sin embargo, realmente, lo que hace mejorar el rendimiento de la aplicación es el tamaño de entrada, ya que, cuanta mayor es la matriz, el reparto por pesos es más efectivo, pues, los nodos con peores capacidades pueden recibir parte del Tile sin resultar un cuello de botella, de hecho, aceleran la ejecución. Por otro lado, cuanto menor es el tamaño de entrada, realmente, los nodos más lentos solo generan pérdidas de rendimiento. De hecho en la Figura 5.2, se muestra como para tamaños de entrada muy pequeños la diferencia de tiempos de ejecución es mínima, llegando en algunos casos incluso a ser negativa, es decir, mayor el tiempo de ejecución en el caso del Layout por pesos.

En conclusión, en el ejemplo del Stencil 2D de Jacobi se observa una mejora utilizando el Layout por pesos en un sistema heterogéneo. De hecho, en aplicaciones de este tipo, la diferencia entre usar un reaprto balanceado y un equitativo, puede llegar a suponer hasta el doble, en tiempo de ejecución.

5.3. Cannon Matrix Multiplication

Cannon MM es un algoritmo paralelo de multiplicación de matrices [38]. Realmente está pensado para distribuciones regulares de las matrices a través de los procesos pero lo utilizaremos para probar que el Layout desarrollado funciona correctamente en casos muy diferentes. La aplicación se encuentra codificada en hitmap/examples/CannonMM

En este caso, utilizaremos las máquinas Hydra y Manticore, con 8 procesos en cada una (16 en total), ya que, para que el funcionamineto del algoritmo sea correcto, necesitamos un número de procesos cuya raíz cuadrada sea exacta, y ya que, el máximo de procesos en Hydra es 12, el número entero más cercano cuya raíz cuadrada del doble sea un número entero es 8. Hemos utilizado dos máquinas para simplificar el ajuste de pesos y la ejecución, además, ya hemos utilizado las tres máquinas en el primer ejemplo y se ha demostrado un correcto funcionamiento. En este caso el ajuste de pesos, será el siguiente:

46 % de la carga se dispondrá en Hydra.

55 % de la carga se situará en Manticore.

Debido a que este algoritmo está pensado para tener una carga balanceada, hemos ajustado unos pesos similares para no provocar una caída de rendimiento, intentando que la carga computacional sea mayor en el mejor de los nodos. Además, se ha utilizado el Layout *plug_ layBlocksWeightedToSelectedDim*, ya que, debido a la implementación, la Topology y, el algoritmo en sí, en este caso, se necesita repartir también la segunda dimensión y, esta es la forma más sencilla.

En la Figura 5.3 se muestra la ejecución del algoritmo CannonMM con el Layout *plug_ layBlocks* y con el Layout implementado *plug_ layBlocksWeightedToSelectedDim*. En general, para tamaños de matriz pequeños, se observa una tendencia general de un mayor tiempo de ejecución en el Layout por pesos. Como se ha comentado anteriormente, el reparto por pesos, tiende a ser más eficiente cuanto mayor es el tamaño de entrada que cada proceso posee. Por otra parte, cuando el tamaño de entrada crece, en general, el tiempo de ejecución del Layout por pesos es más eficiente. Sin embargo, existe un punto aislado ($n = 12000$), en el cual, para un mayor tamaño de entrada se obtiene un menor tiempo de ejecución que en el punto anterior y, además, sorprendentemente, es más eficiente el Layout *plug_ layBlocks*, cuando en el punto anterior, ocurría justo lo contrario. Esto se debe al funcionamiento del algoritmo, que, según como sea el tamaño de entrada y de como cuadren los trozos de las matrices en cada proceso, puede llegar a ser más eficiente con un tamaño de entrada mayor que otro. Debido a que los procesos reciben filas o columnas enteras de otros procesos para realizar la multiplicación de matrices, cuando se le entrega más carga a un proceso que a otro, en realidad, debería ser más lenta la ejecución, ya que, los procesos con menor carga, también reciben, a veces trozos grandes de columnas o filas de otros procesos. Por ello, precisamnete no se busca una mejora en la ejecución de este ejemplo, sino simplemente demostrar que funciona correctamente y que la diferencia entre utilizar un reparto por pesos y uno equitativo, incluso cuando el algoritmo no está preparado puede resultar en una eficiencia similar e incluso posiblemente mejor en algunos casos.

En esta aplicación, se debería realizar un reparto por pesos diferente según el tamaño de entrada de la matriz, ya que, en la mayor parte de los casos lo más eficiente puede ser un reparto absolutamente equitativo, pero, puede haber tamaños de entrada en los cuales pueda llegar a ser útil desbalancear algo la carga para obtener un resultado un poco mejor. Debido a los los Layouts permiten en tiempo de ejecución redistribuir los pesos, incluso se podrían realizar ajustes automáticos para mejorar el tiempo de ejecución en todos los casos. Quizá en este tipo de aplicaciones con comportamientos indefinidos o irregulares para un desbalanceo de la carga, pueda llegar a probarse la mejora de un sistema automático de equilibrado de la carga como el que se diseñó en el Capítulo 2.

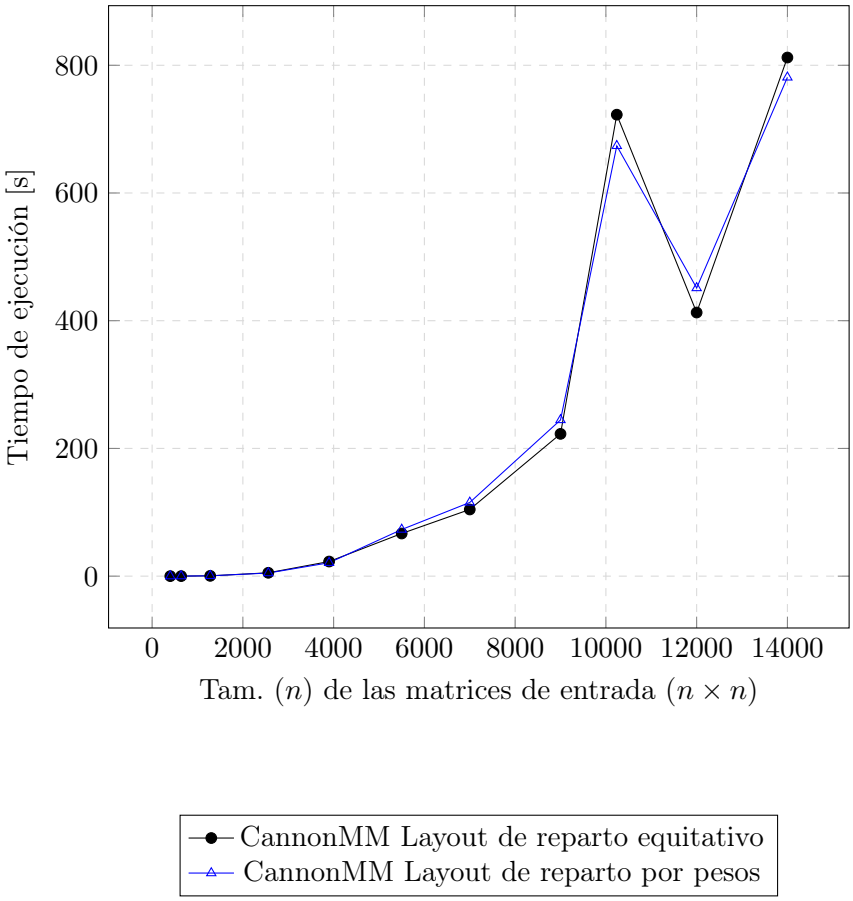

Figura 5.3: Tiempo de ejecución por tamaño de las matrices, Cannon MM

5.4. SmithWaterman

El algoritmo de SmithWaterman realiza un alineamiento de secuencias de proteínas. En [39] se explica su funcionamiento. También se ha utilizado para realizar pruebas de caja negra con el Layout *plug_layDimBlocksWeighted* en estructuras unidimensionales. En el caso de la experimentación, nos sirve para probar la mejora en otro tipo de problemas que no son únicamente iterativos como es el caso del Stencil, sino una aplicación que contiene partes con diferentes algoritmos, una de ellas aplica un algoritmo de Backtracking. Aunque se puede aplicar el algoritmo de SmithWaterman varias iteraciones, nosotros solo utilizaremos una iteración, ya que, lo que nos interesa es probar el funcionamineto y las mejoras del Layout en otro tipo de problema diferente al Stecil y con estructuras unidimensionales. Las secuencias de proteínas de entrada y el PAM son las más grandes que había probadas en ejemplos secuenciales. En este caso se ha ido variando el tamaño de las dos secuencias de proteínas, ambas siempre iguales. Este tamaño es realmente el card de la Signature de cada uno de los HitTiles, que en este caso, son unidimensionales. La aplicación se encuentra codificada en hitmap/examples/SmithWaterman

En este caso, utilizaremos también las máquinas Hydra y Manticore, esta vez con 12 procesos en cada una (24 en total), ya que, el máximo de procesos en Hydra es 12, y queremos tener el mismo número de procesos en ambas máquinas para simular mejor la heterogeneidad de procesos. Hemos utilizado dos máquinas para simplificar el ajuste de pesos y la ejecución, además, ya hemos utilizado las tres máquinas

en el primer ejemplo y se ha demostrado un correcto funcionamiento. En este caso el ajuste de pesos, será el siguiente:

17 % de la carga se dispondrá en Hydra.

83 % de la carga se situará en Manticore.

Se ha entregado bastante más carga al nodo con mayor capacidad de cómputo, ya que, en este caso, en el que la carga computacional sí se puede repartir de forma desigual, para conseguir unos tiempos de ejecución óptimos, hay que disponer prácticamente la totalidad de la carga en el mejor de los nodos.

En la Figura 5.4 se muestra la ejecución del algoritmo SmithWaterman con el Layout *plug_layDimBlocks* y con el Layout implementado *plug_layDimBlocksWeighted*. En genral, se observa un menor tiempo de ejecución con el Layout de reparto por pesos, aunque, tampoco demasiado. En este caso, quizá utilizando un mayor número de nodos podríamos haber observado una mayor aceleración con la carga repartida por pesos. La mayor aceleración se obtiene con el mayor tamaño de la secuencia de proteínas, es decir, 80000. En el caso del Layout por pesos se obtiene un tiempo de 88.84 segundos frente a los 93 obtenidos con el reparto equitativo, es decir una aceleración de menos del 5 % (93/88.84). Aunque no es una mejora demasiado buena, realmente creemos que el entorno de pruebas tampoco ha resultado ser el más acertado, ya que, con solo con dos nodos tan diferentes casi la mejora más evidente es desactivar el nodo menos potente. Posiblemente añadiendo otros dos nodos y una mayor carga computacional, la mejora podría haber sido más notable. Debido a que tampoco se dispone de demasiado tiempo, no se puede repetir la experimentación completa de la aplicación en este trabajo. El reparto por pesos en el caso de utilizar un mayor número de nodos, también habría sido bastante diferente.

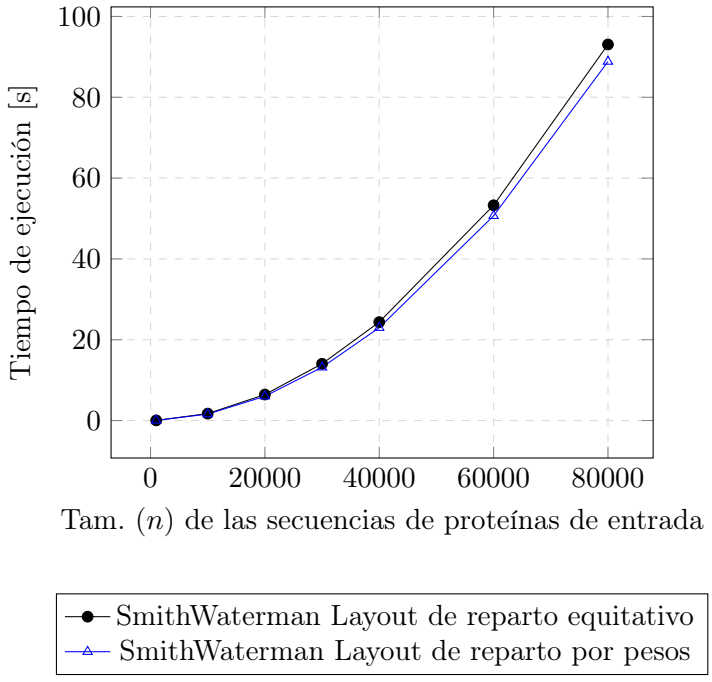

Figura 5.4: Tiempo de ejecución por tamaño de las secuencias de proteínas, SmithWaterman

A pesar de todo, existe una mejora utilizando el Layout implementado, por ello, se comprueba que el Layout por pesos también es más eficiente en este tipo de problemas. Sin embargo, problemas puramente iterativos como el Stencil son en los se consiguen mejores resultados.

5.5. Conclusiones del capítulo

Por medio de la realización del estudio experimental hemos probado que existe un mejor aprovechamiento del entorno heterogéneo al utilizar nuestra propuesta, generando una incremento de la eficiencia en la mayor parte de los casos. En algunos casos como el Stencil 2D, se ha logrado una aceleración de más del doble respecto al reparto por bloques equitativo, en determinadas situaciones. También se ha conseguido una aceleración en la ejecución del algoritmo de SmithWaterman aunque mucho más baja. Por último, en el caso de la aplicación de Cannon MM, se ha conseguido mantener la eficiencia, generalmente, llegando a mejorar los tiempo de ejecución en algunos casos, cuando, realmente el algoritmo no está pensado para realizar repartos irregulares.

Capítulo 6

Conclusiones

En este capítulo se muestran los objetivos cumplidos, las conclusiones extraídas, las líneas de trabajo futuro y la valoración personal del presente Trabajo de Fin de Grado.

6.1. Objetivos cumplidos

Este trabajo presenta una propuesta del equilibrado de carga en sistemas heterogéneos desarrollada en colaboración con el Grupo de Investigación Trasgo. Tras realizar un estudio de la herramienta Hitmap, desarrollada por el Grupo de Investigación Trasgo, se ha diseñado e implementado un plug-in para Hitmap que permite realizar una distribución de carga, balanceada según los pesos que se indiquen para cada proceso, a través de la creación de dos nuevos Layouts. La implementación de este sistema ha sido verificada y validada a través de determinadas pruebas.

Por otro lado, se ha diseñado un sistema de reparto de carga automático en tiempo de ejecución que utiliza los Layouts que se han desarrollado en este trabajo. Finalmente, no ha sido posible realizar su implementación por falta de tiempo. Sin embargo se propone un algoritmo básico de ajuste de los pesos, que será implementado en un futuro.

6.2. Conclusiones extraídas

Gracias al plug-in desarrollado para Hitmap, es posible efectuar un particionado de datos que permita aprovechar la capacidad de cómputo de los sistemas heterogéneos. A través del estudio experimental, se demuestra la aceleración obtenida al repartir correctamente la carga computacional en tiempo de ejecución, llegando a obtener velocidades de ejecución hasta un 110 % más rápidas que con un reparto equitativo, en el entorno experimental estudiado. Además, este sistema permite modificar los pesos de cada proceso en tiempo de ejecución, algo que otros trabajos similares no pueden realizar. Gracias a esta característica, se podrá implementar en el futuro el sistema automático de reparto de la carga, del que hemos proporcionado el diseño.

Este Trabajo de fin de Grado ha dado como resultado dos contribuciones para la comunidad científica:

- El artículo *Mecanismo de equilibrado de carga en sistemas heterogéneos*, que ya ha sido aceptado en

el congreso nacional **XXX Jornadas de Paralelismo (JP2019)** y se presentará en Cáceres del 18 al 20 de Septiembre de 2019.

- El póster *Load Distribution in Heteogeneous Computing*, en el congreso internacional **CMMSE**, que se celebrará en Rota (Cádiz) de los días 30 de Junio al 6 de Julio de 2019.

6.3. Líneas de trabajo futuro

Existen, principalmente, dos líneas diferentes de trabajo futuro. Por un lado, la mejora de los Layouts de Hitmap, permitiendo un reparto a nivel de nodo, indicando los nombres del nodo y los pesos que se quiere repartir a cada nodo; agrupando las sucesivas llamadas al Layout, para un reparto por pesos en múltiples dimensiones, en una sola función; incluyendo nuevos tipos de repartos combinados con un reparto por pesos u otras posibles mejoras del propio Layout. Por otro lado, la implementación y mejora del sistema que equilibra automáticamente la carga en tiempo de ejecución. En esta línea, existe bastante trabajo por realizar, diferentes algoritmos para lograr un reparto equilibrado de la carga, etc. Por último, se realizaría la integración de este sistema automático de equilibrado de carga con la librería Controller [33], para su uso en todo tipo de dispositivos de procesamiento, es decir, tanto en CPUs como en aceleradores hardware.

6.4. Valoración personal

En primer lugar, considero que a nivel académico, el Trabajo Fin de Grado me ha aportado conocimientos complementarios de programación paralela, Computación de Alto Rendimiento, Sistemas Heterogéneos que de cara a un posible futuro dedicado a la investigación en Computación de Alto Rendimiento serán muy útiles, ya que, han sentado ciertas bases para el trabajo e investigación futuros. Además, las dos publicaciones en congresos del Trabajo, han resultado en una tarea de aprendizaje en la escritura de artículos de investigación, texto científico, etc.

En segundo lugar, la experiencia de permanecer este último año colaborando para el Grupo de Investigación Trasgo, mediante la realización de este trabajo, ha sido bastante positiva y me ha permitido observar y participar en el mundo académico de la investigación, que es otra salida diferente al trabajo en una empresa, quizás algo bastante más común y sencillo de encontrar en este tipo de estudios. Además, he aprendido de personas expertas en este área de la Computación de Altas Prestaciones y he podido colaborar con ellas en proyectos muy interesantes.

Por último, tanto la realización del Trabajo de Fin de Grado con un grupo de investigación como la publicación de un artículo y un póster, me han permitido mejorar el currículum académico para una futura salida de investigación en la universidad.

Bibliografía

[1] Wikipedia, *Von neumann architecture*, en *Wikipedia*, Page Version ID: 895354628, 3 de mayo de 2019. dirección: https://en.wikipedia.org/w/index.php?title=Von_Neumann_architecture&oldid=895354628.

[2] Wikibooks. (2019). Digital Circuits/Transistor Basics - Wikibooks, open books for an open world, dirección: https://en.wikibooks.org/wiki/Digital_Circuits/Transistor_Basics.

[3] G. E. Moore, "Cramming more components onto integrated circuits", *IEEE Solid-State Circuits Society Newsletter*, vol. 11, n.º 3, págs. 33-35, sep. de 2006, ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860.

[4] G. S. Almasi y A. Gottlieb, *Highly Parallel Computing*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1989, ISBN: 978-0-8053-0177-9.

[5] M. J. Flynn, "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, vol. C-21, n.º 9, págs. 948-960, sep. de 1972, ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[6] Wikipedia, *Flynn's taxonomy*, en *Wikipedia*, Page Version ID: 898983463, 27 de mayo de 2019. dirección: https://en.wikipedia.org/w/index.php?title=Flynn%27s_taxonomy&oldid=898983463.

[7] ——, *Computer cluster*, en *Wikipedia*, Page Version ID: 898695354, 25 de mayo de 2019. dirección: https://en.wikipedia.org/w/index.php?title=Computer_cluster&oldid=898695354.

[8] Intel. (2019). Procesador Intel® Xeon® Platinum 9282 (77 MB de caché, 2,60 GHz), Intel, dirección: https://www.intel.com/content/www/es/es/products/processors/xeon/scalable/platinum-processors/platinum-9282.html.

[9] Wikipedia, *Field-programmable gate array*, en *Wikipedia, la enciclopedia libre*, Page Version ID: 115703333, 4 de mayo de 2019. dirección: https://es.wikipedia.org/w/index.php?title=Field-programmable_gate_array&oldid=115703333.

[10] Intel. (2019). Procesadores Intel® Xeon Phi™, Intel, dirección: https://www.intel.com/content/www/es/es/products/processors/xeon-phi/xeon-phi-processors.html.

[11] NVIDIA. (2019). NVIDIA Tesla V100, NVIDIA, dirección: https://www.nvidia.com/es-es/data-center/tesla-v100/.

[12] K. Dowd y C. Severance, "High Performance Computing", *openstax cnx*, 2010. dirección: http://oers.taiwanmooc.org/jspui/handle/123456789/129565.

[13] TOP500. (2019). TOP500 Supercomputer Sites, dirección: https://www.top500.org/.

[14] W. D. Hillis y G. L. Steele, "Data parallel algorithms", *Communications of the ACM*, vol. 29, n.º 12, pág. 14, 1986.

[15] J. Dongarra y A. L. Lastovetsky, *High Performance Heterogeneous Computing*. John Wiley & Sons, 11 de ago. de 2009, 285 **pagetotals**, Google-Books-ID: wfC7LPTcRmYC, ISBN: 978-0-470-50819-0.

[16] Wikipedia, *Interfaz de Paso de Mensajes*, en *Wikipedia, la enciclopedia libre*, Page Version ID: 114859802, 27 de mar. de 2019. dirección: https://es.wikipedia.org/w/index.php?title=Interfaz_de_Paso_de_Mensajes&oldid=114859802.

[17] MPI-Forum. (2019). MPI documents, dirección: https://www.mpi-forum.org/docs/.

[18] M. Tutorial. (2019). MPI Hello World, dirección: http://mpitutorial.com/tutorials/mpi-hello-world/.

[19] G. Trasgo. (2019). Hitmap – Grupo Trasgo, dirección: https://trasgo.infor.uva.es/hitmap/.

[20] ——, (2019). Grupo Trasgo – Trasgo Research Group (Universidad de Valladolid), dirección: https://trasgo.infor.uva.es/.

[21] A. Gonzalez-Escribano, Y. Torres, J. Fresno y D. R. Llanos, "An Extensible System for Multilevel Automatic Data Partition and Mapping", *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, n.º 5, págs. 1145-1154, mayo de 2014, ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.83.

[22] L. T. Yang y M. Guo, *High-Performance Computing: Paradigm and Infrastructure*. John Wiley & Sons, 18 de nov. de 2005, 819 **pagetotals**, Google-Books-ID: qA4DbnFB2XcC, ISBN: 978-0-471-73270-9.

[23] B. Barney. (2019). Introduction to Parallel Computing, dirección: https://computing.llnl.gov/tutorials/parallel_comp/#distributions.

[24] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn y M. Pericás, "Trends in Data Locality Abstractions for HPC Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, n.º 10, págs. 3007-3020, oct. de 2017, ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2703149.

[25] D. B. Loveman, "High performance Fortran", *IEEE Parallel Distributed Technology: Systems Applications*, vol. 1, n.º 1, págs. 25-42, feb. de 1993, ISSN: 1063-6552. DOI: 10.1109/88.219857.

[26] I. Foster, "Task Parallelism and High-Performance Languages", *IEEE Parallel Distrib. Technol.*, vol. 2, n.º 3, págs. 27-36, sep. de 1994, ISSN: 1063-6552. DOI: 10.1109/M-PDT.1994.329794. dirección: http://dx.doi.org/10.1109/M-PDT.1994.329794 (visitado 19-06-2019).

[27] B. L. Chamberlain, S. J. Deitz, D. Iten y S.-E. Choi, "User-defined Distributions and Layouts in Chapel: Philosophy and Framework", en *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, ép. HotPar'10, event-place: Berkeley, CA, Berkeley, CA, USA: USENIX Association, 2010, págs. 12-12. dirección: http://dl.acm.org/citation.cfm?id=1863086.1863098.

[28] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato y L. Rauchwerger,
"STAPL: Standard Template Adaptive Parallel Library",
en *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, ép. SYSTOR '10, event-place: Haifa, Israel, New York, NY, USA: ACM, 2010, 14:1-14:10, ISBN: 978-1-60558-908-4. DOI: 10.1145/1815695.1815713. dirección: http://doi.acm.org/10.1145/1815695.1815713.

[29] K. Fürlinger, C. Glass, J. Gracia, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedheb y H. Zhou,
"DASH: Data structures and algorithms with support for hierarchical locality",
en *Euro-Par 2014: Parallel Processing Workshops*, L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart y M. Alexander, eds., ép. Lecture Notes in Computer Science, Springer International Publishing, 2014, págs. 542-552, ISBN: 978-3-319-14313-2.

[30] C. A. Bohn y G. B. Lamont, "Load balancing for heterogeneous clusters of PCs", *Future Generation Computer Systems*, Cluster Computing, vol. 18, n.º 3, págs. 389-400, 1 de ene. de 2002, ISSN: 0167-739X. DOI: 10.1016/S0167-739X(01)00058-9. dirección: http://www.sciencedirect.com/science/article/pii/S0167739X01000589.

[31] J. Dongarra, T. Sterling, H. Simon y E. Strohmaier, "High-performance computing: clusters, constellations, MPPs, and future directions",
Computing in Science Engineering, vol. 7, n.º 2, págs. 51-59, mar. de 2005, ISSN: 1521-9615. DOI: 10.1109/MCSE.2005.34.

[32] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik y O. O. Storaasli, "State-of-the-art in Heterogeneous Computing", *Sci. Program.*, vol. 18, n.º 1, págs. 1-33, ene. de 2010, ISSN: 1058-9244. DOI: 10.1155/2010/540159. dirección: http://dx.doi.org/10.1155/2010/540159.

[33] A. Moreton–Fernandez, H. Ortega–Arranz y A. Gonzalez–Escribano, "Controllers: An abstraction to ease the use of hardware accelerators", *The International Journal of High Performance Computing Applications*, vol. 32, n.º 6, págs. 838-853, 1 de nov. de 2018, ISSN: 1094-3420. DOI: 10.1177/1094342017702962. dirección: https://doi.org/10.1177/1094342017702962.

[34] C. University. (2019). Cornell Virtual Workshop,
dirección: https://cvw.cac.cornell.edu/MPIcc/allgather.

[35] GNU. (2019). The C Preprocessor: Macros,
dirección: https://gcc.gnu.org/onlinedocs/cpp/Macros.html.

[36] D. E. Knuth, *The Art of Computer Programming, Volumes 1-4A Boxed Set*,
Edición: new edition. Revised.
Amsterdam: Addison-Wesley Educational Publishers Inc, 3 de mar. de 2011, 3168 **pagetotals**, ISBN: 978-0-321-75104-1.

[37] S.-h. Zhang y W.-q. Wang, "A stencil of the finite-difference method for the 2D convection diffusion equation and its new iterative scheme", *Int. J. Comput. Math.*, vol. 87, págs. 2588-2600, 2010. DOI: 10.1080/00207160802691637.

[38] A. Gupta y V. Kumar, "Scalability of Parallel Algorithms for Matrix Multiplication", en *1993 International Conference on Parallel Processing - ICPP'93*, vol. 3, ago. de 1993, págs. 115-123. DOI: 10.1109/ICPP.1993.160.

[39] C. W. Yu, K. H. Kwong, K. H. Lee y P. H. W. Leong, "A smith-waterman systolic cell", en *Field Programmable Logic and Application*, P. Y. K. Cheung y G. A. Constantinides, eds., ép. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, págs. 375-384, ISBN: 978-3-540-45234-8.

[40] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev y P. Sadayappan, "Effective Automatic Parallelization of Stencil Computations", en *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ép. PLDI '07, event-place: San Diego, California, USA, New York, NY, USA: ACM, 2007, págs. 235-244, ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250761. dirección: http://doi.acm.org/10.1145/1250734.1250761 (visitado 24-05-2019).

Apéndices

Apéndice A

Contenido del CD-ROM

En el directorio raíz del CD-ROM entregado, se encuentran almacenados los siguientes elementos:

- **hitmap**/ → Directorio que contiene el código fuente de la librería Hitmap con el plug-in, que se ha desarrollado, ya integrado. Además del código fuente, también cuenta con:
  - **examples**/ → Directorio con los ejemplos utilizados en la experimentación y en algunas de las pruebas.
  - **test**/ → Directorio que contiene los test unitarios del código implementado en el trabajo y otras pruebas adicionales.
- **hitmap_doc** → Directorio que contiene la documentación de la librería Hitmap en formato web. Se debe ingresar a la página que figura con el nombre index.html para iniciar la visualización de la documentación.
- **memoria.pdf** → La memoria del TFG, el documento actual.