
**Universidad de Valladolid**

Escuela de Ingeniería Informática

**Trabajo Fin de Grado**

Grado en Ingeniería Informática
(Mención Tecnologías de la Información)

# Distribución dinámica de carga y redistribuciones de datos en aplicaciones paralelas

Autora:
**D.ª María Sánchez Girón**


**Universidad de Valladolid**

Escuela de Ingeniería Informática

**Trabajo Fin de Grado**

Grado en Ingeniería Informática
(Mención Tecnologías de la Información)

# Distribución dinámica de carga y redistribuciones de datos en aplicaciones paralelas

Autora:
**D.ª María Sánchez Girón**

Tutores:
**Dr. Arturo González Escribano**
**Dr. Yuri Torres de la Sierra**

*A mis padres, a Ana y a Javier.*

# Resumen

Actualmente, la computación de alto rendimiento es la técnica utilizada para resolver grandes problemas en diversas áreas de investigación (ciencia, ingeniería, etc.) debido al aumento de rendimiento que proporcionan los supercomputadores. La computación heterogénea permite adaptar un sistema a un rango mayor de aplicaciones gracias a la integración de componentes de naturalezas diferentes en el sistema de cómputo, aprovechando así cada uno de los recursos hardware de cada dispositivo. Los supercomputadores con arquitecturas heterogéneas se cuentan actualmente entre los más potentes del mundo.

*Tiling* es un método utilizado para mejorar el rendimiento de los sistemas paralelos que consiste en dividir el espacio de datos de un problema entre los procesos. Para equilibrar el tiempo de ejecución de cada proceso y mejorar así el tiempo total del programa se puede aplicar el balanceo de carga, un particionado irregular en el que el tamaño asignado a cada proceso depende de su capacidad computacional. La estimación de la capacidad puede realizarse antes de la ejecución de un programa o en tiempo de ejecución. El balanceo de carga adaptativo permite reestimar la carga y modificar el reparto de trabajo entre los procesos a lo largo de la ejecución de un programa, pero es un trabajo que tiene que hacer el programador para su aplicación concreta. No existe una función estándar de balanceo adaptativo dinámico para aplicaciones paralelas.

La librería Hitmap proporciona herramientas para la gestión del particionado y mapeado de arrays de una manera simple y eficiente en un modelo de paralelismo SPMD. Cuenta con diversos tipos de particiones y separa la parte de comunicación del particionado de los datos, adaptando automáticamente las funciones al tipo de partición elegido gracias al uso de abstracciones.

Este trabajo propone una función de balanceo de carga dinámico, adaptativo y transparente para entornos de computación paralela utilizando los recursos de Hitmap. Los resultados experimentales muestran que su uso mejora el rendimiento de los programas, reduciendo el tiempo total de ejecución frente a un reparto equitativo de la carga entre los procesos, sin suponer un sobrecoste de tiempo o recursos.

# Abstract

Nowadays, high performace computing is the technique used to solve big problems in varied research areas (science, engineering, etc.) due to the increase of performance the supercomputers provide. Heterogeneous computing allows to adapt a system to a bigger range of applications thanks to the integration of different natured components into the computing system, making advantage of every one of the hardware resources of each device. Supercomputers with heterogeneous architectures are currently included among the most powerful ones in the world.

Tiling is a method used to improve the performance of parallel systems, consisting of splitting the data space of a problem among the processes. In order to balance the execution time of each process and therefore improving the total execution time of the program, load balancing can be applied, an irregular tiling where the size assigned to each process depends on its computational capacity. The estimation of capacity can be done before the execution of the program or at runtime. Adaptive load balancing allows to re-estimate the load and to modify the distribution of work amongst the processes throughout the execution of a program, but is a task that must be made by the programmer for his particular application. There is not a standard function for dynamic, adaptive load balancing for parallel applications.

The Hitmap library provides tools to manage the tiling and mapping of arrays in a simple and efficient way in a SPMD parallel model. It features many partition types and it isolates the communication part from the data partitioning, automatically adapting the functions to the partition type chosen thanks to the use of abstractions.

This work proposes a dynamic, adaptive and transparent load balancing function for parallel computing environments using Hitmap's resources. The experimental results show that using it improves the performance of the programmes, reducing the execution time compared to an equitative load distribution among the processes, without signifying a time or resources overhead.

# Índice general

# Índice de tablas

# Índice de figuras

# Índice de códigos

# Índice de algoritmos

# Capítulo 1

# Introducción

En este capítulo se introducen los siguientes aspectos:

- El contexto y la motivación del proyecto.
- El planteamiento del problema y los objetivos del proyecto.
- La planificación del desarrollo del proyecto.
- La organización de los contenidos del documento.

## 1.1. Contexto

En la actualidad, la resolución de problemas por computador es una tarea indispensable para un conjunto significativo de áreas de investigación, ya sea en el ámbito científico o en empresas que manipulan grandes cantidades de datos. Aunque el aumento de las capacidades de los ordenadores personales permite realizar muchas de esas tareas con ellos, hoy en día sigue habiendo algunas cuyos requisitos de espacio de almacenamiento, memoria o capacidad computacional no pueden conseguirse con una única máquina computacional [1]. La computación de alto rendimiento (HPC) surge como consecuencia de esa necesidad de máquinas capaces de ejecutar programas muy costosos y con grandes exigencias de recursos. HPC se define como: "la práctica de agregar estaciones de trabajo (y, por tanto, potencia de cómputo) de una manera que proporciona un rendimiento mucho mayor que el que se consigue con una sola máquina, con el objetivo de resolver grandes problemas en ciencia, ingeniería, etc" [2].

Una forma de clasificar el tipo de computación es discernir entre homogénea y heterogénea [3]. Los sistemas homogéneos son aquellos que utilizan un mismo tipo de dispositivo para realizar el cómputo de datos. Normalmente, estos sistemas utilizan un solo modo de paralelismo (SIMD, MIMD, etc. [4]), por lo que no son apropiados para aplicaciones que requieran más de uno, pues una máquina ejecutaría código para el que no está debidamente preparada. Por otra parte, la computación heterogénea integra dispositivos de naturaleza diferente en un mismo sistema. Permite adaptar un sistema a un amplio rango de aplicaciones gracias a su variabilidad de funcionalidades específicas y capacidades. Las arquitecturas heterogéneas están presentes en los supercomputadores modernos más potentes del mundo, que se listan en el ranking Top500 [5].

Para aumentar el rendimiento de los sistemas de computación paralela existen varias técnicas. Una de estas técnicas es el *tiling* [6, 7], que consiste en dividir el dominio de la aplicación en

particiones y asignar cada una a un proceso. Estos bloques, o *tiles*, son de tamaño fijo y pueden tener diversas formas, tales como triangular, rectangular, hexagonal, etc., o sus equivalentes de más dimensiones. La elección de tamaño y forma se puede optimizar para aprovechar la localidad de los datos dentro de la memoria donde se alojan. Las particiones pueden ser iguales para todos los procesos o puede aplicarse el balanceo de carga (*load balancing*) [8] como técnica de particionado irregular. El objetivo del balanceo de carga es repartir el trabajo entre los procesos para equilibrar el tiempo total de ejecución. En muchas aplicaciones, es posible estimar *a priori* la distribución de trabajo de manera que el programador puede realizar el balanceo de carga antes de la ejecución. Por el contrario, en otras aplicaciones no es posible realizar esa estimación inicial y es necesario en tiempo de ejecución estimar las capacidades de cada uno de los dispositivos para realizar el particionado de forma proporcional. Estos dos tipos de balanceo de carga se denominan respectivamente estático y dinámico.

Actualmente existen diversos modelos de programación para gestionar el *tiling* y el mapeado de los bloques entre procesos en programación paralela, como Unified Parallel C [9], una extensión del lenguaje C para máquinas paralelas; HTA (*Hierarchically Tiled Array*) [10], un tipo de datos para representar particionados jerárquicos de datos; o Hitmap [11].

Hitmap [11] es una librería para el particionado y mapeado eficiente de arrays. Desacopla las comunicaciones del particionado de los datos y tiene funcionalidades para crear, manipular, distribuir y comunicar los *tiles*. También permite el particionado con balanceo dinámico de carga. Cuenta con dos versiones, v.1 y v.2, que se diferencian en la API. Hitmap2 proporciona una interfaz más sencilla y añade un tipo para representar *tiles* distribuidos, pero está todavía en desarrollo.

## 1.2. Motivación

Las técnicas de particionado permiten repartir el espacio de datos de un problema entre varios procesos. Tanto en el balanceo de carga estático como en el dinámico, el reparto se realiza una vez, lo que si bien mejora los resultados de un particionado equitativo al ser un reparto más adaptado a la capacidad de cada proceso, puede ocasionar errores. Estos errores pueden deberse a una mala estimación inicial o a comportamientos de las máquinas que cambien los valores de las cargas medidas en tiempo de ejecución. Las estrategias actuales para modelar el rendimiento de las CPUs no pueden predecir los cambios de frecuencias que se producen por la acumulación de calor en éstas, y que dependen del calor residual de la ejecución de trabajos anteriores y de la temperatura del entorno en el que se encuentran las máquinas. Si bien existen métodos para modificar el reparto de trabajo a los procesos a lo largo de la ejecución de un programa, conocidos como balanceos de carga adaptativos, son prototipos hechos por los programadores para una aplicación específica. No existe una función estándar de *tiling* dinámico adaptativo que pueda usarse sobre varios arrays multidimensionales simultáneamente, que distribuya la carga de forma fluida y transparente al usuario, y que pueda aplicarse a programas con características y funcionamientos diversos. Nuestro trabajo se centra en esa carencia en la comunidad científica.

## 1.3. Planteamiento del problema

En este trabajo se propone desarrollar una función de distribución de carga y datos dinámica, adaptativa y transparente para entornos de computación paralela basada en los recursos de la librería Hitmap [11]. Se parte de una distribución inicial de la carga de trabajo total entre los procesos y se mide de forma reiterada el tiempo de ejecución de las tareas sobre el trabajo asignado a cada uno. En base a esos tiempos, se reparten de nuevo los datos quitando carga a los procesos más lentos y asignándola a los más rápidos. El objetivo es adaptar la cantidad de trabajo a la capacidad computacional de cada proceso para equilibrar así los tiempos de ejecución y mejorar el tiempo total.

### 1.3.1. Objetivos

Los objetivos de este proyecto corresponden con las características que debe cumplir la función de distribución dinámica de carga y datos entre procesos para programas de computación paralela, que a partir de ahora será nombrada como WeightedRedistribute como abreviatura de su nombre en la librería, `hit_alb_weightedRedistribute`, que sigue la nomenclatura de los elementos de Hitmap: el prefijo *hit*, el nombre de la clase o estructura a la que se refieren, en este caso ALB (*Automatic Load Balancing*), y el nombre de la fucionalidad en sí.

- **Transparencia.** La redistribución debe realizarse de la manera más transparente posible, ocultando los detalles internos al usuario, y sobre las mismas estructuras pasadas como parámetrox.
- **Mínima intrusión.** El usuario debe modificar lo mínimo posible su programa para permitir la redistribución. Idealmente, solo debería incluir la llamada a la función.
- **Adaptabilidad.** La función debe de ser capaz de distribuir un número de estructuras de datos variable.
- **Mejora de rendimiento.** La función debe ser capaz de mejorar el rendimiento y la eficiencia de las aplicaciones quitando carga a los procesos menos potentes y repartiéndola entre los más potentes. Además, el tiempo de redistribución no debería suponer una carga para el programa global que no compense el tiempo ganado con las mejoras. Tampoco debería implicar un gran consumo de memoria.

Se realizarán dos versiones de WeightedRedistribute, una para cada versión de la librería Hitmap, y un estudio de rendimiento aplicado a un caso de estudio: el problema de difusión de calor con el método iterativo de Jacobi en un espacio bidimensional.

A continuación, se muestran los objetivos genéricos del proyecto relacionados con las tareas a cumplir y competencias a adquirir a lo largo de su desarrollo:

- Estudiar, analizar y comprender la librería Hitmap en sus versiones 1 y 2.
- Diseñar e implementar un prototipo de la función para ser utilizado con Hitmap v1.
- Diseñar e implementar un prototipo de la función para ser utilizado con Hitmap v2.

- Estudiar y comprender el problema de difusión de calor con el método iterativo de Jacobi.
- Diseñar, elaborar y ejecutar un plan de pruebas para validar la eficacia de las funciones.
- Analizar los resultados de la experimentación y extraer conclusiones.

## 1.4. Planificación del proyecto

### 1.4.1. Modelo de desarrollo

Para la elaboración del proyecto se ha elegido un SDLC (*Systems Development Life Cycle*) incremental [12, 13]. Este tipo de ciclo de vida, un enfoque iterativo del modelo de cascada [12, 13], divide el proyecto en etapas que se desarrollan y entregan secuencialmente siguiendo un proceso de cascada: análisis de requisitos, diseño, implementación, pruebas y mantenimiento (figura 1.1). La primera iteración constituye el núcleo del sistema y contiene los requisitos básicos y el resto de iteraciones van añadiendo funcionalidades a esa versión hasta haber implementado todas las deseadas.

El desarrollo escogido es adecuado para proyectos en los que no se conocen todos los requisitos al principio o es probable que cambien, y en los que se necesita tener pronto una versión con funcionalidad básica. Las ventajas frente a otros modelos son la facilidad de implementación y mantenibilidad, que los sub-proyectos asociados a cada iteración son más fáciles de gestionar que todo el proyecto global, y que el *feedback* de cada uno ayuda a mejorar el siguiente.

Se elige este modelo ya que el proyecto se basa en un prototipo al cual se van incorporando nuevas funcionalidades. Hasta que no se aplica la función base a un programa real no se ven todas las necesidades y requisitos. Por otro lado, a partir de un estado intermedio de la función el proyecto se divide en dos vías: empezar a realizar pruebas y seguir desarrollando WeightedRedistribute con los requisitos restantes y aplicando el *feedback* recibido de las pruebas.

#### 1.4.1.1. Etapas

En esta subsección se explican las etapas en las que se ha dividido el proyecto y su duración estimada. Se ha planeado un tiempo total para el proyecto de aproximadamete 15 semanas como un tiempo razonable para cumplir con los objetivos y ajustarse a las horas establecidas para un proyecto de 12 créditos ECTS, equivaliendo un ECTS a 25 horas de trabajo. La figura 1.2 muestra el diagrama de Gantt, con cada etapa indentificada por un número (ID) en la parte izquierda, y la tabla 1.1 indica el nombre de la etapa asociada a cada ID. Todas las fases del desarrollo incremental (de la 2 a la 8) comprenden una parte de diseño, una implementación y unas pruebas. En el caso de las etapas de desarrollo de la función (etapas 3, 4, 6 y 7), las pruebas consisten en ejecutar el programa con varios procesos, tamaños de matriz e iteraciones y contrastar los resultados con los obtenidos en el programa original antes de incluir la función. Este *testing* se realiza para controlar que las funcionalidades añadidas siguen produciendo resultados correctos. En el caso de las experimentaciones (etapas 5 y 8), la parte de pruebas consiste en la mera ejecución de las pruebas de rendimiento diseñadas. Las flechas en el diagrama muestran las dependencias

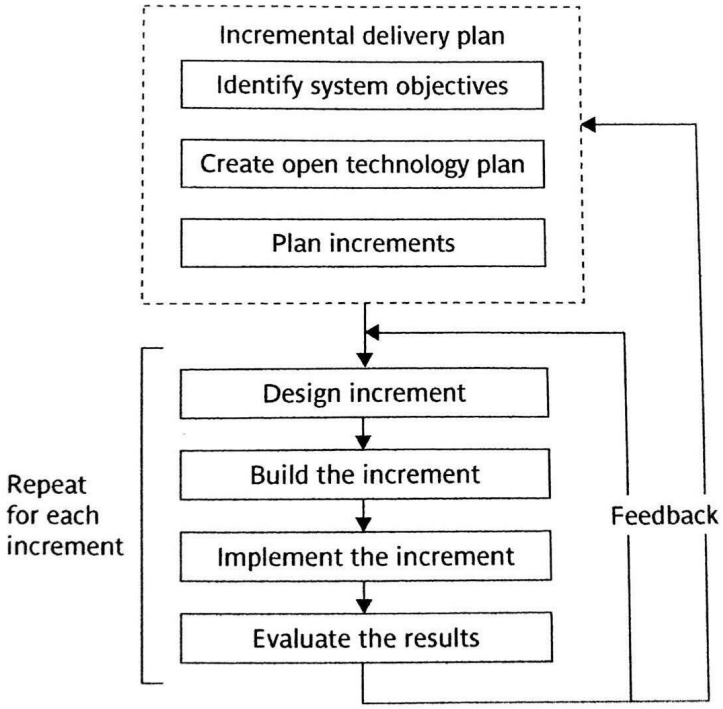



Figura 1.1: Etapas del desarrollo incremental. Imagen extraída de [12].

entre tareas e indican que la tarea con la punta de la flecha necesita de la otra para completarse correctamente. La mayoría de etapas están un par de días solapadas con su predecesora porque mientras se termina de probar un incremento, el siguiente puede comenzar su diseño.

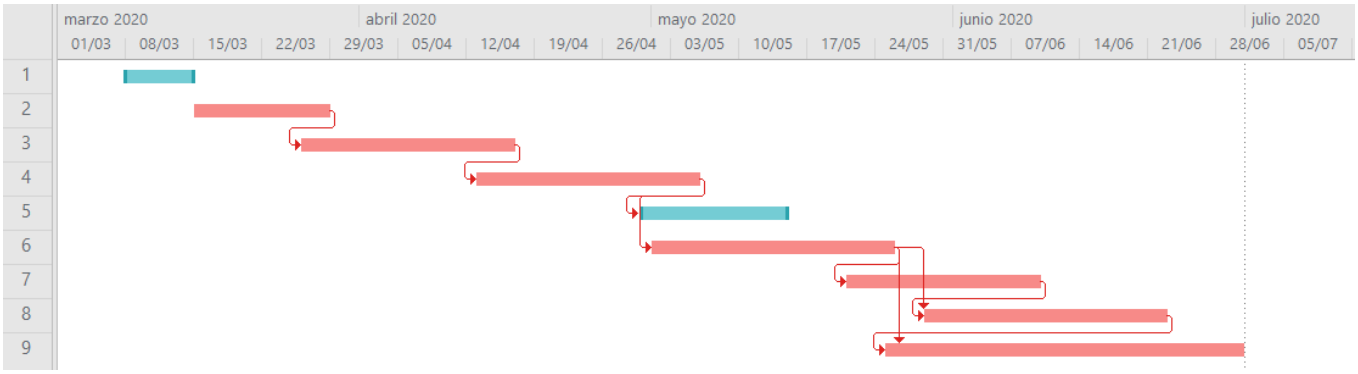



Figura 1.2: Diagrama de Gantt del proyecto.

La primera etapa corresponde al aprendizaje de la estructura y el funcionamiento de Hitmap a través de la lectura de documentación y la ejecución de ejemplos reducidos. Se le asigna una semana ya que se tenían conocimientos previos de la librería. La segunda etapa comienza el desarrollo incremental y consiste en la especificación inicial de objetivos y análisis de trabajos similares existentes. En la tercera se desarrolla una primera versión básica de WeightedRedistribute para Hitmap v1, que se completa con el cálculo de la media en la etapa siguiente. Ambas se prueban aplicadas a un programa de ejemplo sencillo, en el que la carga de cómputo se basa en realizar unas operaciones aritméticas a cada celda de la matriz. Seguidamente, se realiza una primera experimentación de dos semanas que prueba la eficacia de WeightedRedistribute con diversos

| ID | Descripción |
|---|---|
| 1 | Aprendizaje de Hitmap |
| 2 | Especificación de objetivos y análisis |
| 3 | 1ª versión de WeightedRedistribute para Hitmap v1 |
| 4 | 2ª versión: incorporación de la media |
| 5 | Primera experimentación |
| 6 | 3ª versión: aplicación a un caso de estudio (Jacobi) |
| 7 | Versión de WeightedRedistribute para Hitmap v2 |
| 8 | Experimentación final |
| 9 | Elaboración de la memoria |

Tabla 1.1: Descripción de las etapas del diagrama de Gantt del proyecto.

parámetros de entrada.

Al mismo tiempo que se realiza esta experimentación, se aplica la función a un caso de estudio: la distribución de calor con el método iterativo de Jacobi. Al igual que las otras etapas de desarrollo (3 y 4), se le asigna una duración de tres semanas, para adaptar la función al programa y por si surge algún imprevisto. En la etapa 7 se desarrolla la versión de WeightedRedistribute para Hitmap v2 adaptando la versión ya creada y validada para Hitmap v1. La última etapa del desarrollo incremental es una experimentación con las versiones finales de la función. Se le asignan tres semanas, una semana más que la primera experimentación porque se realizará un mayor número de pruebas y medirán más variables. La elaboración de la memoria del proyecto comienza a la mitad de la séptima etapa y dura cinco semanas. Se le ha dado este tiempo para elaborarla poco a poco, añadiendo los cambios relativos a la función a medida que son probados y validados.

#### 1.4.1.2. Camino crítico

El camino crítico [14] es un conjunto de actividades sucesivas que definen la duración de un proyecto. Un retraso en cualquiera de ellas retrasa el proyecto total. En el diagrama de Gantt se representa el camino crítico con las actividades marcadas en rojo. Casi todas las tareas son críticas debido a que el proyecto es en su mayor parte un proceso secuencial, y cuando hay varias tareas solapadas, como en la fase final, las dependencias hacen que las tareas se vuelvan críticas. Las etapas que no están en el camino crítico están relacionadas con el aprendizaje de Hitmap, ya que al conocer previamente la librería no es estrictamente necesario para comenzar el proyecto; y con la primera experimentación, pues es una prueba del rendimiento de la versión sencilla de WeightedRedistribute. La experimentación importante es la segunda, porque se realiza sobre las versiones finales de la función y mide su rendimiento aplicado a un caso real.

#### 1.4.1.3. Seguimiento

En términos generales, la duración real de las etapas del proyecto no se ha correspondido con el tiempo estimado. Las etapas 1 y 2 sí se desarrollaron acorde a su planificación, pero la etapa 3

sufrió un retraso de aproximadamente una semana por fallos en la comprensión de la idea inicial y falta de experiencia en el manejo de Hitmap. Las dos etapas siguientes se realizaron en el tiempo estimado, pero no pudieron compensar el desfase inducido por la tarea anterior. La etapa 6 hace que aumente el retraso del proyecto debido a más imprevistos de los esperados en la adaptación de WeightedRedistribute al caso de estudio. En este momento se decide desechar la implementación de la función para Hitmap v2 y pasar directamente a la experimentación, con el objetivo de intentar reducir el retraso. Sin embargo, en esta etapa también se producen problemas, tanto técnicos en el entorno de ejecución, que obligaron a parar la experimentación, como pruebas que hubo que repetir por una detección tardía de fallos en la función, y la propuesta de nuevas modificaciones para mejorar el rendimiento de ésta. Con todo esto, el proyecto se prolonga diez semanas más, divididas entre las etapas 6, 8 y 9.

### 1.4.2. Análisis de riesgos

Un riesgo se define como "un evento o condición inciertos que, si ocurren, tienen un efecto positivo o negativo en los objetivos de un proyecto" [15]. Los riesgos pueden referirse tanto a la gestión del proyecto, al personal implicado, a las tecnologías utilizadas o a las tareas.

La tabla 1.2 presenta la identificación de los riesgos del proyecto. Se ha hecho partiendo de la lista de los diez riesgos más comunes en el desarrollo de software de Barry Boehm [16], quitando los que no son relevantes para este proyecto y añadiendo otros. A cada riesgo se le asigna un valor de probabilidad de ocurrencia e impacto de 0 a 10, siendo 0 el mínimo y 10 el máximo, y se calcula la exposición al riesgo como el producto de ambos. Los riesgos con los valores más altos son los más importantes y los primeros a tener en cuenta a la hora de planificar los planes de contingencia.

| **Descripción del riesgo** | **Probabilidad** | **Impacto** | **Riesgo** |
|---|---|---|---|
| Enfermedad del desarrollador | 2 | 6 | 12 |
| Estimación de costes y tiempo del proyecto no realistas | 3 | 5 | 15 |
| Desarrollo incorrecto de funciones software | 3 | 3 | 9 |
| *Gold-plating*[1] | 3 | 3 | 9 |
| Desarrollo demasiado difícil técnicamente | 3 | 5 | 15 |
| Cambios tardíos y/o continuos a los requisitos | 4 | 4 | 16 |
| Fallo de las máquinas donde se realiza la experimentación | 4 | 7 | 28 |
| Resultados de la experimentación negativos | 6 | 3 | 18 |

Tabla 1.2: Identificación de riesgos del proyecto.

[1] *Gold-plating* (chapado en oro): añadir funcionalidades a un proyecto innecesarias y que no forman parte de los requisitos [12]. Esas propuestas pueden surgir tanto del lado del cliente como de los desarrolladores.

#### 1.4.2.1. Plan de contingencia

Un plan de contingencia [15] es una acción planeada con antelación que se lleva a cabo si un riesgo se materializa. En este proyecto, al utilizar un desarrollo incremental se mitigan los riesgos relacionados con requisitos o funcionalidades, porque esos cambios se añaden a las iteraciones siguientes. Igualmente ocurre si hay problemas en una función especifica: si no se arregla de una manera rápida y no es crítica, se puede posponer para etapas posteriores. Si la estimación de la duración del proyecto es incorrecta o hay una parte difícil de desarrollar, eso significa que el producto final tendrá menos características de las diseñadas en un principio, pero será funcional, aunque de una manera más sencilla; y las tareas que no ha dado tiempo a realizar se dejarán como trabajo futuro. En cuanto a los riesgos relacionados con la experimentación, se contará con varias máquinas en las que realizar las pruebas, de manera que si existiera algún tipo de fallo se puedan utilizar otras. Si los resultados obtenidos de las pruebas de rendimiento no son los esperados tampoco es un riesgo grave, pues al fin y al cabo se ha demostrado que las hipótesis no eran correctas.

### 1.4.3. Presupuesto

En esta sección se presenta un análisis del presupuesto del proyecto. Incluye los costes de trabajo del personal, el coste de utilización de las estaciones de trabajo y la adquisición de licencias de herramientas software. Para cada grupo se presenta una tabla con el coste de cada uno de sus elementos y una breve explicación del gasto. Finalmente, se calculará el presupuesto total.

En la tabla 1.3 se muestran los costes del personal. Para las horas de trabajo de la alumna se parte de las 300 horas que supone un proyecto de 12 créditos ECTS, siendo un ECTS equivalente a 25 horas de trabajo; pero dado que se decide alargar la duración del proyecto, el número de horas final aumenta. Estas horas cuentan todas las tareas realizadas por la alumna: aprendizaje inicial de Hitmap, desarrollo de todas las versiones de WeightedRedistribute, resolución de problemas, experimentación y escritura de esta memoria. El tiempo de trabajo de los tutores corresponde a las reuniones, el tiempo de corrección y resolución de problemas en el software de la alumna y el tiempo de corrección de la memoria. Para el presupuesto de cada uno se utiliza un sueldo por hora de desarrollador junior para la alumna y senior para los tutores.

| Personal | Coste/hora (€) | Horas | Total (€) |
|---|---|---|---|
| Alumna | 25 | 500 | 12500 |
| Tutores | 50 | 60 | 3000 |
| **Total** | | | **15500** |

Tabla 1.3: Coste de trabajo del personal del proyecto.

La tabla 1.4 muestra el coste de uso de las máquinas utilizadas para la experimentación. El campo de amortización indica el tiempo de vida aproximado la máquina, que corresponde a la duración de un proyecto de investigación. El coste por hora se calcula como el precio de compra de la máquina entre el tiempo de amortización. Las horas de uso cuentan el tiempo empleado en las experimentaciones. Las horas son las mismas en ambas máquinas porque las pruebas siempre se ejecutan en esos dos nodos.

| Máquina | Precio (€) | Amortización (años) | Coste/hora (€) | Horas | Total (€) |
|---|---|---|---|---|---|
| Manticore | 38000 | 4 | 1.08 | 300 | 324 |
| Heracles | 10000 | 4 | 0.29 | 300 | 87 |
| **Total** | | | | | **411** |

Tabla 1.4: Coste de uso de las máquinas en el proyecto.

En la tabla 1.5 se resumen los costes del proyecto detallados anteriormente y se calcula el presupuesto total necesario para su realización. El apartado de software incluiría el precio de las herramientas utilizadas, pero en este proyecto todo el software es gratuito.

| Actividad | Coste (€) |
|---|---|
| Horas de trabajo del personal | 15500 |
| Uso de las máquinas | 411 |
| Licencias de software | 0 |
| **Total** | **15911** |

Tabla 1.5: Coste total del proyecto.

## 1.5. Organización del documento

La estructura de este documento es como sigue: en el capítulo 2 se introducen conceptos y trabajos relacionados, como la librería Hitmap. El capítulo 3 describe el prototipo de WeightedRedistribute e introduce un estudio de medias móviles para incorporar al diseño. El capítulo 4 explica la implementación final de WeightedRedistribute y expone el caso de estudio elegido para incorporar la función. El capítulo 5 presenta la experimentación realizada para validar el funcionamiento de las herramientas creadas y el rendimiento de WeightedRedistribute. Finalmente, el capítulo 6 describe los objetivos cumplidos y las líneas de trabajo futuras a partir de este proyecto.

# Capítulo 2

# Estado del arte y conceptos previos

En este capítulo se introducen los siguientes aspectos:

- La librería Hitmap desarrollada por el Grupo de Investigación Trasgo de la Universidad de Valladolid.
- El estándar MPI para computación paralela.

## 2.1. Librería Hitmap

### 2.1.1. Descripción

Hitmap [11] es una librería diseñada para el particionado (*tiling*) y mapeado eficiente de arrays y estructuras dispersas en lenguaje C, desarrollada por el Grupo de Investigación Trasgo de la Universidad de Valladolid (España) [17]. Al contrario que otras herramientas como HTA [10] o UPC [9], Hitmap desacopla los patrones de comunicación del particionado de datos mediante el uso de abstracciones que se adaptan en tiempo de ejecución al tipo de partición elegido, y cuenta con funcionalidades para crear, manipular, distribuir y comunicar los *tiles*. Tiene un diseño orientado a objetos e implementa el paralelismo con un modelo SPMD (*Single Program Multiple Data*).

### 2.1.2. Funcionalidades

Esta librería introduce conceptos y estructuras novedosas para simbolizar los arrays y gestionar su manejo. Los conceptos y estructuras más significativas se describen a continuación:

- **Signature.** Una Signature $S$ es una terna utilizada para representar un conjunto de índices en un espacio unidimensional, siguiendo una notación similar a Fortran90 o MATLAB. El número de índices diferentes de $S$ se denomina cardinalidad. Se implementa en la clase HitSig.

$$\begin{aligned} S \in \text{Signature} &= (\text{begin} : \text{end} : \text{stride}) \\ \text{Card}(\text{s} \in \text{Signature}) &= \lfloor\ (\text{s.end - s.begin})/\text{s.stride}\ \rfloor \end{aligned} \tag{2.1}$$

- **Shape.** Un Shape $h$ es un conjunto de $n$ Signatures que representa un subespacio de índices en un dominio multidimensional. La cardinalidad de un Shape es el número de combinaciones diferentes de índices en el dominio. Se implementa en la clase HitShape.

$$h \in \text{Shape} = (S_0, S_1, S_2, \ \ldots \ , S_{n-1}) \tag{2.2}$$
$$\text{Card(h} \in \text{Shape)} = \prod_{i=0}^{n-1} \text{Card}(S_i)$$

- **Tile.** Un Tile es un array $n$-dimensional. Se define mediante un Shape y tiene un número de elementos de un tipo dado (int, float, double ...). Se implementa con la clase abstracta HitTile.

$$\text{Tile}_{\ h \in \text{Shape}} : (S_0 \times S_1 \times S_2 \times \ \ldots \ \times S_{n-1}) \rightarrow \text{<type>} \tag{2.3}$$

- **View.** Un View almacena un conjunto de transformaciones a Shapes. Una transformación se define por la dimensión en la que se aplica, una acción y un *offset*. Las acciones pueden ser desplazar el primer índice, el último o ambos un número de posiciones indicadas por el *offset* o ajustar los límites del Shape para tener una cardinalidad dada (*offset*). Se implementa en la estructura HitShpView.

- **Topology.** Esta estructura representa una topología virtual que encapsula los detalles de la topología física del sistema. Hay varias topologías disponibles con características diversas y de hasta cuatro dimensiones, y Hitmap cuenta con plug-ins para crearlas solamente a partir del nombre. Se implementa en la clase HitTopology.

- **Layout.** Un Layout distribuye los índices del dominio de un Shape a los procesos de una topología. Se implementa en la clase HitLayout. Hay varios tipos en función de la manera en que se reparten los índices: solo al procesador principal, de manera cíclica o en bloques con diversas características.

- **Comm.** Un comunicador $C$ o Comm representa una comunicación de Tiles entre procesos. Éstas puede ser punto a punto, colectivas, de desplazamiento a lo largo de una dimensión, etc., y están basadas en el estándar MPI. Se implementa en la clase HitCom.

- **Pattern.** Un Pattern $P$ es un patrón de comunicación que está formado por uno o más comunicadores. Se implementa en la clase HitPattern.

$$P \in \text{Pattern} = (C_0, C_1, \ \ldots \ , C_{n-1}) \tag{2.4}$$

Las funcionalidades proporcionadas por Hitmap se dividen en tres categorías:

- ***Tiling.*** Para la definición y manipulación de arrays y *tiles*. Se pueden utilizar de manera independiente a las otras funciones para generar las distribuciones de datos manualmente. En la figura 2.1 se presenta un ejemplo de creación de Tiles. Como se puede ver en el caso de los Tiles B y C, se puede crear un HitTile como una subselección de otro, creando así una jerarquía.

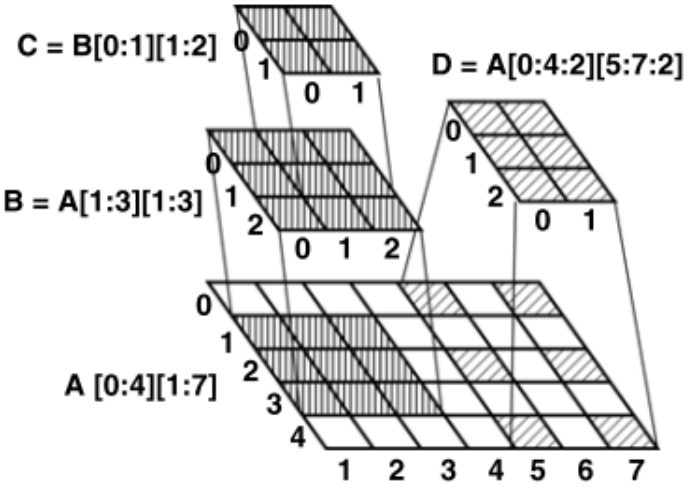



Figura 2.1: Creación de Tiles a partir de un array en Hitmap. Imagen extraída de [11].

- **Mapeado.** Para el particionado y distribución de datos automático en función de la topología seleccionada. Estas funciones necesitan una topología y un Layout, y devuelven una estructura de datos que contiene el rango de cada Tile, la correspondencia entre Tiles y procesadores virtuales e información sobre los vecinos de cada procesador.
- **Comunicación.** Para crear y ejecutar las estructuras de comunicación Comm y Pattern. Estas estructuras son reutilizables, es decir, sus comunicaciones se pueden efectuar múltiples veces pero siempre con las mismas características (mismo emisor, receptor, tipo ...).

En la figura 2.2 se muestra el diagrama de clases de la librería. Las clases con fondo blanco son funcionalidades de tipo *tiling*, las gris claro son de mapeado y las gris oscuro de comunicación.

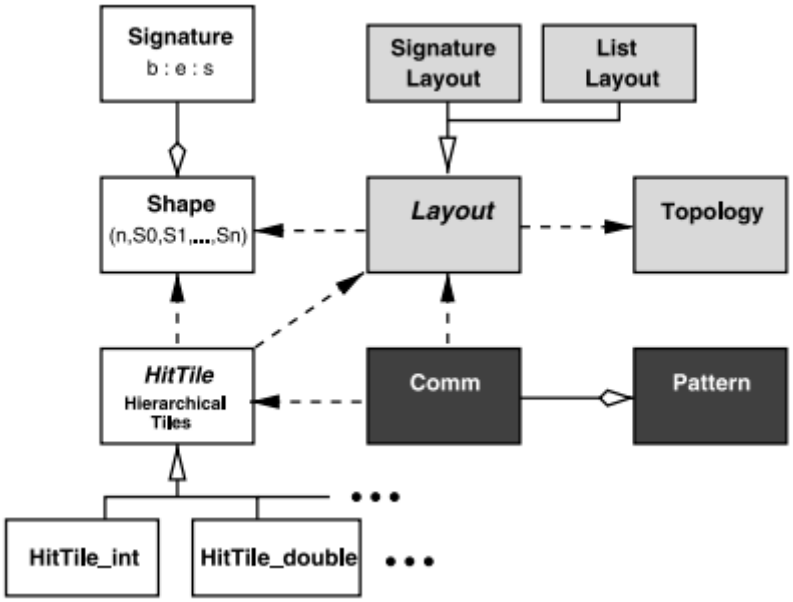



Figura 2.2: Diagrama UML de la arquitectura de Hitmap. Imagen extraída de [11].

### 2.1.3. Versiones

Actualmente, Hitmap cuenta con dos versiones, la v1.3 y la v2.0, que se pueden utilizar de forma individual o conjunta en el mismo programa. La versión 2 está todavía en desarrollo, pero tiene varias funcionalidades disponibles. Hitmap2 facilita el uso de la librería con macros que simplifican las funciones de Hitmap1 (creación de estructuras, acceso a elementos, comunicaciones, liberación de memoria, etc.) con nombres más sencillos y menos parámetros. Hitmap2 introduce un nuevo tipo de Tile distribuido, que se construye a partir de un Layout. De esta manera, el Tile tiene el dominio correspondiente al Shape local de un proceso en en Layout, y se puede obtener a partir de qué Layout se ha construido un Tile, cosa que no se puede hacer en Hitmap1.

En los códigos 2.1 y 2.2 se muestra un ejemplo del mismo programa realizado con las versiones 1.3 y 2.0 de Hitmap, respectivamente. Se puede ver que la creación de las variables es más sencilla en Hitmap2. En la creación del Shape y el acceso a una posición del Tile, no hay que especificar el número de dimensiones (parámetros con valor 2), sino que la función cuenta el número de parámetros y deriva a una función u otra de Hitmap1.

```
1  #include <hitmap.h>
2  hit_tileNewType(int);
3
4  int main() {
5      int rows = 3, columns = 5, value = 5;
6      HitShape sh = hit_shapeStd(2, rows, columns);
7      HitTile_int tile;
8      hit_tileDomainShapeAlloc(&tile, int, sh);
9      hit_tileFill(&tile, &value);
10     int data1_5 = hit_tileElemAt(tile, 2, 1, 5);
11     hit_tileFree(tile);
12 }
```

Código 2.1: Ejemplo de uso de Hitmap v1.3.

```
1  #include <hitmap2.h>
2  hitNewType(int);
3
4  int main() {
5      int rows = 3, columns = 5, value = 5;
6      HitShape sh = hitShape((rows), (columns));
7      HitTile_int tile = hitTile(int, sh);
8      hit_tileFill(&tile, &value);
9      int data1_5 = hit(tile, 1, 5);
10     hit_tileFree(tile);
11 }
```

Código 2.2: Ejemplo de uso de Hitmap v2.0.

### 2.1.4. Funcionalidades útiles para el proyecto

Dentro de las estructuras y funcionalidades mencionadas en la sección 2.1.2, hay varias específicas para el balanceo de carga que pueden ser de utilidad para el desarrollo de WeightedRedistribute. Para la representación de una división de pesos entre procesos se utiliza la estructura HitWeights. Se basa en un array en el que cada posición es el peso asignado al proceso con el rango igual a esa posición. Para el reparto del dominio del problema en bloques definidos por pesos, Hitmap dispone de dos Layouts: `layWeighted_Blocks` y `layWeighted_Copy`. Para construirlos se necesita una topología, un Shape que representa el dominio global (ambos son factores comunes a todos los tipos de Layouts), un índice de dimensión y un HitWeights. El reparto en base a los pesos se aplica solo en la dimensión indicada. En el resto, se divide el dominio en bloques según la topología, en el caso de `layWeighted_Blocks`, o se da todo el dominio a cada proceso, en el caso de `layWeighted_Copy`. El número de pesos en el HitWeights debe ser igual al número de procesos activos en la dimensión indicada, que también se denomina dimensión restringida.

Hitmap tiene dos funciones que redistribuyen datos de un Tile a otro que pueden utilizarse para el balanceo adaptativo, pues cuando se creen las estructuras con los pesos actualizados habrá que trasladar los datos de los Tiles antiguos a los nuevos. Una función distribuye datos entre dos Tiles que pertenecen al mismo Layout (`hit_patternRedistributeCom()`) y otra entre entre Tiles de dos Layouts diferentes (`hit_patternLayRedistribute()`). Ambas devuelven una estructura de comunicación Pattern que al ejecutarse realiza la transferencia.

### 2.1.5. Justificación de uso

Antes del inicio de este proyecto, Hitmap incluía elementos que permitían el reparto de peso y datos entre procesos, como Layouts con balanceo de carga y funciones de distribución de datos entre Tiles. Los Tiles distribuidos incorporados en Hitmap2 simplifican la creación de Tiles a partir de los Layouts con pesos y el manejo y la distribución de datos de unos a otros. Añadir un reparto automático de la carga permitirá explotar todas las funcionalidades de Hitmap y podrá aplicarse a todos los ejemplos que utiliza el Grupo Trasgo para sus experimentaciones, lo que proporcionará otro enfoque de mejora de rendimiento.

## 2.2. MPI

MPI (*Message-Passing Interface*) [18] es un estándar de librería de interfaces del modelo de paso de mensajes de la computación paralela. Se creó a principios de los años 90 por un grupo de 60 personas pertenecientes a organizaciones de Europa y Estados Unidos. Es adecuado para programas MIMD (*Multiple Instructions Multiple Data*) [4] o SPMD (*Single Program Multiple Data*) [19].

La programación con paso de mensajes plantea un sistema con memoria distribuida (no hay memoria compartida, sino que cada proceso tiene su propia memoria local) y múltiples procesos que se comunican entre sí a través del paso de mensajes. Un punto clave de este paradigma es que una transmisión de datos de la memoria de un proceso a la de otro requiere que ambos ejecuten una operación [20]. MPI es una especificación del paradigma que tiene múltiples implementaciones

y es portable a otros lenguajes, como Fortran, C, C++, MATLAB o Python. Incluye definiciones para comunicaciones punto a punto, colectivas y unidireccionales; tipos de datos, agrupación de procesos, gestión de contextos y entornos de comunicación, creación y gestión de procesos, topologías de procesos, operaciones de entrada/salida a ficheros en paralelo, acceso a memoria remota y soporte para otras herramientas, como *debuggers* o analizadores de rendimiento.

MPI es empleado por Hitmap en la parte de comunicaciones. Las funciones y macros tanto de la versión 1.3 como la 2.0 utilizan elementos de MPI para lenguaje C por debajo para manejar el entorno de la comunicación, las transimisiones de datos y las topologías.

# Capítulo 3

# Diseño de la solución

En este capítulo se introducen los siguientes aspectos:

- El diseño inicial del funcionamiento de la función.
- El prototipo de la cabecera y el algoritmo de la función para Hitmap v1.3.
- El estudio de tipos de medias llevado a cabo para la inclusión de una media en la función.

## 3.1. Diseño inicial

En este proyecto se propone la siguiente funcionalidad: a partir de una distribución inicial de los datos a ser computados entre varios procesadores, se redistribuyen de tal manera que a los procesos más rápidos se les asigne una mayor cantidad de datos –y por consiguiente más trabajo– que a los procesos lentos. La previsión es que tras una serie de redistribuciones se alcanzará un equilibrio en el tiempo que tarda un proceso en realizar una iteración de trabajo y ya no será necesario redistribuir más.

Se realiza una medición del tiempo que tarda cada proceso en ejecutar una iteración varias veces y se calcula una media. De esta manera, podemos conocer aquellos procesos que son más rápidos bajo una misma plataforma (esto es, los que tardan menos en ser ejecutados) y podrían cargar con más trabajo y cuáles son más lentos y necesitan liberar algo de carga. Para medir ese tiempo se utilizará una variable estática de tipo HitClock, que empezará a contar al final de la función y parará al principio de cada llamada. Que sea estática hace que se pueda inicializar en la primera llamada a WeightedRedistribute, no se destruya cuando retorna y mantenga su valor en las siguientes llamadas. Después de computar la media de tiempos, cada proceso la transmitirá a los demás a través de una comunicación colectiva tipo *All-to-All* (figura 3.1) para que al final todos los procesos conozcan todos los tiempos. Tras esto, los procesos calcularán los nuevos pesos normalizando esas medias y los utilizarán para crear el nuevo Layout. Por último, se obtiene el Shape que representa la porción del Layout asignada a ese proceso según su peso, se crea un Tile con ese Shape y se realiza la redistribución de datos del Tile antiguo al nuevo con la función de Hitmap que distribuye datos entre dos Tiles de diferentes Layouts (`hit_patternLayRedistribute()`).

La función WeightedRedistribute no tendrá en todos los casos el mismo comportamiento. Por ejemplo, no se hará la distribución si no se han recogido suficientes mediciones para hacer la media de tiempos o si ya se ha alcanzado el equilibrio en los pesos. La situación de equilibrio se

establecerá cuando los pesos de cada proceso se estabilicen, es decir, cuando hayan seguido una tendencia clara durante varias iteraciones y sus valores varíen cada vez menos.

Los parámetros básicos de WeightedRedistribute son un array de punteros a Tiles, para poder distribuir cualquier número de ellos, y un puntero a Layout. Los Tiles deben estar contenidos en él para poder distribuirse completamente, si no solo se distribuirá el trozo que esté dentro del dominio del Layout. Los parámetros se pasan por referencia para modificar esas estructuras con los valores nuevos y no crear otras y retornarlas al final de la función.

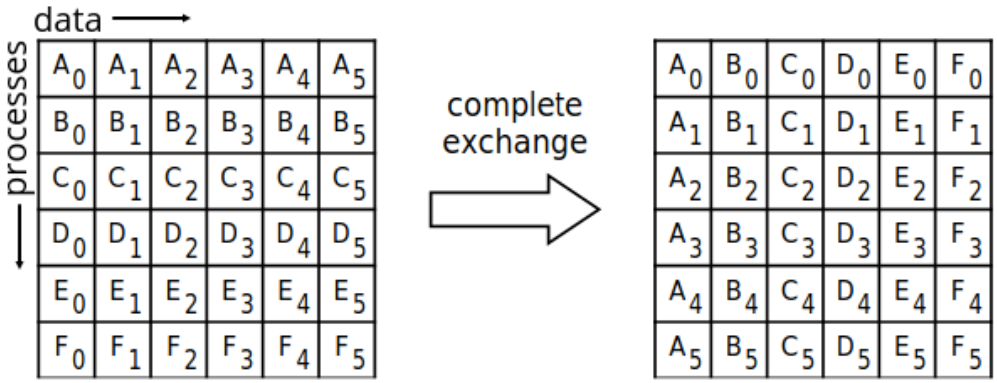



Figura 3.1: Ilustración de una comunicación colectiva *All-to-All* en MPI.
Imagen extraída de [18].

### 3.1.1. Cálculo de pesos

Para el cálculo del peso de cada proceso se parte de las medias de tiempos de iteración de todos los procesos. En una normalización directa, se obtendría el peso de un proceso dividiendo su media entre la suma total como indica la fórmula 3.1, de tal forma que la suma total de pesos sería 1. Sin embargo, lo que se pretende es que los procesos con tiempos más altos tengan pesos menores, para que en la siguiente iteración tengan menos trabajo. Por tanto, se calcula el inverso del peso (fórmula 3.2).

$$w_i = \frac{t_i}{\sum_{i=1}^{nProcs} t_i} \qquad (3.1) \qquad\qquad w'_i = 1 - \frac{t_i}{\sum_{i=1}^{nProcs} t_i} \qquad (3.2)$$

La suma de los $w'_i$ no es 1, sino $nProcs - 1$. El constructor de Layouts puede trabajar con arrays de pesos cuya suma es mayor que 1, pero se desea tener los pesos dentro de un rango y saber la proporción del total que tiene cada proceso. Para ello, se vuelven a normalizar los pesos $w'_i$ utilizando la fórmula 3.1.

### 3.1.2. Elección de tipo de Layout y Topología

Hitmap cuenta con dos tipos de Layouts con pesos: `layWeighted_Blocks` y `layWeighted_Copy` (sección 2.1.4). Para WeightedRedistribute se elige utilizar el Layout con peso y bloques restringido

en la dimensión 0, para dividir la segunda dimensión en función de la topología.

Hitmap dispone de varias topologías de hasta cuatro dimensiones, pero dado que la mayoría de las aplicaciones trabajan con estructuras de una o dos dimensiones solo se discutirán esos tipos. En una topología 1D (figura 3.2a) se realiza el reparto por pesos solo en una dimensión y el resto de dimensiones se copian enteras en cada proceso. Por el contrario, en una topología 2D (figura 3.2b) se dividiría por pesos en ambas dimensiones, dando lugar a bloques de diferentes alturas y anchuras. Este es el comportamiento ideal que se desea en la función; sin embargo, actualmente Hitmap no permite un reparto tan irregular. El resultado utilizando cualquiera de las topologías 2D de Hitmap sería el de la figura 3.2c: todos los procesos con el mismo índice en la dimensión 0 (procesos 0, 2, 4) tienen la misma altura, y los procesos con el mismo índice en la dimensión 1 (procesos 0, 1) tienen la misma anchura. Esta también podría ser una división válida, pero puede haber problemas en la creación del Layout por causa del array de tiempos normalizados.

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 | 4 |

(a) Topología 1D

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 2 | 2 | 5 |
| 0 | 0 | 3 | 3 | 5 |
| 0 | 0 | 3 | 3 | 5 |
| 0 | 0 | 4 | 4 | 5 |
| 1 | 1 | 1 | 1 | 5 |

(b) Topología 2D (ideal)

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 2 | 2 | 4 |
| 0 | 0 | 2 | 2 | 4 |
| 0 | 0 | 2 | 2 | 4 |
| 1 | 1 | 3 | 3 | 5 |
| 1 | 1 | 3 | 3 | 5 |

(c) Topología 2D (Hitmap)

Figura 3.2: Ejemplo de reparto de datos de un Tile 2D con un Layout con pesos y diferentes topologías.

Uno de los parámetros para construir un Layout de tipo *Weighted* es un HitWeights que representa la división de pesos mediante un array. El array debe tener un número de pesos igual al número de procesos activos en la dimensión restringida del Layout. Por ejemplo, en el caso de la figura 3.2a habría que indicar cinco pesos, y en el caso de la figura 3.2c, tres. Podrían utilizarse solo los pesos de los procesos 0, 2 y 4, pero si los tres tuviesen valor nulo se produciría un error al crear el Layout. También, se podría utilizar la suma de todos los procesos en sus "columnas" $(0+1, 2+3, 4+5)$. En esta ocasión, aunque se evita tener un array con todos los valores cero, al dar el mismo peso a todos los procesos de una misma columna puede haber procesos que reciban un bloque cuando por su peso individual no debieran tenerlo.

En este proyecto se utilizará un Layout con división de pesos y bloques y una topología plana, ya que con otro tipo no siempre se obtendrán resultados fiables. Se deja como trabajo futuro la implementación con una topología de dos dimensiones.

## 3.2. Prototipo para Hitmap v1.3

El código 3.1 muestra la cabecera de WeightedRedistribute para Hitmap versión 1. El array de Tiles es del tipo genérico para poder usar Tiles de cualquier tipo base (int, double ...). El planteamiento inicial del algoritmo se puede ver en el algoritmo A.1 en el anexo A.2.

```
1 void weightedRedistribute(HitTile * ttile[], int numTiles,
2     HitLayout * lay);
```

Código 3.1: Prototipo inicial de la cabecera de WeightedRedistribute (Hitmap v1.3).

## 3.3. Estudio de medias

Esta sección incluye un estudio de algunas opciones existentes a la hora de elegir un tipo de media para los tiempos de ejecución. Se decide utilizar una media móvil en lugar de una "estática" por el carácter temporal de los datos, ya que se quiere esperar a tener un número suficiente de ellos para calcularla y poder actualizarla a medida que llega un dato nuevo.

**Definición.** La media móvil [21] es un tipo de análisis de datos que consiste en crear una serie de promedios, cada uno correspondiente a un subconjunto de los datos totales, de un tamaño establecido denominado ventana. Estos promedios se utilizarán para predecir el comportamiento creciente o decreciente del conjunto. Cada vez que llega un valor nuevo, la ventana avanza una posición y se recalcula la media añadiendo el nuevo dato y quitando el más antiguo.

Un inconveniente de este tipo de media es que es un indicador de tipo *lagging*, es decir, que se basa en datos ya pasados, por lo que hay un retraso entre que se produce un cambio en tiempo real en la tendencia de los datos y la media lo detecta. Cuanto mayor es el tamaño de la ventana, mayor es el *lag*. Por otra parte, la media móvil es útil si hay una tendencia más o menos definida en los datos, pero da poca información si los valores son muy oscilantes.

### 3.3.1. Tipos de medidas

Dentro de la media móvil existen varias variantes según el número de elementos que se incluyen en el cálculo o el peso que se asigna a cada uno. Las que se han considerado para este estudio son las siguientes:

#### Media Móvil Simple (SMA)

La media móvil simple [22] es una media aritmética básica sin pesos. Puede ser calculada con datos anteriores (media móvil previa) o con datos anteriores y posteriores en torno a un valor central, el mismo número en ambos sentidos (media móvil central). En este caso se ha escogido la media móvil previa.

La fórmula de la SMA es la siguiente, siendo $d_i$ el dato del día $i$ y $n$ el ancho de la ventana:

$$SMA = \frac{d_1 + d_2 + \quad \ldots \quad + d_n}{n} \tag{3.3}$$

Esta media da mucha más importancia a los datos pasados que otros tipos de medias móviles, lo cual dependiendo del contexto del estudio puede no ser lo más adecuado.

### Media Móvil Acumulativa (CMA)

La media móvil acumulativa o media incremental [23] no utiliza una ventana. En el cálculo se incluyen todos los datos desde el primero hasta el día $n$.

$$CMA = \frac{d_1 + d_2 + \ \ldots \ + d_n}{n} \tag{3.4}$$

Para calcular las siguientes CMAs no es necesario repetir la suma global o almacenarla, se puede calcular iterativamente mediante la CMA del día anterior:

$$CMA_n = \frac{d_n + n \cdot CMA_{n-1}}{n+1} \tag{3.5}$$

### Media Móvil Con Pesos (WMA)

La media móvil con pesos o ponderada aplica un factor de peso ($w_i$) a cada posición de la ventana. Así, en función de las características de los datos se puede elegir dar más importancia (más peso) a los datos más antiguos, más recientes ... El cálculo de la media ponderada se muestra en la fórmula 3.6.

$$WMA = \frac{d_1 \cdot w_1 + d_2 \cdot w_2 + \ \ldots \ + d_n \cdot w_n}{\sum_{i=1}^{n} w_i} \tag{3.6}$$

Un caso particular de esta media es la **LWMA** (*Linearly Weighted Moving Average*) [24], donde los pesos varían de forma lineal, siendo los pesos grandes para los últimos datos. La LWMA reacciona más rápido a los cambios en los valores que las otras medias.

### Media Móvil Exponencial (EMA)

La media móvil exponencial [25] es otro caso particular de la media móvil con pesos, por eso también se denomina EWMA (*Exponentially Weighted Moving Average*). Aquí los pesos decrecen de manera exponencial sin llegar a 0, dando más importancia a los datos más recientes. En zonas planas, esto es, con valores más o menos parecidos, los valores de la media son similares a los de la LWMA.

La fórmula de la media exponencial para el día $n$ se muestra a continuación. Se basa en la EMA del día anterior y un factor de suavizado, $k$. Para la EMA del día 0 se utiliza la SMA.

$$\begin{aligned} EMA_n &= d_n \cdot k + EMA_{n-1} \cdot (1-k) \\ k &= \frac{2}{\text{ancho de la ventana} + 1} \end{aligned} \tag{3.7}$$

### 3.3.2. Elección del número de elementos en la media

El número de muestras para el cálculo de las medias depende en gran medida del tipo de datos que se están midiendo y del estudio que se está llevando a cabo. Las medias a corto plazo (ventanas pequeñas) tienen más en cuenta los datos nuevos y menos a los datos antiguos, y tienen más sensibilidad a los cambios que las medias a largo plazo (ventanas grandes), pero pueden hacer que no se vean correctamente las tendencias generales. Por el contrario, una ventana grande contiene menos ruido, pero puede ocultar detalles relevantes. Una práctica habitual es trabajar con varios tipos de ventanas o tipos de medias e ir comparando resultados.

### 3.3.3. Conclusiones

Se decide hacer uso de las medias simple, exponencial y linealmente ponderada porque son las que se calculan con un número limitado de elementos. Esto es un requisito del diseño, ya que la media debe incluir tiempos de un mismo valor del peso de un proceso. Una vez que se hace una redistribución, los tiempos antiguos han de desecharse en favor de los asociados a la nueva carga del proceso. Por este motivo, se descarta el uso de la media acumulativa. El tipo y el número de datos en la media serán configurables mediante parámetros del programa, y serán las pruebas experimentales las que determinen los valores óptimos de cada uno para este problema.

## 3.4. Conclusiones del capítulo

En este capítulo se ha descrito el primer diseño de la función WeightedRedistribute, el funcionamiento a grandes rasgos y algunas consideraciones que había que tener en cuenta, como el tipo de Layout y la topología a usar. También, se han presentado los prototipos de la cabecera y el algoritmo desarrollados para la versión 1.3 de Hitmap. Finalmente, se ha introducido una pequeña investigación de variantes de medias móviles llevada a cabo para estudiar qué tipo era más adecuado para utilizar en la función.

# Capítulo 4

# Implementación de la solución

En este capítulo se introducen los siguientes aspectos:

- La elaboración de una estructura de datos para representar la media móvil.
- El programa de ejemplo utilizado para aplicar la función.
- Las variaciones que ha sufrido el diseño inicial de la función.
- La cabecera y el algoritmo de la implementación final de la función para Hitmap v1.3.

## 4.1. La clase HitAvg

La creación de una estructura de datos para representar la media surge de la necesidad de almacenar el número de datos necesario hasta poder calcular el promedio y de poder escoger entre varios tipos. Además, cumple con el requisito de transparencia al estar en una clase aislada.

```
1  typedef struct HitAvg{
2      int windowSize;
3      int pointer;
4      int first;
5      int full;
6      double * data;
7      double * weights;
8      double previousAvg;
9      enum HitAvgMode mode;
10 } HitAvg;
```

Código 4.1: Definición de la estructura HitAvg.

Los atributos básicos de esta estructura (código 4.1) son el tipo, definido por un *enumerate*[1] que puede tomar los valores SMA, EMA y WMA; y los arrays de datos y pesos. Las características de estos elementos se establecen en el constructor y no se pueden cambiar. El vector de datos tiene

[1] *Enumerate*: tipo de datos compuesto por un conjunto de constantes nombradas explícitamente [26]. Una variable definida de ese tipo debe tomar como valor uno de los valores predefinidos.

un comportamiento similar a una cola circular, con dos diferencias: no hay función de extracción y si está completo al insertar un nuevo dato se sobrescribe la primera posición. La asignación de los pesos a los datos viene dada por el orden en el que estén en el array: el peso en la posición 0 será para el elemento más antiguo y el peso en la última posición para el elemento insertado más recientemente. En la figura 4.1 se refleja este funcionamiento: al insertar el dato 4, se remplaza el 1 y los pesos se reajustan para que al nuevo elemento más antiguo (2) le corresponda el primer peso (1). Realmente el vector de pesos no varía, sino que se guarda un puntero a la posición que ocupa el primer dato (atributo `first`) y a partir de él se hace la asignación.

El significado del resto de atributos es el siguiente: `pointer` apunta a la posición en la que se insertará el siguiente elemento; `full` es una variable de tipo *flag* que indica si el array de datos está completamente lleno y se utiliza al calcular la media para comprobar si hay datos suficientes; y `previousAvg` guarda el último valor de la media calculado. Este atributo es especialmente útil para calcular la media exponencial.

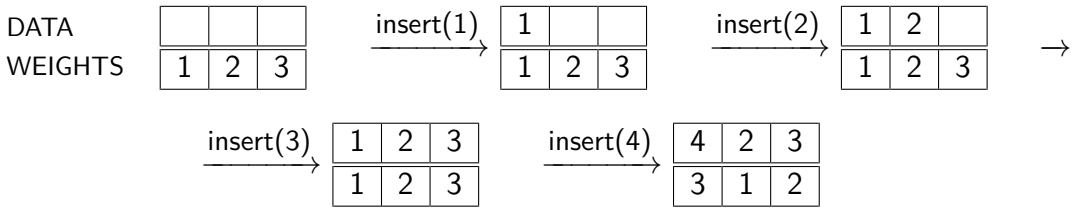



Figura 4.1: Inserción de datos y asignación de pesos en una estructura HitAvg.

Existen cuatro constructores disponibles, uno para cada tipo de media y uno específico para la linealmente ponderada:

- `HitAvg hit_avgSimple(int size)`. El array de pesos se inicializa internamente con valor 1 para todos los datos.
- `HitAvg hit_avgWeighted(int size, double * weights)`. El array de pesos lo establece el usuario. Debe ser al menos del mismo tamaño que la media.
- `HitAvg hit_avgLinearlyWeighted(int size)`. Se hace una distinción con la media ponderada convencional porque en este caso no es necesario el array de pesos como parámetro, ya que se inicializa internamente. Los valores van desde 1 para el primer dato hasta `windowSize`−1 para el último.
- `HitAvg hit_avgExponential(int size)`. El array de pesos se inicializa igual que en la media simple, pero en realidad no es necesario para el cálculo de la media exponencial (fórmula 3.3.1).

El cálculo de la media se realiza tras la inserción de un dato (función `hit_avgInsertData()`) y se almacena en un atributo del objeto, que se puede consultar a través de una función (`hit_avgGetAvg()`). Ambas son funciones genéricas que internamente derivan a otras según el tipo de media.

Un ejemplo de uso de la clase HitAvg se muestra en el código 4.2. En los casos en los que la ventana de datos no está completa, como el primero y el último, el valor por defecto que se devuelve para la media es -1. La función `hit_avgResetData()` devuelve a la estructura a su estado inicial tras la creación, borrando todos los datos pero manteniendo los pesos, el tipo y el tamaño.

```
int main(){

    HitAvg avgsma = hit_avgSimple(10);
    HitAvg avglwma = hit_avgLinearlyWeighted(10);

    hit_avgInsertData(&avgsma, 1);
    hit_avgInsertData(&avglwma, 1);
    printf("sma: %.4lf\n", hit_avgGetAvg(avgsma));    // sma: -1.0000

    for(int i=2; i<11; i++){
        hit_avgInsertData(&avgsma, i);
        hit_avgInsertData(&avglwma, i);
    }
    printf("sma: %.4lf\n", hit_avgGetAvg(avgsma));    // sma: 5.5000
    printf("lwma: %.4lf\n", hit_avgGetAvg(avglwma)); // lwma: 6.1818

    hit_avgResetData(&avgsma);
    printf("sma: %.4lf\n", hit_avgGetAvg(avgsma));    // sma: -1.0000

    return 0;
}
```

Código 4.2: Ejemplo de uso de estructuras HitAvg.

### 4.1.1. Inclusión en la solución

La estructura HitAvg se incluye en WeightedRedistribute como un parámetro (código 4.3). También se pasa por referencia porque se actualiza el array de datos con el tiempo calculado en cada llamada a la función, y cuando hay suficientes datos para computar la media se resetea la estructura tras el cálculo para borrarlos. El usuario tiene que definir una variable de cualquiera de los cuatro tipos de media disponibles, y no debería alterarla a lo largo del programa, pues eso afectaría a la correctitud de la media y por tanto de los pesos. Como trabajo futuro, se plantea que no sea obligatorio que el usuario cree una variable HitAvg si no la necesita más que para la redistribución. Si se introduce `HITAVG_NULL` como parámetro, se utilizaría como media una variable externa de la clase, con unos atributos predefinidos.

```
void weightedRedistribute(HitTile * ttile[], int numTiles,
    HitLayout * lay, HitAvg * avg);
```

Código 4.3: Cabecera de WeightedRedistribute con HitAvg.

## 4.2. Caso de estudio: Difusión del calor con Jacobi 2D

### 4.2.1. Introducción

El caso de estudio que se ha escogido para probar la función WeightedRedistribute es el programa de difusión del calor, que modela la transmisión física de calor sobre una superficie a lo largo del tiempo. Se ha elegido porque es uno de los que más se utilizan en el Grupo de Investigación Trasgo, y por su naturaleza es adecuado para realizar balanceo de carga sin muchas complicaciones técnicas. El programa computa durante un número de iteraciones establecido la estabilidad puntual de la Ecuación Diferencial Parcial de Poisson (PDE) de la difusión del calor para un espacio discreto bidimensional utilizando el método iterativo de Jacobi [27], un algoritmo para resolver sistemas de ecuaciones lineales. El caso de estudio es un *stencil*[2] en el que cada posición de la matriz se actualiza con los valores de las cuatro vecinas (arriba, abajo, izquierda y derecha). En el algoritmo 4.1 se muestra el funcionamiento del programa.

**Algoritmo 4.1:** Algoritmo del problema de difusión del calor con Jacobi2D.

**Datos:** $A$: matriz principal, $B$: matriz copia
$x$: índice de fila, $y$: índice de columna
$numIterations$: nº de iteraciones de la simulación

**inicio**
  **para** $i \leftarrow 0$ **a** *numIterations* **hacer**
    **para** $x, y \in B$ **hacer** $B$[x][y] $\leftarrow$ $A$[x][y]

    **para** $x, y \in A$ **hacer**
      $A$[x][y] $\leftarrow$ ( $B$[x+1][y] + $B$[x-1][y] + $B$[x][y+1] + $B$[x][y-1] )/4
    **fin**
  **fin**
**fin**

### 4.2.2. Aplicación de la función al programa

En el código 4.4 se muestra parte de la implementación del algoritmo anterior en Hitmap. La copia de datos de la matriz principal (tileMat) a la secundaria (tileCopy) al principio de cada iteración se realiza con la función `hit_tileUpdateFromAncestor()`. Para poder utilizar esta función, tileCopy tiene que crearse como una subselección de tileMat (paso 4.1). Tras el cómputo del nuevo valor de las celdas, cada proceso intercambia con sus vecinos los bordes para actualizarlos con los que estos han calculado. Finalmente, se llama a WeightedRedistribute. Se decide colocar la llamada al final para realizar primero todos los cálculos de la iteración, pero se puede colocar en cualquier punto del bucle.

[2]Código *stencil*: tipo iterativo de *kernel* en el que el estado de cada celda de la matriz puede deducirse a partir de su estado previo y los estados de celdas vecinas. La forma de este vecindario es lo que se denomina *stencil* [28].

Al añadir la distribución, cambian tanto el contenido como las dimensiones de los Tiles. Esto afecta a la copia que se hace de tileMat hacia tileCopy (paso 4.4.1 y 4.5). Donde originalmente solo era necesaria la actualización de los datos con `hit_tileUpdateFromAncestor()`, ahora hay que volver a copiar toda la estructura. Esto se puede hacer de dos maneras, igual que en el constructor de tileCopy o usando la función `hit_tileClone()`. En la primera manera, se selecciona todo el dominio de tileMat y se asigna a tileCopy. Después, se libera la memoria del antiguo dominio y se reserva para el nuevo; y por último se copian todos los valores de tileMat a tileCopy. Con la función de clonar, el proceso es bastante parecido: se duplica la memoria de datos del Tile, se reserva el espacio y se copian los elementos del array viejo al nuevo.

Ambas opciones son muy costosas y consumen una cantidad relevante del tiempo del programa. Por ello se proponen dos variaciones para intentar mejorar el rendimiento de esta parte, explicadas en las subsecciones siguientes.

#### 4.2.2.1. Primer enfoque. Iteraciones par-impar

La primera propuesta para sustituir la copia en profundidad es utilizar tileCopy no solo como copia de la matriz principal y realizar parte de los cálculos sobre ella. Se van alternando ambos Tiles como matriz de cálculo y matriz secundaria: en las iteraciones pares se trabaja con tileMat y en las impares con tileCopy. Esto implica que, como tileCopy ya no es una subselección de tileMat, hay que crearla de manera diferente. Se modifica el principio del programa para crearla igual que tileMat y se inicializa también con la función `initMatrix()`. Además, se crea un Pattern para la comunicación entre vecinos de tileCopy que se usará en las iteraciones en las que esta sea la matriz principal. En el código 4.5 se pueden apreciar estos cambios.

Tras la última iteración, se realiza la escritura de los resultados en un fichero. Esto se utiliza para las pruebas de validación de resultados, que comparan el fichero obtenido con el código original frente al generado con el código especificado. Esta escritura se hace sobre los datos de tileMat, y si el número de iteraciones es impar, los últimos datos estarían en tileCopy y el resultado del fichero no sería correcto. Para no modificar la parte de la escritura creando dos casos diferentes igual que en el bucle, porque es código de Hitmap que no forma parte del dominio de este trabajo, se mantiene la copia en profundidad original del paso 4.5, pero se añade que solo se realice si es necesaria, es decir, si las iteraciones son impares.

#### 4.2.2.2. Segundo enfoque. Intercambio de punteros

La segunda manera que se propone mantiene a tileCopy como matriz secundaria y remplaza la copia por un cambio de punteros, que en el fondo hace el mismo efecto (código 4.6). También aplica los cambios respecto al carácter independiente de tileCopy introducidos en el enfoque anterior (pasos 4.1 a 4.3, código 4.5). Usando una estructura auxiliar, tileCopy pasa a apuntar a los datos de tileMat, por lo cual tiene los mismos valores que si se hubiese hecho una copia, pero en otra dirección de memoria; y tileMat apunta a los datos de tileCopy. Esto no es un problema porque en el paso siguiente (4.4.2) se actualizan todos sus valores. Asimismo, el cambio de punteros tiene que realizarse a los patrones de comunicación para actualizar los Tiles sobre los que se ejecutan. La copia en profundidad de la última iteración se sustituye por el cambio de punteros para mantener que los datos más recientes estén en la matriz tileMat.

```
/* 4.1. CREATE AND ALLOCATE LOCAL TILES */
...
hit_tileSelect( &tileMat, &matrix, expandedShape );
hit_tileAlloc( &tileMat );
hit_tileSelect( &tileCopy, &tileMat, HIT_SHAPE_WHOLE );
hit_tileAlloc( &tileCopy );

HitTile * tileArray[2];

/* 4.2. COMMUNICATION PATTERN */
HitPattern neighSync = newPattern(&tileMat, &matLayout);

for (loopIndex = 0; loopIndex < numIter-1; loopIndex++) {

    /* 4.4.1. UPDATE TILE COPY */
    /* Option 1. Subselection and update */
    hit_tileSelect( &tileCopy, &tileMat, HIT_SHAPE_WHOLE );
    hit_tileFree( tileCopy );
    hit_tileAlloc( &tileCopy );
    hit_tileUpdateFromAncestor( &tileCopy );

    /* Option 2. Deep copy */
    // hit_tileFree( tileCopy );
    // hit_tileClone( &tileCopy, &tileMat );

    /* 4.4.2. SEQUENTIALIZED LOOP */
    for ( i=1; i<hit_tileDimCard( tileMat, 0 )-1; i++ )
        for ( j=1; j<hit_tileDimCard( tileMat, 1 )-1; j++ )
            updateCell(
                & hit_tileElemAt( tileMat, 2, i, j ),
                hit_tileElemAt( tileCopy, 2, i-1, j ),
                hit_tileElemAt( tileCopy, 2, i+1, j ),
                hit_tileElemAt( tileCopy, 2, i, j-1 ),
                hit_tileElemAt( tileCopy, 2, i, j+1 )
            );

    /* 4.4.3. COMMUNICATE */
    hit_patternDo( neighSync );

    /* 4.4.4. REDISTRIBUTE */
    tileArray[0] = &tileMat;
    tileArray[1] = &tileCopy;
    weightedRedistribute(tileArray, 2, &matLayout);
}

/* 4.5. LAST ITERATION UPDATE: NO COMMUNICATION AFTER */
// Steps 4.4.1 and 4.4.2
```

Código 4.4: Código de Jacobi2D con copia en profundidad.

```
/* 4.1. CREATE AND ALLOCATE LOCAL TILES */
...
hit_tileSelect( &tileMat, &matrix, expandedShape );
hit_tileAlloc( &tileMat );
hit_tileSelect( &tileCopy, &matrix, expandedShape );
hit_tileAlloc( &tileCopy );

/* 4.2. COMMUNICATION PATTERN */
HitPattern neighSync = newPattern(&tileMat, &matLayout);
HitPattern neighSyncCopy = newPattern(&tileCopy, &matLayout);

/* 4.3. INITIALIZE MATRIX (IN PARALLEL) */
initMatrix( tileMat );
initMatrix( tileCopy );

/* 4.4. COMPUTATION LOOP */
for (loopIndex = 0; loopIndex < numIter-1; loopIndex++) {

    /* 4.4.2. SEQUENTIALIZED LOOP */
    if (loopIndex % 2 == 0){
        for ( i=1; i<hit_tileDimCard( tileMat, 0 )-1; i++ )
            for ( j=1; j<hit_tileDimCard( tileMat, 1 )-1; j++ )
                updateCell(
                    & hit_tileElemAt( tileMat, 2, i, j ),
                    hit_tileElemAt( tileCopy, 2, i-1, j ),
                    hit_tileElemAt( tileCopy, 2, i+1, j ),
                    hit_tileElemAt( tileCopy, 2, i, j-1 ),
                    hit_tileElemAt( tileCopy, 2, i, j+1 )
                );
    }
    else {
        for ( i=1; i<hit_tileDimCard( tileCopy, 0 )-1; i++ )
            for ( j=1; j<hit_tileDimCard( tileCopy, 1 )-1; j++ )
                updateCell(
                    & hit_tileElemAt( tileCopy, 2, i, j ),
                    hit_tileElemAt( tileMat, 2, i-1, j ),
                    hit_tileElemAt( tileMat, 2, i+1, j ),
                    hit_tileElemAt( tileMat, 2, i, j-1 ),
                    hit_tileElemAt( tileMat, 2, i, j+1 )
                );
    }

    /* 4.4.3. COMMUNICATE */
    if(loopIndex % 2 == 0)  hit_patternDo( neighSync );
    else    hit_patternDo( neighSyncCopy );

    ...
}
...
```

Código 4.5: Código de variación de Jacobi2D con iteraciones par-impar.

```
HitTile_double tmp;
HitPattern tmpPat;

/* 4.4. COMPUTATION LOOP */
for (loopIndex = 0; loopIndex < numIter-1; loopIndex++) {

    /* 4.4.1. UPDATE TILE COPY */
    tmp = tileMat;
    tileMat = tileCopy;
    tileCopy = tmp;

    tmpPat = neighSync;
    neighSync = neighSyncCopy;
    neighSyncCopy = tmpPat;

    /* 4.4.2. SEQUENTIALIZED LOOP */
    for ( i=1; i<hit_tileDimCard( tileMat, 0 )-1; i++ )
        for ( j=1; j<hit_tileDimCard( tileMat, 1 )-1; j++ )
            updateCell(
                & hit_tileElemAt( tileMat, 2, i, j ),
                hit_tileElemAt( tileCopy, 2, i-1, j ),
                hit_tileElemAt( tileCopy, 2, i+1, j ),
                hit_tileElemAt( tileCopy, 2, i, j-1 ),
                hit_tileElemAt( tileCopy, 2, i, j+1 )
                );

    /* 4.4.3. COMMUNICATE */
    hit_patternDo( neighSync );
    ...
}
...
```

Código 4.6: Código de variación de Jacobi2D con intercambio de punteros.

#### 4.2.2.3. Conclusiones

Los dos enfoques descritos en las subsecciones anteriores, las iteraciones par-impar y el intercambio de punteros, buscan mejorar el rendimiento del programa evitando una costosa copia en profundidad. El primer enfoque elimina la copia modificando el modo de trabajo y el segundo la realiza de un modo más eficaz. Si bien se apuesta por el cambio de punteros como el más eficiente de los dos, no está totalmente claro, por lo que se probarán ambos en la experimentación y serán los resultados los que lo determinen.

## 4.3. Desviaciones del diseño inicial

En esta sección se explican los cambios que se han añadido a la función WeightedRedistribute que no estaban en en el diseño inicial. Algunos cambios han sido para añadir funcionalidad y otros para solventar problemas que no se habían previsto. De este último grupo, varios se han detectado a raíz de la adaptación de la función al programa de Jacobi.

### 4.3.1. Patrones de comunicación

Inicialmente, no se tuvo en cuenta a los patrones de comunicación, o Patterns, ya que la versión básica del programa no incluía más comunicaciones entre procesos que la redistribución. Estos patrones están compuestos por uno o varios comunicadores, que tienen asociados los Tiles y los Layouts que forman parte de la comunicación. Cuando estos se actualizan en la función, el Pattern no lo hace, por lo que las comunicaciones se seguirán ejecutando en base a los Tiles iniciales y los resultados serán incorrectos. Para resolverlo se incluye un array de punteros a Patterns como parámetro igual que con los Tiles. Cada Pattern está asociado con un Tile en correspondencia 1-1, de tal manera que el Pattern de la posición 0 se encarga de la comunicación del Tile de la posición 0, el Pattern 1 de la del Tile 1 y así sucesivamente. Si un Tile no tiene patrón asociado se puede poner un `HIT_PATTERN_NULL` en la casilla correspondiente o colocar el Tile al final de su array y poner un número menor en el parámetro que indica el número de patrones.

Para la actualización de los Patterns surge un problema. Inicialmente se copiaron las líneas de creación del Pattern dentro de la función, pero esa es una manera específica para el caso Jacobi. Para tener una creación de Patterns genérica para cualquier programa se añade una función de creación como parámetro. Se define externamente un tipo, `HitPatternFunction`, que representa una función que recibe dos punteros, uno a HitTile y uno a HitLayout y devuelve un HitPattern, y se incorpora como parámetro de WeightedRedistribute. Así, el usuario podrá definir su propia función siempre y cuando cumpla ese formato. Además, tiene que asignarla a una variable del tipo definido, que será la que pase como parámetro a la función (código 4.7):

La declaración de la función actualizada con el uso de patrones se expone en el código 4.8.

```
HitPattern myPattConstructor (void * tile, HitLayout * lay){...}

int main(){
    HitPatternFunction pattf = myPattConstructor;
    ...
}
```

Código 4.7: Declaración de función de creación de Patterns.

```
void weightedRedistribute(HitTile * ttile[], int numTiles,
    HitLayout * lay, HitAvg * avg, HitPattern * patt[],
    int numPatterns, HitPatternFunction pattf);
```

Código 4.8: Cabecera de WeightedRedistribute con Patterns.

### 4.3.2. Expansión de Shapes e inicialización de Tiles

Los dos cambios de este apartado están relacionados y ambos surgen debido a comportamientos específicos de Jacobi. En este programa, el borde de la matriz grande original no se actualiza en las iteraciones, por lo que el Layout se comprime una posición hacia dentro para no tenerlo en cuenta. Por otra parte, las matrices locales están formadas por los datos de cada proceso más un borde exterior con los datos de los vecinos, que se utilizan en el cálculo del nuevo valor de cada celda. La adición del borde se hace expandiendo el Shape con el dominio local una posición hacia fuera en ambas dimensiones. La figura 4.2 muestra esta expansión. El solapamiento de dominios entre varios procesos se representa con la mezcla de colores.

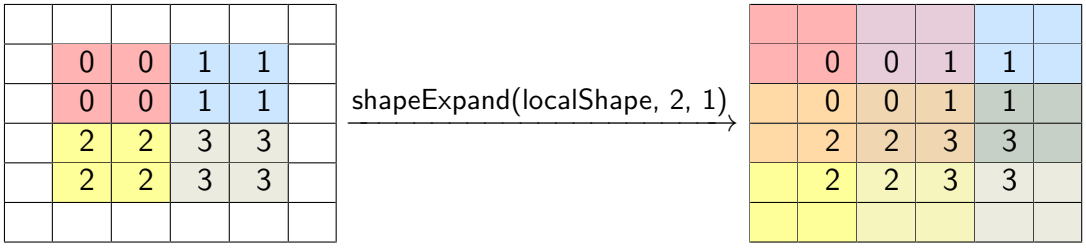



Figura 4.2: Dominio de cada proceso antes y después de la expansión de Shapes.

En WeightedRedistribute no se hacía la expansión del nuevo Shape, lo que producía Tiles incorrectos u obligaba al usuario a realizarla fuera de la función, contradiciendo el requisito de mínima intrusión. Para evitar esto se añade la expansión después de la obtención del Shape local con los nuevos pesos y se crea el nuevo Tile a partir de ese Shape expandido. Para hacerlo de una manera genérica se incluye un parámetro HitShpView. El usuario almacenará en él las transformaciones que quiere que se apliquen al Shape, o `HIT_SHPVIEW_NULL_STATIC` si no hay ninguna.

El hecho de que el Layout sea más pequeño que el Tile original plantea un problema porque solo se distribuye la parte de un Tile que está contenida en el Layout. El borde exterior, si bien no se va actualizando, es necesario para el cálculo de las celdas adyacentes; y al no participar en la redistribución su valor en los nuevos Tiles es erróneo. La solución a esto tiene dos partes. En

primer lugar, hay que inicializar esos bordes, y para ello se decide utilizar un parámetro función igual que se hizo con los Patterns. Este método recibe un puntero a HitTile y da un valor inicial a sus celdas. Se invocará tras la creación del nuevo Tile pero antes de la redistribución. Así, en los procesos que tengan borde exterior este será correcto, y no afectará al resto de procesos porque sus bordes se sobrescribirán después. En segundo lugar, hay que comunicar los bordes vecinos después de recrear el Pattern con los nuevos Layout y Tile. Esto se hace porque la función `redistribute()` de Hitmap no trabaja bien con celdas que pertenecen a más de un proceso. Los bloques que distribuye son como los Shapes antes de la expansión, cuando no había solapamiento de celdas. Comunicando los vecinos de nuevo se actualiza el valor de esas casillas. El código 4.9 muestra la cabecera de la función actualizada con estos dos cambios.

```
1 void weightedRedistribute(HitTile * ttile[], int numTiles,
2     HitLayout * lay, HitAvg * avg, HitPattern * patt[],
3     int numPatterns, HitPatternFunction pattf,
4     HitTileInitFunction tilef, HitShpView shpview);
```

Código 4.9: Cabecera de WeightedRedistribute con HitShpView y función de inicialización de Tiles.

### 4.3.3. Layout de comunicaciones y tiempo de procesos inactivos

En Hitmap, cuando un proceso deja de estar activo en un Layout se elimina de él, se reasigna la numeración de los procesos en ese Layout y se deja de tener en cuenta al proceso en las operaciones que se realizan sobre él. En este trabajo es preciso que todos los procesos, incluso los inactivos, participen en el *All-to-All* para poder crear los nuevos Layouts con los pesos de todos. Antiguamente, el *All-to-All* se hacía sobre el Layout pasado como parámetro, es decir, el que divide los datos, y cuando había procesos que se quedaban sin bloque se producían errores en la comunicación.

Para que todos procesos participen en todas las comunicaciones se declara un Layout estático que se inicializa en la primera llamada a la función (al comprobar si la variable HitClock sigue siendo nula) con un Shape que contiene un solo índice, pues su dominio no es importante. Al definirse al principio del programa y no tener modificaciones, se asegura que todos los procesos estén siempre activos. La transmisión de tiempos y otras operaciones que requieran a todos los procesos deben ejecutarse sobre él.

Otro problema concierne al tiempo de los procesos inactivos. Sea $P_i$ un proceso inactivo y $P_a$ un proceso activo. Aunque $P_i$ tenga asignado un bloque nulo, genera un tiempo de iteración mayor que 0, aunque sea un valor cercano. Esto hace que su peso calculado también sea mayor que 0 y que, en casos con tamaños de la matriz muy pequeños, sea similar al peso de los $P_a$, porque estos tardan también muy poco en computar su bloque. Al generar el nuevo Layout, hay celdas que se quedan sin asignar a un proceso porque por su peso los procesos activos calculan que van a ir a parar a $P_i$, pero al estar inactivo en el Layout anterior ya no vuelve a participar en el programa nunca más. Esto hace que fallen las comunicaciones entre vecinos porque un proceso espera a recibir datos de una celda que no pertenece a nadie, por lo que no puede ser enviada. La solución es comprobar si un proceso está inactivo en el Layout que se pasa como parámetro, y si es así

insertar no el tiempo de la iteración anterior, sino 0. De esta manera, su media será 0 y el resto de los $P_a$ también le asignarán un bloque nulo, repartiéndose entre ellos toda la matriz.

### 4.3.4. Posición de la llamada a la función en el caso de estudio

La ejecución de los Patterns en WeightedRedistribute solo es realmente necesaria si hay solapamiento de celdas y procesos, y supone un añadido al tiempo total del programa. Viendo que se hace una ejecución justo antes de la llamada a la función WeightedRedistribute y otra al final de esta, se valora si existe una manera para evitar una de las dos. La solución es hacer en el *main* la llamada a la redistribución después del cómputo de las nuevas celdas y antes del Pattern. Los bordes no se comunican en la redistribución, luego no importa no tener el valor correcto en ese momento. De esta forma, se puede quitar la comunicación de vecinos tras la actualización de Patterns en WeightedRedistribute y dejar solo la de la función *main*, lo que permite mejorar un poco el tiempo total del programa. Se informará al usuario del comportamiento de la función en cuanto a celdas compartidas por varios procesos, de manera que es tarea suya gestionar los valores de esas posiciones si se da esa situación.

### 4.3.5. Medición de tiempos

Como resultado de las primeras pruebas de rendimiento (sección 5.3.3.1), se vio que en muchos casos los procesos recibían pesos iguales o con diferencias muy pequeñas a pesar de pertenecer a máquinas de diferente potencia, cuando lo esperado era que hubiese desigualdades más notables. Eso hizo plantear que la forma de asignar los pesos según el tiempo de una iteración quizá no era adecuada.

En una iteración de Jacobi hay cuatro partes: la copia entre estructuras secundarias y principales, el cómputo de las celdas, la comunicación entre vecinos y la llamada a la redistribución. El tiempo que se pasa como parámetro a la función es el de cómputo más el de comunicación, y esa es la causa de las diferencias tan pequeñas. En la comunicación se produce una sincronización implícita porque los procesos tienen que esperar a que sus cuatro vecinos acaben de calcular las nuevas celdas y se las envíen. Entonces, un proceso que ha sido muy rápido en la parte de cómputo se contrarresta tardando más en la comunicación porque tiene que esperar a los procesos más lentos; y un proceso que tarda mucho en los cálculos tiene poco tiempo de comunicación porque sus vecinos ya le han enviado los datos. La solución es utilizar para la media solo el tiempo de cálculo, pues es ahí donde se distingue la rapidez de los procesos. Para ello se pasa ese tiempo a la función como un parámetro en vez de utilizar un HitClock para medir el tiempo de toda la iteración. Igual que en el apartado anterior, si el proceso no está activo el tiempo que se inserta en la media es 0, para no generar contratiempos a la hora de repartir los pesos.

### 4.3.6. Comunicaciones no bloqueantes

Una manera de mejorar el rendimiento de un sistema es solapar la computación y las comunicaciones y esto puede hacerse utilizando comunicaciones de tipo no bloqueante [18]. En las comunicaciones bloqueantes, un proceso se bloquea hasta que la operación se ha completado (envío,

recepción . . . ). Por el contrario, en las no bloqueantes se inicia la operación en segundo plano y se continúa con las instrucciones siguientes, pudiendo comprobar más adelante si ésta ha terminado.

La comunicación que se decide transformar en no bloqueante es el *All-to-All*. Una vez que el array de datos de la media está completo, se inicia el *All-to-All* y la función retorna al *main*. Se solapan el cálculo de la iteración siguiente y la transmisión de medias y en la siguiente llamada a WeightedRedistribute se espera a que la comunicación se complete y se prosigue con el cálculo de pesos y la creación de las nuevas estructuras. El Tile donde se reciben las medias de todos los procesos se declara como estático para que se mantengan sus valores de una llamada a otra.

Con el objetivo de que la distribución se haga el mismo número de veces que antes se modifica la media para que tenga un elemento menos que el indicado por el usuario. Por ejemplo, si el tamaño de media especificado es 10, con una media de 10 datos anteriormente comunicación y distribución se harían en la iteración 9. Ahora, la comunicación empezaría en la iteración 9 pero la distribución se haría en la siguiente llamada, la 10. Tener el mismo número de redistribuciones en las versiones bloqueante y no bloqueante permite compararlas mejor, ya que una diferencia afectaría al tiempo de ejecución total. Aunque se haya sacrificado tener un dato más en la media, no afectará mucho a su valor, pues todos los tiempos de cómputo suelen ser bastante parecidos.

### 4.3.7. Variantes de cálculo de pesos

Tras aplicar el cambio del cálculo de la media en base solo a los tiempos de cómputo de las celdas, se observó (sección 5.3.3.1) que las diferencias de peso entre las máquinas eran más grandes, pero seguían apareciendo picos en muchos casos. Por otra parte, los tiempos de ejecución de las iteraciones aumentaban con cada redistribución en vez de estabilizarse. Estos son indicios de que el reparto de pesos sigue sin ser correcto, por lo que se plantean varias alternativas para calcularlos.

La fórmula 4.1 representa una nueva manera de calcular los pesos como una proporción inversa de la media del tiempo. No normaliza las medias para que la suma sea 1 como la manera inicial (fórmulas 3.1 y 3.2), pero es una fórmula más sencilla y rápida. Además, el constructor de Layouts se encarga de normalizar los pesos de la estructura HitWeights si éstos no lo están.

$$w_i = \frac{\sum_{i=1}^{nProcs} t_i}{t_i} \tag{4.1}$$

La fórmula 4.3 propone un ajuste más gradual de los pesos para intentar evitar las oscilaciones y cambios bruscos. Se calcula el promedio de las medias de todos los procesos (fórmula 4.2), y después se crean los pesos como una corrección de la mitad de la diferencia entre el promedio total y el de cada proceso.

$$\bar{t} = \frac{\sum_{i=1}^{nProcs} t_i}{nProcs} \tag{4.2}$$

$$w_i = \frac{\sum_{i=1}^{nProcs} t_i}{\frac{t_i - \bar{t}}{2} + \bar{t}} \tag{4.3}$$

Las dos maneras anteriores se aplican tras la transmisión de medias entre procesos mediante el *All-to-All*. El tiempo con el que se calculan esas medias es el de cómputo (o el de iteración, según la versión de WeightedRedistribute) de todas las filas del proceso. La siguiente variante propone comunicar en el *All-to-All* el tiempo medio por fila (fórmula 4.4), en lugar de el tiempo medio total. Así, se verá mejor la potencia de cada proceso, ya que utilizando el tiempo total los pesos se asignan en función del tamaño del bloque, pues más filas normalmente indica más tiempo. Tras la comunicación colectiva, se calculan los pesos como la proporción inversa de los $r_i$ utilizando la fórmula 4.1.

$$r_i = \frac{t_i}{\text{filas del proceso } i} \tag{4.4}$$

### 4.3.8. Intervalo de redistribución creciente

A medida que se hacen redistribuciones, se espera que los pesos asignados a los procesos sean más regulares, hasta un punto que sus valores apenas varíen. Este concepto de estabilidad, introducido en el diseño inicial, implica que los mayores cambios en los dominios de los procesos se realizan en las primeras redistribuciones, luego a medida que se alcanza la estabilidad es menos necesario seguir redistribuyendo. Esta modificación propone aumentar el intervalo en el que éstas se realizan, en lugar de mantenerlo constante. Inicialmente, el intervalo tiene el mismo tamaño que la media, y cada vez que se hace una redistribución, se le incrementa en ese tamaño (`windowSize`). El tamaño de la media no varía, por lo que cuando el intervalo es más grande los datos se van sobrescribiendo y el promedio se calcula con los `windowSize` últimos. Utilizando este tipo de intervalo, cuanto más estables son los pesos, más iteraciones se tarda en hacer la redistribución siguiente, lo que supone reducir el tiempo de ejecución en comparación con utilizar un intervalo siempre del mismo tamaño.

## 4.4. Solución con Hitmap v1.3

El código 4.10 presenta la cabecera de la versión final de WeightedRedistribute para Hitmap v1.3. El algoritmo de la solución se muestra en el algoritmo A.2 (anexo A.2), y el código completo puede consultarse en el software complementario. Esta versión es la que ha resultado mejor en las pruebas de rendimiento. Tiene las siguientes características: los pesos se calculan en base al tiempo de cómputo por fila con la proporción inversa propuesta en la sección 4.3.7, y se utilizan comunicaciones no bloqueantes y un intervalo de redistribución creciente.

```
1 void weightedRedistribute(HitTile * ttile[], int numTiles,
2     HitLayout * lay, HitAvg * avg, HitPattern * patt[],
3     int numPatterns, HitPatternFunction pattf,
4     HitTileInitFunction tilef, HitShpView shpview, double time);
```

Código 4.10: Cabecera final de WeightedRedistribute (Hitmap v1.3).

## 4.5. Conclusiones del capítulo

En este capítulo se han definido los pasos que forman parte de la implementación de WeightedRedistribute a partir del diseño inicial. En primer lugar, el desarrollo de una clase aparte para gestionar la media móvil, HitAvg, y su inclusión en la función como un parámetro. En segundo lugar, la aplicación de WeightedRedistribute a un problema real, la difusión de calor con el método iterativo de Jacobi en un espacio bidimensional. Se han propuesto dos enfoques para mejorar el rendimiento de la parte secuencial evitando una copia en profundidad, que se probarán en la etapa de experimentación para determinar el más eficiente.

Se han detallado las desviaciones que ha sufrido la función desde la fase de diseño. Varias de ellas han sido descubiertas gracias a la aplicación al programa de Jacobi: la inclusión de patrones de comunicación como parámetro para su actualización, la inicialización y expansión opcional de Tiles y el problema de la redistribución con celdas compartidas. Otras modificaciones introducen posibles mejoras de rendimiento, que también se probarán en la parte experimental. Es el caso de la medición de tiempos solo a la parte de cómputo en vez de al cómputo y a la comunicación, la comunicación *All-to-All* no bloqueante, las variaciones para calcular los pesos y el intervalo de distribución creciente. Finalmente, se han presentado la cabecera y algoritmo finales para la versión 1.3 de Hitmap.

# Capítulo 5

# Experimentación

En este capítulo se introducen los siguientes aspectos:

- Los objetivos de la experimentación.
- Las características de la plataforma en la que se ha realizado la experimentación.
- La metodología seguida para realizar la experimentación.
  - Las pruebas a la clase HitAvg.
  - Las pruebas de rendimiento de WeightedRedistribute integrada en Jacobi 2D.
- Los resultados de la experimentación.

## 5.1. Metodología

### 5.1.1. Objetivos de la experimentación

Los objetivos propuestos para esta experimentación consisten en estudiar las siguientes consideraciones:

- Si el uso de WeightedRedistribute mejora el rendimiento de un programa frente a la ejecución sin ninguna redistribución.

- Si el uso de WeightedRedistribute mejora el rendimiento de un programa frente a hacer una redistribución con un reparto equitativo de la carga entre los procesos.

- Si el uso de WeightedRedistribute no tiene un gran sobrecoste en el tiempo de las iteraciones en las que se aplica.

- Si la cantidad de la carga asignada a cada proceso se estabiliza a lo largo de las redistribuciones.

- Si los procesos de la máquina más potente reciben más carga que los de la menos potente.

### 5.1.2. Plataforma de ejecución

La experimentación se ha realizado en un *cluster* perteneciente al Grupo Trasgo. Las pruebas de rendimiento de WeightedRedistribute se ejecutan en los nodos Manticore y Heracles, y las pruebas de HitAvg solo en Manticore. Todos los programas se compilan con GCC v7.2.0 en un sistema operativo CentOS v.7. La máquina Manticore tiene 2 CPUs Intel® Xeon® Scalable Platinum 8160 @ 2,1 GHz de 24 cores cada una y 48 hilos físicos (96 hilos en total), con tecnología Hyper-Threading y 256 GB de memoria RAM DDR4. La máquina Heracles tiene 8 CPUs AMD Opteron™ 6376 @ 2,2 Ghz de 8 cores cada una (36 hilos en total) y 256 GB de memoria RAM DDR3. La red de interconexión entre los nodos es Ethernet de 1 Gb.

### 5.1.3. Pruebas realizadas

A lo largo del desarrollo del proyecto se han realizado varias versiones de WeightedRedistribute. Algunas añadían elementos necesarios para que todo funcionara correctamente, mientras que otras solamente intentaban mejorar el rendimiento final. De entre estas, en ocasiones hay variaciones para la misma mejora, como es el caso de la sustitución de la copia en profundidad de Jacobi o las comunicaciones bloqueantes o no bloqueantes; y es necesario probarlas para determinar qué manera es más eficiente.

Las pruebas ejecutadas en este trabajo se dividen en dos grupos: las realizadas a funcionalidades aisladas y las realizadas a toda la función. En el primer grupo se encuentran las pruebas a la clase HitAvg, y en el segundo las pruebas de rendimiento junto con las comparaciones de las variaciones de mejora tras la integración de WeightedRedistribute en el programa de Jacobi. El procedimiento seguido en cada grupo se explica con más detalle en los apartados siguientes.

#### 5.1.3.1. Pruebas de tipos de medias con HitAvg

Tras el estudio teórico de medias y la implementación de la clase HitAvg, se realiza un ejemplo con una serie de datos para comprobar el correcto funcionamiento de la misma y reflejar las diferencias entre los tipos de medias. Se incluyen variaciones uniformes, series ascendentes y series descendentes para apreciar el comportamiento de cada media en situaciones lineales y cambios bruscos. Se utilizan tres estructuras, una con media simple, una exponencial y una linealmente ponderada, todas con un tamaño de ventana de diez elementos. Para cada dato insertado se obtienen los tres promedios y se elaboran una tabla y una gráfica con los resultados.

#### 5.1.3.2. Pruebas de rendimiento

Estas pruebas comparan el tiempo total obtenido con el programa de Jacobi original frente a las diferentes versiones de WeightedRedistribute. Se han dividido en 2 fases: la primera compara los enfoques de sustitución de la copia en profundidad en Jacobi sin aplicar redistribución; y la segunda engloba las pruebas a las versiones de WeightedRedistribute.

Las pruebas se ejecutan en máquinas con distintas capacidades para que sea más señalada la diferencia entre procesos lentos y rápidos. La primera mitad de los procesos se asigna a la

máquina Heracles y la segunda a la máquina Manticore. Cada versión del código se ejecuta con combinaciones diferentes de número de procesos, iteraciones, dimensiones de la matriz y elementos en la media, varias ejecuciones con cada combinación. Se miden los tiempos de ejecución total, de WeightedRedistribute, de cómputo de las celdas, de comunicaciones entre vecinos y de cada iteración completa (cómputo y comunicacación); y los pesos asignados a cada proceso en cada iteración. Con los resultados se generan gráficos que muestran la evolución de los tiempos a lo largo de las iteraciones. Para comparar dos versiones entre sí se utiliza un intervalo de confianza del 95 %. El método se muestra en las fórmulas 5.1 y 5.2: se calcula el conjunto $D$ como diferencia de los tiempos de ejecución obtenidos en cada versión (conjuntos $X$ e $Y$) y se computa el intervalo de confianza de límites $c_1$ y $c_2$ para la media de ese conjunto de diferencias. Como el número de muestras es menor que 30, se utilizan los valores críticos de la distribución t de Student.

$$\begin{aligned} X = (x_1, x_2, \ \dots \ , x_n) \ , \ Y = (y_1, y_2, \ \dots \ , y_n) \\ D = (d_1, d_2, \ \dots \ , d_n) \quad / \quad d_i = x_i - y_i \end{aligned} \tag{5.1}$$

$$IC = [c_1, c_2] \quad / \quad c_1 = \overline{d} - z_{1-\alpha/2}\frac{s_{\overline{d}}}{\sqrt{n}} \ , \ c_2 = \overline{d} + z_{1-\alpha/2}\frac{s_{\overline{d}}}{\sqrt{n}} \tag{5.2}$$

**Fase 1**

La fase 1 corresponde a la comparación entre el enfoque de intercambio de punteros y las iteraciones par-impar en el programa de Jacobi original, antes de incluir a WeightedRedistribute. También se utilizaba para las primeras pruebas realizadas a la función. Esta fase se divide a su vez en dos subfases:

- La **Fase 1.1** utiliza los tamaños, iteraciones, etc. de otras pruebas realizadas por el Grupo Trasgo al programa de Jacobi.
- La **Fase 1.2** aumenta el tamaño de la matriz, en previsión de que los valores de la fase 1.1 eran muy pequeños para que la redistribución produjese un efecto notable en la duración del programa. Para contrarrestar el aumento de tiempo producido por los nuevos tamaños, se decide bajar el número de iteraciones y aumentar los procesos.

Para referirse a las versiones que se utilizan en cada versión se emplean unas abreviaturas que representan sus características. Las versiones de la fase 1 son las siguientes:

- **Orig.** El programa original sin ninguna redistribución con copia en profundidad.
- **Orig-Punteros.** El programa original sin redistribución con intercambio de punteros.
- **Orig-ParImpar.** El programa original sin redistribución con iteraciones par-impar.

Los valores de los parámetros de entrada de las pruebas en cada subfase se pueden ver en la tabla 5.1. Para cada versión se ejecuta el programa tantas veces como combinaciones de procesos, iteraciones y tamaños hay en cada fase. Por ejemplo, para la fase 1.1 se realizan 80 casos diferentes.

| Fase de pruebas | Procesos | Iteraciones | Tamaño de la matriz (N × N) | Tamaño de la media |
|---|---|---|---|---|
| Fase 1.1 | 15, 20, 30, 40 | 500 | 1000, 2500, 5000, 10000, 15000 | 10, 25, 40, 50 |
| Fase 1.2 | 20, 35, 40, 50 | 200 | 20000, 25000, 30000, 37500 | 10, 25, 40, 50 |

Tabla 5.1: Valores de los parámetros de entrada en las pruebas de rendimiento (Fase 1).

**Fase 2**

La fase 2 prueba las versiones de WeightedRedistribute desarrolladas con las modificaciones descritas en la sección 4.3, tales como la medición de tiempos solo a la parte de cómputo o las comunicaciones no bloqueantes. Todas utilizan el intercambio de punteros en el bucle principal, pues en la fase anterior se comprobó que era mejor enfoque. Se incrementa mucho el número de iteraciones con el fin de estudiar la estabilidad en los tiempos y en el reparto de pesos. El número de procesos se modifica a una potencia de dos para que encajen en los nodos NUMA de las máquinas de forma natural. Se asignan 10000 filas a cada proceso, y un número menor de columnas para que la matriz resultante no resulte tan grande. Esta fase también se divide en dos partes:

- La **Fase 2.1** compara todas las variaciones excepto el el intervalo de redistibución creciente.
- La **Fase 2.2** añade el intervalo de redistribución creciente a la versión que resulte mejor de la fase 2.1. Aumenta el número de iteraciones para analizar el comportamiento de un intervalo creciente frente a un intervalo fijo en una experimentación larga.

Los valores de los parámetros de entrada se muestran en la tabla (tabla 5.2), y las versiones de WeightedRedistribute que se prueban en esta fase se enumeran a continuación. El formato de las abreviaturas es "$a-b-c-d$", siendo ($a$) el tipo de intervalo de redistribución (fijo o incremental), ($b$) el tiempo con el que se calculan los pesos (iteración o cómputo), ($c$) la forma de cálculo de los pesos (con o sin normalización) y ($d$), el tipo de *All-to-All* (bloqueante o no bloqueante).

- **Orig-Punteros.**
- **FixI-Iter-OldW-Bloq.** Los pesos se calculan y normalizan de la manera indicada en el diseño inicial (fórmulas 3.1 y 3.2).
- **FixI-Iter-NewW-Bloq.** Los pesos se calculan como el cociente entre la suma de todas medias y la media local de un proceso (fórmula 4.1).
- **FixI-Com-Pre-Bloq.** Los pesos se establecen a mano, repartiendo equitativamente el peso total (1) entre los procesos activos.
- **FixI-Com-OldW-Bloq.**
- **FixI-Com-OldW-NBloq.**
- **FixI-Com-AvgW-NBloq.** Los pesos se calculan como una corrección entre el promedio de medias de todos los procesos y la media local (fórmula 4.3).
- **FixI-ComRow-NewW-NBloq.** Se utiliza el tiempo de cómputo por fila en lugar del tiempo de cómputo total (fórmula 4.4).

- **IncrI-ComRow-NewW-NBloq.** Mismas características que la versión anterior pero con intervalo creciente en lugar de fijo.

| Fase de pruebas | Procesos | Iteraciones | Tamaño de la matriz (N × N) | Tamaño de la media |
|---|---|---|---|---|
| Fase 2.1 | 16 | 2000 | 160000 × 2500 | 100 |
| Fase 2.2 | 16 | 4000 | 160000 × 2500 | 100 |

Tabla 5.2: Valores de los parámetros de entrada en las pruebas de rendimiento (Fase 2).

## 5.2. Pruebas de tipos de medias con HitAvg

Los datos y resultados numéricos de la experimentación de tipos de medias (media móvil simple, linealmente ponderada y exponencial) con HitAvg pueden verse en la tabla 5.3 y los gráficos en la figura 5.1. Las series ascendentes y descendentes aparecen delimitadas por líneas horizontales en la tabla. En la gráfica, el eje de abscisas marca los días del experimento y el de ordenadas indica el nuevo dato y el valor calculado de cada media en ese día, en segundos.

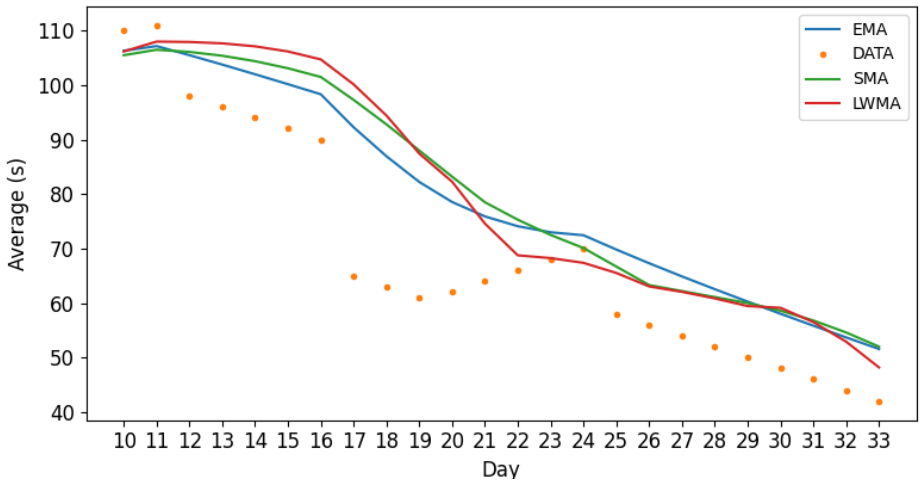



Figura 5.1: Resultados gráficos de la comparación de medias utilizando HitAvg.

En la gráfica (figura 5.1) se observa que la media exponencial tiene en general una curva más suave que las otras medias y es la que más fielmente sigue la tendencia real. La gráfica cambia el mismo día que la tendencia de los datos, al contrario que las otras medias que lo hacen uno o dos días después. En la mayor parte del período, la línea de la EMA es paralela a la formada por los valores de los datos. En los días 20 al 24, el número de datos crecientes es insuficiente para contrarrestar la evolución descendente anterior, pero hace que la pendiente se ralentice.

La media ponderada tiene los valores más cercanos al valor real en los períodos en los que se ha seguido una tendencia clara durante un tiempo, como los primeros días y los últimos. Esta media

| Día | Dato (s) | EMA (s) | SMA (s) | LWMA (s) |
|---|---|---|---|---|
| 1 | 101 | - | - | - |
| 2 | 102 | - | - | - |
| 3 | 103 | - | - | - |
| 4 | 104 | - | - | - |
| 5 | 105 | - | - | - |
| 6 | 106 | - | - | - |
| 7 | 107 | - | - | - |
| 8 | 108 | - | - | - |
| 9 | 109 | - | - | - |
| 10 | 110 | 106,3182 | 105,5000 | 106,1818 |
| 11 | 111 | 107,1694 | 106,5000 | 108,0000 |
| 12 | 98 | 105,5023 | 106,1000 | 107,9273 |
| 13 | 96 | 103,7746 | 105,4000 | 107,6727 |
| 14 | 94 | 101,9974 | 104,4000 | 107,1273 |
| 15 | 92 | 100,1797 | 103,1000 | 106,1818 |
| 16 | 90 | 98,3288 | 101,5000 | 104,7273 |
| 17 | 65 | 92,2690 | 97,3000 | 100,1455 |
| 18 | 63 | 86,9474 | 92,8000 | 94,4182 |
| 19 | 61 | 82,2297 | 88,0000 | 87,4364 |
| 20 | 62 | 78,5516 | 83,2000 | 82,2727 |
| 21 | 64 | 75,9058 | 78,5000 | 74,5818 |
| 22 | 66 | 74,1048 | 75,3000 | 68,7636 |
| 23 | 68 | 72,9948 | 72,5000 | 68,2545 |
| 24 | 70 | 72,4503 | 70,1000 | 67,3818 |
| 25 | 58 | 69,8230 | 66,7000 | 65,5273 |
| 26 | 56 | 67,3097 | 63,3000 | 63,0545 |
| 27 | 54 | 64,8898 | 62,2000 | 62,0545 |
| 28 | 52 | 62,5462 | 61,1000 | 60,8545 |
| 29 | 50 | 60,2650 | 60,0000 | 59,4545 |
| 30 | 48 | 58,0350 | 58,6000 | 59,1273 |
| 31 | 46 | 55,8468 | 56,8000 | 56,5091 |
| 32 | 44 | 53,6929 | 54,6000 | 52,9091 |
| 33 | 42 | 51,5669 | 52,0000 | 48,1818 |

Tabla 5.3: Comparación de medias utilizando la estructura HitAvg.

es muy susceptible a los cambios drásticos en los valores, como se observa en el fuerte descenso del día 16. En los días 29 y 30, la LWMA se mantiene casi constante pese a que los datos siguen descendiendo. Esto se debe a que el primer dato de la ventana cambia del día 19 al 20, cuando empieza la tendencia ascendente de nuevo. Una vez que avanzan los días, los valores de la media vuelven a bajar.

En último lugar, la media simple es la que peor refleja la tendencia de los datos. Su gráfica es muy lineal comparada con la LWMA y la EMA, que son algo más curvas. No es capaz de reflejar el aumento en los valores de los datos que se produce a partir del día 20, lo que pone de manifiesto el problema de los falsos positivos o negativos: si se utilizase la media simple para predecir la tendencia de los datos, los resultados indicarían que ésta es siempre descendente, cuando la realidad es que hay unos días que los valores han crecido.

### 5.2.1. Conclusiones

El cálculo de medias mediante una estructura HitAvg produce los mismos resultados que una forma convencional de cálculo, y es útil para consultar los datos que participan en el promedio y el peso asociado a cada uno. Los tipos de medias estudiados (simple, linealmente ponderada y exponencial) predicen la tendencia de los datos de maneras diferentes. La EMA y la LWMA dan más importancia a los datos recientes, por lo que reaccionan antes a los cambios que la SMA. La EMA es la que ha ofrecido una previsión más acertada, y la LWMA tiene los valores más cercanos a los datos reales en períodos de estabilidad. Se espera que los tiempos de Jacobi2D no sean tan variables como en este ejemplo, luego no está claro el tipo de media más acertado para esa aplicación y habrá que utilizar los tres en las pruebas con WeightedRedistribute para decidirlo.

## 5.3. Pruebas de rendimiento

### 5.3.1. Consideraciones sobre la notación de tablas y gráficas

Para comparar los tiempos totales de ejecución entre dos versiones de WeightedRedistribute se utiliza un intervalo de confianza y se muestran los resultados en una tabla. Tomando como ejemplo la tabla 5.4, cada celda corresponde a la diferencia en las ejecuciones con una combinación de número de procesos, elementos en la media y tamaño de la matriz. El fondo de la celda se colorea según los valores del intervalo. Las celdas blancas son en las que el valor 0 está incluido en el intervalo, e indican que se puede afirmar con un 95 % de confianza que las diferencias calculadas se deben a errores aleatorios en la medición y no a que una versión sea mejor que otra. En las celdas verdes ambos extremos tienen valores negativos, e indican que se puede decir con un 95 % de confianza que la versión que se ha puesto como primera en las diferencias (la que se escribe primero en la descripción de la tabla, en este caso Orig-Punteros) tarda menos en ejecutar el programa que la otra. En las celdas rojas el intervalo tiene valores positivos, y significa que la segunda versión (Orig-ParImpar) es mejor que la primera con un 95 % de confianza.

Las gráficas de tiempos de redistribución, cómputo, comunicación de vecinos e iteración tienen el mismo formato. Cada línea representa los tiempos de un proceso a lo largo del programa. El eje

de abscisas marca las iteraciones y el eje de ordenadas el tiempo en segundos. Este eje se muestra en escala logarítmica para ampliar el rango de valores entre 0 y 1, que es donde se sitúan la mayoría de tiempos. Las gráficas de pesos muestran la evolución de la asignación de peso a cada proceso a lo largo del programa. El eje de abscisas indica el índice de la iteración y el eje de ordenadas el valor del peso normalizado en el rango [0,1]. Las líneas tienen una forma cuadrada porque los pesos solo varían cuando se hace una redistribución. En la iteración 0 todos los procesos tienen el mismo peso debido a que se parte de un reparto equitativo.

### 5.3.2. Fase 1. Pruebas de Jacobi2D original

#### 5.3.2.1. Comparación de enfoques par-impar y punteros

En esta subsección se realiza la comparación del tiempo en ejecutar el programa original de Jacobi entre la versión que utiliza el enfoque par-impar y la que utiliza el intercambio de punteros; y la comparación de la mejor de las dos con el programa con copia en profundidad, para estudiar la mejora que supone la sustitución de esa función. Las tablas que se muestran son un resumen de los tamaños de matriz con resultados más relevantes para cada comparación. Las tablas completas con todos los resultados de las fases 1.1 y 1.2 se pueden consultar en el anexo B.1.

La tabla 5.4 presenta los resultados de la comparación de los enfoques par-impar y punteros, colocando como mejor al intercambio de punteros. Como se puede apreciar, con este enfoque se puede afirmar con un 95 % de confianza que los resultados son mejores en la mayoría de casos, sobre todo al ir aumentando las dimensiones de la matriz. Cuando el tamaño es pequeño (1000, 2500 ...), la ejecución normalmente es casi inmediata porque el trabajo de cada proceso es muy poco, por lo que no se pueden apreciar correctamente las variaciones entre los dos enfoques. Se observa que los intervalos tienen una amplitud pequeña y son cercanos al valor 0, lo que significa que las diferencias no son muy notables. Una excepción es el caso de tamaño de matriz 30000, 35 procesos y 10 elementos en la media, donde la amplitud del intervalo es de 14. Esto es debido a un *outlier* en las muestras del enfoque par-impar que toma un valor mucho mayor que la media.

La tabla 5.5 muestra los resultados obtenidos con la comparación entre la versión con intercambio de punteros y la versión con copia en profundidad. Se observa que las mejoras son bastante significativas, y aumentan a medida que lo hace el tamaño de la matriz. Esto confirma la hipótesis de que la copia en profundidad suponía una parte importante del coste del programa.

### 5.3.3. Fase 2. Pruebas de WeightedRedistribute

#### 5.3.3.1. Resultados individuales de las versiones desarrolladas

A continuación se comentarán las gráficas de tiempo de cómputo, comunicación de vecinos, iteración completa (cómputo y comunicación), redistribución y reparto de pesos obtenidas con cada versión de WeightedRedistribute con los parámetros de la fase 2.1: matriz de 160000 filas y 2500 columnas, 16 procesos, 2000 iteraciones y 100 elementos en la media. De cada versión se ha escogido la gráfica más representativa de todas las ejecuciones realizadas.

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-0.548, 1.699] | [-1.576, 0.765] | [-2.302, -0.178] | [-3.191, -0.431] |
| 20 | [-0.005, -0.003] | [-0.003, 0.004] | [0.000, 0.007] | [-0.001, 0.007] |
| 30 | [-3.352, 0.032] | [-1.659, 1.802] | [-0.512, 1.411] | [-0.368, 0.593] |
| 40 | [-0.024, -0.000] | [-0.022, 0.011] | [-0.003, 0.024] | [0.004, 0.017] |

(a) Tamaño matriz: 1000 × 1000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-0.032, -0.003] | [-0.043, -0.020] | [-0.027, -0.016] | [-0.028, -0.016] |
| 20 | [-0.014, 0.000] | [-0.025, -0.004] | [-0.008, 0.008] | [-0.025, -0.007] |
| 30 | [-0.652, 0.270] | [-0.244, 0.532] | [-0.177, 0.631] | [-0.303, 0.664] |
| 40 | [-0.022, -0.000] | [-0.017, -0.004] | [-0.002, 0.005] | [-0.018, -0.009] |

(b) Tamaño matriz: 5000 × 5000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-0.207, -0.179] | [-0.308, -0.231] | [-0.287, -0.189] | [-0.323, -0.239] |
| 20 | [-0.223, -0.100] | [-0.190, -0.103] | [-0.232, -0.173] | [-0.217, -0.145] |
| 30 | [-0.157, -0.119] | [-0.124, -0.079] | [-0.117, -0.082] | [-0.681, 0.028] |
| 40 | [-0.090, -0.036] | [-0.120, -0.060] | [-0.111, -0.054] | [-0.072, -0.026] |

(c) Tamaño matriz: 15000 × 15000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 20 | [-0.549, -0.505] | [-0.602, -0.436] | [-0.512, -0.374] | [-0.573, -0.524] |
| 35 | [-0.266, -0.225] | [-0.273, -0.240] | [-0.341, -0.313] | [-0.298, -0.275] |
| 40 | [-0.295, -0.260] | [-0.195, -0.167] | [-0.290, -0.220] | [-0.247, -0.202] |
| 50 | [-0.257, -0.231] | [-0.204, -0.169] | [-0.229, -0.162] | [-0.180, -0.126] |

(d) Tamaño matriz: 25000 × 25000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 20 | [-0.950, -0.810] | [-0.925, -0.810] | [-0.987, -0.779] | [-0.949, -0.803] |
| 35 | [-10.139, 3.896] | [-0.519, -0.419] | [-0.470, -0.398] | [-0.462, -0.397] |
| 40 | [-0.467, -0.361] | [-0.504, -0.398] | [-0.384, -0.346] | [-0.470, -0.358] |
| 50 | [-0.294, -0.245] | [-0.274, -0.221] | [-0.305, -0.265] | [-0.325, -0.248] |

(e) Tamaño matriz: 30000 × 30000

Tabla 5.4: Comparación del tiempo de ejecución total (segundos) del programa de Jacobi2D con parámetros de la fase 1 (resumen). Versiones Orig-Punteros y Orig-ParImpar.

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-1.804, 0.436] | [-1.991, 0.678] | [-0.675, 0.300] | [-1.778, 0.934] |
| 20 | [0.009, 0.020] | [0.014, 0.017] | [0.015, 0.020] | [0.006, 0.020] |
| 35 | [-2.243, 0.100] | [-0.114, 4.185] | [0.153, 2.664] | [-1.705, 1.523] |
| 40 | [0.046, 0.086] | [0.059, 0.067] | [0.048, 0.088] | [0.055, 0.086] |

(a) Tamaño matriz: 1000 × 1000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-6.945, -6.934] | [-6.961, -6.943] | [-6.940, -6.924] | [-6.954, -6.935] |
| 20 | [-5.099, -5.050] | [-5.100, -5.080] | [-5.118, -5.052] | [-5.087, -5.064] |
| 35 | [-3.633, -2.456] | [-3.558, -2.973] | [-3.471, -2.540] | [-3.481, -2.192] |
| 40 | [-2.351, -2.325] | [-2.360, -2.329] | [-2.356, -2.330] | [-2.346, -2.336] |

(b) Tamaño matriz: 5000 × 5000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-65.406, -65.253] | [-65.462, -65.387] | [-65.481, -65.281] | [-65.514, -65.395] |
| 20 | [-49.657, -49.421] | [-49.520, -49.453] | [-49.533, -49.377] | [-49.697, -49.441] |
| 35 | [-32.310, -32.296] | [-32.301, -32.233] | [-32.303, -32.270] | [-32.364, -32.331] |
| 40 | [-24.280, -24.206] | [-24.294, -24.191] | [-24.314, -24.278] | [-24.265, -24.175] |

(c) Tamaño matriz: 15000 × 15000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 20 | [-53.958, -53.799] | [-53.872, -53.756] | [-53.759, -53.676] | [-54.084, -53.802] |
| 35 | [-30.978, -30.914] | [-30.951, -30.881] | [-31.001, -30.947] | [-41.087, -26.448] |
| 40 | [-27.212, -27.066] | [-27.049, -26.995] | [-27.211, -27.135] | [-27.201, -27.020] |
| 50 | [-21.263, -21.184] | [-21.317, -21.233] | [-21.323, -21.256] | [-21.354, -21.106] |

(d) Tamaño matriz: 25000 × 25000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 20 | [-124.457, -121.611] | [-125.532, -123.521] | [-125.686, -121.963] | [-125.971, -123.420] |
| 35 | [-69.203, -68.778] | [-68.922, -68.708] | [-68.974, -68.572] | [-69.258, -68.716] |
| 40 | [-59.817, -59.517] | [-60.457, -59.791] | [-60.231, -59.781] | [-59.679, -59.369] |
| 50 | [-47.801, -47.656] | [-47.786, -47.709] | [-47.868, -47.738] | [-47.881, -47.699] |

(e) Tamaño matriz: 37500 × 37500

Tabla 5.5: Comparación del tiempo de ejecución total (segundos) del programa de Jacobi2D con parámetros de la fase 1 (resumen). Versiones OrigPunteros y Orig.

**Orig-Punteros** (figura 5.2). Se observa que los tiempos de iteración y de cómputo tienen una tendencia creciente. Los procesos con mayores tiempos de cómputo, que corresponden a los de la máquina Heracles, tienen menor tiempo de comunicación. Al contrario, los procesos de Manticore tardan menos en computar las celdas, pero tienen un mayor tiempo de comunicación. Esta situación ya se expuso en la sección 4.3.5. Al no haber redistribución, los pesos son siempre constantes y el tiempo de la función WeightedRedistribute es nulo.

**FixI-Iter-OldW-Bloq** (figura 5.3). La tendencia de tiempos de cómputo y de iteración es similar al caso anterior, salvo los picos en las iteraciones donde se hace la redistribución. Los picos descendentes en la comunicación (figura 5.3b) coinciden con los ascendentes en el tiempo de cómputo (figura 5.3a). Se observa también que el tiempo de WeightedRedistribute es mínimo excepto cuando se realizan las redistribuciones, y que los procesos de Manticore reciben más peso que los de Heracles, pero las diferencias son muy pequeñas, de menos de una milésima. Esto es un indicio de que la fórmula de reparto de pesos no es correcta.

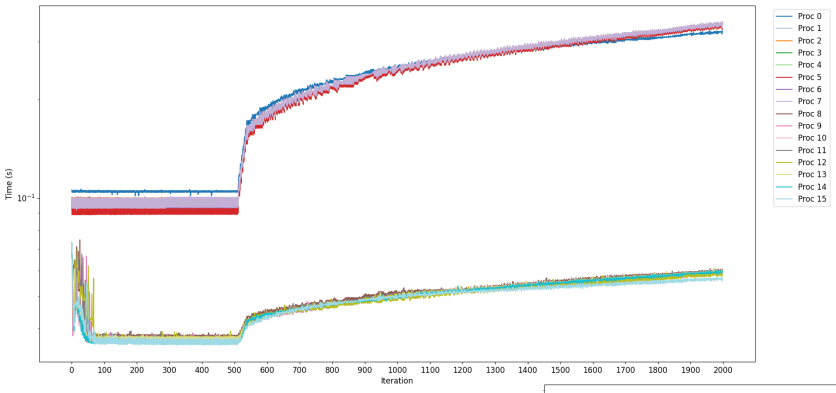
**FixI-Iter-NewW-Bloq** (figura 5.4). Esta versión solo difiere con la anterior en que los pesos se calculan de diferente manera y no se normalizan. Ahora las diferencias de peso son un poco mayores, pero siguen sin ser tan notables como se esperaba. Los tiempos de iteración son crecientes, por lo que este cálculo de pesos tampoco es correcto.

**FixI-Com-Pre-Bloq** (figura 5.5). En esta versión se comunican los tiempos de cómputo entre todos los procesos, pero los pesos se reparten equitativamente, como refleja la figura 5.5e. En los tiempos de iteración se aprecia que esta es una versón menos eficiente que las que utilizan un balanceo adaptativo.

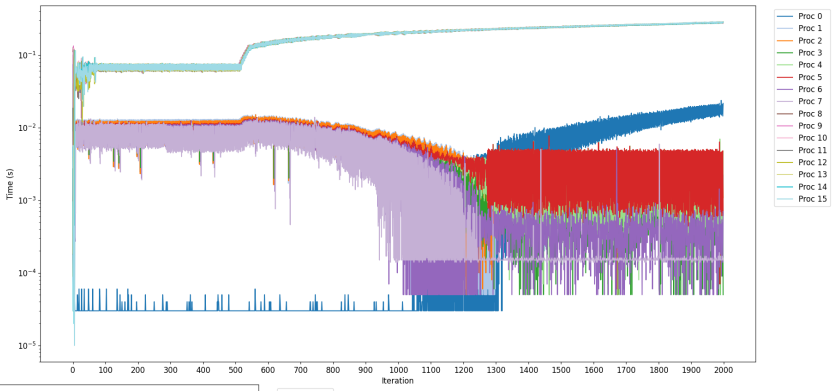
**FixI-Com-OldW-Bloq** (figura 5.6). Se puede apreciar que, aunque se utiliza la fórmula antigua de normalización de pesos, éstos tienen una diferencia mayor entre Manticore y Heracles que en el caso anterior a causa de utilizar el tiempo solo de cómputo. Además, los pesos están mucho más agrupados por máquinas. Los tiempos de iteración tienen una pendiente ligeramente menos ascendente que en las versiones anteriores, pero tampoco es el resultado esperado.

**FixI-Com-OldW-NBloq** (figura 5.7). Los tiempos de esta versión, que utiliza una comunicación *All-to-All* no bloqueante, son prácticamente iguales que en la versión anterior con *All-to-All* bloqueante. En ambas versiones, los pesos en Heracles son más inconsistentes que en Manticore, y eso se refleja en los tiempos de las gráficas de cómputo (figura 5.7a) y comunicación de vecinos (figura 5.7b). Se decide elegir como mejor el enfoque entre los dos el no bloqueante, porque si bien no mejora el tiempo de ejecución, solapar el cómputo de una iteración con la comunicación de medias es una buena práctica para que los procesos estén ociosos el menor tiempo posible.

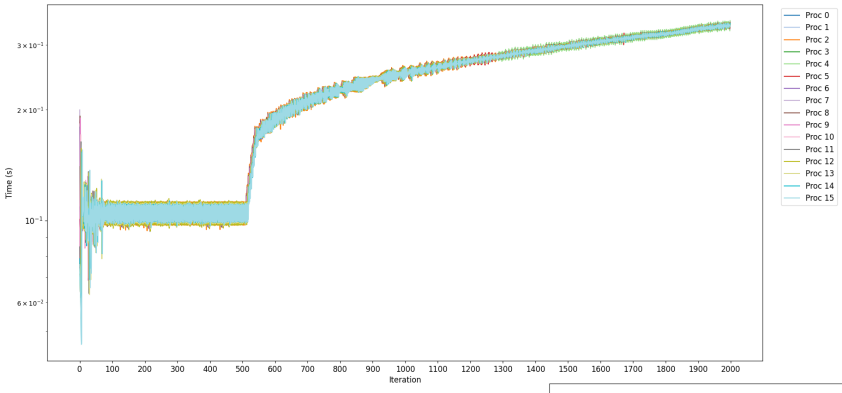
**FixI-Com-AvgW-NBloq** (figura 5.8). Esta fórmula de cálculo de pesos produce mejores resultados que la normalización y la división entre la suma de medias. La tendencia del tiempo de iteración no es tan ascendente, pero los pesos en los procesos de Heracles son muy variables. Por otra parte, el tiempo de redistribución es algo más alto que en otras versiones, lo que puede deberse a que la fórmula es algo más compleja que las anteriores.



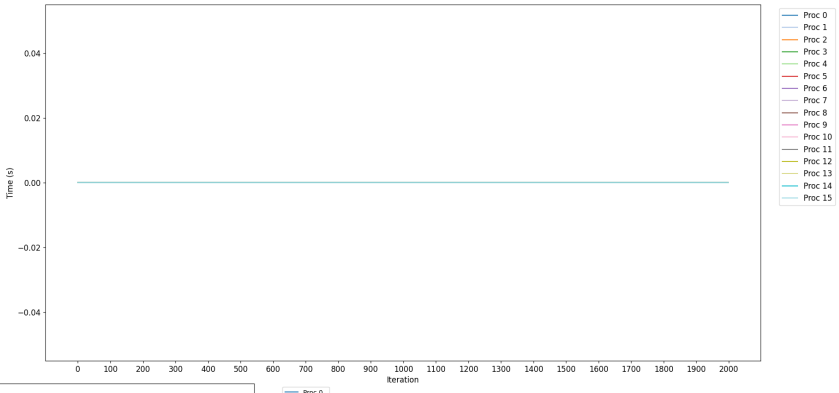

(a) Tiempo de cómputo de nuevas celdas en cada iteración (segundos)



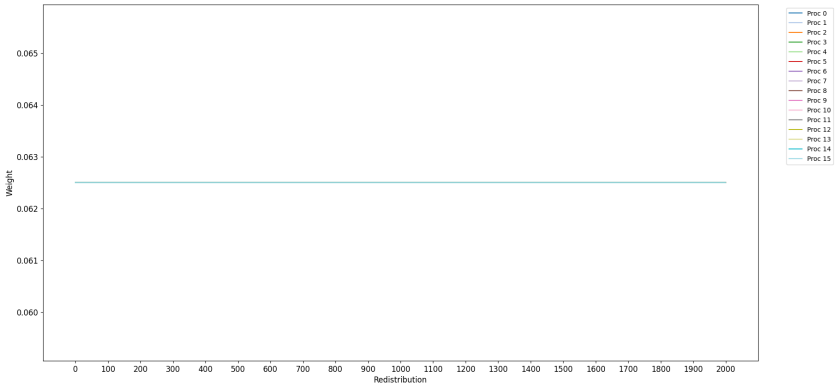

(b) Tiempo de comunicación de vecinos en cada iteración (segundos)




(c) Tiempo de ejecución de cada iteración (segundos)



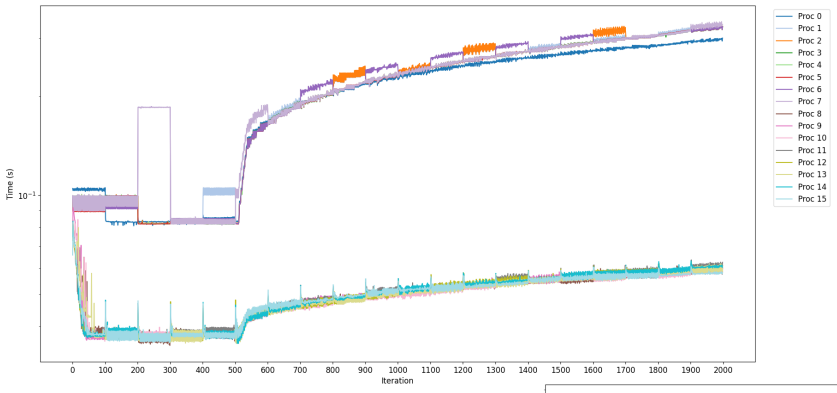

(d) Tiempo de WeightedRedistribute en cada iteración (segundos)



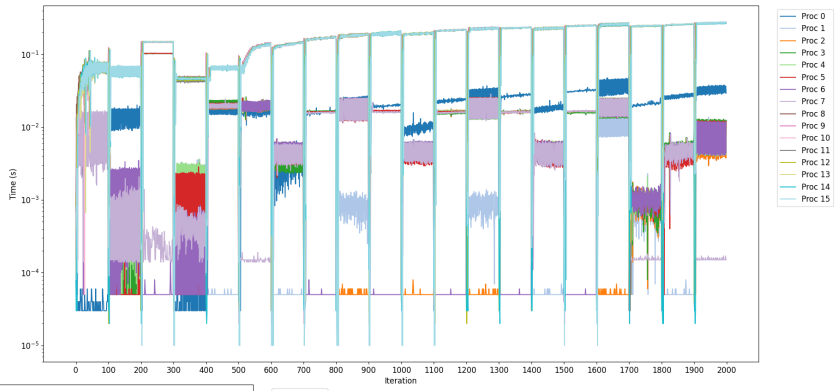

(e) Reparto de pesos en cada redistribución



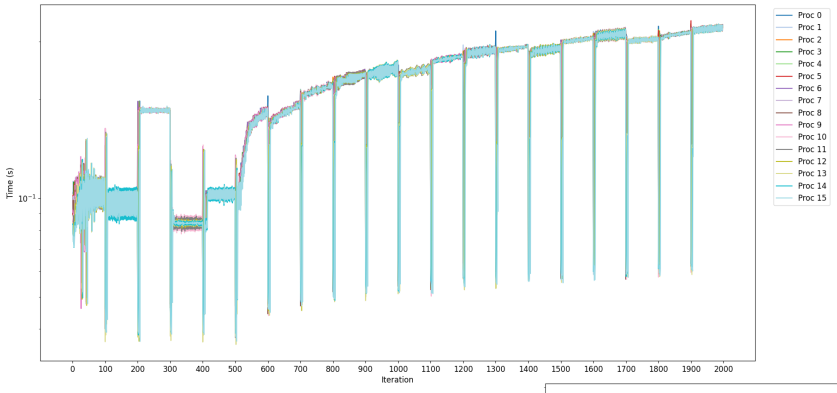
Figura 5.2: Resultados gráficos de la ejecución del programa de Jacobi2D con parámetros de la fase 2.1. Versión Orig-Punteros.



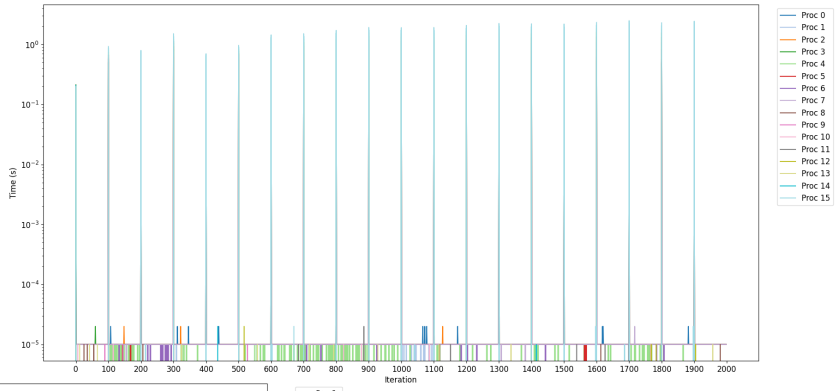

(a) Tiempo de cómputo de nuevas celdas en cada iteración (segundos)



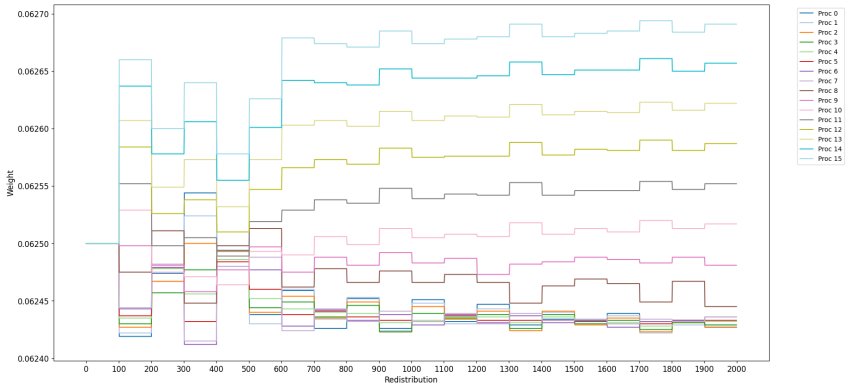

(b) Tiempo de comunicación de vecinos en cada iteración (segundos)




(c) Tiempo de ejecución de cada iteración (segundos)

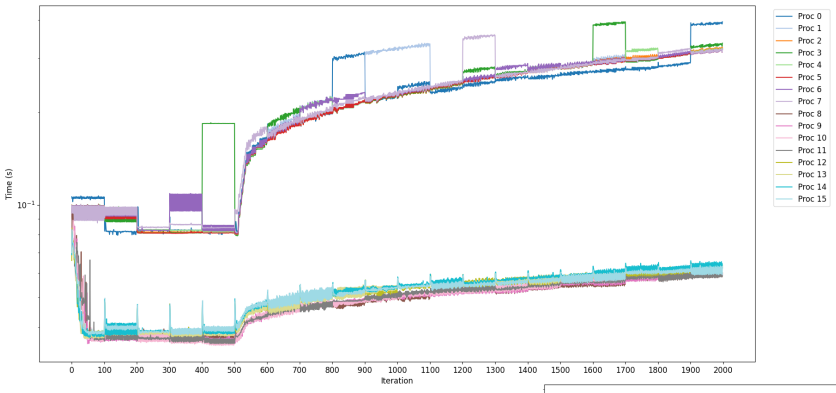



(d) Tiempo de WeightedRedistribute en cada iteración (segundos)

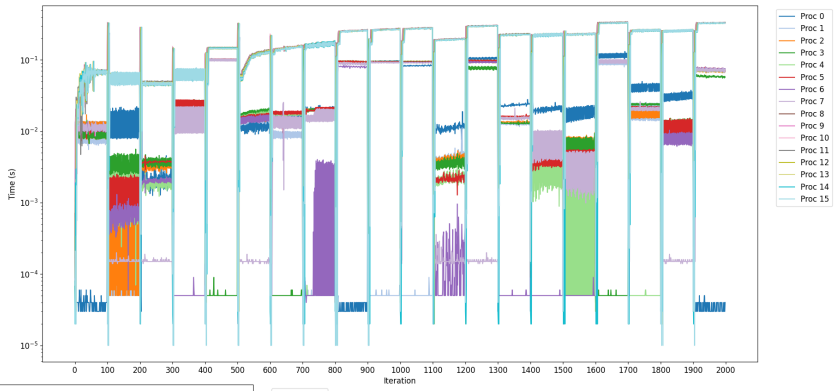



(e) Reparto de pesos en cada iteración (proporción [0,1])

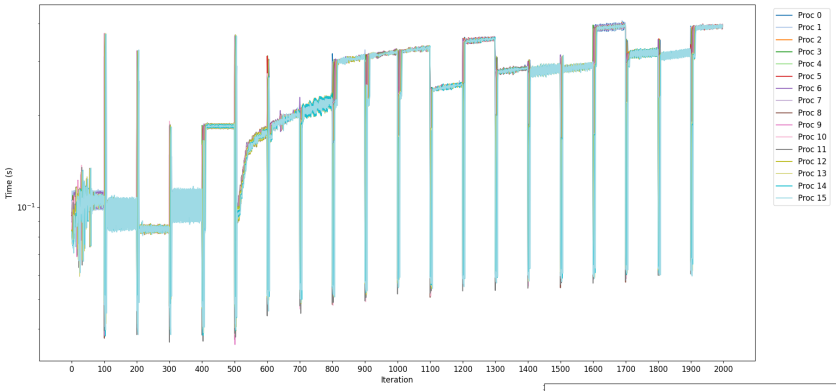
Figura 5.3: Resultados gráficos de la ejecución del programa de Jacobi2D con parámetros de la fase 2.1. Versión FixI-Iter-OldW-Bloq.



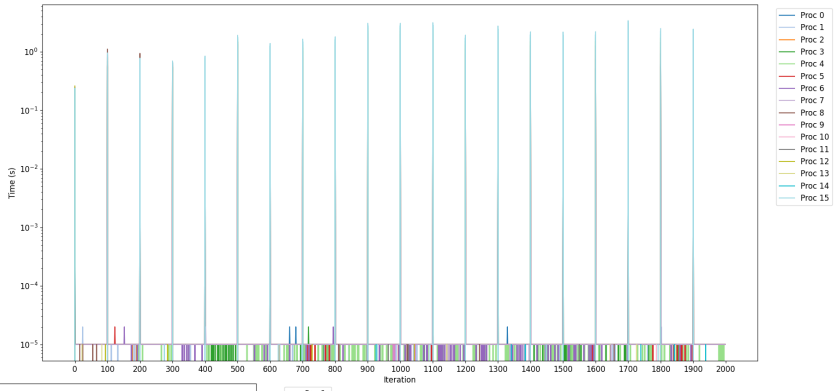

(a) Tiempo de cómputo de nuevas celdas en cada iteración (segundos)



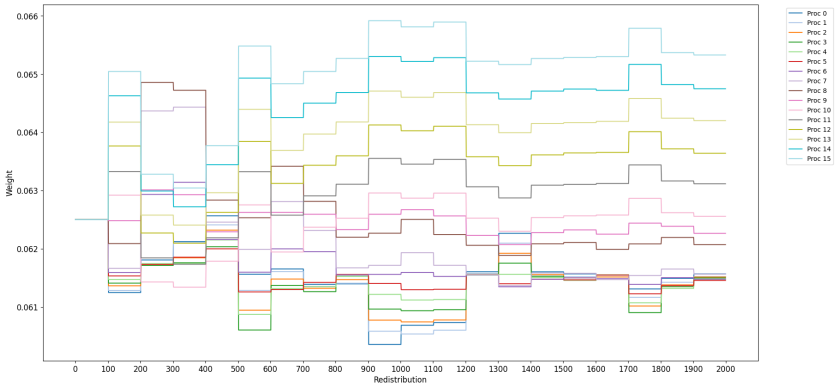

(b) Tiempo de comunicación de vecinos en cada iteración (segundos)




(c) Tiempo de ejecución de cada iteración (segundos)



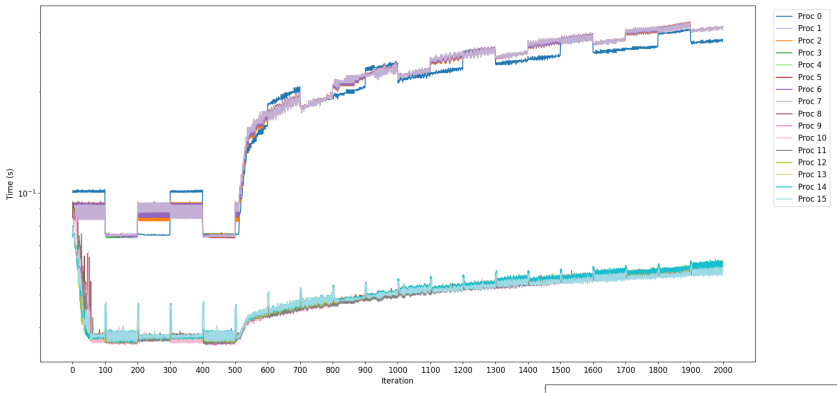

(d) Tiempo de WeightedRedistribute en cada iteración (segundos)



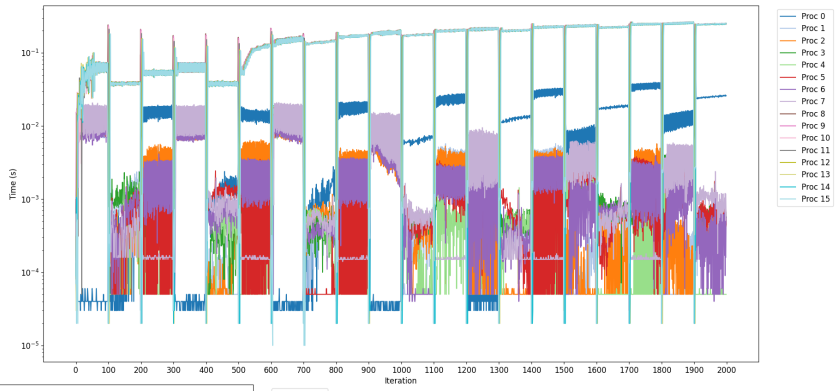

(e) Reparto de pesos en cada iteración (proporción [0,1])



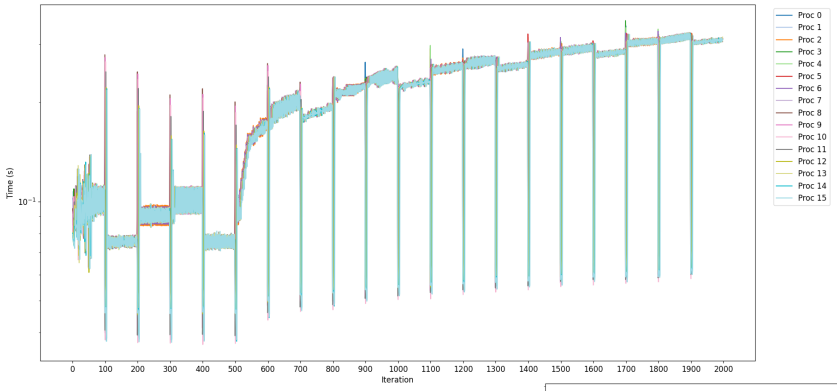
Figura 5.4: Resultados gráficos de la ejecución del programa de Jacobi2D con parámetros de la fase 2.1. Versión FixI-Iter-NewW-Bloq.



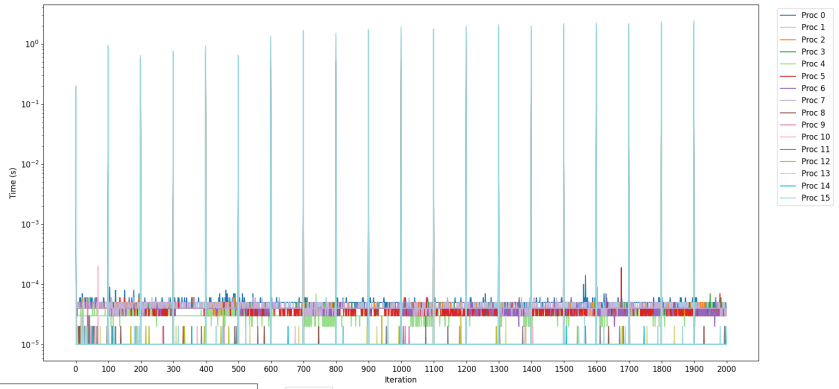

(a) Tiempo de cómputo de nuevas celdas en cada iteración (segundos)



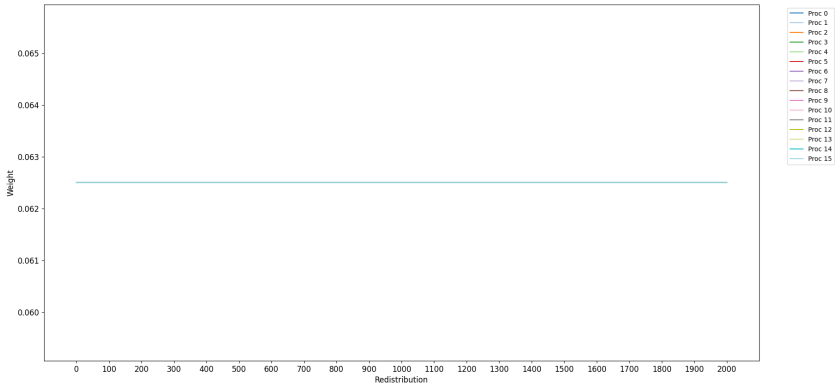

(b) Tiempo de comunicación de vecinos en cada iteración (segundos)




(c) Tiempo de ejecución de cada iteración (segundos)

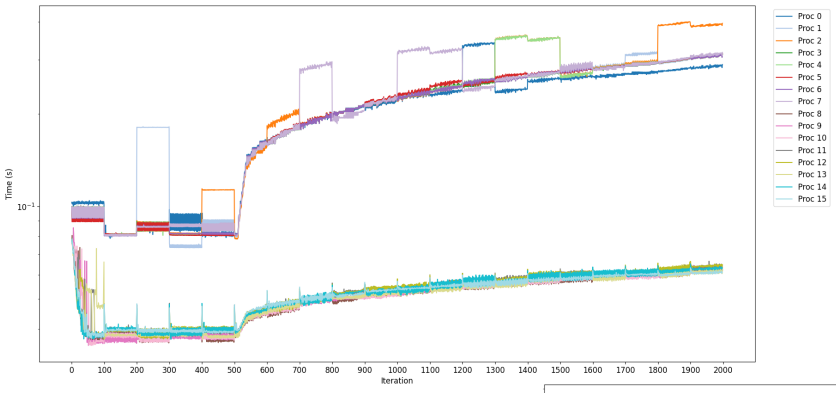



(d) Tiempo de WeightedRedistribute en cada iteración (segundos)

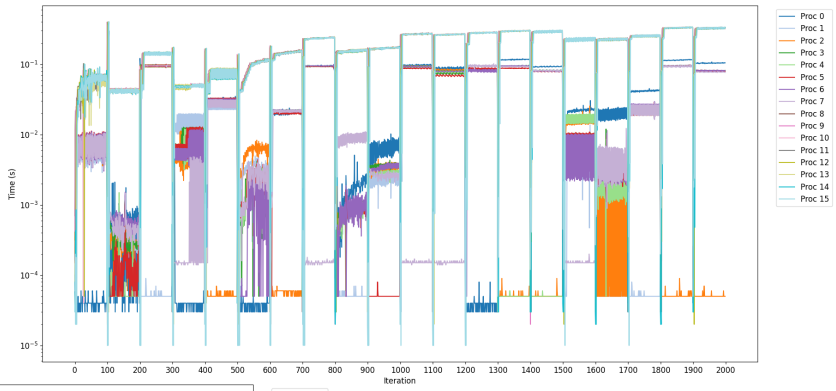



(e) Reparto de pesos en cada iteración (proporción [0,1])

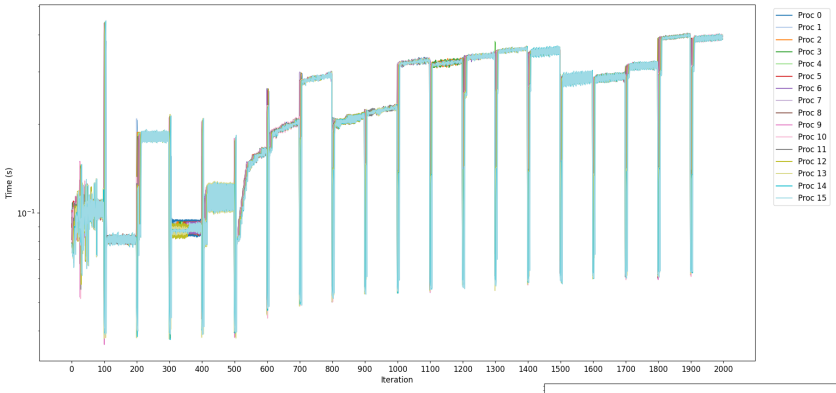
Figura 5.5: Resultados gráficos de la ejecución del programa de Jacobi2D con parámetros de la fase 2.1. Versión FixI-Com-Pre-Bloq.



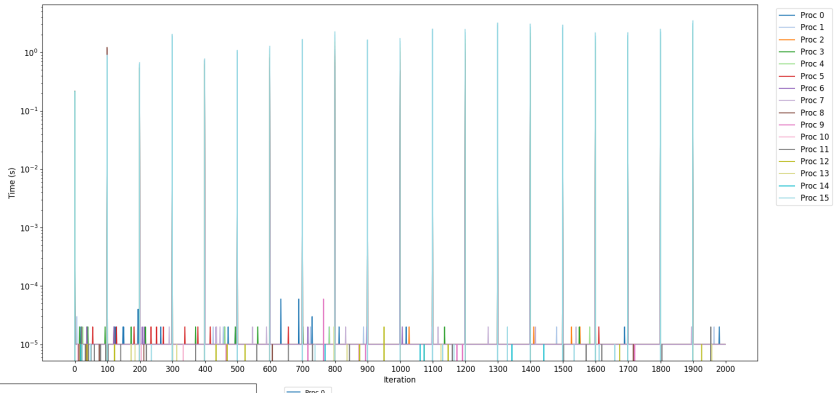

(a) Tiempo de cómputo de nuevas celdas en cada iteración (segundos)



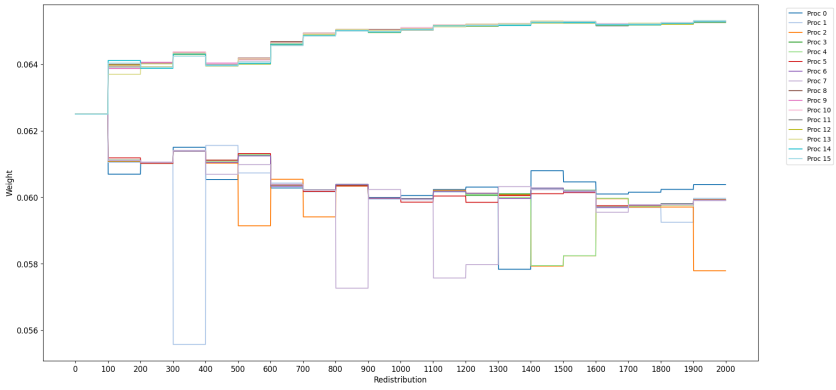

(b) Tiempo de comunicación de vecinos en cada iteración (segundos)




(c) Tiempo de ejecución de cada iteración (segundos)



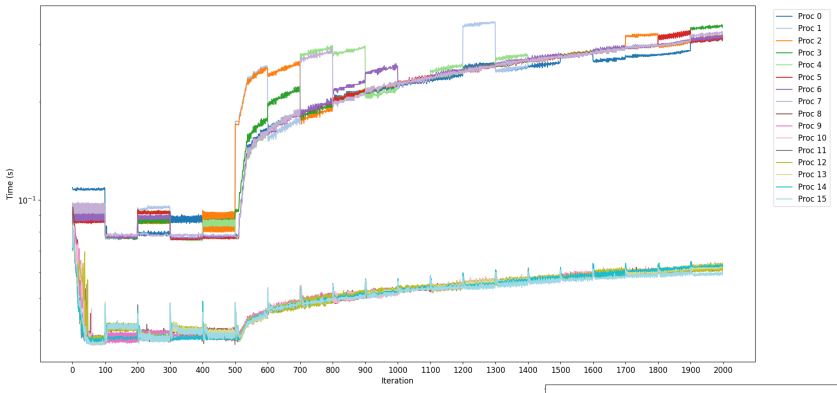

(d) Tiempo de WeightedRedistribute en cada iteración (segundos)



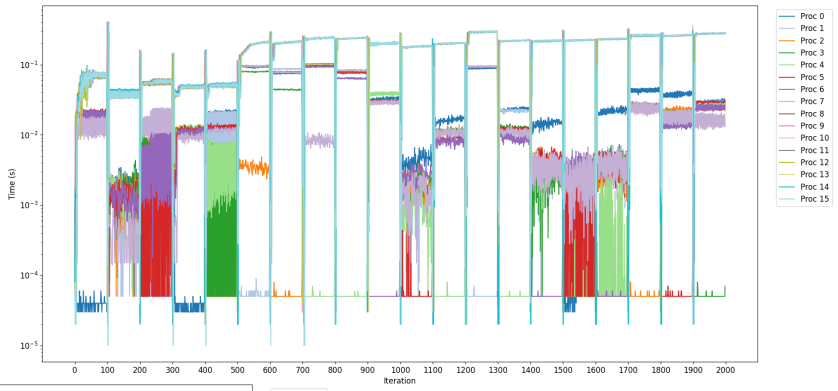

(e) Reparto de pesos en cada iteración (proporción [0,1])



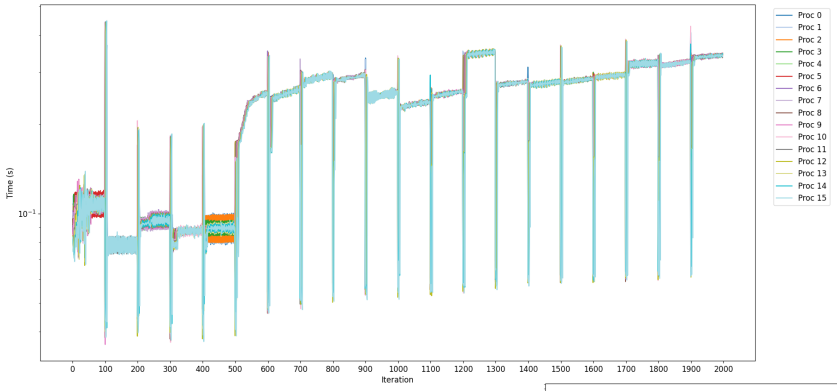
Figura 5.6: Resultados gráficos de la ejecución del programa de Jacobi2D con parámetros de la fase 2.1. Versión FixI-Com-OldW-Bloq.



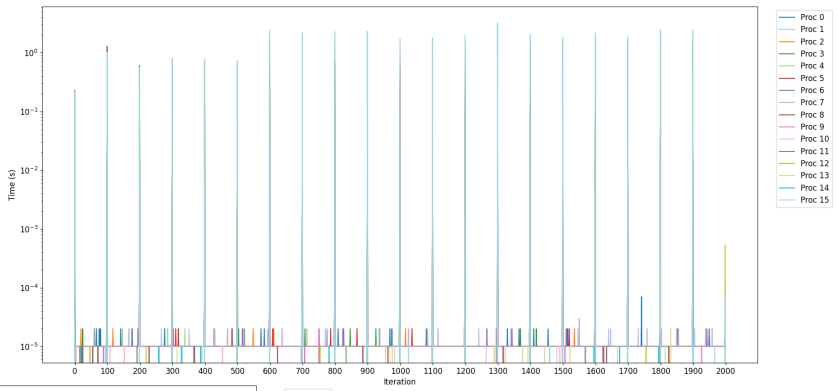

(a) Tiempo de cómputo de nuevas celdas en cada iteración (segundos)



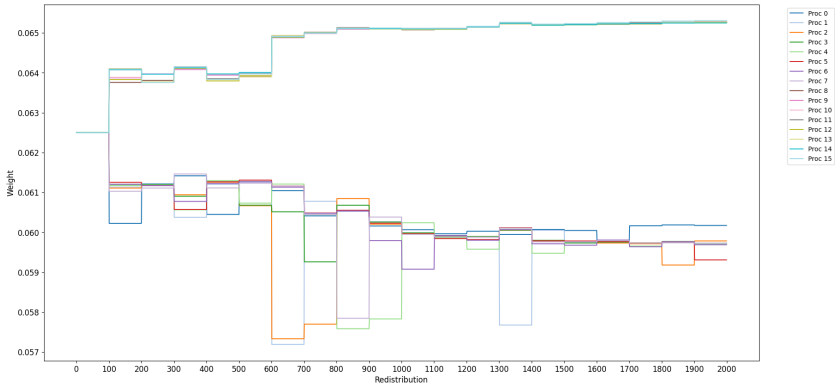

(b) Tiempo de comunicación de vecinos en cada iteración (segundos)




(c) Tiempo de ejecución de cada iteración (segundos)

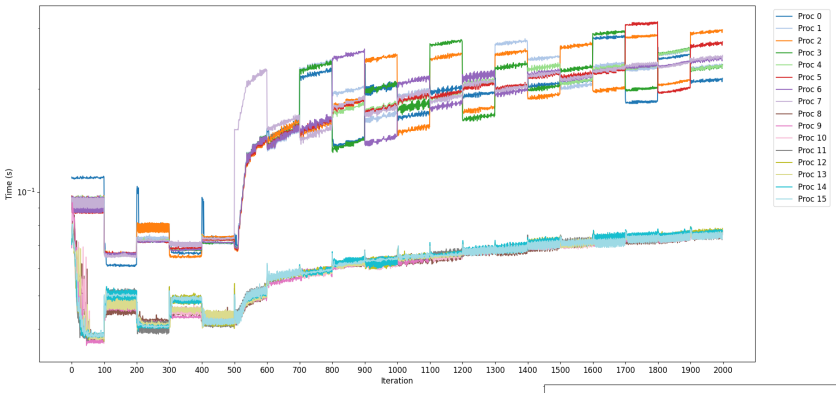



(d) Tiempo de WeightedRedistribute en cada iteración (segundos)

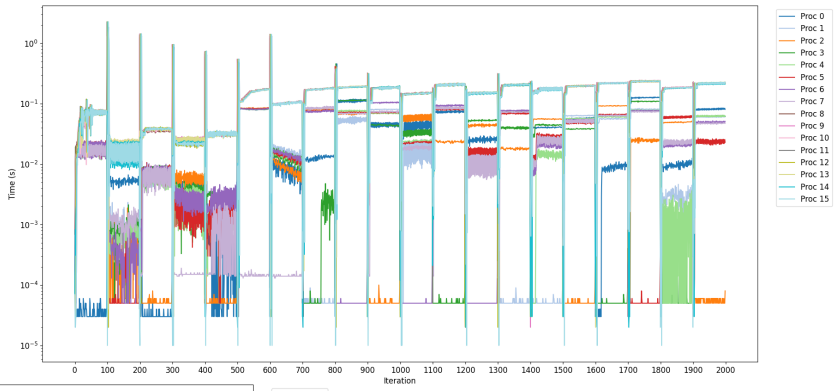



(e) Reparto de pesos en cada iteración (proporción [0,1])

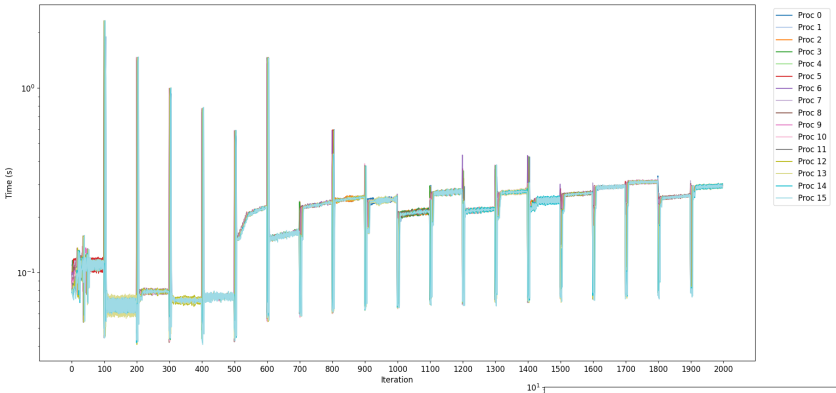
Figura 5.7: Resultados gráficos de la ejecución del programa de Jacobi2D con parámetros de la fase 2.1. Versión FixI-Com-OldW-NBloq.



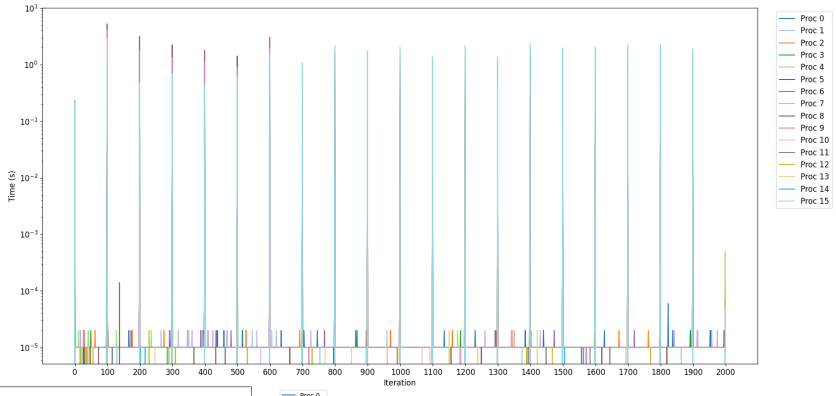

(a) Tiempo de cómputo de nuevas celdas en cada iteración (segundos)



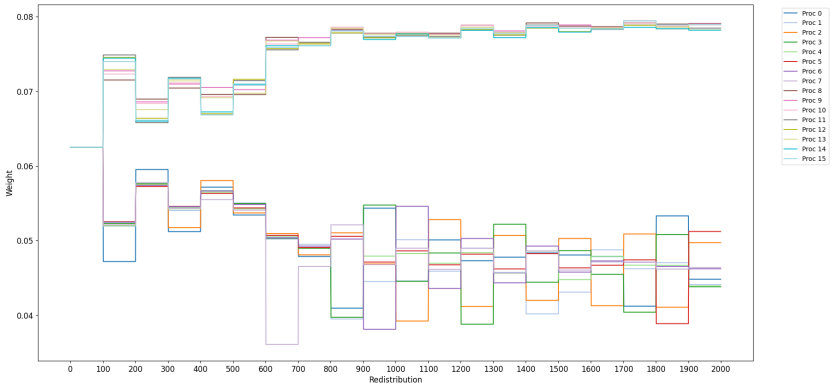

(b) Tiempo de comunicación de vecinos en cada iteración (segundos)




(c) Tiempo de ejecución de cada iteración (segundos)




(d) Tiempo de WeightedRedistribute en cada iteración (segundos)




(e) Reparto de pesos en cada iteración (proporción [0,1])



Figura 5.8: Resultados gráficos de la ejecución del programa de Jacobi2D con parámetros de la fase 2.1. Versión FixI-Com-AvgW-NBloq.

**FixI-ComRow-NewW-NBloq** (figura 5.9). Los resultados de esta versión sí corresponden a las expectativas que se tenían del funcionamiento de WeightedRedistribute. Los tiempos de iteración (figura 5.9c) son más bajos que en las versiones anteriores, y se van estabilizando a lo largo de las iteraciones. Al contrario que sucedía anteriormente, los picos inducidos por las redistribuciones son ascendentes. Esto se debe a un incremento del tiempo de comunicación entre vecinos en las iteraciones en las que se distribuye. El comportamiento anómalo en el proceso 15 en la iteración 400 puede deberse a un problema en ese proceso, que provoca que tarde mucho en computar su bloque (figura 5.9a), por lo que en la iteración siguiente se le asigna un peso muy pequeño.

#### 5.3.3.2. Comparación entre versiones

La tabla 5.6 muestra los resultados de la comparación de tiempo total de ejecución entre la mejor versión de WeightedRedistribute de las desarrolladas (FixI-ComRow-NewW-NBloq) y la versión original sin redistribución (Orig-Punteros), tomando como mejor la versión con redistribución. El intervalo está muy alejado del valor cero, lo que indica que la mejora de rendimiento es bastante grande y valida la hipótesis de que el uso de la redistribución mejora el tiempo de un programa.

| | | **Tam. media 100** |
|---|---|---|
| **Procs** | **16** | [-216.483, -201.153 ] |

(a) Tamaño matriz: 160000 × 2500

Tabla 5.6: Comparación del tiempo total de ejecución (segundos) del programa de Jacobi2D con parámetros de la fase 2.1. Versiones FixI-ComRow-NewW-NBloq y Orig-Punteros.

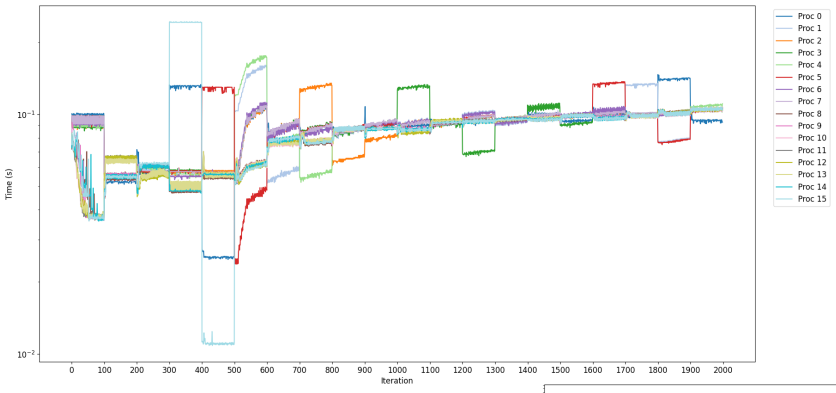
La tabla 5.6 muestra los resultados de la comparación entre la versión FixI-ComRow-NewW-NBloq y la versión con un reparto equitativo de la carga (FixI-Com-Pre-Bloq), tomando como mejor a FixI-Iter-OldW-Bloq. La mejora de rendimiento es notable, por lo que también se valida la consideración de que la redistribución en base a la potencia de cada proceso es mejor que la redistribución utilizando un reparto equitativo.

| | | **Tam. media 100** |
|---|---|---|
| **Procs** | **16** | [-193.071, -172.692 ] |

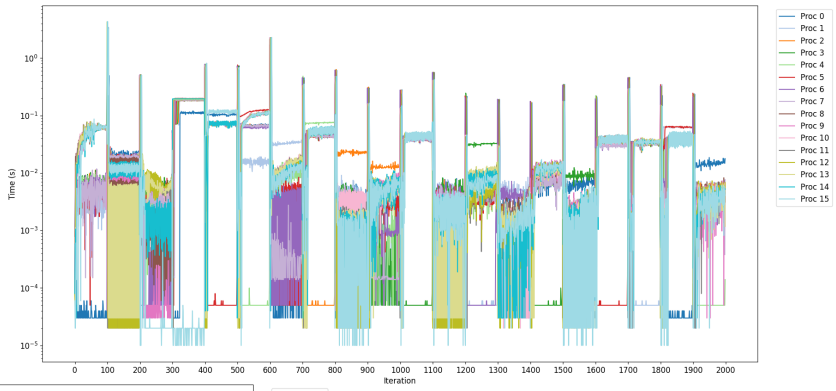
(a) Tamaño matriz: 160000 × 2500

Tabla 5.7: Comparación del tiempo total de ejecución (segundos) del programa de Jacobi2D con parámetros de la fase 2.1. Versiones FixI-ComRow-NewW-NBloq y FixI-Com-Pre-Bloq.

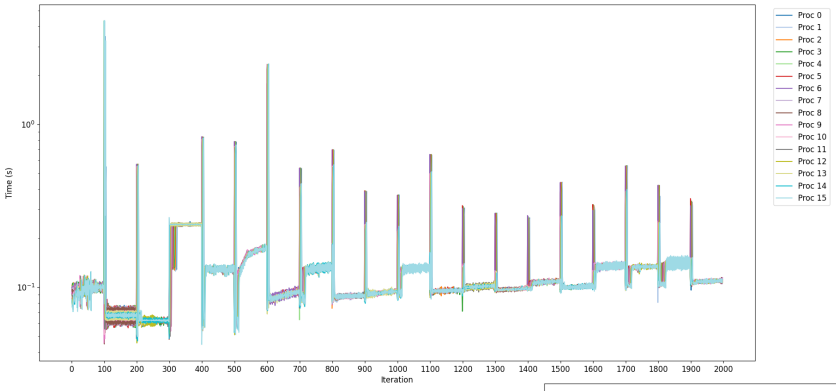
Finalmente, la tabla 5.8 muestra el resultado de la comparación entre las versiones FixI-ComRow-NewW-NBloq, que utiliza un intervalo de redistribución constante e igual a 100 iteraciones; e IncrI-ComRow-NewW-NBloq, que tiene las mismas características de cálculo de pesos pero utiliza un intervalo creciente. Se toma como mejor el código con intervalo creciente. El uso de este intervalo mejora el rendimiento del programa al realizarse un menor número de redistribuciones, lo que supone un menor sobrecoste.



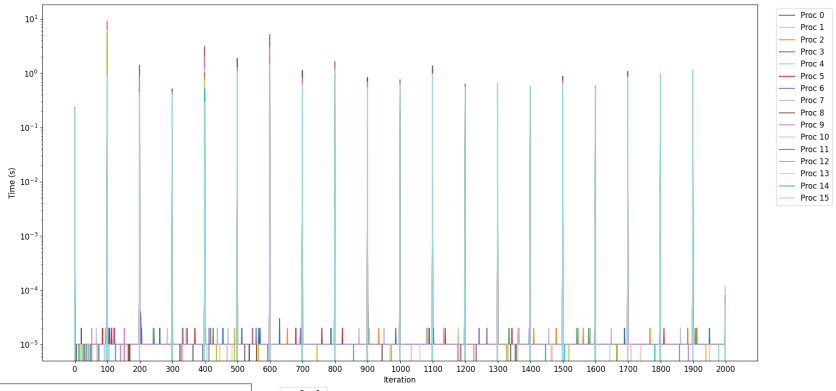

(a) Tiempo de cómputo de nuevas celdas en cada iteración (segundos)



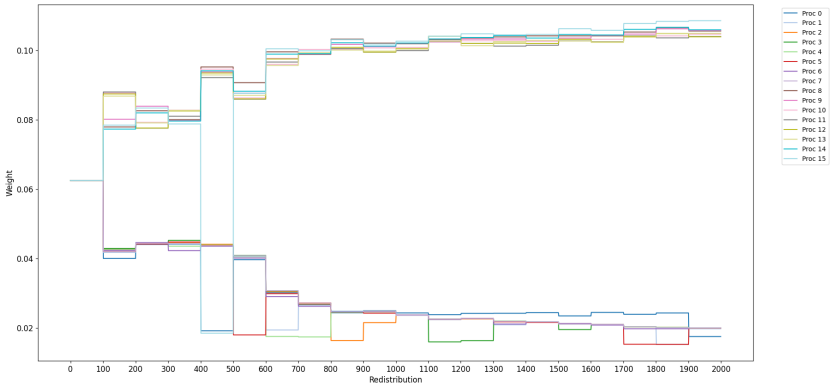

(b) Tiempo de comunicación de vecinos en cada iteración (segundos)




(c) Tiempo de ejecución de cada iteración (segundos)




(d) Tiempo de WeightedRedistribute en cada iteración (segundos)




(e) Reparto de pesos en cada iteración (proporción [0,1])



Figura 5.9: Resultados gráficos de la ejecución del programa de Jacobi2D con parámetros de la fase 2.1. FixI-ComRow-NewW-NBloq.

| | | Tam. media 100 |
|---|---|---|
| **Procs** | **16** | [-78.479, -21.738] |

(a) Tamaño matriz: 160000 × 2500

Tabla 5.8: Comparación del tiempo total de ejecución (segundos) del programa de Jacobi2D con parámetros de la fase 2.2. Versiones IncrI-ComRow-NewW-NBloq y FixI-ComRow-NewW-NBloq.

### 5.3.4. Conclusiones

Todas las versiones de WeightedRedistribute desarrolladas tenían como objetivo mejorar el rendimiento del programa Jacobi2D original sin redistribución. Sin embargo, solo dos de ellas, FixI-ComRow-NewW-NBloq e IncrI-ComRow-NewW-NBloq, lo han conseguido, ya que el resto presentaban errores en el cálculo de los pesos.

El cálculo en base al tiempo de una iteración introduce el problema de la sincronización implícita al ejecutar el Pattern de comunicación de vecinos. Esto se soluciona utilizando el tiempo de cómputo, pero si se mide como el total de todas las filas no refleja correctamente la rapidez de cada proceso. Midiendo el tiempo de cómputo por fila se asegura que las medias reflejen el tiempo empleado en ejecutar la misma cantidad de trabajo. Por otra parte, la normalización de los pesos es un paso innecesario ya que el constructor de Layouts se encarga de normalizarlos al repartir los bloques entre los procesos.

Se ha observado que la parte de comunicaciones, tanto la transmisión de bordes entre vecinos, como la transmisión de medias, como la redistribución en sí, tiene un gran peso en el tiempo total del programa, ya que como se ha comentado introduce una sincronización que provoca que los procesos más rápidos tengan que esperar ociosos a recibir los datos. Esto se ha solucionado en una de las comunicaciones utilizando una versión no bloqueante.

Por último, hay que mencionar la influencia del tamaño del intervalo en el tiempo de ejecución de la aplicación. Un intervalo grande reduce el sobrecoste de la función al ejecutarse menos redistribuciones, pero no se alcanza la estabilidad en los pesos si el número de redistribuciones es bajo. Por el contrario, un intervalo pequeño introduce más *overhead* pero permite llegar antes al punto de equilibrio en los tiempos de iteración de cada proceso. Es por ello que un intervalo creciente es una estrategia interesante para obtener las ventajas de las dos opciones.

## 5.4. Conclusiones del capítulo

En este capítulo se ha presentado la experimentación realizada en este proyecto. Se han descrito las consideraciones(hipótesis) a validar o refutar en las pruebas de rendimiento, la metodología seguida en cada fase de pruebas y los detalles técnicos de la plataforma de ejecución. Seguidamente, se han mostrado los resultadoso obtenidos. Las pruebas realizadas a la clase HitAvg han servido para validar su buen funcionamiento y analizar el comportamiento de los tres tipos de media probados (SMA, EMA y LWMA) con diferentes tendencias de los valores. Las pruebas de rendimiento al programa de Jacobi2D antes de introducir la función de redistribución han mostrado el coste

que suponía realizar las copias en profundidad en comparación con el intercambio de punteros. Finalmente, en las pruebas realizadas al programa tras la aplicación de WeightedRedistribute se ha estudiado el comportamiento de las versiones desarrolladas con las variaciones hechas al diseño inicial, se han observado los motivos del mal rendimiento de varias de ellas y se ha determinado la mejor versión, con la que se han validado las hipótesis planteadas.

# Capítulo 6

# Conclusiones

En este capítulo se introducen los siguientes aspectos:

- Los objetivos del proyecto que se han completado.
- Las líneas de trabajo futuras a partir de este proyecto.
- Una opinión personal sobre el desarrollo del proyecto.

## 6.1. Objetivos cumplidos

Este trabajo es parte de un proyecto de investigación realizado y dirigido por el Grupo Trasgo. Se ha desarrollado una función de distribución dinámica y adaptativa de carga y datos entre procesos utilizando los recursos de la librería Hitmap, llamada WeightedRedistribute. El uso de la función mejora el rendimiento de los programas al repartir la carga de trabajo adecuadamente a la capacidad computacional de cada proceso. De entre los objetivos genéricos propuestos para este proyecto, descritos en la sección 1.3.1, se han conseguido los siguientes:

- Se ha estudiado y comprendido la librería Hitmap en las versiones 1.3 y 2.0.
- Se ha diseñado e implementado un prototipo de WeightedRedistribute para Hitmap v1.3.
- Se ha estudiado y comprendido el problema de difusión del calor de Jacobi en espacios bidimensionales y se ha adaptado la función a las necesidades del programa.
- Se ha diseñado y ejecutado un plan de pruebas para validar la eficacia de WeightedRedistribute aplicada al programa de Jacobi y se han analizado y extraído conclusiones de los resultados.

A continuación, se enumera el resultado de los objetivos propuestos relacionados con las características que debía tener la función:

- Se ha conseguido que la función oculte todos los detalles de su funcionamiento al usuario, y aplique los cambios de nueva carga y datos sobre las estructuras pasadas como parámetros.
- No se ha conseguido el objetivo de mínima intrusión en los programas. El usuario ha de definir funciones de inicialización de Tiles y Patterns de los tipos predefinidos (`HitTileInitFunction` y `HitPatternFunction`) y una variable HitAvg.

- Se ha conseguido que la función sea adaptable a un número de Tiles y Patterns variable.
- Se ha conseguido que la función mejore el rendimiento de los programas en los que se aplica, asignando menos trabajo a los procesos menos potentes y viceversa.
- Se ha conseguido que la función no suponga un gran consumo de tiempo y memoria. Todas las estructuras secundarias que se utilizan son liberadas al final de cada llamada.

## 6.2. Trabajo futuro

Tras la elaboración de este Trabajo de Fin de Grado, surgen varias líneas de trabajo futuro. Antes de abordar las modificaciones, habría que completar la experimentación con la función existente, para obtener más datos y sacar conclusiones más precisas sobre las comparaciones. Por otra parte, falta incluir la comparación entre tipos de medias (SMA, LWMA y EMA).

La implementación de una versión de la función para Hitmap v.2, tarea que estaba prevista para este proyecto, tiene una parte directa de traducir WeightedRedistribute a las funciones y macros de Hitmap2, y otra de adaptación de los Tiles y Layouts al uso de Tiles distribuidos.

La posibilidad de trabajar con topologías de dos dimensiones, cuyos problemas de implementación actuales se expusieron en la sección 3.1.2, mejoraría la adaptabilidad de WeightedRedistribute y aumentaría el espacio de experimentación, permitiendo explorar qué topología es más eficiente para la redistribución.

El uso de intervalos de redistribución crecientes reduce el sobrecoste de la función sin afectar a alcanzar el punto de estabilidad en el reparto de pesos. Una propuesta futura es probar con otros valores para aumentar el intervalo, o que sea la función quien establezca el tamaño del intervalo estudiando la tendencia de los tiempos. Al detectar que se ha alcanzado la estabilidad en los pesos, cesaría las redistribuciones.

## 6.3. Valoración personal

Este proyecto ha supuesto una toma de contacto más profunda con la metodología de trabajo en un Grupo de Investigación, que ya había comenzado con la realización de las Prácticas en Empresa. He comprobado que en ocasiones, como ha sido esta, la idea inicial para un nuevo trabajo es muy simple, así que se realiza un primer diseño muy sencillo y es en la fase de implementación donde se van añadiendo cambios según se ven necesarios. La fase experimental es muy importante para medir el rendimiento del trabajo con parámetros diversos, por lo que ocupa gran parte del tiempo de desarrollo del proyecto. La elaboración de documentación y artículos (en este caso esta memoria) también tiene mucha importancia para mostrar al resto de la comunidad científica el funcionamiento de las herramientas creadas y los resultados y conclusiones obtenidos en la experimentación.

Habría que hacer una reflexión sobre los problemas que han causado la prolongación del proyecto. Por una parte, los fallos en una de las máquinas de la experimentación se habían contemplado en el análisis de riesgos, pero en lugar de utilizar otra máquina como se había establecido en el

plan de contingencia, se decidió esperar a que estuviese arreglada para no tener que repetir todas las pruebas previas. Por otra parte, la planificación del proyecto no ha sido del todo acertada. Si se hubiese hecho un análisis más profundo del programa de Jacobi, quizá se hubieran detectado las necesidades de ese programa y se habrían añadido al diseño inicial, en vez de aparecer como desviaciones. También, la parte experimental necesitaba algo más de tiempo del asignado.

En resumen, en general estoy satisfecha con el desarrollo del trabajo. Este proyecto ha ampliado mis conocimientos sobre computación paralela, que había adquirido en la asignatura del mismo nombre. También he aprendido sobre diseño de experimentaciones para cubrir los casos relevantes, análisis de resultados y extracción de conclusiones; y sobre búsqueda y buen uso de bibliografía. Quedan abiertas líneas de investigación para probar la función con nuevas aplicaciones y nuevas fórmulas que puedan hacer que WeightedRedistribute sea más funcional y mejore más el rendimiento de los programas, y este trabajo ha sentado las bases para hacerlo.

# Bibliografía


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

[2] *What is high performance computing?*, `https://insidehpc.com/hpc-basic-training/what-is-hpc/`, (Último acceso: 9-sep-2020).

[3] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban y C.-L. Wang, «Heterogeneous computing: Challenges and opportunities», *Computer*, vol. 26, n.º 6, págs. 18-27, 1993.

[4] M. J. Flynn, «Very high-speed computing systems», *Proceedings of the IEEE*, vol. 54, n.º 12, págs. 1901-1909, 1966.

[5] *Top500*, `https://www.top500.org/`, (Último acceso: 2-sep-2020).

[6] J. Xue, *Loop tiling for parallelism*. Springer Science & Business Media, 2000, vol. 575.

[7] M. Wolfe, «More iteration space tiling», en *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, 1989, págs. 655-664.

[8] G. Cybenko, «Dynamic load balancing for distributed memory multiprocessors», *Journal of parallel and distributed computing*, vol. 7, n.º 2, págs. 279-301, 1989.

[9] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks y K. Warren, «Introduction to UPC and language specification», Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, inf. téc., 1999.

[10] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua y C. Von Praun, «Programming for parallelism and locality with hierarchically tiled arrays», en *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, págs. 48-57.

[11] 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, 2013.

[12] B. Hughes y M. Cotterell, «Software Project Management», en, 5.ª ed. McGraw-Hill Education, 2009, cap. 4.

[13] A. Alshamrani y A. Bahattab, «A comparison between three SDLC models waterfall model, spiral model, and Incremental/Iterative model», *International Journal of Computer Science Issues (IJCSI)*, vol. 12, n.º 1, pág. 106, 2015.

[14] B. Hughes y M. Cotterell, «Software Project Management», en, 5.ª ed. McGraw-Hill Education, 2009, cap. 6.

[15] B. Hughes y M. Cotterell, «Software Project Management», en, 5.ª ed. McGraw-Hill Education, 2009, cap. 7.

[16] B. W. Boehm, «Software risk management: principles and practices», *IEEE software*, vol. 8, n.º 1, págs. 32-41, 1991.

[17] *Grupo Trasgo*, `https://trasgo.infor.uva.es/`, (Último acceso: 2-sep-2020).

[18] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, `https://www.mpi-forum.org/`, ver. 1.3, 4 de jun. de 2015.

[19] F. Darema, «The spmd model: Past, present and future», en *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, Springer, 2001, págs. 1-1.

[20] W. Gropp, W. D. Gropp, E. Lusk, A. Skjellum y A. D. F. E. E. Lusk, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[21] A. Hayes, *Moving Average (MA)*, `https://www.investopedia.com/terms/m/movingaverage.asp`, (Último acceso: 9-sep-2020).

[22] A. Hayes, *Simple Moving Average (SMA)*, `https://www.investopedia.com/terms/s/sma.asp`, (Último acceso: 9-sep-2020).

[23] T. Finch, «Incremental calculation of weighted mean and variance», *University of Cambridge*, vol. 4, n.º 11-5, págs. 41-42, 2009.

[24] C. Mitchell, *Linearly Weighted Moving Average*, `https://www.investopedia.com/terms/l/linearlyweightedmovingaverage.asp`, (Último acceso: 9-sep-2020).

[25] A. Hayes, *Exponential Moving Average (EMA)*, `https://www.investopedia.com/terms/e/ema.asp`, (Último acceso: 9-sep-2020).

[26] C Reference, *Enumerations*, `https://en.cppreference.com/w/c/language/enum`, (Último acceso: 21-ago-2020).

[27] T. Wang, Y. Yao, L. Han, D. Zhang e Y. Zhang, «Implementation of Jacobi iterative method on graphics processor unit», en *2009 IEEE International Conference on Intelligent Computing and Intelligent Systems*, IEEE, vol. 3, 2009, págs. 324-327.

[28] A. Schäfer y D. Fey, «High performance stencil code algorithms for GPGPUs», *Procedia Computer Science*, vol. 4, págs. 2027-2036, 2011.

# Apéndice A

# Algoritmos

## A.1. Consideraciones sobre la notación de los algoritmos

A continuación, se indican algunas consideraciones acerca de la notación utilizada para los algoritmos en este documento y el significado de cada expresión según su tipografía o estilo. El apartado Entrada al principio de un algoritmo nombra los parámetros del procedimiento, y el apartado Datos explica algunas variables utilizadas a lo largo del mismo. En este apartado solo se incluyen las variables más importantes y las que su función no se define completamente a través de su nombre. Las que se han declarado como estáticas aparecen subrayadas. En cuanto a los estilos, las palabras reservadas se escriben en negrita (**si ... entonces**), las variables en cursiva (*callClock, oldTile*) y los comentarios en fuente monoespaciada (`// Normalizar pesos`). Por último, para las funciones se establecen dos tipos. Las funciones ya existentes en Hitmap u otra librería se escriben en versalita (AllToAll(), Alloc()), y en ocasiones se abrevia su nombre para que la lectura del algoritmo resulte más sencilla. Las funciones que aparecen en fuente estándar (añadirDatoAMedia(), **si** suficientes datos ...) indican acciones a realizar que no tienen en ese momento un método específico para ello.

## A.2. Algoritmos de WeightedRedistribute

**Algoritmo A.1:** Diseño inicial del algoritmo de WeightedRedistribute (Hitmap v1.3).

**Entrada:** *ttile, numTiles, lay*
**Datos:** *callClock*: HitClock para medir el tiempo de una iteración,
*times*, *allTimes*: Tiles para comunicar los tiempos de iteración,
*weights*: estructura HitWeights para almacenar los pesos

```
inicio
    callClock ← HIT_CLOCK_NULL
    si callClock = HIT_CLOCK_NULL entonces
        /* primera llamada a la función, no hay datos de la iteración anterior
        y no se hace nada salvo inicializar callClock                          */
    fin
    en otro caso
        STOP(callClock)
        añadirDatoAMedia(callClock.seconds)
        si suficientes datos para calcular la media entonces
            average ← Calcular media de tiempos de iteración
            ALLOC(times, hit_NProcs)
            FILL(times, average)
            ALLOC(allTimes, hit_NProcs)
            ALLTOALL(lay, times, allTimes)

            // Normalizar pesos
            total ← ∑ allTimes
            para i ← 0 a hit_NProcs hacer  w[i] ← 1 − (allTimes[i]/total)
            total ← ∑ w
            para i ← 0 a hit_NProcs hacer  w[i] ← w[i]/total
            weights ← w

            newLayout ← LAYOUT(layWeighted, topologyPlain, lay.originalShape, weights)

            para i ← 0 a numTiles hacer
                oldTile ← ttile[i]
                newTile ← oldTile
                newShape ← LOCALSHAPE(newLayout)
                SHAPEALLOC(newTile, newShape)
                REDISTRIBUTE(lay, newLayout, oldTile, newTile)
            fin

            lay ← newLayout

            Resetear media
            Liberar estructuras auxiliares
        fin
    fin
    START(callClock)
fin
```

**Algoritmo A.2:** Algoritmo final de WeightedRedistribute (Hitmap v1.3).

**Entrada:** *ttile, numTiles, tilef, lay, avg, patt, numPatterns, pattf, shpview, time*
**Datos:** *callClock*: HitClock para medir el tiempo de una iteración,
*layoutComm*: Layout para comunicaciones entre procesos,
*allTimes*: medias de tiempos de todos los procesos,
*redisInterval*: tamaño del intervalo de redistribución,
*valuesToRedis*: medias calculadas hasta la redistribución

```
inicio
    callClock ← HIT_CLOCK_NULL
    si callClock = HIT_CLOCK_NULL entonces
        layoutComm ← LAYOUT(layCopy, topologyPlain, SHAPE(1,1))
        redisInterval ← avg.windowSize
        valuesToRedis ← avg.windowSize − 1
    fin
    si ¬ PROCISACTIVE(lay) entonces time = 0
    INSERT(avg, time)
    average ← GETAVG(avg)
    si average ≠ −1 entonces
        valuesToRedis ← valuesToRedis + 1
        si valuesToRedis = redisInterval entonces
            timePerRow ← average/ DIMCARD(ttile[0], 0)
            ALLOC(times, hit_NProcs)
            FILL(times, timePerRow)
            ALLOC(allTimes, hit_NProcs)
            FILL(allTimes, 0)
            IALLTOALL(layoutComm, times, allTimes)
        fin
        si no, si valuesToRedis = redisInterval + 1 entonces
            WAIT(IALLTOALL)
            total ← ∑ allTimes
            para i ← 0 a hit_NProcs hacer  w[i] ← total/allTimes[i]
            weights ← w
            newLayout ← LAYOUT(layWeighted, topologyPlain, lay.originalShape, weights)
            para i ← 0 a numTiles hacer
                oldTile ← ttile[i]
                newTile ← oldTile
                oldPattern ← patt[i]
                newShape ← LOCALSHAPE(newLayout)
                newShape ← SHAPEEXPAND(expandedShape, shpview)
                SHAPEALLOC(newTile, newShape)
                TILEF(newTile)
                REDISTRIBUTE(oldLayout, newLayout, oldTile, newTile)
```

```
                    si i < numPatterns entonces
                        newPattern ← PATTF(newTile, newLayout)
                        FREE(oldPattern)
                        oldPattern ← newPattern
                    fin
                    FREE(oldTile)
                fin
            fin
            FREE(lay)
            lay ← newLayout
            RESET(avg)
            redisInterval ← redisInterval + avg.windowSize
            valuesToRedis ← avg.windowSize − 1
            FREE(times)
            FREE(allTimes)
        fin
    fin
    START(callClock)
fin
```

# Apéndice B

# Tablas

## B.1. Comparación de enfoques par-impar y punteros

Esta sección incluye las tablas completas de la comparación entre los enfoques de iteraciones par-impar y el intercambio de punteros en el programa original de difusión del calor con Jacobi 2D en las fases 1.1 y 1.2 de las pruebas de rendimiento; y la comparación entre el intercambio de punteros y la versión con copia en profundidad en el programa original, también en las fases 1.1 y 1.2.

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-0.548, 1.699] | [-1.576, 0.765] | [-2.302, -0.178] | [-3.191, -0.431] |
| 20 | [-0.005, -0.003] | [-0.003, 0.004] | [0.000, 0.007] | [-0.001, 0.007] |
| 30 | [-3.352, 0.032] | [-1.659, 1.802] | [-0.512, 1.411] | [-0.368, 0.593] |
| 40 | [-0.024, -0.000] | [-0.022, 0.011] | [-0.003, 0.024] | [0.004, 0.017] |

(a) Tamaño matriz: 1000 × 1000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-0.094, 0.037] | [-0.500, 0.259] | [-0.303, 1.048] | [-1.138, 0.971] |
| 20 | [0.004, 0.011] | [-0.013, -0.003] | [-0.019, -0.002] | [-0.002, 0.010] |
| 30 | [-0.738, 1.572] | [-4.519, -0.326] | [-0.198, 1.071] | [-0.424, 0.925] |
| 40 | [0.003, 0.010] | [-0.002, 0.009] | [-0.003, 0.018] | [0.005, 0.009] |

(b) Tamaño matriz: 2500 × 2500

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-0.032, -0.003] | [-0.043, -0.020] | [-0.027, -0.016] | [-0.028, -0.016] |
| 20 | [-0.014, 0.000] | [-0.025, -0.004] | [-0.008, 0.008] | [-0.025, -0.007] |
| 30 | [-0.652, 0.270] | [-0.244, 0.532] | [-0.177, 0.631] | [-0.303, 0.664] |
| 40 | [-0.022, -0.000] | [-0.017, -0.004] | [-0.002, 0.005] | [-0.018, -0.009] |

(c) Tamaño matriz: 5000 × 5000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-0.133, -0.050] | [-0.144, -0.083] | [-0.126, -0.091] | [-0.139, -0.072] |
| 20 | [-0.066, -0.044] | [-0.056, 0.001] | [-0.108, -0.009] | [-0.090, -0.063] |
| 30 | [-0.910, 1.913] | [-0.069, -0.023] | [-0.042, -0.014] | [-2.425, 1.045] |
| 40 | [-0.060, -0.025] | [-0.037, -0.017] | [-0.045, -0.017] | [-0.084, -0.047] |

(d) Tamaño matriz: 10000 × 10000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-0.207, -0.179] | [-0.308, -0.231] | [-0.287, -0.189] | [-0.323, -0.239] |
| 20 | [-0.223, -0.100] | [-0.190, -0.103] | [-0.232, -0.173] | [-0.217, -0.145] |
| 30 | [-0.157, -0.119] | [-0.124, -0.079] | [-0.117, -0.082] | [-0.681, 0.028] |
| 40 | [-0.090, -0.036] | [-0.120, -0.060] | [-0.111, -0.054] | [-0.072, -0.026] |

(e) Tamaño matriz: 15000 × 15000

Tabla B.1: Comparación del tiempo de ejecución total (segundos) del programa de Jacobi2D con parámetros de la fase 1.1. Versiones Orig-Punteros y Orig-ParImpar.

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 20 | [-0.355, -0.295] | [-0.378, -0.267] | [-0.403, -0.292] | [-0.355, -0.286] |
| 35 | [-0.182, -0.120] | [-0.219, -0.188] | [-0.226, -0.201] | [-1.232, 2.192] |
| 40 | [-0.196, -0.148] | [-0.177, -0.116] | [-0.182, -0.165] | [-0.173, -0.139] |
| 50 | [-0.181, -0.164] | [-0.155, -0.108] | [-0.197, -0.166] | [-0.155, -0.118] |

(a) Tamaño matriz: 20000 × 20000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 20 | [-0.549, -0.505] | [-0.602, -0.436] | [-0.512, -0.374] | [-0.573, -0.524] |
| 35 | [-0.266, -0.225] | [-0.273, -0.240] | [-0.341, -0.313] | [-0.298, -0.275] |
| 40 | [-0.295, -0.260] | [-0.195, -0.167] | [-0.290, -0.220] | [-0.247, -0.202] |
| 50 | [-0.257, -0.231] | [-0.204, -0.169] | [-0.229, -0.162] | [-0.180, -0.126] |

(b) Tamaño matriz: 25000 × 25000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 20 | [-0.950, -0.810] | [-0.925, -0.810] | [-0.987, -0.779] | [-0.949, -0.803] |
| 35 | [-10.139, 3.896] | [-0.519, -0.419] | [-0.470, -0.398] | [-0.462, -0.397] |
| 40 | [-0.467, -0.361] | [-0.504, -0.398] | [-0.384, -0.346] | [-0.470, -0.358] |
| 50 | [-0.294, -0.245] | [-0.274, -0.221] | [-0.305, -0.265] | [-0.325, -0.248] |

(c) Tamaño matriz: 30000 × 30000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 20 | [-1.184, -0.790] | [-1.435, -1.224] | [-3.048, -0.864] | [-2.959, -0.991] |
| 35 | [-0.673, -0.180] | [-0.860, -0.087] | [-0.973, -0.304] | [-0.788, -0.626] |
| 40 | [-0.718, -0.214] | [-1.191, -0.831] | [-1.256, -0.670] | [-0.627, -0.225] |
| 50 | [-0.410, -0.328] | [-0.467, -0.332] | [-0.393, -0.299] | [-0.576, -0.416] |

(d) Tamaño matriz: 37500 × 37500

Tabla B.2: Comparación del tiempo de ejecución total (segundos) del programa de Jacobi2D con parámetros de la fase 1.2. Versiones Orig-Punteros y Orig-ParImpar).

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-1.804, 0.436] | [-1.991, 0.678] | [-0.675, 0.300] | [-1.778, 0.934] |
| 20 | [0.009, 0.020] | [0.014, 0.017] | [0.015, 0.020] | [0.006, 0.020] |
| 35 | [-2.243, 0.100] | [-0.114, 4.185] | [0.153, 2.664] | [-1.705, 1.523] |
| 40 | [0.046, 0.086] | [0.059, 0.067] | [0.048, 0.088] | [0.055, 0.086] |

(a) Tamaño matriz: 1000 × 1000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-2.146, -1.131] | [-1.650, 0.166] | [-2.597, -0.547] | [-1.605, -0.510] |
| 20 | [-0.994, -0.978] | [-0.998, -0.993] | [-1.020, -0.999] | [-0.998, -0.989] |
| 35 | [-0.993, 1.076] | [-4.637, -0.653] | [-0.353, 0.947] | [-0.216, 1.572] |
| 40 | [-0.240, -0.228] | [-0.240, -0.232] | [-0.235, -0.231] | [-0.234, -0.214] |

(b) Tamaño matriz: 2500 × 2500

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-6.945, -6.934] | [-6.961, -6.943] | [-6.940, -6.924] | [-6.954, -6.935] |
| 20 | [-5.099, -5.050] | [-5.100, -5.080] | [-5.118, -5.052] | [-5.087, -5.064] |
| 35 | [-3.633, -2.456] | [-3.558, -2.973] | [-3.471, -2.540] | [-3.481, -2.192] |
| 40 | [-2.351, -2.325] | [-2.360, -2.329] | [-2.356, -2.330] | [-2.346, -2.336] |

(c) Tamaño matriz: 5000 × 5000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-29.171, -29.043] | [-29.168, -29.135] | [-29.184, -29.098] | [-29.146, -29.028] |
| 20 | [-21.224, -21.140] | [-21.148, -21.086] | [-21.197, -21.063] | [-21.265, -21.135] |
| 35 | [-15.157, -12.351] | [-14.306, -14.280] | [-14.270, -14.246] | [-14.310, -14.234] |
| 40 | [-10.483, -10.463] | [-10.523, -10.468] | [-10.499, -10.455] | [-10.531, -10.487] |

(d) Tamaño matriz: 10000 × 10000

| Procs \ Tam. media | 10 | 25 | 40 | 50 |
|---|---|---|---|---|
| 15 | [-65.406, -65.253] | [-65.462, -65.387] | [-65.481, -65.281] | [-65.514, -65.395] |
| 20 | [-49.657, -49.421] | [-49.520, -49.453] | [-49.533, -49.377] | [-49.697, -49.441] |
| 35 | [-32.310, -32.296] | [-32.301, -32.233] | [-32.303, -32.270] | [-32.364, -32.331] |
| 40 | [-24.280, -24.206] | [-24.294, -24.191] | [-24.314, -24.278] | [-24.265, -24.175] |

(e) Tamaño matriz: 15000 × 15000

Tabla B.3: Comparación del tiempo de ejecución total (segundos) del programa de Jacobi2D con parámetros de la fase 1.1. Versiones Orig-Punteros y Orig.

| | | Tam. media | | | |
|---|---|---|---|---|---|
| | | 10 | 25 | 40 | 50 |
| Procs | 20 | [-33.827, -33.697] | [-33.931, -33.885] | [-33.989, -33.792] | [-33.762, -33.740] |
| | 35 | [-19.763, -19.726] | [-19.795, -19.727] | [-19.811, -19.768] | [-20.843, -17.432] |
| | 40 | [-17.430, -17.375] | [-17.396, -17.343] | [-17.414, -17.330] | [-17.388, -17.315] |
| | 50 | [-13.785, -13.726] | [-13.783, -13.753] | [-13.784, -13.770] | [-13.794, -13.752] |

(a) Tamaño matriz: 20000 × 20000

| | | Tam. media | | | |
|---|---|---|---|---|---|
| | | 10 | 25 | 40 | 50 |
| Procs | 20 | [-53.958, -53.799] | [-53.872, -53.756] | [-53.759, -53.676] | [-54.084, -53.802] |
| | 35 | [-30.978, -30.914] | [-30.951, -30.881] | [-31.001, -30.947] | [-41.087, -26.448] |
| | 40 | [-27.212, -27.066] | [-27.049, -26.995] | [-27.211, -27.135] | [-27.201, -27.020] |
| | 50 | [-21.263, -21.184] | [-21.317, -21.233] | [-21.323, -21.256] | [-21.354, -21.106] |

(b) Tamaño matriz: 25000 × 25000

| | | Tam. media | | | |
|---|---|---|---|---|---|
| | | 10 | 25 | 40 | 50 |
| Procs | 20 | [-78.511, -77.904] | [-78.022, -77.851] | [-78.198, -77.857] | [-78.310, -77.979] |
| | 35 | [-44.495, -44.366] | [-69.334, -33.707] | [-44.612, -44.496] | [-44.544, -44.481] |
| | 40 | [-39.617, -39.414] | [-39.403, -39.332] | [-39.546, -39.445] | [-39.526, -39.302] |
| | 50 | [-31.149, -31.038] | [-31.094, -30.969] | [-31.070, -31.020] | [-31.032, -30.915] |

(c) Tamaño matriz: 30000 × 30000

| | | Tam. media | | | |
|---|---|---|---|---|---|
| | | 10 | 25 | 40 | 50 |
| Procs | 20 | [-124.457, -121.611] | [-125.532, -123.521] | [-125.686, -121.963] | [-125.971, -123.420] |
| | 35 | [-69.203, -68.778] | [-68.922, -68.708] | [-68.974, -68.572] | [-69.258, -68.716] |
| | 40 | [-59.817, -59.517] | [-60.457, -59.791] | [-60.231, -59.781] | [-59.679, -59.369] |
| | 50 | [-47.801, -47.656] | [-47.786, -47.709] | [-47.868, -47.738] | [-47.881, -47.699] |

(d) Tamaño matriz: 37500 × 37500

Tabla B.4: Comparación del tiempo de ejecución total (segundos) del programa de Jacobi2D con parámetros de la fase 1.2. Versiones Orig-Punteros y Orig.

# Apéndice C

# Descripción del software complementario

En la página del Grupo Trasgo (enlace adjunto) puede descargarse un archivo .zip con el software desarrollado y utilizado en este trabajo. El archivo contiene los siguientes elementos:

- Un directorio, *hitmap*, que contiene la librería Hitmap en su versión 1.3.
- Un documento README donde se explican brevemente los cambios que se han realizado en la librería: clases añadidas o editadas, programas de ejemplo creados, etc.

# Apéndice D

# Manual de uso

El código D.1 muestra cómo se deben ejecutar los programas de Jacobi bidimensional con WeightedRedistribute para Hitmap v1.3, incluidos en el software complementario. Para lanzarlos con varios procesos hay que incluir `mpiexec -n <nProcs>` antes del nombre del ejecutable. Los tres argumentos obligatorios de los programas son el número de filas y columnas de la matriz y el número de iteraciones de la simulación. Todos tienen que ser números enteros.

```
1  ./jacobi2DWeightedRedistribute [--hit_alb_ema | --hit_alb_lwma]
2   [--hit_alb_window=<n>] <numRows> <numCols> <numIters>
```

Código D.1: Sintaxis de los ejecutables de Jacobi2D con WeightedRedistribute.

Los argumentos opcionales sirven para variar el tipo y el tamaño de la media que se utiliza, que por defecto es simple y con un número de datos igual a 10. Las opciones también siguen la nomenclatura de los elementos de Hitmap.

- `--hit_alb_ema`. La media es de tipo exponencial.
- `--hit_alb_lwma`. La media es de tipo linealmente ponderado.
- `--hit_alb_window=<n>`. La media contendrá el número de datos indicado. Tiene que ser un número entero.