Universidad de Valladolid

E.T.S Ingeniería Informática

Trabajo Fin De Grado

Grado en Ingeniería Informática

Explotación de una política de partición de datos para aplicaciones paralelas

Autor:
Carlos Baeza Sanz

Tutor:
Dr. Arturo González Escribano

Resumen

El paralelismo de datos es un paradigma de computación paralela donde se ejecutan las mismas tareas sobre diferentes datos de una estructura. Repartiendo adecuadamente los datos entre los diferentes elementos de proceso se reparten las tareas a realizar y se incrementa la velocidad de ejecución. La política de partición de datos empleada influye en gran medida en el rendimiento final de la aplicación y debe de ser elegida teniendo en cuenta el tipo de problema que se quiere paralelizar.

Este proyecto se centra en el estudio de una política de partición de datos compleja: la partición bloque-cíclica. Se estudia el efecto en el rendimiento de un parámetro clave de esta política, el tamaño de bloque. Así mismo, se estudia el impacto del sistema de almacenamiento y acceso a los datos de estructura tipo array cuando se utiliza esta técnica de partición. El estudio se realizará en el contexto del desarrollo de Hitmap, una biblioteca de funciones para la programación de aplicaciones paralelas basadas en el manejo de arrays jerárquicos.

Abstract

Data-parallelism is a parallel computing paradigm where the same tasks are executed for all the data in a given data-structure. Applying a proper partition of the data across the processing elements, the tasks are also properly distributed and the program obtains a better speed-up. Thus, the data-partition policy chosen has an important impact on the final performace of the application, and should be tailored for the specific kind of problem to be parallelized.

This work is focused on the study of a complex data-partition policy: The block-cyclic partition. We study the performance effect due to the choice of a key parameter of the policy, the block-size. We also study the impact of the storage layout and data-access mechanisms of array data, when using this policy. The study is performed in the context of the development of Hitmap. Hitmap is a functions library for parallel programming based on hierarchical array management.

Índice general

Capítulo 1. Introducción....1
1.1 Motivación....1
1.2 Objetivos....2
1.3 Estructura....2

Capítulo 2. Gestión del Proyecto....5
2.1 Metodología de desarrollo software....5
2.2 Planificación....5
2.2.1 Planificación temporal....5
2.2.2 Análisis de riesgos....8
2.2.3 Seguimiento....10
2.2.4 Presupuesto....11

Capítulo 3. Preliminares....13
3.1 Paralelismo de datos....13
3.2 Tipos de particiones de datos....15
3.3 Hitmap....16
3.3.1 Estructura de Hitmap....17
3.3.2 Modo implementación partición en bloques en Hitmap....19
3.3.2.1 Ejemplo de realización de una partición en bloques cíclica en Hitmap....20
3.3.3 Modo de almacenamiento y acceso de datos en Hitmap....20
3.4 Caso de estudio: Resolución de sistema de ecuaciones con factorización LU....21
3.4.1 Factorización LU....21
3.4.1.1 Algoritmo de factorización LU....22
3.4.1.2 Ejemplo matemático factorización LU....22
3.4.2 Resolución de sistema de ecuaciones por substitución....23
3.4.2.1 Algoritmo de resolución sistema ecuaciones....24
3.4.2.2 Ejemplo matemático resolución sistema ecuaciones....24
3.4.3 Particionado de datos en LU....25
3.4.4 Partición bloque-cíclica en LU....28
3.5 Resumen del capítulo....29

Capítulo 4. Desarrollo de software....31
4.1 Propuesta....31
4.2 Diseño e implementación....32
4.2.1 Signatures de Hitmap ampliadas....32
4.2.1.1 Ejemplo Signatures ampliadas....33
4.2.2 Diseño de métodos de acceso....33
4.2.3 Interfaz....35
4.2.3.1 Acceso nuevos campos de las Signatures....35
4.2.3.2 Creación de un Tile....36
4.2.3.3 Reserva y liberación de memoria de un Tile....36
4.2.3.4 Traducir Signatures de bloques tradicionales con 3 campos a las nuevas con 7 campos....36
4.2.3.5 Selección de determinado bloque (x,y)....37
4.2.3.6 Subselección de un conjunto de elementos mediante un Shape....38
4.2.3.7 Métodos de acceso a los elementos....39
4.2.3.8 Otras modificaciones de la librería Hitmap necesarias....40
4.3 Ejemplo de uso de la interfaz propuesta....40

4.4 Pruebas....41
4.4.1 Pruebas sobre el Shape de la matriz de datos con las nuevas Signatures....41
4.4.2 Pruebas acceso a los datos....42
4.4.2.1 Acceso por coordenadas de Tile....42
4.4.2.2 Acceso por coordenadas de bloque....43
4.4.2.3 Accesos mediante extracción de subselecciones en bloques....43
4.5 Resumen del capítulo....44

Capítulo 5. Estudio experimental....45
5.1 Metodología de la experimentación....45
5.2 Experimentos....46
5.2.1 Acceso datos con diferente modo de almacenamiento....46
5.2.2 Diferentes versiones de implementación del código LU....48
5.2.3 Experimento del tamaño de bloque en LU....49
5.3 Resultados....49
5.3.1 Experimento suma de matrices con diferente modo de almacenamiento....49
5.3.1.1 Almacenamiento Filas-Columnas....49
5.3.1.2 Almacenamiento en Bloques....52
5.3.2 Experimento LU comprando rendimiento con diferentes implementaciones....53
5.3.3 Experimento tamaño de bloque en LU....55
5.4 Resumen del capítulo....59

Capítulo 6. Conclusiones....61
6.1 Conclusiones....61
6.2 Trabajo futuro....62

Anexo A. Contenido del CD-ROM....65

Índice de figuras

Figura 2.1: Planificación inicial del proyecto....6
Figura 2.2: Diagrama de Gantt de la planificación inicial....7
Figura 2.3: Seguimiento del proyecto....11
Figura 2.4: Estimación costes asociados al proyecto....12
Figura 3.1: Ejemplos de distribución de datos en bloques, cíclica y bloque-cíclica con tamaño de bloque 2 para N procesadores en un array unidimensional....15
Figura 3.2: Ejemplo partición de datos bloque-cíclica de una matriz 8x8 con bloques de 2x2, distribuida entre 4 procesadores....16
Figura 3.3. Estructura de Hitmap....17
Figura 3.4: Ejemplo subselección de un Tile....18
Figura 3.5: Ejemplo topología 2D con cuatro procesadores....18
Figura 3.6: HitBlockTile (HitTile_HitTile)....19
Figura 3.7. Ejemplo de realización de una partición en bloques cíclica en Hitmap....20
Figura 3.8: Ejemplo fórmula de acceso al elemento (2,3) en una matriz de 4x4....21
Figura 3.9. Factorización LU....22
Figura 3.10: Algoritmo de factorización LU....22
Figura 3.11: Algoritmo de resolución sistema de ecuaciones (Fase 1-sustitución progresiva)....24
Figura 3.12: Algoritmo de resolución sistema de ecuaciones (Fase 2-sustitución regresiva)....24
Figura 3.13: Etapas de la factorización LU trabajando por bloques....25
Figura 3.14: Partición por bloques entre 4x4 procesadores, con bloques de 4x4 elementos....26
Figura 3.15: Partición cíclica entre 2x2 procesadores....27
Figura 3.16: Partición bloque-cíclica entre 2x2 procesadores con tamaño de bloque 2x2....27
Figura 4.1: Ejemplo modos almacenamiento de datos para una matriz 4x4 dividida en bloques de 2x2....31
Figura 4.2: Implementación en Hitmap de HitSig....33
Figura 4.3: Ejemplo de Signatures ampliadas de una matriz 8x8 dividida en bloques de 2x2 distribuida de forma bloque-cíclica entre 4 procesadores....34
Figura 4.3: Ejemplo de Signatures ampliadas de matriz 8x8 dividida en bloques de 2x2 distribuida entre 4 procesadores....34
Figura 4.4: Ejemplo de fórmula de acceso al elemento(2,3) en una matriz 4x4 con tamaño de bloque de 2x2 35
Figura 4.5: Ejemplo cálculo puntero data para la selección del bloque(1,1) de una matriz 4x4 con tamaño de bloque 2x2....38
Figura 4.6: Ejemplo cálculo puntero data selección de bloques {(0,1), (0,3), (2,1), (2,3)} con bloques de 1x1 en una matriz 4x4....38
Figura 4.7: Ejemplo uso de la interfaz de partición en bloques propuesta....40
Figura 5.1: Suma de matrices recorriendo por filas-columnas....47
Figura 5.2: Suma de matrices recorriendo por coordenadas de bloque....47
Figura 5.3: Suma de matrices recorriendo por bloques, obteniendo un Tile de bloque....48
Figura 5.4: Suma de matrices 16384x16384 con almacenamiento por filas-columnas....50
Figura 5.5: Suma de matrices 16384x16384 con implementación tradicional de Hitmap mediante HitBlockTile(Tile de punteros a bloques)....51
Figura 5.6: Suma de matrices 16384x16384 por fila-columnas con nueva implementación....51
Figura 5.7: Suma de matrices 16384x16384 con almacenamiento por bloques....52
Figura 5.8: Rendimiento diferentes versiones de la aplicación LU para una matriz 4096x4096 con 64 procesos....53

Figura 5.9: Rendimiento resolución sistema LU para una matriz de 4096x4096 en Heracles.....54
Figura 5.10: Rendimiento resolución sistema LU para una matriz de 4096x4096 en Chimera.....55
Figura 5.11: Rendimiento resolución sistema LU para una matriz de 4096x4096 separando el tiempo de ejecución de cada algoritmo con 64 procesos en Heracles.....56
Figura 5.12: Rendimiento resolución sistema LU para una matriz de 4096x4096 separando el tiempo de ejecución de cada algoritmo con 24 procesos en Chimera.....56
Figura 5.13: Rendimiento algoritmo factorización LU para una matriz de 4096x4096 en Heracles con 32 procesos.....57
Figura 5.14: Rendimiento algoritmo de resolución del sistema de ecuaciones para una matriz de 4096x4096 en Heracles con 32 procesos.....57

Capítulo 1. Introducción

En este capítulo se proporciona una muy breve introducción del problema que se abarca en este proyecto, así como cuáles son los objetivos que se pretenden cumplir.

1.1 Motivación

La computación paralela es una forma de cómputo en la que varias instrucciones de un programa se ejecutan simultáneamente. Consiste en dividir un problema grande en trozos más pequeños para después ejecutarlos en distintos procesadores al mismo tiempo, ahorrando tiempo de ejecución.

La manera en que se dividen y se reparten las tareas del problema a resolver es una parte crítica que afecta directamente al rendimiento de la aplicación, ya que lo que se busca es mantener una carga de datos equilibrada entre los procesadores de forma que se maximice la paralelización. La política con la que se reparten las tareas debe de ser elegida en función del tipo de problema que se quiera paralelizar. En concreto, como se explicará más adelante, la partición de datos bloque-cíclica es la más apropiada para una importante clase de aplicaciones numéricas paralelas. En esta política de partición, el rendimiento final de la aplicación depende en gran parte de la selección de un parámetro básico de la política: el tamaño de bloque.

Por otra parte, el modo en que se almacenan y acceden a los datos en memoria en principio puede tener un impacto en el rendimiento de la aplicación, ya que el acceso consecutivo a posiciones de memoria contiguas es clave para la explotación adecuada de las memorias caché.

En el grupo de investigación Trasgo de la Universidad de Valladolid se trabaja en el desarrollo de una biblioteca de funciones para programación paralela denominada Hitmap [1]. Es una biblioteca focalizada en el reparto y asignación automática de tareas a realizar en arrays jerárquicos (selecciones anidadas y/o arrays de arrays). Hitmap reduce significativamente el esfuerzo de desarrollo de programas paralelos, empleando para ello una capa de abstracción que oculta detalles de programación, y a la vez, mantiene un buen nivel de rendimiento, similar al de implementaciones optimizadas manualmente.

En esta herramienta las políticas de partición se incluyen de forma modular y ortogonal al resto de desarrollo. El uso de parámetros adecuados para las políticas de partición, y el manejo eficiente del almacenamiento y acceso a los elementos de los arrays jerárquicos resulta clave para el rendimiento obtenido por las aplicaciones programadas en Hitmap.

Mientras que otras políticas de partición más simples han sido estudiadas y su uso optimizado en el contexto de Hitmap, el soporte para aplicaciones que utilizan la partición bloque-cíclica presenta posibles mejoras a investigar.

1.2 Objetivos

En este trabajo se plantea un estudio de la partición bloque-cíclica en el contexto del desarrollo de la biblioteca de funciones Hitmap. En concreto, se plantean los siguientes objetivos:

- Estudiar la librería Hitmap y aplicaciones que necesiten que necesiten particionado y almacenamiento bloque-cíclico.
- Diseñar e implementar en la librería Hitmap un método de almacenamiento de matrices en memoria basado en bloques, con métodos de acceso específicos a elementos de bloques, por índices de matriz, o extracción de bloques enteros.
- Realizar una experimentación para estudiar el impacto que tiene en el rendimiento de una aplicación el modo en que se acceden a los datos dependiendo de cómo estén almacenados en memoria.
- Comprobar ventajas e inconvenientes del nuevo método de almacenamiento basado en bloques sobre el método tradicional basado en filas-columnas.
- Realizar un estudio experimental utilizando una aplicación paralela de ejemplo determinar la relación entre el rendimiento y el tamaño de bloque, en función de las características de la plataforma de ejecución.

1.3 Estructura

La estructura de la memoria es la siguiente:

- En este primer capítulo se ha presentado una breve introducción de lo que se va a presentar en este proyecto, y de los objetivos que se pretenden cumplir.
- En el capítulo 2 se explica cómo se ha llevado la gestión del proyecto, presentando la estimación inicial de planificación, junto con un análisis de los riesgos que pueden dificultar el desarrollo. Se proporcionará un seguimiento del proyecto, para conocer cómo han ido el desarrollo del proyecto realmente frente a la estimación inicial. Por ultimo, se calculará el presupuesto necesario para llevar a cabo este Trabajo Fin de Grado..
- En el capítulo 3 se presentará en primer lugar qué es el paralelismo de datos y diferentes formas de realizar las particiones. A continuación, se realizará un resumen la biblioteca Hitmap empleada en este trabajo, la forma actual que emplea para realizar las particiones de datos en bloques, cómo se almacenan los datos en memoria, y cómo se realizan los accesos a los datos. Finalmente, se explicará la aplicación que se va a emplear como caso de estudio representativo de aplicación paralela con particiones de datos bloque-cíclicas.

- En el capítulo 4 se desarrollará la propuesta de este Trabajo Fin de Grado, basada en el desarrollo de un método de almacenamiento de datos contiguos por bloques.
- En el capítulo 5, en primer lugar, se lleva a cabo un estudio experimental para comprobar cómo afecta el modo en que se recorren los datos, según estén almacenados estos en memoria. En segundo lugar, también se realizará un estudio experimental para observar el rendimiento de la propuesta de almacenamiento frente a la tradicional. Y por último se realizará un estudio experimental para intentar determinar de la influencia del tamaño del bloque en el rendimiento de la aplicación para un caso de estudio concreto.
- Finalmente, en el capítulo 6 se presentarán los objetivos cumplidos, las conclusiones experimentales obtenidas y las posibles vías de continuación de este Trabajo Fin de Grado.

Capítulo 2. Gestión del Proyecto

En este capítulo se describen detalles relacionados con la gestión de tiempo, recursos y riesgos a lo largo del desarrollo del trabajo.

2.1 Metodología de desarrollo software

En un proyecto de investigación, principalmente orientado al estudio de optimizaciones y mecanismos de implementación de aplicaciones, el desarrollo de software está supeditado a continuas pruebas y cambios.

La parte de desarrollo software de este proyecto se ha realizado siguiendo un modelo de desarrollo evolutivo[11], que se basa en la idea de desarrollar una implementación inicial e ir refinándola a través de diferentes versiones hasta desarrollar un sistema software que satisfaga todos los requerimientos.

El desarrollo evolutivo permite que los requerimientos no estén totalmente especificados para comenzar con el desarrollo del software, lo que permite adaptarse a los cambios en los requisitos que es lo que se busca en este proyecto, al tener una naturaleza de investigación, dónde los resultados finales no se conocen y puede que no ajusten a las previsiones realizadas inicialmente.

El desarrollo evolutivo tiene el inconveniente de que al no necesitar inicialmente de los requerimientos, es probable que el software desarrollado pueda estar mal estructurado o sea difícil de mantener. Para evitar en este principal inconveniente que acompaña al modelo, se intentará seguir una serie pautas comentando el código desarrollado con detalle.

2.2 Planificación

2.2.1 Planificación temporal

En esta sección se especifica la planificación realizada al comienzo de este proyecto. Las principales tareas en las que se divide la planificación son:

- Aprendizaje y estudio: Primera fase dónde se adquieren los conocimientos necesarios para poder empezar a desarrollar este Trabajo Fin de Grado.
- Análisis, diseño e implementación de la solución: Esta programado para que se realice en varias iteraciones, dónde se vayan obteniendo cada vez versiones con mayor funcionalidad.

Id	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	Inicio	0 días	lun 06/10/14	lun 06/10/14	
2	**Aprendizaje conceptos teóricos**	**15 días**	**lun 06/10/14**	**vie 24/10/14**	**1**
3	Estudio de diferentes tipos de particiones de datos	5 días	lun 06/10/14	vie 10/10/14	
4	Estudio de partición de bloque cíclica	3 días	lun 13/10/14	mié 15/10/14	3
5	Estudio de partición de bloque cíclica para el caso de estudio LU	10 días	lun 13/10/14	vie 24/10/14	3
6	**Estudio de Hitmap**	**35 días**	**lun 27/10/14**	**vie 12/12/14**	**5**
7	Estudio conceptual de Hitmap	15 días	lun 27/10/14	vie 14/11/14	
8	Estudio partición en bloques en Hitmap	5 días	lun 17/11/14	vie 21/11/14	7
9	Estudio del modo de almacenamiento de datos en Hitmap	5 días	lun 17/11/14	vie 21/11/14	7
10	Estudio de versión paralela del código LU implementada en Hitmap	20 días	lun 17/11/14	vie 12/12/14	7
11	**Diseño e Implementación de la solución**	**64 días**	**lun 15/12/14**	**mar 07/04/15**	**10**
12	**Iteración 1**	**32 días**	**lun 15/12/14**	**vie 20/02/15**	
13	**Diseño de la solución**	**7 días**	**lun 15/12/14**	**mar 23/12/14**	
14	Análisis y diseño de la solución	7 días	lun 15/12/14	mar 23/12/14	
15	**Implementación de la solución**	**20 días**	**lun 05/01/15**	**vie 13/02/15**	**13**
16	Implementación parcial (Realización subselecciones por bloques)	20 días	lun 05/01/15	vie 13/02/15	
17	**Pruebas subselecciones**	5 días	lun 16/02/15	vie 20/02/15	15
18	**Iteración 2**	**32 días**	**lun 23/02/15**	**mar 07/04/15**	**12**
19	**Diseño de la solución**	**5 días**	**lun 23/02/15**	**vie 27/02/15**	
20	Revisar análisis y diseño de la solución	5 días	lun 23/02/15	vie 27/02/15	
21	**Implementación de la solución**	**22 días**	**lun 02/03/15**	**mar 31/03/15**	**19**
22	Implementación completa (100%)	22 días	lun 02/03/15	mar 31/03/15	
23	**Pruebas acceso elementos**	5 días	mié 01/04/15	mar 07/04/15	21
24	**Implementación de códigos de casos de estudio**	24 días	mié 08/04/15	lun 11/05/15	11
25	**Pruebas códigos de casos de estudio**	12 días	mar 12/05/15	mié 27/05/15	24
26	**Experimentación**	**30 días**	**jue 28/05/15**	**mié 08/07/15**	**25**
27	Estudio del rendimiento aplicaciones paralelas particionadas en bloques según el modo de almacenamiento de datos	10 días	jue 28/05/15	mié 10/06/15	
28	Comparación rendimientos implementación propuesta en el TFG del LU vs implementación tradicional en Hitmap	10 días	jue 11/06/15	mié 24/06/15	27
29	Influencia de la elección del tamaño del bloque en aplicaciones con partición bloque-cíclica	10 días	jue 25/06/15	mié 08/07/15	28
30	**Documentación**	197 días	lun 06/10/14	vie 31/07/15	1
31	Fin	0 días	vie 31/07/15	vie 31/07/15	30

Figura 2.1: Planificación inicial del proyecto

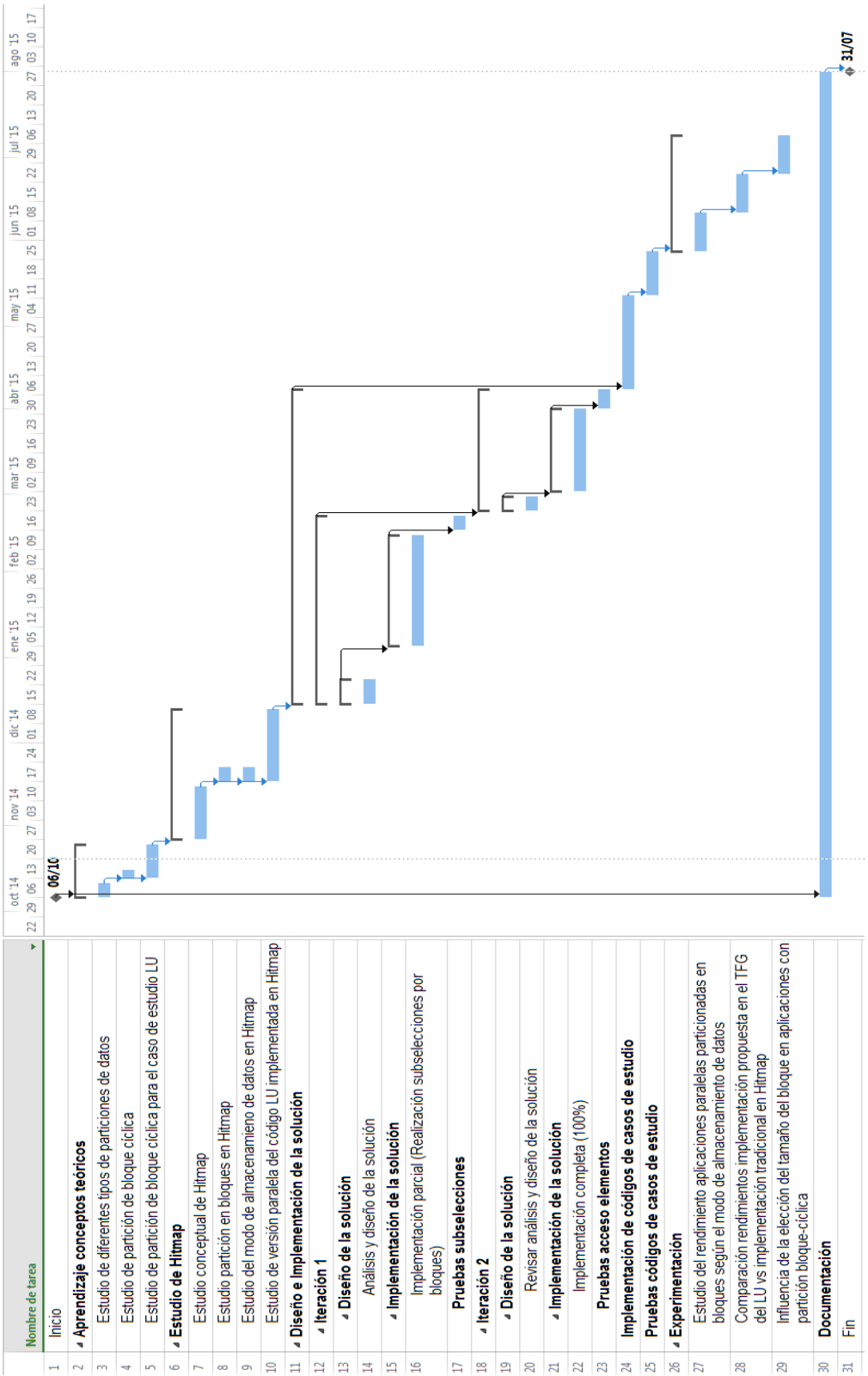


Figura 2.2: Diagrama de Gantt de la planificación inicial

- Implementación y pruebas de los códigos de los casos de estudio: Aquí se desarrollan o se adaptan los códigos que van a ser necesarios para la posterior fase de experimentación.
- Experimentación: Se realizan los experimentos tomando las mediadas oportunas y razonando las conclusiones que se pueden extraer.
- Documentación: Etapa que se extiende durante toda el tiempo del proyecto dónde se va generando de manera progresiva la presente memoria.

2.2.2 Análisis de riesgos

En este apartado se detallan los riesgos que se pueden producir durante el desarrollo del proyecto, se han identificado los riesgos potenciales, evaluando su probabilidad de ocurrencia y su posible impacto, seguido de la creación de planes de acción que permitan responder de forma eficaz y controlada ante cualquiera de estas posibles situaciones en caso de ocurrencia. Por tanto, los riesgos serán analizados en estos aspectos:

- Descripción: Breve descripción del riesgo.
- Probabilidad: Indica la probabilidad de que el riesgo ocurra. Se emplea la siguiente clasificación: Muy Alta (>70%), Alta (60% - 70%), Media (30% - 60%), Baja (<30%).
- Impacto: Se clasifican en 4 niveles dependiendo de cómo afecte al proyecto, en caso de que el riesgo de produjese:
  - Muy Alto: Objetivos críticos del proyecto con alto grado de impacto o de no cumplimiento.
  - Alto: Objetivos críticos del proyecto están amenazados.
  - Medio: Algunos objetivos del proyecto pueden verse afectados.
  - Bajo: Los objetivos del proyecto no serán afectados. Fácilmente remediable.
- Plan de mitigación: Medidas a tomar en el proyecto para evitar la aparición del riesgo o minimizar su futuro impacto.
- Plan de contingencia: Medidas a tomar en el proyecto en caso de que un riesgo haya ocurrido.

- **Riesgo-01: Planificación poco realista**
  - Descripción: Se realiza una planificación inicial incorrecta
  - Probabilidad: Muy alto
  - Impacto: Alto
  - Plan de mitigación: Realizar una revisión continua y periódica del estado del proyecto para detectar los posibles retrasos y poder tomar medias al instante.
  - Plan de contingencia: Realizar una nueva planificación de las tareas.

- **Riesgo-02: Pérdida de información del proyecto**
  - Descripción: Pérdida de versiones o de algún artefacto obtenido durante el desarrollo.
  - Probabilidad: Medio

- Impacto: Alto
- Plan de mitigación: Realizar varias copias de seguridad de todos los elementos del proyecto, en diferentes lugares de almacenamiento.
- Plan de contingencia: Volver a realizar de nuevo las partes del proyecto que han sido perdidas.

- **Riesgo-03: Inexperiencia en proyectos de este tipo**
  - Descripción: El desarrollador no tiene experiencia en desarrollo de proyectos de este tipo.
  - Probabilidad: Medio
  - Impacto: Medio
  - Plan de mitigación: Replanificar el proyecto incluyendo un tiempo de aprendizaje.
  - Plan de contingencia: Retrasar el desarrollo del proyecto hasta adquirir ciertos conocimientos.

- **Riesgo-04: Desconocimiento de la librería y/o del lenguaje de desarrollo**
  - Descripción: El desarrollador desconoce el funcionamiento interno de la librería en uso y/o del lenguaje de programación.
  - Probabilidad: Alto
  - Impacto: Medio
  - Plan de mitigación: Replanificar el proyecto incluyendo un tiempo de aprendizaje.
  - Plan de contingencia: Retrasar la implementación hasta que se dominen los conceptos necesarios.

- **Riesgo-05: Restricciones de recursos informáticos**
  - Descripción: Alguno de los equipos informáticos no está disponible para la realización del proyecto.
  - Probabilidad: Bajo
  - Impacto: Medio
  - Plan de mitigación: Tener varios frentes de trabajo abiertos para poder ir avanzando en caso de fallo en algún equipo concreto.
  - Plan de contingencia: Contactar con el responsable de los equipos informáticos, reparar y/o adquirir nuevos equipos.

- **Riesgo-05: Plan de pruebas incompleto**
  - Descripción: La batería de pruebas realizada no ha contemplado casos críticos o situaciones de posible fallo.
  - Probabilidad: Medio

- Impacto: Medio
- Plan de mitigación: Utilizar métodos y técnicas formales para realización de las pruebas.
- Plan de contingencia: Rehacer plan de pruebas y volver a realizarlas contemplado los nuevos casos.

2.2.3 Seguimiento

Id	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	Inicio	0 días	lun 06/10/14	lun 06/10/14	
2	**Aprendizaje conceptos teóricos**	**15 días**	**lun 06/10/14**	**vie 24/10/14**	**1**
3	Estudio de diferentes tipos de particiones de datos	5 días	lun 06/10/14	vie 10/10/14	
4	Estudio de partición de bloque cíclica	3 días	lun 13/10/14	mié 15/10/14	3
5	Estudio de partición de bloque cíclica para el caso de estudio LU	10 días	lun 13/10/14	vie 24/10/14	3
6	**Estudio de Hitmap**	**35 días**	**lun 27/10/14**	**vie 12/12/14**	**5**
7	Estudio conceptual de Hitmap	15 días	lun 27/10/14	vie 14/11/14	
8	Estudio partición en bloques en Hitmap	5 días	lun 17/11/14	vie 21/11/14	7
9	Estudio del modo de almacenamiento de datos en Hitmap	5 días	lun 17/11/14	vie 21/11/14	7
10	Estudio de versión paralela del código LU implementada en Hitmap	20 días	lun 17/11/14	vie 12/12/14	7
11	**Diseño e Implementación de la solución**	**70 días**	**lun 15/12/14**	**mié 15/04/15**	**10**
12	**Iteración 1**	**37 días**	**lun 15/12/14**	**vie 27/02/15**	
13	**Diseño de la solución**	**7 días**	**lun 15/12/14**	**mar 23/12/14**	
14	Análisis y diseño de la solución	7 días	lun 15/12/14	mar 23/12/14	
15	**Implementación de la solución**	**24 días**	**lun 05/01/15**	**jue 19/02/15**	**13**
16	Implementación parcial (Realización subselecciones por bloques)	24 días	lun 05/01/15	jue 19/02/15	
17	**Pruebas subselecciones**	6 días	vie 20/02/15	vie 27/02/15	15
18	**Iteración 2**	**33 días**	**lun 02/03/15**	**mié 15/04/15**	**12**
19	**Diseño de la solución**	**4 días**	**lun 02/03/15**	**jue 05/03/15**	
20	Revisar análisis y diseño de la solución	4 días	lun 02/03/15	jue 05/03/15	
21	**Implementación de la solución**	**24 días**	**vie 06/03/15**	**mié 08/04/15**	**19**
22	Implementación completa (100%)	24 días	vie 06/03/15	mié 08/04/15	
23	**Pruebas acceso elementos**	5 días	jue 09/04/15	mié 15/04/15	21

Id	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
24	**Experimento 1**	**17 días**	**jue 16/04/15**	**vie 08/05/15**	**11**
25	Diseño experimento recorrer matrices con diferente modo de almacenamiento	1 día	jue 16/04/15	jue 16/04/15	
26	Implementación de códigos suma de matrices	8 días	vie 17/04/15	mar 28/04/15	25
27	Pruebas códigos suma de matrices	3 días	mié 29/04/15	vie 01/05/15	26
28	Estudio experimental del rendimiento según el modo de almacenamiento de datos	5 días	lun 04/05/15	vie 08/05/15	27
29	**Experimento 2**	**39 días**	**lun 11/05/15**	**jue 02/07/15**	**24**
30	Diseño experiemento LU con diferentes versiones	1 día	lun 11/05/15	lun 11/05/15	
31	Re implementación de código LU	24 días	mar 12/05/15	vie 12/06/15	30
32	Pruebas código LU	10 días	lun 15/06/15	vie 26/06/15	31
33	Experimento comparando rendimientos implementación propuesta en el TFG del LU vs implementación tradicional en Hitmap	**4 días**	**lun 29/06/15**	**jue 02/07/15**	**32**
34	**Experimento 3**	**9 días**	**vie 03/07/15**	**mié 15/07/15**	**29**
35	Diseño experimento	1 día	vie 03/07/15	vie 03/07/15	
36	Experimento sobre la Influencia de la elección del tamaño del bloque en aplicaciones con partición bloque-cíclica	**8 días**	**lun 06/07/15**	**mié 15/07/15**	**35**
37	**Documentación**	219 días	lun 06/10/14	mar 01/09/15	1
38	Fin	0 días	mar 01/09/15	mar 01/09/15	37

Figura 2.3: Seguimiento del proyecto

En el la figura de arriba se muestra el tiempo dedicado realmente a cada tarea del proyecto. Se ve que respecto a la planificación original se han producido retrasos tanto en la parte de diseño e implementación como en el desarrollo de los experimentos. Estos retrasos se traducen en un mes más de lo inicialmente previsto.

Al final, en vez de en primer lugar realizar la implementación de todos los códigos necesarios antes de entrar a realizar los experimentos, se han ido implementando los códigos de los experimentos según se iban necesitando a la vez que se iba realizando cada experimento.

2.2.4 Presupuesto

En la siguiente tabla se muestra una estimación de los costes asociados al desarrollo de este proyecto. Se tienen en cuenta tanto los costes de las herramientas utilizadas, como el coste asociado a la amortización de las máquinas en uso y por supuesto, el coste humano.

Concepto	Coste Total (€)
Herramientas de desarrollo	
GCC v4.8.3	0 €
GDB v7.6.1	0 €
VIM - Vi IMproved v7.4	0 €
MPICH v 3.1.3	0 €
Slurm v2.0	0 €
Microsoft Project 2013	0 €
LibreOffice v4.4.2.2	0 €
Máquinas	
Portátil Toshiba A500-13Z (26€/mes) (duración estimada de 4 años)	260 €
Servidores (10€/hora por core usado)	360 €
Trabajo humano	
Coste Ingeniero (1400 €/mes)	14000 €
TOTAL	**14620 €**

Figura 2.4: Estimación costes asociados al proyecto

Para estimar el coste de uso de los servidores, se ha tenido en cuenta el método con el que se calculan estos en los centros de supercomputación. Allí te cobran por tiempo de ejecución, y tienes los compiladores y otras herramientas de pago instaladas, como es nuestro caso que requerimos del uso del compilador de Intel, que tiene un coste de licencia por usuario de 699 $, pero al estar instalado en los servidores, el coste de uso de licencia compartida va incluido en el coste por horas de los servidores. Se estiman 2 horas de cómputo con una media de 18 cores, por lo que salen 360€.

Capítulo 3. Preliminares

En este capítulo se presentan conceptos preliminares sobre el tema de estudio del trabajo, una descripción de algunos detalles relevantes relacionados con la herramienta Hitmap, y un caso de estudio no trivial que se utilizará durante el desarrollo del trabajo.

3.1 Paralelismo de datos

La cantidad de computación necesaria para ejecutar un algoritmo va asociada a la cantidad de datos. Por tanto, cuanto mayor sea el tamaño del problema, mayor tiempo se requerirá para resolverlo.

El principal objetivo cuando se diseña un algoritmo paralelo y se escribe un programa paralelo es conseguir mejores prestaciones que una versión secuencial. El primer paso para diseñar un algoritmo paralelo es descomponer el problema en problemas más pequeños. Estos subproblemas se reparten entre los procesadores para ser resueltos de forma simultánea.

El paradigma de paralelismo de datos [12] consiste en ejecutar un misma secuencia de instrucciones sobre diferentes datos de una estructura. Para la aplicación práctica de este paradigma, es necesaria una descomposición de dominio, donde los datos son divididos en partes y las partes son asignadas a diferentes procesadores. Cada procesador trabaja sólo con las partes de datos que se le asignan, y éstos pueden necesitar comunicarse para intercambiar datos en determinados puntos de la ejecución.

Para conseguir explotar el paralelismo adecuadamente y minimizar el tiempo de ejecución de un programa basado en paralelismo de datos hay que tener en cuenta varios aspectos:

- **Equilibrado de la carga**

Equilibrar la carga consiste en dividir equitativamente el trabajo entre los procesadores disponibles. Al repartir los datos, se reparte tiempo de computación. Cada problema concreto se beneficia más utilizando un tipo concreto de política de partición para equilibrar la carga ya que hay que tener en cuenta la forma y momento en que se usan los datos dentro del contexto del algoritmo completo y las restricciones asociadas a las dependencias de operaciones a realizar, que pueden necesitar datos de otros procesadores.

- **Minimización de las comunicaciones**

El tiempo total para completar la ejecución es la preocupación principal de la programación paralela, y en él intervienen tres componentes:

  - Tiempo de computación: El tiempo de computación es el tiempo empleado en realizar cálculos sobre los datos. Idealmente, esperaríamos que al disponer de P procesadores trabajando sobre un problema pudiésemos terminar el trabajo en un tiempo igual al tiempo empleado en un procesador secuencial entre el número de procesadores. Este sería el caso ideal, en el caso que se pudiera dividir los datos de un programa totalmente entre los procesos y estos no tuvieran que comunicarse, de forma que todo el tiempo de todos los procesadores pudiera ser empleado en realizar computación.

  - Tiempo ocioso: El tiempo ocioso es el tiempo que un procesador gasta esperando datos de otros procesadores, o cuando una determinada parte del problema no puede ser dividida entre los procesadores disponibles y sólo puede ser realizada por un procesador o un subconjunto de ellos. Durante este tiempo algún procesador no realiza trabajo útil.

  - Tiempo de comunicación: El tiempo de comunicación es el tiempo empleado en enviar y recibir datos mediante algún mecanismo de comunicación. En el caso de modelos de memoria compartida el tiempo de comunicación depende de las latencias de memoria. En el caso de los modelos de paso de mensajes el coste de las comunicaciones en el tiempo total de ejecución puede ser expresado en términos de latencia y ancho de banda, siendo la latencia el tiempo necesario para preparar el envío de cada mensaje y el ancho de banda la velocidad de transmisión por dato de los que forman el mensaje. Los programas secuenciales no emplean comunicaciones entre procesos, por lo que debe minimizarse siempre el tiempo de comunicaciones que supone en todo caso un sobrecoste.

- **Solapamiento de computación y comunicaciones**

Siempre que sea posible, deben evitarse los tiempos ociosos, procurando que los procesadores realicen cálculos útiles mientras esperan datos necesarios provenientes de otros procesadores. Las comunicaciones son necesarias cuando se parte el dominio de forma que los datos necesarios para realizar la computación de un nuevo valor para un elemento del dominio no están en la misma parte. Los datos de los que depende el computo de un nuevo valor para un dato concreto determinan lo que se denomina de forma genérica "dependencias de datos". Estas dependencias son específicas de cada problema. Por tanto, las políticas de partición de datos deben buscar simultáneamente el equilibrio de carga y la minimización de las comunicaciones. Estas tareas a veces están contrapuestas, o el análisis de las dependencias de datos es muy compleja. Sin embargo, muchos problemas comparten características similares en cuanto a distribución de carga o dependencias de datos, lo que permite definir políticas de partición que sirven para grandes clases de problemas.

3.2 Tipos de particiones de datos

Como se acaba de comentar, lo que se siempre se intenta buscar es una relación adecuada entre equilibrio de carga y tiempo de comunicación, evitando que los procesadores queden inactivos durante el menor tiempo posible. Los tres tipos de políticas de partición más habituales y que se aplican a una gran cantidad de problemas son:

- Partición en bloques: Consiste en dividir los datos en bloques del mismo tamaño y repartirlos entre los procesadores. Así, para dividir N datos entre P procesadores, cada proceso recibiría un bloque con N/P datos.
- Partición cíclica: Consiste en repartir los datos cíclicamente entre los procesadores. De forma que cada dato *i* es asignado al proceso *i* mod P.
- Partición de bloque-cíclica: Consiste en una combinación de las dos técnicas de partición de datos anteriores, de manera que los datos se dividen en bloques de un cierto tamaño(tb), no necesariamente N/P y luego los bloques son distribuidos cíclicamente entre los procesadores de forma que el dato i es asignado al procesador (i div tb) mod P.

En la figura 3.1 se muestran ejemplos de los tres tipos clásicos de partición para un array unidimensional.

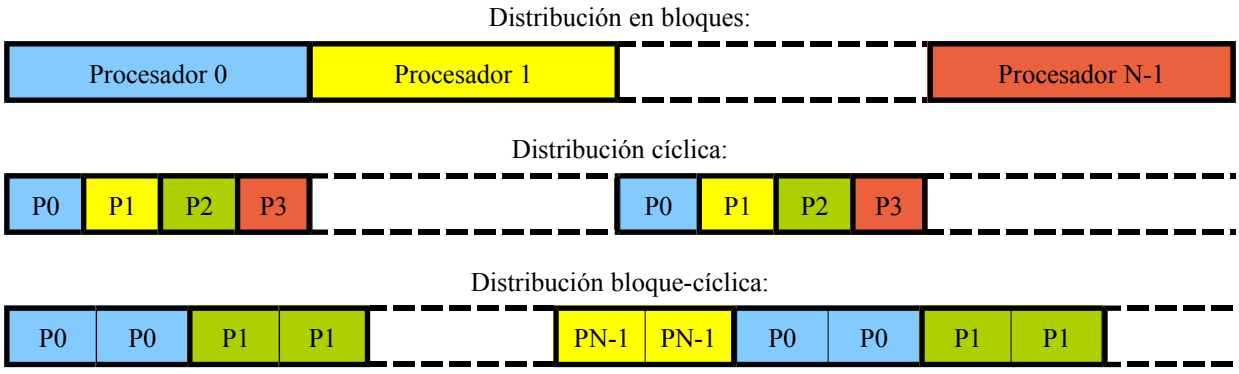


Figura 3.1: Ejemplos de distribución de datos en bloques, cíclica y bloque-cíclica con tamaño de bloque 2 para N procesadores en un array unidimensional

A continuación en la figura 3.2 se muestra con otro ejemplo cómo una matriz es particionada siguiendo una política de partición bloque-cíclica, dónde los datos se dividen en bloques y se reparten cíclicamente entre los procesos. En concreto, se muestra cómo se dividiría una matriz de 8x8 elementos en bloques de 2x2 entre un grid 2D de 4 procesadores. En primer lugar se realiza una partición en bloques de 2x2 y luego se asignan de forma cíclica a los procesadores, por lo que cada procesador tendría un trozo de 4x4 elementos, es decir, de 2x2 bloques.

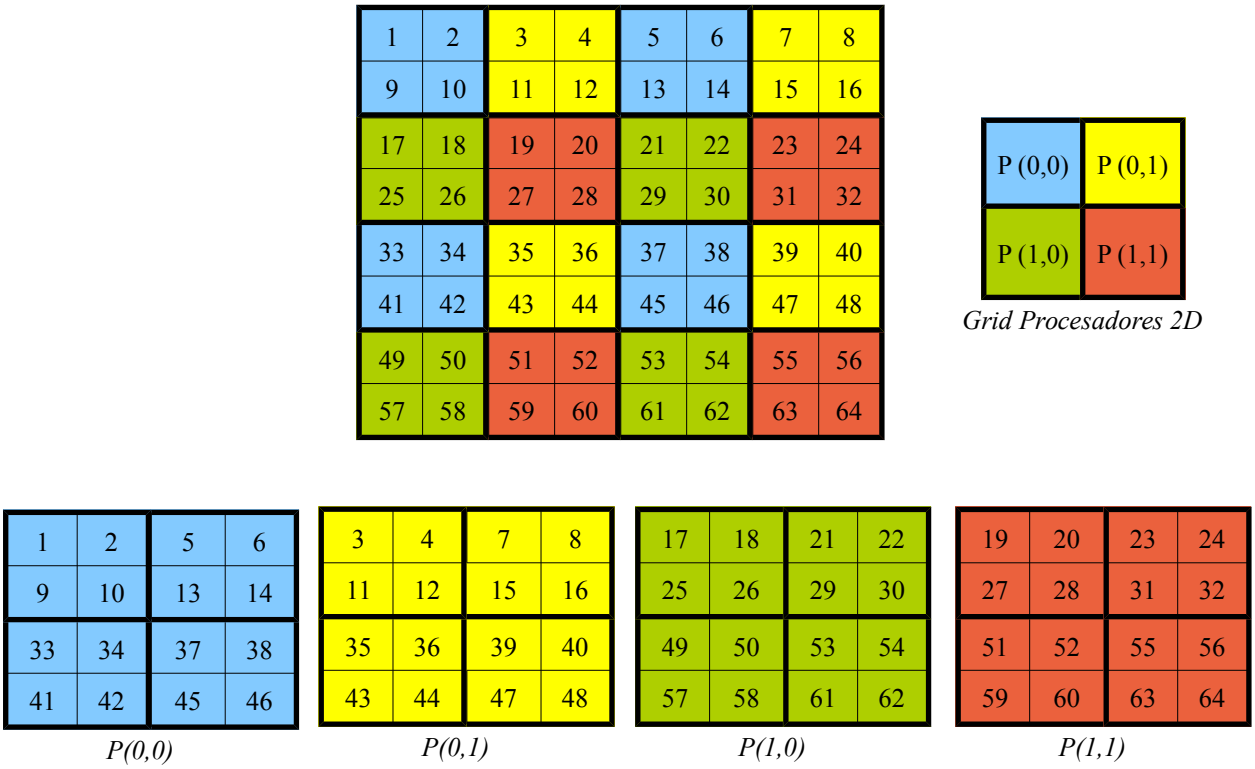


Figura 3.2: Ejemplo partición de datos bloque-cíclica de una matriz 8x8 con bloques de 2x2, distribuida entre 4 procesadores

3.3 Hitmap

Hitmap [1] es una biblioteca de funciones para la programación paralela, desarrollada por el grupo Trasgo de la Universidad de Valladolid. Las funcionalidades incluídas permiten la creación, manipulación, distribución y comunicación de estructuras de datos. Está diseñado siguiendo un modelo de programación distribuido SPMD (Single Program, Multiple Data), es decir, múltiples procesadores autónomos ejecutan simultáneamente el mismo programa en puntos independientes.

El objetivo de Hitmap es proporcionar una capa de abstracción para desarrollar código paralelo. Utiliza abstracciones denominadas Tiles para la declaración de estructuras o subestructuras de datos. Realiza de forma automática la partición, distribución y comunicación de dichos Tiles, reduciendo significativamente el esfuerzo de desarrollo y mantenimiento de programas paralelos, y a la vez, manteniendo un buen nivel de rendimiento, similar al de implementaciones optimizadas manualmente.

Para las comunicaciones utiliza internamente el estándar MPI (Message Passing Interface) [10], basado en el modelo de paso de mensajes.

3.3.1 Estructura de Hitmap

Hitmap esta diseñado con una aproximación de orientación a objetos, aunque está programado en lenguaje C. Las clases están implementadas como estructuras de C con funciones asociadas. En la Figura 2.1 se puede ver dicha estructura[1].

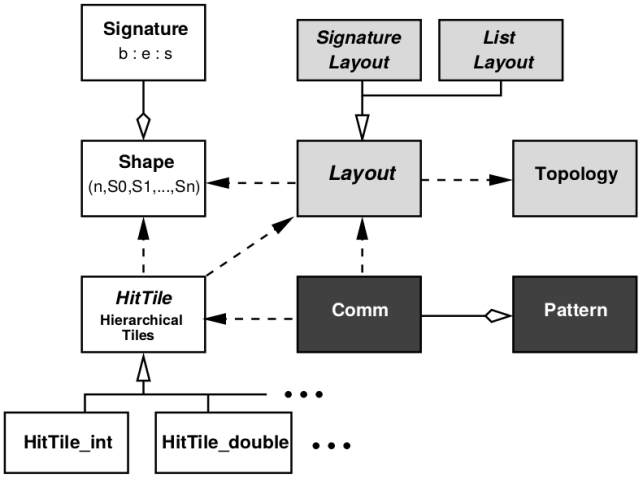


Figura 3.3. Estructura de Hitmap

Las clases Signature, Shape y Tile definen los tres tipos de datos que utiliza Hitmap para definir los dominios de datos, y las estructuras que contienen sus valores:

- Signature(HitSig): Es una tupla de tres elementos enteros (begin, end, stride), que representa un subespacio de índices de un array en una dimensión. La cardinalidad de un Signature queda definida entonces como: ((end-begin)/stride)+1. Por ejemplo, un Signature definido como (0:7:1) representa un array unidimensional de 8 elementos contiguos.

- Shape(HitShape): Es una n-tupla de Signatures representando un subespacio de índices de un array en un dominio multidimensional. Es decir, un Shape es un conjunto de Signatures. Por ejemplo, un Shape de dos dimensiones puede definirse como {(0:4:1),(0:4:1)} representando un dominio de 5x5 elementos contiguos.

- Tile(HitTile): Es un array n-dimensional. Su dominio está definido por un Shape y sus elementos tienen un tipo definido. Se pueden realizar subselecciones jerárquicas de un Tile usando la información de las Signatures para localizar y acceder a los datos.
  Por ejemplo: Se declara un Tile de 4x4 elementos y se realiza una subselección mediante un Shape de elementos de 0 a 2 con salto de dos en dos, es decir, se seleccionarían los elementos correspondientes a las coordenadas {(0,0), (0,2), (2,0), (2,2)} como se pude ver en la Figura 2.2.

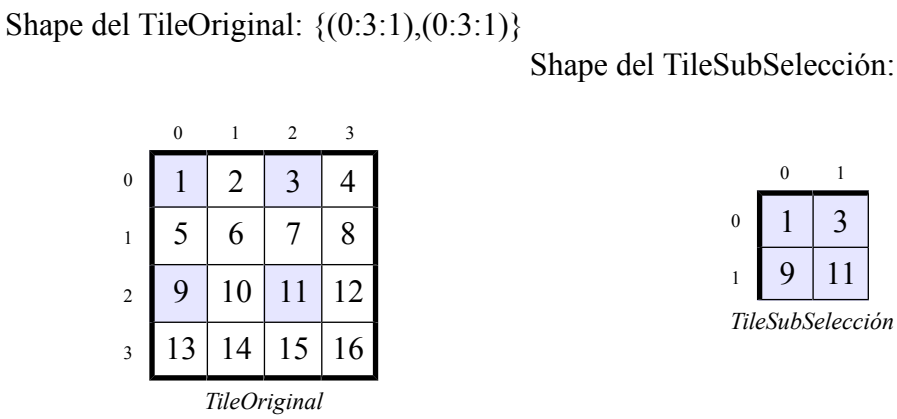


Figura 3.4: Ejemplo subselección de un Tile

Además, un Tile puede ser definido con o sin reservar memoria, lo que permite definir previamente la estructura de un Tile sin necesidad de que se reserve memoria para este, y luego que cada procesador reserve sólo en memoria la subselección del Tile que va a utilizar.

Las clases abstractas Topology y Layout encapsulan las funciones relacionadas con la partición y distribución de datos. Estas funciones están implementadas con un sistema plug-in. De esta forma permite seleccionar una topología concreta por su nombre, al invocar el método constructor; y así también facilitar al programador el desarrollo e integración de nuevos plug-ins.

- Topology: Clase abstracta con funciones que permiten crear topologías virtuales, ocultando los detalles de la topología física. Dependiendo de la topología elegida se definen qué procesadores están activos en tiempo de ejecución, y se marcan como inactivos aquellos que no se vayan a utilizar, de forma que sean transparentes para el programador.

Figura 3.5: Ejemplo topología 2D con cuatro procesadores

- Layout: Clase abstracta con funciones que automáticamente distribuyen un dominio de datos, seleccionado mediante un Shape, entre los procesadores activos de una topología virtual creada previamente. La política de partición aplicada depende del tipo de Layout, escogido por su nombre de plugin. El objeto layout además almacena las relaciones de vecindad de unos procesos con otros, para luego poder utilizarlas en las comunicaciones si es necesario seleccionando vecinos de forma abstracta.

Por último, la clases Comunication y Pattern contiene la información y métodos para permitir la comunicación de Tiles entre los procesos. Utilizan internamente MPI. Proporciona diferentes tipos de comunicación como: comunicaciones punto a punto, intercambios entre vecinos, comunicaciones colectivas, etc.

3.3.2 Modo implementación partición en bloques en Hitmap

Como ya se ha dicho, Hitmap emplea objetos HitTile para representar arrays multidimensionales o subselecciones de arrays. HitTile está definido en Hitmap como un tipo abstracto que se especializa al elegir el tipo de datos de los elementos contenidos en él, así por ejemplo, un Tile de elementos de tipo double se definiría como HitTile_double.

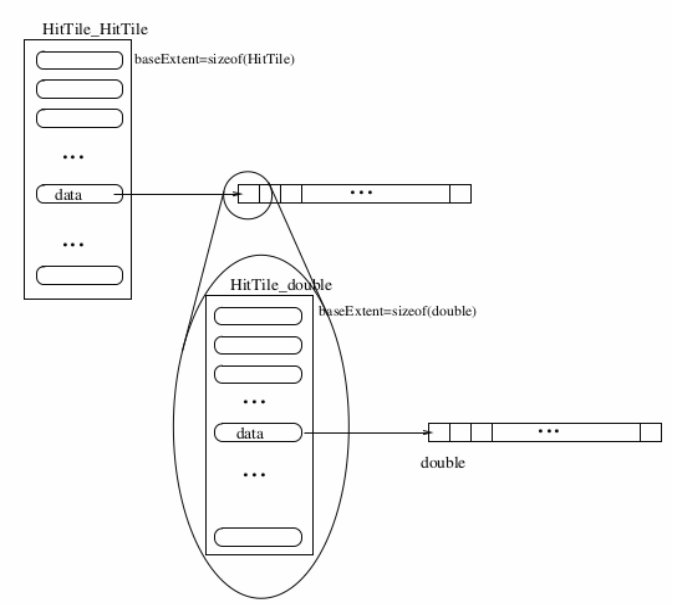


Figura 3.6: HitBlockTile (HitTile_HitTile)

Para implementar una distribución que reparte bloques de un dominio, en Hitmap actualmente se emplea un Tile con dos niveles jerárquicos. En el más profundo los datos se agrupan en bloques. En el superior, el dominio de los bloques se reparten automáticamente con alguna política de la clase Layout. Un Tile con dos niveles jerárquicos consiste en un objeto HitTile_HitTile[2] (para el que se creó un alias: HitBlockTile), es decir, un HitTile que contienen punteros de memoria que apuntan a su vez a otro HitTile que contiene la información para localizar los elementos. Gráficamente la representación de esto puede verse en la Figura 3.6

3.3.2.1 Ejemplo de realización de una partición en bloques cíclica en Hitmap

```
1.  HitTopology  topo = hit_topology(plug_topArray2DComplete);
2.  HitTile_double matrix;
3.  hit_tileDomain( &matrix, double, 2, n, n );
4.  int blockSizes[2] = { blockSize, blockSize };
5.  HitBlockTile matrixTiles;
6.  hit_blockTileNew( &matrixTiles, &matrix, blockSizes );
7.  HitLayout matrixLayout = hit_layout(plug_layCyclic,topo, hit_tileShape( matrixTiles ), HIT_LAYOUT_NODIM );
8.  HitBlockTile matrixSelection;
9.  hit_blockTileSelect( &matrixSelection, &matrixTiles, hit_layShape( matrixLayout ) );
10. hit_blockTileAlloc( &matrixSelection, double );
11. HitTile_double block;
12. block = hit_blockTileElemAtNS( matrixSelection, double, 2, 0, 0 );
```

Figura 3.7. Ejemplo de realización de una partición en bloques cíclica en Hitmap

En la figura 2.3. se puede ver el proceso de creación de una matriz de elementos de tipo double, su división en bloques, distribución entre los procesadores, y obtención de un determinado bloque. A continuación se describe el proceso.

En la línea 1 se crea topología virtual 2D de procesadores, usando un plug-in de Hitmap, que automáticamente obtiene información acerca de la topología física. En líneas 2 y 3 se declara una matriz de elementos de tipo double y se define su dominio(n x n). A continuación, en líneas 5 y 6 se crea una matriz de tipo HitBlockTile(HitTile_HitTile), que es un matriz de punteros a bloques, cada uno de tamaño blockSize x blockSize. En línea 7 se llama a un plug-in de Layout que distribuye el Tile de bloques entre los procesadores. El objeto matrixLayout entonces contendrá la información acerca qué elementos han sido asignados al procesador local, y en línea 9 se utiliza dicha información para seleccionar los bloques apropiados de la matriz de bloques original. En la línea 10 cada procesador reserva memoria para los bloques que le han sido asignados. En líneas 12 y 13 se declara y se obtiene un bloque de elementos, en este caso el bloque (0,0) y de tipo double, del Tile de bloques local.

3.3.3 Modo de almacenamiento y acceso de datos en Hitmap

Actualmente, en Hitmap los datos se almacenan de forma continua en un vector de datos. Un objeto de tipo HitTile tiene un campo llamado data, que es un puntero a la posición inicial del vector que contiene los elementos del Tile. Para acceder a ellos, se emplean diferentes fórmulas de acceso dependiendo de la dimensión, que calculan la posición del vector en la que se encuentra el elemento que se quiere acceder. Por ejemplo, para acceder a elemento (pos1,pos2) de un Tile de dos dimensiones la fórmula de acceso sería:

$$ElemAt(pos1, pos2) = pos1 \cdot n + pos2$$

Donde pos1 es la coordenada en el eje vertical, pos2 la coordenada en el eje horizontal, y n es el tamaño del eje horizontal. Ejemplo: Para acceder a la posición (2,3), sería acceder al elemento en la posición 11 del vector.

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

Almacenamiento (Filas-Columnas):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figura 3.8: Ejemplo fórmula de acceso al elemento (2,3) en una matriz de 4x4

3.4 Caso de estudio: Resolución de sistema de ecuaciones con factorización LU

Como caso de estudio de particionamiento bloque-cíclico se ha escogido un problema típico de resolución de ecuaciones lineales, que emplea dos algoritmos[7]: Factorización LU y resolución del sistema ecuaciones por substitución en una matriz factorizada.

Estos dos algoritmos presentan características que hacen de la partición bloque-cíclica la más adecuada. Además, utilizan dos patrones de dependencias (y por tanto de comunicaciones) diferentes, y representativos de varias clases de problemas similares. Por último, la cantidad de computación tiene diferentes ordenes de magnitud, lo que hace que el equilibrio entre computación y comunicación sea diferente en ambos casos, y se puedan observar diferentes efectos en el rendimiento cuando se modifican características de su implementación.

3.4.1 Factorización LU

La factorización LU se basa en la descomposición de la matriz original de coeficientes (A) en el producto de dos matrices (L y U), donde L es una matriz triangular inferior (con 1's en su diagonal) y U es una matriz triangular superior. A=LxU

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{pmatrix} \times \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

Figura 3.9. Factorización LU

3.4.1.1 Algoritmo de factorización LU

```
for k=0 to n-1 do
    {actualizar elementos de la matriz L}
    for i=k+1 to n-1 to
        a(i,k)=a(i,k)/a(i,kk)
    end for

    {actualizar resto de elementos}
    for j=k+1 to n-1 do
        for i=k+1 to n-1 do
            a(i,j)=a(i,j)-a(k,j)*a(i,k)
        end for
    end for
end for
```

Figura 3.10: Algoritmo de factorización LU

El coste temporal asintótico del algoritmo es de $O(n^3)$.

3.4.1.2 Ejemplo matemático factorización LU

Considerando el siguiente sistema de ecuaciones lineal

$$2x_1+3x_2+4x_3=6$$
$$4x_1+5x_2+10x_3=16$$
$$4x_1+8x_2+2x_3=2$$

Tenemos que la matriz de coeficientes es:

$$A=\begin{pmatrix} 2 & 3 & 4 \\ 4 & 5 & 10 \\ 4 & 8 & 2 \end{pmatrix} \qquad b=\begin{pmatrix} 6 \\ 16 \\ 2 \end{pmatrix}$$

Para obtener su factorización:
Primero convertimos en cero todos los elementos debajo del primer elemento diagonal de A. Para esto, sumamos la primera fila de A multiplicada por (-2) a la segunda fila de A y sumamos la primera fila de A también multiplicada por (-2) a la tercera fila de A. Obteniendo la siguiente matriz

$$U=\begin{pmatrix}2 & 3 & 4\\ 0 & -1 & 2\\ 0 & 2 & -6\end{pmatrix}$$

Mientras tanto, comenzamos la construcción de una matriz triangular inferior(L), con 1's en la diagonal principal. Para hacer esto, colocamos los opuestos de los multiplicadores utilizados en las operaciones de fila en la primera columna de L debajo del primer elemento diagonal y obtenemos la siguiente matriz

$$L=\begin{pmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 2 & 0 & 1\end{pmatrix}$$

Ahora sumamos (2) veces la segunda fila de U a la tercera fila. Colocamos los opuestos de los multiplicadores debajo del segundo elemento diagonal de L y obtenemos

$$L=\begin{pmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 2 & -2 & 1\end{pmatrix} \quad U=\begin{pmatrix}2 & 3 & 4\\ 0 & -1 & 2\\ 0 & 0 & -2\end{pmatrix}$$

3.4.2 Resolución de sistema de ecuaciones por substitución

La factorización LU puede ser empleada para resolver sistemas de ecuaciones, calcular la inversa o calcular determinantes. En este caso, nos vamos a centrar en el proceso de resolución de un sistema de ecuaciones $Ax=b$. En este caso, el sistema de ecuaciones dado podría representarse de la forma $LUx=b$. Si denominamos y a la matriz columna de n filas resultado del producto de las matrices Ux, tenemos que la anterior ecuación, se puede reescribir del siguiente modo: $Ly=b$. Por tanto, el algoritmo planteado para resolver el sistema de ecuaciones se puede resolver en dos etapas:

1) $Ly=b$, para obtener y aplicando un algoritmo de sustitución progresiva.

2) $Ux=y$, para obtener los valores de x aplicando un algoritmo de sustitución regresiva.

3.4.2.1 Algoritmo de resolución sistema ecuaciones

1) Resolver Ly=b

$$y_0 = \frac{b_0}{l_{0,0}}$$

For i=1,…,n-1

$$y_i = \frac{b_i - \sum_{j=0}^{i-1} l_{i,j} \cdot x_j}{l_{i,j}}$$

Figura 3.11: Algoritmo de resolución sistema de ecuaciones (Fase 1-sustitución progresiva)

2) Resolver Ux=y

$$x_{n-1} = \frac{y_{n-1}}{u_{n-1,n-1}}$$

For n-2,…,0

$$x_i = \frac{b_i - \sum_{j=i+1}^{n-1} u_{ij} \cdot x_j}{u_{ij}}$$

Figura 3.12: Algoritmo de resolución sistema de ecuaciones (Fase 2-sustitución regresiva)

El coste temporal asintótico de cada fase del algoritmo es de $O(n^2)$, por lo que el orden total del algoritmo de de 2 $O(n^2)$.

3.4.2.2 Ejemplo matemático resolución sistema ecuaciones

Considerando el siguiente sistema de ecuaciones lineal

$$2x_1 + 3x_2 + 4x_3 = 6$$
$$4x_1 + 5x_2 + 10x_3 = 16$$
$$4x_1 + 8x_2 + 2x_3 = 2$$

Tenemos que la matriz de coeficientes es $A = \begin{pmatrix} 2 & 3 & 4 \\ 4 & 5 & 10 \\ 4 & 8 & 2 \end{pmatrix}$ y $b = \begin{pmatrix} 6 \\ 16 \\ 2 \end{pmatrix}$

Y su factorización LU:

$$L=\begin{pmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 2 & -2 & 1\end{pmatrix} \quad U=\begin{pmatrix}2 & 3 & 4\\ 0 & -1 & 2\\ 0 & 0 & -2\end{pmatrix}$$

Empleando la ecuación $Ly=b$ por sustitución progresiva obtenemos:

$$\begin{pmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 2 & -2 & 1\end{pmatrix}\begin{pmatrix}y_1\\ y_2\\ y_3\end{pmatrix}=\begin{pmatrix}6\\ 16\\ 2\end{pmatrix} \qquad \begin{aligned} y_1&=6\\ y_2&=16-2\,y_1=4\\ y_3&=2+2\,y_2-2\,y_1=-2\end{aligned} \qquad y=\begin{pmatrix}6\\ 4\\ -2\end{pmatrix}$$

Empleando la ecuación $Ux=y$ por sustitución regresiva obtenemos:

$$\begin{pmatrix}2 & 3 & 4\\ 0 & -1 & 2\\ 0 & 0 & -2\end{pmatrix}\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix}=\begin{pmatrix}6\\ 4\\ -2\end{pmatrix} \qquad \begin{aligned} x_1&=\frac{6-4\,x_3-3\,x_2}{2}=4\\ x_2&=\frac{4-2\,x_3}{-1}=-2\\ x_3&=1\end{aligned} \qquad x=\begin{pmatrix}4\\ -2\\ 1\end{pmatrix}$$

3.4.3 Particionado de datos en LU

El algoritmo de factorización LU modificado para trabajar en bloques, va factorizando y actualizando la matriz de entrada, recorriéndola en diagonal de arriba hacia abajo y de izquierda a derecha. El proceso consta básicamente de cuatro etapas, que pueden verse gráficamente en la Figura 3.13:

1) Factorizar un bloque diagonal
2) Actualizar bloques en la misma columna que el bloque diagonal, utilizando el bloque diagonal.
3) Actualizar bloques en la misma fila que el bloque diagonal, utilizando el bloque diagonal.
4) Actualizar resto de la matriz utilizando para ello los bloques previamente actualizados en los pasos 2 y 3.

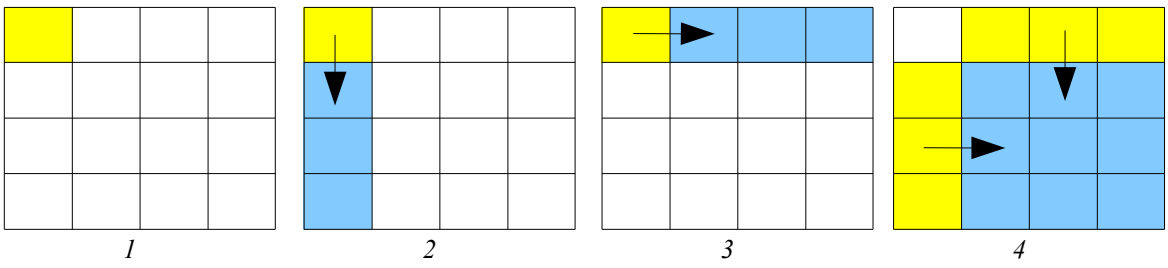
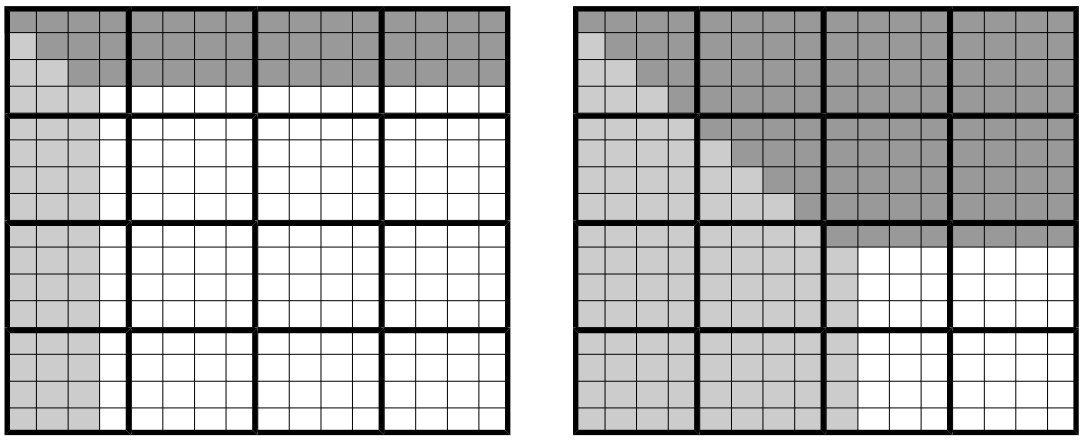


Figura 3.13: Etapas de la factorización LU trabajando por bloques

La paralelización de los algoritmos de factorización LU y resolución del sistema de ecuaciones por substitución presentan un problema de desequilibrio de carga [4] si se utiliza una partición en bloques consecutivos clásica, y un problema de exceso de comunicación [5] si se utiliza una partición cíclica pura.

Con la partición por bloques lo que pasa es que cuando va avanzando la computación se van quedando los primeros procesadores sin carga, por lo que hay un tiempo en el que no se aprovechan eficientemente los recursos computacionales disponibles, estando parte de los procesadores inactivos.

En la siguiente figura 3.14 se muestra un ejemplo de lo que pasa cuando se divide una matriz de 16x16 en bloques entre 16 procesadores. La parte de cuadros en blanco significa la parte de la matriz que se está computando en cada iteración. Podemos ver que en la iteración 3, la parte que está computando abarca todavía los 16 procesadores, pero cuando llega a la iteración 9 sólo los 4 últimos procesadores están computando, habiendo terminado el resto.

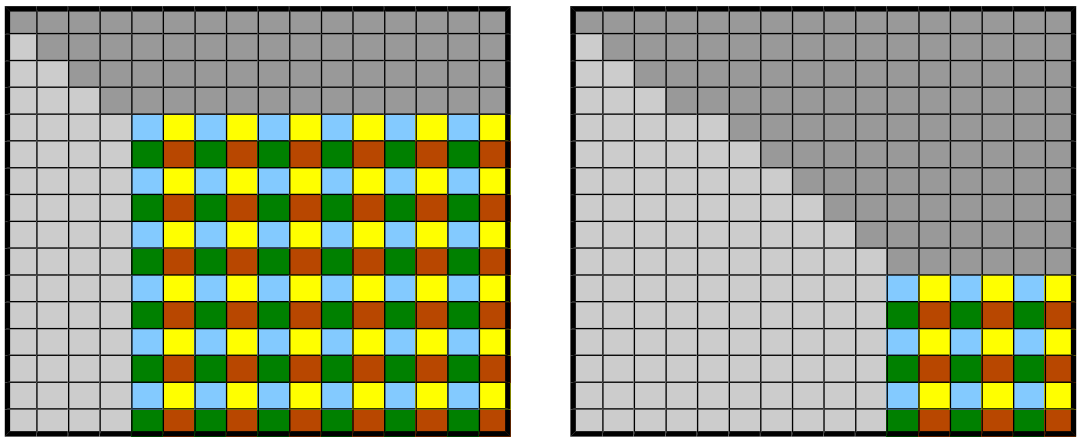


Iteración 3 *Iteración 9*

Figura 3.14: Partición por bloques entre 4x4 procesadores, con bloques de 4x4 elementos

Con un partición cíclica pura se soluciona el problema de equilibrio de carga, pero surge un gran problema, y es que en cada cálculo de elemento de la diagonal se dispara el número de comunicaciones pequeñas.

Por ejemplo, en la siguiente figura 3.15 se muestra cómo se realizaría una partición cíclica en un grid de 2x2 procesadores dónde cada procesador es representado por un color. Aquí se ve que tanto en la iteración 4, como en la 10, los 4 procesadores están activos teniendo todavía datos por computar. Parece que en principio se soluciona el problema de equilibrio de carga, pero como ya hemos comentado surge el problema de que para cada cálculo el procesador al no tener ningún elemento contiguo, necesita de los datos que poseen el resto de los procesadores, produciéndose comunicaciones de un elemento todo el rato.

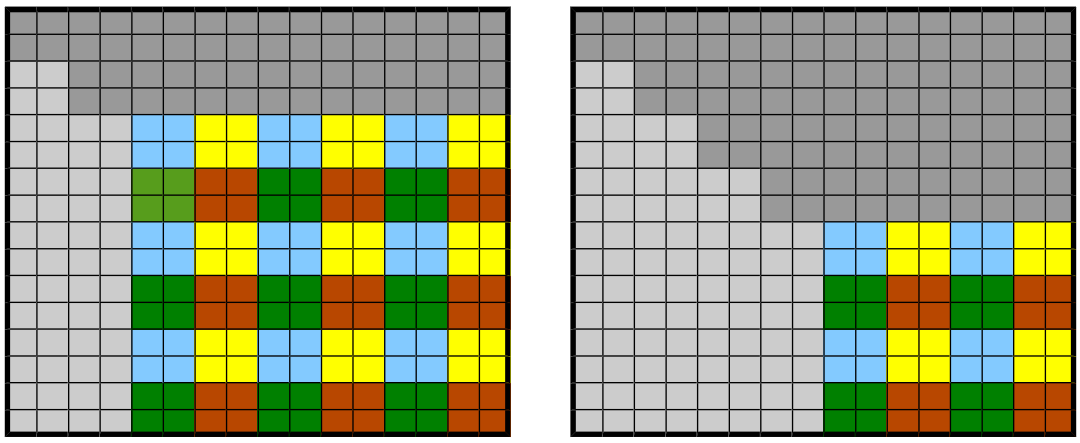


Iteración 4 *Iteración 10*

Figura 3.15: Partición cíclica entre 2x2 procesadores

Las soluciones habituales, confían en una combinación de las dos particiones anteriores: la partición bloque-cíclica, para compensar ambos problemas. De esta forma se aprovechan las ventajas de las dos (bloque y cíclica), reduciendo el número de comunicaciones y manteniendo un equilibrio de la carga entre los procesos.

En la siguiente figura 3.16 se puede observar que por ejemplo tanto en la iteración 4 como en la 8 todos los procesadores tienen carga de trabajo, y ya no es necesario que para cada calculo de un elemento se produzca una comunicación, pudiéndose comunicar bloques enteros entre sí.



Iteración 4 *Iteración 8*

Figura 3.16: Partición bloque-cíclica entre 2x2 procesadores con tamaño de bloque 2x2

3.4.4 Partición bloque-cíclica en LU

Cuando los datos se distribuyen entre un grid de procesadores como en la Figura 3.2, cada procesador tiene una tarea diferente a realizar en cada momento dependiendo de los bloques que le hayan sido asignados. Además, para mejorar el ratio de tiempos de comunicación vs tiempos computación, en el algoritmo paralelo de LU se realizan una serie de optimizaciones, en dónde la comunicación ahora ya no se limita a bloques enteros, si no que también se comunican entre sí filas y columnas de bloques, mejorando con ello el rendimiento de la aplicación paralela. Por tanto, la secuencia de tareas que tiene que realizar cada proceso se resumen en:

a) Si es un proceso que tiene un bloque diagonal de la matriz:
   1) Actualiza columna del bloque.
   2) Envía la columna del bloque a los procesos de la misma fila.
   3) Actualiza el resto de filas del bloque.
   4) Actualiza resto de bloques en la misma fila.
   5) Envía el bloque actualizado a otros procesos en la misma columna.
   6) Actualiza resto de bloques debajo del bloque diagonal.
   7) Envía los bloques actualizados a otros procesos en la misma columna.
   8) Actualiza resto de bloques (derecha/debajo) del bloque diagonal.

b) Si es un proceso que tiene un bloque en la misma columna que el bloque diagonal:
   1) Recibe un bloque diagonal.
   2) Actualiza los bloques en la misma columna que el bloque diagonal.
   3) Recibe una fila de bloques de otros procesos en la misma columna.
   4) Envía la fila de bloques actualizada a otros procesos en la misma.
   5) Actualiza resto de bloques (derecha/debajo) del bloque diagonal.

c) Si es un proceso que tiene un bloque en la misma fila que el bloque diagonal:
   1) Recibe una columna de un bloque.
   2) Actualiza los bloques en la misma fila.
   3) Recibe una columna de bloques.
   4) Actualiza resto de bloques (derecha/debajo) del bloque diagonal.

d) Si es un proceso que tiene bloque que no pertenece ni a la misma fila, ni a la misma columna que el bloque diagonal:
   1) Recibe una fila de bloques.
   2) Recibe una columna de bloques.
   3) Actualiza sus bloques.

3.5 Resumen del capítulo

En este capítulo se han presentado los conocimientos previos necesarios para entender el problema abarcado en este trabajo y la propuesta de solución que se presenta en el siguiente capítulo. En concreto, en primer lugar se ha comentado lo qué significa paralelismo de datos y los diversos tipos de particiones que se pueden realizar. También se ha introducido cómo es la librería de programación paralela Hitmap, y se ha visto la forma en la que se implementan las particiones en bloques y el acceso a los datos. Después se ha analizado el problema de LU, viendo los problemas que se presentan al elegir una determinada política de particiones, y el por qué la partición de bloque-cíclica es la que mejor se adapta para este algoritmo.

Capítulo 4. Desarrollo de software

En este capítulo se presenta el origen de la idea de crear un modo diferente de almacenamiento de datos en la librería Hitmap y cómo se se llevado a cabo su desarrollo.

4.1 Propuesta

Analizando el modo en que actualmente se implementa en Hitmap el almacenamiento de los datos en memoria para aplicaciones paralelas particionadas en bloques, se concluye que el almacenamiento jerárquico es una estructura compleja y dinámica (creada utilizando punteros), que puede afectar a las potenciales optimizaciones que pueden realizar los compiladores modernos. En el caso de intentar utilizar una estructura no jerárquica, creando una forma de determinar y localizar los bloques sobre los datos almacenados en un único array (o Tile en términos de Hitmap), no se estaría aprovechando la localidad de los datos. Al acceder a los elementos de un bloque y estos estar almacenados por filas y columnas, como se puede ver en la Figura 4.1, se producirían accesos a elementos no contiguos en memoria, que podrían implicar frecuentes fallos de caché y con ello pérdida, en parte, de rendimiento.

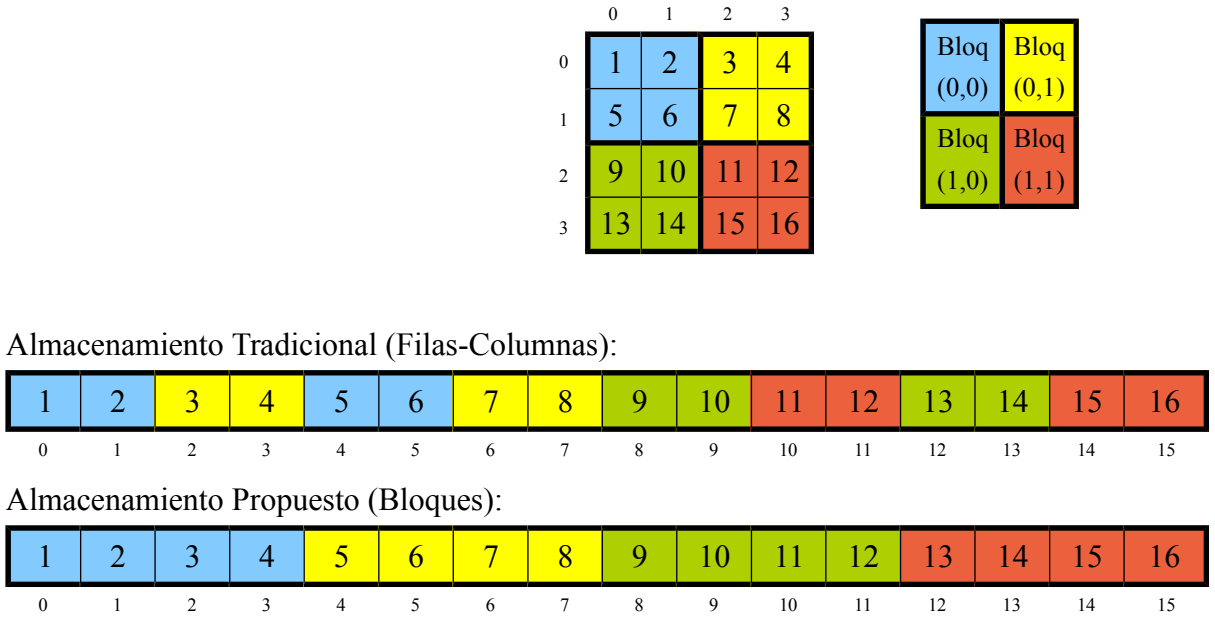


Figura 4.1: Ejemplo modos almacenamiento de datos para una matriz 4x4 dividida en bloques de 2x2

Por este motivo, se propone una nueva implementación para cambiar la forma en que se almacenen los datos en Hitmap: almacenando consecutivamente los datos de cada bloque, para así intentar realizar el mayor número de acceso a datos contiguos en memoria posible y no tener que estar cargando y descargando datos de la caché con tanta frecuencia.

Para llevar a cabo el nuevo modo de almacenamiento, se diseña en primer lugar realizar un cambio en la estructura de datos de las Signatures, ampliándolas con nuevos campos para así poder representar un subespacio de un Tile mediante índices de bloques. Posteriormente, se desarrollará una nueva interfaz en Hitmap, en la que se necesitarán de funciones para:

- Acceder a los nuevos campos de las Signatures ampliadas.
- Definir las dimensiones de un Tile y de sus bloques.
- Reservar/liberar memoria para los Tiles.
- Seleccionar un determinado bloque (x,y) de un Tile.
- Seleccionar conjunto de bloques definido por un Shape de un Tile.
- Acceder a los datos:
  - Por coordenada global del Tile.
  - Por coordenada local dentro de un bloque escogido por coordenadas de bloque.
- Comprobar si un índice está en dominio del Tile.
- Obtener el Tile antecesor, del cuál se ha realizado una subselección.

4.2 Diseño e implementación

4.2.1 Signatures de Hitmap ampliadas

Si antes las Signatures se componían de una tupla de tres elementos, que representaban un subespacio de índices de elementos:

- Begin: Índice de comienzo de los elementos
- End: Índice final de los elementos
- Stride: Salto entre los elementos

Ahora se decide ampliarlas añadiendo otros cuatro campos: uno para saber el tamaño del bloque y otros tres para representar un subespacio de índices de bloques.

- SizeBlock: Tamaño del bloque
- BeginBlock: Índice de comienzo de los bloques
- EndBlock: Índice final de los bloques
- StrideBlock: Salto entre bloques

```
typedef struct {
    int begin;          /**< The begin index of the dimension */
    int end;            /**< The end index of the dimension */
    int stride;         /**< The stride for regular sparse domains */
    int sizeBlock;      /**< The size of a block */
    int beginBlock;     /**< The begin index block of the dimension */
    int endBlock;       /**< The end index block of the dimension */
    int strideBlock;    /**< The stride block for regular sparse domains */
} HitSig;
```

Figura 4.2: Implementación en Hitmap de HitSig

La necesidad de representar un subespacio de un Tile con índices de bloques surge del problema de conocer a qué bloques originales corresponden los bloques de una subselección, una vez que los datos están almacenados de forma contigua por bloques.

4.2.1.1 Ejemplo Signatures ampliadas

Para entenderlo mejor, en la figura 4.3, se muestra un ejemplo en el que se ve en el pie de cada matriz el Shape que lo identifica.

Así, por ejemplo, como el Tile de la figura 4.3 tiene dos dimensiones, el subespacio quedaría definido por un Shape con dos Signatures.

Vemos que el Shape de la matriz original de 8x8 elementos, dividida en bloques de 2x2 sería {(0:7:1:2:0:3:1) , (0:7:1:2:0:3:1)}. Los tres primeros números indican los índices de los elementos (de 0 a 7 de uno en uno), el cuarto indica el tamaño del bloque (2), y los tres últimos son los índices de bloques (de 0 a 3 bloques de uno en uno).

Para una subselección del Tile, por ejemplo, tomando la del proceso (1,0) {(0:3:1:2:1:3:2), (0:3:1:2:0:2:2)} se selecciona un trozo de 4x4 elementos (de 0 a 3 de uno en uno), el tamaño de bloque es 2, y el conjunto de bloques seleccionados son: en el eje vertical de 1 a 3 de dos en dos, y en el eje horizontal de 0 a 2 de dos de dos en dos, es decir, se seleccionan los bloques (1,0), (1,2), (3,0), (3,2) de la matriz original.

4.2.2 Diseño de métodos de acceso

Para acceder a los datos con esta nueva organización por bloques, se requiere de la aplicación de nuevas fórmulas de acceso, que calculen la nueva posición en que se encuentran datos. Se desarrollarán dos métodos de acceso a los elementos: por coordenadas de Tile y por coordenadas de bloque.

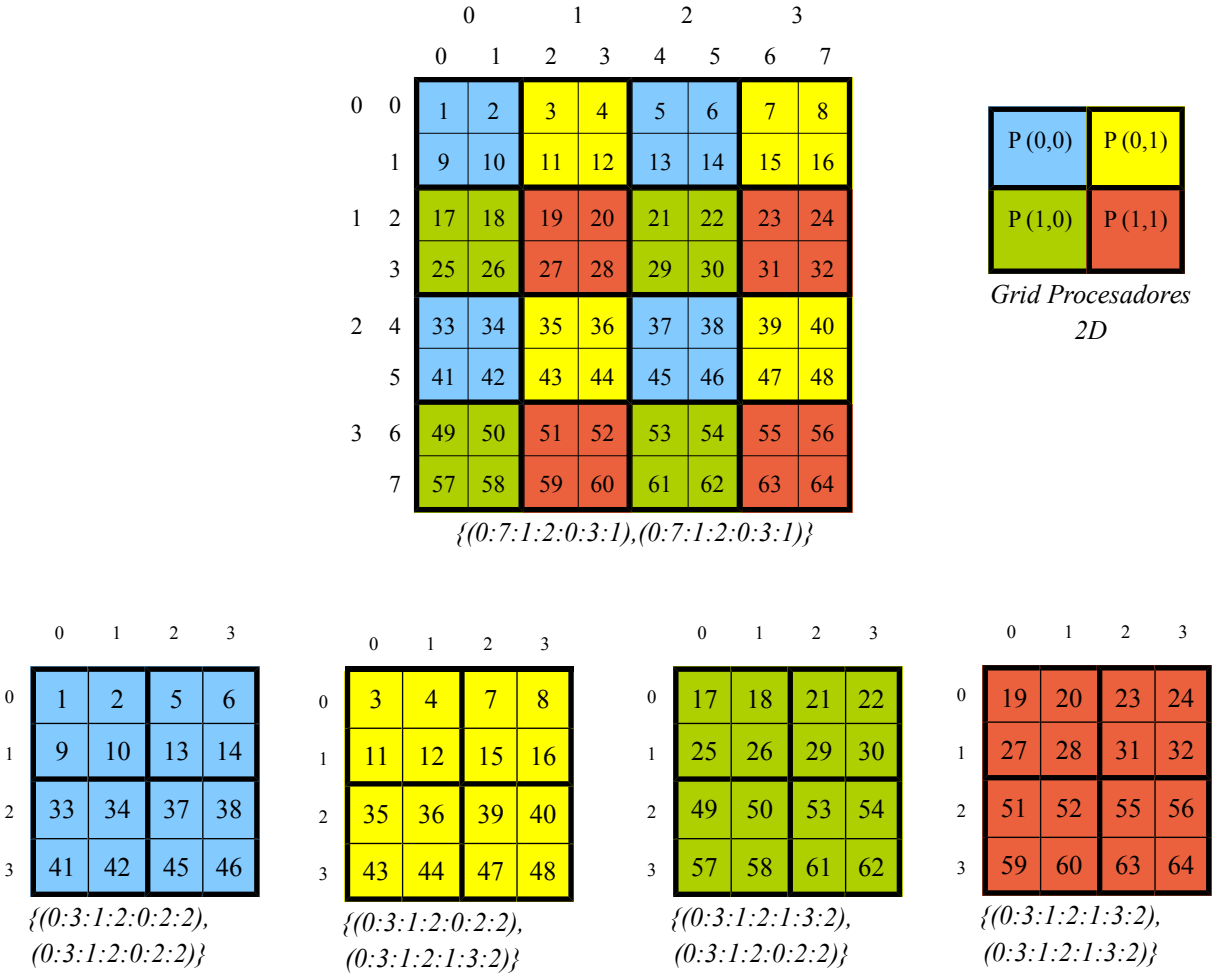


Figura 4.3: Ejemplo de Signatures ampliadas de una matriz 8x8 dividida en bloques de 2x2 distribuida de forma bloque-cíclica entre 4 procesadores

- Fórmula acceso Elemento(pos1,pos2) en Tile de dos dimensiones:

$$ElemAt(pos1, pos2) = DivEnt(\frac{pos1}{tb1}) \cdot n \cdot tb1 + RestDiv(\frac{pos1}{tb1}) \cdot tb + DivEnt(\frac{pos2}{tb2}) \cdot tb1 \cdot tb2 + RestDiv(\frac{pos2}{tb2})$$

- Fórmula acceso Elemento(pos1,pos2) de un Bloque(bloq1,bloq2) en Tile de dos dimensiones:

$$ElemAtBlock(bloq1, bloq2, pos1, pos2) = bloq1 \cdot n \cdot tb1 + bloq2 \cdot tb1 \cdot tb2 + pos1 \cdot tb2 + pos2$$

Siendo:

- *pos1* la coordenada de acceso en el eje vertical
- *pos2* la coordenada de acceso en el eje horizontal
- *tb1* el tamaño de bloque en la dimensión vertical
- *tb2* el tamaño de bloque en la dimensión horizontal
- *n* el número de elementos en en el eje horizontal de la matriz
- *DivEnt(x,y)* es el cociente obtenido de aplicar la división entera de *x* entre *y*
- *RestDiv(x,y)* es el resto obtenido de aplicar división entera de *x* entre *y*

Así por ejemplo, como se muestra en la siguiente figura, en una matriz 4x4 con bloques de 2x2, para acceder al elemento (2,3) por coordenadas de Tile sería:

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

$$ElemAt(2,3) = DivEnt(\frac{2}{2}) \cdot 4 \cdot 2 + RestDiv(\frac{2}{2}) \cdot 2$$
$$+ DivEnt(\frac{3}{2}) \cdot 2 \cdot 2 + RestDiv(\frac{3}{2}) = 8 + 0 + 4 + 1 = 13$$

Almacenamiento (Filas-Columnas):

1	2	5	6	3	4	7	8	9	10	13	14	11	12	15	16
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figura 4.4: Ejemplo de fórmula de acceso al elemento(2,3) en una matriz 4x4 con tamaño de bloque de 2x2

Se puede ver la diferencia de acceso que existe con el método anterior de almacenamiento de datos por filas-columnas comparando con la Figura 3.8 dónde se accedía al mismo elemento (2,3) y en aquel caso se accedía a la posición 11 del puntero data en vez de la 13 como ahora.

4.2.3 Interfaz

4.2.3.1 Acceso nuevos campos de las Signatures

Para acceder en un Tile a los nuevos campos de las Signatures ampliadas se definen unos macros, siguiendo la misma forma de implementación que se usa para acceso a los tres campos de las Signatures originales.

```
#define hit_tileDimSizeBlock(var,dim)        (hit_tileDimSig(var,dim).sizeBlock)
#define hit_tileDimBeginBlock(var,dim)       (hit_tileDimSig(var,dim).beginBlock)
#define hit_tileDimEndBlock(var,dim)         (hit_tileDimSig(var,dim).endBlock)
#define hit_tileDimStrideBlock(var,dim)      (hit_tileDimSig(var,dim).strideBlock)
```

Al igual que se obtiene la cardinalidad de los elementos de un Signature con los 3 primeros campos(begin, end, stride), como se introdujo en el capítulo 3 cuando se explicó que era una Signature, se define un macro para obtener la cardinalidad en bloques de un Signature, aplicando la misma fórmula, pero empleando los nuevos campos (beginBlock, endBlock, strideBlock).

```
#define hit_sigCardBlock(sig)   (((sig).endBlock-(sig).beginBlock)/(sig).strideBlock+1)
```

También usando estos nuevos campos de las Signatures se define otro macro para comprobar si un índice de bloques es válido para el dominio.

```
#define hit_sigBlockIn(sig,a)   (a>=(sig).beginBlock && a<=(sig).endBlock &&
                                 ((a-(sig).beginBlock)%(sig).strideBlock == 0))
```

4.2.3.2 Creación de un Tile

Para crear un nuevo Tile se emplea una función, basada en la original de Hitmap, en la que se define el baseType y numero de dimensiones del Tile y de sus bloques. El primer parámetro es el puntero al nuevo Tile, el segundo indica el tipo de los elementos contenidos, el tercero indica el número de dimensiones del Tile, y el cuarto permite un número variable de argumentos como parámetros en dónde se indica el tamaño de las dimensiones del Tile y de sus bloques. El orden de estos últimos parámetros viene dado por la dimensión, por lo que, por ejemplo, el orden correcto de pasar los parámetros para declarar un Tile 2D 8x8, con bloques de 2x2, sería 8,2,8,2. Es decir, tamMatriz0, tamBloque0, tamMatriz1, tamBloque1.

```
void hit_bloqTileDomain( void *newVarP, size_t baseType, int numDims, ...)
```

4.2.3.3 Reserva y liberación de memoria de un Tile

Las funciones para reservar y liberar memoria de un Tile no necesitan cambios respecto a la versión original de Hitmap, ya que los elementos que contiene un Tile siguen siendo representados por los 3 primeros campos de cada Signature.

```
#define hit_bloqTileAlloc( var )     hit_tileAllocInternal( var, #var, __FILE__, __LINE__);
#define hit_bloqTileFree( tile )     hit_tileFree( tile )
```

4.2.3.4 Traducir Signatures de bloques tradicionales con 3 campos a las nuevas con 7 campos

Como las funciones de Layout originales, que distribuyen los datos del Tile entre los procesadores de la topología, emplean solo los 3 primeros campos de cada Signature, se decide en vez de adaptar todas las funciones de la clase Layout a las nuevas Signaturas, crear una función que directamente traduce el Shape que devuelven las funciones de Layout en los nuevos Shape de con Signatures de 7 campos, que son las que se emplean actualmente con este método del almacenamiento por bloques. Para ello hay que pasarle el Shape de bloques de las funciones de Layout, y el tamaño de bloque de cada dimensión.

HitShape hit_bloqShape3ToShape7(HitShape sh, ...)

4.2.3.5 Selección de determinado bloque (x,y)

Un bloque es al fin y al cabo un Tile que corresponde a una subselección del Tile original. El acceso a un determinado bloque (x,y) de un Tile se lleva a cabo mediante una función que crea un nuevo Tile, en el que el puntero interno de la nueva estructura apunta a la posición de memoria de inicio del bloque, ya que al estar los elementos del bloque contiguos, con saber la posición inicial ya se puede acceder inmediatamente al resto de elementos del bloque.

*void hit_bloqTileGetBlock (void *newVarP, const void *oldVarP, int bloqx, int bloqy)*

La posición de memoria de inicio del bloque se calcula teniendo en cuenta las coordenadas de bloque introducidas por el usuario (bloqx, bloqy), el tamaño del bloque, y la dimensión del eje horizontal del Tile. Aquí hay que distinguir dos casos: cuando se selecciona un bloque de un Tile del que previamente se ha hecho un alloc en memoria, o si no se ha hecho alloc. La diferencia está en que si no se ha reservado memoria antes, hay que multiplicar las coordenadas de bloques por los saltos de bloque en dichas coordenadas, ya que significaría que el Tile sobre el que se quiere seleccionar un bloque sería un puntero a otro Tile.

```
/* Calcular el desplazamiento del puntero */
if ( oldVar->memStatus == HIT_MS_NOMEM ) //Si se ha reservado memoria para el Tile
  offset =
    (size_t)bloqx*(size_t)(oldVar->origAcumCard[1])* (size_t)hit_tileDimSizeBlock(*oldVar,0) +
    (size_t)bloqy*(size_t)hit_tileDimSizeBlock(*oldVar,1)*(size_t)hit_tileDimStrideBlock(*oldVar,1);

else //Si no se ha reservado memoria para el Tile
  offset =
     (size_t)bloqx * (size_t)(oldVar->origAcumCard[1]) * (size_t)hit_tileDimSizeBlock(*oldVar,0) *
       (size_t)hit_tileDimStrideBlock(*oldVar,0) +
    (size_t)bloqy * (size_t)hit_tileDimSizeBlock(*oldVar,0) * (size_t)hit_tileDimSizeBlock(*oldVar,1)*
       (size_t)hit_tileDimStrideBlock(*oldVar,1);
```

A continuación se muestra un ejemplo del cálculo del puntero para realizar el acceso al bloque (1,1) de una matriz 4x4 con bloques de 2x2.

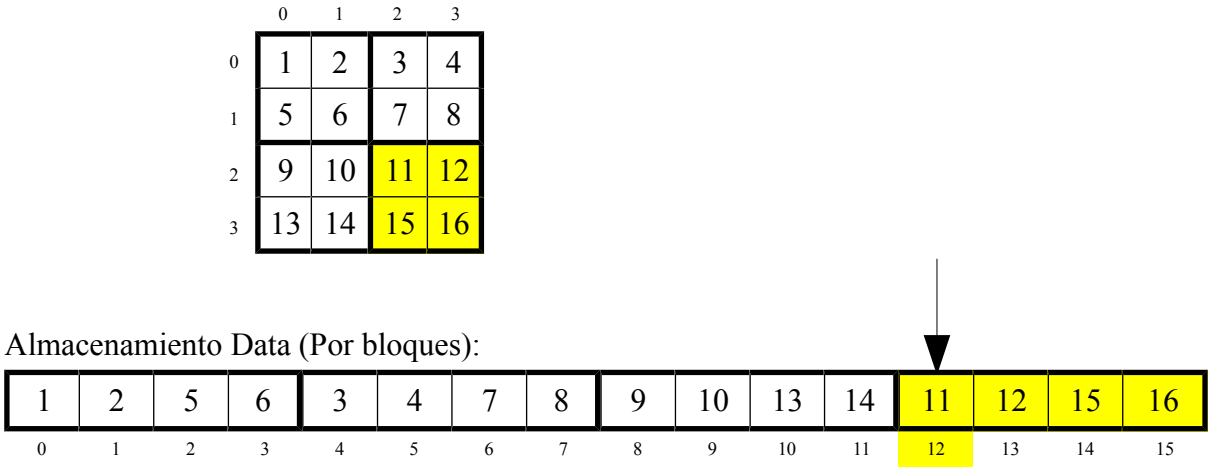


Figura 4.5: Ejemplo cálculo puntero data para la selección del bloque(1,1) de una matriz 4x4 con tamaño de bloque 2x2

4.2.3.6 Subselección de un conjunto de elementos mediante un Shape

Para realizar la selección de un conjunto de bloques definido por un Shape, al igual que con la selección de un bloque se crea un Tile con el puntero data apuntando la dirección de memoria de inicio de la selección, calculado a partir de la posición de inicio del primer bloque, el tamaño del bloque y la dimensión del Tile.

*int hit_bloqTileSelect(void *newVarP, const void *oldVarP, HitShape sh)*

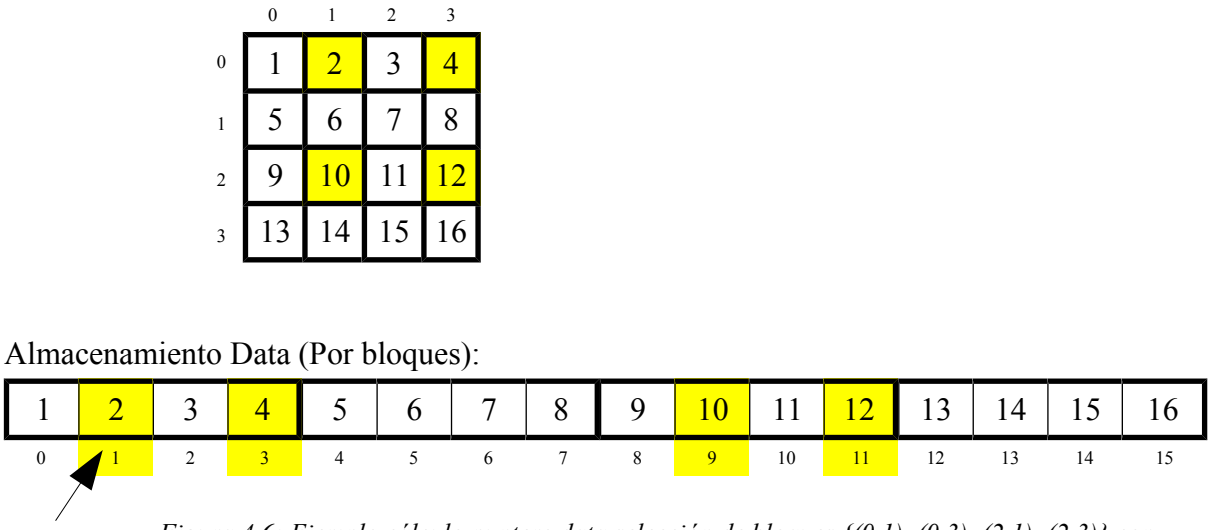


Figura 4.6: Ejemplo cálculo puntero data selección de bloques {(0,1), (0,3), (2,1), (2,3)} con bloques de 1x1 en una matriz 4x4

4.2.3.7 Métodos de acceso a los elementos

Para implementar las fórmulas de acceso, al igual que con el resto de fórmulas, se decide seguir la idea original de Hitmap de usar macros frente al uso de funciones, porque exponen al compilador las formulas aritméticas dentro del contexto en el que se usan, de forma que los módulos de optimización pueden analizarlas y hacer substituciones sin necesidad de análisis interprocedural, que no está soportado en general en los compiladores.

Como ya se comentó se implementan dos formas de acceder a los datos: coordenadas de Tile y coordenadas de bloque.

- Acceso por coordenadas de Tile:

```
#define hit_bloqTileElemAt(var,ndims,...)   hit_bloqTileElemAt##ndims(var,__VA_ARGS__)
#define hit_bloqTileElemAt1(var, pos)       ((var).data[(pos)*(var).qstride[0]])
#define hit_bloqTileElemAt2(var, pos1, pos2) ((var).data[(
    (int(pos1)/hit_tileDimSizeBlock(var,0))*(var).origAcumCard[1]*hit_tileDimSizeBlock(var,0)+
    (((pos1%hit_tileDimSizeBlock(var,0))*hit_tileDimSizeBlock(var,1)))+
    (int)pos2/hit_tileDimSizeBlock(var,1))*hit_tileDimSizeBlock(var,0)*hit_tileDimSizeBlock(var,1) +
    (pos2%hit_tileDimSizeBlock(var,1))])
```

- Acceso por coordenadas de bloque:

```
#define hit_bloqTileElemAtBloq(var,ndims,...)          hit_bloqTileElemAtBloq##ndims(var,__VA_ARGS__)
#define hit_bloqTileElemAtBloq1(var,nbloq,pos) ((var).data[nbloq*hit_tileDimSizeBlock(var,0)+
(pos)*(var).qstride[0]])
#define hit_bloqTileElemAtBloq2(var, bloq1, bloq2, pos1, pos2)
    ((var).data[(var).origAcumCard[1]*hit_tileDimSizeBlock(var,0)*bloq1+
    bloq2*hit_tileDimSizeBlock(var,0)*hit_tileDimSizeBlock(var,1)+
    pos1*hit_tileDimSizeBlock(var,1)+pos2])
```

También se hace necesario el uso de fórmulas para comprobar si un índice de coordenadas concreto está o no contenido en el Tile.

```
#define hit_tileDimHasBlockArrayCoord(var, dim, pos) \
    ( pos>=hit_tileDimBeginBlock(var,dim) && (pos<=hit_tileDimEndBlock(var,dim)) && \
    ((pos-hit_tileDimBeginBlock(var,dim))%hit_tileDimStrideBlock(var,dim) == 0) )
#define hit_tileHasBlockArrayCoords(var, ndims, ...)   hit_tileHasArrayCoords##ndims(var, __VA_ARGS__)
#define hit_tileHasBlockArrayCoords2(var, pos0, pos1) \
    ( pos0>=hit_tileDimBeginBlock(var,0) && (pos0<=hit_tileDimEndBlock(var,0)) && \
        pos1>=hit_tileDimBeginBlock(var,1) && (pos1<=hit_tileDimEndBlock(var,1)) &&
        ((pos0-hit_tileDimBeginBlock(var,0))%hit_tileDimStrideBlock(var,0) == 0) &&
        ((pos1-hit_tileDimBeginBlock(var,1))%hit_tileDimStrideBlock(var,1) == 0) )
```

4.2.3.8 Otras modificaciones de la librería Hitmap necesarias

- Implementar un nuevo tipo derivado MPI para permitir realizar la comunicación de forma correcta con el nuevo modo de almacenamiento de datos en bloques. Hitmap emplea el uso de tipos derivados de MPI para crear para cada comunicación una estructura interna de MPI que hace automáticamente el cálculo de posiciones de los datos a partir de un puntero en el momento de comunicar. Es necesario cambiar la implementación del método *hit_comType()* para que ahora calcule correctamente las posiciones de los datos a comunicar con el método de almacenamiento en bloques propuesto.

4.3 Ejemplo de uso de la interfaz propuesta

A continuación se muestra un ejemplo para mostrar al lector cómo sería la estructura de la implementación de un típico código paralelo. El ejemplo consiste en la creación, partición y distribución cíclica de los datos de una matriz y un vector entre los procesadores. Además el ejemplo es una parte de la estructura del caso de estudio de resolución de resolución de sistemas de ecuaciones lineales.

```
1.  // 1. VIRTUAL TOPOLOGY
2.  HitTopology topo;
3.  topo = hit_topology(plug_topArray2DComplete);
4.
5.  // 2. CREATION OF TILE (DECLARE THE FULL DOMAINS)
6.  HitTile_double matrix, vector;
7.  hit_bloqTileDomain( &matrix, double, 2, n, blockSize, n, blockSize );
8.  hit_bloqTileDomain( &vector, double, 2, 1, 1, n, blockSize );
9.
10. // 3. COMPUTE LAYOUTS
11. HitLayout matrixLayout = hit_layout(plug_layCyclic, topo, hit_bloqTileShape( &matrix ), HIT_LAYOUT_NODIM );
12. HitLayout vectorLayout = hit_layout(plug_layCyclic, topo, hit_bloqTileShape( &vector ), HIT_LAYOUT_NODIM );
13.
14. // 4. ONLY ACTIVE PROCESSES CONTINUE WITH THE COMPUTATION
15. if ( hit_layImActive( matrixLayout ) ) {
16.     // 4.1. DECLARE AND ALLOCATE LAYOUT SELECTIONS
17.     HitTile_double matrixSelection, vectorSelection;
18.     hit_bloqTileSelect(&matrixSelection,&matrix,hit_bloqShape3ToShape7(hit_layShape(matrixLayout),
        blockSize,blockSize ));
19.     hit_bloqTileSelect(&vectorSelection,&vector,hit_bloqShape3ToShape7(hit_layShape( vectorLayout ), 1,blockSize ));
20.     hit_tileAlloc( &matrixSelection );
21.     hit_tileAlloc( &vectorSelection );
22. // COMPUTATION
23.  }
```

Figura 4.7: Ejemplo uso de la interfaz de partición en bloques propuesta

En primer lugar se define la topología a usar. Luego se declaran los Tile y su domino, en este caso, en la línea 7 se declara un Tile de dos dimensiones de n x n elementos divididos en bloques de tamaño blockSize x blockSize; y en la línea 8 se declara un Tile de 1 x n elementos divididos en bloques de 1 x blockSize. A continuación en líneas 11 y 12 se lleva a cabo la distribución en bloques de manera cíclica entre los procesadores dada la topología y el Shape de bloques de cada Tile. A partir de la línea 15, sólo los procesadores activos continúan. En líneas 18 y 19 cada procesador realiza una selección de los bloques que le han sido asignados de la matriz original, y posteriormente se reserva memoria para dichos conjuntos de bloques. Finalmente se llevan a cabo los cálculos del problema de forma paralela, cada proceso trabajando sobre sus datos que le han sido asignados.

4.4 Pruebas

Con objeto de comprobar que la interfaz desarrollada funciona correctamente para los casos que se van a tener en cuenta en el estudio experimental, se llevará a cabo un conjunto de pruebas. Decir que puede que no se abarquen todos los casos específicos posibles, ya que el objetivo no es desarrollar un sistema modo de almacenamiento en bloques con funciones completamente genérico. Lo que se busca es realizar una implementación de manera que permita realizar un estudio experimental adecuado, ya que no sabemos si esto va a ir mejor o peor que el método anterior.

Los distintos tipos de pruebas que se han realizado han sido: pruebas sobre los Shape de la matriz de datos con las nuevas Signatures,

4.4.1 Pruebas sobre el Shape de la matriz de datos con las nuevas Signatures

- Un procesador
  - Tamaño de matriz 4x4

Tamaño de bloque	Salida esperada	Resultado
1	{(0:3:1:1:0:3:1),(0:3:1:1:0:3:1)}	Correcto
2	{(0:3:1:2:0:1:1),(0:3:1:2:0:1:1)}	Correcto
3	{(0:5:1:3:0:1:1),(0:5:1:3:0:1:1)}	Correcto
4	{(0:3:1:4:0:0:1),(0:3:1:4:0:0:1)}	Correcto

  - Tamaño de matriz 7x7

Tamaño de bloque	Salida esperada	Resultado
1	{(0:6:1:1:0:6:1),(0:6:1:1:0:6:1)}	Correcto
2	{(0:7:1:2:0:3:1),(0:7:1:2:0:3:1)}	Correcto

3	{(0:8:1:3:0:2:1),(0:8:1:3:0:2:1)}	Correcto
4	{(0:7:1:4:0:1:1),(0:7:1:4:0:1:1)}	Correcto
5	{(0:9:1:5:0:1:1),(0:9:1:5:0:1:1)}	Correcto
6	{(0:11:1:6:0:1:1),(0:11:1:6:0:1:1}	Correcto
7	{(0:6:1:7:0:0:1),(0:6:1:7:0:0:1)}	Correcto

- Dos procesadores
  - Tamaño de matriz 4x4

Tamaño de bloque	Salida esperada	Resultado
1	Pro(0,0): {(0:1:1:1:0:2:2),(0:3:1:1:0:3:1)} Pro(1,0): {(0:1:1:1:1:3:2),(0:3:1:1:0:3:1)}	Correcto
2	Pro(0,0): {(0:1:1:2:0:0:2),(0:3:1:2:0:1:1)} Pro(1,0): {(0:1:1:2:1:1:2),(0:3:1:2:0:1:1)}	Correcto
4	Pro(0,0): {(0:3:1:4:0:0:2),(0:3:1: 4: 0:0:1)}	Correcto

4.4.2 Pruebas acceso a los datos

4.4.2.1 Acceso por coordenadas de Tile

- Tamaño de matriz 4x4

Tamaño de bloque	Resultado
1	Correcto
2	Correcto
3	Correcto
4	Correcto

- Tamaño de matriz 9x9

Tamaño de bloque	Resultado
1	Correcto
2	Correcto
3	Correcto
4	Correcto
5	Correcto
6	Correcto

7	Correcto
8	Correcto
9	Correcto

4.4.2.2 Acceso por coordenadas de bloque

- Tamaño de matriz 4x4

Tamaño de bloque	Resultado
1	Correcto
2	Correcto
3	Correcto
4	Correcto

- Tamaño de matriz 9x9

Tamaño de bloque	Resultado
1	Correcto
2	Correcto
3	Correcto
4	Correcto
5	Correcto
6	Correcto
7	Correcto
8	Correcto
9	Correcto

4.4.2.3 Accesos mediante extracción de subselecciones en bloques

- Tamaño de matriz 4x4

Tamaño de bloque	Resultado
1	Correcto
2	Correcto
3	Correcto
4	Correcto

- Tamaño de matriz 9x9

Tamaño de bloque	Resultado
1	Correcto
2	Correcto
3	Correcto
4	Correcto
5	Correcto
6	Correcto
7	Correcto
8	Correcto
9	Correcto

4.5 Resumen del capítulo

En este capítulo se ha presentado en primer lugar el origen de la idea de crear un modo diferente de almacenamiento de datos en la librería Hitmap. Después se a explicado la necesidad de tener que realizar ciertos cambios en la estructurales y de interfaz para poder llevar a cabo la implementación de la propuesta. Se han detallado los cambios realizados y las funciones desarrolladas. Posteriormente, se ha mostrado con un ejemplo de paralización típico cómo sería la estructura del código empleando la interfaz desarrollada. Y finalmente se ha hecho una traza de pruebas para verificar que el código desarrollado funciona correctamente.

Capítulo 5. Estudio experimental

Se ha desarrollado un estudio experimental relacionado con la eficiencia de la partición bloque-cíclica, en función de su implementación y del parámetro del tamaño de bloque. Se ha desarrollado un estudio para comprobar las ventajas/desventajas de nuestra propuesta de sistema de almacenamiento de datos en bloques frente a la tradicional (por filas-columnas). Para ello, en primer lugar, se realizará un experimento para estudiar el impacto que tiene en el rendimiento de una aplicación el modo en que se acceden a los datos dependiendo de cómo estén almacenados en memoria. A continuación, se realizará un experimento para verificar las ventajas y desventajas de nuestra propuesta de almacenamiento en bloques frente a la tradicional y la versión MPI original de la aplicación LU. Por último, se ha llevado a cabo un estudio para observar el modo en que afecta la selección de un determinado tamaño de bloque en el rendimiento de una aplicación.

5.1 Metodología de la experimentación

1) Se han escogido benchmarks representativos:
   (a) Suma de matrices porque representa un tipo de problema en dónde es fácil comprobar cómo de eficientes son los accesos a los datos según el modo en que estén almacenados en memoria, ya que para recorrer los elementos tan sólo es necesario emplear unos bucles sobre los índices y la computación se reduce a una única operación aritmética, por lo que en el tiempo de ejecución tiene más impacto el tiempo de acceso a memoria.
   (b) Factorización LU y resolución del sistema de ecuaciones porque, como ya se comentó en el capítulo 3, es un ejemplo típico y representativo donde se es necesario el uso de una partición de datos bloque-cíclica y los dos algoritmos implicados presentan características muy diferentes en cuanto a dependencias, carga y comunicaciones.
2) Se ha implementado varias versiones la suma de matrices, con los diferentes modos de recorrer los datos(filas-columnas,bloques) y las dos formas de almacenamiento (filas-columnas,bloques) de los datos, para comprobar cómo afectan al rendimiento.
3) Se ha reimplementado la aplicación LU aplicando el modo de almacenamiento de datos propuesto por bloques empleando para ello la nueva interfaz.
4) Para el desarrollo de los experimentos se han seleccionado estructuras de tamaños de datos suficientemente grandes para obtener tiempos de experimentación estables, y que permitan escalar para el número de procesadores de nuestras plataformas.

5) Las pruebas experimentales se han llevado a cabo en plataformas con diferentes características para los experimentos donde pueda resultar relevante. Las dos plataformas paralelas disponibles en el grupo que se han seleccionado para la realización de las pruebas han sido:
   - Heracles: Es una máquina perteneciente al grupo Trasgo de la Universidad de Valladolid. Consta de 4 AMD Opteron 6376 cuya velocidad de procesador es 2.3 GHz con 16 núcleos cada procesador, por tanto, tiene un total de 64 núcleos. Cada procesador tiene 16 cachés L1 de 16 KB, 8 cachés L2 de 2 MB y 2 cachés L3 de 8 MB. La máquina dispone de 256 GB de RAM.
   - Chimera: También es una máquina perteneciente al grupo Trasgo de la Universidad de Valladolid. Consta de 2 Intel Xeon CPU E5-2620 v2 cuya velocidad es de 2.10 GHz con 6 núcleos cada uno. Con el sistema de hyperthreading de Intel activado cada procesador puede ejecutar simultáneamente el doble de hilos que núcleos reales, por lo que aparentemente cada procesador se ve desde el sistema operativo como si tuviera 12 núcleos. Lo que nos da un total de 24 núcleos. Cada procesador tiene caché 6 cachés L1 de 32 KB, 6 cachés L2 de 256 KB,y una caché L3 de 15 MB. La máquina dispone de 32 GB de RAM.
   - ¿Por qué son máquinas tan diferentes?
     - La arquitectura del procesador es distinta. Intel tienen menos núcleos por procesador duplicados por hyperthreading, pero comparten cachés. Hay diferencias en los sistemas internos de planificación de hilos, en los ratios de velocidad de computo vs. latencia de memoria, lo que afecta a los tiempos relativos de las comunicaciones tanto entre núcleos del mismo procesador, como entre procesadores.
   - En todos los experimentos la máquina por defecto mientras no se especifique lo contrario es Heracles.

5.2 Experimentos

5.2.1 Acceso datos con diferente modo de almacenamiento

- Objetivo

Estudiar el impacto que tiene en el rendimiento de una aplicación el modo en que se acceden a los datos dependiendo de cómo estén éstos almacenados en memoria.

- Diseño del experimento

Se van a realizar dos experimentos, uno para cada forma de almacenamiento (por filas-columnas o bloques, como se mostró en la Figura 4.1). Las pruebas de rendimiento se realizarán tomando la suma de matrices como algoritmo de prueba, comparando los distintos modos de acceso a los datos de un Tile de dos dimensiones. Las tres formas posibles de recorrer los datos son:

- Recorrer una matriz por filas-columnas: Esto es recorrer la matriz de la forma típica con dos bucles anidados, para ir accediendo a los elementos mediante una fórmula de acceso a un determinado elemento (i,j) del Tile.

```
void sumMat( HitTile_double matResult, HitTile_double matA, HitTile_double matB ) {
    int ind[2];
    hit_tileForDimDomain ( matResult, 0, ind[0] ) {
        hit_tileForDimDomain ( matResult, 1, ind[1] ) {
            hit_tileElemAtNoStride( matResult ,2 , ind[0], ind[1]) =
                                    hit_tileElemAtNoStride( matA, 2, ind[0], ind[1] ) +
                                    hit_tileElemAtNoStride(matB, 2 ,ind[0], ind[1]);
}}}
```

Figura 5.1: Suma de matrices recorriendo por filas-columnas

- Recorrer matriz por bloques accediendo por coordenadas de bloque: Significa recorrer la matriz con cuatro bucles anidados, de manera que se accede directamente con una fórmula a cada elemento (bloqx,bloqy,i,j).

```
void sumMat( HitTile_double matResult, HitTile_double matA, HitTile_double matB ) {
    int indBloq[2], numBloq[2], ind[2];
    numBloq[0] = hit_tileDimCard( matA ,0) / hit_tileDimSizeBlock( matA, 0);
    numBloq[1] = hit_tileDimCard( matA ,1) / hit_tileDimSizeBlock( matA, 1);
    //Iterando sobre los bloques
    for(indBloq[0]=0; indBloq[0]<numBloq[0]; indBloq[0]++){
        for(indBloq[1]=0; indBloq[1]<numBloq[1]; indBloq[1]++){
            //Iterando dentro de los bloques
            for(ind[0]=0; ind[0]<hit_tileDimSizeBlock( matA, 0); ind[0]++){
                for(ind[1]=0; ind[1]<hit_tileDimSizeBlock( matA, 1); ind[1]++){
                hit_tileElemAtBloq(matResult,2,indBloq[0],indBloq[1],ind[0],ind[1])=
                            hit_tileElemAtBloq( matA,2,indBloq[0],indBloq[1],ind[0],ind[1] ) +
                            hit_tileElemAtBloq( matB,2,indBloq[0],indBloq[1],ind[0],ind[1] ) ;

}}}}}
```

Figura 5.2: Suma de matrices recorriendo por coordenadas de bloque

- Recorrer matriz por bloques, obteniendo un Tile de bloque: Consiste en recorrer la matriz con cuatro bucles anidados, para en primer lugar ir obteniendo los Tile de cada bloque(bloqx,bloqy), y a continuación ir recorriendo internamente los bloques(i,j).

```
void sumMat( HitTile_double matResult, HitTile_double matA, HitTile_double matB ) {
    int indBloq[2],numBloq[2],ind[2];
    numBloq[0] = hit_tileDimCard( matA ,0) / hit_tileDimSizeBlock( matA, 0);
    numBloq[1] = hit_tileDimCard( matA ,1) / hit_tileDimSizeBlock( matA, 1);
    HitTile_double blockA,blockB,blockResult;

    //Iterando sobre los bloques
    for(indBloq[0]=0; indBloq[0]<numBloq[0]; indBloq[0]++){
        for(indBloq[1]=0; indBloq[1]<numBloq[1]; indBloq[1]++){
            hit_bloqTileGetBlock( &blockA, &matA, indBloq[0], indBloq[1] );
            hit_bloqTileGetBlock( &blockB, &matB, indBloq[0], indBloq[1] );
            hit_bloqTileGetBlock( &blockResult, &matResult, indBloq[0], indBloq[1] );
            //Iterando dentro de los bloques
            hit_tileForDimDomain ( blockA, 0, ind[0] ) {
                hit_tileForDimDomain ( blockA, 1, ind[1] ) {
                hit_tileElemAtNoStride(blockResult,2,ind[0],ind[1])=
                        hit_tileElemAtNoStride(blockA,2,ind[0],ind[1])+
                        hit_tileElemAtNoStride( blockB,2,ind[0],ind[1] ) ;
}}}}}
```

Figura 5.3: Suma de matrices recorriendo por bloques, obteniendo un Tile de bloque

El experimento se ha llevado a cabo en la máquina Chimera lanzando un solo proceso para un tamaño de matriz de 16384x16384. Se ha probado variando la selección del tamaño de bloque desde 1 hasta 64 en potencias de dos, para que los datos alineen con líneas de caché y ver así la influencia del tamaño del bloque en el rendimiento dependiendo del modo de acceso y almacenamiento de los datos.

5.2.2 Diferentes versiones de implementación del código LU

- Objetivo

Determinar las ventajas e inconvenientes de la propuesta de almacenamiento de datos comparando la nueva versión de implementación en Hitmap, sobre la tradicional de Hitmap y la original en MPI en una aplicación de ejemplo con la que se ya se ha trabajado anteriormente: LU.

- Diseño del experimento

El experimento se ha llevado a cabo en la máquina Heracles. Se han ido ejecutando la diferentes versiones de la aplicación LU: LU en Hitmap con almacenamiento de datos por bloques, LU en Hitmap con almacenamiento de datos por filas columnas y partición de bloques mediante HitBlockTile y LU con MPI. Se ha seleccionado una matriz de tamaño 4096x4096, que se ejecutado con diferente número de procesos, de 1 a 64 y variando el tamaño de bloque desde 4 a 512.

5.2.3 Experimento del tamaño de bloque en LU

- Objetivo

Determinar la relación entre el rendimiento y el tamaño de bloque, en función de las características de la plataforma de ejecución.

- Diseño del experimento

Tomando como algoritmos de prueba el de factorización LU, y posterior resolución de un sistema de ecuaciones, se han realizado pruebas de rendimiento en distintas máquinas, con diferentes características. Para ello, se ha seleccionado un tamaño de matriz que nos permita obtener un tiempo estable y significativo, en este caso se han tomado matrices de 4096x4096. Se han ido tomando medidas variando el tamaño de bloque elegido para el particionado, siempre eligiendo potencias de dos para que los datos alineen con líneas de caché, obteniendo mejor rendimiento, y variando el número de procesos en uso, desde uno hasta el máximo disponible. Se ha elegido realizar el experimento en dos máquinas con diferentes características: Heracles y Chimera.

También se ha hecho un análisis de la aplicación LU, obteniendo por separado el tiempo dedicado a cada algoritmo, y separando el tiempo dedicado a computación y comunicación, para así, intentar entender mejor los resultados obtenidos.

5.3 Resultados

5.3.1 Experimento suma de matrices con diferente modo de almacenamiento

5.3.1.1 Almacenamiento Filas-Columnas

En la figura 5.4 se puede ver el tiempo de ejecución en relación al tamaño de bloque para el algoritmo de suma de matrices con un tamaño de matriz de 16384x16384, con los datos almacenados en memoria por filas-columnas.

Se pueden observar los siguientes resultados:

- Recorrer por filas–columnas es la que mejor rendimiento proporciona para este modo de almacenamiento. Los elementos se recorren de forma contigua en memoria. Se puede ver que la gráfica no varía en función del tamaño del bloque, ya que no interviene en las fórmulas de acceso a los elementos.

$$ElemAt(pos1, pos2) = pos1 \cdot n + pos2$$

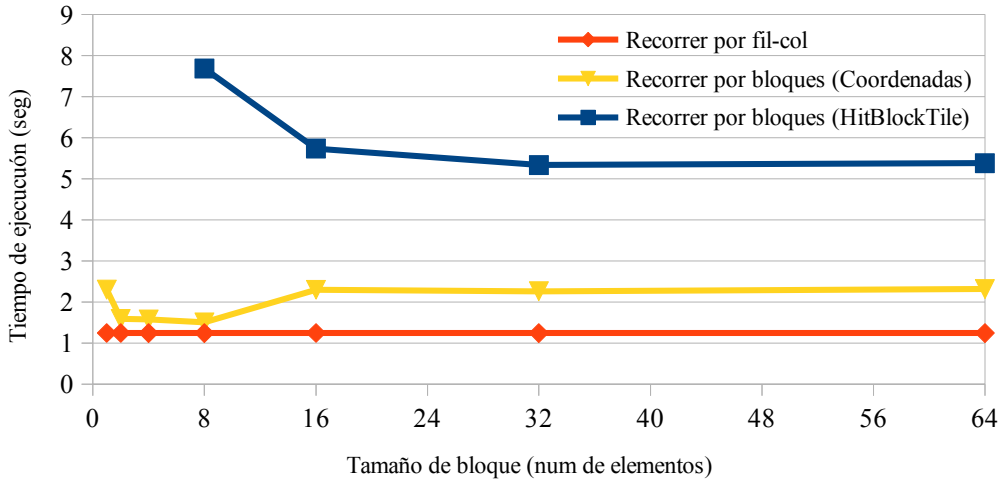


Figura 5.4: Suma de matrices 16384x16384 con almacenamiento por filas-columnas

- Al recorrer la matriz por bloques accediendo mediante coordenadas consecutivas, se puede ver que, como era de esperar, se obtiene un rendimiento peor que recorriendo por fila-columna, aunque limitado, especialmente en el caso de ciertos tamaños de bloque pequeños (entre 2 y 4). Esto se debe, en principio, a dos factores. Uno, que la fórmula de acceso es un poco más compleja al intervenir el tamaño del bloque en el cálculo de la posición de memoria del elemento.

$$ElemAtBlock(bloq1, bloq2, pos1, pos2) = n \cdot tb1 \cdot bloq1 + tb2 \cdot bloq2 + pos1 \cdot n \cdot tb1 + pos2$$

  Y dos, que al no estar los elementos contiguos en memoria por filas (sino por bloques), los fallos de caché asociados a tener que hacer saltos en memoria, cargando y descargando datos puede afectar al rendimiento. No hay que olvidar que el método de almacenamiento consecutivo (sin bloques) está especialmente pensado para favorecer la contigüidad de los datos al acceder por filas.

- Recorrer por bloques usando la tradicional forma de recorrer y realizar las particiones en bloques (HitBlockTile) es la peor opción para este caso con diferencia. Si desglosamos el tiempo de ejecución en el utilizado por cada parte del programa, como se puede ver en la gráfica 5.5, es posible observar que aunque el tiempo empleado en realizar la operación de suma de matrices es mas o menos similar que en los casos anteriores, la enorme pérdida de rendimiento se produce al reservar memoria las estructuras de datos.

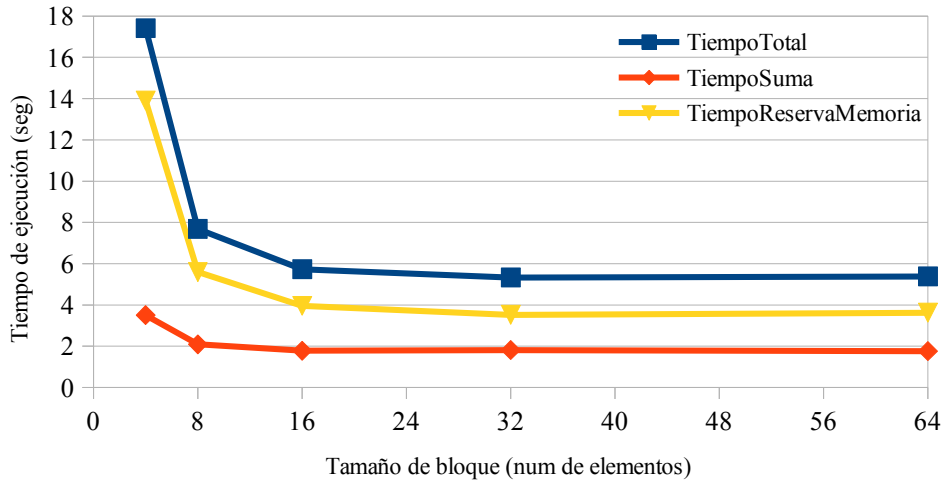


Figura 5.5: Suma de matrices 16384x16384 con implementación tradicional de Hitmap mediante HitBlockTile(Tile de punteros a bloques)

Se puede observar que el tiempo de reserva de memoria de las 3 matrices (las dos de suma, y la de resultados) consume entre el 65-80% del tiempo total, siendo mayor cuanto menor es el tamaño del bloque. La causa de esta pérdida de rendimiento para tamaños de bloque pequeños encuentra su explicación en que el la implementación en Hitmap original de la partición en bloques (ver sección 3.3.2) en la que cada bloque se va reservando memoria para cada bloque, por lo que para bloques pequeños en relación al tamaño de la matriz, se tienen que realizar muchas llamadas a la función de reserva, tantas como bloques contenga.

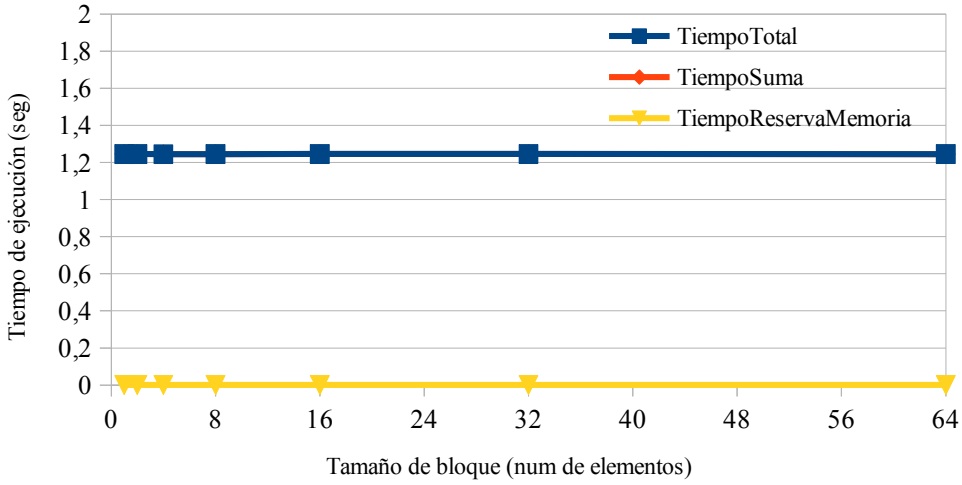


Figura 5.6: Suma de matrices 16384x16384 por fila-columnas con nueva implementación

Con la nueva propuesta de almacenamiento de datos por bloques, los elementos de un Tile ya no son punteros a los bloques, si no que es un Tile de elementos normal y se reserva memoria para toda la matriz en un solo paso independientemente del tamaño del bloque elegido. Podemos ver en la Figura 5.6 que el tiempo de reserva de memoria siempre es casi nulo (0,00002%), por lo que el tiempo total se solapa con el tiempo dedicado a la suma.

5.3.1.2 Almacenamiento en Bloques

En la figura 5.7 se puede ver el tiempo de ejecución en relación al tamaño de bloque para el algoritmo de suma de matrices con un tamaño de matriz de 16384x16384, con los datos almacenados en memoria por bloques.

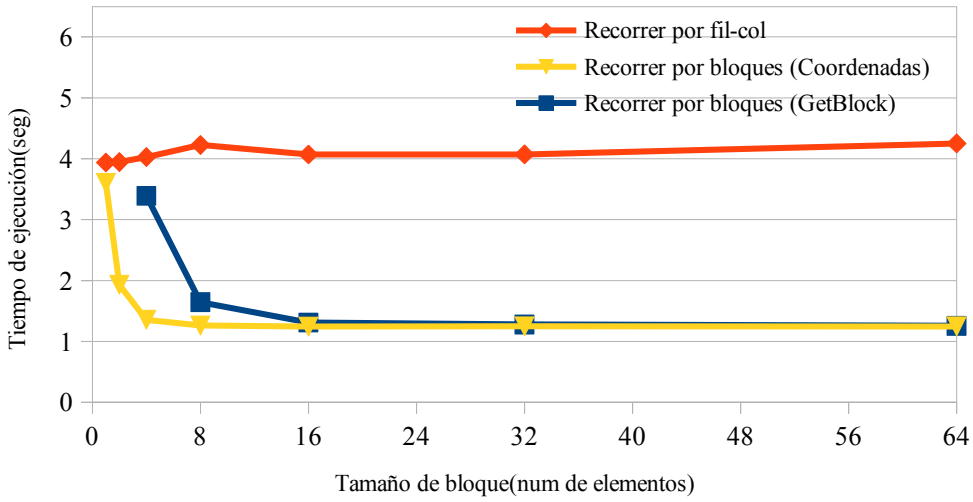


Figura 5.7: Suma de matrices 16384x16384 con almacenamiento por bloques

Se pueden observar los siguientes resultados:

- Recorrer por filas-columnas da un rendimiento penoso siempre para este tipo de almacenamiento. En mayor parte es debido a que al estar almacenados los elementos por bloques, la fórmula para calcular dónde está un índice de la matriz es muy compleja comparando con las otras en cuanto a operaciones y emplea mucho tiempo.

$$ElemAt(pos1, pos2) = DivEnt\left(\frac{pos1}{tb1}\right) \cdot n \cdot tb1 + RestDiv\left(\frac{pos1}{tb1}\right) \cdot tb$$
$$+ DivEnt\left(\frac{pos2}{tb2}\right) \cdot tb1 \cdot tb2 + RestDiv\left(\frac{pos2}{tb2}\right)$$

  También, parte de la pérdida de rendimiento puede ser por los fallos cachés asociados a tener que hacer saltos en memoria.

- Recorrer por la matriz por bloques accediendo por coordenadas de bloque, es la que va mejor para este tipo de almacenamiento, la fórmula para localizar un elemento de la matriz accediendo por bloques es relativamente sencilla:

$$ElemAtBlock(bloq1, bloq2, pos1, pos2) = n \cdot tb1 \cdot bloq1 + bloq2 \cdot tb1 \cdot tb2 + pos1 \cdot tb2 + pos2$$

  Al estar los elementos los elementos de un bloque contiguos en memoria, no se tienen que realizar saltos en memoria.

- Recorrer la matriz, por bloques, obteniendo la dirección de comienzo del bloque y luego recorriéndolo de forma continua, vemos que nos da unos tiempos buenos, al igual que recorrerlo por coordenadas de bloque. Aquí el tiempo se pierde en primer lugar en la función que crea un nuevo Tile con la dirección de memoria apuntando al comienzo del bloque, y después en la macro de acceso a los elementos del bloque. Esta fórmula de acceso es muy sencilla ya que se trata de un acceso a elementos que están contiguos.

5.3.2 Experimento LU comprando rendimiento con diferentes implementaciones

En la figura 5.8 se puede ver el tiempo de ejecución en relación al tamaño de bloque para las diferentes implementaciones del sistema de almacenamiento de bloques. El tradicional de Hitmap (jerárquico) y con nuestra nueva propuesta, para un tamaño de matriz de 4096x4096. Se añade también el rendimiento de un código de referencia desarrollado y optimizado directamente en MPI para demostrar que los métodos de almacenamiento de Hitmap no incurren en penalizaciones relevantes.

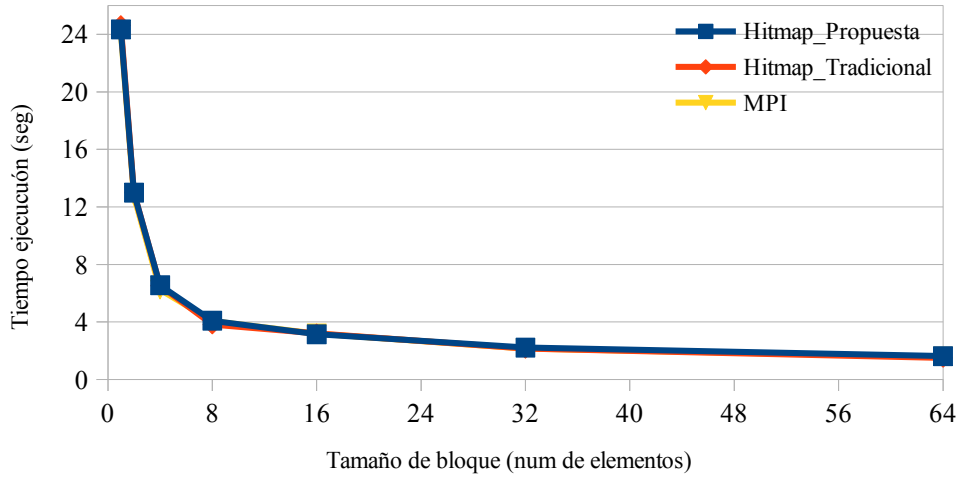


Figura 5.8: Rendimiento diferentes versiones de la aplicación LU para una matriz 4096x4096 con 64 procesos

Se pueden observar los siguientes resultados:

Se puede observar que las dos propuestas (la tradicional jerárquica, y la nueva) obtienen resultados similares. Dada la gran cantidad de computación global que requiere esta aplicación, el efecto relativo de la pérdida de rendimiento de la inicialización de estructuras con el método tradicional se minimiza y pasa desapercibido.

La implementación de estos algoritmos ha sido adaptada de la versión diseñada para el sistema tradicional, que está basada en obtener la dirección y estructura de un bloque antes de acceder sistemáticamente a sus datos. En la estructura jerárquica, los datos de cada bloque también están contiguos en memoria, por lo que después de el bloque, la mayor parte de las operaciones de los algoritmos de LU y resolución del sistema son similares en cuanto a la gestión de la memoria.

Esto nos muestra que la nueva forma de almacenamiento, que simplifica enormemente la gestión de la memoria en una única zona, y puede permitir el diseño de algoritmos con acceso directo a los datos de un bloque sin necesidad de obtener antes su posición y estructura, no produce penalizaciones a nivel global debido a sus métodos de acceso.

5.3.3 Experimento tamaño de bloque en LU

En la figura 5.9 se muestra el tiempo de ejecución en relación al tamaño de bloque elegido para el algoritmo de LU, con un tamaño de matriz de 4096x4096.

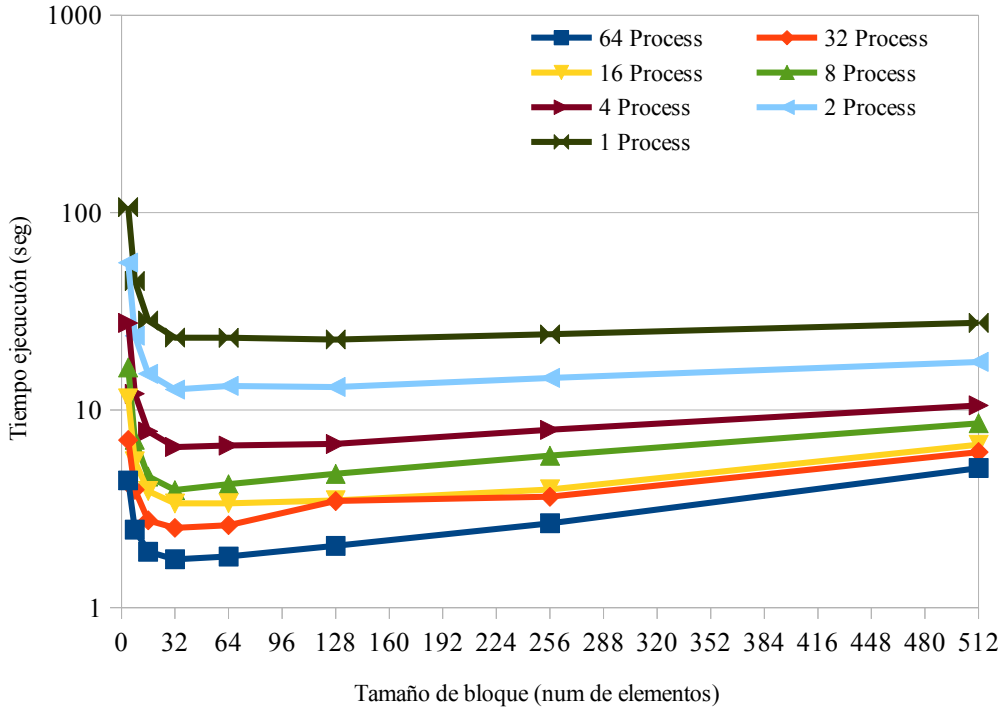


Figura 5.9: Rendimiento resolución sistema LU para una matriz de 4096x4096 en Heracles

De esta figura 5.9, se puede observar que el rendimiento óptimo de la aplicación LU siempre se obtiene, independientemente del número de procesos, eligiendo un tamaño de bloque de 32. En la siguiente figura 5.10, en cambio, se observa que el rendimiento óptimo de la aplicación LU, en la máquina Chimera, se obtiene eligiendo un tamaño de bloque de 64.

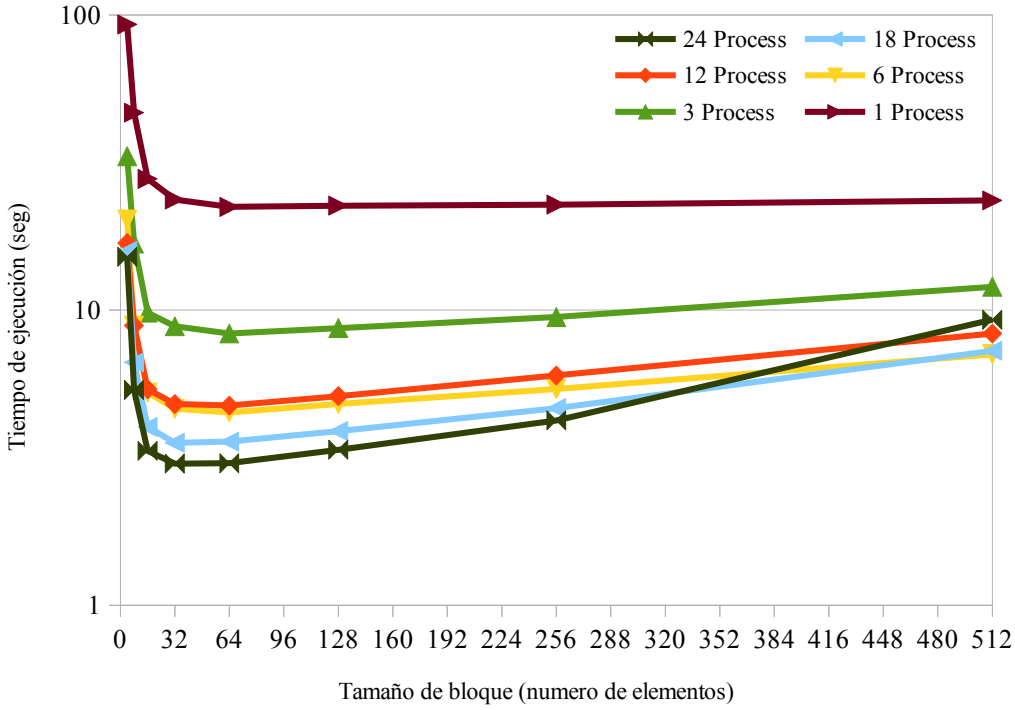


Figura 5.10: Rendimiento resolución sistema LU para una matriz de 4096x4096 en Chimera

De estas dos gráficas (Figura 5.9 y 5.10), se puede verificar que existe una relación entre el rendimiento obtenido (tiempo de ejecución), y el tamaño en que se realizan las particiones de bloques. Se puede deducir, que para cada plataforma de ejecución, existe un punto de inflexión que se puede determinar de forma experimental, en el que en rendimiento de la aplicación es óptimo. En este caso, un tamaño de bloque de 32 elementos para Heracles y de 64 para Chimera.

A continuación se analiza con más en detalle la aplicación LU, separando los tiempos de ejecución de los tres algoritmos en que se compone: inicialización de la matriz A, factorización LU de la matriz A y resolución del sistema de ecuaciones, para ver cómo afecta la elección del tamaño del bloque a cada algoritmo.

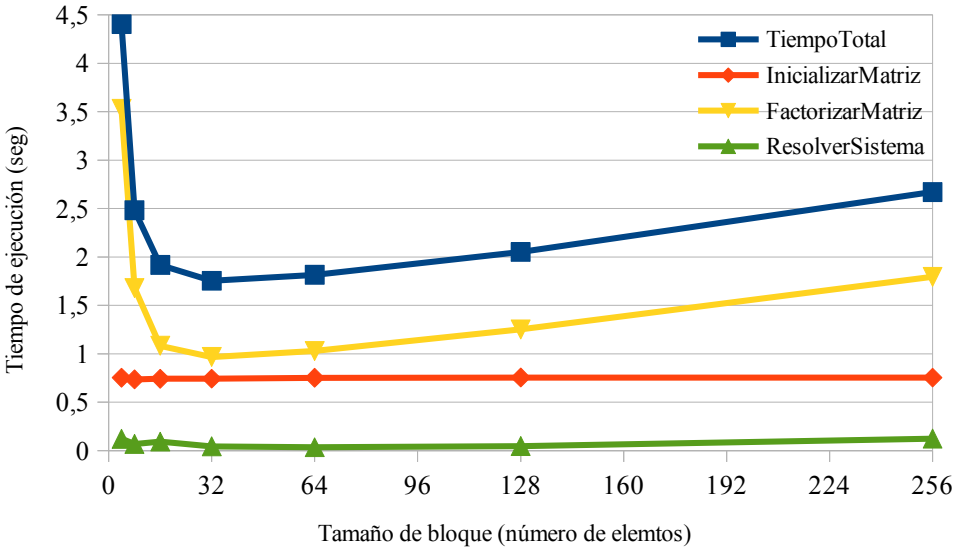


Figura 5.11: Rendimiento resolución sistema LU para una matriz de 4096x4096 separando el tiempo de ejecución de cada algoritmo con 64 procesos en Heracles

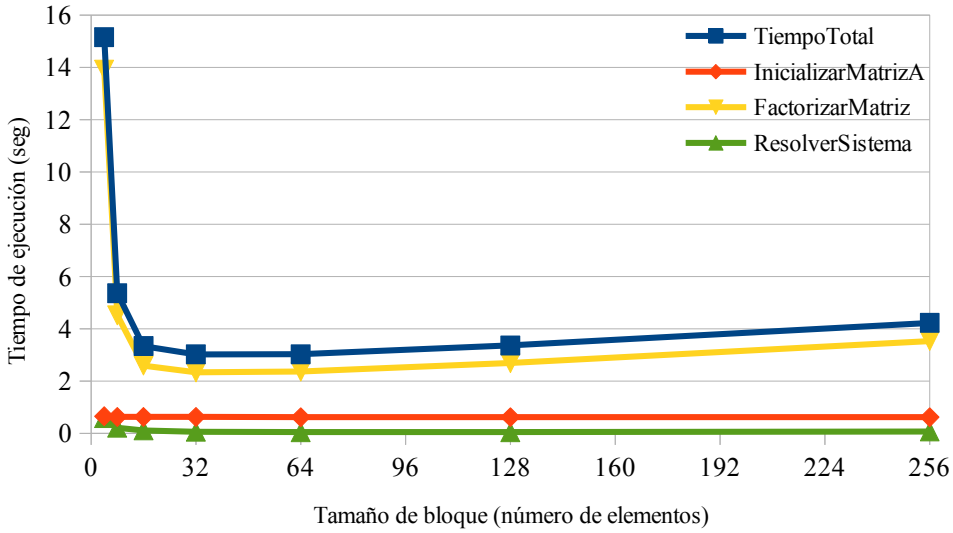


Figura 5.12: Rendimiento resolución sistema LU para una matriz de 4096x4096 separando el tiempo de ejecución de cada algoritmo con 24 procesos en Chimera

De las figuras 5.11 y 5.12 se obtiene que la mayor parte del tiempo se consume en el algoritmo de factorización LU, en el que el punto de rendimiento óptimo coincide con el de la aplicación completa,

32 para Herecles y 64 para Chimera. Otra parte considerable del tiempo se pierde en la inicialización de las matrices, pero este tiempo es prácticamente constante, no le afecta el tamaño de bloque elegido. Por último, el algoritmo de resolución del sistema de ecuaciones consume un tiempo muy pequeño, en relación con el total de la aplicación y en esta escala no se puede apreciar si le afecta o no la elección del tamaño de bloque. Así que ahora se va a analizar cada algoritmo por separado además dividiendo los tiempos dedicados a computación y comunicación.

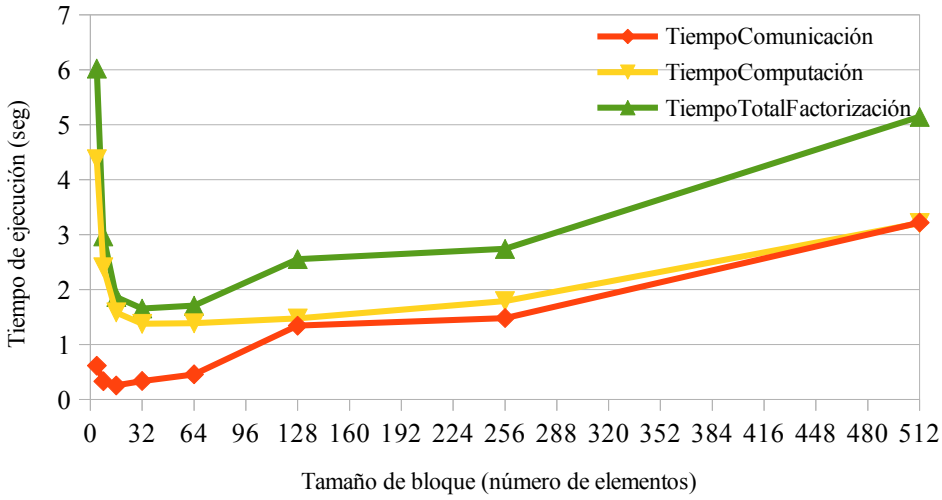


Figura 5.13: Rendimiento algoritmo factorización LU para una matriz de 4096x4096 en Heracles con 32 procesos

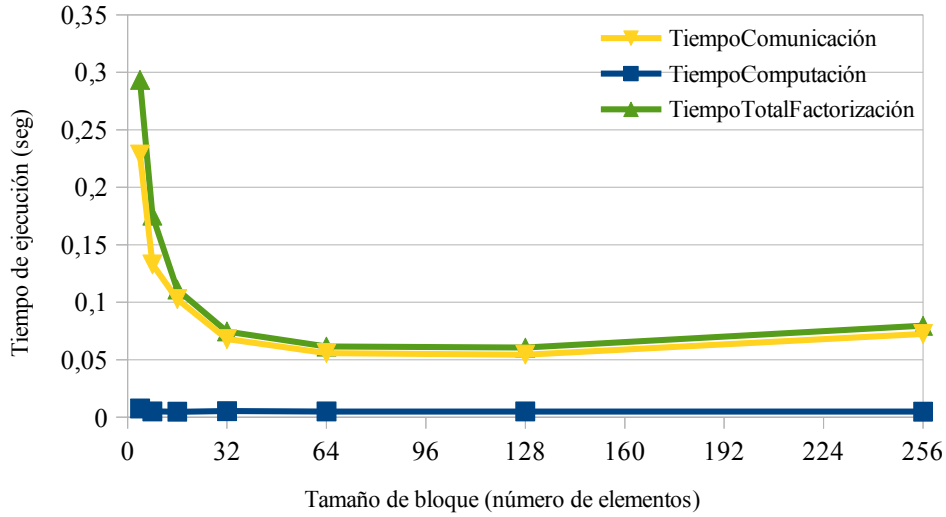


Figura 5.14: Rendimiento algoritmo de resolución del sistema de ecuaciones para una matriz de 4096x4096 en Heracles con 32 procesos

Observando la figura 5.13, vemos que para el algoritmo de factorización se confirma que el punto de rendimiento óptimo coincide con el de la aplicación completa(32), por ser el algoritmo que tiene el mayor peso en la aplicación. Pero en el caso de la resolución del sistema de ecuaciones, este punto óptimo se alcanza para un tamaño de bloque más grande, 128.

Se deduce que el tamaño de bloque para el que se consigue el mejor rendimiento varía dependiendo de las características de cada algoritmo concreto y sus patrones de comunicación y acceso a memoria.

5.4 Resumen del capítulo

El estudio demuestra que cada forma de acceso (por filas, por bloques, etc.) se beneficia de una forma de almacenamiento diseñada específicamente. También se demuestra que complejas estructuras jerárquicas complican la gestión de la memoria (de forma notable en el rendimiento para aplicaciones con carga computacional moderada), sin obtener mejores rendimientos que formas directas de almacenamiento que utilizan sencilla aritmética en sus métodos de acceso.

Los resultados muestran también que no es posible determinar la relación entre el rendimiento y el tamaño de bloque de una forma sencilla, ya que depende de una combinación de factores que implican características de la plataforma y del algoritmo.

Capítulo 6. Conclusiones

En este capítulo se detallan las conclusiones a las que se llega con los resultados obtenidos. Además se incluyen una serie de posibles vías de continuación del trabajo realizado.

6.1 Conclusiones

En este Trabajo Fin de Grado se han cumplido los siguientes objetivos:

- Se ha realizado un estudio de Hitmap y aplicaciones que necesiten que necesiten particionado y almacenamiento bloque-cíclico, en concreto la aplicación de resolución de sistemas de ecuaciones lineales mediante factorización LU.
- Se ha diseñado e implementado en la librería Hitmap un método de almacenamiento de datos en memoria basado en bloques.
- Se ha estudiado de forma experimental el impacto que tiene en el rendimiento de una aplicación el modo en que se acceden a los datos dependiendo de cómo estén almacenados en memoria, ya sea por filas-columnas o por bloques.
- Se ha comprobado que el nuevo método de almacenamiento propuesto y desarrollado para Hitmap simplifica enormemente la gestión de memoria sin perder eficiencia en los accesos. Al utilizar métodos homogéneos de acceso a los datos basados en aritmética sencilla, estos permiten acceder a datos concretos de un bloque por coordenadas con la misma eficiencia que recuperando la estructura de un bloque previamente. El estudio realizado valida la nueva propuesta como una alternativa muy interesante al método anterior de almacenamiento jerárquico, ya que permitirá diseñar algoritmos que utilicen directamente coordenadas mixtas de bloque y elemento, sin necesidad de recuperar primero la estructura del bloque.
- Se estudiado de forma experimental la relación entre el rendimiento y el tamaño de bloque, en función de las características de la plataforma de ejecución para aplicaciones que emplen particionado de datos en bloques como LU.

Se han extraído las siguientes conclusiones experimentales tras la realización del proyecto:

- Se ha concluido cuál la mejor manera de recorrer y acceder a los datos dependiendo de cómo estén almacenados en memoria:

  - Almacenamiento Filas-Columnas: Se ha visto que la mejor opción para esta caso de almacenamiento era recorrer datos accediendo por filas-columnas.
  - Almacenamiento Bloques: Para este otro modo de almacenamiento, se ha visto que la mejor opción era recorrer por bloques, ya sea accediendo directamente por coordenadas de bloque, o obteniendo en un Tile el bloque y a continuación recorriendo dicho Tile.

  En ambos casos se pude concluir que la mejor forma de recorrer los datos es según están estos almacenados en memoria ya que siempre se accederían a datos que se encuentran contiguos en memoria.
- La tradicional forma de partir un Tile en bloques con una jerarquía de punteros, se ha visto que consumía mucho más tiempo en su reserva de memoria, que la forma propuesta con un sólo Tile. Sobre todo para tamaños de bloque pequeños en relación con el tamaño de la matriz al tener que reservar memoria para cada bloque de forma independiente.
- Se verifica que existe una relación entre el rendimiento obtenido (tiempo de ejecución) y el tamaño en que se realizan las particiones de bloques. Pero no se puede extraer una conclusión directa relacionada con las características de la plataforma de ejecución. Lo que sí que se puede extraer es que existe un punto de inflexión, que se puede determinar de forma experimental, para cada algoritmo, en el que en rendimiento es óptimo.

6.2 Trabajo futuro

- Estudiar otros tipos de almacenamiento más específicos (por bandas, en zig-zag, etc.).
- Estudiar otras aplicaciones que puedan aprovecharse del almacenamiento bloque-cíclico, e implementaciones directas que utilicen coordenadas mixtas de bloque y elemento.
- Intentar modelar el comportamiento de estas aplicaciones y los parámetros de hardware necesarios para determinar tamaños de bloque candidatos automáticamente sin necesidad de realizar un estudio empírico.

Bibliografía

[1] **A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D.R. Llanos.** *An extensible system for multilevel automatic data partition and mapping.* IEEE TPDS, 2013.

[2] **C. de Blas Cartón, A. Gonzalez-Escribano and D.R. Llanos.** *Effortless and Efficient Distributed Data-partitioning in Linear Algebra*. IEEE ICHPCC, 2010

[3] **Gough, B.** *Una introducción a GCC.* Segunda edición, Network Theory Limited, 2011

[4] **Francisco Almeida, Domingo Giménez, José Miguel Mantas, Antonio M. Vidal.** *Introducción a la programación paralela*. Paraninfo. Pag 420- 446. ISBN: 978-84-9732-674-2

[5] **Barry Wilkinson, Michael Allen.** Parallel Programming. *Thechniques and Applications Using Networked Workstations and Parallel Computers.* Prentice-Hall, Inc. 1999. Pag 313-317. ISBN: 0-13-671710-1

[6] **Gene Golub, James M. Ortega.** *Scientific Computing. An Introdution with Parallel Computing.* Academic Press, Inc. 1993. Pag 215-230. ISBN: 0-12-289253-4

[7] **Michael J.Quinn.** *Parallel computing. Theory and Practice.* McGRAW-HILL, INC. 1994. 2008. Pag 217- 237. ISBN: 0-07-051294-9

[8] **Michael T. Heath.** *Scientifig Computing. An Introduction Survey.* McGRAW-HILL INC. 1997. Pag 35-41. ISBN:0-07-115336-5

[9] **David A. Patterson, John L. Hennessy.** *Estructura y diseño de computadores. La interfaz hardware/software.* Reverté. 2011. Pag 450-525. ISBN:978-84-291-2620-4.

[10] http://www.mpi-forum.org/

[11] **Ian Sommerville.** *Ingeniería del Software*. Séptima edición. Pearson Education S.A. 2005. Pag 63-64 ISBN:84-7826-074-5

[12] **Yair Censor, Stavros A. Zenios** *Parallel Optimization: Theory, Algorithms, and Applications (Numerical Mathematics and Scientific Computation)* Oxford University Press. 1998. Pag 12-24. ISBN: 978-0195100624

Anexo A. Contenido del CD-ROM

La estructura del CD-ROM se divide en dos carpetas:

- Documentacion: Incluye la presente memoria del Trabajo Fin de Grado en formato PDF.
- Hitmap: Incluye el código de la librería de Hitmap dónde se ha integrado nuestra propuesta presentada en este Trabajo Fin de Grado. Los directorios fundamentales son siguientes:
  - include: Ficheros de cabecera de la librería Hitmap
  - src: Códigos fuente de la librería Hitmap
  - examples: Códigos de ejemplos empleados en los casos de estudio.
    - LU: Aquí se encuentran los códigos de la aplicación de resolución de sistemas mediante factorización LU.
    - SumMat: Aquí se encuentran los diferentes códigos del algoritmo de suma de matrices.