Paralelización de un sistema de compresión de imágenes de satélite

Adrián Marín Villar

Resumen– Debido al gran avance en tecnologías de captación de imágenes, cada vez, los dispositivos que implementan estas tecnologías permiten obtener imágenes y vídeos de alta resolución con el coste de ocupar, cada vez más, un espacio considerable en nuestros dispositivos. Debido a este hecho, existen diversos algoritmos que permiten obtener la misma imagen con una calidad menor debido a las pérdidas que genera la compresión de esta para poder minimizar el espacio que ocupa. Esta situación provoca el afrontamiento del *trade-off* entre calidad y coste de almacenamiento referente al espacio digital. No obstante, otro aspecto a considerar es el tiempo de compresión el cual, en imágenes de alta resolución, puede llegar a ser elevado. En este artículo se discutirán los resultados obtenidos mediante el uso de *multi-threading* en un compresor, de las imágenes obtenidas comparándolas con la ejecución del mismo de forma secuencial.

Palabras clave– *Java*, paralelización, procesado de imágenes, teoría de la comunicación, *compresión*, *descompresión*, *satélites*, *multi-threading*

1 Introducción

Actualmente, las imágenes captadas vía satélite utilizan un modelo de procesado secuencial, es decir, dicha imagen a tomar es interpretada como una matriz escaneada fila por fila, generando al finalizar, una matriz de datos donde cada celda contiene un dato que es conocido como *sample* (muestra), en caso de que cada celda de la matriz represente un color, o píxel, el cual es un vector de n muestras que forman un único color, como es el caso de la codificación RGB (*Red Green Blue*).

$X_{0,0,0}$	$X_{0,1,0}$	$X_{0,2,0}$	$X_{0,511,0}$
$X_{1,0,0}$	$X_{1,1,0}$	$X_{1,2,0}$	$X_{1,3,0}$
$X_{2,0,0}$	$X_{2,1,0}$	$X_{2,2,0}$	$X_{2,3,0}$
$X_{3,0,0}$	$X_{3,1,0}$	$X_{3,2,0}$	$X_{511,511,0}$

Fig. 1: Representación de una imagen en forma matricial.

A medida que la tecnología de las cámaras avanza, estas proporcionan una resolución cada vez mayor, lo que implica un aumento del tamaño de las imágenes que captan. Debido a este hecho, se espera que estos sistemas incorporen un compresor que permita almacenar dichas imágenes con un tamaño reducido al original ya sea con pérdida de calidad como puede ser el formato *JPEG* o, en caso de requerir que la imagen conserve la calidad original como es el caso de la toma de radiografías, el formato *RAW*.

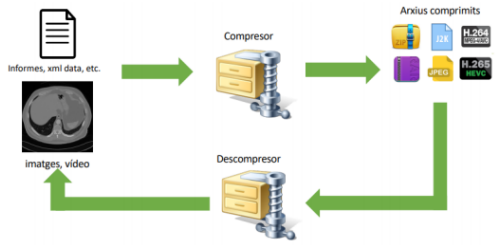

Fig. 2: Esquema de compresión y descompresión de imágenes.

Sin embargo, dicho proceso de captación de imágenes necesita disponer de la matriz completa que conforma la figura para realizar el proceso de compresión debido a que las muestras adyacentes a cualquier muestra de la matriz se encuentran correladas entre si, es decir, existe una dependencia entre ellas. Por lo tanto, la compresión de dichas imágenes puede llegar a ser muy lenta dependiendo de su resolución. Para poder optimizar el compresor, es posible la implementación de hilos (*threads*) para minimizar el tiempo de compresión. No obstante, como se ha comentado anteriormente, se necesitan todas las muestras para poder realizar la compresión. Para solventar esta problemática, es posible dividir el proceso de compresión

• E-mail de contacte: adrian.marinv@e-campus.uab.cat
• Mención realizada: Tecnologías de la Información
• Trabajo tutoritzado por: Joan Bartrina Rapesta (dEIC)
• Curso 2019/20

en *N* hilos repartiendo los datos de la imagen obtenida, comprimirlos y posteriormente volver a unir los datos de cada *thread* en sus respectivas posiciones de la matriz obtenida originalmente.

El esquema de paralelización propuesto implica que ningún hilo de ejecución dispondrá de toda la matriz de datos con lo cual, para obtener un valor aproximado de la siguiente muestra a procesar, se empleará un predictor, el cual permite estimar, dependiendo de los datos anteriores de los cuales disponga dicho *thread*, el valor de la próxima muestra a comprimir.

En este artículo se realizará un análisis del entorno, es decir, estudiar el comportamiento del compresor y el descompresor. A continuación, se describirá un planteamiento para aplicar la paralelización a dicho entorno y, para finalizar, se analizarán los resultados obtenidos de las imágenes procesadas por dicho esquema paralelizado.

2 ESTADO DEL ARTE

2.1 Paralelización del sistema de compresión

Para poder comprender el comportamiento del esquema reflejado en la figura 2, cal entender la naturaleza de los datos a analizar. En este entorno, se trabaja con el concepto de imagen a nivel matricial tal y como se muestra en la figura 1. Una vez asimilado el concepto de imagen que se va a tratar, hay que conocer la estructura que dicho fichero sigue como los meta-datos de esta. Conociendo los meta-datos, pasamos al siguiente bloque de la estructura el cual está formado por las muestras de la imagen. No obstante, dichas muestras pueden ser codificadas de distintas formas, ya sea a partir de su formato de almacenamiento (*BigEndian/LittleEndian*) o el rango de valores que pueden alcanzar. En caso de ser valores con signo (*signed*), sus valores pueden estar comprendidos en el rango [0, 65.536] y, en caso de ser sin signo (*unsigned*), sus valores pertenecen al rango [-32.767, 32.768].

Una vez analizado la estructura de los datos que van a ser procesados y que hay que tener en cuenta a la hora de implementar la paralelización del sistema, se procede a interiorizar en el concepto de compresión y paralelización. El concepto de compresión, desde el punto de vista ideal, es obtener la misma información con la mínima cantidad de datos posible [1] [2]. No obstante, este hecho no implica que no exista un límite de compresión de la información, aquí es donde aparece el concepto de *entropía*. La entropía determina la cantidad de datos mínima, expresada en bits, que se requieren para obtener una información concreta y es calculada mediante la siguiente ecuación:

$$H(x) = -\sum_{i} p(x_i) \cdot log_2 p(x_i) \qquad (1)$$

Como se refleja en (1), la entropía depende de la probabilidad que tiene un símbolo, o en este caso, una muestra en aparecer en la imagen. Para obtener la mínima entropía, es decir, la mínima cantidad de bits necesarios para representar una información, en este caso imágenes, cal disponer de todos los datos que se desean comprimir. En este proyecto no será posible alcanzar dicha entropía debido a que al paralelizar el proceso de compresión, los *threads* no dispondrán de toda la información de la imagen, lo cual lleva al concepto conocido como compresión con pérdidas como por ejemplo el formato *jpeg*.

No obstante, en (1) se considera que todos los píxeles están incorrelados entre si, es decir, son independientes. Ya que este hecho no se corresponde con nuestro entorno, se considera que los píxeles que sean adyacentes a un píxel concreto guardan cierta dependencia, dicho de otra forma, son dependientes entre si. Por lo tanto, es posible reformular (1) para obtener una entropía condicional a partir de los píxeles adyacentes al que se desea estimar.

$$H(x|y) = -\sum_{i}\sum_{j} p(y_j, x_i) \cdot log_2 p(x_j|y_i) \qquad (2)$$

Comparando (1) y (2), la entropía condicionada proporcionará un valor menor debido a que al utilizar píxeles adyacentes, la incertidumbre de los datos disminuye.

A continuación, una vez entendidos los conceptos presentados anteriormente, cal indagar sobre el uso de *threads* para poder alcanzar el objetivo de este proyecto, la paralelización del sistema. No obstante, no es suficiente con conocer cómo emplear dicho mecanismo sino, a partir de conocer su funcionamiento, extrapolar el concepto de compresión basándose en la estructura de los datos, en este caso, imágenes, conseguir enlazar los resultados de los distintos hilos de ejecución que se utilicen para obtener un resultado idéntico o casi-idéntico (debido a la predicción de los valores de las muestras) al original.

2.2 Proceso de paralelización

A causa de la paralelización, los *threads* no dispondrán de toda la información de la imagen a la hora de comprimir sus respectivos fragmentos de los datos originales. Debido a este hecho, se emplea un predictor que, a partir de las muestras adyacentes a una, permite obtener el posible valor de una *sample* de la cual, en ese mismo instante de ejecución, no se dispone de ella. Por lo tanto, la identificación de dependencias es esencial para poder obtener la imagen original una vez comprimida.

Una vez identificadas las dependencias, se procederá a paralelizar el proceso de compresión. Para ello, se implementarán los *threads* mediante una nueva clase similar a la del codificador secuencial, con la única diferencia que implementará la clase *Runnable* [3]. Esta clase permite la generación, manipulación y destrucción de *threads*. Hay que tener en cuenta en qué momento se generar y se destruyen los *threads* ya que hacer estas acciones en instantes inapropiados dará lugar a resultados incorrectos. La implementación *multi-threading* ha de ser implementada en ambos módulos principales, es decir, tanto en el codificador como en el descodificador y ambos deben utilizar el mismo número de *threads*.

3 IMPLEMENTACIÓN

Las implementaciones se centran básicamente en el codificador y en el descodificador. No obstante, debido a que el predictor debe conocer la cantidad de datos a procesar, es decir, el número de muestras horizontales (*rows*) que procesará, habrá realizar alguna modificación.

3.1 Codificador

El codificador está compuesto por cuatro objetos que permiten realizar todo el proceso de compresión:

- **Codificador aritmético:** El codificador aritmético realiza el proceso de codificar cada bit a partir de la probabilidad de este.
- **Codificador contextual:** Obtiene el *contexto* de una muestra. El término *contexto* se basa en que si se escoge un píxel de la imagen de forma aleatoria, la probabilidad de que los píxeles adyacentes tengan el mismo valor o similar es muy alta.
- **Probabilidad del contexto (datos):** A partir del contexto de la muestra a estimar, se calcula una probabilidad del posible valor que tendrá dicha muestra.

Como se ha mencionado en el codificador contextual, el concepto de *contexto* se refiere a la información de la cual se dispone de los píxeles vecinos a una muestra a estimar. Una vez realizada dicha estimación, en caso de no obtener el valor real que debería tener dicho píxel provocará un error el cual podemos definir como:

$$e_{ij} = X_{ij} - P_{ij} \tag{3}$$

Si el predictor realiza una buena estimación del valor de la muestra este error será mínimo. No obstante, este *contexto* no lo forman todos los píxeles adyacentes ya que algunos de ellos aún no han sido procesados tal y como se ilustra en la siguiente figura.

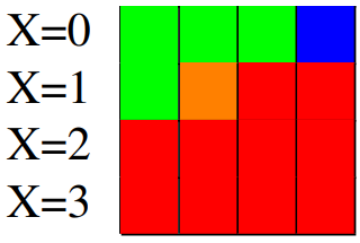

Fig. 3: Contexto de un píxel en una imagen 5x5.

Como se refleja en la figura anterior, se dispone de de los píxeles en X=0 (en color verde) los cuales ya han sido procesados. El píxel azul ha sido procesado pero al no ser adyacente a la muestra a estimar no será utilizado para el contexto. El píxel a estimar se refleja en color naranja y los píxeles de color rojo son aquellos que aún no han sido procesados ya que el *thread* recorre el subconjunto de datos que se le ha proporcionado de los datos originales de forma secuencial. Debido a esto, no se dispone del contexto "total"de un píxel en este esquema *multi-threading*.

Una vez se hayan codificado todos los datos de todos los *threads* que se tengan en ejecución, cal unir los datos resultantes de cada *thread* en el mismo orden en el que se han dividido antes de comenzar el proceso de compresión ya que sino la imagen resultante al descodificar no será la original.

El fichero que generará el codificador una vez haya acabado todo el procesado de compresión sigue la siguiente estructura:

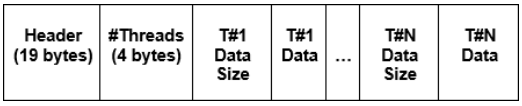

Fig. 4: Estructura del fichero codificado generado por el codificador.

En la figura anterior se ilustra la estructura del fichero de salida que generará el codificador. En él, cabe destacar que tanto la cabecera como el espacio reservado para el campo *Threads*, que indica la cantidad de *threads* que se han utilizado en el proceso de codificación, tienen una medida fija siendo estas de 19 y 4 bytes respectivamente. Es necesario codificar el número de hilos utilizados ya que el descodificador necesitará obtener los datos de cada *thread* a partir de este valor y el tamaño de los datos que cada hilo a generado a partir del campo *TN Data Size*.

A partir del código original y teniendo en cuenta las dependencias de los datos se crea una nueva clase la cual permite generar un hilo de ejecución con sus respectivos datos.

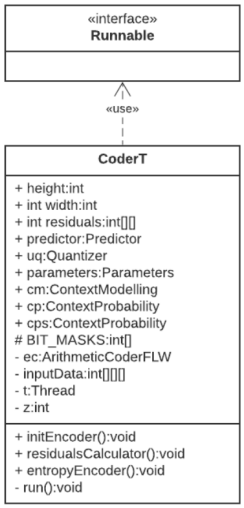

Fig. 5: Diagrama de clase del codificador.

3.2 Descodificador

Hasta ahora, se ha analizado todo el proceso de codificación en paralelo. Una vez se dispone de todos los datos codificados entra en juego el descodificador para observar si todo el proceso realizado anteriormente es correcto.

Para comenzar este proceso, cal leer el fichero generado por el codificador, a partir de la estructura que se muestra

tanto en la figura 4 como en la 5, para obtener únicamente los datos que corresponden a cada *thread*. Montado el mecanismo que permite obtener los datos de cada hilo, se procede a generar el mismo número de *threads* que se han empleado en el proceso de codificación. Esto se debe a que para descodificar, también es necesario utilizar el contexto de los píxeles. Por lo tanto, si los *threads* no procesas sus correspondientes datos, el contexto variará y con ello el resultado de la imagen obtenida.

El planteamiento e implementación a seguir es semejante a la empleada en el codificador solo que en este caso hay que realizar el proceso inverso, cal convertir datos codificados en los datos originales ya sean estos idénticos o no a los esperados.

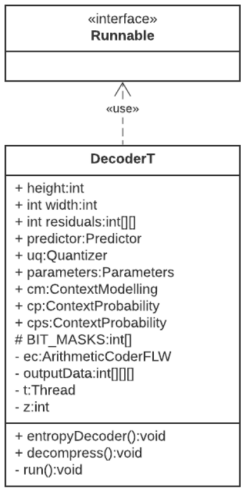

Fig. 6: Diagrama de clase del descodificador.

4 METODOLOGÍA

Para poder planificar y desarrollar las diferentes etapas que conforman el objetivo de este trabajo se han empleado las siguientes herramientas:

- **Eclipse IDE:** Entorno de desarrollo para aplicaciones Java (estándar).
- **Apache Ant:** Librería Java enfocada a la automatización de compilación de software, similar a *Make* en C/C++.
- **ImageJ:** Software de análisis de imágenes que permite realizar funciones de procesado de estas como la convolución, análisis de Fourier, transformaciones geométricas entre otras.
- **Trello:** Software de administración de proyectos que utiliza un sistema *kanban* para el registro de actividades con tarjetas virtuales para organizar tareas. Permite agregar listas, adjuntar ficheros, etiquetar eventos, agregar comentarios y compartir tableros para realizar proyectos en equipo. Permite obtener una visión general del proyecto y su avance.
- **FlexHEX:** Editor que permite visualizar, editar y comparar ficheros hex.

Para poder realizar una implementación que disminuyera los posibles errores que podrían surgir, se ha desglosado el objetivo principal del proyecto en cuatro sub-objetivos:

- Codificación y descodificación con imagen gris utilizando un único thread.
- Codificación y descodificación con imagen gris utilizando un conjunto de threads (2, 4, 8, 16 y 64).
- Codificación y descodificación con imagen en color (multicomponente) utilizando un único thread.
- Codificación y descodificación con imagen en color (multicomponente) utilizando un conjunto de threads (2, 4, 8, 16 y 64).

Para realizar la implementación de threads en *Java* se han creado las clases *CoderT* para el codificador y *DecoderT* para el descodificador tal y como se muestra en los diagramas de clases de las figuras 5 y 6 respectivamente. Ambas clases implementan la clase *Runnable* [4] para poder utilizar el método *run* el cual se llamará al comenzar la ejecución del hilo. No obstante, se debe esperar a que todos los threads finalicen para poder generar tanto el fichero codificado por el codificador como para la imagen descodificada por el descodificador. La clase *Runnable* únicamente proporciona el método *join* el cual pone en estado de espera el hilo de ejecución principal hasta que el thread que ejecute dicha función finalice. Esta situación no proporciona paralelización, sino concurrencia, ya que hasta que un thread no finalice, no se puede ejecutar el siguiente hilo. Por lo tanto, se ha utilizado la interfície *ExecutorService* [5], la cual permite manipular y controlar los threads que estén ejecución y, para este caso, ejecutar múltiples threads al mismo tiempo y esperar a que todos ellos finalicen proporcionando paralelismo en cambio de concurrencia.

Debido a que el proyecto se basa en la paralelización del código, para poder debugarlo ha sido necesario configurar el *debugger* para que permitiera suspender los hilos mientras se recorría el código mediante *breakpoints*. En caso de no habilitar esta opción, al intentar acceder a una etapa del thread mientras se debugaba el código, provocaba la finalización del programa ya que toda la máquina virtual (VM) se suspendía.

Mediante herramientas como *FlexHex* es posible visualizar la estructura descrita anteriormente con el fichero que se ha obtenido del codificador.

En la figura anterior se ilustra la comparativa entre el fichero obtenido sin utilizar ningún *thread*, es decir, todo el procesado en el hilo de ejecución principal y el fichero obtenido utilizando un único hilo el cual cabe destacar los campos resaltados. En rojo se encuentra marcado el número de *threads* utilizados en el proceso de compresión, en blanco la longitud de los datos generados por este hilo y en morado, los datos obtenidos al finalizar el procesado por el codificador aritmético.

Para poder corroborar que los sub-objetivos se completaban correctamente, se ha utilizado la herramienta *ImageJ* la cual permite visualizar imágenes *RAW* especificando el

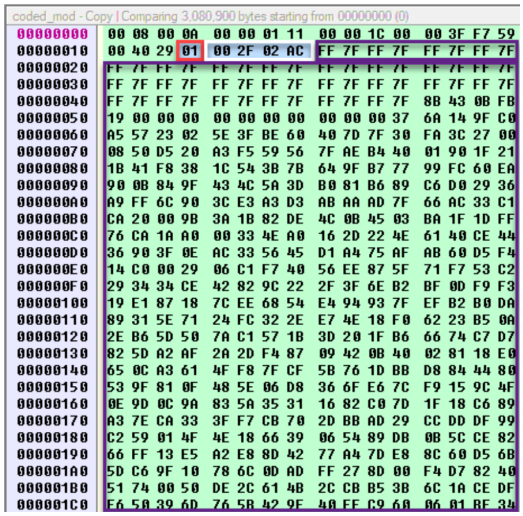

Fig. 7: Estructura del fichero codificado generado por el codificador (1 *thread*) mediante *FlexHex*.

tipo, el ancho y la altura de la imagen. En este caso, se han utilizado dos configuraciones para poder visualizar tanto la imagen en gris como la imagen en color.

	Imagen GRAY	**Imagen COLOR**
Tipo	8-bit	24-bit RGB
Altura	2560 píxeles	
Anchura	2048 píxeles	

Ha sido necesario utilizar este software a causa de que el formato *RAW* no se encuentra estandarizado y es necesario un visualizador de imágenes alternativo al que disponen por defecto los sistemas operativos.

Una vez se han cumplido todos los sub-objetivos se ha extraído la información necesaria siendo esta los bits por muestra (bps) de cada imagen y el tiempo de ejecución de la codificación y descodificación de datos para todos los conjuntos de hilos. La extracción de datos se ha implementado en el script *bash* que se ha utilizado durante todo el proyecto para compilarlo añadiendo la funcionalidad de generar un fichero *txt* en el cual se almacena toda la información comentada anteriormente para cada una de las imágenes a procesar.

Para poder indicar al software el número de threads que debe utilizar, se ha añadido un nuevo argumento *-tn* que permite especificar este atributo. Mediante dicho argumento, se ha implementado un bucle para poder realizar la codificación y descodificación de ambas imágenes variando el número de threads en cada iteración ya que, en el caso de la imagen multicomponente, el tiempo de ejecución del programa era significativamente más elevado, como se verá en la parte de resultados, que en el caso de la imagen en gris.

5 Resultados

Para obtener las conclusiones, se realizarán dos tipos de análisis según el número de *threads* que se utilicen en la ejecución:

- Análisis de rendimiento sobre el tiempo de ejecución del codificador/descodificador.
- Análisis sobre el ratio de muestras (*sample rate*).

En el tiempo de ejecución, únicamente se ha tenido en cuenta el procesado de datos originales (por parte del codificador) y de datos codificados (por parte del descodificador). El tiempo de escritura a disco que suele ser el *bottleneck* en un programa ha sido omitido.

5.1 Componente única

En esta sección se analizarán resultados respecto a la imagen pero la cual contiene una única componente, es decir, dicha imagen se compone por una escala de colores grises.

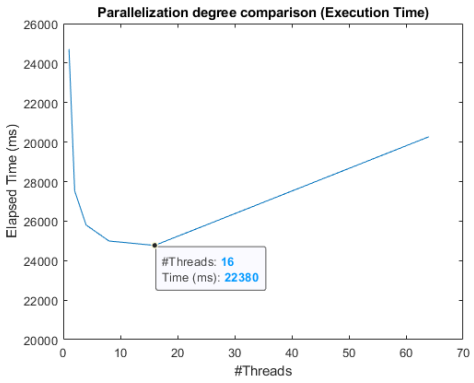

Fig. 8: Gráfica sobre el tiempo de ejecución respecto al número de *threads* en ejecución.

En la figura se refleja un comportamiento exponencial decreciente a medida que se utilizan más *threads* para el proceso de codificación. No obstante, a partir de 16 hilos, donde se encuentra el mínimo tiempo de ejecución, este se ve incrementado de forma lineal a causa de la ejecución con 64 threads. Este hecho nos indica que en el código existe alguna dependencia que provoca un *bottleneck* (cuello de botella) que no permite minimizar más el tiempo de ejecución.

Por otra parte, la medida de *bps* para esta imagen a partir de un número de threads determinado describe el siguiente comportamiento.

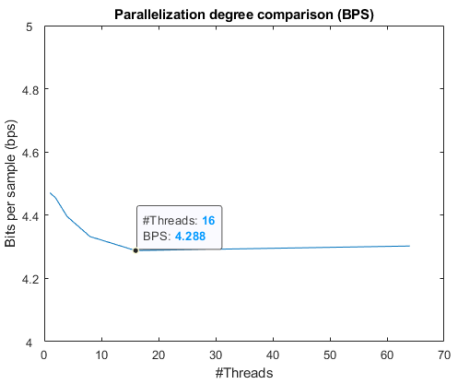

Fig. 9: Gráfica sobre los bits por muestra respecto al número de *threads* en ejecución.

A medida que el número de threads aumenta, el número de bits por muestra disminuye lo cual significa que a medida que el grado de paralelización del codificador aumenta, el ratio de compresión de la imagen es mayor, dicho de otra manera, se disminuye la entropía de la imagen.

5.2 Multicomponente

A continuación, se analizarán los datos obtenidos, al igual que en el caso de imagen con componente única siendo estos el tiempo de ejecución y los *bps*.

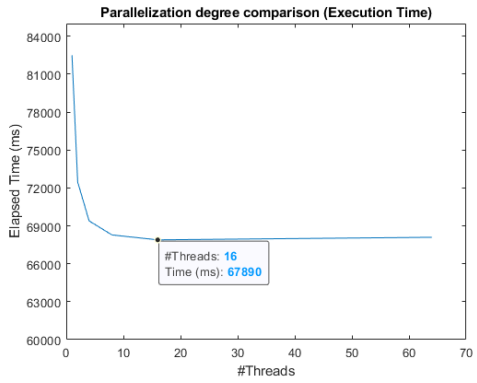

Fig. 10: Gráfica sobre el tiempo de ejecución respecto al número de *threads* en ejecución.

A diferencia de la imagen con componente única, la imagen en color alcanza su mínimo tiempo de ejecución al utilizar 16 threads, con la salvedad de que al utilizar un número mayor de hilos, en este caso 64, el tiempo de ejecución apenas sufre una variación respecto a utilizar 16 threads.

Desde el punto de vista de la medida *bps*, en la imagen multicomponente se obtiene el comportamiento opuesto al esperado, el cual sería el que se refleja en la figura 9.

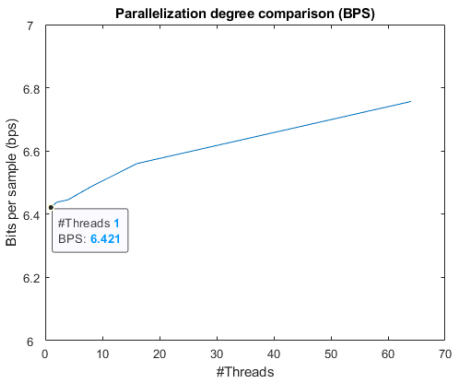

Fig. 11: Gráfica sobre los bits por muestra respecto al número de *threads* en ejecución.

A medida que el grado de paralelización (número de threads) aumenta, los bits por muestra aumentan describiendo un comportamiento lineal con una pendiente mínima. No obstante, este hecho significa que a medida que se utiliza un mayor número de threads, la entropía de los datos aumenta provocando que el tamaño de la imagen resultante aumente.

6 Conclusiones

Durante el desarrollo del proyecto se ha adquirido una base sobre el funcionamiento de diferentes hilos de ejecución en un programa escrito en *Java* el cual era el objetivo principal del trabajo. No obstante, los conocimientos esenciales que este proyecto ha aportado han sido el funcionamiento tanto en términos de compresión como de descompresión de imágenes el cual ha sido interesante y la detección de dependencias para obtener los resultados esperados al paralelizar todo el proceso del procesado de las imágenes. Al ser un proyecto en que el resultado es visual ha sido posible pensar en los posibles fallos que han podido provocar ciertos errores al no tener en cuenta diversos factores como el uso compartido de variables entre threads o la distribución de los datos obtenidos de las imágenes originales entre los diversos hilos. Este proyecto se encaraba a la paralelización, en caso de haber querido obtener unos resultados mejores, en términos de tiempo de ejecución, sería interesante implementar el mismo proyecto en un lenguaje de medio nivel como C/C++ o, en caso de querer seguir utilizando *Java*, analizar detenidamente el código mediante un perfilador.

Agradecimientos

Durante todo este camino desde que entre en ingeniería, he conocido a muchas personas que compartían mis mismas inquietudes. Gracias a amigos que han hecho que el transcurso de la carrera haya sido más pasajero y entretenido. Gracias a los profesores que me han ayudado tanto en dudas respecto a sus asignaturas como a proyectos propios que he ido desarrollando por cuenta propia a lo largo de la carrera. Dar las gracias a mi tutor Joan Bartrina por la paciencia que ha tenida debido a las circunstancias que han surgido durante el proyecto y la atención que ha tenido a la hora de resolver dudas y aportar soluciones al desarrollo del proyecto. Y por último y más importante, a mis padres, por todo el sacrificio que han hecho para que actualmente me encuentre en este momento. Espero que este sea el comienzo de un largo camino de aprendizaje.

Referències

[1] Cover, T. M.; Thomas, J. A. Elements of Information Theory. Wiley, 2nd edition, 2006.

[2] Joan Bartrina, "Practica compressió de dades" Teoria de la Informació, Grau d'Enginyeria de Dades UAB.

[3] Java Doc, Runnable class, `https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html`

[4] Oaks, Scott, and Henry Wong. Java Threads. 2nd ed. The Java Series. Sebastopol, CA: O'Reilly Associates, 1999.

[5] Java Doc, ExecutorService class, `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html`