aingeru ramos
leroy deniz
GRUPO 01

ARQUITECTURA DE COMPUTADORES

INFORME TECNICO

COMPUTO PARALELO

Programa para la aceleración del proceso de digitalización de la biblioteca del Fondo de los Benedictinos

índice

abstract

En función de la necesidad de contar con un proceso automático y ágil para la digitalización de los libros de la Biblioteca del Fondo de los Benedictinos, se analiza a continuación el problema, desde una perspectiva de utilización del cómputo paralelo como una herramienta de optimización de tiempo y recursos, frente a un trabajo donde su tiempo serial sería ampliamente superior.

La estrategia del cómputo paralelo está en hacer de un gran problema recurrente, una serie de problemas pequeños capaces de ejecutarse de manera paralela, aspirando a que el tiempo de trabajo sea tantas veces más rápido como unidades de ejecución paralelas utilicemos.

En la puesta en producción de este programa, cada hoja de libro pasará por tres etapas: filtrado, para obtener una imagen con menos ruido; cifrado, para guardar las imágenes con un mínimo de seguridad; carga al servidor, para poder contar con ellas en la nube en todo momento. Los tiempos medidos se toman en función de una muestra de tres hojas de libro de distintos tamaños, cuyo procesamiento será la base del cálculo de tiempo total para la estimación de la digitalización de la biblioteca en forma íntegra.

Este informe propone dar una panorámica, no solo de una solución eficiente a una problemática propuesta, sino además, de una forma de entender el paralelismo en términos de tiempo, carga de trabajo y sus correspondientes comparativas en otras configuraciones de planificación.

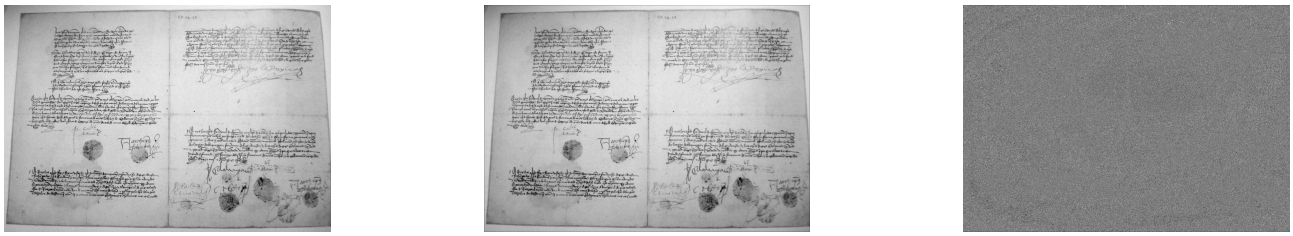
| presentación del problema

El Centro de Documentación de los Benedictinos de Lazkao, lleva varios años recopilando documentación de temática vasca y archivándola en el denominado Fondo de los Benedictinos. Parte de este patrimonio está digitalizado y se puede acceder desde Internet (http://www.lbf.eus/es/).

El trabajo de digitalización se está llevando a cabo durante los últimos años en orden cronológico inverso, y ahora toca digitalizar los documentos más antiguos de los que se dispone. El proceso consiste en escanear las páginas de los libros a digitalizar. Para mejorar la calidad de los documentos y reducir en parte el "ruido" que introduce el escáner, las páginas son filtradas, es decir, limpiadas y archivadas en una plataforma online para que puedan ser accedidas desde Internet. Los documentos se guardan cifrados para mantener la seguridad de la información. Para ello, antes de subirlas a la plataforma hay que cifrarlas.

El Centro de Documentación de los Benedictinos quiere finalizar con el proceso de digitalización en un tiempo limitado. Por ello, han comprado un multiprocesador de 48 procesadores para poder procesar la información en paralelo y así acabar antes. Es necesario por tanto programar un algoritmo paralelo que sea capaz de procesar las páginas de los documentos en el menor tiempo posible, o lo que es lo mismo, que pueda tratar la mayor cantidad de páginas por minuto. Se quieren procesar 10 millones de páginas en total, con un tamaño medio de 20 MB.

La Figura 1 (fuente: https://ca.wikipedia.org/wiki/Ferran_el_Cat%C3%B2lic) presenta un ejemplo del proceso que se quiere llevar a cabo. Página escaneada (Figura 1a), página filtrada (Figura 1b) y página cifrada (Figura 1c).



(a) (b) (c)

Figura 1. Procesado de una página de 11 MB: (a) pág. escaneada, (b) pág. filtrada y (c) pág. cifrada.

marco teórico

Contextualización

El objetivo del cómputo paralelo está en replicar las unidades de tratamiento de información con el objetivo de repartir las tareas, es decir, en realizar una misma tarea recurrente en varios procesadores a la vez. En este sentido, se busca optimizar los tiempos de ejecución de un bucle en n veces, siendo n el número de procesadores disponibles involucrados en este proceso.

Podemos clasificar el paralelismo en dos tipos: (a) de datos, que se utiliza si todos los procesos ejecutan el mismo algoritmo sobre diferentes datos, es decir, se repite el cuerpo del bucle y se reparten los datos a iterar y (b) de función, se usa cuando se reparten los cálculos (procedimientos, funciones, subprogramas) entre los procesadores. Por ejemplo, repartir un programa con *n* funciones entre K procesadores. Entonces cada procesador ejecuta un trozo del programa original.

Para eso, se requiere el análisis de tres aspectos fundamentales para un uso correcto de esta metodología: planificación, sincronización y rendimiento. A continuación se explica, a grandes rasgos, cada uno de ellos.

Planificación

El objetivo de la planificación está en mantener balanceada la carga de trabajo de cada hilo y, además, de reducir las operaciones de sincronización. Para esto, cada bucle se puede configurar en tres tipos: static, dynamic y guided.

La planificación static se caracteriza por un reparto de tareas estática en tiempo de compilación, es decir, cada hilo tiene su número de tareas fijada de antemano, resultando una planificación equilibrada pero poco optima en términos de tiempo ya que no asegura que todas las tareas requieran el mismo tiempo y terminen lo más próximas posibles.

La planificación dynamic propone un reparto de tareas dinámico, es decir, a medida que cada hilo va quedando libre de carga, va tomando nuevas tareas para procesar. De esta forma se reducen considerablemente la sobrecarga de hilos pero, en cambio, se aumenta el tiempo de sincronización.

La planificación guided propone un reparto similar al dynamic, con la diferencia que los trozos de código a ejecutar (cantidad de tareas), van siendo más pequeños conforme avanza el procesamiento del bucle.

Podemos aplicar una clasificación del reparto en tres niveles: (a) grano grueso, cuando se reparten funciones o procedimientos enteros a cada hilo, (b) grano medio, cuando se reparten bucles enteros y (c) grano fino, cuando se reparten iteraciones del bucle.

El reparto de iteraciones se puede también distribuir de dos formas: (a) reparto consecutivo, donde cada procesador ejecuta un grupo de iteraciones consecutivas del bloque, como por ejemplo, instrucciones de un bucle con N/3 instrucciones por procesador en tres procesadores y (b) reparto entrelazado, donde se reparten una a una por procesador.

Pueden repartirse entrelazadas de dos o más también; para bucles con instrucciones independientes, da igual el reparto; para bucles con sincronización, es mejor entrelazado ya que consecutivo sería como ir en serie.

Sincronización

En una máquina MIMD, la ejecución de los programas se divide en P procesos o hilos, que se ejecutan en paralelo en los procesadores del sistema. En general, la ejecución de esos procesos no es completamente independiente, sino que se comunican entre ellos, bien sea para pasarse datos o para sincronizar su ejecución.

En resumen, para poder ejecutar un programa en P procesadores, a menudo es necesario sincronizar el uso de las variables compartidas. La sincronización entre procesos puede resolverse por software o por hardware. Si se hace en hardware, suele ser más rápida pero menos flexible; si se hace por software (bibliotecas), se suelen obtener soluciones más flexibles.

Las estrategias básicas de sincronización son dos: exclusión mutua (mediante funciones lock/unlock) y sincronización por eventos (punto a punto, mediante indicadores o flags, o global, mediante barreras).

Un mecanismo de sincronización adecuado debe cumplir algunas condiciones, entre las que cabe destacar:

- Baja latencia: se debe gastar el menor tiempo posible en efectuar la operación de sincronización, sea cual fuera la situación del programa.
- Tráfico limitado: el tráfico que se genera en el acceso y uso de las variables de sincronización debe ser el mínimo posible, para evitar saturar la red de comunicación.
- Buena escalabilidad: tanto la latencia como el tráfico no deben crecer con el número de procesadores del sistema.

Poco coste de memoria: no se debe utilizar mucha memoria para la sincronización.

Igualdad de oportunidades: todos los procesos deben tener las mismas oportunidades de resolver sus peticiones de sincronización.

Se utiliza la *exclusión mutua* para controlar la ejecución de un trozo de código que, aunque está replicado en P procesos, no puede ser ejecutado por más de un proceso simultáneamente. Ese trozo de código forma una sección crítica y nunca debe haber más de un proceso ejecutándolo. Para proteger el acceso a una sección crítica se utilizan dos funciones específicas, *lock* y *unlock*, que manejan una variable de tipo cerrojo.

El cerrojo puede tomar dos valores: 0 y 1, Si el cerrojo vale 0 (abierto), no hay problema alguno para ejecutar la sección crítica; en cambio, si el cerrojo vale 1 (cerrado) hay que esperar, ya que otro proceso está ejecutando en ese momento la sección crítica.

Para poder gestionar secciones críticas necesitamos disponer de instrucciones atómicas de tipo RMW (read-modify-write) que permitan efectuar una operación de lectura y escritura sobre una variable (el cerrojo) en modo atómico.

Decimos que la sincronización es *punto a punto* si sólo toman parte en la misma dos procesadores (o grupos): el primero avisa al segundo de que se ha ejecutado determinada operación. La sincronización se suele ejecutar mediante un bucle de espera activa sobre una variable común que hace las veces de flag o indicador. El flag o indicador es una variable de control que permite sincronizar ambos procesos.

En la ejecución en paralelo de los programas suele ser muy habitual que se necesite sincronizar un grupo de procesos entre sí de manera global, todos a la vez. Para ese tipo de sincronización se utilizan *barreras*

(barrier). Para construir una barrera de sincronización se utiliza una variable cerrojo, un contador y un flag. En la barrera se sincronizan P procesos. Cuando los procesos llegan a la barrera, incrementan el valor de un contador —en exclusión mutua— y se quedan esperando a que lleguen todos los procesos. Cuando llega el último, activa el indicador de barrera abierta, y todos los procesos abandonan la misma.

Rendimiento

Como el coste de la paralelización es elevado, el objetivo de estudio del rendimiento está en obtener que un programa ejecutado en n procesadores, se ejecute n veces más rápido. Para el análisis de esta característica, se utilizan dos parámetros: factor de aceleración (speed-up) y eficiencia (efficiency). Ambos parámetros son cuantificables utilizando las fórmulas a continuación.

Speed-up: $fa = T_S / T_P$	**Eficiencia: $efic = fa / P$**

En este punto existen dos Leyes que merecen la pena destacar: la Ley de Amdahl y la Ley de Gustafson.

La Ley de Amdahl plantea que, si en un algoritmo existe una parte que necesariamente debe ser ejecutada en serie o con menos procesadores, es muy difícil conseguir factores de aceleración altos y la eficiencia se reduce considerablemente.

En contraposición a la primera, la Ley de Gustafson plantea que, si el código que efectivamente se ejecuta en paralelo es considerablemente grande, entonces se puede alcanzar un factor de aceleración tendiente al ideal.

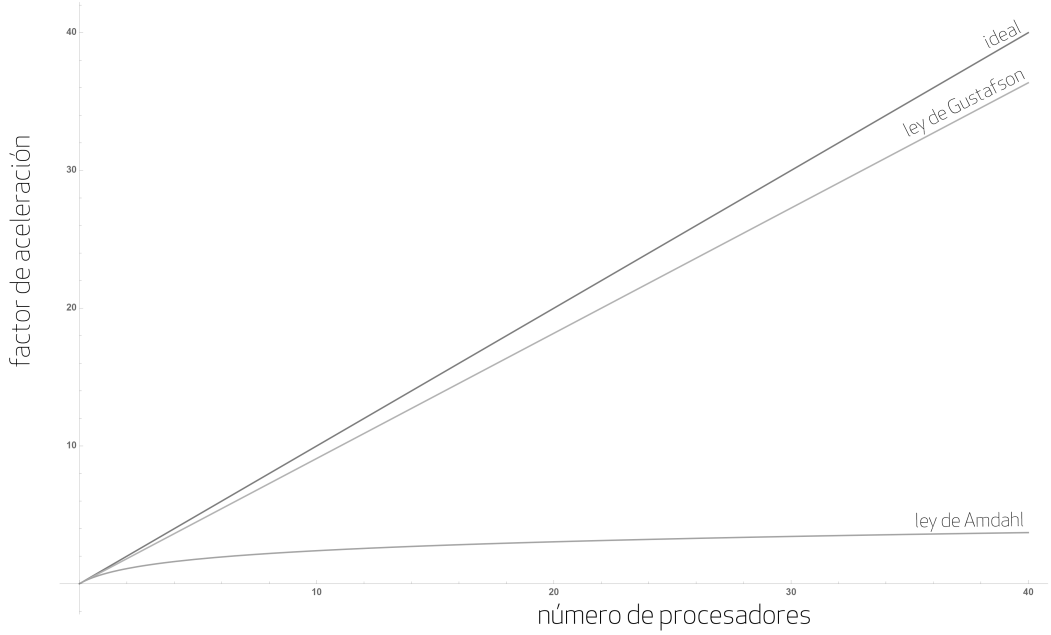


Figura 2. Gráfico del factor de aceleración según las leyes de Amdahl y Gustafson.
Ambas curvas representan la máxima aceleración que puede obtenerse.

objetivos e hipótesis

Objetivo general

Implementación de un algoritmo para el proceso de digitalización del archivo de Libros.

Objetivos específicos

1. Analizar diferentes combinaciones de planificación para decantar por la más eficiente en cada bucle.
2. Estudiar el rendimiento particular de cada bucle y los tiempos del algoritmo general.
3. Establecer un tiempo promedio de digitalización de la totalidad de los libros a procesar.

Hipótesis

Una hipótesis para el proceso de filtrado, es que la utilización de una configuración estática en 0 es la más eficiente, puesto que no habría dependencias con otros pixeles.

En cuanto al proceso de cifrado, la configuración estática en 0 presentaría mejores resultados, en función de que son tratados cada par de pixeles de forma independiente al resto de la imagen. Esto daría mejores datos de tiempo a la hora de su tratamiento en paralelo.

Finalmente, para el proceso de carga al servidor, estimamos como óptima una configuración dinámica en 32, en función que el proceso es similar al proceso de cifrado pero puede tener dependencias en la carga.

desarrollo del proyecto

Tal cómo se expuso supra, cada imagen pasa por tres etapas: la primera es el filtrado que permite reducir el ruido de la misma, generando así una imagen más clara; la segunda etapa es de cifrado, donde se encriptan dos pixeles a la vez mediante el Algoritmo de Hill; la tercer etapa es la de la carga a la nube, donde se van enviando al servidor la información de cada pixel de la imagen encriptada.

Ahora bien, pasamos a explicar las funciones y procedimientos del algoritmo.

Filtrado

```
void generar_pagina_filtrada(pagina *in_page, pagina *out_page, float limite);
```
Procedimiento principal del proceso de filtrado, cuando culmina, obtenemos la página filtrada.

```
void copiar_pagina(pagina *in_page, pagina *out_page);
```
Copia la página in_page a la página out_page.

```
double calcularMatrizFiltro(pagina *in_page, int x, int y);
```
Calcula la media de los pixeles de alrededor del pixel im[x][y], siendo in_page la página desde donde se cogerán los pixeles y *x* e *y* la fila y columna respectivamente.

```
double filtrar_pagina(pagina *in_page, pagina *out_page);
```
Aplica el filtro a cada pixel de la matriz de la imagen a filtrar, apoyándose en la función calcularMatrizFiltro para realizar el cálculo del valor del pixel en cuestión.

Encriptado

```
void generar_pagina_encriptada(pagina in_page, pagina *out_page);
```
Procedimiento principal del proceso de encriptado, cuando culmina, obtenemos la página cifrada.

```
void encriptar_pixeles (unsigned char *v1, unsigned char *v2);
```
Función para encriptar dos pixeles a través del algoritmo de cifrado de Hill.

Preparar subida

```
void preparar_subida(pagina in_page);
```
Toma pixel a pixel y, a través de la función upload, va cargando los datos al servidor.

Carga al servidor

```
void upload (unsigned char vp1, int i, int j, int h, int w);
```
Función de carga al servidor.

Manejo de imágenes

```
void  generar_pagina(pagina *opage, int h, int w, unsigned char val);
```
Constructor de páginas con los valores que recibe por parámetro.

```
void  borrar_pagina(pagina ipage);
```
Destructor de estructuras página.

```
int load_pixmap(char *filename, pagina *ipage);
```
Carga los pixels de una imagen a la estructura creada.

```
void store_pixmap(char *filename, pagina opage);
```
Persiste los pixels de una imagen en un archivo.

resultados y conclusiones

Tiempos en serie

Para medir los tiempos en serie, vamos a probar tres opciones de optimización del compilador GCC pero a trabajaremos en base a una aplicación sin optimización por parte del compilador.

```
marq01@u010415:~/AP/versionSerie$ gcc pagina_s.c filter.c encript.c preparar_subida.c
                                    pixmap.o upload.o -lm -o app / -O1 / -O2 / -O3
```

TIEMPO TOTAL EN SERIE				
	SIN PARÁMETRO	**-O1**	**-O2**	**-O3**
Libro1.pgm	12837,3	4707,4	3628,6	2588,1
Libro2.pgm	23797,7	8677,7	6696,1	4699,2
Libro3.pgm	55955,6	20773,4	16397,9	11446,1

```
--> Procesando la pagina: Libro1.pgm (2758 x 3986). Limite: 177.0

FILTRADO: 10 iteraciones, limite 176.723

    Filtrar la pagina:          Tej. serie = 10952.9 ms
    Cifrar la pagina:           Tej. serie = 207.2 ms
    Preparar la subida:         Tej. serie = 709.1 ms

    TOTAL:                      Tej. serie = 11869.1 ms


--> Procesando la pagina: Libro2.pgm (4096 x 4980). Limite: 177.5

FILTRADO: 10 iteraciones, limite 177.068

    Filtrar la pagina:          Tej. serie = 20331.3 ms
    Cifrar la pagina:           Tej. serie = 387.0 ms
    Preparar la subida:         Tej. serie = 1291.3 ms

    TOTAL:                      Tej. serie = 22009.7 ms


--> Procesando la pagina: Libro3.pgm (6520 x 8080). Limite: 125.0

FILTRADO: 9 iteraciones, limite 124.932

    Filtrar la pagina:          Tej. serie = 48035.2 ms
    Cifrar la pagina:           Tej. serie = 988.0 ms
    Preparar la subida:         Tej. serie = 3552.8 ms

    TOTAL:                      Tej. serie = 52576.1 ms
```

Tiempos en paralelo

A continuación probaremos distintas configuraciones de planificación con varios tamaños en distinta cantidad de hilos para las tres imágenes de referencia. El mejor tiempo de cada funcionalidad para cada imagen nos va a dar el dato de qué configuración es la óptima y, si la cantidad de trabajo amerita, una configuración distinta.

Libro 1 | Tiempo de Filtrado (ms)														
		STATIC							DYNAMIC					
	#	0	1	2	4	8	16	32	1	2	4	8	16	32
THREADS	2	5655,2	5854,8	5667,9	5671,3	5633,9	5654,7	5658,2	5676,5	5667,3	5660,4	5910,5	5657,9	5684,5
	4	2892,3	2938,4	2910,8	2903,7	2890,1	2898,0	2946,6	2949,3	2900,8	2907,1	2900,3	2936,7	3012,4
	8	1516,2	1553,9	1553,1	1541,4	1558,1	1574,7	1560,1	1525,4	1532,0	1530,0	1543,2	1549,0	1564,3
	16	902,9	939,4	880,9	921,4	918,5	947,8	952,1	868,5	885,7	878,4	868,6	859,3	939,3
	24	696,8	721,3	691,8	719,7	739,9	791,6	738,9	664,4	630,0	688,6	663,1	701,7	718,8
	32	536,4	641,6	638,0	654,2	605,9	611,3	661,5	571,2	570,1	602,3	586,3	654,5	599,5

Libro 1 | Tiempo de Cifrado (ms)														
		STATIC							DYNAMIC					
	#	0	1	2	4	8	16	32	1	2	4	8	16	32
THREADS	2	120,8	121,9	121,8	121,2	122,6	120,7	124,3	134,6	122,0	121,3	121,2	121,1	121,4
	4	62,7	64,2	63,7	64,4	65,2	63,8	63,8	65,6	63,8	64,9	64,5	63,8	63,7
	8	36,0	34,8	35,7	35,4	35,7	36,0	35,2	35,6	35,4	35,7	35,6	35,6	35,2
	16	20,9	20,9	21,2	20,6	20,3	20,8	20,3	21,5	20,7	21,1	21,1	20,8	21,1
	24	15,5	15,3	16,5	16,4	16,4	15,8	15,5	16,8	16,3	16,2	15,6	16,1	16,6
	32	13,5	14,3	13,9	14,3	14,3	14,4	14,0	14,3	13,9	14,1	14,0	14,8	14,0

Libro 1 | Tiempo de Preparar subida (ms)														
		STATIC							DYNAMIC					
	#	0	1	2	4	8	16	32	1	2	4	8	16	32
THREADS	2	660,4	887,5	632,4	503,7	443,8	408,7	392,6	1544,2	916,3	632,3	566,7	442,8	417,3
	4	208,3	445,6	318,2	258,4	224,9	210,1	206,8	1535,8	785,1	532,4	363,1	239,9	221,7
	8	117,5	223,9	159,6	127,5	112,6	113,9	109,1	1299,0	763,8	446,9	280,6	167,0	122,6
	16	56,0	113,3	81,3	65,6	67,2	57,3	54,2	1289,6	802,7	436,6	240,3	130,3	80,7
	24	37,6	77,0	59,7	63,5	45,5	38,6	36,5	1281,8	813,2	430,4	234,7	117,7	72,4
	32	44,9	66,9	60,9	45,4	34,7	29,5	27,1	1301,9	795,8	245,0	225,2	114,1	65,5

Libro 2 | Tiempo de Filtrado (ms)														
		STATIC							DYNAMIC					
	#	0	1	2	4	8	16	32	1	2	4	8	16	32
THREADS	2	10539,0	10477,1	10600,7	10536,3	10523,1	10509,1	10461,6	10540,7	10506,3	10531,3	10511,4	10512,2	10509,6
	4	5371,0	5418,6	5423,5	5370,1	5337,5	5386,9	5386,3	5395,0	5398,6	5398,8	5353,7	5379,6	5388,2
	8	2828,9	2849,1	2831,1	2841,6	2836,7	2807,7	2799,1	2831,7	2837,6	2821,6	2811,6	2832,6	2810,1
	16	1666,7	1643,7	1677,4	1674,6	1688,6	1652,9	1629,0	1622,0	1587,5	1568,2	1569,8	1586,6	1574,5
	24	1233,6	1390,6	1298,8	1294,8	1204,3	1276,9	1288,8	1137,8	1146,3	1146,8	1168,7	1174,2	1242,8
	32	943,6	1375,0	1227,4	1367,7	1152,1	1054,3	1061,8	1077,5	989,5	970,1	970,8	949,9	944,5

Libro 2 | Tiempo de Cifrado (ms)														
		STATIC							DYNAMIC					
	#	0	1	2	4	8	16	32	1	2	4	8	16	32
THREADS	2	224,7	229,3	226,7	224,2	223,5	223,9	224,4	224,3	224,7	225,8	225,8	223,7	225,7
	4	117,6	117,9	118,1	118,0	116,7	118,0	117,5	120,2	121,5	118,4	118,3	118,9	118,4
	8	63,0	64,1	64,6	64,6	64,7	64,2	63,6	64,5	64,5	64,0	64,2	63,6	63,7
	16	37,1	38,2	38,4	37,6	38,4	37,8	37,0	37,2	37,6	37,2	37,3	37,3	37,6
	24	28,2	29,5	28,8	28,5	28,4	28,6	28,3	28,5	29,7	29,4	28,5	28,7	27,5
	32	23,5	24,6	25,1	24,6	23,9	23,6	24,5	25,1	24,0	24,1	24,01	24,0	23,8

Libro 2 | Tiempo de Preparar subida (ms)														
		STATIC							DYNAMIC					
	#	0	1	2	4	8	16	32	1	2	4	8	16	32
THREADS	2	1159,0	1156,5	1175,5	923,3	814,1	748,2	717,0	2824,0	1685,9	1175,0	943,6	812,3	746,3
	4	602,4	829,0	585,2	463,0	408,4	375,1	366,9	2524,6	1427,7	970,5	608,3	439,8	390,9
	8	300,9	421,2	294,5	234,2	211,3	209,1	202,4	2633,1	1394,7	860,6	495,8	317,7	220,6
	16	150,7	209,9	152,2	121,5	124,6	105,5	100,5	2448,4	1556,2	805,2	419,0	253,3	158,9
	24	111,0	143,2	125,3	117,5	83,4	73,00	69,0	2479,7	1471,5	838,8	435,8	255,8	149,8
	32	75,3	133,4	109,4	83,3	63,5	53,5	51,0	2560,9	1534,2	810,3	401,8	205,8	125,7

Libro 3 | Tiempo de Filtrado (ms)														
		STATIC							DYNAMIC					
	#	0	1	2	4	8	16	32	1	2	4	8	16	32
THREADS	2	25115,3	25463,8	25415,3	25404,9	25311,9	25299,7	25306,4	25301,6	25422,8	25290,5	25277,5	25248,2	25224,0
	4	13112,1	13360,2	13245,6	13323,8	13247,7	13296,0	13196,5	13303,5	13273,8	13239,1	13254,6	13209,0	13204,8
	8	7143,3	7197,1	7266,2	7251,9	7186,6	7552,4	7238,4	7229,1	7232,0	7211,7	7176,5	7194,3	7232,3
	16	4270,1	4207,3	4245,1	4247,0	4248,0	4298,0	4294,7	4241,0	4236,3	4199,4	4208,2	4253,0	4253,1
	24	3236,5	3314,9	3220,4	3282,7	3280,7	3227,7	3385,8	3267,0	3242,4	3218,2	3218,9	3209,3	3307,9
	32	2736,5	2754,5	2763,6	2744,2	2819,0	2809,3	2841,0	2772,8	2788,9	2742,7	2745,4	2747,1	2801,3

Libro 3 | Tiempo de Cifrado (ms)														
		STATIC							DYNAMIC					
	#	0	1	2	4	8	16	32	1	2	4	8	16	32
THREADS	2	574,5	574,6	574,0	577,8	574,9	580,6	575,3	574,9	574,7	575,6	574,2	574,3	575,0
	4	299,0	302,3	298,9	302,3	299,5	302,4	300,0	299,2	299,7	300,0	300,1	301,6	302,1
	8	161,1	162,5	160,6	161,7	161,6	163,0	164,4	161,3	161,5	161,3	164,7	161,8	162,4
	16	91,8	92,8	91,4	92,1	92,3	92,1	92,5	91,9	92,5	92,0	93,8	93,8	92,3
	24	70,0	69,0	69,3	69,2	97,8	70,0	70,3	69,7	70,3	70,7	70,0	70,6	70,2
	32	56,5	57,9	61,7	59,1	57,3	58,9	57,6	57,3	58,5	57,3	58,3	57,9	57,4

Libro 3 | Tiempo de Preparar subida (ms)														
		STATIC							DYNAMIC					
	#	0	1	2	4	8	16	32	1	2	4	8	16	32
THREADS	2	3120,9	4320,7	3055,7	2402,8	2101,3	1938,8	1870,7	7278,5	4386,4	3065,8	2434,1	2129,2	1945,4
	4	1555,2	2168,9	1531,6	1207,7	1073,7	975,3	971,2	6557,6	3740,2	2341,3	1513,6	1150,4	1029,6
	8	795,4	1087,9	771,5	613,1	534,6	547,6	518,6	6107,4	3393,3	2194,2	1318,1	863,7	590,3
	16	401,8	552,3	395,3	317,3	328,4	275,4	265,3	6907,1	3992,8	2120,9	1151,8	658,8	408,5
	24	275,6	374,2	291,9	293,8	223,7	184,8	173,9	6818,8	3987,7	2117,6	1083,7	579,4	356,3
	32	211,3	305,8	270,5	216,2	164,7	139,2	131,0	7117,4	3724,7	1997,6	1072,8	535,6	321,1

Conclusiones

Nuestro criterio de selección de tiempo fue ejecutar tres veces cada configuración y tomar el tiempo que estaba en el medio. En este caso, si los tiempos medidos en las tres ejecuciones fueron 448,9; 448,5 y 451, tomamos 448,9 como el valor de referencia para dicha configuración.

En función de las tablas supra, es posible ver que, en la imagen Libro1, el menor tiempo de filtrado es 536,4ms en static,0 con 32 hilos. Esa misma configuración es donde conseguimos el mejor tiempo en la imagen Libro2 con 943,6ms, lo simismo que en la imagen Libro3, con 2736,5ms.

Ahora bien, en cuanto al tiempo de cifrado, es claramente evidente que el menor tiempo en el procesado de las tres imágenes lo presenta la configuración static,0 en 32 hilos. Su tiempo más próximo lo presenta static,32, lo que nos da a pensar que este bucle no tiene dependencias, por lo tanto, cuanto mayor sea el bloque de instrucciones a repartir, el tiempo de sincronización será despreciable y la carga estará bien balanceada. Además, como conocemos el cálculo interior, podemos decir que el tiempo de cada instrucción será constante.

Cabe destacar que, tanto en el proceso de filtrado como en el de encriptado, el factor de aceleración es prácticamente igual en todas las configuraciones de planificación. Esto nos lleva a intuir que no hay dependencias de datos en los bucles que se paralelizan y, por tanto, da igual el número de instrucciones que tenga que ejecutar en paralelo y cómo se repartan. En este caso, elegimos static,32 por una diferencia ínfima con el resto, únicamente porque la diferencia en las diez millones de imágenes a procesar producirían una diferencia notoria, pero para volúmenes de procesamiento no tan grandes, sería también correcto.

Finalmente, para el proceso de carga del archivo al servidor, es visible en las tablas, que el mejor tiempo de subida lo presenta static,32 en 32 hilos para las tres imágenes, con 27,1ms, 51,0ms y 131,0ms respectivamente.

Esta diferencia frente a los otros dos procesos da a pensar que hay una instrucción de tiempo significativo cada , por lo menos, 4 instrucciones a ejecutar.

Revisión de hipótesis

En referencia a las hipótesis planteadas al comienzo, estuvimos en lo correcto al pensar que static,0 representaría el mejor tiempo en el filtrado, lo mismo para el proceso de encriptado, cuanto mayor es el bloque de instrucciones que se asigna a un procesador, mejor es el tiempo.

En el proceso de subida al servidor, el mejor tiempo lo presenta static,32 en comparación con nuestra apuesta de dynamic,32. En las tablas se ve cómo la diferencia entre uno y otro es 2.45 veces más rápido.

TIEMPOS PARA LIBRO1.PGM			
	Filtrado	**Cifrado**	**Subida**
Configuración	static,0 en 32 hilos	static,0 en 32 hilos	static,32 en 32 hilos
Tiempo	536,4	13,5	27,1
Speed-up	10952,9 / 536,4 = 20,42	207,2 / 13,5 = 15,35	709,1 / 27,1 = 26,17
Eficiencia	20,42 / 32 = 0,64	15,35 / 32 = 0,48	26,17 / 32 = 0,82

Tiempos con la planificación óptima

A continuación mostramos una ejecución con los tiempos óptimos que calculamos en el apartado anterior para cada Libro.

```
 --> Procesando la pagina: Libro1.pgm (2758 x 3986). Limite: 177.0

FILTRADO: 10 iteraciones, limite 176.723

     Filtrar la pagina:      Tej. serie = 536.4 ms
     Cifrar la pagina:       Tej. serie = 13.5 ms
     Preparar la subida:     Tej. serie = 27.1 ms

     TOTAL:                  Tej. serie = 577.0 ms

 --> Procesando la pagina: Libro2.pgm (4096 x 4980). Limite: 177.5

FILTRADO: 10 iteraciones, limite 177.068

     Filtrar la pagina:      Tej. serie = 943.6 ms
     Cifrar la pagina:       Tej. serie = 23.5 ms
     Preparar la subida:     Tej. serie = 51.0 ms

     TOTAL:                  Tej. serie = 1018.1 ms

 --> Procesando la pagina: Libro3.pgm (6520 x 8080). Limite: 125.0

FILTRADO: 9 iteraciones, limite 124.932

     Filtrar la pagina:      Tej. serie = 2736.5 msz
     Cifrar la pagina:       Tej. serie = 56.5 ms
     Preparar la subida:     Tej. serie = 131.0 ms

     TOTAL:                  Tej. serie = 2924.0 ms
```

Representaciones gráficas

Filtrado

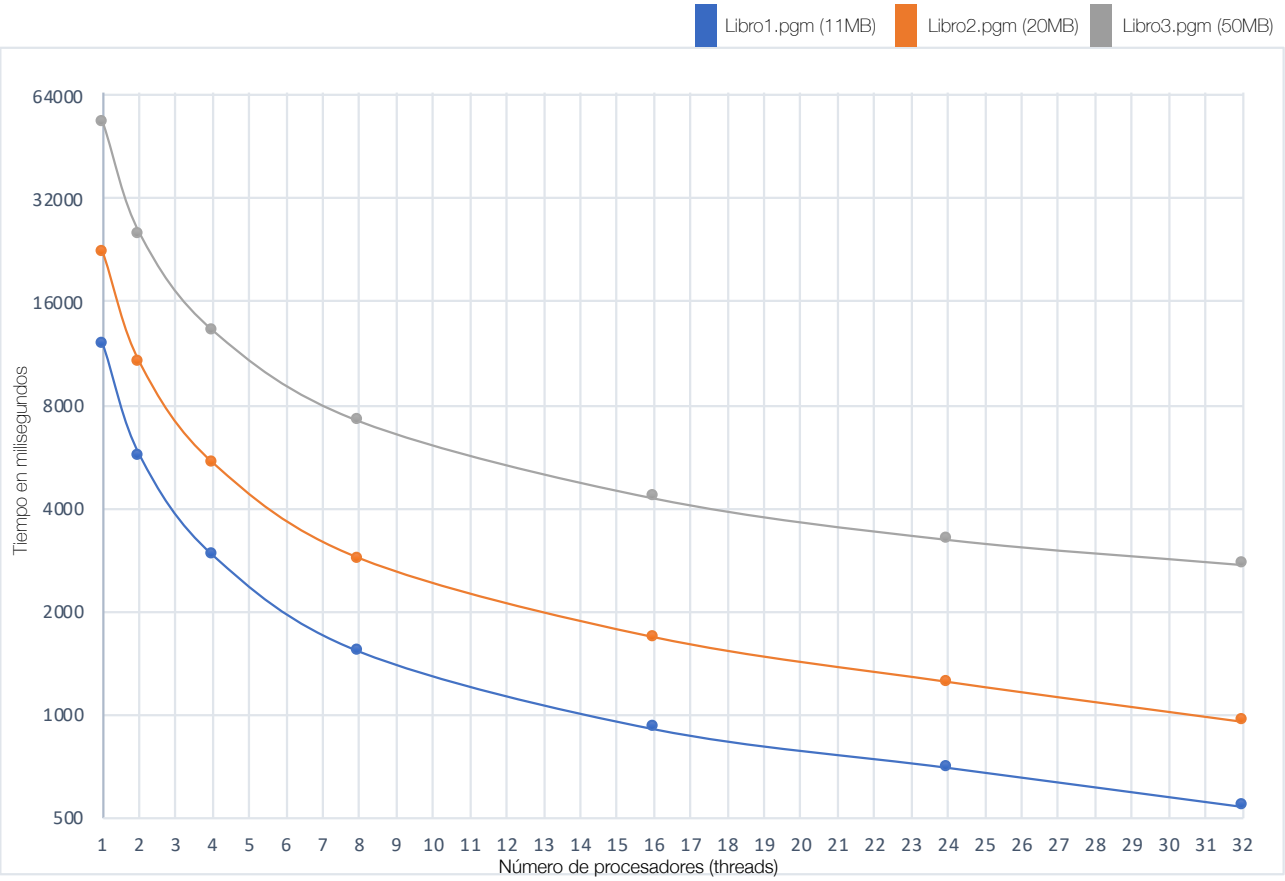


Figura 3. Gráfico del del tiempo de filtrado de cada imagen en función de la cantidad de procesadores utilizados.

Como expusimos en las conclusiones, es importante destacar que el mejor tiempo lo conseguimos cuando utilizamos 32 procesadores para ejecutar el bucle de filtrado. Además, en cuanto a la gráfica, los valores son tan similares, que basta con graficar una de las configuraciones de planificación para representarlas a todas.

También podemos apreciar que la carga de trabajo es proporcional con el tiempo en los tres casos, es decir, que hay suficiente trabajo para paralelizar con una cantidad considerable de procesadores sin generar costos de armado extra.

Cifrado

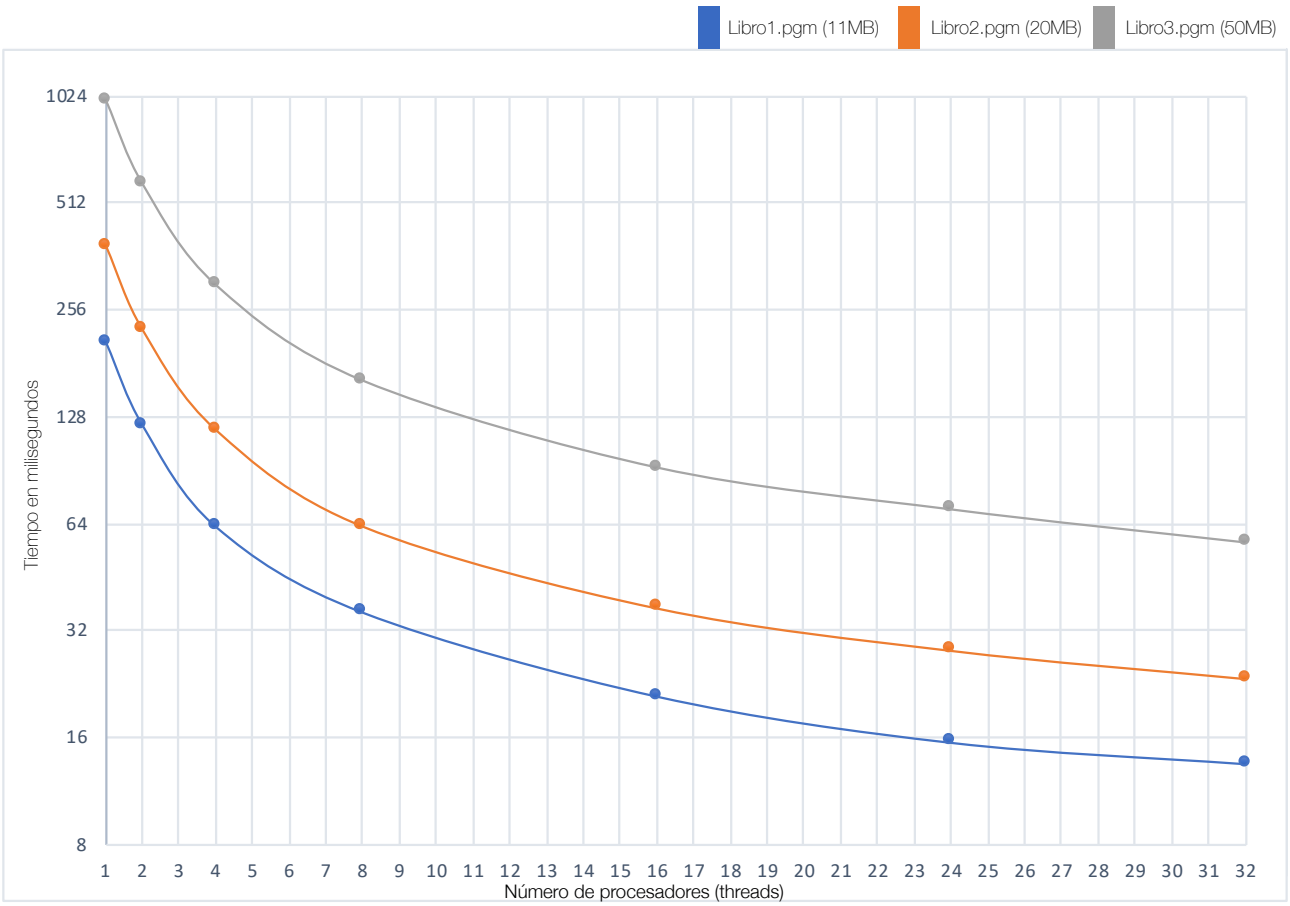


Figura 4. Gráfico del del tiempo de encriptado de cada imagen en función de la cantidad de procesadores utilizados.

Idem para Encriptado. Como expusimos en las conclusiones, es importante destacar que el mejor tiempo lo conseguimos cuando utilizamos 32 procesadores para ejecutar el bucle de encriptado. Además, en cuanto a la gráfica, los valores son tan similares, que basta con graficar una de las configuraciones de planificación para representarlas a todas.

También podemos apreciar que la carga de trabajo es proporcional con el tiempo en los tres casos, es decir, que hay suficiente trabajo para paralelizar con una cantidad considerable de procesadores sin generar costos de armado extra.

Subida

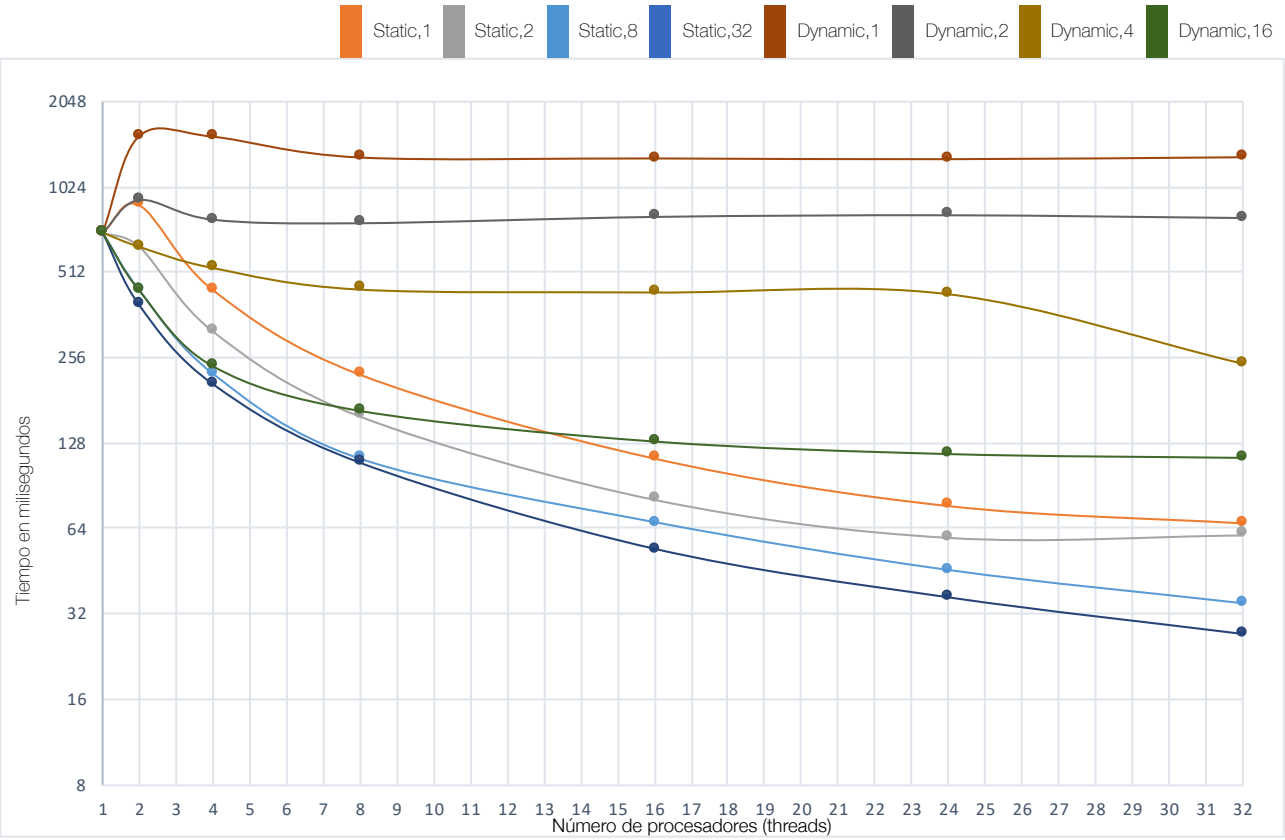


Figura 5. Gráfico del del tiempo de subida de Libro1.pgm en varias configuraciones de planificación en función de la cantidad

En la gráfica se puede ver cómo los tiempos en planificación dinámica son significativamente más altos, incluso el más bajo de ellos, en dynamic 16. En esta gráfica hemos omitido dynamic,32 ya que la curva es similar a dynamic,16.

El mejor tiempo lo presenta Static,32 con 32 procesadores. Probablemente esté sucediendo que hayan instrucciones cuyo tiempo sea mayor en las primeras iteraciones, lo que hace que, repartiendo las instrucciones en bloques grandes, esté mejor distribuida la carga, en comparación con un dynamic,1 donde el tiempo de sincronización y de trabajo es influyente en los resultados.

Para las imágenes Libro2 y Libro3, las gráficas son similares a la anterior, salvaguardando la escala a la correspondiente, pero todas se comportan de la misma forma, lo que da lugar a sacar las mismas conclusiones.

Respuesta al problema

Para responder a la pregunta del problema, usamos Libro2.pgm cuyo peso es de 20,4MB, correspondiéndose así con el peso relativo de cada página, según se plantea.

Habiendo que procesar 10.000.000 de páginas en total a un tamaño promedio de 20MB, tomando en consideración los tiempos obtenidos a lo largo de este proceso para libro2, siendo este 1018,1ms, podemos estimar que el tiempo que insumirá este trabajo es de 117,8 días.

$$10.000.000_{\text{imágenes}} \times 1,0181_{\text{seg}} = 2028,0555 \text{ horas} = 117,8 \text{ días}$$

Si en vez de utilizar paralelismo hubiéramos utilizado las opciones de optimización del compilador, tendríamos un tiempo de 4699,2 en serie frente a 1018,1 por imagen en serie. De esta forma vamos 4,6 veces más rápido que en la compilación más óptima que insumiría 541,8 días

| bibliografía

William Stallings, *Computer Organization and Architecture* 5th Edition, Prentice Hall, 2000, ISBN: 0130812943.

J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd Edition, Morgan Kaufmann Publishing Co., 2002, ISBN: 1558607242.

Andrew S. Tanenbaum, *Structured Computer Organization* 4th Edition, Prentice Hall, 1998, ISBN: 0130959901.

anexos

Anexo I: Código fuente de la aplicación paralela

pagina_s.c

```
/*****************************************************************************
   Fichero: pagina.c (SERIE)

   Se aplica a una pagina en formato PGM el siguiente tratamiento
     -- filtrado para minimizar el rudio
     -- encriptacion
     -- preparacion del proceso para la carga de la pagina en la nube

   Ficheros de entrada:   xx.pgm (pagina en formato pgm)
   Ficheros de salida:      xx_fil_ser.pgm    pagina filtrada
                          xx_cif_ser.pgm    pagina encriptada

   Compilar el programa junto con filter.c, encript.c y preparar_subida.c

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>

#include "pixmap.h"
#include "filter.h"
#include "encript.h"
#include "preparar_subida.h"


/************************                MAIN              *********************/
/*****************************************************************************/

void main(int argc, char **argv)
{
  char name[100];
  int i;

  // Pagina: original, filtrada y cifrada
  pagina page_ori,page_fil,page_cif;

  // calculo de tiempos
  struct timeval t0,t1;
  double tej1,tej2,tej3,tej;

  if (argc != 3)
  {
    printf ("\nUSO: programa pagina limite\n");
    exit (0);
  }

  // Lectura de la pagina de entrada: solo paginas graylevel en formato .pgm
  if (load_pixmap(argv[1], &page_ori) == 0)
  {
    printf ("\nError en lectura del fichero de entrada: %s\n\n",argv[1]);
    exit (0);
  }

  /* Procesado de pagina: generar nuevas paginas (filtrada y encriptada)                        */
  /**********************************************************************************************/
  printf("\n --> Procesando la pagina: %s (%d x %d). Limite: %.1f\n",argv[1],page_ori.h,page_ori.w,atof(argv[2]));

  gettimeofday(&t0, 0);
  generar_pagina_filtrada(&page_ori,&page_fil,atof(argv[2]));
  gettimeofday(&t1, 0);
  tej1 = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
  printf("\n     Filtrar la pagina: \tTej. serie = %1.1f ms", tej1*1000);

  gettimeofday(&t0, 0);
  generar_pagina_encriptada(page_fil,&page_cif);
  gettimeofday(&t1, 0);
  tej2 = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
```

```
  printf("\n     Cifrar la pagina: \t\tTej. serie = %1.1f ms", tej2*1000);

  gettimeofday(&t0, 0);
  preparar_subida(page_cif);
  gettimeofday(&t1, 0);
  tej3 = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
  printf("\n     Preparar la subida: \tTej. serie = %1.1f ms", tej3*1000);

  tej = tej1+tej2+tej3;
  printf("\n\n     TOTAL: \t\t\tTej. serie = %1.1f ms\n\n", tej*1000);

  /* Escritura de las paginas en el disco                                      */
  /*****************************************************************************/
  strcpy(name,argv[1]);
  name[strlen(name)-4]='\0';
  strcat(name,"_fil_ser.pgm");
  store_pixmap(name,page_fil);

  strcpy(name,argv[1]);
  name[strlen(name)-4]='\0';
  strcat(name,"_cif_ser.pgm");
  store_pixmap(name,page_cif);

  // Borrar las paginas
  borrar_pagina(page_ori);
  borrar_pagina(page_fil);
  borrar_pagina(page_cif);
}
```

filter.c

```
/*****************************************************************************
   Fichero: filter.c

   Contiene las funciones para el filtrado de la pagina
*****************************************************************************/
#include <math.h>
#include "pixmap.h"
#include <stdio.h>
#include <omp.h>

#define NUM_IT 100


/* Copia la pagina in_page en la pagina out_page  */
/************************************************/
//Parametro
         //in_page --> La pagina a copiar
         //out_page --> La copia de in_page

void copiar_pagina(pagina *in_page, pagina *out_page)
{
         int i,j;
         for (i=0;i<in_page->h;i++)
                  for (j=0;j<in_page->w;j++)
                           out_page->im[i][j]=in_page->im[i][j];
}

//Calcula la media de los pixeles de alrededor del pixel im[x][y]
//Parametros
         //in_page--> Pagina de donde se cogeran los pixeles
         //x --> Fila del pixel
         //y --> Columna del pixel

int calcularMatrizFiltro(pagina *in_page, int x, int y)
{
   int result = 0;

   /* este for no amerita paralelizarlo porque solo recoreo 9 elementos */
   for(int i=-1; i<=1; i++) {
      for(int j=-1; j<=1; j++) {
         result += in_page->im[x+i][y+j];
      }
   }
   return result/9;
}

/* Aplicar filtro a una pagina                                */
/**************************************************************/
//Parametros
         //in_page --> La pagina a filtrar
         //out_page --> La pagina una vez filtrada

double filtrar_pagina(pagina *in_page, pagina *out_page)
{
   int i, j, dim = ((in_page->w)-2) * ((in_page->h)-2), minimo = 255;
   double sum = 0.0, media = 0.0, valor = 0.0;
```

```
////////////////////////////////////////////////////////////////////
/* POR HACER: aplicar el filtro de suavizado a la pagina in_page        */
/* se aplica a la pagina in_page, dejando el resultado en out_page      */
/* la funcion devuelve dos valores:                                     */
/* el valor medio y minimo de todos los pixeles de la pagina filtrada   */
/* la función devuelve el valor medio menos el mínimo                   */
////////////////////////////////////////////////////////////////////////

   #pragma omp parallel for schedule(static,0) private(j,valor) reduction(+:sum) reduction(min:minimo)
   for(i=1; i< in_page->h-1; i++) {
      for(j=1; j< in_page->w-1; j++) {
         valor = calcularMatrizFiltro(in_page, i, j);
         out_page->im[i][j] = valor;
         sum += valor;
         if(valor < minimo) {
            minimo = valor;
         }
      }
   }
   return (sum / dim) - minimo;
}

/* Genera la pagina filtrada               */
/*****************************************/
//Parametros
        //in_page --> La pagina original
        //out_page --> La pagina filtrda
        //limite --> Limite de filtrado de la imagen

void generar_pagina_filtrada(pagina *in_page, pagina *out_page, float limite)
{
   generar_pagina(out_page,in_page->h,in_page->w,NEGRO);
   copiar_pagina(in_page,out_page);
   double result=0;
   for(int i=0; i<NUM_IT; i++) {
        if (i%2==0){
                result = filtrar_pagina(in_page, out_page);
                if(result < limite) {
                        printf ("\nFILTRADO: %d iteraciones, limite %.3f\n", i+1, result);
                        copiar_pagina(out_page,in_page);
                        break;
                }
        } else {
                result = filtrar_pagina(out_page, in_page);
                if(result < limite) {
                        printf ("\nFILTRADO: %d iteraciones, limite %.3f\n", i+1, result);
                        break;
                }
        }
   }

 ////////////////////////////////////////////////////////////////////////
 /* POR HACER: generar la pagina filtrada a partir de la pagina original */
 /* Proceso iterativo. Condición final de filtrado:                      */
 /*     (a) superar el numero máximo de iteraciones                      */
 /*     (b) valor medio menos minimo inferior al limite                  */
 /* asegurar que la pagina final queda en out_page                       */
 ////////////////////////////////////////////////////////////////////////
}
```

encript.c

```
/*****************************************************************************
     Fichero: encript.c

        Generar la pagina encriptada

*****************************************************************************/

#include "pixmap.h"
#include <stdio.h>
#include <stdlib.h>



////////////////////////////////////////////////////////////////////////////////
/*          codigo para el encriptado de 2 pixeles con el metodo de HILL            */
////////////////////////////////////////////////////////////////////////////////
//Parametros
        //v1 --> Primer pixel
        //v2 --> Segundo pixel

void encriptar_pixeles (unsigned char *v1, unsigned char *v2) {

        //DEFINO LA MATRIZ DE ENCRIPTADO
```

```
        int matrizEnc [2][2];
        matrizEnc[0][0] = 21;
        matrizEnc[0][1] = 35;
        matrizEnc[1][0] = 18;
        matrizEnc[1][1] = 79;

        //EN LOS MISMOS PUNTEROS QUE RECIBE COMO PARAMETRO, GUARDA LOS PIXELS ENCRIPTADOS
        *v1 = (unsigned char)( matrizEnc[0][0]*(int)(*v1) + matrizEnc[0][1]*(int)(*v2) )%256;
        *v2 = (unsigned char)( matrizEnc[1][0]*(int)(*v1) + matrizEnc[1][1]*(int)(*v2) )%256;

} //END ENCRIPTAR_PIXELES


////////////////////////////////////////////////////////////////////////////////
/*      codigo para generar la pagina encriptada ultilizando la pagina filtrada     */
/*                                                                                */
/*Parametros                                                                      */
/*       in_page --> Pagina filtrada                                              */
/*       out_page --> Pagina encriptada despues del proceso                       */
////////////////////////////////////////////////////////////////////////////////

void generar_pagina_encriptada(pagina in_page, pagina *out_page) {

        generar_pagina(out_page,in_page.h,in_page.w,NEGRO);
        int height, width;
        unsigned char pixel1, pixel2;

        //FOR QUE RECORRA EL ARRAY DE LA IMAGEN PARA ENCRIPTAR DE A DOS PIXELS
        #pragma omp parallel for schedule(static,0) private(width,pixel1,pixel2) collapse(2)
        for (height=0;height<in_page.h;height++) {
                for (width=0;width<in_page.w;width=width+2) {
                        pixel1 = in_page.im[height][width];
                        pixel2 = in_page.im[height][width+1];

                        //encriptar_pixeles (&in_page.dat[width], &in_page.dat[width+1]);
                        encriptar_pixeles(&pixel1, &pixel2);
                        out_page->im[height][width] = pixel1;
                        out_page->im[height][width+1] = pixel2;
                }
        }
} //END GENERAR_PAGINA_ENCRIPTADA
```

preparar_subida.c

```
/*****************************************************************************
     Fichero: preparar_subida.c

       Contiene las funciones para preparar la subida de la pagina a la nube
*****************************************************************************/

#include "pixmap.h"
#include "upload.h"
#include <omp.h>

/*****************************************************************************/
/* Prepara la subida de la pagina encriptada a la nube                       */
//Parametros
        //in_page --> La pagina encriptada
/*****************************************************************************/

void preparar_subida(pagina in_page)
{
        int i,j;
        #pragma omp parallel for schedule(static,32) private (j) collapse(2)
        for(i=0; i<in_page.h; i++) {
           for(j=0; j<in_page.w; j++) {
                upload(in_page.im[i][j], i, j, in_page.h, in_page.w);
           }
        }

////////////////////////////////////////////////////////////////////////////////
/* POR HACER: codigo para preparar la transmision de la pagina               */
/* codigo para preparar la subida de la pagina                               */
/* tened en cuenta la cabecera de la funcion upload (upload.h)               */
//                                                                            //
/* extern void upload (unsigned char vp1, int i, int j, int h, int w);       */
/* pixel de la pagina, su posicion (i,j) y dimensiones de la pagina (h,w)    */
/* se trata de aplicar la funcion upload a cada pixel                        */
////////////////////////////////////////////////////////////////////////////////

/*
        generar_pagina(out_page,in_page.h,in_page.w,NEGRO);
        copiar_pagina(in_page,out_page);
*/
}
```

Anexo II: Script de bash para test de configuraciones

Ejecuta todas las combinaciones de {static, dynamic} con bloques continuos o entrelazados de {0, 1, 2, 4, 8, 16, 32} en {2, 4, 8, 16, 24, 32} hilos para las tres imágenes.

script.sh

```
#!/bin/sh
#BUCLE PARA RECORRER LAS TRES IMAGENES
for imagen in Libro1.pgm Libro2.pgm Libro3.pgm
do
  #!BUCLE PARA VARIAR LA PLANIFICACION
  for plan in static dynamic
  do
    #BUCLE PARA VARIAR EL NUMERO DE INSTRUCCIONES
    for instrucciones in 0 1 2 4 8 16 32
    do
      export OMP_SCHEDULE="$plan,$instrucciones"

      #BUCLE PARA VARIAR EL NUMERO DE HILOS
      for hilos in 2 4 8 16 24 32
      do
        echo
        echo "**********************($plan, $instrucciones) con $hilos hilos en $imagen **********************"

        export OMP_NUM_THREADS=$hilos
        echo

        if [ $imagen = "Libro1.pgm" ]; then
          ./app "$imagen" 177
          ./app "$imagen" 177
          ./app "$imagen" 177
        fi
        if [ $imagen = "Libro2.pgm" ]; then
          ./app "$imagen" 177.5
          ./app "$imagen" 177.5
          ./app "$imagen" 177.5
        fi
        if [ $imagen = "Libro3.pgm" ]; then
          ./app "$imagen" 125
          ./app "$imagen" 125
          ./app "$imagen" 125
        fi
        echo
      done
    done
  done
done
```

Se ejecuta y se guarda el resultado en un archivo de texto, donde luego tomaremos los datos para procesar medias de tiempos.

```
marq01@u010415:~/AP/versionParalela$ sh script.sh >> resultados.txt
```

Nota: En el resultado del script anterior, hay que descartar los resultados de la planificación dynamic,0.