Aplicación de una transformada wavelet discreta en imágenes

Arquitectura de Computadores

Marcos Vives Del Sol

Cristian Milá Pallás

https://github.com/socram8888/DWTAC

Objetivo y utilidad del algoritmo

El objetivo de nuestro proyecto es desarrollar una versión paralelizada del algoritmo PLHaar en su versión discreta. El algoritmo PLHaar es una DWT (Discrete Wavelet Transform) y se emplea en tareas de compresión además de otros campos.

Las transformadas Wavelet descomponen la señal una serie de subseñales organizadas de forma jerárquica por relevancia de los datos. Cada transformación separa la señal en dos subseñales, una aproximación de la señal original y una que recoge los detalles extraídos de la aproximación. Estas subseñales pueden ser utilizadas para reconstruir la señal original sin ninguna pérdida.

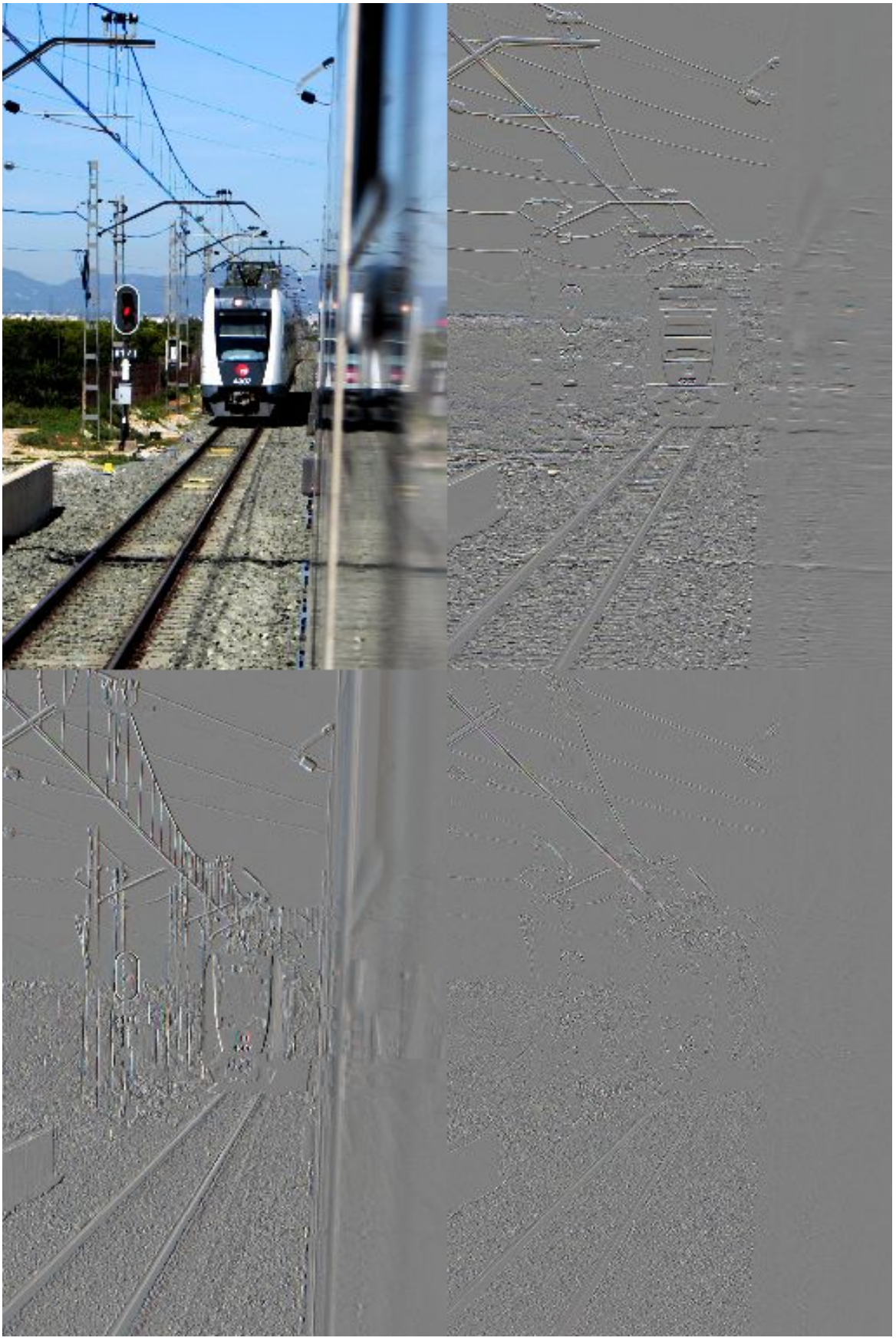
En este proyecto vamos a utilizar esta transformada para el tratamiento de imágenes. Como reza el enunciado del trabajo, se utilizará sobre una imagen de un mínimo de 32x2 píxeles, en escala de grises o en RGB, con 4 coeficientes y un nivel arbitrario de profundidad.

Esto nos permitirá obtener una serie de imágenes que se comportarán como las señales propuestas en el párrafo anterior, de forma que tendremos una imagen reducida de la original y tres más adicionales donde estarán representados los detalles verticales, horizontales y diagonales de la primera imagen procesada, obteniendo así los 4 coeficientes.

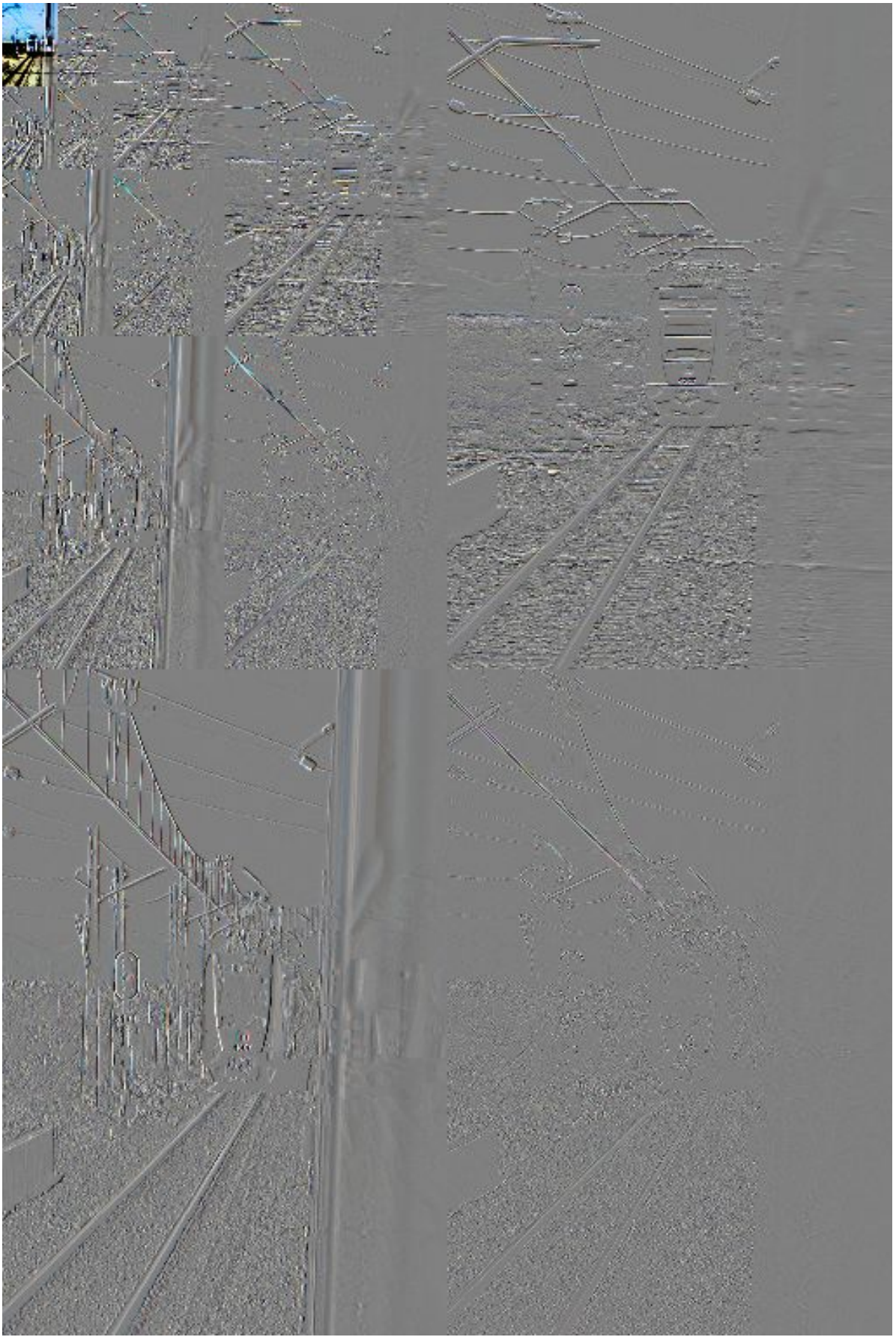
En cada iteración se reducirá el tamaño de la imagen original, tantas veces como niveles de profundidad se quieran obtener.

Finalizado el proceso, el algoritmo es capaz de recomponer la imagen primigenia sin ninguna pérdida.

Imagen original



PLHaar con profundidad 1



*PLHaar de profundidad 4 - **imagen real obtenida de la ejecución del programa***

Imagen original (©TysonTan; CC-BY-SA;, y PLHaar con profundidad 3. El contraste de la imagen calculada ha sido aumentado.

Descripción del algoritmo

PLHaar dispone de dos entradas: A y B, que son bytes que contienen dos valores de un mismo canal de dos píxeles contiguos. La salida es L (lowpass), que representa en la nueva imagen una aproximación de la original; y H (highpass), que representa la diferencia entre los píxeles.

La función PLHaar tiene hay dos entradas y dos salidas, y se encuentra definida a trozos:

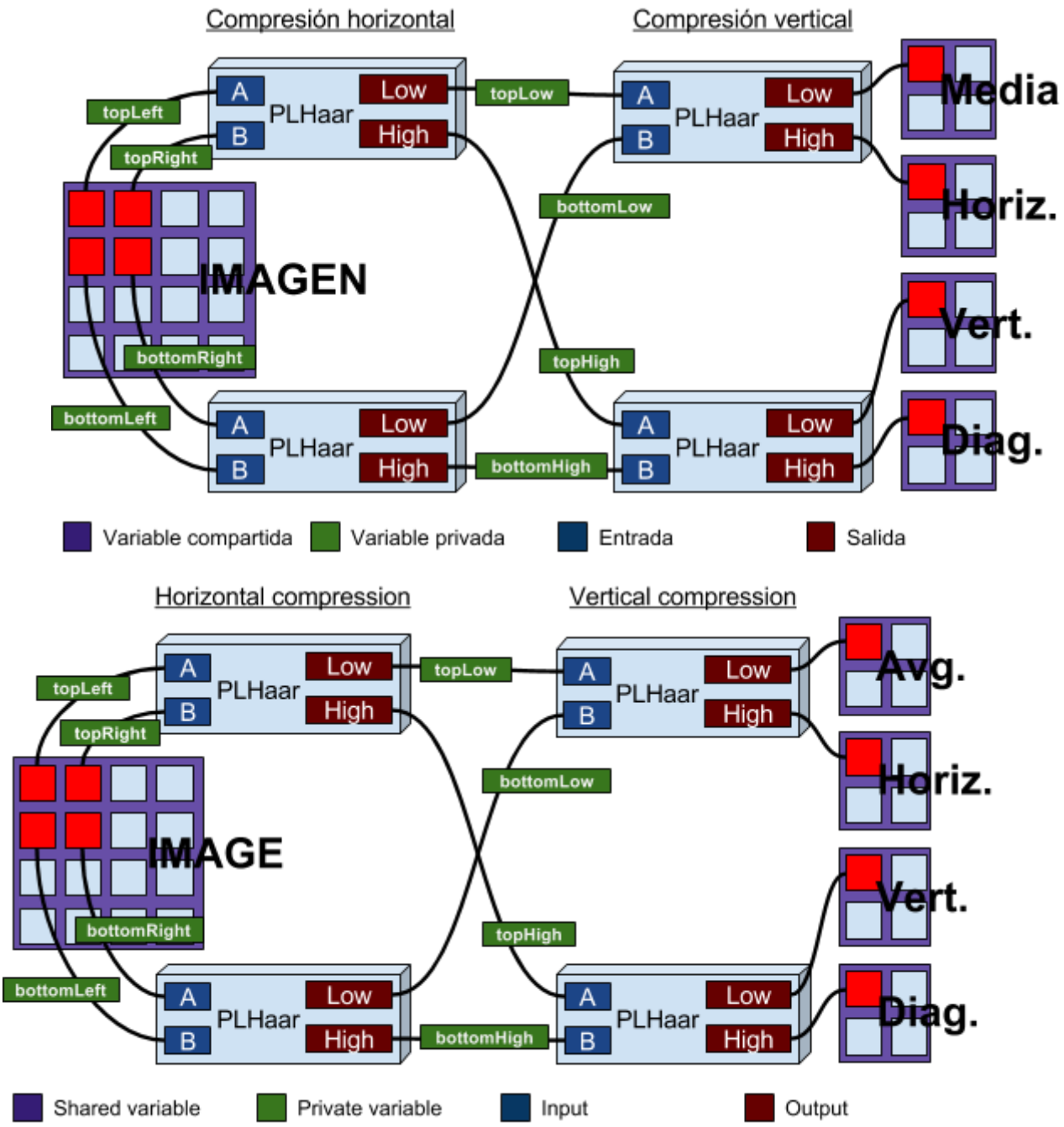
1. Si $(a \geq 128) = (b \geq 128) \wedge a > b$:
   - $h = (a - b) \bmod 256$
   - $l = a$
2. Si $(a \geq 128) = (b \geq 128) \wedge a \leq b$:
   - $h = (a - b) \bmod 256$
   - $l = b$
3. Si $(a \geq 128) \neq (b \geq 128) \wedge a < b$:
   - $h = (-b) \bmod 256$
   - $l = (a + b - 128) \bmod 256$
4. Si $(a \geq 128) \neq (b \geq 128) \wedge a \geq b$:
   - $h = a$
   - $l = (a + b - 128) \bmod 256$

Este proceso se realiza con todos los píxeles contiguos -horizontales, verticales y diagonales- para obtener 4 imágenes de tamaño la mitad del de la imagen fuente: uno es la aproximación real y las otras 3 contienen información sobre las diferencias horizontales, verticales y diagonales de los píxeles que forman la original.

A continuación una muestra de la implementación del código:

```
void plhaar_int(uint8_t a, uint8_t b, uint8_t * l, uint8_t * h) {
    const uint8_t c = 128;
    const uint8_t s = (a < c), t = (b < c);

    a += s; b += t;          // asymmetry: nudge origin to (+0,+0)
    if (s == t) {            // A * B > 0?
        a -= b - c;          // H = A - B
        if ((a < c) == s) {  // |A| > |B|?
            b += a - c;      // L = A (replaces L = B)
        }
    } else {                 // A * B < 0
        b += a - c;          // L = A + B
        if ((b < c) == t) {  // |B| > |A|?
            a -= b - c;      // H = -B (replaces H = A)
        }
    }
    a -= s; b -= t;          // asymmetry: restore origin
    *l = b; *h = a;          // store result
}
```

Esta operación se ejecuta con cuatro píxeles colindantes (superior izquierdo, superior derecho, inferior izquiero y inferior derecho), y como resultado se obtienen la media, la diferencia horizontal, diferencia vertical y diferencia diagonal:



Análisis del coste y memoria necesaria

El coste de nuestro proyecto, definiendo como coste el número de accesos a los píxeles, si tenemos n píxeles, en la primera iteración es de n, ya que operamos con todos los píxeles, que producirá una salida un cuarto del tamaño de la original. En la siguiente iteración emplearemos este cuarto, y los reduciremos en un cuarto, con lo que tendríamos un dieciseisavo de la original.

Matemáticamente, si k es el número de iteraciones, y n el número de píxeles (resultado de la multiplicación del ancho por el alto de la imagen), se puede definir recursivamente como:

$$Coste(1) = n$$

$$Coste(i) = n + \frac{Coste(i-1)}{4} \quad \forall i \in \mathbb{N} \wedge i > 1$$

Definiéndolo como un sumatorio:

$$\sum_{i=1}^{k} \frac{n}{4^{i-1}} = \frac{n(1-(\frac{1}{4})^k)}{1-\frac{1}{4}} = \frac{n(1-\frac{1}{4^k})}{\frac{3}{4}} = \frac{4n}{3}(1 - \frac{1}{4^k})$$

La memoria necesaria para la ejecución de nuestro algoritmo es igual al doble de la imagen original cargada.

Esto se debe a que necesitamos la memoria para cargar la imagen a procesar y otro espacio de tamaño idéntico para la formación de la imagen resultado, compuesta por la aproximación de la original y tres más (también de tamaño la mitad de la original) para los detalles horizontales, verticales y diagonales generados.

Análisis de dependencia de datos

La **dependencia de datos** en nuestro proyecto se limita a dos factores: **la iteración anterior y los pares contiguos de píxeles**.

Comenzando por lo referente a los pares, se debe a que el cómputo se realiza entre un píxel y el que tiene a cualquiera de sus lados. Por tanto en línea horizontal, si suponemos una imagen de 8x8 píxeles, en la primera fila dependerían el 1 y el 2 entre sí, el 2 y el 3 entre sí, y así consecutivamente. Como se puede ver, el píxel 4 no depende de los dos primeros, así como el píxel 2 no depende de los dos últimos. Esto mismo ocurrirá también en el plano vertical. Esto nos permitirá **dividir la imagen en sectores para el cálculo paralelo de cada uno de ellos**.

En relación a la dependencia de la iteración anterior, observamos que **por cada iteración se genera una nueva imagen que será la entrada de la siguiente iteración**. De esta forma, la primera iteración no depende de ninguna pero la segunda sí lo hará de la primera, así como la tercera dependerá de la imagen obtenida de la segunda iteración, y así consecutivamente hasta la última.

Propuestas de paralelización

En nuestro proyecto vamos a proponer la utilización tanto de paralelización de datos como de las tareas que realiza el algoritmo.

Para la paralelización de los datos vamos a utilizar las **extensiones de SIMD de Intel®** para que cada núcleo trabaje con múltiples píxeles a la vez, aumentando la eficiencia de las operaciones a realizar.

Para la paralelización de las tareas utilizaremos la **librería OpenMP**. Dividiremos la imagen a procesar en secciones de píxeles consecutivos de tamaño inversamente proporcional al número de núcleos disponibles para el cómputo. De esta manera cada núcleo se encargará de una zona distinta de la imagen.

También se probará a realizar una **implementación empleando MPI**, de modo que haya varios procesadores operando con una misma imagen.

Paralelización con SSE

Empleando SSE v4.1, es posible emplear **registros especiales de 128 bits**, que permiten **operar en paralelo**. Por lo tanto, empleando cuatro registros de 128 bits, que pueden almacenar 16 bytes, es posible **operar con 64 píxeles en una sola ejecución** del algoritmo.

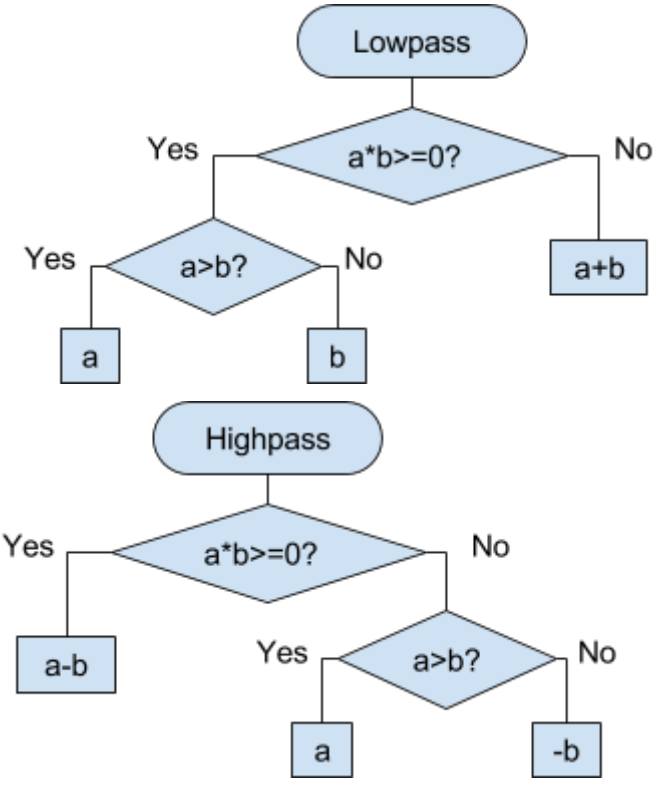
SSE no permite el uso de operaciones condicionales clásicas, los *if* no existen por así decirlo.

Sin embargo, existe una operación determinada *blend* desde SSE v4.1, que dados registros SSE A, B y X, de 16 bytes cada uno, ejecuta:

```
if (x[i] == 0) {
    salida[i] = a[i];
} else {
    salida[i] = b[i];
}
```

Como se puede observar en el código original adjunto en el punto "Descripción del algoritmo", **hay cuatro posibles funciones que emplear en base a las entradas**.

De este modo, **se puede definir la salida como un árbol**, dependiendo de las entradas:



Por lo tanto, para **adaptar el código secuencial a SSE**, se ha optado por **calcular todos los valores** posibles para las salidas (a, b, -b, a+b y a-b), y después **podar el árbol** mediante *blend*s.

```
#include <smmintrin.h>

void plhaar_sse(
        __m128i valuesA,
        __m128i valuesB,
        __m128i lowpass,
        __m128i highpass) {
    __m128i s = _mm_cmplt_epi8(valuesA, _mm_setzero_si128());
    __m128i t = _mm_cmplt_epi8(valuesB, _mm_setzero_si128());

    valuesA = _mm_sub_epi8(valuesA, s);
    valuesB = _mm_sub_epi8(valuesB, t);

    __m128i ifA = _mm_sub_epi8(valuesA, valuesB);
    __m128i aLessZero = _mm_xor_si128(s,
            _mm_cmplt_epi8(ifA, _mm_setzero_si128()));
    __m128i ifB = _mm_blendv_epi8(valuesA, valuesB, aLessZero);

    __m128i ifNotB = _mm_add_epi8(valuesB, valuesA);
    __m128i minusB = _mm_sub_epi8(_mm_setzero_si128(), valuesB);
    __m128i bLessZero = _mm_xor_si128(t,
            _mm_cmplt_epi8(ifNotB, _mm_setzero_si128()));
    __m128i ifNotA = _mm_blendv_epi8(minusB, valuesA, bLessZero);

    __m128i positive = _mm_xor_si128(s, t);
    highpass = _mm_add_epi8(_mm_blendv_epi8(ifA, ifNotA, positive), s);
    lowpass = _mm_add_epi8(_mm_blendv_epi8(ifB, ifNotB, positive), t);
}
```

Para realizar la implementación con SSE, en lugar de operar directamente con ensamblador, se ha optado por emplear los llamados **"intrinsics": macros que hacen de puente entre C y ensamblador**, pudiendo **especificar exactamente las instrucciones** de ensamblador a ejecutar, pero **permitiendo a GCC reordenar código** y **adaptarlo al número de registros** SSE de que el sistema dispone (ocho cuando el procesador está en modo protegido -32 bits-, y dieciséis en modo largo -64 bits-).

En estos intrinsics, se puede notar un patrón. Tomando por ejemplo **`_mm_add_epi8`**:

- **`_mm`**: prefijo para instrucciones MultiMedia (MMX, el predecesor de SSE)
- **`_add`**: operación de suma
- **`_epi8`**: los operandos packed-integer de 8 bits

Por lo tanto, **`_mm_add_epi8`** ejecuta una suma de dos registros que contienen enteros de 8 bits, y devuelve el resultado.

Cada pseudo-variable **`__m128i`**, que representa un registro de 128 bits que GCC escogerá, puede almacenar 16 bytes. Por tanto, cada hilo ejecutándose en un procesador compatible es capaz de ejecutar 16 operaciones con enteros de 8 bits en paralelo, lo que resulta en un speedup enorme, y por ende una mayor eficiencia, manteniendo el mismo número de procesadores.

Paralelización con OpenMP

Descripción del algoritmo paralelo e implementación

Refinando la propuesta anterior, vamos a discutir cómo hemos optado por paralelizar nuestro código.

Tal y como se comenta en el apartado anterior, la idea era dividir la imagen en pequeñas imágenes independientes para su procesado independiente. Esta habría sido una buena aproximación, pero se puede hacer de una forma más simple.

Nuestro código son en realidad dos bucles for anidados que computan la imagen primero en horizontal y luego en vertical. Por la escasa dependencia de datos que comentamos en la sección correspondiente del documento, es posible paralelizar ambos cálculos de forma que resulta innecesario establecer un orden concreto. Por ello hemos optado por la siguiente aproximación:

```
void plhaar2d_transform_naive_mp(
    const uint8_t * values,
    size_t width,
    size_t height,
    uint8_t * aver,
    uint8_t * hori,
    uint8_t * vert,
    uint8_t * diag
) {
    #pragma omp parallel for \
    collapse(2) \
    schedule(static) \
    shared(values, width, height, aver, hori, vert, diag)
    for (size_t y = 0; y < height; y += 2) {
        for (size_t x = 0; x < width; x += 2) {
            /*
             * Load four pixels into local thread-private variables
             */
            uint8_t topLeft     = values[(x + 0) + (y + 0) * width];
            uint8_t topRight    = values[(x + 1) + (y + 0) * width];
            uint8_t bottomLeft  = values[(x + 0) + (y + 1) * width];
            uint8_t bottomRight = values[(x + 1) + (y + 1) * width];

            /*
             * Compute PLHaar horizontally
             * Top left with top right, and bottom left with bottom right
             */
            uint8_t topLow, topHigh, bottomLow, bottomHigh;
            plhaar_int(topLeft,    topRight,    &topLow,    &topHigh);
            plhaar_int(bottomLeft, bottomRight, &bottomLow, &bottomHigh);

            /*
             * Now compute PLHaar vertically
             * Lowpass from top row with lowpass from bottom row, and
             * highpass from top row with highpass from bottom row.
             * The output is written to aver (average), hori (horizontal),
             * vert (vertical) and diag (diagonal) vectors.
             */
            size_t offset = y / 2 * width / 2 + x / 2;
            plhaar_int(topLow,  bottomLow,  aver + offset, hori + offset);
            plhaar_int(topHigh, bottomHigh, vert + offset, diag + offset);
        }
    }
}
```

Como podemos ver, la instrucción empleada finalmente es:

```
#pragma omp parallel for collapse(2) schedule(static)
```

Hemos omitido las directivas **`private()`** ya que el compilador automáticamente asigna como *private* los índices de los bucles *for*.

Para nuestra función hemos considerado que el valor óptimo para el planificador sería **`static`**, debido a que siempre será una carga fija. Es decir, el coste de cada iteración del bucle es siempre el mismo, con lo que no tiene sentido usar asignación dinámica.

Por último empleamos la cláusula **`collapse(2)`** para que el planificador distribuya la carga entre el total de iteraciones a computar agregando las de ambos bucles. De esta forma si el bucle exterior hiciera 10 iteraciones y el interior 4, sin esta cláusula se distribuiría la carga entre 10 *threads*, desperdiciando el resto si los hubiera -en nuestro caso los hay-. Con ella, la distribución se realiza entre 10 * 4 = 400 *threads*, lo que *a priori* desembocará en un mejor reparto de cálculo entre ellos.

La memoria necesaria para la ejecución de este algoritmo es mínimamente superior en términos absolutos que en la versión secuencial del algoritmo. Esto es debido a que cada *thread* necesitará su propio índice. Además, cada *thread* creará sus variables para almacenar el resultado aunque el resultado de todos los *threads* va directamente a su sitio en memoria ya que se les pasa la dirección de memoria donde deben almacenar el dato. Por tanto, la memoria necesaria para cada *thread* es de 28 bytes (7 variables * 4 byte/variable) multiplicado por las iteraciones que ejecute.

El código fuente está disponible en: https://github.com/socram8888/DWTAC/tree/master.

Benchmark usando OpenMP y/o SSE

A continuación se muestran gráficas comparando el rendimiento de diferentes configuraciones: **variando el número de núcleos, y variando la implementación**.

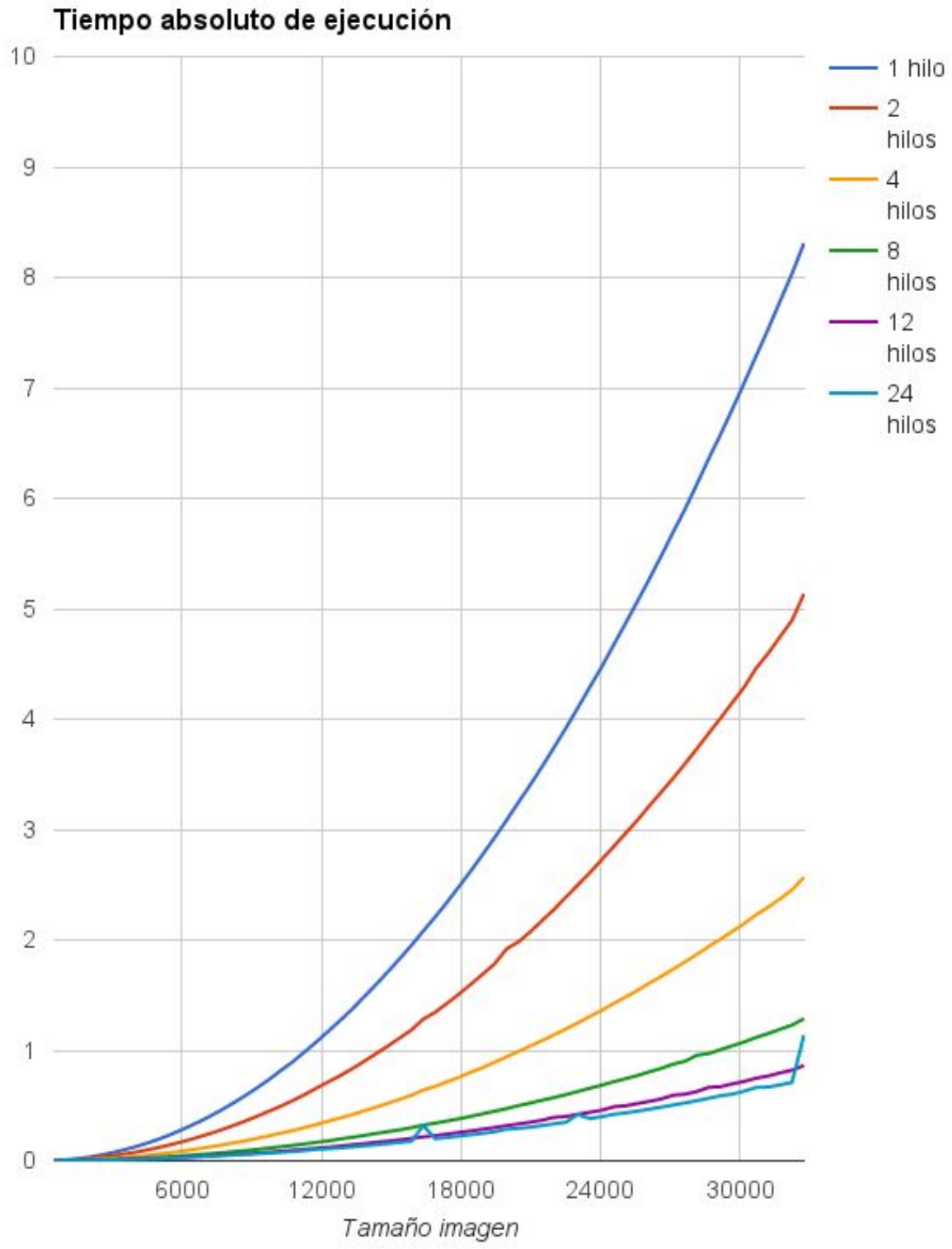
Para dibujar las siguientes gráficas, **se ha ejecutado cada configuración cuatro veces,** ejecutadas secuencialmente sin pausas, y se ha **cogido el mejor tiempo** de cada configuración.

Para cada ejecución, se ha generado una imagen nueva, generando así gran cantidad de I/O, evitando que el sistema cachease trozos del programa y que pudiera beneficiar a ejecuciones posteriores.

Variando el número de núcleos:

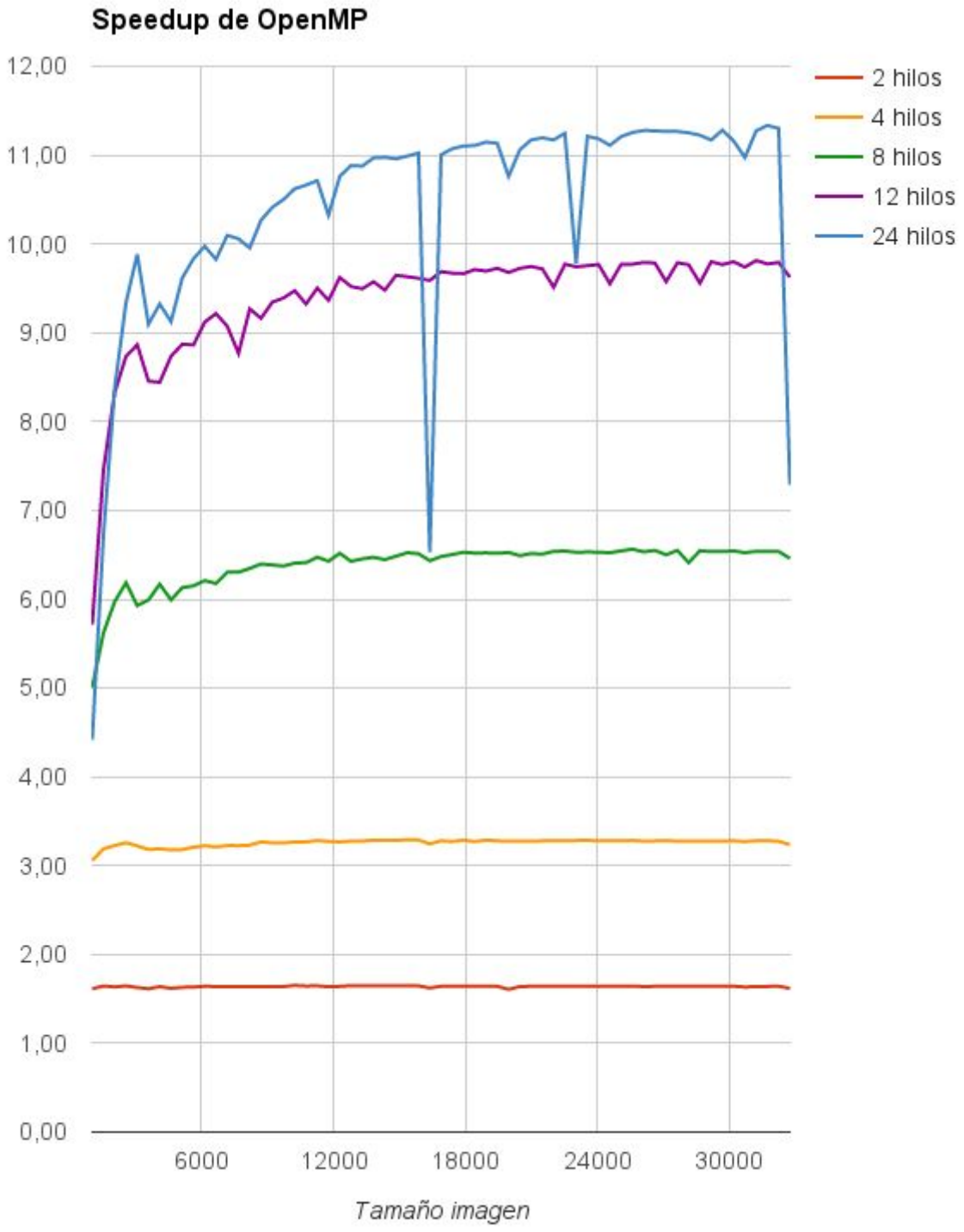
Ejecuta secuencial, OpenMP en 2, 4, 6, 8, 12, 16, 24, 32 y 128 hilos, con imágenes entre 512x512 y 32768x32768 píxeles.

```
#!/bin/bash

genimage() {
    SIZE=$1
    echo -ne "P5\n$SIZE $SIZE 255\n" >image.pgm
    head -c $(($SIZE*$SIZE)) /dev/zero >>image.pgm
}

maintimed() {
    ./main $* | grep 'Process' | cut -c 13-
}

CORES=(2 4 6 8 12 16 24 32 128)
SIZE=512

echo -ne "SIZE\tSEQ"
for I in ${CORES[@]}; do
    echo -ne "\tMP_$I"
done
echo ""

while [ $SIZE -le 32768 ]; do
    echo -ne "$SIZE\t"

    genimage $SIZE
    maintimed -t -i image.pgm -n 1 | tr -d '\n'

    for I in ${CORES[@]}; do
        genimage $SIZE
        export OMP_PROC_BIND=true
        export OMP_NUM_THREADS=$I
        export OMP_DYNAMIC=false

        echo -ne "\t"
        maintimed -t -m -i image.pgm -n 1 | tr -d '\n'
    done

    echo ""

    SIZE=$(($SIZE + 512))
done
```



En esta imagen se muestra el coste de la **implementación OpenMP naïve (sin SSE)** en segundos, variando el número de hilos.

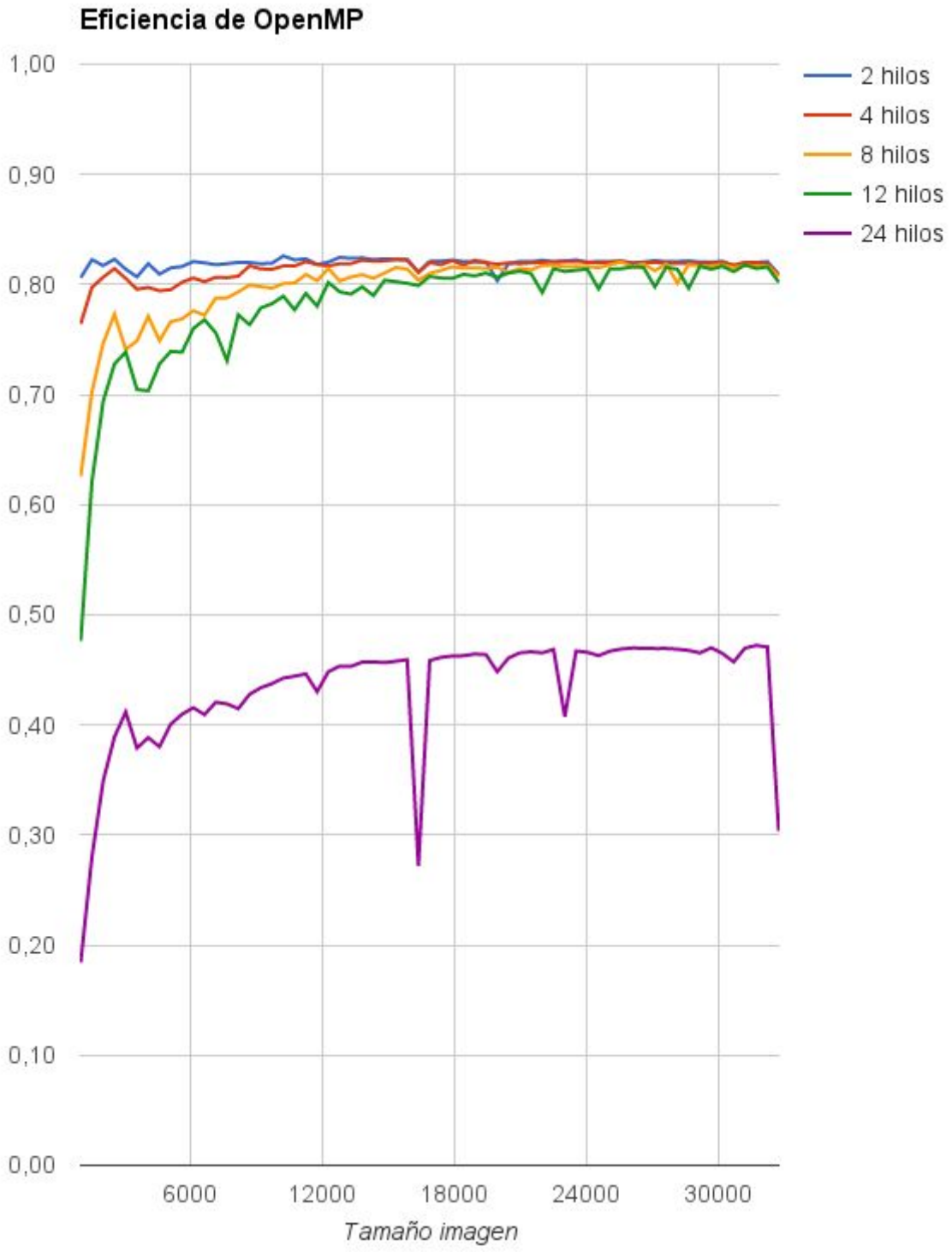
Para realizar esta gráfica se ha ejecutado las pruebas cuatro veces, y se han cogido los mejores tiempos de cada configuración, **excluyendo carga y guardado**.



Aquí podemos ver el speedup de OpenMP comparado con el algoritmo secuencial sin OpenMP, para imágenes comprendidas entre 1024x1024 y 32768x32768 píxeles.

Como se puede observar, el algoritmo **escala perfectamente**, y el número óptimo de procesos es igual al número de cores virtuales (vCPU).

Los picos decrecientes son debidos a desajustes de la memoria caché cuando el lado de la imagen es una potencia de 2. Esto provoca expulsiones de datos por lo que incrementa el coste. La solución habitual es sobredimensionar la matriz para evitar estos casos.

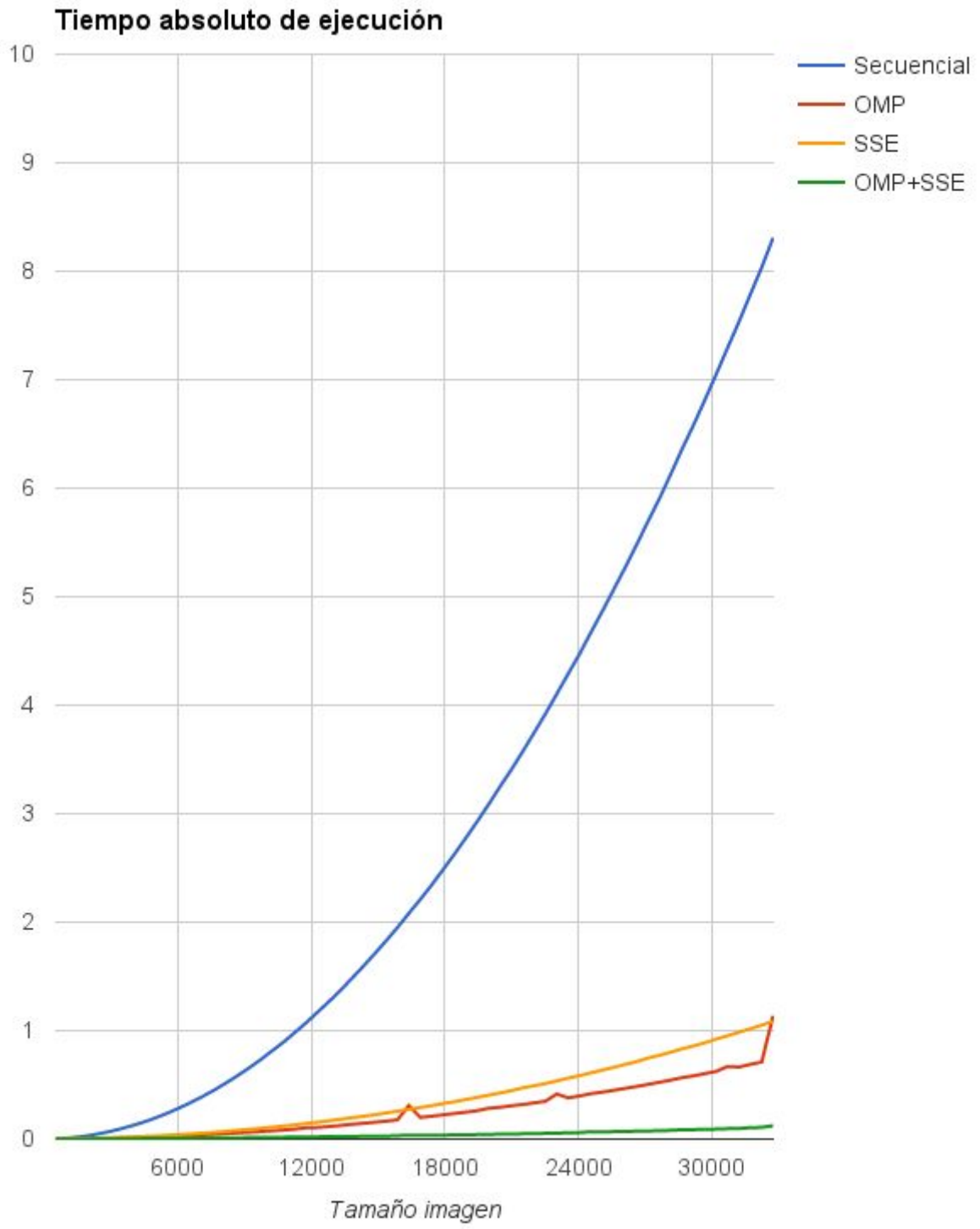


Aquí se puede comprobar cómo el coste de inicialización al tener más hilos penaliza el arranque, pero después se estabiliza. Debido a que 12 de los núcleos son "virtuales" (de HyperThreading), **la eficiencia de 24 núcleos no llega nunca a igualarse con 12 o menos**.

Variando la implementación

Ejecuta secuencial, OpenMP en 24 núcleos, SSE, y OpenMP más SSE en 24 núcleos, con imágenes entre 512x512 y 32768x32768 píxeles.

```
#!/bin/bash

genimage() {
    SIZE=$1
    echo -ne "P5\n$SIZE $SIZE 255\n" >image.pgm
    head -c $(($SIZE*$SIZE)) /dev/zero >>image.pgm
}

maintimed() {
    ./main $* | grep 'Process' | cut -c 13-
}

SIZE=512

echo -e "SIZE\tSEQ\tOMP\tSSE\tOMPSSE"

export OMP_PROC_BIND=true
export OMP_NUM_THREADS=24
export OMP_DYNAMIC=false

while [ $SIZE -le 32768 ]; do
    echo -ne "$SIZE\t"

    genimage $SIZE
    maintimed -t -i image.pgm -n 1 | tr '\n' '\t'

    genimage $SIZE
    maintimed -t -i image.pgm -n 1 -m | tr '\n' '\t'

    genimage $SIZE
    maintimed -t -i image.pgm -n 1 -s | tr '\n' '\t'

    genimage $SIZE
    maintimed -t -i image.pgm -n 1 -s -m

    SIZE=$(($SIZE + 512))
done
```



En esta imagen se puede ver el tiempo absoluto de ejecución empleando el algoritmo secuencial, **OpenMP en 24 núcleos** (como se ha determinado empíricamente que es óptimo para **`boe.uv.es`**), **SSE**, y empleando **OpenMP más SSE**.

En los tiempos se **excluye el tiempo de carga y guardado** de las imágenes procesados a disco. Para generar esta gráfica, se han realizado las pruebas cinco veces y cogido los mejores tiempos para cada configuración.

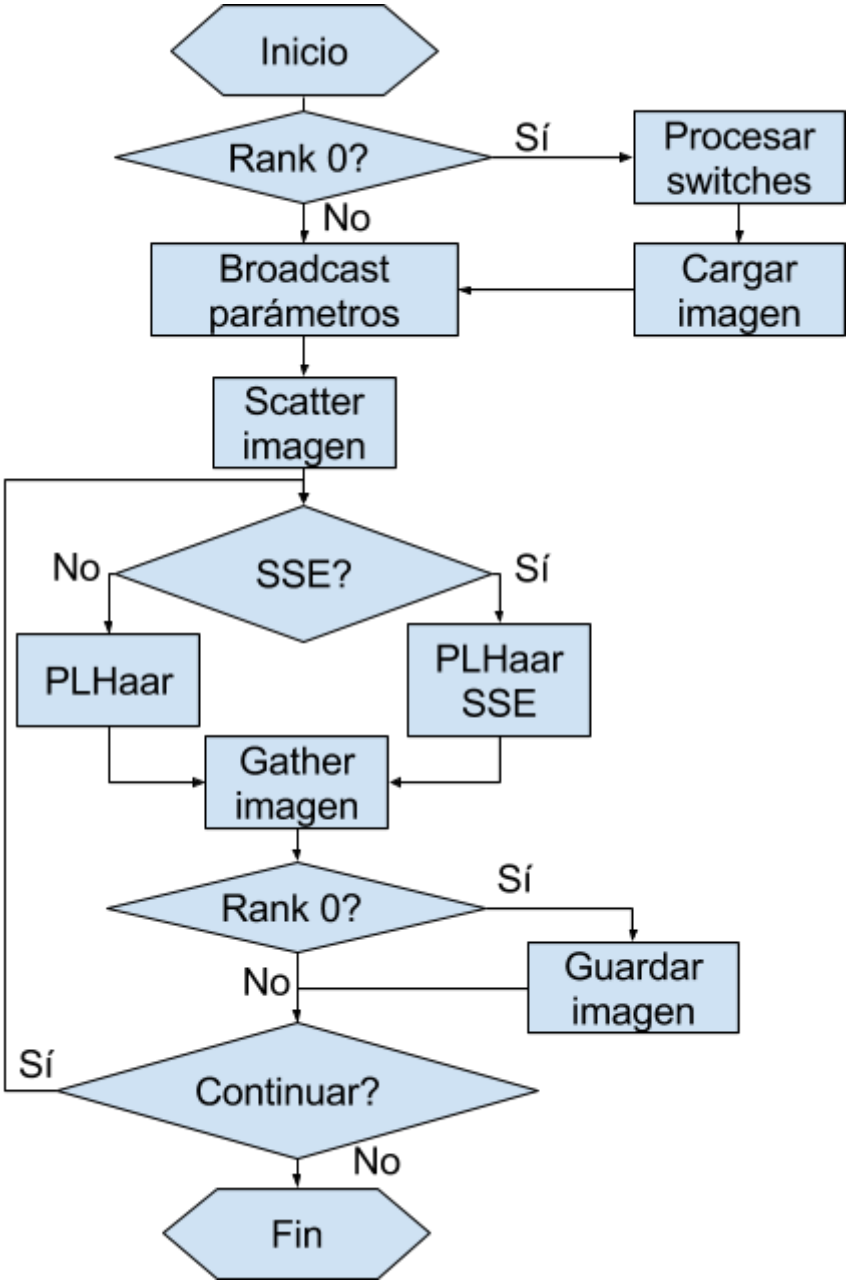
Como se puede observar, para **imágenes pequeñas**, debido al coste de inicialización de los hilos, es preferible **usar SSE sólo**, en un único núcleo. Para **imágenes más grandes**, es preferible emplear **OpenMP más SSE**, que obtienen un speedup superior al 5000% para imágenes mayores de 12000x12000 píxeles, y un speedup de 2500% para imágenes de tamaño habitual del orden de la decena de megapíxeles como se emplean en las cámaras de fotos digitales actuales.

Paralelización con MPI

Para realizar la paralelización, **el procesador maestro**, con rank cero, se va a encargar de **distribuir** las porciones de imagen al principio, y **reensamblarlas** al recibirlas para guardar al disco duro. Los esclavos, por otra parte, se van a encargar de realizar el procesamiento.

Para ello, se va a emplear **`MPI_Broadcast`** para comunicar a todos los parámetros de la imagen (tamaño, iteraciones, y si emplear SSE), después **se distribuirá por `MPI_Scatter`**, y se **recogerá por `MPI_Gather`** para cada una de las iteraciones.

El código fuente de la implementación está disponible en: https://github.com/socram8888/DWTAC/tree/mpi.



Versión reducida del ejecutable

```
int main(int argc, char ** argv) {
    // ... inicialización de MPI ...

    if (worldRank == 0) {
        // ... procesar switches ...
        netpbm_load(&image, inputfile); // Cargar imagen
    }

    MPI_Bcast(&image.width, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
    MPI_Bcast(&image.height, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
    MPI_Bcast(&iters, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&usesse, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);

    // ... mallocs de memoria de trabajo ...

    MPI_Scatter(image.data, chunkSize, MPI_BYTE,
            chunk, chunkSize, MPI_BYTE, 0, MPI_COMM_WORLD);

    for (int iter = 0; iter < iters; iter++) {
        size_t chunkLines = image.height / worldSize;
        size_t chunkSize = chunkLines * image.width;

        if (usesse)
            plhaar2d_transform_sse(chunk, image.width, chunkLines, chunkAvg,
                    chunkHorizontal, chunkVertical, chunkDiagonal);
        else
            plhaar2d_transform_naive(chunk, image.width, chunkLines, chunkAvg,
                    chunkHorizontal, chunkVertical, chunkDiagonal);

        if (worldRank == 0) {
            MPI_Gather(chunkHorizontal, chunkSize / 4, MPI_BYTE,
                    sub.data, chunkSize / 4, MPI_BYTE, 0, MPI_COMM_WORLD);
            // ... guardar horizontal ...

            MPI_Gather(chunkVertical, chunkSize / 4, MPI_BYTE,
                    sub.data, chunkSize / 4, MPI_BYTE, 0, MPI_COMM_WORLD);
            // ... guardar vertical ...

            MPI_Gather(chunkDiagonal, chunkSize / 4, MPI_BYTE,
                    sub.data, chunkSize / 4, MPI_BYTE, 0, MPI_COMM_WORLD);
            // ... guardar diagonal ...

            MPI_Gather(chunkAvg, chunkSize / 4, MPI_BYTE,
                    sub.data, chunkSize / 4, MPI_BYTE, 0, MPI_COMM_WORLD);
            // ... guardar media ...
        } else {
            MPI_Gather(chunkHorizontal, chunkSize / 4, MPI_BYTE,
                    NULL, 0, MPI_BYTE, 0, MPI_COMM_WORLD);
            MPI_Gather(chunkVertical, chunkSize / 4, MPI_BYTE,
                    NULL, 0, MPI_BYTE, 0, MPI_COMM_WORLD);
            MPI_Gather(chunkDiagonal, chunkSize / 4, MPI_BYTE,
                    NULL, 0, MPI_BYTE, 0, MPI_COMM_WORLD);
            MPI_Gather(chunkAvg, chunkSize / 4, MPI_BYTE,
                    NULL, 0, MPI_BYTE, 0, MPI_COMM_WORLD);
        }

        image.width /= 2; image.height /= 2;
    }

    MPI_Finalize();
    return 0;
}
```

Análisis de costes

Costes de computación

El coste de computación en términos de accesos a memoria, es el mismo que para la versión original secuencial, es decir:

$$\frac{4n}{3}(1 - \frac{1}{4^k})$$

Donde k es el número de iteraciones, y n el número de píxeles, es decir, el resultado de multiplicar anchura por altura.

La demostración matemática se encuentra en el CP1.

Costes de comunicaciones

El algoritmo recibe todos los datos en el rank 0 tras cada iteración para poder redistribuir la media (LL) entre los nodos.

Sabemos que en la primera iteración tiene que enviar la imagen entera, y recibir tres cuartos de la imagen original (HL, LH y HH). Después, volverá a recibir tres cuartos de la cuarta parte de la original (de nuevo, HL, LH y HH), y por último, se recibirá el LL restante:

$$n + \frac{3n}{4} + \frac{3n}{4\times4} + \ldots + \frac{3n}{4\times\ldots\times4} n + \frac{n}{4\times\ldots\times4}$$

Matemáticamente se puede expresar el coste de comunicaciones en bytes enviados y recibidos como:

$$n + \frac{n}{4^k} + \sum_{i=1}^{k} \frac{3n}{4^i}$$

Donde k es el número de iteraciones, y n el número de píxeles, es decir, el resultado de multiplicar anchura por altura; y donde el primer término de la suma es el coste de enviar la imagen entera, el segundo de recibir el LL al terminar, y el sumatorio el coste de recibir todas las HH, LH y HL.

Si simplificamos, vemos que el coste en bytes de red, se nos queda como:

$$2n$$

Ya que básicamente tenemos que enviar la imagen entera, y recibir la imagen entera transformada.

Benchmark usando MPI

En esta sección se puede ver el rendimiento de ejecución de algunas de las configuraciones probadas. Para todas, el SSE está deshabilitado.

Para dibujar las siguientes gráficas, se ha dividido el procesado en cinco partes ejecutadas secuencialmente sin pausas: cargado, scatter, procesado, gather y guardado.

Se ha cogido el mejor subtiempo de un total de tres ejecuciones, y sumado cargado, scatter, procesado y gather, excluyendo los tiempos de cargado e imagen de disco duro.

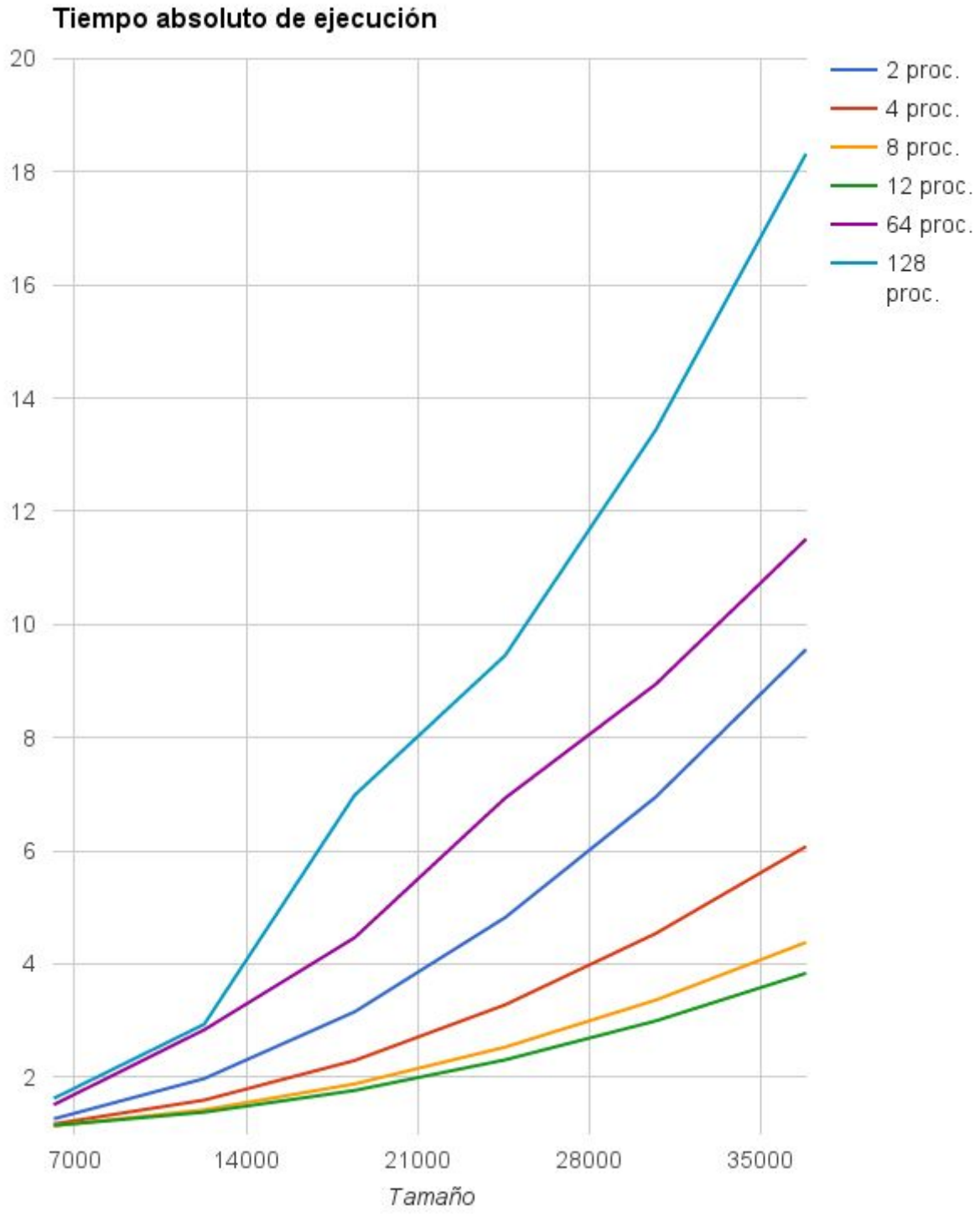
Las pruebas se han realizado en el servidor **`boe.uv.es`**, empleando los ocho sistemas auxiliares **`compute-0-0`** a **`compute-0-7.`**

```
boe.uv.es slots=24
compute-0-0 slots=24
compute-0-1 slots=24
compute-0-2 slots=24
compute-0-3 slots=24
compute-0-4 slots=24
compute-0-5 slots=24
compute-0-6 slots=24
compute-0-7 slots=24
```

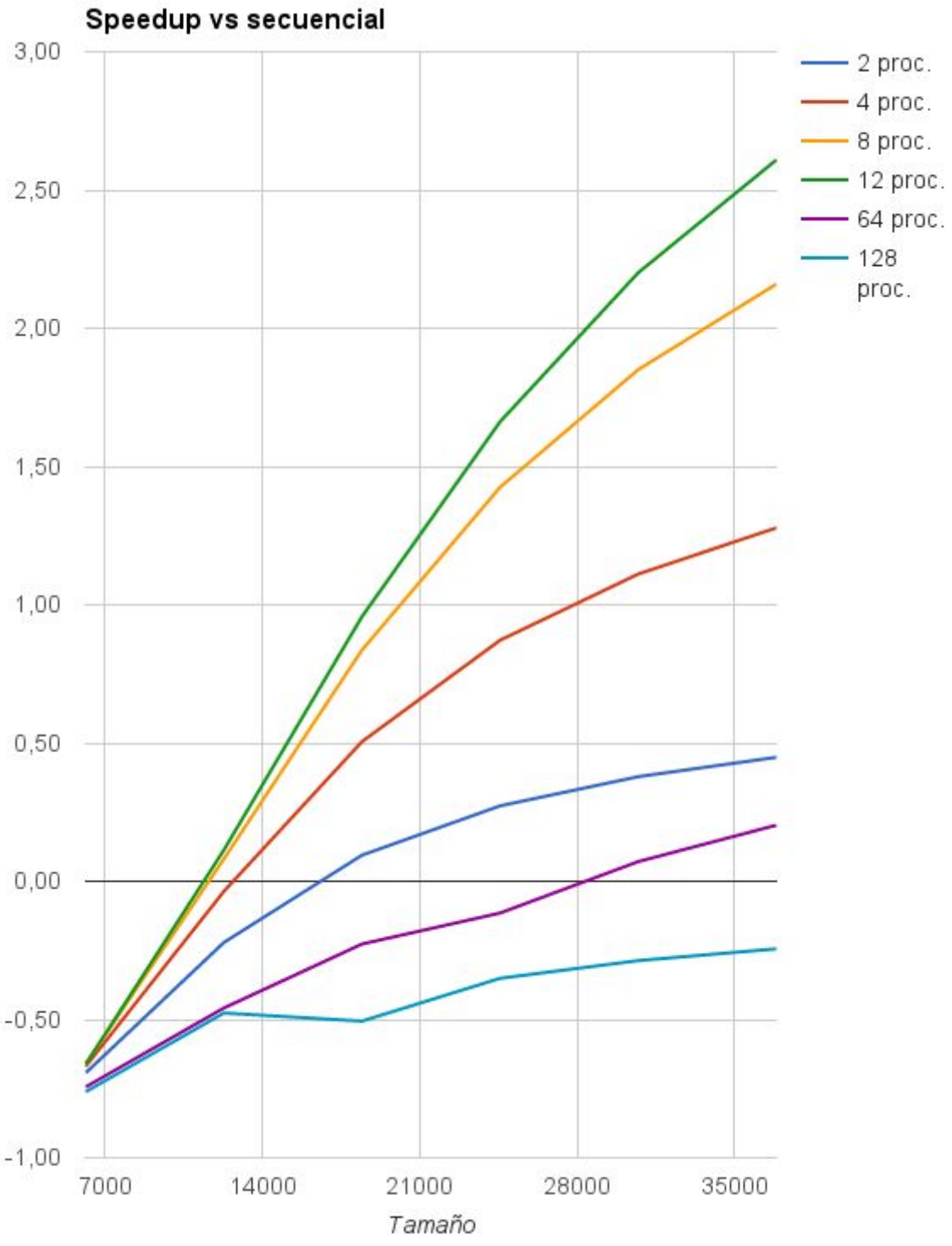
Se ha decidido emplear los 24 slots de cada sistema, es decir, incluyendo los núcleos virtuales del HyperThreading, para poder ejecutar pruebas con 128 núcleos y ver la escalabilidad del sistema.

Para cada ejecución, se ha generado una imagen nueva, generando así gran cantidad de I/O, evitando que el sistema cachease trozos del programa y que pudiera beneficiar a ejecuciones posteriores.

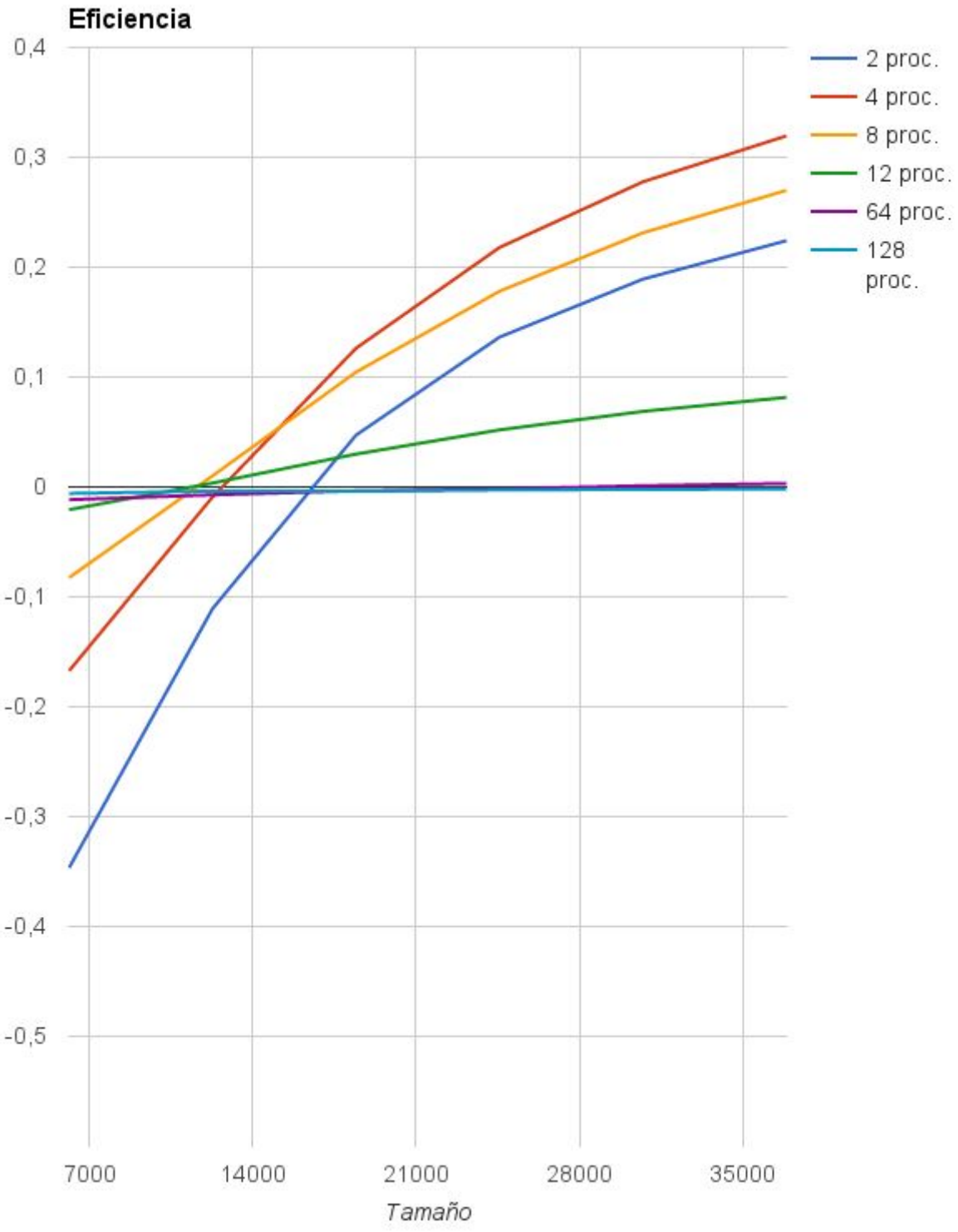
```
#!/bin/bash
CORES=(2 4 8 12 64 128)

genimage() {
	SIZE=$1
	echo -ne "P5\n$SIZE $SIZE 255\n" >image.pgm
	head -c $(($SIZE*$SIZE)) /dev/zero >>image.pgm
}

echo -e "SIZE\tCORES\tINIT\tLOAD\tSCATTER\tPROCESS\tGATHER\tSAVE"
SIZE=6144
while [ $SIZE -le 40000 ]; do
	for I in ${CORES[@]}; do
		echo -ne "$SIZE\t$I\t"
		genimage $SIZE

		# Cinco iteraciones
		mpirun -np $I --hostfile hostfile \
		./main -t -i image.pgm -n 5 | cut -c 13- | tr '\n' '\t'

		echo ""
	done
	SIZE=$(($SIZE + 6144))
done
```



Aquí se puede observar el tiempo absoluto de ejecución para algunas de las configuraciones probadas. Como se puede observar, cuando **el número de procesos excede de 24** y empieza a demandar red, el overhead de esta comunicación **anula la ventaja de emplear más procesadores**.

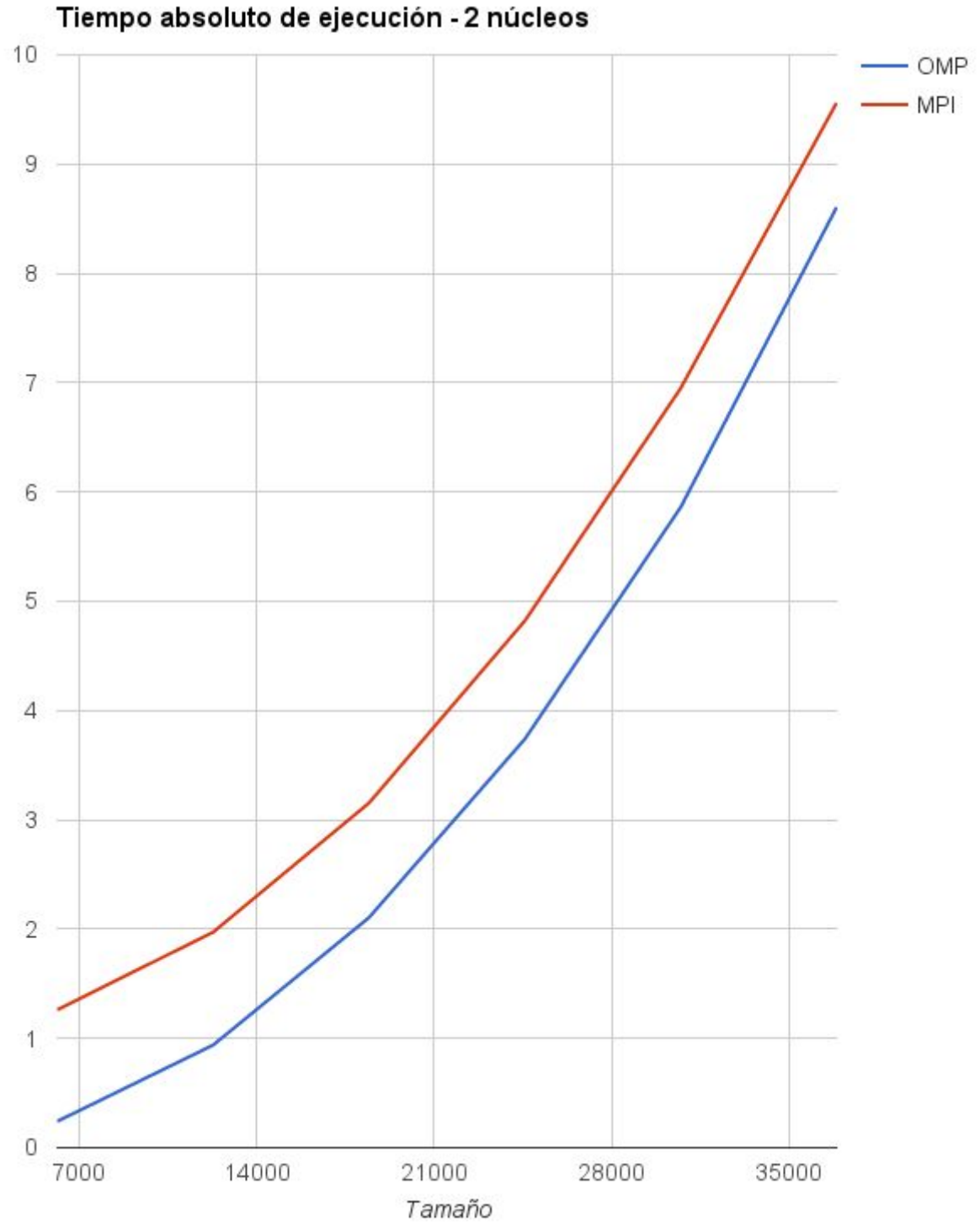


Aquí se puede observar el speed-up para algunas de las configuraciones probadas, comparadas con la implementación naïve (sin SSE). **Únicamente es ventajoso con un número de procesadores reducidos**, y como veremos a continuación, en estos funciona **peor que OpenMP**.

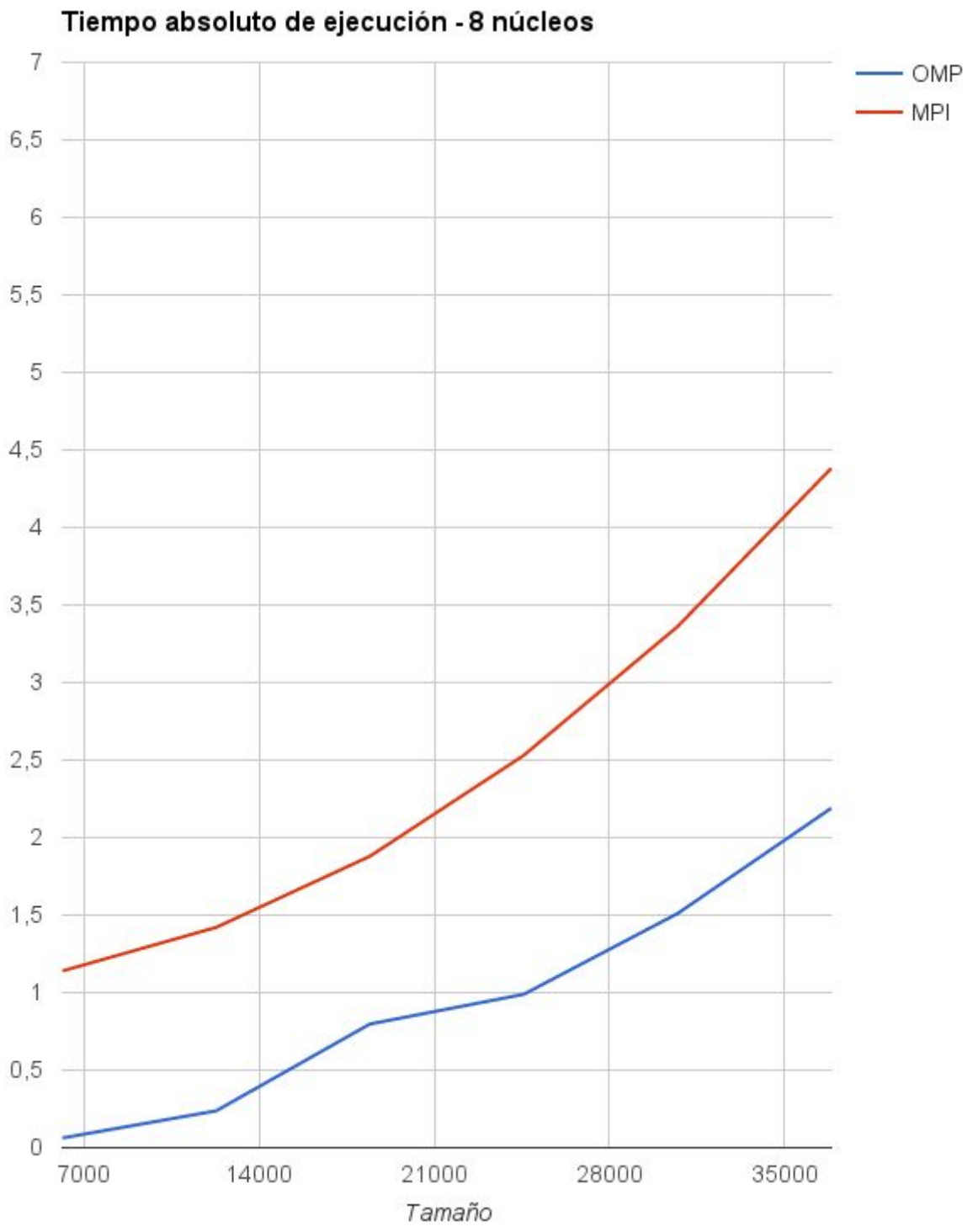


En este gráfico se puede observar una escasa variación de la eficiencia en términos generales, aunque se pueden diferenciar dos grupos: las ejecuciones con menos de 24 procesos y con más de 24 procesos. Este es el punto a partir del cual se transmiten datos por la red Gigabit Ethernet entre boe y las máquinas *compute*. En ese momento se introduce un gran *overhead* que impide el aumento de la eficiencia con un mayor número de procesos.

Aprovechando que MPI se está con números pequeños ejecutando en el mismo **`boe.uv.es`**, se ha considerado oportuno comparar MPI con OpenMP, ambos sin SSE, para comprobar el overhead extra que impone la copia de memoria y la comunicación interprocesos de MPI.



En esta imagen se puede observar cómo el overhead en todos las imágenes probadas para ocho núcleos constituye una penalización de alrededor de un segundo en los tiempos de ejecución.



En esta imagen se puede ver cómo la **penalización del overhead** empeora conforme aumenta el tamaño de la imagen: comienza siendo de un segundo, que es el tiempo principalmente empleado en iniciar las comunicaciones (`MPI_Init`), y cómo conforme aumenta el tamaño, OpenMP se va distanciando de MPI debido al **coste de la copia de memoria y al paso de mensajes**.

Autoevaluación

CP1

Tiempo dedicado al CP1: cuatro horas.

Concepto evaluado	Valor
Completitud	0.9
Claridad de la redacción	1
Objetivo y utilidad del algoritmo	0.9
Descripción del algoritmo	0.8
Estructura y tipos de datos, así como la cantidad de memoria utilizada por el algoritmo	0.9
Análisis del coste del algoritmo	0.9
Análisis de dependencia de datos	0.9
Propuestas de paralelización	0.9
Referencias	1
Formato del documento enviado	0.9
Total (sobre 10)	**9.1**

CP2

Tiempo dedicado al CP2: cinco horas.

Concepto evaluado	Valor
Completitud	0.6
Claridad de la redacción	0.8
Referencia al análisis de dependencias de la CP1	0.3
Discusión sobre los paradigmas de programación aplicables	0.5
Análisis del coste computacional del algoritmo paralelo	1
Análisis de las necesidades de memoria del algoritmo paralelo	0.6
Propuesta de directivas de OpenMP y organización del código para la paralelización	1.5
Estimación analítica de la aceleración, eficiencia y escalabilidad	-
Referencias nuevas	0
Formato del documento	0.5
Total (sobre 10)	

P1S23

Tiempo dedicado al P1S23: veinticinco horas.

Concepto evaluado	Valor
Completitud	0.9
Claridad de la redacción	0.7
Código del algoritmo final con OpenMP	1
Presentación de los resultados mediante tablas y/o gráficos	0.9
Completitud de las tallas y procesadores empleados	0.8
Estudio de diversas opciones de planificación y variables	0.5
Estudio experimental de velocidad, aceleración y eficiencia.	1.7
Justificación de los datos experimentales	0.5
Total (sobre 10)	

CP3

Tiempo dedicado al CP3: cinco horas.

Concepto evaluado	Valor
Completitud	0.4
Claridad de la redacción	0.5
Referencia al análisis de dependencias	0.2
Discusión sobre los paradigmas de programación aplicables	0
Análisis de costes de computación y comunicaciones del algoritmo paralelo	1.8
Análisis de necesidades de memoria y distribución de los datos del algoritmo paralelo (0-2)	1.6
Propuesta de rutinas de MPI para la paralelización del algoritmo	1.2
Estimación analítica de su aceleración, eficiencia y escalabilidad (0-1)	0.9
Referencias	0.4
Formato del documento sin comprimir	0.5
Total (sobre 10)	

P2S23:

Tiempo dedicado al P2S23: seis horas.

Concepto evaluado	Valor
Completitud del estudio experimental	0.8
Claridad de la redacción y presentación de los datos	0.9
Código del algoritmo final con MPI	0.6
Presentación de los resultados resultantes mediante gráficas	1
Completitud de las tallas y procesadores	0.8
Estudio de diversas opciones de distribución de datos	0.6
Estudio experimental de velocidad, aceleración y eficiencia	1.6
Estudio experimental de escalabilidad	0.9
Justificación de los datos experimentales obtenidos en función del algoritmo y la arquitectura del computador	0.8
Total (sobre 10)	

Referencias

Senecal, J. Lindstrom, P. Duchaineau, M. Joy, K. (2004). “An Improved N-Bit to N-Bit Reversible Haar-Like Transform”, UCRL-CONF-204061, disponible en: https://e-reports-ext.llnl.gov/pdf/307631.pdf.

Senecal, J. (2005). “Length-Limited Data Transformation and Compression”, UCRL-TH-212406, disponible en: https://e-reports-ext.llnl.gov/pdf/320388.pdf.

Intel Intrinsics Guide, disponible en: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

MPI Routines Documentation: http://www.mpich.org/static/docs/v3.2/www3/.

Programación Distribuida, Universidad de Granada: http://lsi.ugr.es/jmantas/pdp/ayuda/mpi_ayuda.php?ayuda=MPI_Gather.

Discrete Wavelet Transform, Wikipedia: https://en.wikipedia.org/wiki/Discrete_wavelet_transform.