# Caso de Estudio 2 - LPP
# Filtrado de imágenes con MPI

**Alumno :** *Álvaro Jesús Graciá Gil*

**Grupo :** *4IC11*

**Curso:** *2018 / 2019*

## DECISIONES DE DISEÑO Y MODIFICACIONES DEL CÓDIGO BASE

A continuación se van a detallar los distintos cambios y enfoques aplicados a la paralelización del código propuesto, el cual permite aplicar un filtrado de desenfoque a imágenes en formato *PPM (Portable PixMap).*

### PARAMETRIZACIÓN DE "RADIO" y "PASOS"

Para facilitar las pruebas y aumentar la genericidad del programa, se han añadido dos parámetros de invocación para determinar el radio y el número de pasos a aplicar.

```
int radius = DEFAULT_DIST_RADIO;
int steps = DEFAULT_NUM_PASOS;
...
if(argc >= 5)
        sscanf(argv[4], "%d", &steps);

if(argc >= 6)
        sscanf(argv[5], "%d", &stepMode);
...
```

### DIRECTIVAS DEL PREPROCESADOR Y FUNCIONES PARA DEPURACIÓN

Se han añadido unas directivas que permiten variar el programa en tiempo de compilación para cambiar algunos parámetros y para mostrar información que puede resultar útil en tareas de depuración.

| | |
|---|---|
| `#define INFO` | Muestra el tamaño de la imagen, el radio, la cantidad de pasos, el nº de procesos MPI, el número de hilos OPENMP (si procede) y el modo de comunicaciones entre pasos. |
| `#define DEBUG` | Muestra información para depuración. |
| `#define DEBUG_EXTRA` | Muestra mucha más información para depuración. No se recomienda con una imagen muy grande. |
| `#define FAKE_ROWS X` | Permiten falsear el número de filas y de columnas que se procesarán de la imagen. |
| `#define FAKE_COLS Y` | |

Además se ha incluido una función que permite visualizar vectores de enteros.

```
void printVector(char *name, int *v, unsigned int n)
...
Ejemplo:
printVector("counts", v, 4); → counts = [1, 2, 3, 4]
```

## REPARTO DE CARGA ENTRE PROCESOS

El reparto se realiza por bloques de filas completas. Se centra en distribuir de la forma más equilibrada posible la cantidad de filas a calcular por cada proceso.

En caso de que por la cantidad de filas haya una distribución heterogénea, al proceso con rango 0 se le repartirá menor carga que a los otros, con el fin de disminuir los posibles retardos por la sobrecarga de las comunicaciones que sólo le afectan a él.

## TIPOS MPI

Para facilitar las tareas de reparto a nivel conceptual se han creado dos tipos derivados para MPI.

El tipo píxel, que es un tipo estructura formada por 3 char consecutivos.

```
MPI_Datatype pxType;
int blockLen[] = { 3 };
MPI_Aint indices[] = { offsetof(struct pixel, r) };
MPI_Datatype types[] = { MPI_CHAR };
MPI_Type_create_struct(1, blockLen, indices, types, &pxType);
MPI_Type_commit(&pxType);
…
MPI_Type_free(&pxType);
```

offsetof es una macro de la librería *stddef.h* que facilita el cálculo de direcciones.

Además, también se ha creado un tipo fila, el cual se trata de un vector de píxeles cuya dimensión equivale al número de columnas.

```
MPI_Datatype rowType;
MPI_Type_vector(1, colsCount, 1, pxType, &rowType);
MPI_Type_commit(&rowType);
…
MPI_Type_free(&rowType);
```

## OPENMP EN LA FUNCIÓN FILTRO

Para aprovechar el máximo todas las prestaciones de *Kahan*, resulta conveniente realizar una paralelización mixta con MPI y OpenMP.

Mientras que MPI se utiliza para dividir la imagen en bloques de filas consecutivas y repartirlas a los distintos nodos, OpenMP se encarga de procesar esas filas de forma paralela en cada nodo.

En busca de el mejor rendimiento se ha optado por un sistema de tareas, ya que aunque la carga es bastante simétrica, las filas cercanas a los límites tienen un poco menos de carga computacional.

```
void Filtro(int radio, struct pixel **ppsImagenOrg, struct pixel **ppsImagenDst,
int **ppdBloque, int n, int nMax, int nOffset, int m) {
        …
        for (i = nOffset; i < (n+nOffset); i++)
        {
                //Paralelización OPENMP por tareas
                #pragma omp task firstprivate(i) private(j, k, l, resultado, tot, v)
                        shared(nOffset, n, nMax, m, radio, ppsImagenOrg, ppsImagenDst)
                for (j = 0; j < m; j++)
                { … }
        }
        #pragma omp taskwait
        return;
}
```

En el anexo al final del documento se encuentra el código completo

**CAMBIO DE LA ESTRUCTURA DE LA FUNCIÓN FILTRO**

La función “Filtro” sugerida para el caso de estudio contempla que todas las iteraciones de los distintos pasos se realicen dentro de la propia función. Paralelizar el código de esta forma es posible y potencialmente correcto, pero fuerza a añadir la lógica de las comunicaciones dentro del propio algoritmo de filtrado.

Por ello, en esta implementación se ha optado por reducir al máximo la función “Filtro” para que únicamente sea el algoritmo de filtrado de la imagen.

Para conseguirlo, se ha optado por sacar la reserva y generación de la matriz ppdBloque (que contiene los pesos a aplicar en el desenfoque) a una función externa `(int createPpdBloque(int ***ppdBloque, int radio))` que se llamará una única vez antes de comenzar el filtrado de la imagen.

Además, la iteración sobre los pasos también se ha extraído, ya que realmente su único cometido es aplicar el algoritmo un número determinado de veces.

**FUNCIÓN PARA RESERVA DE MEMORIA**

Esta función reserva los buffers de memoria origen (ImgOrg) y destino (ImgDst) del tamaño correcto según en qué proceso se encuentren. La única excepción es que el buffer de origen en el proceso con rango 0 es reservado al leer la imagen (*función lee_ppm*)

```
int allocateImageBuffers(struct pixel ***ImgOrg, struct pixel ***ImgDst, int rank,
    int rowsCountOrg, int rowsCountDst, int colsCount)
```

**SISTEMA DE COMUNICACIONES ENTRE PASOS**

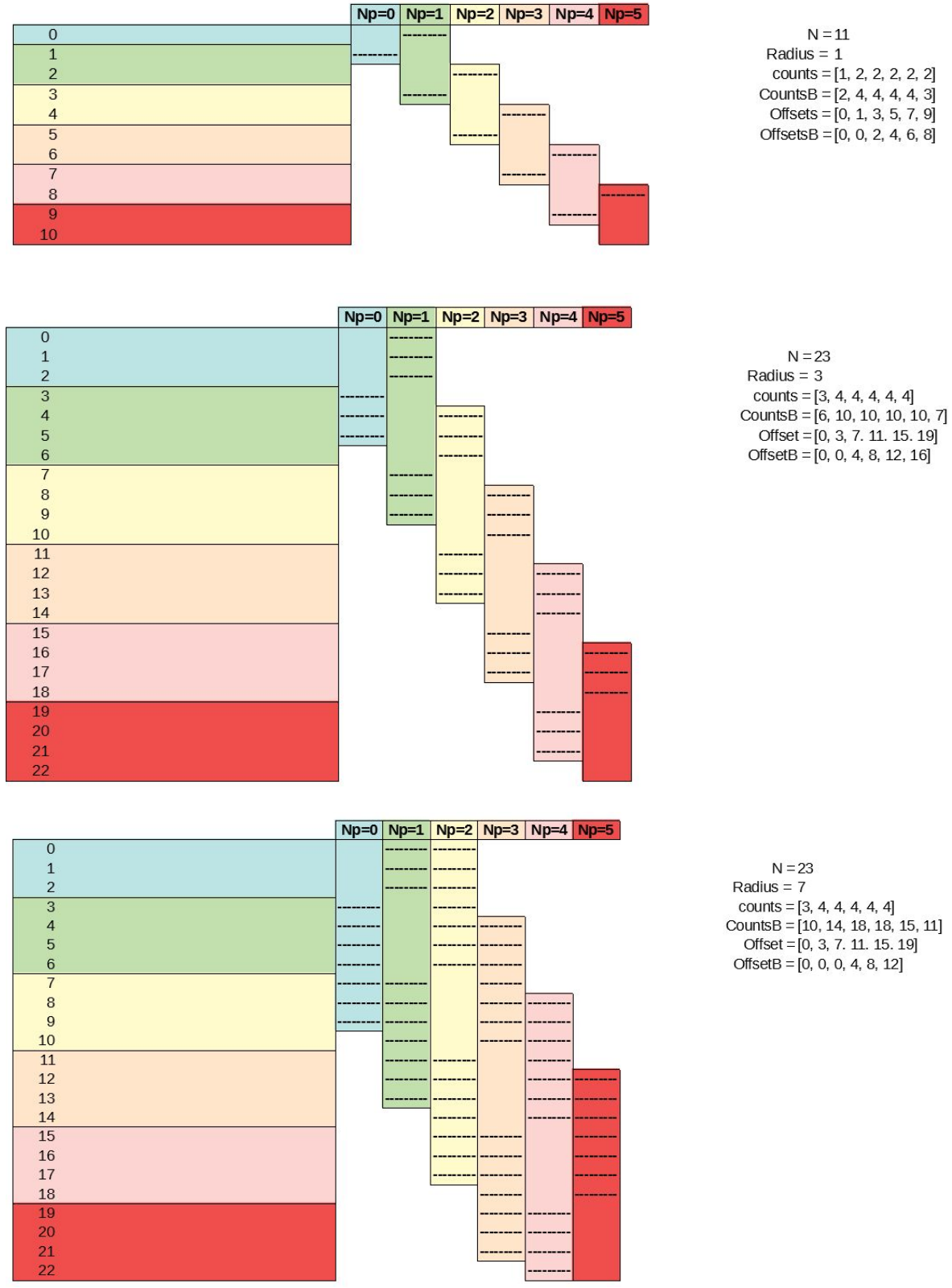
La implementación propuesta en este caso de estudio contempla un parámetro adicional que indica el sistema de comunicaciones a emplear entre pasos.
Se contemplan dos posibilidades:

- **Empleando Gatherv/ScatterV:** De la misma forma que ocurre tras acabar el filtrado de cada parte, todos los procesos envían su resultado al proceso 0 mediante una llamada a la función *GatherV.* Resulta muy sencillo volver a repartir el resultado mediante una llamada a *ScatterV* y continuar con el algoritmo.

  El problema de emplear este sistema es que obliga a que haya una espera global entre todos los procesos a que hayan terminado su cómputo y así poder juntar los resultados.
  Además añade una sobrecarga en el proceso 0, ya que debe recibir todos los resultados y volver a mandarlos a cada proceso.

- **Empleando Isend/Irecv:** Para evitar los problemas detallados en el punto anterior, se diseñó este sistema. Usando funciones no bloqueantes (para evitar orden e interbloqueos) se envían y se reciben las filas necesarias del proceso que las posee al que las necesita, de forma que no se involucra a ningún proceso más que a los directamente implicados.

Aquí se pueden ver varios ejemplos de cómo deben ser los envíos entre procesos variando el radio y el tamaño de la imagen:



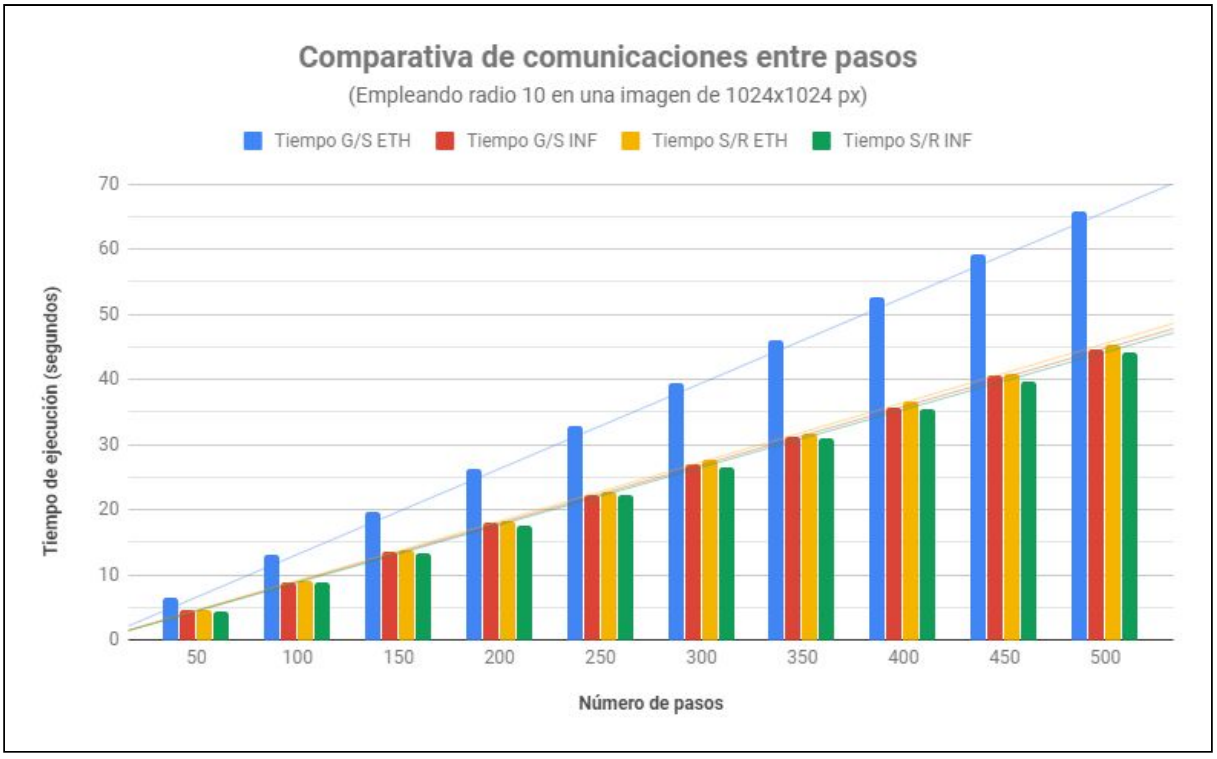

Las filas con trazos discontinuos son las que se refieren al radio.

- counts contiene la cantidad de filas que le corresponden a cada proceso.
- countsB contiene la cantidad de filas y el radio de cada proceso.
- offset contiene el desplazamiento donde empieza la primera fila a procesar.
- offsetB contiene el desplazamiento donde empieza la primera fila (incluyendo el radio).

A continuación se muestran y analizan unos resultados experimentales que comparan los dos sistemas de comunicaciones entre pasos en *Kahan* y valorando cómo afecta el emplear tecnología Infiniband o Ethernet.

| Pasos | Tiempo G/S ETH | Tiempo G/S INF | Tiempo S/R ETH | Tiempo S/R INF | Speedup Sr frente a Gs en ETH | Speedup Sr frente a Gs en INF |
|---|---|---|---|---|---|---|
| 50 | 6,578785 | 4,536206 | 4,588972 | 4,44795 | 1,43360757 | 1,01984195 |
| 100 | 13,206381 | 8,939626 | 9,031767 | 8,873781 | 1,462214537 | 1,007420174 |
| 150 | 19,710759 | 13,517932 | 13,755544 | 13,318782 | 1,432931987 | 1,014952568 |
| 200 | 26,304067 | 17,967095 | 18,319699 | 17,586701 | 1,435835108 | 1,021629639 |
| 250 | 32,919523 | 22,360688 | 22,771115 | 22,255774 | 1,445670227 | 1,004714013 |
| 300 | 39,451711 | 26,983525 | 27,596159 | 26,421877 | 1,429608773 | 1,02125693 |
| 350 | 45,997061 | 31,219453 | 31,76177 | 30,881574 | 1,448189474 | 1,01094112 |
| 400 | 52,57858 | 35,761721 | 36,594171 | 35,493543 | 1,436802052 | 1,007555684 |
| 450 | 59,19954 | 40,558201 | 40,94353 | 39,809942 | 1,445882658 | 1,018795782 |
| 500 | 65,704735 | 44,690903 | 45,44238 | 44,188501 | 1,445891148 | 1,011369519 |

Todos los resultados son obtenidos en *Kahan* empleando un radio 10.



Como se podía aproximar de forma teórica, el sistema de comunicaciones Isend/Irecv obtiene mejores tiempos en general debido a su diseño.

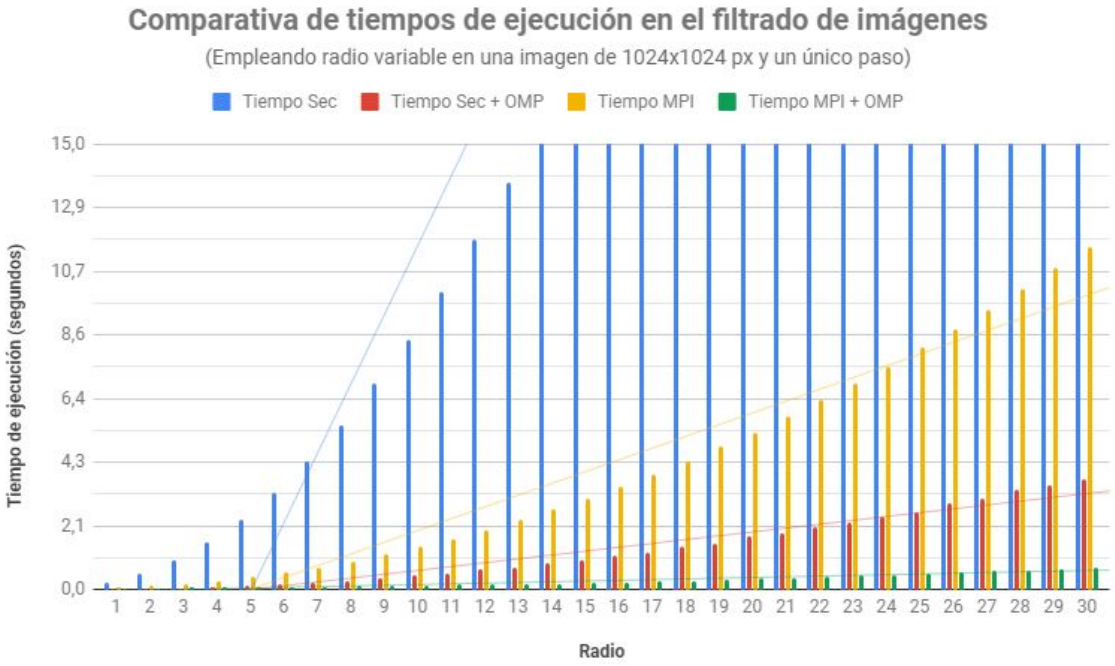
Aunque es importante destacar que con un tamaño de imagen medio como es 1024x1024px y empleando una red Infiniband, la diferencia es mínima (un speedup medio de 1,01 → 1%). El cambio se aprecia cuando se utiliza una red Ethernet. En este caso los tiempos que se obtienen son prácticamente idénticos a utilizar una red Infiniband, lo cual permitiría reducir en gran medida los costes a la hora de desplegar el hardware.
Por lo que se concluye que utilizar un patrón Isend/Irecv con Ethernet frente a Gatherv/Scatterv con Ethernet proporciona un speedup medio de **1,44 → 44%**.

## EVALUACIÓN DEL ALGORITMO PARALELIZADO

Se va a proceder a comparar distintos parámetros de la implementación paralela frente a la secuencial, tanto con OpenMP como sin él. Para ello se va a variar el radio a aplicar sobre una imágen de 1024x1024px y empleando el cluster *Kahan* al completo (6 nodos y 32 núcleos por nodo).

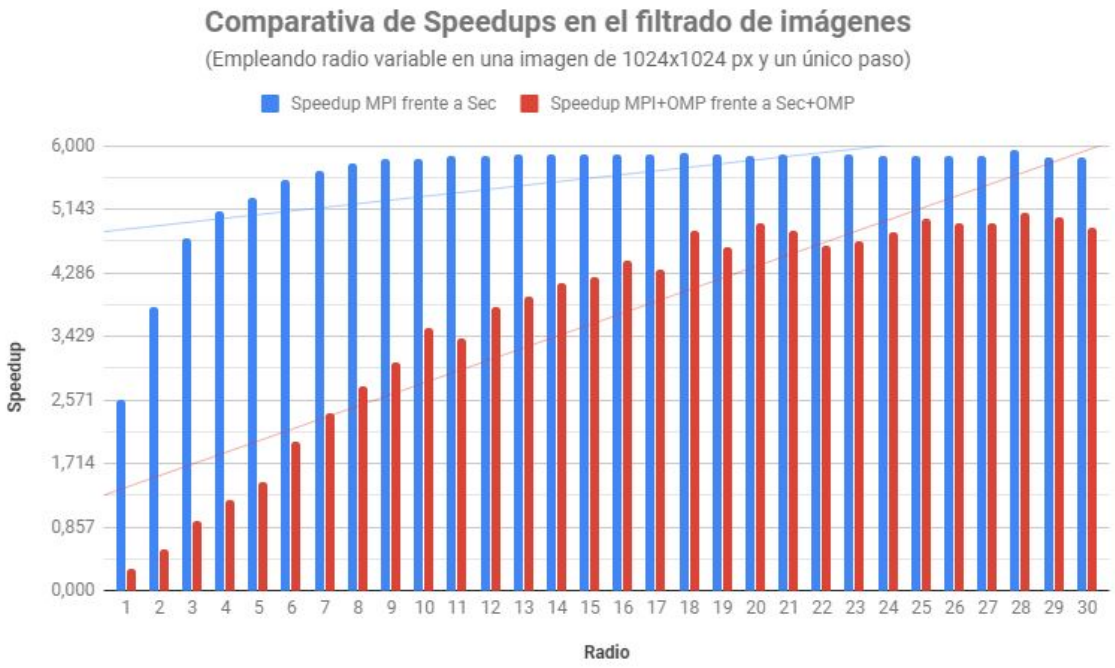
| Radio | Tiempo Sec | Tiempo Sec + OMP | Tiempo MPI | Tiempo MPI + OMP | Speedup MPI frente a Sec | Speedup MPI+OMP frente a Sec+OMP | Speedup Seq+OMP frente a Sec | Speedup MPI+OMP frente a MPI | Speedup MPI+OMP frente a Seq |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0,2231 | 0,0170 | 0,0864 | 0,0578 | 2,582 | 0,293 | 13,152 | 1,495 | 3,859 |
| 2 | 0,5383 | 0,0334 | 0,1407 | 0,0595 | 3,827 | 0,561 | 16,140 | 2,366 | 9,053 |
| 3 | 0,9885 | 0,0593 | 0,2082 | 0,0633 | 4,747 | 0,936 | 16,664 | 3,287 | 15,604 |
| 4 | 1,5871 | 0,0909 | 0,3103 | 0,0740 | 5,114 | 1,229 | 17,452 | 4,193 | 21,441 |
| 5 | 2,3310 | 0,1319 | 0,4402 | 0,0898 | 5,296 | 1,469 | 17,669 | 4,901 | 25,954 |
| 6 | 3,2865 | 0,1860 | 0,5939 | 0,0925 | 5,534 | 2,011 | 17,668 | 6,420 | 35,527 |
| 7 | 4,3393 | 0,2479 | 0,7659 | 0,1036 | 5,666 | 2,393 | 17,504 | 7,394 | 41,890 |
| 8 | 5,5464 | 0,3115 | 0,9624 | 0,1131 | 5,763 | 2,755 | 17,808 | 8,513 | 49,060 |
| 9 | 6,9201 | 0,3847 | 1,1895 | 0,1253 | 5,818 | 3,070 | 17,990 | 9,495 | 55,237 |
| 10 | 8,4088 | 0,4786 | 1,4424 | 0,1349 | 5,830 | 3,547 | 17,570 | 10,690 | 62,323 |
| 11 | 10,0349 | 0,5564 | 1,7124 | 0,1640 | 5,860 | 3,393 | 18,036 | 10,442 | 61,193 |
| 12 | 11,8030 | 0,6697 | 2,0117 | 0,1750 | 5,867 | 3,826 | 17,625 | 11,495 | 67,440 |
| 13 | 13,7089 | 0,7586 | 2,3326 | 0,1909 | 5,877 | 3,974 | 18,072 | 12,219 | 71,814 |
| 14 | 15,7778 | 0,8726 | 2,6847 | 0,2099 | 5,877 | 4,158 | 18,081 | 12,791 | 75,173 |
| 15 | 17,9751 | 1,0049 | 3,0506 | 0,2377 | 5,892 | 4,228 | 17,887 | 12,833 | 75,617 |
| 16 | 20,2900 | 1,1242 | 3,4510 | 0,2521 | 5,880 | 4,459 | 18,049 | 13,687 | 80,471 |
| 17 | 22,7761 | 1,2580 | 3,8688 | 0,2901 | 5,887 | 4,337 | 18,105 | 13,338 | 78,524 |
| 18 | 25,5278 | 1,4414 | 4,3254 | 0,2970 | 5,902 | 4,853 | 17,711 | 14,564 | 85,955 |
| 19 | 28,2835 | 1,5600 | 4,8059 | 0,3370 | 5,885 | 4,630 | 18,130 | 14,262 | 83,935 |
| 20 | 31,0674 | 1,8171 | 5,3022 | 0,3669 | 5,859 | 4,952 | 17,098 | 14,449 | 84,665 |
| 21 | 34,1141 | 1,8874 | 5,8087 | 0,3889 | 5,873 | 4,854 | 18,075 | 14,938 | 87,728 |
| 22 | 37,2795 | 2,0831 | 6,3637 | 0,4484 | 5,858 | 4,645 | 17,896 | 14,191 | 83,135 |
| 23 | 40,6953 | 2,2392 | 6,9185 | 0,4741 | 5,882 | 4,723 | 18,174 | 14,593 | 85,837 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 24 | 44,0112 | 2,4617 | 7,5079 | 0,5094 | 5,862 | 4,832 | 17,878 | 14,738 | 86,391 |
| 25 | 47,5976 | 2,6306 | 8,1282 | 0,5247 | 5,856 | 5,013 | 18,094 | 15,491 | 90,714 |
| 26 | 51,2887 | 2,9030 | 8,7456 | 0,5870 | 5,865 | 4,945 | 17,667 | 14,898 | 87,368 |
| 27 | 55,2406 | 3,0449 | 9,4185 | 0,6156 | 5,865 | 4,946 | 18,142 | 15,300 | 89,735 |
| 28 | 59,9662 | 3,3491 | 10,1006 | 0,6571 | 5,937 | 5,097 | 17,905 | 15,372 | 91,261 |
| 29 | 63,2401 | 3,4934 | 10,8101 | 0,6930 | 5,850 | 5,041 | 18,103 | 15,600 | 91,259 |
| 30 | 67,4211 | 3,7302 | 11,5328 | 0,7620 | 5,846 | 4,895 | 18,074 | 15,135 | 88,480 |




A simple vista queda muy marcada la diferencia en el crecimiento entre las soluciones. Aunque todas tienen un crecimiento exponencial, las que paralelizan (ya sea por MPI, o por OpenMP, o ambos) tienen un crecimiento claramente ralentizado.

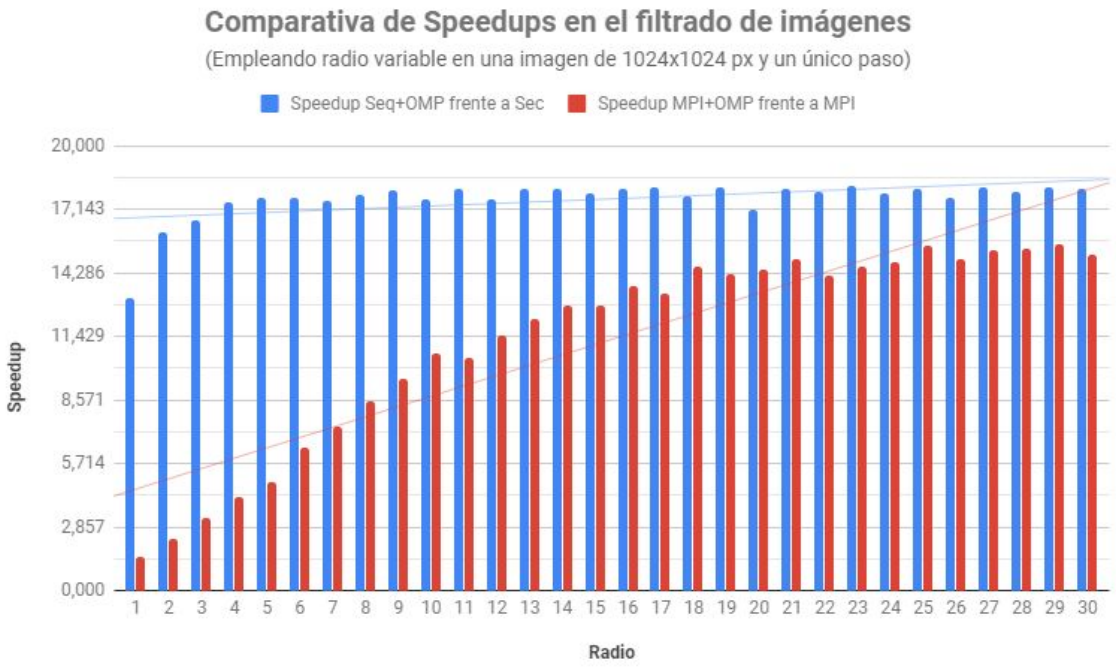
Para imágenes de tamaño medio tampoco tiene mucho sentido aplicar un radio mayor a 30, pues comienza a deformar la imagen y a marcar los colores. Junto a la memoria he adjuntado una imagen GIF donde se puede apreciar la imagen de Lenna y qué ocurre al ampliar el radio progresivamente de 1 a 100.




Speedup MPI frente a Secuencial:
Se aprecia como el Speedup cambia su tendencia con un radio 8 o superior para estabilizarse en un valor cercano a 6 (máximo 5,937), lo cual coincide con el número de nodos de *Kahan* y muestra una aceleración cercana al máximo teórico.

Speedup MPI+OMP frente a Secuencial+OMP:
Entiéndase Secuencial+OMP como una versión paralelizada sólo por hilos, sin MPI.

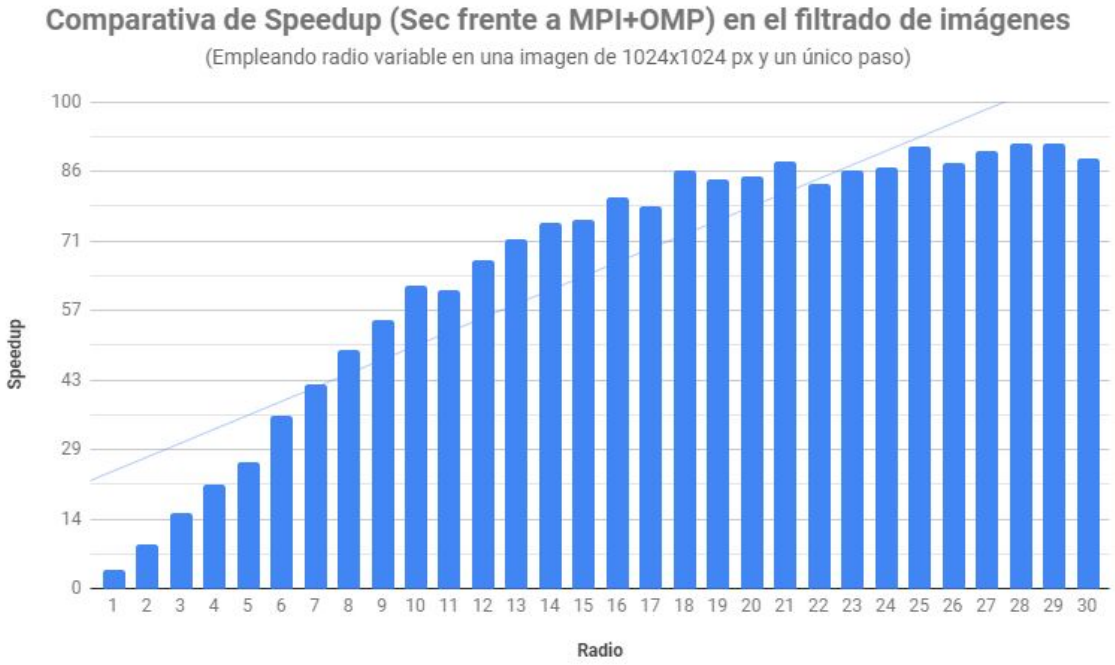
Existe una tendencia similar al punto anterior aunque no tan marcada. Esto es debido a que realmente el speedup es el conjunto de ambos (MPI y OpenMP) y da lugar a una aceleración media inferior, aunque los tiempos de ejecución realmente sean menores.




Este gráfico tiene una forma muy similar al anterior, pero en este caso evalúa el cómo afecta OpenMP a las prestaciones.

Es destacable como el Speedup que obtiene OpenMP es mucho mayor al que obtiene MPI. Esto es debido a que OpenMP carece de el Overhead de las comunicaciones y además cada nodo dispone de 32 hilos, mientras que sólo disponemos de 6 nodos.
Esto fuerza a que el techo teórico del Speedup en MPI es de 6 y el de OpenMP es 32.

También cabe destacar que, mientras en la comparativa anterior se conseguía un Speedup cercano al máximo, en este caso supera ligeramente la mitad (alrededor de 17)



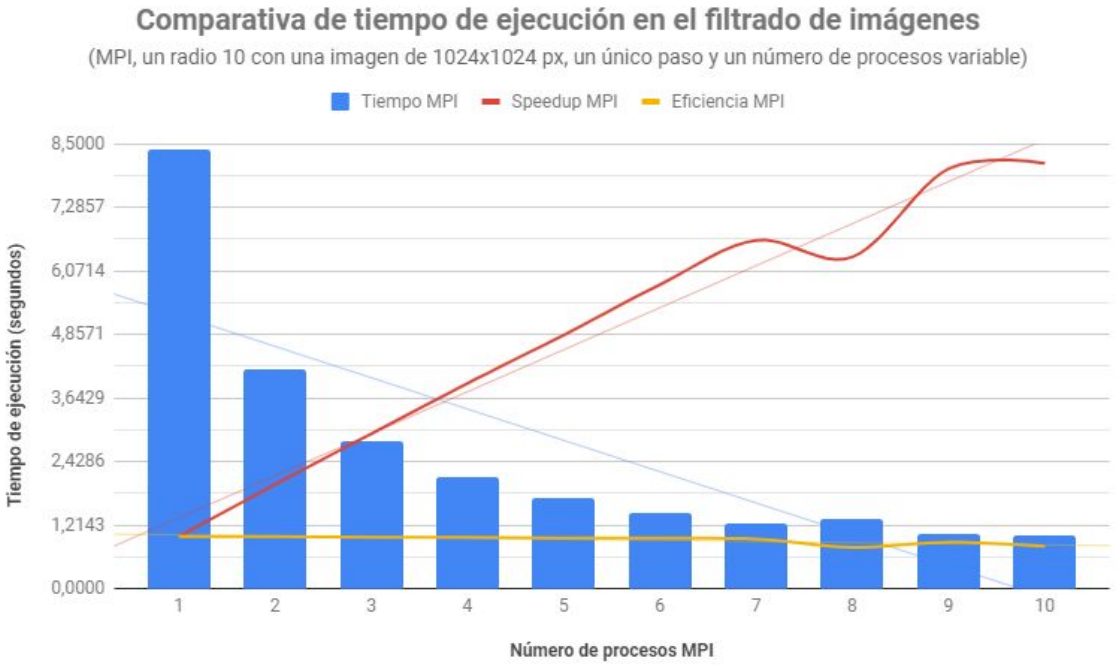

Esta comparativa plasma la aceleración obtenida comparando el tiempo secuencial con el paralelizado con MPI y OpenMP conjuntamente.

Se aprecia un crecimiento hasta un radio alrededor de 21, donde se obtiene el máximo speedup. Dicho Speedup máximo es **91,26 → 9026,12%**.
El speedup máximo teórico serían 192 (6·32), dando lugar a una eficiencia del **47%.**

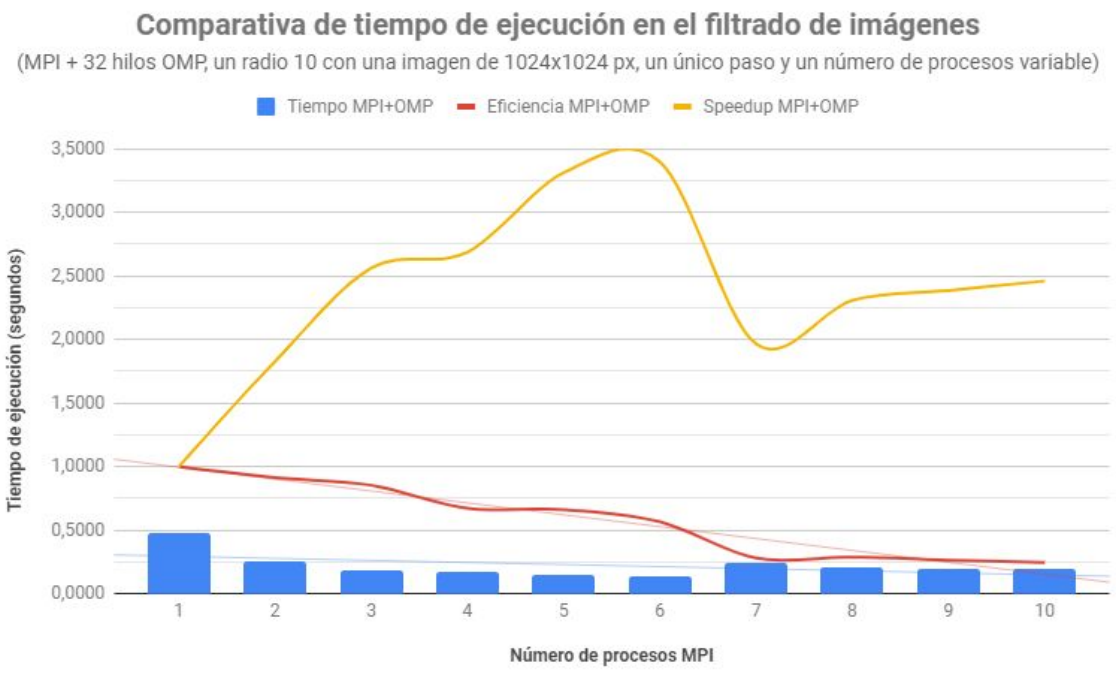
## COMPARATIVA DEL NÚMERO DE PRCESOS MPI

Las mediciones se han realizado con una imagen de 1024x1024px, un radio 10, un único paso y se ha variado el número de procesos MPI en *Kahan*.

| Nº Procs | Tiempo MPI | Tiempo MPI+OMP | Speedup MPI | Speedup MPI+OMP | Eficiencia MPI | Eficiencia MPI+OMP |
|---|---|---|---|---|---|---|
| 1 | 8,3796 | 0,4757 | 1,0000 | 1,0000 | 1,0000 | 1,0000 |
| 2 | 4,2008 | 0,2602 | 1,9948 | 1,8287 | 0,9974 | 0,9144 |
| 3 | 2,8298 | 0,1857 | 2,9613 | 2,5616 | 0,9871 | 0,8539 |
| 4 | 2,1343 | 0,1770 | 3,9262 | 2,6874 | 0,9815 | 0,6718 |
| 5 | 1,7291 | 0,1437 | 4,8463 | 3,3117 | 0,9693 | 0,6623 |
| 6 | 1,4418 | 0,1400 | 5,8119 | 3,3975 | 0,9686 | 0,5663 |
| 7 | 1,2587 | 0,2420 | 6,6574 | 1,9656 | 0,9511 | 0,2808 |
| 8 | 1,3219 | 0,2060 | 6,3392 | 2,3089 | 0,7924 | 0,2886 |
| 9 | 1,0446 | 0,1994 | 8,0219 | 2,3858 | 0,8913 | 0,2651 |
| 10 | 1,0307 | 0,1934 | 8,1302 | 2,4597 | 0,8130 | 0,2460 |




Se observa como el tiempo de ejecución desciende conforme el número de procesos aumenta.
Hay un punto en el que tanto el speedup como la eficiencia se ven afectados. Este punto es a partir de que hay más de 6 procesos en ejecución, ya que *Kahan* sólo dispone de 6 nodos y en caso de tener más procesos, la carga computacional no es homogénea.
En cualquier caso, la eficiencia relativamente se mantiene.




Este caso es similar al anterior pero con un gran matiz.
Al emplear OpenMP se están utilizando todos los núcleos disponibles en cada nodo, de forma que cuando el número de procesos es mayor a 6, los nodos con más de un proceso MPI se ven obligados a realizar cambios de contexto y detener los distintos hilos. Todo esto se aprecia muy claramente en la caída del Speedup y en la caída de la eficiencia. Incluso el tiempo de ejecución con 7 procesos es mayor que con 6.

## ANEXO - imagenesMPI.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>
#include <omp.h>
#include <stddef.h>

#define max(a, b) ((a) > (b) ? (a) : (b))
#define min(a, b) ((a) < (b) ? (a) : (b))
#define MAXCAD 100

#define DEFAULT_NUM_PASOS    1
#define DEFAULT_DIST_RADIO   5

#define ERROR_EXIT_F(A, B) { fprintf(stderr, A, B); return -1; }
#define ERROR_EXIT(A) { fprintf(stderr, A); return -1; }

#define STEP_GATHERV  2
#define STEP_SENDRECV 1

#define INFO
//#define DEBUG
//#define DEBUG_EXTRA
//#define FAKE_ROWS          23
//#define FAKE_COLS   4
```

```c
struct pixel
{
	unsigned char r, g, b;
};

int lee_ppm(char *nomfich, struct pixel ***img, int *nf, int *nc)
{
	FILE *df;
	char tipo[MAXCAD];
	int nivmax, height, width, i, j;

	df = fopen(nomfich, "r");
	if (!df)
		return -1;   /* fopen ha fallado */

	fscanf(df, "%s", tipo);
	if (strcmp(tipo, "P3") != 0)
		return -2;   /* formato erróneo */

	fscanf(df, "%d%d", &width, &height);

	if ((*img = (struct pixel **)malloc(sizeof(struct pixel *) * height)) == NULL)
		return -3;   /* malloc ha fallado */

	if (((*img)[0] = (struct pixel *)malloc(sizeof(struct pixel) * width * height)) == NULL)
		return -3;   /* malloc ha fallado */

	for (i = 1; i < height; i++)
		(*img)[i] = (*img)[i - 1] + width;

	fscanf(df, "%d", &nivmax);

	for (i = 0; i < height; i++)
	{
		for (j = 0; j < width; j++)
		{
		  fscanf(df, "%hhu", &((*img)[i][j].r));
		  fscanf(df, "%hhu", &((*img)[i][j].g));
		  fscanf(df, "%hhu", &((*img)[i][j].b));
		}
	}

	*nf = height;
	*nc = width;

	fclose(df);

	return 0;
}


int escribe_ppm(char *nomfich, struct pixel **img, int nf, int nc)
{
	FILE *df;
	int nivmax = 255, i, j;

	df = fopen(nomfich, "w");
	if (!df)
		return -1;   /* fopen ha fallado */

	fprintf(df, "P3\n");
	fprintf(df, "%d %d\n", nc, nf);
	fprintf(df, "%d\n", nivmax);

	for (i = 0; i < nf; i++)
	{
		for (j = 0; j < nc; j++)
		{
```

```
                fprintf(df, "%d ", img[i][j].r);
                fprintf(df, "%d ", img[i][j].g);
                fprintf(df, "%d ", img[i][j].b);
            }

            fprintf(df, "\n");
        }

        fclose(df);

        return 0;
}

//Esta función aplica una vez el filtrado de la imagen.
//Se ha modificado respecto al código propuesto para el caso de estudio con la intención de extraer todas las comunicaciones y así
//simplificar la paralelización con MPI.
void Filtro(int radio, struct pixel **ppsImagenOrg, struct pixel **ppsImagenDst, int **ppdBloque, int n, int nMax, int nOffset, int m)
{
        int i, j, k, l, tot;
        struct { int r, g, b; } resultado;
        int v;

        for (i = nOffset; i < (n+nOffset); i++)
        {
            //Paralelización OPENMP por tareas
            #pragma omp task firstprivate(i) private(j, k, l, resultado, tot, v) shared(nOffset, n, nMax, m, radio, ppsImagenOrg, ppsImagenDst)
            for (j = 0; j < m; j++)
            {
                resultado.r = 0;
                resultado.g = 0;
                resultado.b = 0;
                tot = 0;

                for (k = max(0, i - radio); k <= min(nMax - 1, i + radio); k++)
                {
                    for (l = max(0, j - radio); l <= min(m - 1, j + radio); l++)
                    {
                        v = ppdBloque[k - i + radio][l - j + radio];
                        #ifdef DEBUG_EXTRA
                            printf("Pos[i=%d][j=%d] -> k=%d, l=%d --> ppdBloque[%d][%d] = %d\n", i, j, k, l, (k - i + radio), (l - j + radio), v);
                        #endif

                        resultado.r += ppsImagenOrg[k][l].r * v;
                        resultado.g += ppsImagenOrg[k][l].g * v;
                        resultado.b += ppsImagenOrg[k][l].b * v;
                        tot += v;
                    }
                }

                resultado.r /= tot;
                resultado.g /= tot;
                resultado.b /= tot;
                ppsImagenDst[i-nOffset][j].r = resultado.r;
                ppsImagenDst[i-nOffset][j].g = resultado.g;
                ppsImagenDst[i-nOffset][j].b = resultado.b;
            }
        }

        #pragma omp taskwait
        return;
}

//Función auxiliar para mostrar un vector.
void printVector(char *name, int *v, unsigned int n)
```

```
{
	unsigned int i = 0;

	printf("%s = [", name);
	for(i = 0; i < n; i++)
	{
		printf("%d", v[i]);
		if(i < (n-1))
			printf(", ");
	}
	printf("]\n");
}

//Esta función reserva y crea la matriz con los pesos para aplicar el filtro.
int createPpdBloque(int ***ppdBloque, int radio)
{
	int i, j;

	if ((*ppdBloque = (int **)malloc(sizeof(int *) * (2*radio + 1))) == NULL)
		return 0;   /* malloc ha fallado */

	if (((*ppdBloque)[0] = (int *)malloc(sizeof(int) * (2*radio + 1) * (2*radio + 1))) == NULL)
		return 0;   /* malloc ha fallado */

	for (i = 1; i < 2*radio + 1; i++)
		(*ppdBloque)[i] = (*ppdBloque)[i - 1] + 2*radio + 1;

	for (i = -radio; i <= radio; i++)
		for (j = -radio; j <= radio; j++)
			(*ppdBloque)[i + radio][j + radio] = (radio - abs(i)) * (radio - abs(i)) + (radio - abs(j)) *
(radio - abs(j)) + 1;

	return 1;
}

//Esta función se encarga de reservar los buffers en memoria con los tamaños adecuados para cada proceso.
int allocateImageBuffers(struct pixel ***ImgOrg, struct pixel ***ImgDst, int rank, int rowsCountOrg, int
rowsCountDst, int colsCount)
{
	int i;

	//IMG SRC
	if(rank > 0)
	{
		if ((*ImgOrg = (struct pixel**)malloc(sizeof(struct pixel *) * rowsCountOrg)) == NULL)
			return 0;   /* malloc ha fallado */

		if (((*ImgOrg)[0] = (struct pixel*)malloc(sizeof(struct pixel) * rowsCountOrg * colsCount)) == NULL)
			return 0;   /* malloc ha fallado */

		for (i = 1; i < rowsCountOrg; i++)
			(*ImgOrg)[i] = (*ImgOrg)[i - 1] + colsCount;

		#ifdef DEBUG
			memset(ImgOrg[0], (sizeof(struct pixel) * rowsCountOrg * colsCount), 0);
		#endif
	}

	//IMG DST
	if ((*ImgDst = (struct pixel**)malloc(sizeof(struct pixel *) * rowsCountDst)) == NULL)
		return 0;   /* malloc ha fallado */

	if (((*ImgDst)[0] = (struct pixel*)malloc(sizeof(struct pixel) * rowsCountDst * colsCount)) == NULL)
		return 0;   /* malloc ha fallado */

	for (i = 1; i < rowsCountDst; i++)
		(*ImgDst)[i] = (*ImgDst)[i - 1] + colsCount;
```

```
	#ifdef DEBUG
		memset(ImgDst[0], (sizeof(struct pixel) * rowsCountDst * colsCount), 0);
	#endif

	return 1; //OK
}

int main(int argc, char *argv[])
{
	struct pixel **ImgOrg, **ImgDst;
	int **ppdBloque;
	unsigned int rowsCount, colsCount;
	int i, j, rc;
	double prevTime, postTime;
	int stepMode = STEP_SENDRECV;

	int *counts, *countsB, *offsets, *offsetsB;
	int radius = DEFAULT_DIST_RADIO;
	int steps = DEFAULT_NUM_PASOS;

	int rank, size;
	MPI_Datatype rowType, pxType;

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);

	int blockLen[] = { 3 };
	MPI_Aint indices[] = { offsetof(struct pixel, r) };
	MPI_Datatype types[] = { MPI_CHAR };
	MPI_Type_create_struct(1, blockLen, indices, types, &pxType);
	MPI_Type_commit(&pxType);

	if(argc < 3)
		ERROR_EXIT_F("Use: %s srcImg destImg [radius] [steps] [sendmode between steps (1=Gatherv,
2=SendRecv)]\n", argv[0]);

	if(argc >= 4)
		sscanf(argv[3], "%d", &radius);

	if(argc >= 5)
		sscanf(argv[4], "%d", &steps);

	if(argc >= 6)
		sscanf(argv[5], "%d", &stepMode);

	if(rank == 0)
	{
		rc = lee_ppm(argv[1], &ImgOrg, &rowsCount, &colsCount);
		if (rc)
			ERROR_EXIT_F("Error al leer el fichero %s\n", argv[1]);

		radius = min(radius, rowsCount);

		#ifdef FAKE_ROWS
			rowsCount = FAKE_ROWS;
			#endif
		#ifdef FAKE_COLS
			colsCount = FAKE_COLS;
		#endif

		#ifdef INFO
			printf("Abierta una imagen de %d filas y %d columnas.\n", rowsCount, colsCount);

			printf("Radio %d y %d pasos.\n", radius, steps);
			printf("MPI: %d procesos\n", size);
			#ifdef _OPENMP
				printf("OPENMP: %d hilos\n", omp_get_max_threads());
```

```
                        #endif
                        if(stepMode == STEP_GATHERV)
                                printf("Usando Gatherv/Scatterv entre pasos\n");
                        else
                                printf("Usando Isend/Irecv entre pasos\n");

                        fflush(stdout);
                #endif

                prevTime = MPI_Wtime();
        }

        //Enviamos a todos los procesos el número de filas y columnas
        MPI_Bcast(&rowsCount, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(&colsCount, 1, MPI_INT, 0, MPI_COMM_WORLD);

        //Creamos la matriz PPD Bloque
        //(Anteriormente se encontraba en la función Filtro, pero debido al cambio de la estructura del programa,
procede sacarlo a aquí para evitar tareas repetidas)
        if(!createPpdBloque(&ppdBloque, radius))
                ERROR_EXIT("Error creating ppdBloque");

        counts  = (int*)malloc(size * sizeof(int));
        countsB  = (int*)malloc(size * sizeof(int));
        offsets  = (int*)malloc(size * sizeof(int));
        offsetsB = (int*)malloc(size * sizeof(int));
        unsigned int tempCounts = (rowsCount / size);
        unsigned int remCounts = (rowsCount % size);

        //Este reparto tiene como objetivo distribuir de la forma más equilibrada posible el número de filas entre
los distintos procesos.
        //Además evita en la medida de lo posible aplicar menos carga al proceso con rango 0, el cual generalmente
se encarga mayormente de las comunicaciones y por ello tiene mayor carga.
        for(i = 0; i < size; i++)
        {
                counts[i] = tempCounts;

                if(remCounts > 0 && i != 0)
                {
                        counts[i]++;
                        remCounts--;
                }

                if(i > 0)
                        offsets[i] = (offsets[i-1] + counts[i-1]);
                else
                        offsets[i] = 0;
        }

        for(i = 0; i < size; i++)
        {
                countsB[i] = counts[i] + min(radius, offsets[i]) + min(radius, rowsCount-offsets[i]);
                offsetsB[i] = max(offsets[i] - radius, 0);

                if(countsB[i] + offsetsB[i] > rowsCount)
                        countsB[i] = (rowsCount - offsetsB[i]);
        }

        #ifdef DEBUG
                if(rank == 0)
                {
                        printVector("counts", counts, size);
                        printVector("countsB", countsB, size);
                        printVector("offsets", offsets, size);
                        printVector("offsetsB", offsetsB, size);

                        #ifdef DEBUG_EXTRA
                                char name[64];
```

```
                                int s = (2*radius + 1);
                                for(i = 0; i < s; i++)
                                {
                                        sprintf(name, "ppdBloque[%d]", i);
                                        printVector(name, ppdBloque[i], s);
                                }
                        #endif
                }
        #endif

        MPI_Type_vector(1, colsCount, 1, pxType, &rowType);
        MPI_Type_commit(&rowType);

        //Reserva de los buffers de memoria para ImgOrg e ImgDst
        if(!allocateImageBuffers(&ImgOrg, &ImgDst, rank, countsB[rank], (rank == 0 ? rowsCount : counts[rank]),
colsCount))
                ERROR_EXIT("Error allocating image buffers\n");

        //Reparto inicial
        MPI_Scatterv(ImgOrg[0], countsB, offsetsB, rowType, ImgOrg[0], countsB[rank], rowType, 0, MPI_COMM_WORLD);

        //Desplazamiento donde comienza la fila a cacular en Filtro
        int rowOffset = (offsets[rank] - offsetsB[rank]);
        for(i = 0; i < steps; i++)
        {
                #pragma omp parallel
                #pragma omp single
                Filtro(radius, ImgOrg, ImgDst, ppdBloque, counts[rank], countsB[rank], rowOffset, colsCount);

                if(i < (steps-1)) //Si hay un paso siguiente
                {
                        if(stepMode == STEP_GATHERV)
                        {
                                //Solución 1 para la comunicación entre pasos: GatherV y ScatterV.
                                //Esta solución es la más sencilla a nivel de código pero añade una sobrecarga en las
comunicaciones entre pasos.
                                //Se envía el resultado parcial de vuelta al proceso con rango 0 para que este lo
vuelva a distribuir entre los procesos como en el reparto inicial.
                                //Esto resulta en un proceso ineficiente, ya que genera una sobrecarga sobre el
proceso 0 y además envía en bloque más filas de las necesarias para continuar
                                //el cómputo entre pasos.

                                MPI_Gatherv(ImgDst[0], counts[rank], rowType, ImgDst[0], counts, offsets, rowType, 0,
MPI_COMM_WORLD);
                                MPI_Scatterv(ImgDst[0], countsB, offsetsB, rowType, ImgOrg[0], countsB[rank], rowType,
0, MPI_COMM_WORLD);
                        }
                        else if(stepMode == STEP_SENDRECV)
                        {
                                //Solución 2 para la comunicación entre pasos: Isend y Irecv
                                //Esta solución trata de resolver los problemas de la solución anterior.
                                //Cada proceso envía a sus procesos adyacentes el número de filas necesario para
continuar con el Filtrado de la imagen.
                                //De esta forma, no sólo se evita sobrecargar al proceso 0, sino que además se reduce
el número de datos a enviar a sólo los necesarios.
                                //Se emplean comunicaciones no bloqueantes para evitar un orden de envío y recepción
entre los procesos (además de posibles interbloqueos).

                                int myStartRow = offsets[rank];
                                int myEndRow = (myStartRow + counts[rank] - 1);
                                int myStartRowB = offsetsB[rank];
                                int myEndRowB = (myStartRowB + countsB[rank] - 1);

                                MPI_Request recvReq[size];
                                MPI_Request sendReq[size];
                                int recvReqIndex = 0;
                                int sendReqIndex = 0;
```

```
                        for(j = 0; j < size; j++)
                        {
                                int pStartRow = offsets[j];
                                int pEndRow = (pStartRow + counts[j] - 1);
                                int pStartRowB = offsetsB[j];
                                int pEndRowB = (pStartRowB + countsB[j] - 1);

                                if(j < rank) //Envío y recepción a los procesos con un rango menor
                                {
                                        int rowsToRecv = min(pEndRow - myStartRowB + 1, counts[j]);
                                        int rowsToSend = min(pEndRowB - myStartRow + 1, counts[rank]);

                                        if(rowsToRecv > 0 || rowsToSend > 0)
                                        {
                                                #ifdef DEBUG
                                                        printf("P%d -> GOING TO RECV %d ROWS IN ImgOrg[%d] FROM
%d PROC (LOWER)\n", rank, rowsToRecv, max(offsets[j] - offsetsB[rank], 0), j);
                                                        printf("P%d -> GOING TO SEND %d ROWS FROM ImgDst[0] TO %d
PROC (LOWER)\n", rank, rowsToSend, j);
                                                #endif

                                                MPI_Isend(ImgDst[0], rowsToSend, rowType, j, 1, MPI_COMM_WORLD,
&sendReq[sendReqIndex++]);
                                                MPI_Irecv(ImgOrg[max(offsets[j] - offsetsB[rank], 0)],
rowsToRecv, rowType, j, 1, MPI_COMM_WORLD, &recvReq[recvReqIndex++]);

                                        }
                                }
                                else if(j > rank) //Envío y recepción a los procesos con un rango mayor
                                {
                                        int rowsToRecv = min(myEndRowB - pStartRow + 1, counts[j]);
                                        int rowsToSend = min(myEndRow - pStartRowB + 1, counts[rank]);

                                        if(rowsToRecv > 0 || rowsToSend > 0)
                                        {
                                                #ifdef DEBUG
                                                        printf("P%d -> GOING TO RECV %d ROWS IN ImgOrg[%d] FROM
%d PROC (HIGHER)\n", rank, rowsToRecv, offsets[j] - offsetsB[rank], j);
                                                        printf("P%d -> GOING TO SEND %d ROWS FROM ImgDst[%d] TO
%d PROC (HIGHER)\n", rank, rowsToSend, counts[rank]-1, j);
                                                #endif

                                                MPI_Isend(ImgDst[counts[rank] - rowsToSend], rowsToSend, rowType,
j, 1, MPI_COMM_WORLD, &sendReq[sendReqIndex++]);
                                                MPI_Irecv(ImgOrg[offsets[j] - offsetsB[rank]], rowsToRecv,
rowType, j, 1, MPI_COMM_WORLD, &recvReq[recvReqIndex++]);
                                        }
                                }
                        }

                        //Copiamos la parte que corresponde a la sección del proceso actual en ImgOrg para que
esté disponible en el próximo paso.
                        //Este paso se realiza aquí para aprovechar el retardo de la copia para que las
comunicaciones se completen
                        memcpy(ImgOrg[rowOffset], ImgDst[0], (counts[rank] * colsCount * sizeof(struct
pixel)));

                        //Esperamos a que todas las comunicaciones de envío y recepción se completen.
                        MPI_Waitall(sendReqIndex, sendReq, MPI_STATUSES_IGNORE);
                        MPI_Waitall(recvReqIndex, recvReq, MPI_STATUSES_IGNORE);
                }
        }
    }

    //Obtenemos los resultados de todos los procesos
    MPI_Gatherv(ImgDst[0], counts[rank], rowType, ImgDst[0], counts, offsets, rowType, 0, MPI_COMM_WORLD);
```

```
	if(rank == 0)
	{
		postTime = MPI_Wtime();
		printf("Tiempo total: %f\n", (postTime - prevTime));

		rc = escribe_ppm(argv[2], ImgDst, rowsCount, colsCount);
		if (rc)
			ERROR_EXIT_F("Error al escribir la imagen en el fichero %s\n", argv[2]);
	}

	free(ppdBloque[0]);
	free(ppdBloque);
	free(ImgOrg[0]);
	free(ImgDst[0]);
	free(ImgOrg);
	free(ImgDst);
	free(counts);
	free(countsB);
	free(offsets);
	free(offsetsB);

	MPI_Type_free(&pxType);
	MPI_Type_free(&rowType);
	MPI_Finalize();
	return 0;
}
```