Laboratorio de Paralelismo

Prácticas MPI

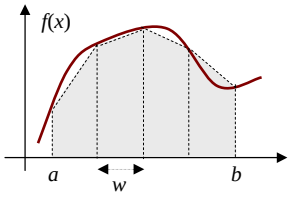
Tras los ejemplos MPI que hemos ido haciendo en el laboratorio, hay que realizar estos cuatro ejercicios. Los programas de partida los tienes en el directorio de trabajo. El primero consiste simplemente en medir los tiempos de ejecución de uno de los programas que hemos trabajado en el laboratorio, para obtener alguna conclusión sobre el tamaño de grano mínimo que deben tener las tareas paralelas que se ejecuten en un *cluster* del tipo que estamos utilizando (PCs + Gigabit Ethernet). En los otros tres ejercicios se trata de escribir un programa MPI que ejecute una determinada tarea y, en su caso, obtener tiempos de ejecución y las correspondientes conclusiones. Los programas incluyen variables, funciones, etc. como ayuda, pero puedes modificarlos en lo que quieras (manteniendo la semántica, claro). No olvides añadir `#include <mpi.h>` al principio de los programas. Recuerda que el compilador `icc` (`mpicc`) tiene por defecto la optimización –O2.

En los casos en los que vayas a tomar medidas del tiempo de ejecución, para evitar problemas de interferencias con otras tareas y obtener tiempos lo más "estables" posibles, añade un bucle al programa que repita la prueba *N* veces, mide los tiempos de ejecución en cada repetición, y quédate con el menor (obtén también la media y la desviación típica de todas las pruebas salvo la primera, tal vez eliminando aquellas que se separen claramente de la media).

Al igual que en el caso OpenMP, hay que redactar un **informe con los resultados** obtenidos en cada ejercicio (numéricos y gráficos, programas…), y explicar de manera razonada el comportamiento que se observa. Recuerda que el trabajo forma parte de la evaluación de la asignatura. **Se valorarán** las soluciones propuestas, los resultados obtenidos y la explicación de los mismos, las diferentes pruebas o experimentos que se propongan, etc., así como la calidad (estructuración, redacción, explicaciones, gráficos…) del informe.
Plazo de entrega de resultados de esta segunda parte: **12 de Mayo**.

Estos son los ejercicios que hay que trabajar.

Ejercicio 1 `integral_s.c / integral_p.c`



Tal como hemos visto en el laboratorio, el programa calcula la integral de una función, en este caso $f(x)$ = 1.0 / (sen x + 2.0) + 1.0 / (sen x * cos x + 2.0), mediante la suma del área de un cierto número de trapecios que cubren la curva de la función entre dos puntos.
El programa solicita los dos extremos de la función y el número de trapecios a sumar. `integral_s.c` es el programa serie e `integral_p.c` la versión paralela MPI (`broadcast` para el reparto de datos y `reduce` para la recogida de resultados).
El objetivo del ejercicio es **medir el tiempo de ejecución de ambos programas**, usando los siguientes parámetros:

- **límites** de la integral: 0.0 y 100.0
- **número de trapecios**: 256 (2^8), 512, 1.024... hasta 134.217.728 (2^{27})
- **número de procesadores** (caso paralelo): 2, 4, 8, 16 y 32

Añade al programa un bucle que efectúe las integrales en función del número de trapecios y otro que efectúe las repeticiones de las que obtener el valor mínimo (elimina la petición de los límites de la integral —añádelos como constantes— y del número de trapecios).

Representa gráficamente los tiempos de ejecución serie y paralelo en función del número de trapecios sumados, así como el factor de aceleración (*speed-up*) en relación al caso serie y la eficiencia obtenida. Utiliza las escalas más adecuadas para los gráficos, de manera que se observe correctamente el comportamiento del programa. Interpreta y justifica los resultados obtenidos.

Ejercicio 2 `anillo.c`

N procesos forman un anillo en el que la comunicación es `i` → `(i+1)` *mod* `num_proc`. Queremos medir cómo se comporta la red de comunicación en función del tamaño de los mensajes que se envían, para lo que vamos a ejecutar un programa que haga circular un vector de diferentes tamaños entre los procesadores del anillo, dando una vuelta al mismo. Al comienzo, se pide la longitud del vector que se va a transmitir.

El objetivo del ejercicio es **completar el programa** y obtener el tiempo de ejecución del mismo en un anillo de 16 procesadores, siendo el tamaño del vector a transmitir de 2^0, 2^1, 2^2, ..., 2^{22} enteros. Se recomienda hacer un bucle para todos los experimentos, empezando con un mensaje de 2^0 elementos e incrementando la longitud del vector que se envía multiplicándola por 2. Igual que en el caso anterior, añade un bucle de repeticiones del experimento para obtener valores mínimos, medios, etc.

A partir de los resultados que obtengas, calcula el **tiempo de transmisión** `i` → `i+1` y el **ancho de banda** conseguido en la transmisión (MB/s), en función de la longitud del vector transmitido. Dibuja ambas curvas de manera adecuada e interpreta los resultados.

Ejercicio 3 `pladin.c`

Se quiere repartir la ejecución de una determinada cola de tareas de manera dinámica entre *n* procesos. Uno de los procesos (*manager*) hace la veces de gestor de la cola de tareas; su trabajo consiste en ir distribuyendo las tareas **a petición de los otros procesos**. El resto de los procesos (*workers*) ejecuta la tarea encomendada, y, al finalizar la misma, solicita una nueva tarea, hasta terminar de esta manera con todas las tareas. Se trata de una estructura estándar de **gestión dinámica de tareas** y, como ejemplo de la misma, vamos a considerar el caso de procesamiento de una determinada matriz. El programa `pladin_ser.c` recoge la versión serie del trabajo que hay que realizar en paralelo.

Escribe el programa `pladin.c`, que procesa en paralelo la matriz `M` repartiendo las filas de la matriz de manera dinámica, tipo *self scheduling*, es decir, fila a fila. Recuerda: uno de los procesos, P0, es el encargado de ir **repartiendo los datos (filas) bajo demanda**, y el resto, *n*–1 procesos, procesan las filas, devuelven resultados y solicitan una nueva tarea. Al final, la matriz `M` debe quedar actualizada en P0.

Confirma los resultados de la versión paralela con los que obtienes en la versión serie. Finalmente, obtén los tiempos de ejecución y el *speed-up* para los casos de 1+1, 1+2, 1+4, 1+8, 1+16 y 1+31 procesos.

Si quieres, prueba con otro tipo de *scheduling* y compara los resultados que obtengas.

Ejercicio 4 `imagen.c`

El procesado de imágenes es una tarea típica que se puede efectuar en paralelo y que ofrece buenos resultados. Una imagen es simplemente una matriz de píxeles, y sobre ella se efectúan diferentes operaciones matemáticas. Como ejemplo de este tipo de tareas, te proponemos paralelizar el programa `imagen.c`, que efectúa sobre una imagen de entrada —en formato `pgm` (*portable grey map*), escala de grises [0 .. 255]— unas cuantas operaciones típicas.

A. En primer lugar, aplica dos veces a los píxeles de la imagen las siguientes operaciones:

1. **Máscara para extraer contornos:**

   ```
   imagen[i][j] = 8*imagen[i][j]
                - imagen[i-1][j-1] - imagen[i-1][j] - imagen[i-1][j+1]
                - imagen[i][j-1]   - imagen[i][j+1]
                - imagen[i+1][j-1] - imagen[i+1][j] - imagen[i+1][j+1]
   ```

   La máscara corresponde a una pequeña matriz de 3×3 elementos con contenido:

   ```
   -1   -1   -1
   -1    8   -1
   -1   -1   -1
   ```

   que se aplica a los elementos de la imagen original. Esta máscara se conoce como máscara de Laplace, y su efecto es resaltar los contornos o siluetas de la imagen.

2. **Proceso de "umbralización":**

   Tras "filtrar" la imagen con la máscara anterior, los píxeles de valor menor que uno dado se convierten al valor mínimo (0, negro), y los de valor mayor que uno dado, al valor máximo (255, blanco).

B. Tras efectuar el proceso anterior dos (`NUM_ITER`) veces, se obtienen dos "**proyecciones**" de la nueva imagen, "proyección vertical" y "proyección horizontal", que contienen sendos vectores que contabilizan, respectivamente, el **número de píxeles de cada columna y de cada fila de la imagen final que son distintos de 0 (negro)**.

Como resultado final del programa se genera una imagen que agrupa a la imagen obtenida tras el proceso de resaltado de los contornos y a las imágenes correspondientes a ambas proyecciones. Si el fichero con la imagen de partida es `pz.pgm`, el resultado final queda en el fichero `pz_imagconproy.pgm`; además obtenemos otros cuatro ficheros `pgm` con las imágenes de salida de cada iteración del bucle de obtención de contornos (`pz_lp0` y `pz_lp1`) y las dos proyecciones de la imagen final (`pz_proy_hori` y `pz_proy_vert`).

El programa serie `imagen.c` contiene unas cuantas rutinas ya programadas que permiten efectuar estas operaciones. El fichero `pixmap.c` (y `pixmap.h`) contiene las rutinas necesarias para poder leer y almacenar imágenes en el formato estándar `pgm`.

El programa `imagen.c` utiliza un `struct` para guardar las imágenes, con la siguiente estructura (ver fichero `pixmap.h`):

```
typedef struct {
  int w;                /* numero de columnas */
  int h;                /* numero de filas */
  unsigned char *dat;   /* imagen almacenada en bytes [0..255] */
  unsigned char **im;   /* vector de punteros a filas - acceso bidimensional */
} imagen;

imagen i1;        /* declaracion de la variable i1 de tipo imagen */
```

Los dos primeros enteros indican el tamaño de la imagen; `dat` almacena todos los bytes de la imagen en un vector de longitud `w*h`; `im` es un vector de punteros que apuntan al comienzo de cada fila de la imagen, para poder referenciar la imagen como `i1.im[i][j]`.

Para compilar tu programa, ejecuta:

```
> icc –o imagen imagen.c pixmap.c        (versión serie)
> mpicc –o imagen imagen.c pixmap.c      (versión paralela MPI)
```

Para visualizar las imágenes se puede utilizar cualquier visor compatible con el formato `pgm`. Por ejemplo, está instalada la aplicación `ImageMagick` que permite ver las imágenes y efectuar transformaciones sencillas de las mismas. Los dos comandos principales son `display` y `convert`. Por ejemplo:

```
> display imagen.pgm
```
Presenta en pantalla la imagen contenida en el fichero `imagen.pgm`.

```
> convert imagen.jpg imagen.pgm
```
Convierte la imagen en formato `jpg` al formato `pgm`.

[`display` o `convert -help` para obtener información sobre el comando]

Sugerencias para la versión paralela

El procesado de la imagen se puede paralelizar de muchas maneras. Por ejemplo, se puede repartir la imagen por filas entre los procesos. En todo caso, probablemente el tamaño de la imagen no será múltiplo del número de procesos, por lo que habrá que repartir trozos de tamaño diferente.

Previo a escribir código paralelo, te recomendamos que, tras entender cómo funciona el proceso en serie, hagas un esquema básico de cómo quieres repartir y procesar los datos, para tener claro el tipo de funciones MPI que debes utilizar.

Finalmente, prueba tu programa con diferente número de procesos y compara los resultados (la imagen final) con los que obtienes del caso serie.

Ejercicio 1: integral_s.c

```
/*****************************************************************************
     integral_s.c
     Integral de una funcion mediante sumas de areas de trapecios
 *****************************************************************************/

#include <stdio.h>
#include <sys/time.h>
#include <math.h>


/*************  FUNCION f: funcion a integrar  **********************/
double f(double x)
{
  double y;

  y = 1.0 / (sin(x) + 2.0) + 1.0 / (sin(x)*cos(x) + 2.0);

  return y;
}


/*************  MAIN  ***********************************************/
int main()
{
  struct timeval  t0, t1;
  double          tej;

  double      a, b, w;          // Limites de la integral
  long long   n, i;             // Numero de trapecios a sumar
  double      resul, x;
  float  aux_a, aux_b, aux_n;


  printf("\n  Introduce a, b (limites) y n (num. de trap.) \n");
  scanf("%f %f %f", &aux_a, &aux_b, &aux_n);

  a = (double) aux_a;
  b = (double) aux_b;
  n = (long long) aux_n;

  gettimeofday(&t0,0);

// CALCULO DE LA INTEGRAL

  w = (b-a) / n;                        // anchura de los trapecios

  resul = (f(a) + f(b)) / 2.0;
  x = a;

  for (i=1; i<n; i++)
  {
    x = x + w;
    resul = resul + f(x);
  }
  resul = resul * w;


  gettimeofday(&t1,0);
  tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec)/1e6;

   printf("\n Integral de f (a=%1.2f, b=%1.2f, n=%1.0f) = %1.12f\n",a,b,(double)n,resul);
  printf("  T. ejec. (serie) = %1.3f ms \n\n", tej*1000);

  return 0;
} /* main */
```

Ejercicio 1: integral_p.c

```
/*****************************************************************************
        MPI: integral_p.c
        Integral de una funcion mediante sumas de areas de trapecios
        Broadcast para el envio de datos y Reduce para la recepcion
*****************************************************************************/

#include <stdio.h>
#include <math.h>
#include <mpi.h>

double  t0, t1;


/*************  FUNCION f: funcion a integrar  ***************/
double f(double x)
{
  double y;

  y = 1.0 / (sin(x) + 2.0) + 1.0 / (sin(x)*cos(x) + 2.0);

  return y;
}


/*************  FUNCION Leer_datos  *************************/
void Leer_datos(double* a_ptr, double* b_ptr, long long* n_ptr, int pid)
{
  float aux_a, aux_b, aux_n;
  int   root;

  if (pid == 0)
  {
    printf("\n  Introduce a, b (limites) y n (num. de trap.)  \n");
    scanf("%f %f %f", &aux_a, &aux_b, &aux_n);

    t0 = MPI_Wtime();
  }

// castings para convertir a y b a double y n a long long (evitar overflow)

  (*a_ptr) = (double)aux_a;
  (*b_ptr) = (double)aux_b;
  (*n_ptr) = (long long)aux_n;

  root = 0;
  MPI_Bcast(a_ptr, 1, MPI_DOUBLE, root, MPI_COMM_WORLD);
  MPI_Bcast(b_ptr, 1, MPI_DOUBLE, root, MPI_COMM_WORLD);
  MPI_Bcast(n_ptr, 1, MPI_LONG_LONG, root, MPI_COMM_WORLD);

} /* Leer_datos */


/*************  FUNCION Integrar: calculo local   *********************/
double Integrar(double a_loc, double b_loc, long long n_loc, double w)
{
  double resul_loc, x;
  long long    i;              //  n_loc es un long de 8 bytes

  resul_loc = (f(a_loc) + f(b_loc)) / 2.0;
  x = a_loc;

  for (i=1; i<n_loc; i++)
  {
    x = x + w;
    resul_loc = resul_loc + f(x);
  }
  resul_loc = resul_loc * w;

  return (resul_loc);
} /*  Integrar  */
```

```
/*************  MAIN    *******************************************/
int main(int argc, char** argv)
{
  int          pid, npr, root;            // Identificador y numero de proc, y nodo
raiz.
  double      a, b, w, a_loc, b_loc;
  long long   n, n_loc;
  double      resul, resul_loc;         // Resultado de la integral


  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);


// Lectura y distribucion de datos a todos los procesadores

  Leer_datos(&a, &b, &n, pid);
  w = (b-a) / n;

  // por si n no es divisible entre npr; ojo overflow
  n_loc = (pid+1)*n/npr - pid*n/npr;
  a_loc = a + (pid)*n/npr * w;
  b_loc = a + (pid+1)*n/npr * w;

// Calculo de la integral local

  resul_loc = Integrar(a_loc, b_loc, n_loc, w);


// Suma de resultados parciales

  resul = 0.0; root = 0;
  MPI_Reduce(&resul_loc, &resul, 1, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);


// Impresion de resultados

  if (pid == 0)
  {
    t1 = MPI_Wtime();

     printf("\n  Integral de f (a=%1.2f, b=%1.2f, n=%f) = %1.12f", a,b,(double)n,resul);
    printf("\n  T. ejec. (%d proc.) = %1.3f ms \n\n",npr,(t1-t0)*1000);
  }

  MPI_Finalize();
  return 0;
} /*  main  */
```

Ejercicio 2: anillo.c

```
/*****************************************************************************
   MPI: anillo.c
   Se envia un vector de proceso a proceso en un anillo de npr procesos
   Calcula el tiempo de transmision y el ancho de banda.
   Pide la longitud maxima del vector
    -- por completar
   Repite el experimento variando la longitud del vector a enviar
   desde 1 hasta 2**lgmax
   En cada caso, repite el proceso NREP veces y obten el tiempo minimo
*****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

#define NREP 10                 // numero de repeticiones del experimento //

int main(int argc, char *argv[])
{
  int     pid, npr;
  int     siguiente, anterior;
  int     *vector, longi, lgmax, tam;
  int     i;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);


// direcciones de los vecinos en el anillo //
  siguiente = (pid + 1) % npr;
  anterior = (pid + npr - 1) % npr;

// lectura de parametros //
  if (pid == 0)
  {
    printf("\nLongitud maxima del vector a enviar (potencia de 2, maximo 22):   ");
    scanf("%d", &lgmax);
  }

// DISTRIBUCION DE DATOS A TODOS LOS PROCESOS //


/* Reserva de memoria, inicializacion del vector */
  tam = pow(2,lgmax);
  vector = (int *)malloc(sizeof(int)*tam);
  if (pid == 0) for(i=0; i<tam; i++) vector[i] = i % 100;


// COMIENZO DE LA TRANSMISION  //
// desde longi = 1 hasta el valor maximo solicitado (en potencias de 2)

// repetir el proceso NREP veces (para quedarse con el menor tiempo de transmision)

// transmitir vector de tamaino longi del 0 al 1, del 1 al 2, etc., dando 1 vuelta al anillo

// Finalmente, obtener como resultado el tiempo de una vuelta en el anillo,
// el tiempo de transmision i --> i+1
// y el ancho de banda de la transmision i-->i+1 = num_bits / tiempo  (en Mb/s)

  MPI_Finalize();
  free(vector)
  return 0;
}
```

Ejercicio 3: pladin_ser.c

```
/******************************************************************************
    pladin_ser.c
    version serie de un sistema de gestion de una cola de tareas
    (filas de una matriz) mediante planificacion dinamica
******************************************************************************/

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

#define NF 3000
#define NC 500

int main(int argc, char **argv)
{
  struct timeval t0,t1;
  double tej;

  int   i, j, k, x;
  int   M[NF][NC], sum;


// inicializacion de la matriz
  for(i=0; i<NF; i++)
  for(j=0; j<NC; j++)
     M[i][j] = rand() % 1000 - 200;


  gettimeofday(&t0,0);

// Procesamiento de la matriz. CODIGO A PARALELIZAR
  for (i=0; i<NF; i++)
  for (j=0; j<NC; j++)
  {
    if (M[i][j] > 0)
    {
      x = 1;
      for (k=2; k <= M[i][j]; k++) x = (x * k) % (M[i][j]+1);
      M[i][j] = x % (M[i][j]+1);
    }
  }

  sum = 0;
  for (i=0; i<NF; i++)
  for (j=0; j<NC; j++)
    sum = (sum + M[i][j]) % 1000;


  gettimeofday(&t1,0);

// Imprimir resultados
  printf("\n Suma (mod 1000) de la matriz transformada = %d\n", sum);

  tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec)/1e6;
  printf("\n T. ejec. (serie) = %1.3f s\n\n", tej);

  return 0;
}
```

Ejercicio 4: imagen.c

```
/*****************************************************************************
   Fichero: imagen.c

   Se aplica a una imagen en formato PGM (P5: binario) el siguiente tratamiento
   para resaltar los contornos de la imagen:
     -- una mascara de laplace de 3 x 3
     -- unos umbrales maximo y minimo
   El proceso se repite NUM_ITER (2) veces.

   De la imagen final se calcula un histograma por filas y otro por columnas,
   para contar el numero de pixeles distintos de 0 (negro).

   Ficheros de entrada:   xx.pgm (imagen en formato pgm)
   Ficheros de salida:    xx_imagconproy.pgm    imagen final e histogramas
                          xx_lp1 / xx_lp2 ...   imagenes parciales de cada iteracion
                          xx_proy_hori.pgm      histograma por filas
                          xx_proy_vert.pgm      histograma por columnas

   Compilar el programa junto con pixmap.c

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



#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "pixmap.h"

#define NEGRO   0
#define BLANCO  255
#define MAX_VAL 256     /* Rango de posibles valores de grises */
#define NUM_ITER 2      /* Numero de repeticiones del proceso de la imagen */
#define TAM_PROY 150    /* Los histogramas son de 150 * h/w pixeles */

struct timeval t0,t1;
double tej;


/*                FUNCIONES DE GESTION DE IMAGENES                          */

/* 1: Copia la imagen iimagen en oimagen.   Reserva memoria                 */
/*****************************************************************************/

void copiar_imagen (imagen *oimagen, imagen iimagen)
{
  int i, h, w;

  h = iimagen.h;
  w = iimagen.w;

  if ((oimagen->dat = (unsigned char *) malloc(w*h * sizeof(unsigned char))) == NULL)
  {
    fprintf(stderr, "Funcion copiar_imagen: malloc error\n");
    exit(1);
  }

  oimagen->h = h;
  oimagen->w = w;
  oimagen->im = (unsigned char **) malloc(h * sizeof(unsigned char *));

  for (i=0; i<h; i++) oimagen->im[i] = &oimagen->dat[i*w];
  memcpy(oimagen->dat, iimagen.dat, h*w);

  return;
}
```

```
/*  2: Crea una imagen y la inicializa a un valor constante                    */
/*****************************************************************************/

void  generar_imagen(imagen *oimagen, int h, int w, unsigned char val)
{
  int i, j;

  oimagen->h = h;
  oimagen->w = w;

  // asignar memoria a la imagen

  if ((oimagen->dat = (unsigned char *) malloc(w*h * sizeof(unsigned char))) == NULL)
  {
    fprintf(stderr, "Funcion generar_imagen: malloc error\n");
    exit(1);
  }

  // crear la estructura de punteros a las filas

  if ((oimagen->im = (unsigned char **) malloc(h * sizeof(unsigned char *))) == NULL)
  {
    fprintf(stderr, "Funcion generar_imagen: malloc error\n");
    exit(1);
  }
  for(i=0; i<h; i++) oimagen->im[i] = &oimagen->dat[i*w];

  // inicializar los pixeles de la imagen

  memset(oimagen->dat, val, h*w);

  return;
}


/* 3: Libera memoria de una imagen                                            */
/*****************************************************************************/

void  liberar_imagen(imagen iimagen)
{
  free(iimagen.dat);
  free(iimagen.im);

  return;
}



/*                  FUNCIONES DE TRATAMIENTO DE IMAGENES                     */

/*4: Calcula un vector con la proyeccion vertical (reserva memoria)          */
/*****************************************************************************/

void  calcular_proyeccion_vertical(int **proy, imagen iimagen)
{
  int i, j, h, w;


  h = iimagen.h;
  w = iimagen.w;

  // Reservar memoria para el vector proyeccion
  if ( (*proy = (int *) malloc(w * sizeof(int))) == NULL )
  {
    fprintf(stderr, "Funcion calcular_proyeccion_vertical: malloc error\n");
    exit(1);
  }

  for (j=0; j<w; j++) (*proy)[j] = 0;

  for (i=0; i<h; i++)
   for (j=0; j<w; j++)
    if (iimagen.im[i][j] != NEGRO) (*proy)[j]++;

  return;
}
```

```
/* 5: Crea una imagen con la proyeccion vertical (reserva memoria)                */
/*********************************************************************************/

void  crear_imagen_proyeccion_vertical(imagen *proyimagen,int *proy, int longi)
{
  int maximo, valor;
  int i, j;

  // Crea una imagen para la proyeccion vertical, toda en negro
  generar_imagen(proyimagen, TAM_PROY, longi, NEGRO);

  // calcula el valor maximo para escalar la imagen del histo a 150 pixeles
  maximo = -1;
  for (i=0; i<longi; i++) if(proy[i] > maximo) maximo = proy[i];

  for (j=0; j<longi; j++)
  {
    if (maximo == 0) valor = 0;                          // por si max fuera 0
    else valor = proy[j] * TAM_PROY/maximo;              // escalar al valor maximo

   // deja una linea en negro, de tamaino proporcional al valor correspondiente
    for (i=0; i < TAM_PROY-valor; i++) proyimagen->im[i][j] = BLANCO;
  }

  return;
}

/* 6: Calcula un vector con la proyeccion horizontal (reserva memoria)            */
/*********************************************************************************/

void  calcular_proyeccion_horizontal(int **proy, imagen iimagen)
{
  int i, j, h, w;

  h = iimagen.h;
  w = iimagen.w;

  // Reservar memoria para el vector proyeccion
  if ( (*proy = (int *) malloc(h * sizeof(int))) == NULL )
  {
    fprintf(stderr, "Funcion calcular_proyeccion_horizontal: malloc error\n");
    exit(1);
  }
  for (i=0; i<h; i++) (*proy)[i] = 0;

  for (i=0; i<h; i++)
   for (j=0; j<w; j++)
    if (iimagen.im[i][j] != NEGRO) (*proy)[i]++;

  return;
}

/* 7: Crea una imagen con la proyec. horizontal (reserva memoria)                 */
/*********************************************************************************/

void  crear_imagen_proyeccion_horizontal(imagen *proyimagen,int *proy, int longi)
{
  int maximo, valor;
  int i, j;

  // Crea una imagen para la proyeccion vertical, toda en negro
  generar_imagen(proyimagen, longi, TAM_PROY, NEGRO);

  // calcula el valor maximo para escalar la imagen del histo a 150 pixeles
  maximo = -1;
  for (i=0; i<longi; i++) if(proy[i] > maximo) maximo = proy[i];

  for (i=0; i<longi; i++)
  {
    if (maximo == 0) valor = 0;                          // por si max fuera 0
    else valor = proy[i]* TAM_PROY/maximo;               // escalar al valor maximo

    // deja una linea en negro, de tamaino proporcional al valor correspondiente
    for (j=0; j < TAM_PROY-valor; j++) proyimagen->im[i][j] = BLANCO;
  }

  return;
}
```

```
/* 8: Crea una unica imagen con iimagen y sus dos proyecciones                    */
/*********************************************************************************/

void  crear_imagen_con_proyecciones(imagen *oimagen, imagen iimagen, imagen proyvert,
                                    imagen proyhori)
{
  int i, j, w, h;

  h = oimagen->h = iimagen.h + proyvert.h;
  w = oimagen->w = iimagen.w + proyhori.w;

  // Crea la imagen total, inicializada a negro
  generar_imagen(oimagen, h, w, NEGRO);


  // Copia iimagen
  for (i=0; i<iimagen.h; i++)
   for (j=0; j<iimagen.w; j++)
    oimagen->im[i][j] = iimagen.im[i][j];

  // copia a la derecha la proyeccion horizontal
  for (i=0; i<proyhori.h; i++)
   for (j=0; j<proyhori.w; j++)
    oimagen->im[i][iimagen.w + j] = proyhori.im[i][j];

  // copia debajo la proyeccion vertical
  for (i=0; i<proyvert.h; i++)
   for (j=0; j<proyvert.w; j++)
    oimagen->im[i + iimagen.h][j] = proyvert.im[i][j];

  //printf ("\nCreada imagen con proyecciones vertical y horizontal\n");
  return;
}


/* 9: Filtro de laplace (3x3): detecta ejes (iimagen -> oimagen)                  */
/*********************************************************************************/

void aplicar_laplace_contorno(imagen *oimagen, imagen iimagen)
{
  int suma;
  int i, j, k, l, h, w;
  int mask_laplace[3][3];

  // Mascara de laplace
  mask_laplace[0][0] = -1; mask_laplace[0][1] = -1; mask_laplace[0][2] = -1;
  mask_laplace[1][0] = -1; mask_laplace[1][1] =  8; mask_laplace[1][2] = -1;
  mask_laplace[2][0] = -1; mask_laplace[2][1] = -1; mask_laplace[2][2] = -1;

  h = iimagen.h;
  w = iimagen.w;

  generar_imagen(oimagen, h, w, NEGRO);

  // Aplicar mascara y dejar resultado en oimagen
  for (i=0; i<=h-1; i++)
   for (j=0; j<=w-1; j++)
   {
     if (i==0 || i==h-1) suma = 0;   // los bordes de la imagen se dejan en negro (0)
     else if(j==0 || j==w-1) suma = 0;
     else
     {
       suma = 0;
       for (k=-1; k<=1; k++)
        for (l=-1; l<=1; l++)
          suma = suma + ((int)iimagen.im[i+k][j+l] * mask_laplace[k+1][l+1]);
     }
     if (suma<0) suma = 0;
     if (suma>255) suma = 255;
     (oimagen->im)[i][j] = (unsigned char)suma;
   }

  //printf ("\nAplicado laplace_contorno\t\t(w: %d h: %d)\n",w,h);
  return;
}
```

```
/* 10: Aplica umbrales minimo y maximo a la imagen                              */
/*******************************************************************************/

void aplicar_umbrales(imagen *oimagen)
{
  int i, j, h, w;

  // Valores umbral: por debajo o por encima, se convierten al minimo o al maximo
  unsigned char umbral_min =  40;
  unsigned char valor_min  =   0;
  unsigned char umbral_max = 215;
  unsigned char valor_max  = 255;

  h = oimagen->h;
  w = oimagen->w;

  for (i=0; i<=h-1; i++)
   for (j=0; j<=w-1; j++)
   {
     if (oimagen->im[i][j] <= umbral_min)      oimagen->im[i][j] = valor_min;
     else if (oimagen->im[i][j] >= umbral_max) oimagen->im[i][j] = valor_max;
   }

  return;
}



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

int main(int argc, char **argv)
{
  int i, sum=0;

  char *name;
  name = malloc (sizeof(char)*100);

  // Imagenes de salida de cada iteracion; 0 = entrada
  imagen lpimagen[NUM_ITER+1];

  // Vectores de proyecciones e imagenes correspondientes
  int *proy_vert, *proy_hori;
  imagen proyimagevert, proyimagehori, imagenconproyecciones;

  if (argc != 2)
  {
    printf ("\nUSO: programa imagen\n");
    printf ("     [extension .pgm implicita]\n");
    exit (0);
  }

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

  printf("\n --> Procesando imagen de hxw = %dx%d pix.\n", lpimagen[0].h, lpimagen[0].w);

  gettimeofday(&t0, 0);

  /* Proceso imagen: NUM_ITER veces (laplace + umbral)                          */
  /*******************************************************************************/

  for (i=1; i<=NUM_ITER; i++)
  {
    aplicar_laplace_contorno(&lpimagen[i], lpimagen[i-1]);

    aplicar_umbrales(&lpimagen[i]);
  }
```

```
  /* Calculo de las proyecciones de la imagen final                              */
  /*******************************************************************************/

  calcular_proyeccion_vertical(&proy_vert, lpimagen[NUM_ITER]);
  calcular_proyeccion_horizontal(&proy_hori, lpimagen[NUM_ITER]);

  /* test de prueba y toma de de tiempos                                         */
  /*******************************************************************************/

  gettimeofday(&t1, 0);

  // test de prueba: sumar el valor de todos los pixeles de la imagen final, modulo 255

  for (i=0; i<lpimagen[NUM_ITER].h * lpimagen[NUM_ITER].w; i++)
    sum = (sum + lpimagen[NUM_ITER].dat[i]) % 255;
  printf("\n     Test de prueba  sum = %d \n", sum);

  tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
  printf("\n     Tej. serie = %1.1f ms\n\n", tej*1000);


  /* Escritura de resultados en disco; liberar memoria                           */
  /*******************************************************************************/

  // Guardar imagenes de salida lpimagen[i], proyecciones e imagen final conjunta

  for (i=1; i<=NUM_ITER; i++)
  {
    sprintf(name, "%s_lp%d.pgm", argv[1], i);
    store_pixmap(name, lpimagen[i]);
  }

  strcpy(name, argv[1]);
  name = strcat(name, "_proy_vert.pgm");
  crear_imagen_proyeccion_vertical(&proyimagevert, proy_vert, lpimagen[NUM_ITER].w);
  store_pixmap(name, proyimagevert);

  strcpy(name, argv[1]);
  name = strcat(name, "_proy_hori.pgm");
  crear_imagen_proyeccion_horizontal(&proyimagehori, proy_hori, lpimagen[NUM_ITER].h);
  store_pixmap(name, proyimagehori);

  strcpy(name, argv[1]);
  name=strcat(name, "_imagconproy.pgm");
  crear_imagen_con_proyecciones(&imagenconproyecciones, lpimagen[NUM_ITER],
                                proyimagevert, proyimagehori);
  store_pixmap(name, imagenconproyecciones);


  // Liberar memoria de las imagenes y proyecciones

  for (i=0; i<=NUM_ITER; i++) liberar_imagen(lpimagen[i]);
  liberar_imagen(imagenconproyecciones);
  liberar_imagen(proyimagevert);
  liberar_imagen(proyimagehori);

  free(proy_hori);
  free(proy_vert);
  free(name);

  return 0;
}
```