# DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS

**Grado en Ingeniería Informática**


CURSO 2022/2023

## Prácticas

**Curso 2022/2023**

Dpnt. de Ciència de la Computació i Intel·ligència Artificial
Dpto. de Ciencia de la Computación e Inteligencia Artificial

Universitat d'Alacant
Universidad de Alicante

Grupo de Computación de Altas Prestaciones y Paralelismo


# DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS
## PRÁCTICAS y PONDERACIÓN

Las prácticas a desarrollar junto con su ponderación y fechas de entrega son:

| **Práctica** | **Peso** | **Observaciones** | **Fechas** |
|---|---|---|---|
| ANILLO | 20 % | – | 13 de marzo |
| PRIMOS TCOM | 20 % | Elegir una | 3 de abril |
| ARISTAS | 30 % | – | 8 de mayo |
| MMmalla | 30 % | – | 22 de mayo |
| FW | – | Optativa | 22 de mayo |

Las prácticas se entregan tal y como se detalla en la explicación de las mismas.


**DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS**

# PROCEDIMIENTO

El procedimiento que se sigue para que el alumnado pueda trabajar en el laboratorio es el siguiente:

1. Apertura de cuenta:

   Todo el alumnado debe disponer de una cuenta propia que servirá para todas las prácticas que ha de realizar. El nombre de la cuenta se corresponde con el usuario EPS y el mismo password.

2. Publicación de información referente a la asignatura:

   Procedimientos habituales (anuncios en UACloud, Moodle, Telegram, ...).

3. Directorios de prácticas:

   Todo el alumnado dispone de un directorio de prácticas que corresponde con su HOME. En concreto para esta asignatura será:

   /home/CUENTA

   donde CUENTA es el nombre de la cuenta personal.

   Dentro del directorio de prácticas se deberá entregar cada práctica en un subdirectorio concreto que contendrá lo establecido en el enunciado de prácticas y todo aquello que sea imprescindible para su funcionamiento. Esto permitirá que el alumnado cree otros directorios para almacenar ficheros de pruebas, etc, que no son de interés a la hora de la corrección.

   Todas las cuentas del alumnado de esta asignatura están almacenadas en un servidor virtual. Estas cuentas se montan en los clientes cuando se accede al laboratorio. Aunque la información se mantiene de una sesión a otra, es conveniente, como siempre, realizar copias de seguridad.

4. Entrega de prácticas:

   La entrega de la práctica se formalizará creando en el directorio de la misma un fichero llamado ENTREGA (en mayúsculas). Este fichero contendrá una línea que identifica a la persona que haya realizado la práctica; el formato ha de ser:

   APELLIDO1 APELLIDO2, NOMBRE

   El alumnado que no vaya a entregar la práctica no debe crear el fichero ENTREGA.

5. Calificación de las prácticas:

   Cada práctica se calificará en función de la implementación realizada, rendimiento y, en su caso, memoria entregada. La escala de calificación será de 0 a 10 con la ponderación previamente estableciada para cada una de ellas. Toda práctica que no compile o no se ajuste a la implementación solicitada será calificada con un 0.


# DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS

## PRÁCTICA 1: ANILLO

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/ANILLO (CUENTA=cuenta personal).

Los ficheros o directorios que sea necesario crear pero no vayan a ser entregados al profesor deberán crearse directamente en el directorio /home/CUENTA y no dentro del subdirectorio de prácticas ANILLO.

En esta práctica se desea que se realice una primera implementación paralela sobre MPI. El objetivo es familiarizarse con la estructura de un programa MPI utilizando las funciones básicas de inicialización y finalización (`MPI_Init, MPI_Comm_rank, MPI_Comm_size` y `MPI_Finalize`) y las de envío y recepción (`MPI_Send` y `MPI_Recv`).

El proceso número 0 (proceso root) debe aceptar un input estándar correspondiente a un número entero. Después de sumarle una unidad, este entero se enviará al proceso número 1, el cual le sumará una unidad y se lo enviará al proceso número 2, y así sucesivamente. El último proceso enviará el entero al proceso root. Así, si el valor del entero era inicialmente 8 y existen 5 procesos en total, el proceso root recibirá al final el número $8 + 5 = 13$. Una vez finalizado todo el proceso de comunicación, el proceso root escribirá en el output estándar el valor del entero.

Ficheros a entregar:

**anillo.c** Contendrá la unidad principal.

**makefile** Makefile utilizado para la compilación.


**DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS**

# PRÁCTICA 2: PRIMOS

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/PRIMOS (CUENTA=cuenta personal).

Los ficheros o directorios que sea necesario crear pero no vayan a ser entregados al profesor deberán crearse directamente en el directorio /home/CUENTA y no dentro del subdirectorio de prácticas PRIMOS.

En esta práctica se desea que se realice la implementación paralela, sobre MPI, para obtener el número de enteros primos menores que uno dado $n$.

Se dice que un entero positivo $p$ es primo si tiene exactamente dos divisores positivos distintos. Por ejemplo:

- Los divisores positivos de 13 son 1 y 13. Luego 13 es primo.
- El entero 15 no solo tiene como divisores positivos el 1 y 15. Tiene otros divisores, el 3 y el 5. Luego 15 no es primo.
- El entero 1 solo tiene un divisor positivo: el propio 1. Luego 1 no es primo.

Los algoritmos existentes para determinar si un entero es primo tienen el suficientemente coste computacional como para poder incluir paralelismo. Si a esto le añadimos que este cálculo se debe realizar muchas veces, el coste computacional se incrementa.

Un algoritmo básico para obtener todos los primos menores que `n` consiste en comprobar todos los enteros entre 2 y `n`, es decir,

**Algoritmo número de primos**

```
1        total_primos = 0;
2        for (int i = 2; i <= n; i++)
3           if (esPrimo(i) == 1)
4              total_primos++;
```

Donde `esPrimo(m)` representa un algoritmo básico para determinar si un entero `m`, mayor que 1, es primo. La función `esPrimo(m)` busca posibles divisores del entero `m` entre 2 y $\sqrt{m}$:

```
int esPrimo(int m) {
   for (int i = 2; i <= sqrt(m); i++)
      if (m%i == 0)
         return 0;
   return 1;
}
```

Si disponemos de un total de `nproc` procesos trabajando en paralelo, el **Algoritmo número de primos** anterior puede ser fácilmente paralelizado teniendo en cuenta que los cálculos representados por el bucle (2) son independientes. En consecuencia, lo único que debemos plantearnos es una estrategia para distribuir la carga de trabajo entre los procesos.

La distribución de los cálculos representados por el bucle (2) se realizará teniendo en cuenta una distribución cíclica. Por ejemplo, si `n=100` y `p=4`:

- El proceso 0 evalúa los enteros $2, 6, 10, \ldots$
- El proceso 1 evalúa los enteros $3, 7, 11, \ldots$
- El proceso 2 evalúa los enteros $4, 8, 12, \ldots$
- El proceso 3 evalúa los enteros $5, 9, 13, \ldots$

De esta forma, el código que desarrollaría un proceso, de un total de `nprocs`, determinado por su rango `myrank` ($0 \leq$ `myrank` $<$ `nprocs`), sería:

```
total_primos = 0;
for (int i = 2 + myrank; i <= n; i = i + nprocs)
   if (esPrimo(i) == 1)
      total_primos++;
```

La implementación en paralelo debe seguir un esquema similar a la secuencial. En concreto, dados tres enteros:

**n_min** Valor mínimo de `n` (=500),

**n_max** Valor máximo de `n` (=5000000) y

**n_factor** Factor de incremento para `n` (=10).

Se desea obtener el número de enteros primos para distintos valores de $n$. En concreto, los valores de `n` estarán comprendidos entre `n_min` y `n_max`, empezando por `n=n_min`, continuando por `n = n·n_factor`, sin sobrepasar `n_max`. Por ejemplo, para los valores indicados, se calculará el número de enteros primos para los valores de `n` = 500, 5000, 50000, 500000, 5000000 (ver versión secuencial).

Al igual que en el caso secuencial, el cálculo del número de enteros primos menores que `n` debe efectuarse, por cada proceso, dentro de una función que llamaremos `numero_primos()`. De esta forma, cada proceso calculará un número de primos local que deberá ser sumado en el proceso `root` mediante la función `MPI_Reduce()`. El proceso `root` calculará el tiempo total necesario incluyendo la recepción de los resultados parciales.

Cada proceso (incluyendo el `root`) calculará, además, el tiempo que ha necesitado para obtener el número de enteros primos locales. Al finalizar, estos tiempos se remitirán al proceso `root`. El envío y recepción de estos tiempos parciales no debe contabilizarse en el cálculo del tiempo total que realiza el proceso `root`.

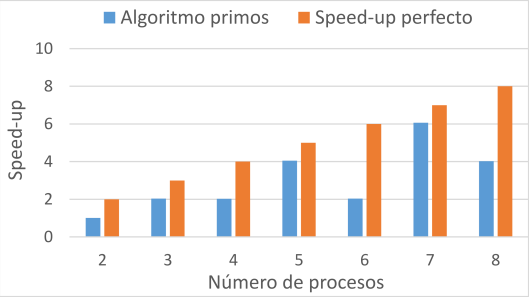
Adicionalmente, antes de iniciar el paralelismo, el proceso `root` calculará de forma secuencial el número de enteros primos menores que `n`, midiendo el tiempo secuencial, para poder calcular, con posterioridad, el speed-up y la eficiencia.

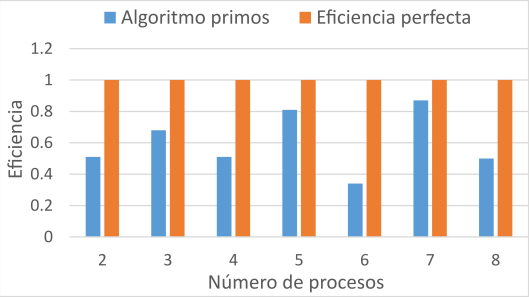
Al finalizar, el proceso `root` debe escribir por pantalla los siguientes datos (se muestra un ejemplo de ejecución para `n=5000` y `nprocs = 3` sin indicar tiempos reales):

Primos menores que 5000: 669. Tiempo secuencial: 6.00 s.
Primos menores que 5000: 669. Tiempo paralelo : 2.00 s. Tiempos parciales: 2.00 1.99 2.00
Speed-up: 3.00, Eficiencia: 1.00

**Memoria a entregar:** Se debe entregar una memoria que contenga:

1. Explicación de los pasos seguidos con comentarios del programa.
2. ¿Por qué los tiempos de algunos procesos son bastante inferiores a los del resto?
3. Gráficas comentadas de las mediciones obtenidas del speed-up y la eficiencia. Las gráficas se deben obtener para los distintos valores de `n` y para 2, 3, 4, 5, 6, 7 y 8 procesos. Por ejemplo, unas gráficas para $n = 5000000$ deben contener la siguiente información:




Para estas gráficas, se debe contestar a las siguientes preguntas: ¿Por qué el speed-up y la eficiencia no se acercan, en especial en algunos casos, al speed-up y eficiencia perfectos? ¿Explica qué solución podrías adoptar?

Ficheros a entregar:

**primos_mpi.c** Contendrá la unidad principal y la función `numero_primos()` que devolverá el número de primos calculado por cada proceso siguiendo una distribución cíclica de los datos. También contendrá la función ya suministrada `esPrimo()` que debe utilizar la función `numero_primos()`.

**makefile** Makefile utilizado para la compilación.


**DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS**

**PRÁCTICA 3:** TCOM

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/TCOM (CUENTA=cuenta personal).

Los ficheros o directorios que sea necesario crear pero no vayan a ser entregados al profesor deberán crearse directamente en el directorio /home/CUENTA y no dentro del subdirectorio de prácticas TCOM.

El objetivo de este ejercicio es evaluar los valores de los parámetros que determinan el modelo de coste de las comunicaciones en la plataforma del laboratorio. El coste de las comunicaciones entre dos procesadores determinados viene dado por la expresión:

$$t_{com} = \beta + \tau \cdot \text{Tamaño\_Mensaje}.$$

Los parámetros $\beta$ y $\tau$ indican respectivamente la latencia necesaria para el envío de un mensaje y el tiempo necesario para enviar un byte. Estos parámetros se pueden estimar de la siguiente forma:

- El valor de $\beta$ será el tiempo necesario en enviar un mensaje sin datos.
- El valor de $\tau$ será el tiempo requerido para enviar un mensaje menos el tiempo de latencia, dividido por el numero de bytes del mensaje.

Dado que `MPI_Send` y `MPI_Recv` no admiten un mensaje sin datos, lo que haremos para estimar el valor de la latencia $\beta$ es comunicar un solo byte, usando el tipo `MPI_BYTE`, a partir de por ejemplo una variable del tipo `char`. El tipo de dato `MPI_BYTE` no se corresponde con un tipo en C y es un dato de 8 bits sin interpretación alguna.

Teniendo en cuenta que se utiliza el protocolo TCP/IP como medio de transmisión de los mensajes, realmente el valor del tiempo de transferencia será diferente para mensajes pequeños y grandes, ya que el protocolo de comunicaciones fragmenta los mensajes en bloques. Se pide estimar el valor del parámetro $\tau$ para tamaños de mensaje variando entre los siguientes tamaños: 256 bytes (por ejemplo, $256/8 = 32 = 2^5$ reales en doble precisión, `double` en C y `MPI_DOUBLE` con MPI), 512 bytes ($2^6$ `double`'s), 1K (1024 bytes, $2^7$ `double`'s), 2K ($2^8$ `double`'s), 4K, 8K, 16K, 32K, 64K, 128K, 256K, 512K, 1MB (1024K), 2MB, 4MB ($2^{19}$ `double`'s).

Como sería imposible sincronizar los relojes perfectamente entre ambos procesos, para la medición del tiempo de comunicaciones, deberemos utilizar dos mensajes (uno de ida y otro de vuelta, usualmente conocido como ping/pong) y dividir el tiempo entre dos. Se deberán realizar varias repeticiones y tomar la media de ellas, descartando, en su caso, los valores atípicos.

Para la medición de tiempos se utilizará la función `MPI_Wtime`. Expresar los resultados en microsegundos. Teniendo en cuenta que la función `MPI_Wtime` devuelve segundos, no tendremos más que multiplicar por $10^6$ para expresarlo en microsegundos.

**Memoria a entregar:** Se debe entregar una memoria que contenga:

1. Explicación de los pasos seguidos.
2. Listado comentado del programa.
3. Valores de $\beta$ y $\tau$ (este último para los diferentes tamaños de mensaje).
4. Gráficas comentadas de las mediciones obtenidas.

Ficheros a entregar:

**tcom.c** Unidad principal.

**makefile** Makefile utilizado.


**DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS**

# PRÁCTICA 4: ARISTAS

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/ARISTAS (CUENTA=cuenta personal).

Los ficheros o directorios que sea necesario crear pero no vayan a ser entregados al profesor deberán crearse directamente en el directorio /home/CUENTA y no dentro del subdirectorio de prácticas ARISTAS.

En esta práctica se desea que se realice la implementación paralela, sobre MPI, de aplicación de una máscara de Laplace para extraer contornos o aristas de una imagen.

Las imágenes con las que trabajar estarán en formato PGM debido a su facilidad de uso. Los archivos PGM son archivos portables que contienen datos de imagen en forma de mapas de escala de grises. Estos archivos pueden guardarse mediante dos representaciones: como texto sin formato (ASCII) o como archivos binarios, indicado en ambos casos en el encabezado del archivo PGM. Si el encabezado incluye la cadena "P2" se trata de texto, mientras que si la cadena es "P5" es una representación binaria. En nuestra práctica usaremos la representación binaria. El encabezado del archivo contiene diversos datos, como la anchura y la altura de la imagen y el número máximo del valor de gris (normalmente 255). La información de la imagen se almacena en un array 2D con distintos números enteros, que representan la escala de gris del píxel correspondiente de la imagen, desde negro (0) hasta blanco (255).

Para almacenar estas imágenes usaremos una estructura en c:

```
typedef   struct {
          int row;          // alto de la imagen
          int col;          // ancho de la imagen
          int max_gray;     // máximo valor de gris
          int **matrix;     // array 2D con los valores de cada píxel
} PGMData;
```

La máscara para extraer contornos o aristas de una imagen corresponde a una pequeña matriz de $3 \times 3$ elementos con contenido:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix},$$

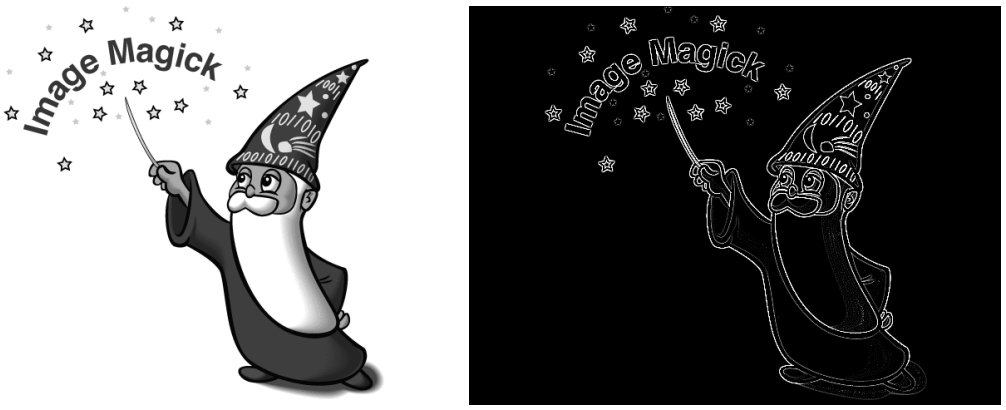
que se aplica a los elementos de la imagen original y su efecto es resaltar los contornos de la imagen. Su aplicación sobre la matriz de píxeles de una imagen consiste en:

$$\begin{aligned} imagen\_edge[i][j] \leftarrow\ & 8 * imagen[i][j] \\ & - imagen[i-1][j-1] - imagen[i-1][j] - imagen[i-1][j+1] \\ & - imagen[i][j-1] - imagen[i][j+1] \\ & - imagen[i+1][j-1] - imagen[i+1][j] - imagen[i+1][j+1] \end{aligned}$$

Existen diferentes estrategias para tratar los píxeles de los bordes (que no tienen 8 vecinos alrededor). Una de las más comunes consiste en dejar a cero (negro) los bordes de la imagen. Otra posibilidad similar, que es la que aplicaremos en esta práctica, consiste en cambiar a 255 (blanco) estos bordes. Con ello podremos detectar ciertos errores en la construcción del algoritmo paralelo.

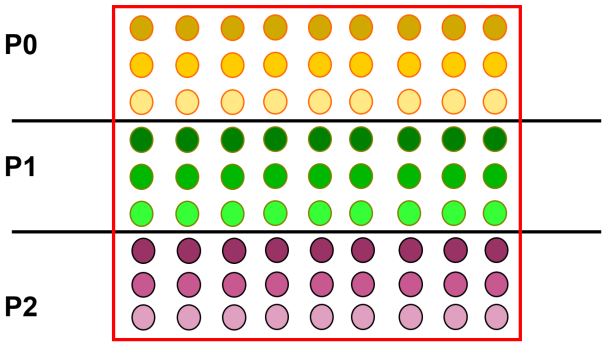
Tras filtrar la imagen con la máscara anterior, los píxeles de valor menor que 0 se convierten al valor mínimo (0, negro), y los de valor mayor que 255, al valor máximo (255, blanco); ver versión secuencial, aristas.c).

Las siguientes imágenes muestran un ejemplo de aplicación a partir de una imagen original:

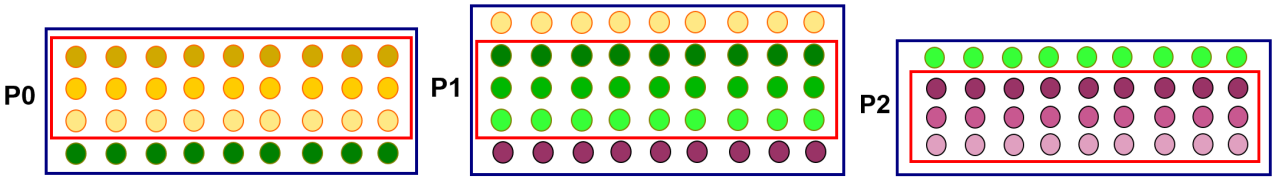



La implementación debe usar un esquema de variables similar a la implementación secuencial suministrada. El proceso número 0 (proceso root) se encargará de leer la imagen original. Adicionalmente, una vez aplicado el filtro de Laplace de forma paralela, será el responsable de guardar la imagen de aristas.

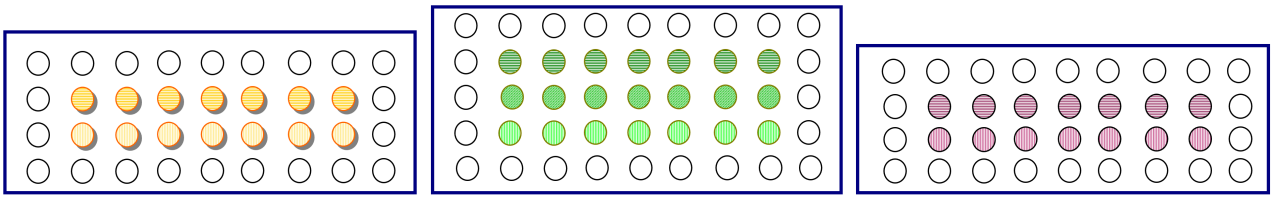
La implementación paralela del algoritmo estará basada en una descomposición unidimensional por bloques de filas consecutivas de la matriz de píxeles de la imagen. Si el número de procesos no divide al número de filas, el exceso de filas se le asociará al proceso número 0 (proceso root) (ver ejemplo psdot.c). Cada proceso será responsable de la actualización de una o más filas adyacentes de la matriz de píxeles. La siguiente imagen muestra la distribución de píxeles de una imagen $9 \times 9$ entre 3 procesos:



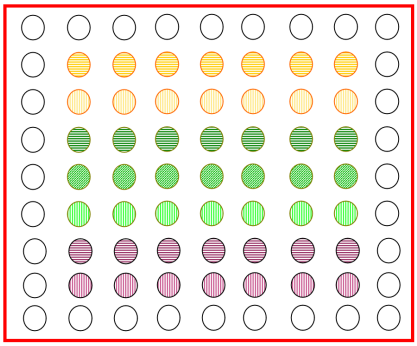

Teniendo en cuenta el esquema de dependencias en el que la actualización de cada píxel se realiza a partir de sus ocho vecinos más próximos, cada proceso necesitará el siguiente almacenamiento en el ejemplo anterior.




A partir de estas imágenes locales, cada proceso aplica la máscara sobre su imagen.



Una vez finalizado, las filas correspondientes de las imágenes locales se envían al proceso root que guardará la imagen de aristas.



Ficheros a entregar:

**aristas_mpi.c** Contendrá la unidad principal.

**aux.c** Contiene las funciones auxiliares (ya suministrado con la versión secuencial).

**Archivos de imágenes** Imágenes usadas como input.

**makefile** Makefile utilizado para la compilación.

## Unidad `aristas_mpi`

- **Lectura de datos**: El proceso 0 se encarga de leer la imagen original en la estructura `img_data` del tipo `PGMData`.
- **Envío y recepción de datos comunes a todos los procesos**: el número de filas y columnas de la imagen, `row` y `col` (`row = img_data.row` y `col = img_data.col`).
- **Cálculo del número de filas asignadas a cada proceso**: Cada proceso calcula el número de filas con las que va a trabajar: `rowlocal`.
- **Envío y recepción de filas de la matriz de píxeles de la imagen**: El proceso 0 envía las filas de `img_data.matrix`. El resto de procesos no es necesario que reciban dentro de una estructura `PGMData`; pueden recibir sus filas en un array 2D de enteros, por ejemplo en el array `matrix`. Notar que además de recibir un total de `rowlocal` filas, cada proceso distinto del 0 debe disponer también de la fila superior y de la fila inferior para poder realizar correctamente el filtro de Laplace (el último proceso solo dispondrá adicionalmente de la fila superior).
- **Aplicación del filtro de Laplace**:
  - El proceso 0 aplica el filtro de Laplace solo a sus filas de la matriz de píxeles de la imagen. El resultado lo almacena en el array `img_edge.matrix` de la estructura `img_edge` del tipo `PGMData`.
  - Los procesos distintos de 0 aplican el filtro de Laplace sobre el array 2D `matrix`. El resultado del filtro lo almacenan en otro array 2D, `matrix_edge`.
- **Envío y recepción de resultados**:
  - El proceso 0 recibe las distintas filas correspondientes a la aplicación del filtro de Laplace por parte del resto de procesos. La recepción se realiza directamente en el array `img_edge.matrix` de la estructura `PGMData`.
  - Los procesos distintos de 0 envían sus filas del array 2D, `matrix_edge`.
- **Escritura de resultados:** El proceso 0 guarda la imagen resultado de la detección de contornos, `img_edge`, en archivo, siguiendo el formato `PGM`. El nombre del archivo debe ser `nombreImagen_edge_paralelo.pgm`, donde `nombreImagen` se refiere al nombre del archivo con la imagen original, `nombreImagen.pgm`.


**DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS**

**PRÁCTICA 5:** MMmalla

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/MMmalla (CUENTA=cuenta personal).

Los ficheros o directorios que sea necesario crear pero no vayan a ser entregados al profesor deberán crearse directamente en el directorio /home/CUENTA y no dentro del subdirectorio de prácticas MMmalla.

En esta práctica se desea que se realice la implementación paralela, sobre MPI, del producto matriz por matriz, utilizando el algoritmo desarrollado para una arquitectura de malla (algoritmo de Cannon). Las matrices `A` y `B` deben estar particionadas en `r` $\times$ `r` bloques (`r` variable y no superior a `rmax=4`). Todos los bloques deben ser del mismo tamaño dado por la variable `bloqtam` que no debe ser superior a `maxbloqtam=100`. Todos los bloques de matrices que son necesarios definir: `A,B,C,ATMP` se considerarán almacenados en un vector. Por ejemplo, si `A`$= [a_{ij}]_{0\leq i,j\leq m-1}$, los elementos de esta matriz estarán almacenados en el vector:

$$a = (a_{0,0}, a_{0,1}, \ldots, a_{0,m-1}, a_{1,0}, a_{1,1}, \ldots, a_{1,m-1}, \ldots, a_{m-1,0}, a_{m-1,1}, \ldots, a_{m-1,m-1}).$$

Esto permite trabajar más fácilmente con las comunicaciones.

Podemos establecer el producto `c = a·b`, cuando las matrices implicadas están almacenadas en forma de vector y de orden `m`. La siguiente función obtiene este producto y adicionalmente lo acumula en `c` (`c = c + a·b`, operación que se necesita en el algoritmo):

```
void mult(double a[], double b[], double *c, int m) {
int i,j,k;
for (i=0; i<m; i++)
    for (j=0; j<m; j++)
       for (k=0; k<m; k++)
          c[i*m+j]=c[i*m+j]+a[i*m+k]*b[k*m+j];
return; }
```

Esta función puede ser utilizada para obtener los distintos productos por bloques que calculan los procesos.

Vamos a dar ahora unas nociones sobre cómo programar este producto matriz por matriz sobre MPI sin utilizar topologías de procesos. Con MPI no hay restricciones sobre qué tareas pueden comunicarse con otras. Sin embargo, para este algoritmo deseamos ver los procesos como situados en una malla abierta. Utilizaremos para ello la numeración que cada proceso obtiene cuando llama a `MPI_Comm_rank(MPI_COMM_WORLD, &myrank)`. El proceso 0 (proceso `root`) se encargará de obtener por el input estándar el único dato necesario: el orden de cada bloque, `bloqtam`. Además, comprobará que el número de procesos, `nproc`, sea un cuadrado perfecto de manera que se ajuste a la estructura de una malla cuadrada. Posteriormente enviará el parámetro necesario a los otros procesos, `bloqtam`. Las tareas con número de orden distinto de

cero recibirán dicho parámetro. Notar que ésta es la única diferencia que hay entre lo que debe hacer el proceso 0 y el resto.

Ahora, cada tarea necesita conocer la `fila` y la `columna` en la que se encuentra localizada en la malla. Esto es sencillo, ya que la división entera `myrank/r` nos da la `fila` de la tarea y el resto de esa división entera nos da la `columna` (`fila, columna = 0,1,2,r-1`). Ahora, usando las variables `myrank`, `nproc` y `r` podemos conocer sin demasiada dificultad los números de orden de los procesos situadas `arriba` y `abajo` en la misma columna. Estos números de orden serán utilizados para la posterior rotación de los bloques de `B` entre las columnas.

En cada etapa del algoritmo se envía un bloque de matriz `A` a todas las tareas de una misma fila. Así pues, es conveniente almacenar en un array, que llamaremos `mifila[]`, los números de orden de las tareas situadas en la fila de la tarea en cuestión.

Como este algoritmo supone que cada bloque está almacenado ya en cada tarea, cada proceso definirá su bloque `a` como:

$$\texttt{a[i]=i*(float)(fila*columna+1)}^2\texttt{/bloqtam}^2\texttt{, i=0,1,2,...,bloqtam*bloqtam-1},$$

y los bloques de `b` de manera que `B` sea la matriz identidad. Esto permitirá chequear al final del algoritmo si `C=A`.

Finalmente, como ya conocemos todas las variables necesarias se procede con el bucle principal del algoritmo.

Se aconseja que se usen distintas etiquetas de mensaje en cada iteración del algoritmo y que se especifique el número de orden en `MPI_Recv()`, es decir, que no se use como número de orden el valor `MPI_ANY_SOURCE`.

Cuando los cálculos estén terminados se comprobará que `A=C` para verificar que la multiplicación se ha efectuado correctamente.

Cada proceso debe contar el número de errores que ha cometido (si todo funciona correctamente serán cero errores) y enviarle la información al proceso `root`, el cual escribirá por la salida estándar el número de errores para cada proceso identificado por su número de orden.

Ficheros a entregar:

**matriz.c** Contendrá la unidad principal y la función *mult* que calcula el producto de dos matrices almacenadas como vectores.

**makefile** Makefile utilizado para la compilación.


# DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS
## PRÁCTICA 6: FW

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/FW (CUENTA=cuenta personal).

Los ficheros o directorios que sea necesario crear pero no vayan a ser entregados al profesor deberán crearse directamente en el directorio /home/CUENTA y no dentro del subdirectorio de prácticas FW.

En esta práctica se desea que se realice la implementación paralela, sobre MPI, del algoritmo de Floyd-Warshall para el cálculo de los caminos más cortos entre todos los pares de vértices en un grafo ponderado.

La implementación paralela del algoritmo de Floyd-Warshall estará basada en una descomposición unidimensional por bloques de filas consecutivas de las matrices intermedias en cada iteración. Si el número de procesos no divide al número de filas, el exceso de filas se le asociará al proceso número 0 (proceso root) (ver ejemplo psdot.c). Cada proceso será responsable de la actualización de una o más filas adyacentes de la matriz de pesos y de la matriz de caminos.

La implementación debe usar un esquema de variables similar a la implementación secuencial suministrada. En particular, deben existir unos parámetros fijos en el programa:

**maxn** Máximo número de vértices en el grafo (=1000).

**maxnumprocs** Máximo número de procesos (=8).

La generación del grafo (definición de la matriz de pesos) se realizará también de la misma forma que en la versión secuencial. Esta generación se realizará por el proceso root, debiendo ser enviado cada bloque de filas correspondiente a cada proceso distinto del root. Para este envío debe usarse la función `MPI_Scatter()`.

Cuando se finalice el algoritmo, el proceso root recibirá del resto de procesos, las filas evaluadas de la matriz de distancias y de la matriz de caminos. Esta comunicación debe realizarse con la función `MPI_Gather()`.

De forma similar a como se hace en la versión secuencial, el proceso root debe mostrar la matriz de pesos y la matriz de distancias final solo cuando el número de vértices sea menor o igual que 10. Adicionalmente, debe ser capaz de mostrar el peso y el camino más corto entre dos vértices dados.

Ficheros a entregar:

**fw.c** Contendrá la unidad principal y todas las funciones auxiliares.

**makefile** Makefile utilizado para la compilación.