DESARROLLO DE SOFTWARE

EN

ARQUITECTURAS PARALELAS

Grado en Ingeniería Informática

CURSO 2020/2021

Prácticas

Curso 2020/2021

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 son:

Práctica	**Peso**	**Observaciones**
ANILLO	10 %	–
PSDOTMPI	20 %	–
TCOM PIMPI	20 %	Elegir una
MVPI FW	25 %	Elegir una
MMmalla MANDELBROT	25 %	Elegir una
IMAGEN JAC-ELECT	Práctica voluntaria Práctica voluntaria	

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:

   - Cuenta en servidor de computo cluster3: Se le proporcionará al alumnado una cuenta en el servidor de computo cluster3. A dicho servidor se puede acceder de forma remota y será en el que se llevará a cabo la entrega de prácticas.
   - Laboratorio L01: 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 (colgar anuncios en UACloud, ...)

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.

   En el laboratorio L01 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. Además se deberá avisar al profesor mediante una tutoría en UACloud que la práctica ha sido entregada y puede ser corregida.

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.

El nombre del directorio de entrega ANILLO debe estar en mayúsculas. 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`). En esta práctica se desea que se realice la implementación paralela, sobre MPI, que encadene el envío y recepción de un mensaje. Los mensajes se enviarán de forma encadenada, lo que quiere decir que el primero enviará un mensaje al segundo, el segundo recibirá uno del primero y enviará uno al tercero, y así sucesivamente para todos los procesos lanzados. El proceso número 0 (proceso root) debe aceptar un input estándar correspondiente a un número entero. Este entero se lo enviará al proceso número 1, el cual mostrará en pantalla el entero recibido, a continuación le sumará una unidad y se lo enviará al proceso número 2, y así sucesivamente. El último proceso, tras recibir el entero correspondiente, mostrará en pantalla dicho entero recibido, le sumará una unidad y enviará el entero resultante al proceso root, concluyendo así la comunicación en anillo.

Output: Todo proceso al recibir el entero debe mostrarlo en pantalla de la forma "Soy el proceso X. El entero que he recibido es: I", siendo X el rango del proceso y I el entero recibido. Es decir, si el valor del entero introducido inicialmente por teclado era 5, el proceso 1 mostrará "Soy el proceso 1. El entero que he recibido es : 5", etc. En la Figura 1 se muestra un ejemplo de salida al lanzar 4 procesos: mpirun -np 4 anillo

```
Introduce un entero para transmitir:
5
El entero introducido es 5
Soy el proceso 1. El entero que he recibido es: 5
Soy el proceso 2. El entero que he recibido es: 6
Soy el proceso 3. El entero que he recibido es: 7
Soy el proceso 0. El entero que he recibido es: 8
```

Figura 1: Ejemplo de output

Ficheros a entregar:

anillo.c Código paralelo que contendrá la unidad principal.

makefile Makefile utilizado para la compilación.

DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS

PRÁCTICA 2: PSDOTMPI

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/PSDOTMPI (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 PSDOTMPI.

El nombre del directorio de entrega PSDOTMPI debe estar en mayúsculas. En esta práctica se desea que se realice la implementación paralela, sobre MPI, del producto escalar de dos vectores $x, y \in \mathbb{R}^n$, es decir:

$$< x, y >= \sum_{i=1}^{n} x_i y_i.$$

Ejemplo: El producto escalar de los vectores $x = (1, \frac{1}{2}, \frac{1}{3}),\ y = (1, 2, 3)$, es

$$< x, y >= 1 \cdot 1 + \frac{1}{2} \cdot 2 + \frac{1}{3} \cdot 3 = 3.$$

La idea básica para paralelizar este producto escalar usando p procesos consiste en dividir el trabajo a realizar lo más equitativamente posible entre los procesos. Para ello asignaremos a cada proceso distinto del proceso 0 el cálculo relacionado con un total de n/p (división entera) componentes consecutivas de los vectores x e y. El proceso 0 será el responsable del cálculo para las primeras $n/p + mod(n, p)$ componentes. Por ejemplo, si n = 103 y p = 4, el cálculo local que debe realizar cada proceso es el siguiente:

p_0 :	$x_1y_1 + x_2y_2 + \ldots + x_{28}y_{28}$	28 componentes asignadas a p_0
p_1 :	$x_{29}y_{29} + x_{30}y_{30} + \ldots + x_{53}y_{53}$	25 componentes asignadas a p_1
p_2 :	$x_{54}y_{54} + x_{55}y_{55} + \ldots + x_{78}y_{78}$	25 componentes asignadas a p_2
p_3 :	$x_{79}y_{79} + x_{80}y_{80} + \ldots + x_{103}y_{103}$	25 componentes asignadas a p_3

Los vectores para los que hay que calcular el producto escalar son:

$$x(k) = 1{,}0/(k+1),\quad y(k) = k+1,\quad k = 0, 1, 2, \ldots, n-1.$$

La implementación ha de desarrollarse de forma que será el proceso 0 el que calcule los vectores $x,\ y$. El esquema a seguir es el siguiente:

- Los arrays x, y se han de crear en memoria dinámica haciendo uso de malloc(). Notar que la asignación de memoria dinámicamente han de llevarla a cabo todos los procesos.
- El proceso 0 efectúa la lectura de la longitud de los vectores a multiplicar.

- El proceso 0 define los dos vectores x e y.
- El proceso 0 manda cada porción de vector x e y a cada uno del resto de procesos usando la rutina de envío MPI_SEND. Los procesos distintos del 0 reciben las porciones de vector x e y usando la rutina de recepción MPI_RECV.
- Cada proceso (incluido el proceso 0) realiza el producto escalar a partir de los datos recibidos.
- Cada proceso distintinto del 0 envia al 0 su producto escalar parcial.
- El proceso 0 suma los productos escalares parciales calculados por todos los procesos.
- El proceso 0 visualiza los resultados: el producto escalar de x e y.

Deben existir unos parámetros fijos:

- **maxnprocs = 8:** Máximo número de procesos.
- **maxn = 100000000:** Dimensión máxima para los vectores x e y.

Ficheros a entregar:

- Código paralelo: psdotmpi.c
- makefile

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 nombre del directorio de entrega TCOM debe estar en mayúsculas. El objetivo de este ejercicio es evaluar los valores de los parámetros β y τ 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 β y τ 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 β será el tiempo necesario para enviar un mensaje sin datos.
- El valor de τ será el tiempo requerido para enviar un mensaje menos el tiempo de latencia, dividido por el numero de bytes del mensaje, i.e.,

$$\tau = \frac{t_{com}(\text{Mensaje}) - \beta}{\text{Tamaño\_Mensaje}}.$$

Dado que `MPI_Send` y `MPI_Recv` no admiten un mensaje sin datos, lo que haremos para estimar el valor de la latencia β es comunicar un solo byte, usando el tipo `MPI_BYTE`, a partir de por ejemplo una variable C 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 τ para tamaños de mensaje variando entre los siguientes tamaños:

- 256 bytes: Teniendo en cuenta que en C un tipo de dato `double` utiliza 8 bytes, para enviar un mensaje de 256 bytes se puede enviar un mensaje con $\frac{256}{8} = 32$ `double`'s. Por ejemplo, se puede enviar un array `x` de tipo `double` en C, `MPI_DOUBLE` en MPI, con tamaño $256/8 = 32 = 2^5$.
- 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 1000 repeticiones de cada envío y tomar la media aritmética de ellas, descartando, en su caso, los valores atípicos.

Para la medición de tiempos se utilizará la función `MPI_Wtime`, cuyo uso queda ilustrado en el siguiente código

```
...
double start_time, end_time;
. . .
start_time = MPI_Wtime();
...
... calculamos
...
end_time = MPI_Wtime();
Printf(‘‘Tiempo de ejecucion = %f seg\n’’, end_time - start_time);
```

Los resultados se tienen que expresar en microsegundos. Teniendo en cuenta que la función `MPI_Wtime` devuelve segundos, para expresar el tiempo en microsegundos se tendrá que multiplicar el tiempo obtenido por 10^6.

Nuestro objetivo es evaluar la red de comunicaciones en la plataforma del laboratorio. Por este motivo, tienes que asegurarte que la comunicación ping/pong se efectúa entre dos ordenadores distintos. Ten en cuenta que al ejecutar en dos procesos, aunque utilices la opción `-machinefile`, mpi trabajará en local, es decir lanzará los dos procesos en la misma máquina pues se trata de un multicore. Si queremos utilizar menos procesos por nodo podemos utilizar la opción `-npernode`. Por ejemplo, si queremos ejecutar dos procesos, cada uno en una máquina distinta:

```
mpirun -machinefile machinefile -np 2 -npernode 1 ejecutable
```

Como trabajo complementario, no obligatorio, puedes comparar los resultados obtenidos con los que se obtienen al efectuar las comunicaciones ping/pong entre dos cores de un solo servidor. En este caso, mpi está gestionando el paso de mensajes a través de la memoria RAM local y sin utilizar la red ethernet.

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

1. Explicación de los pasos seguidos.
2. Listado comentado del programa.
3. Valores de β y τ (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: PIMPI

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/PIMPI (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 PIMPI .

En esta práctica se desea que se realice la implementación paralela del **cálculo de** π, utilizando la regla del rectángulo y haciendo uso de MPI. La implementación ha de desarrollarse siguiendo el siguiente esquema:

- El proceso master efectúa la lectura del número de intervalos n por la entrada estándar y envía este dato al resto de procesos.
- La idea básica para paralelizar este algoritmo usando p procesos consiste en dividir el trabajo a realizar lo más equitativamente posible entre los procesos. Para ello asignaremos a cada proceso distinto del proceso 0 el cálculo relacionado con un total de n/p (división entera) intervalos. El proceso 0 será el responsable del cálculo para las primeras $n/p + mod(n, p)$ áreas de los rectángulos (se procedió de forma similar en la práctica PSDOTMPI).
- Debéis calcular el resultado final de π como suma de los resultados parciales obtenidos por cada proceso, de la forma que consideréis más eficiente.
- La función $f(x) = 4/(1 + x^2)$ se calculará mediante una función.
- Al finalizar el cálculo de π en paralelo, el proceso master también ha de calcular en secuencial el número π.
- El proceso master debe visualizar en pantalla:
  - El valor obtenido para π en paralelo.
  - El tiempo necesitado para calcular π en paralelo. El error para esta aproximación tomando $\pi = 3{,}14159265358979323846264 3$.
  - El valor obtenido para π en secuencial.
  - El tiempo necesitado para calcular π en secuencial.
  - El error para esta aproximación secuencial tomando $\pi = 3{,}14159265358979323846264 3$.
  - El speed-up y la eficiencia.

Ficheros a entregar:

pimpi.c unidad principal.

makefile Makefile utilizado para la compilación.

DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS

PRÁCTICA 5: MVPI

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/MVPI (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 MVPI.

En esta práctica se desea que se realice la implementación paralela, sobre MPI, del **producto matriz vector**, utilizando el algoritmo basado en el producto interno, con distribución equitativa de todas las filas de la matriz entre los procesos. 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 enunciado de la práctica PSDOTMPI o los ejemplos ejemplo8.c, envioarray.c). En la implementación se debe crear una unidad llamada matvec que multiplique una matriz por un vector y devuelva su resultado. Deben existir unos parámetros fijos en el programa:

maxm Máximo número de filas en la matriz (=500).

maxn Máximo número de columnas en la matriz (=500).

maxnumprocs Máximo número de procesos (=8).

La matriz con la que se trabaje será de orden `m` × `n`, y el vector `x` de orden `n`, ambos reales doble precisión. Los valores de `m` y `n` deben ser variables y controlados de manera que no sobrepasen los valores máximos. La matriz y el vector serán **dimensionados de forma dinámica** y definidos por el proceso 0 (proceso root) como:

```
a[i][j]=(i+j), i=0,1,2,...,m-1, j=0,1,2,...,n-1,

x[j]=j, j=0,1,2,...,n-1,
```

debiendo ser enviado cada bloque de filas correspondiente a cada proceso distinto del root.

Cuando todos los procesos hayan finalizado sus cálculos, estos deben ser enviados al proceso root, el cual escribirá la solución global. Los procesos distintos del root también deben escribir su solución parcial, junto con la identificación del número de proceso. En la Figura 2 se muestra un ejemplo de salida.

```
Numero de filas (1 - 1000):
10
Numero de columnas (1 - 1000):
10
     x =   0.000   1.000   2.000   3.000   4.000
           5.000   6.000   7.000   8.000   9.000

   A=     0       1       2       3       4       5       6       7       8       9
       0:  0.000   1.000   2.000   3.000   4.000   5.000   6.000   7.000   8.000   9.000
       1:  1.000   2.000   3.000   4.000   5.000   6.000   7.000   8.000   9.000  10.000
       2:  2.000   3.000   4.000   5.000   6.000   7.000   8.000   9.000  10.000  11.000
       3:  3.000   4.000   5.000   6.000   7.000   8.000   9.000  10.000  11.000  12.000
       4:  4.000   5.000   6.000   7.000   8.000   9.000  10.000  11.000  12.000  13.000
       5:  5.000   6.000   7.000   8.000   9.000  10.000  11.000  12.000  13.000  14.000
       6:  6.000   7.000   8.000   9.000  10.000  11.000  12.000  13.000  14.000  15.000
       7:  7.000   8.000   9.000  10.000  11.000  12.000  13.000  14.000  15.000  16.000
       8:  8.000   9.000  10.000  11.000  12.000  13.000  14.000  15.000  16.000  17.000
       9:  9.000  10.000  11.000  12.000  13.000  14.000  15.000  16.000  17.000  18.000

Soy el proceso 1 de un total de 4.
   A=     0       1       2       3       4       5       6       7       8       9
       0:  4.000   5.000   6.000   7.000   8.000   9.000  10.000  11.000  12.000  13.000
       1:  5.000   6.000   7.000   8.000   9.000  10.000  11.000  12.000  13.000  14.000

     y = 465.000 510.000

Soy el proceso 2 de un total de 4.
   A=     0       1       2       3       4       5       6       7       8       9
       0:  6.000   7.000   8.000   9.000  10.000  11.000  12.000  13.000  14.000  15.000
       1:  7.000   8.000   9.000  10.000  11.000  12.000  13.000  14.000  15.000  16.000

     y = 555.000 600.000

Soy el proceso 3 de un total de 4.
   A=     0       1       2       3       4       5       6       7       8       9
       0:  8.000   9.000  10.000  11.000  12.000  13.000  14.000  15.000  16.000  17.000
       1:  9.000  10.000  11.000  12.000  13.000  14.000  15.000  16.000  17.000  18.000

     y = 645.000 690.000

   A*x = 285.000 330.000 375.000 420.000 465.000
         510.000 555.000 600.000 645.000 690.000
```

Figura 2: Ejemplo de output

Se proporciona versión secuencial mv_sec que incluye las unidades vermatriz, vervector que se deberán utilizar en la versión paralela.

Ficheros a entregar:

mvpi.c Contendrá las siguientes unidades:

- Unidad principal.
- Unidad vermatriz ya suministrada para visualizar una matriz.
- Unidad vervector ya suministrada para visualizar un vector.
- Unidad matvec que multiplique una matriz por un vector y devuelva su resultado.

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). 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.

DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS

PRÁCTICA 7: 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:

`a[i]=i*(float)(fila*columna+1)`^{2}`/bloqtam`^{2}`, 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 8: MANDELBROT

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/MANDELBROT (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 MANDELBROT.

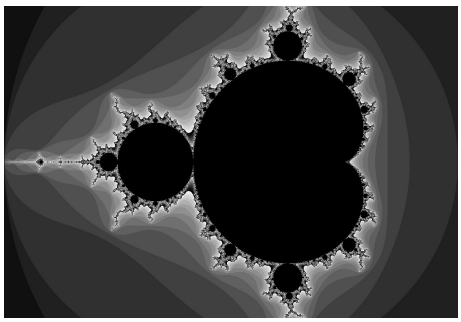
Los fractales son representaciones gráficas de funciones matemáticas que suelen parecerse a paisajes y formas de la naturaleza, donde pueden observarse muchos ejemplos de estructuras repetitivas que configuran hojas, dunas, etc. Como el objetivo no es dominar la teoría de fractales, lo que se abordará en esta práctica es la representación parcial de un fractal conocido como conjunto de Mandelbrot. Dicho conjunto está compuesto por los números complejos, $c = a + bi$, para los que la relación de recurrencia:

$$z_{k+1} = z_k^2 + c, \quad k = 0, 1, 2, \ldots,$$

converge a un número complejo finito, siendo $z_0 = 0$ la condición inicial.

Se puede demostrar que si existe un m para el cual $|z_m| > 2$, la sucesión diverge y entonces c no pertenece al conjunto de Mandelbrot. Como el valor de m, de existir, podría ser muy grande, la primera aproximación que se hace es llegar como máximo hasta el elemento $z_{IterMax}$. Si se alcanza dicho $z_{IterMax}$ sin que ningún $|z_i|$ sea mayor que 2, se supone que c pertenece al conjunto de Mandelbrot.

En esta práctica se desea que se realice la implementación paralela, sobre MPI, de la técnica para la obtención del conjunto de Mandelbrot.



La implementación debe usar un esquema de variables similar a la implementación secuencial suministrada. El proceso número 0 (proceso root) se encargará de leer, en su caso, los datos iniciales:

- El número de píxeles a considerar pixelXmax x pixelYmax. Por defecto 1024 x 1024.
- El número máximo de iteraciones IterMax. Por defecto, 1000.

- El nombre del archivo de salida con la imagen del conjunto de Mandelbrot obtenido. Por defecto imgA.pgm, imgB.pgm.
- Seleccionar un área del plano complejo de las consideradas previamente. Por defecto, la parte real varía en $[-2, 1]$ y la parte imaginaria se encuentra en $[-1{,}5, 1{,}5]$.

La implementación paralela debe basarse en la idea de establecer un pool de procesos. Definiremos una tarea como el procesado de una fila de píxeles de la imagen del conjunto de Mandelbrot. Estas tareas deben ir asignándose a los procesos conforme estos quedan inactivos.

Una vez finalizado el algoritmo, el proceso 0 guardará en archivo la imagen final del conjunto de Mandelbrot obtenido.

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

- Explicación de la implementación.
- Distintas ejecuciones con distinto número de procesos. Se presentarán tablas indicando para cada ejecución, el tiempo secuencial, el tiempo paralelo, el speed-up y la eficiencia. Para alguna ejecución se puede presentar la imagen del conjunto de Mandelbrot obtenida.

Las distintas ejecuciones pueden corresponder, por ejemplo, a los siguientes datos:

Ancho imagen	Alto imagen	Iteraciones	Dominio
1024	1024	100	0
		1000	1
		2000	3
		3000	4
		1000	6
		4000	7
		2000	8

Ficheros a entregar:

mandelbrot_mpi.c Contendrá la unidad principal y todas las funciones auxiliares.

makefile Makefile utilizado para la compilación.

Unidad mandelbrot_mpi

- Proceso 0: Lectura datos iniciales.
- Porceso 0: Se crea el archivo que contendrá la imagen y se escribe la cabecera.
- Envío y recepción de datos comunes a todos los procesos: `IterMax`, `pixelXmax`, `pixelYmax`, `RealMin`, `RealMax`, `ImMin` e `ImMax`.
- Cálculo de `AnchoPixel` y `AltoPixel`.
- Proceso 0: Crear la matriz de píxeles.
- Proceso 0: Asignar las primeras `nproc-1` filas de píxeles a los procesos distintos de 0 (filas `0,1,nproc-2`).
- Proceso 0:

  `for (pixelY=nproc-1;pixelY<pixelYmax;pixelY++)`

  Recibir resultado de un proceso identificado con `status.MPI_SOURCE`

  Realizar nueva asignación de trabajo: fila `pixelY` se asigna al proceso `status.MPI_SOURCE`
- Proceso 0: Recibir las últimas `nproc-1` filas de píxeles ya procesadas que quedarán pendientes.
- Proceso 0: Enviar señal de terminación a todos los procesos.
- Proceso 0: Escribir imagen en archivo.
- Procesos > 0: Crear el vector necesario para almacenar `pixelXmax` píxeles.
- Procesos > 0: Proceso iterativo (`while (1)`)
  - Recibir asignación de trabajo del proceso 0.
  - Comprobar si la etiqueta de envío indica que se debe parar.
  - Procesar la fila de píxeles asignada.
  - Enviar al proceso 0 la fila de píxeles obtenida.

DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS
PRÁCTICA 9: IMAGEN

La práctica se ENTREGARÁ en el directorio:

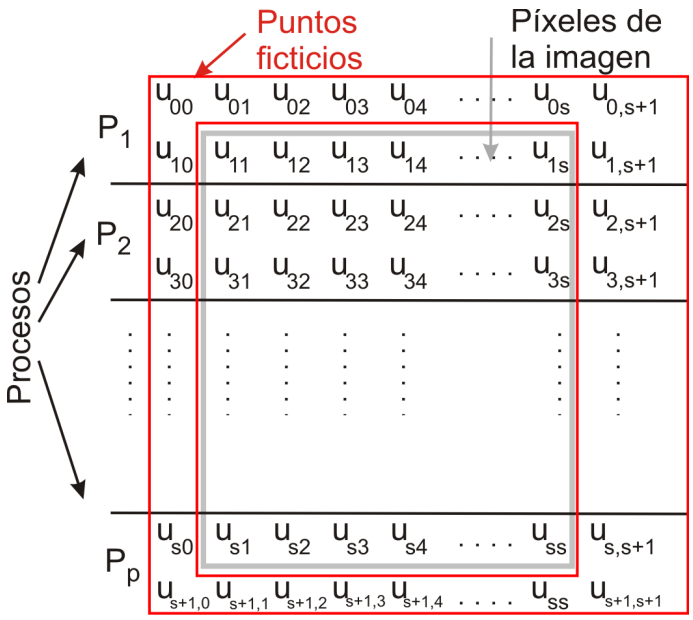
/home/CUENTA/IMAGEN (CUENTA=cuenta personal).

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 IMAGEN.

En esta práctica se desea que se realice la implementación paralela, sobre MPI, de la técnica para restaurar una imagen a partir de la salida que se obtiene después de aplicar el filtro de Laplace para la detección de aristas. Las siguientes imágenes muestran un ejemplo de aplicación a partir de una imagen original:

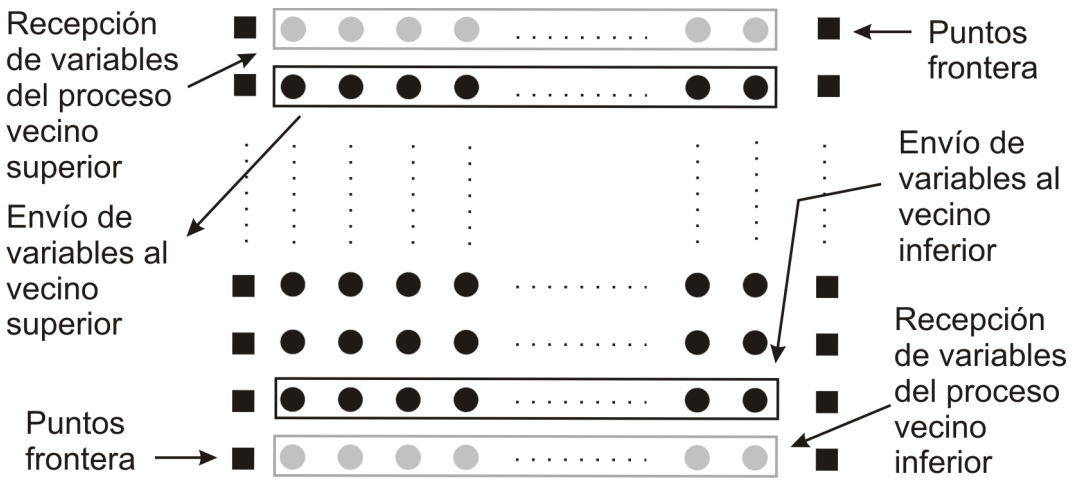
La implementación debe usar un esquema de variables similar a la implementación secuencial suministrada. El proceso número 0 (proceso root) se encargará de leer la imagen y cualquier dato inicial necesario (nombre del archivo con la imagen original y el número de iteraciones máximo, `IterMax`). Adicionalmente, será el responsable de aplicar el filtro de Laplace y, en su caso, guardar la imagen de aristas.

La implementación paralela del algoritmo de restauración estará basada en una descomposición unidimensional por bloques de filas consecutivas de la matriz de píxeles de la imagen una vez aplicado el filtro de Laplace. Si el número de procesos no divide al número de filas, el exceso de filas se le asociará al proceso número 0 (proceso root) (como se ha procedido en ocasiones anteriores). Cada proceso será responsable de la actualización de una o más filas adyacentes de la matriz de píxeles. Si las variables u_{ij} representan el valor del pixel (i, j), entonces, la siguiente figura ilustra la distribución de datos necesaria.



En la siguiente figura se muestra el esquema típico de cálculo y comunicación para un proceso cualquiera (intermedio), suponiendo que todas las variables se almacenan en un array dos dimensional `img_new[0:rowlocal+1][0:col+1]`.

- Puntos ficticios: marcados con un cuadrado negro. Almacenados en las columnas `0` y `col+1` de `img_new`, es decir, en las posiciones `img_new[0:rowlocal+1][0]` e `img_new[0:rowlocal+1][col+1]`.
- Cada proceso debe calcular un total de `rowlocal` $\times$ `col` variables; las indicadas con un círculo negro. Corresponden a las posiciones `[1:rowlocal][1:col]` del array `img_new`.
- Cada proceso P_i debe enviar las filas `1` y `rowlocal` del array `img_new` a los procesos P_{i-1} y P_{i+1}, respectivamente.
- Cada proceso P_i debe recibir de P_{i-1} y P_{i+1} en las filas `0` y `rowlocal+1` del array `img_old`.



El criterio de parada que debe utilizarse en cada iteración será:

$$\left(\sum_{i,j=1}^{s}(img\_new(i,j) - img\_old(i,j))^2\right)^{\frac{1}{2}} < \text{cota}, \quad \text{cota} = \sqrt{row * col}, \tag{1}$$

o que el número de iteraciones sea mayor o igual que `IterMax`.

Cada proceso $P_i,\ i = 1, 2, \ldots, nproc$, debe calcular su norma parcial, es decir, calculará

$$\sum_{i,j}(img\_new(i,j) - img\_old(i,j))^2.$$

Esta norma parcial será sumada utilizando `MPI_Allreduce()`, para posteriormente calcular su raíz cuadrada (`sqrt`). Se comprobará el criterio de parada (1). En caso de que no haya convergencia, el proceso iterativo continuará.

Una vez alcanzada la convergencia, se recolectarán los datos de la imagen restaurada en el proceso 0 mediante `MPI_Gather()`. El proceso 0 escalará y guardará en archivo la imagen final.

Ficheros a entregar:

imagen_mpi.c Contendrá la unidad principal y todas las funciones auxiliares.

Archivos de imágenes Imágenes usadas como input.

makefile Makefile utilizado para la compilación.

Unidad imagen_mpi

- Proceso 0: Lectura datos iniciales e imagen original (`img_data`).
- Proceso 0: Aplicación del filtro de Laplace para obtener `img_mod`.
- Proceso 0: Se guarda la imagen de aristas en archivo.
- Envío y recepción de datos comunes a todos los procesos: el número de iteraciones a realizar (`IterMax`), y el número de filas y columnas de la matriz de píxeles (`row` y `col`).
- Cálculo del número de filas asignadas a cada proceso: `rowlocal`.
- Proceso 0: Crear el array de reales `img_edge` a partir de `img_mod`.
- Envío y recepción de filas del array `img_edge`.
- Crear `img_old` e `img_new`. Inicializar `img_old`.
- Proceso iterativo (`while ((norma >cota) && (k <IterMax))`)
  - Actualizar `img_new` a partir de `img_old`.
  - Copiar `img_new` en `img_old`.
  - Calcular la porción de la norma correspondiente, entre img_new y img_old.
  - Llamar a MPI_Allreduce para poder obtener el valor final de la norma.
  - Comunicaciones entre vecinos. Notar que todo proceso debe recibir dos mensajes salvo el primero y el último que sólo reciben un mensaje.
- Copiar `img_new` final en `img_mod` (array de enteros).
- Recolectar en el proceso 0, mediante `MPI_Gather()` los cálculos de cada proceso almacenados en `img_mod`.
- Proceso 0: Escalar imagen restaurada `img_mod`.
- Proceso 0: Guardar imagen restaurada `img_mod` en archivo.

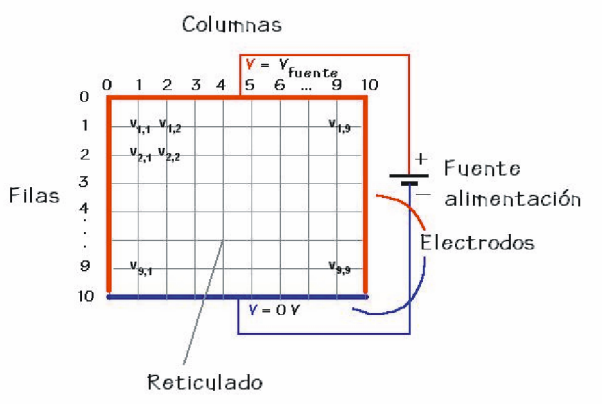
DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS
PRÁCTICA 10: JAC-ELECT

La práctica se ENTREGARÁ en el directorio:

/home/CUENTA/JAC-ELECT (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 JAC-ELECT.

En esta práctica se pretende obtener el campo eléctrico en la región comprendida entre dos electrodos de geometría sencilla, tal y como indica la figura:



Se desea que se realice la implementación paralela, sobre MPI, del método iterativo de Jacobi para resolver el problema anterior una vez discretizado.

Deben existir unos parámetros fijos en el programa:

smax Máximo número de nodos interiores (=200).

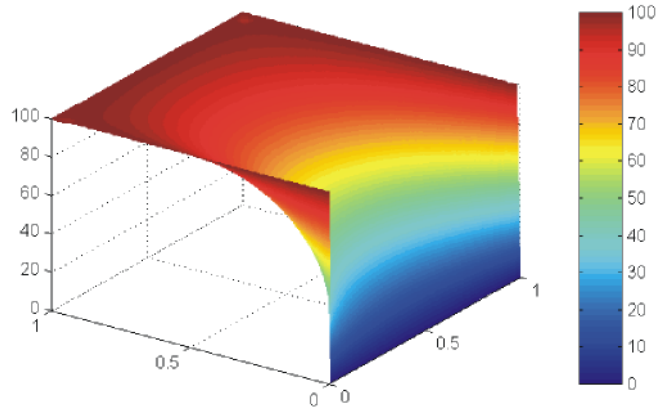
maxproc Máximo número de procesos totales (=8).

cota Cota de parada para el método iterativo (=0.01)

Las condiciones de contorno para nuestro caso particular, están estipuladas por el potencial aplicado a los electrodos. Por tanto, para el ejemplo de la figura en el que se ha considerado un número total de nodos interiores $s = 9$, en el cálculo teórico se supondrá una superficie límite con potenciales:

$$V_{0,j} = V_{i,0} = V_{i,10} = V_{fuente}, \quad V_{10,j} = 0V.$$

La obtención de este campo eléctrico sigue la ecuación de Laplace, por lo que podemos utilizar los estudios teóricos realizados en el tema correspondiente para dar solución a este problema, que gráficamente vendría dada por la siguiente figura en la que hemos considerado $V_{fuente} = 100V$:

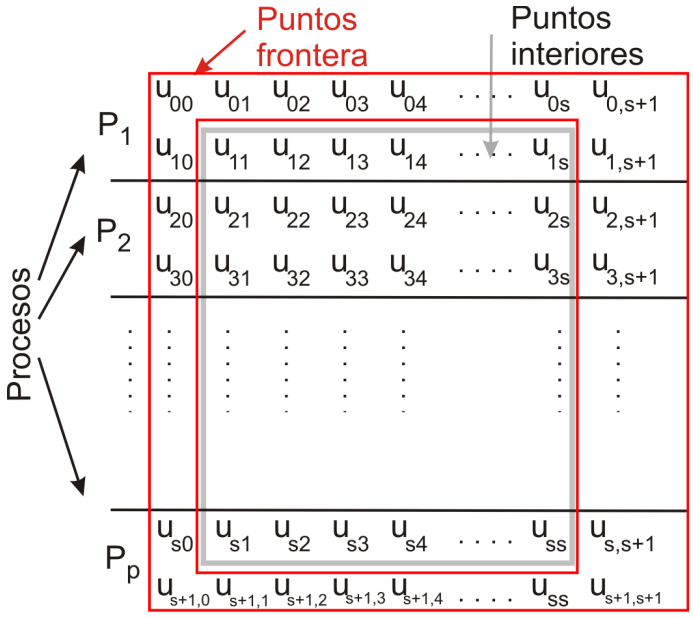


Considerando la discretización del dominio, el método iterativo de Jacobi resulta ser

$$u_{ij}^{(k+1)} = \frac{1}{4}\left[u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}\right],$$

$$i, j = 1, 2, \ldots, s, \;\; k = 0, 1, 2, \ldots.$$

En la implementación paralela consideraremos una descomposición en dominios útil para una topología de anillo bidireccional de manera que cada proceso se encargará del cálculo de una porción de filas consecutivas de las variables u_{ij}, tal y como indica la figura siguiente:

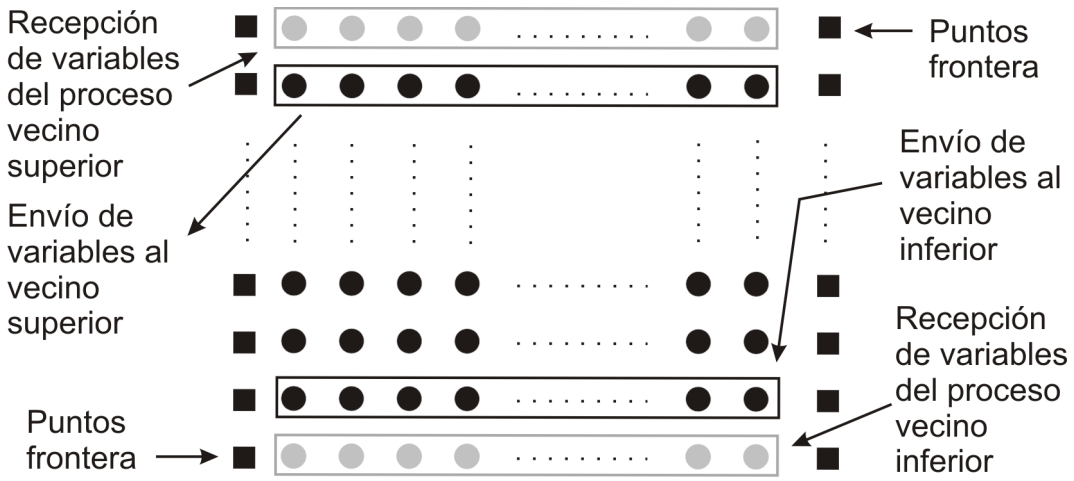


Se considerará una distribución equitativa de todas las filas entre los procesos. Si el número de procesos no divide al número de filas, el exceso de filas se le asociará al proceso número 0 (proceso root) (ver ejemplo psdot.c)

En la siguiente figura se muestra el esquema típico de cálculo y comunicación para un proceso cualquiera (intermedio), suponiendo que todas las variables se almacenan en un array dos dimensional `x[0:slocal+1][0:s+1]`.

- Puntos frontera: marcados con un cuadrado negro. Almacenados en la columnas `0` y `s+1` de `x`, es decir, en las posiciones `x[0:slocal+1][0]` y `x[0:slocal+1][s+1]`.

- Cada proceso debe calcular un total de `slocal` $\times$ `s` variables; las indicadas con un círculo negro. Corresponden a las posiciones `[1:slocal][1:s]` del array `x`
- Cada proceso P_i debe enviar las filas `1` y `slocal` del array `x` a los procesos P_{i-1} y P_{i+1}, respectivamente.
- Cada proceso P_i debe recibir de P_{i-1} y P_{i+1} en las filas `0` y `slocal+1` del array `xold`.



El programa debe aceptar como datos (ver la versión secuencial):

1. El número de nodos interiores por fila (o columna): `s`.
2. El número de iteraciones máximas a realizar: `IterMax`.
3. En su caso, el número de iteraciones internas: `inner`.

El criterio de parada que debe utilizarse en cada iteración será:

$$\left(\sum_{i,j=1}^{s}(x(i,j)-xold(i,j))^2\right)^{\frac{1}{2}} < \text{cota}, \tag{2}$$

o que el número de iteraciones sea mayor o igual que `IterMax`.

Cada proceso $P_i,\ i = 1, 2, \ldots, nproc$, debe calcular su norma parcial, es decir, calculará:

$$\sum_{i,j}(x(i,j)-xold(i,j))^2.$$

Esta norma parcial será sumada utilizando `MPI_Allreduce()`, para posteriormente calcular su raíz cuadrada (`sqrt`). Se comprobará el criterio de parada (2). En caso de que no haya convergencia, el proceso iterativo continuará.

La aproximación inicial con la que empezar debe ser:

$$(1, 1, \ldots, 1)^t$$

Una vez alcanzada la convergencia debe escribirse la solución junto con los valores frontera, es decir, `x[0:s+1][0:s+1]`, en un archivo `res.m` con el formato siguiente y recorriendo el array por filas:

$$
\begin{array}{l}
mx = valor.s; \\
sol = [\\
x[0][0] \;\; x[0][1] \;\; \cdots \;\; x[0][s+1] \\
x[1][0] \;\; x[1][1] \;\; \cdots \;\; x[1][s+1] \\
\vdots \\
x[s+1][0] \;\; x[s+1][1] \;\; \cdots \;\; x[s+1][s+1] \\
];
\end{array}
$$

Esta solución será utilizada por un programa MatLAB que visualice la solución obtenida.

Una vez que el programa funcione correctamente, modificarlo de manera que la fase de actualización se realice un número predeterminado de veces (`inner` veces) antes de comunicar (ver código).

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

- Explicación de la implementación.
- Distintas ejecuciones indicando para cada una de ellas, el número de iteraciones necesarias para convergencia, el valor de la norma para el cual se paró y, para alguna de ellas, la imagen correspondiente a la solución.
- Se deben presentar gráficas de tiempos comparando ejecuciones con `inner=1` e `inner>1`. Indicar qué conclusiones se extraen de estas gráficas.

Las distintas ejecuciones deben corresponder, como mínimo, a los siguientes datos:

`s`	`inner`	Número de procesos
50	1	3
50	10	3
50	15	3
50	50	3
50	100	3
50	150	3
100	1	4
100	10	4
100	15	4
100	50	4
100	75	4
100	100	4
100	150	4

Ficheros a entregar:

jac.c Contendrá la unidad principal y todas las funciones auxiliares.

makefile Makefile utilizado para la compilación.

Unidad jac

- Proceso 0: Lectura datos iniciales (`s` e `IterMax`).
- Envío y recepción datos iniciales comunes a todos.
- Comprobación datos correctos.
- Cálculo de la carga de trabajo por proceso (`slocal`).
- Inicializar `x`, `xold` (mediante la llamada a `inicializar`), `norma` e `iter`.
- Proceso iterativo (`while ((norma >cota) && (iter <IterMax))`)
  - Copiar porción de `x` en `xold`.
  - Actualizar `x` a partir de `xold`.
  - Calcular la porción de la norma correspondiente, entre `x` y `xold`.
  - Llamar a `MPI_Allreduce` para poder obtener el valor final de la norma.
  - Comunicaciones entre vecinos. Notar que todo proceso debe recibir dos mensajes salvo el primero y el último que sólo reciben un mensaje.
- Convergencia alcanzada.
- Escritura de resultados:
  - El proceso 0 escribe su resultado: `x[0:slocal][0:s+1]`.
  - El proceso 0 recibe del 1, sobreescribe en `x[1:slocalhijos][0:s+1]` y escribe resultado.
  - El proceso 0 recibe del 2 (suponiendo que no es el último), sobreescribe en `x[1:slocalhijos][0:s+1]` y escribe resultado.
  - ...
  - El proceso 0 recibe del último, sobreescribe en `x[1:slocalhijo+1][0:s+1]` y escribe resultado.

Unidad jac con `inner` iteraciones

- Lectura datos iniciales (`s, IterMax` e `inner`).
- Comprobación datos correctos.
- Envío y recepción datos iniciales comunes a todos.
- Cálculo de la carga de trabajo por proceso (`slocal`).
- Inicializar `x, xant` (mediante la llamada a `inicializar`), `norma` e `iter`.
- Proceso iterativo (`while ((norma >cota) && (iter <IterMax))`)
  - Copiar porción de `x` en `xold`.
  - Copiar porción de `x` en `xant`.
  - Actualizar `x` a partir de `xant`.

  } `inner` iteraciones
  - Calcular la porción de la norma correspondiente, entre `x` y `xold`.
  - Llamar a `MPI_Allreduce` para poder obtener el valor final de la norma.
  - Comunicaciones entre vecinos. Notar que todo proceso debe recibir dos mensajes salvo el primero y el último que sólo reciben un mensaje.
- Convergencia alcanzada.
- Escritura de resultados:
  - El proceso 0 escribe su resultado: `x[0:slocal][0:s+1]`.
  - El proceso 0 recibe del 1, sobreescribe en `x[1:slocalhijos][0:s+1]` y escribe resultado.
  - El proceso 0 recibe del 2 (suponiendo que no es el último), sobreescribe en `x[1:slocalhijos][0:s+1]` y escribe resultado.
  - ...
  - El proceso 0 recibe del último, sobreescribe en `x[1:slocalhijo+1][0:s+1]` y escribe resultado.