UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Computación Paralela (CPA)

Práctica 3: Paralelización con MPI

Curso 2021/22

Índice

Introducción

Esta práctica consta de 4 sesiones, correspondiendo cada una de ellas a un apartado de este documento. La siguiente tabla muestra el material de partida para realizar cada uno de los apartados:

Sesión 1	Cálculo de Pi	`mpi_pi.c`, `ping-pong.c`
Sesión 2	Fractales	`newton.c`
Sesión 3	Producto matriz-vector	`mxv1.c`, `mxv2.c`
Sesión 4	Sistemas de ecuaciones lineales	`sistbf.c`

Igual que se hizo en las prácticas 1 y 2, crea una carpeta en `W` para la práctica. Por ejemplo, `W/cpa/prac3`. A continuación, entra en `kahan` y crea también una carpeta para la práctica, directamente en el home:

```
$ ssh -Y -l usuario@alumno.upv.es kahan.dsic.upv.es
$ mkdir prac3
```

La opción `-Y` es opcional. Sirve para poder ver en nuestra pantalla cualquier programa gráfico ejecutado en `kahan` (por ejemplo, el programa `display` para visualizar una imagen).

1. Primeros pasos con MPI

El objetivo de la primera sesión de la práctica es familiarizarse con la compilación y ejecución de programas MPI sencillos.

1.1. Hola mundo

Empezamos con el típico programa "hola mundo" en el que cada proceso imprime un mensaje por la salida estándar. El código de la Figura 1 muestra un programa mínimo, con inicialización y finalización de MPI, y un `printf` mostrando un mensaje.

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

int main (int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello world\n");
  MPI_Finalize();
  return 0;
}
```

Figura 1: Programa "hola mundo" en MPI.

Copia el programa en un fichero, por ejemplo `hello.c`, en la carpeta de la práctica (`W/cpa/prac3`). A continuación, compílalo y ejecútalo. Para compilar el programa, la máquina tiene que tener MPI instalado, por lo que lo haremos desde `kahan` (también es posible hacerlo desde cualquier ordenador donde hayamos instalado MPI). La compilación se hace igual que en las prácticas anteriores, pero usando el comando `mpicc` en vez de `gcc` (`mpicc` es una utilidad que simplemente invoca al compilador, en este caso `gcc`, con las opciones apropiadas para la instalación particular de MPI – con "`mpicc -show`" se muestran dichas opciones). Por ejemplo:

```
$ cd ~/prac3
$ mpicc -Wall -o hello ~/W/cpa/prac3/hello.c
```

Dado que es un programa cuya ejecución es muy corta, podemos lanzarlo directamente en el nodo *front-end* de `kahan`. Para ello hay que usar la orden `mpiexec` indicando mediante la opción `-n` el número de procesos que queremos ejecutar. Por ejemplo:

```
$ mpiexec -n 4 hello
```

Todos los procesos se lanzarán sobre el mismo nodo.

Esa forma de ejecutar puede servir para pruebas rápidas, pero por supuesto normalmente las ejecuciones habrá que lanzarlas a través del sistema de colas de `kahan`, de manera que el programa se ejecute en los nodos de cálculo (no en el nodo *front-end*). Para ello, ya sabes que debes crear un fichero de trabajo y utilizar `sbatch` para lanzar el trabajo.

```
#!/bin/sh
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --time=5:00
#SBATCH --partition=cpa
scontrol show hostnames $SLURM_JOB_NODELIST

mpiexec ./hello
```

Figura 2: Fichero de trabajo para ejecutar en el sistema de colas.

En la Figura 2 hay un ejemplo de un fichero de trabajo en el cual se reservan 2 nodos. Hasta ahora nuestros trabajos solo podían usar 1 nodo, puesto que usaban memoria compartida, pero con MPI podemos usar varios nodos sin problema. El trabajo del ejemplo ejecuta el programa `hello` mediante `mpiexec`, pero en este caso el número de procesos se indica mediante la opción `ntasks` del trabajo. En el ejemplo, se ejecutan 2 procesos y se usan 2 nodos, por lo que habrá un proceso en cada nodo.

Dado que un nodo del clúster tiene múltiples cores, puede ser útil lanzar varios procesos en el mismo nodo, en vez de un solo proceso por nodo. Para ello, habrá que usar un valor de `ntasks` mayor que el número de nodos. Por ejemplo, con `ntasks=4` y `nodes=2`, tendríamos 2 procesos por nodo.

La orden "`scontrol show hostnames...`" no es realmente necesaria; sirve para que sepamos qué nodos del clúster han sido reservados para nuestra ejecución.

Importante: ten en cuenta que `kahan` tiene solo 4 nodos, por lo que se recomienda usar **solo 1 o 2 nodos** en cada trabajo, de forma que todos los usuarios puedan trabajar.

Ejercicio 1: Realiza diversas ejecuciones del programa usando el sistema de colas, con diferente número de procesos y nodos.

Ejercicio 2: Modifica el programa para obtener el identificador de proceso y el número de procesos, y mostrar esa información en el saludo. Recuerda que para ello debes usar las funciones `MPI_Comm_rank` y `MPI_Comm_size`.

1.2. Cálculo de Pi

Vamos ahora a trabajar con un programa MPI que realiza un cálculo en paralelo. En particular, vamos a calcular una aproximación del valor de π. El valor de π se puede calcular de muchas formas, una de ellas es mediante la integral definida

$$\int_0^1 \frac{1}{1+x^2} dx = \frac{\pi}{4} .$$

El programa `mpi_pi.c` calcula esta integral mediante el método de los rectángulos que vimos en la primera práctica. El intervalo $[0, 1]$ se descompone en n subintervalos (rectángulos) y cada uno de los p procesos realiza el cálculo asociado a n/p rectángulos. Más concretamente, la sentencia `for`:

```
for (i = myid + 1; i <= n; i += numprocs) {
```

es el resultado de paralelizar un bucle que recorre los n rectángulos:

```
for (i = 1; i <= n; i++) {
```

repartiendo las iteraciones de forma cíclica entre los procesos. Si bien en OpenMP existe una directiva que reparte las iteraciones de un bucle de forma automática (`#pragma omp for`), en MPI no hay tal construcción, por lo que el reparto debe hacerse de forma explícita.

Mediante el bucle anterior cada proceso calcula una suma parcial y únicamente falta acumular esas sumas en el resultado final. Eso se hace mediante la operación de comunicación colectiva `MPI_Reduce`:

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Por medio de esta función, los valores de `mypi` de cada uno de los procesos se suman, y el resultado se almacena en la variable `pi` del proceso 0. Observa que todos los procesos realizan la llamada a la función, dado que todos deben cooperar aportando el valor de su variable `mypi`. Sin embargo, el resultado solo se obtiene en el proceso 0.

Ejercicio 3: Modifica el programa, sustituyendo la llamada a la función `MPI_Reduce` por un fragmento de código que sea equivalente pero utilice solo comunicaciones punto a punto (`MPI_Send` y `MPI_Recv`). Para ello, lo más sencillo es hacer que todos los procesos envíen su suma parcial (`mypi`) al proceso 0, el cual se encargará de recibir cada valor y acumularlo sobre la variable `pi`.

1.3. El programa *ping-pong*

Los programas MPI utilizan la red Ethernet rápida de 25 Gb disponible en el clúster de prácticas. Esta red tiene mejores prestaciones que una red Ethernet convencional, tanto en ancho de banda como en tiempo de establecimiento. Aunque se pueden consultar las especificaciones técnicas de la red, es habitual realizar un estudio experimental para obtener los parámetros de la red a partir de la ejecución de un programa real.

Esto puede hacerse mediante un programa de tipo *ping-pong*, el cual consta de dos procesos P_0 y P_1, de manera que P_0 envía un mensaje a P_1, y este le devuelve el mensaje inmediatamente después. P_0 mide el tiempo transcurrido desde la operación de envío hasta que termina la recepción de la respuesta.

Ejercicio 4: Completa el programa `ping-pong.c` para que haga lo que se explica en el párrafo anterior, teniendo en cuenta lo siguiente:

- El programa tiene como argumento el tamaño del mensaje, n (en bytes).
- Usa la función `MPI_Wtime()` para medir tiempos. Se usa exactamente igual que la función de OpenMP `omp_get_wtime()`.
- Para que los tiempos medidos sean significativos, el programa debe repetir la operación cierto número de veces (`NREPS`) y mostrar el tiempo medio.
- Utiliza las primitivas estándar para envío y recepción de mensajes: `MPI_Send` y `MPI_Recv`, indicando `MPI_BYTE` como tipo de los datos a enviar/recibir.

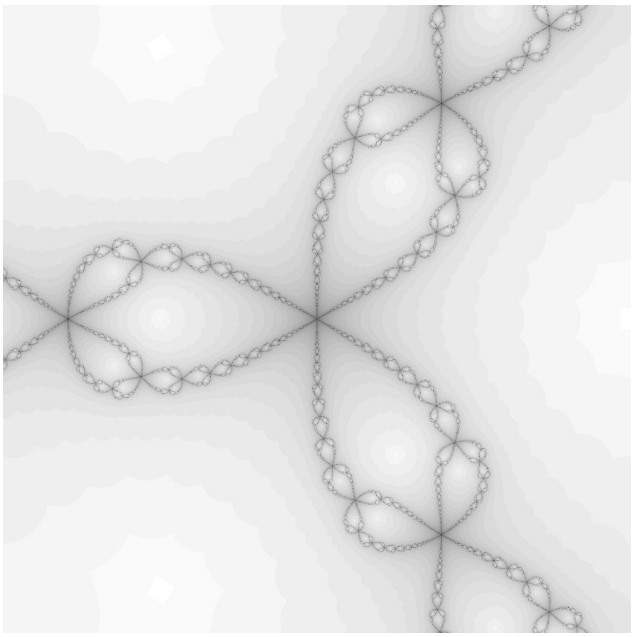
Ejercicio 5: ¿Por qué se envían dos mensajes en cada iteración del bucle? ¿se podría eliminar el mensaje de respuesta de P_1 a P_0?

Ejercicio 6: En cada iteración, el proceso P_0 tiene que hacer un envío y una recepción. ¿Podría utilizar para ello la función `MPI_Sendrecv_replace`? ¿Y el proceso P_1?

2. Fractales de Newton

Un fractal es un objeto geométrico cuya estructura básica se repite a diferentes escalas. En determinados casos, la representación de un fractal conlleva un coste computacional considerable. En esta práctica se va a trabajar con un programa que genera fractales de Newton.

Se proporciona el código fuente de un programa paralelo que utiliza el esquema maestro-trabajador y la librería de comunicaciones MPI para calcular diferentes fractales de Newton: `newton.c`. El objetivo de la práctica es modificar las comunicaciones de este programa, pasando a usar comunicaciones no bloqueantes, para lograr que el proceso maestro también procese parte de la imagen.



```
Para y = y1, ..., y2
  Para x = x1, ..., x2
    col = num_iteraciones_Newton(función,x,y)
    Pintar el punto (x,y) con el color col
  Fin_para
Fin_para
```

Figura 3: Fractal de Newton de la función $f(z) = z^3 - 1$ (izquierda) y algoritmo utilizado para su generación (derecha). Nota: La imagen del fractal se muestra invertida (se ve su negativo).

2.1. Algoritmo secuencial

En los fractales de Newton se trabaja buscando los ceros de funciones complejas de variable compleja. Dichos ceros se calculan utilizando el método de Newton de búsqueda de raíces, el cual realiza más o menos iteraciones dependiendo del valor inicial. Dada una función compleja, se selecciona una zona del plano complejo en la que dibujar el fractal y se busca algún cero de la función partiendo de todos los puntos en esa zona. El número de iteraciones necesarias para alcanzar una solución desde cada punto dictamina el color con el que se pintará ese punto en el fractal.

Por ejemplo, en la Figura 3 (izquierda) puede verse el fractal de Newton para la función $f(z) = z^3 - 1$ en la zona con x e y entre -1 y 1. Un algoritmo en pseudo-código para el dibujo de un fractal de Newton se muestra en la Figura 3 (derecha).

2.2. Algoritmo maestro-trabajadores clásico

El programa proporcionado (`newton.c`) es una implementación paralela siguiendo el esquema maestro-trabajadores para generar fractales de Newton.

Este programa permite calcular fractales de Newton de diferentes funciones. Tiene múltiples opciones que afectan al fractal generado (pueden verse directamente en el código fuente). Cabe destacar la opción `-c` a la que se le proporciona el número de función a utilizar para el fractal. Actualmente hay 4 funciones, identificadas de 1 a 4, aunque el parámetro permite indicar también el caso 5. El caso 5 en realidad es una ampliación de una zona blanca del caso 4. No es "muy bonito", pero tiene un coste mayor y es más útil para ver la ganancia en paralelo. Se aconseja usar los casos 1-4 para comprobar (viendo la imagen generada) que el programa funciona bien en paralelo y luego el caso 5 para hacer medida de prestaciones.

El programa almacena el fractal calculado en cada ocasión en el fichero `newton.pgm` (si no se cambia usando la opción `-o`). Por defecto genera una imagen en escala de grises donde el color blanco se corresponde con el mayor número de iteraciones necesitado en un punto de la imagen.

Ejercicio 1: Compila y ejecuta el programa usando el caso 1. Haz una copia de la imagen obtenida (`newton.pgm`) con el nombre `ref.pgm`. Este fichero te servirá para comprobar que la modificación que realices

```
Si yo=0 entonces (soy el maestro)

  siguiente_fila <- 0
  Para proc = 1 ... np-1
    envía al proceso proc petición de hacer la fila número siguiente_fila
    siguiente_fila <- siguiente_fila + 1
  Fin_para

  filas_hechas <- 0
  Mientras filas_hechas < filas_totales
    recibe de cualquier proceso una fila calculada
    proc <- proceso que ha enviado el mensaje
    num_fila <- número de fila
    envía al proceso proc petición de hacer la fila número siguiente_fila
    siguiente_fila <- siguiente_fila + 1
    copia fila calculada a su sitio, que es la fila num_fila de la imagen
    filas_hechas <- filas_hechas + 1
  Fin_mientras

si_no (soy un trabajador)

  recibe número de fila a hacer en num_fila
  Mientras num_fila < filas_totales
    procesa la fila número num_fila
    envía la fila recién calculada al maestro
    recibe número de fila a hacer en num_fila
  Fin_mientras

Fin_si
```

Figura 4: Pseudo-código del algoritmo maestro-trabajadores.

al programa funciona correctamente.

En el programa `newton.c`, la función `fractal_newton` es la encargada de realizar el algoritmo de fractales de Newton proporcionado anteriormente (Figura 3).

Esta función dibuja en la matriz A de $w \times h$ píxeles un fractal de Newton para el rectángulo con esquinas (x_1, y_1) y (x_2, y_2). Se buscan ceros de la función con una tolerancia dada por `tol` y con un número máximo de iteraciones dado por `maxiter`. Además, se calcula y se devuelve el número máximo de iteraciones que aparece en la imagen, que se utilizará para el color blanco.

El código proporcionado ya implementa una versión paralela del algoritmo, siguiendo el esquema tradicional maestro-trabajadores. El pseudo-código correspondiente se muestra en la Figura 4. Recuerda que en el esquema clásico maestro-trabajadores uno de los procesos actúa como maestro y va encargando trabajo a los demás, los trabajadores, que devolverán al maestro los resultados de cada trabajo realizado.

Es conveniente que estudies y entiendas el código básico maestro-trabajadores que se ha utilizado en el código proporcionado.

Ejercicio 2: Repasa el algoritmo utilizado (Figura 4) y su implementación en lenguaje C realizada en la función `fractal_newton` del programa proporcionado. Responde a las siguientes preguntas:

- Cuando un proceso trabajador le envía una fila al proceso maestro, ¿cómo le indica de qué fila se trata?
- ¿Cómo sabe un proceso trabajador que ya no tiene que procesar más filas?

```
siguiente_fila <- 0
Para proc = 1 ... np-1
  envía al proceso proc petición de hacer la fila número siguiente_fila
  siguiente_fila <- siguiente_fila + 1
Fin_para

filas_hechas <- 0
Mientras filas_hechas < filas_totales
  inicia la recepción no bloqueante de una fila calculada de cualquier proceso
  Mientras no se ha recibido nada y siguiente_fila < filas_totales
    procesa la fila número siguiente_fila
    siguiente_fila <- siguiente_fila + 1
    filas_hechas <- filas_hechas + 1
  Fin_mientras
  Si no se ha recibido nada entonces
    espera (de forma bloqueante) a recibir algo
  Fin_si
  proc <- proceso que ha enviado el mensaje
  num_fila <- número de fila
  envía al proceso proc petición de hacer la fila número siguiente_fila
  siguiente_fila <- siguiente_fila + 1
  copia fila calculada a su sitio, que es la fila num_fila de la imagen
  filas_hechas <- filas_hechas + 1
Fin_mientras
```

Figura 5: Pseudo-código del maestro haciendo que también trabaje (el código de los trabajadores no cambia).

2.3. Algoritmo maestro-trabajadores con el maestro trabajando

En el algoritmo maestro-trabajadores clásico, el maestro va mandando trabajos que hacer a los trabajadores y recogiendo los resultados, pero él no realiza ningún trabajo. Si ejecutamos reservando un procesador para este maestro, estaremos desaprovechando la potencia de cálculo de este procesador.

Una forma de no desaprovechar esta potencia extra es haciendo que el proceso maestro también realice trabajos. Para ello, hay que hacer que sus comunicaciones sean no bloqueantes. De esta manera, en lugar de quedarse bloqueado esperando la respuesta de algún trabajador, podrá ir trabajando al mismo tiempo que espera esta respuesta.

En la Figura 5 tienes un algoritmo en pseudo-código para que el maestro se comporte de esta manera. El código de los trabajadores no cambia.

Ejercicio 3: Lee y entiende el algoritmo mostrado en la Figura 5 e impleméntalo sobre una copia del programa original, para luego poder comparar ambos programas.

En la implementación, ten en cuenta:

- Cuando el proceso maestro recibe una fila, la guarda en el array `B`. Mientras espera a recibirla, el maestro podrá procesar filas también, pero no debería modificar dicho array. En su lugar, puedes hacer que modifique directamente el array `A`.
- El array `A` se usa en el proceso maestro para guardar la imagen entera. Es un array unidimensional, pero hay una macro que permite usarlo como una matriz:

  ```
  #define A(i, j) A[(i)*w + (j)]
  ```

  de manera que puedes escribir `A(i,j)` para referirte al elemento de la fila `i`, columna `j`.

```
Inicializar M,x,v
Para iter=1,2,...num_iter
   x = M*x+v
Fin_para
norma = suma de los valores absolutos de x
mostrar norma
```

Figura 6: Algoritmo iterativo básico de `mxv1.c` y `mxv2.c`.

Comprueba que la nueva versión del programa es correcta. Para ello, ejecútala con el caso 1 y comprueba que la imagen obtenida es la misma que guardaste previamente en el fichero `ref.pgm` (usa el comando `cmp` para comparar los ficheros).

Ejercicio 4: Utiliza algún fractal más costoso (el caso 5, por ejemplo) para sacar medida de prestaciones comparando los resultados de ambos programas. Ejecuta ambos programas, por ejemplo con 4 y con 8 procesos, y comprueba cuál de las dos versiones funciona mejor.

3. Producto matriz-vector

En esta sesión vamos a trabajar con operaciones de comunicación colectiva de MPI. Para ello, utilizaremos un núcleo computacional muy utilizado: el producto de una matriz por un vector.

3.1. Descripción del problema

Muchos problemas en computación numérica permiten su resolución mediante lo que se conoce como métodos iterativos. En estos métodos, se parte de una aproximación a la solución y mediante alguna operación que se repite durante múltiples iteraciones se van obteniendo sucesivas aproximaciones de la solución, que bajo unas determinadas condiciones van acercándose cada vez más a la solución buscada.

Vamos a trabajar con un método iterativo basado en la expresión:

$$x^{k+1} = Mx^k + v,$$

según la cual en cada iteración (k indica la iteración actual), partimos de una aproximación a la solución (vector x^k) y calculamos una nueva aproximación (vector x^{k+1}). M y v son una matriz y un vector conocidos. Cada nueva aproximación se calcula mediante un producto matriz-vector y una suma vectorial, a partir de la aproximación anterior.

Esta fórmula iterativa es muy utilizada para la resolución de sistemas de ecuaciones lineales. Lo habitual es hacer tantas iteraciones como sean necesarias hasta garantizar que la aproximación obtenida esté suficientemente cerca de la solución. Sin embargo, por simplicidad, en nuestro caso vamos a realizar un número fijo de iteraciones.

Partimos de dos implementaciones paralelas (`mxv1.c` y `mxv2.c`) del mismo algoritmo básico que aparece en la Figura 6. En este código trabajamos con una matriz M y un vector v aleatorios, pero construidos de tal forma que el sistema converja (vaya acercándose a la solución). Al final de todo, el programa muestra la 1-norma del vector x resultado (la suma de los elementos de x, en valor absoluto). Este valor se puede utilizar a modo de *hash* para comprobar que diferentes ejecuciones son correctas. (El resultado de las dos versiones difiere, porque cada una trabaja con M y v diferentes.)

Los dos programas proporcionados difieren en la forma de almacenar y repartir los datos entre los distintos procesos.

- En `mxv1.c` almacenamos la matriz M por filas y la repartimos por bloques de filas entre los procesos.
- En `mxv2.c` la matriz se almacena por columnas y se reparte por bloques de columnas.

Estas formas diferentes de repartir la matriz hacen que las comunicaciones para efectuar los cálculos sean distintas en cada uno de los casos.

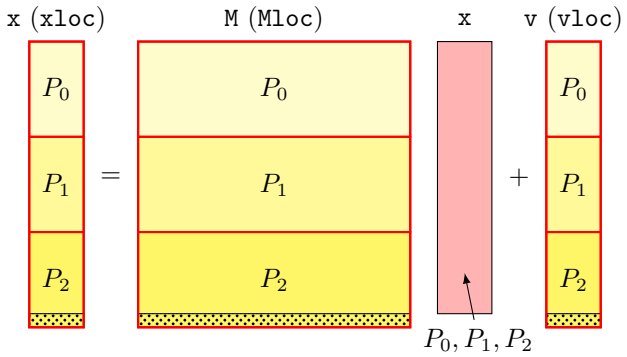


Figura 7: Distribución de datos en `mxv1.c`.

3.2. Programa con distribución por bloques de filas (`mxv1`)

Ejercicio 1: Compila el programa `mxv1.c` y ejecútalo. Puedes hacer alguna ejecución con un tamaño muy pequeño directamente en el *front-end*, por ejemplo:

```
$ mpiexec -n 3 mxv1 -n 5 -i 5
```

La orden anterior ejecuta el programa con 3 procesos, con un tamaño de matriz de 5 filas y columnas (`-n 5`) y 5 iteraciones (`-i 5`). Para ejecuciones más largas se debe utilizar el sistema de colas.

En la figura 7 se ilustra la forma de repartir los datos en el programa `mxv1.c` suponiendo 3 procesos. La matriz M tiene n filas y n columnas, los vectores x y v tienen n elementos. Cada proceso almacena en `Mloc` su parte local de M, en `vloc` su parte local de v y en `x` el vector x actual entero. Usando esos datos, cada proceso calcula en `xloc` su bloque del nuevo vector x.

La dimensión n puede no ser divisible entre el número de procesos, por lo que el número de filas de M, v y el nuevo x se expande (hasta `n2` filas) para que todos los procesos tengan el mismo número de filas (`mb`). Esas filas extra (zona punteada en la Figura 7) facilitan la distribución de los datos, pero no se tienen en cuenta al hacer el producto matriz-vector.

Ejercicio 2: Analiza el código de `mxv1.c` para entender cómo se realiza el producto matriz-vector y la suma de vectores.

Utilización de comunicaciones colectivas

Las versiones proporcionadas del algoritmo paralelo para el método iterativo han sido desarrolladas usando solamente operaciones de comunicación punto a punto. Sin embargo, como el alumno ya sabe, en MPI hay muchas funciones de comunicación colectiva que facilitan la programación cuando se requieren este tipo de comunicaciones.

Ejercicio 3: Modifica el programa `mxv1.c` para sustituir las comunicaciones punto a punto por comunicaciones colectivas, en aquellos puntos del programa donde sea posible. En el código estos puntos están marcados con un comentario previo con la palabra `COMMUNICATIONS`.

Se aconseja no esperar a tener todas las comunicaciones cambiadas para probar el programa. Es conveniente ir probando el programa tras cambiar cada comunicación y así asegurarse de que no se ha alterado el resultado con el cambio.

Aunque aquí sustituimos operaciones punto a punto por operaciones colectivas, lo normal es escribir el programa desde el principio utilizando operaciones colectivas. Además de facilitar la programación, las operaciones colectivas de MPI pueden estar optimizadas para realizar las comunicaciones de forma eficiente.

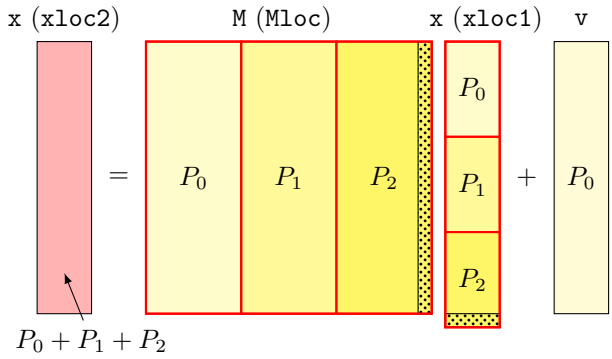


Figura 8: Distribución de datos en `mxv2.c`

3.3. Programa con distribución por bloques de columnas (`mxv2`)

En el programa `mxv2.c`, la matriz M se almacena por columnas y se distribuye por bloques de columnas, como se puede ver en la Figura 8 para el caso de 3 procesos. Cada proceso almacena en `Mloc` su parte local de M y en `xloc1` su parte local del vector x actual. Usando esos datos, cada proceso calcula en `xloc2` su contribución local al nuevo vector x. Dicho vector se obtiene a continuación sumando los vectores `xloc2` de todos los procesos, más el vector v (que está en P_0).

El número de columnas de M y el de filas del vector x actual se expande (hasta `n2` columnas/filas) para que todos los procesos tengan el mismo número de columnas de M (`nb`). Esas columnas/filas extra (zona punteada en la Figura 8) facilitan la distribución de los datos, pero no se tienen en cuenta al hacer el producto matriz-vector.

Ejercicio 4: Modifica el programa `mxv2.c` para sustituir las comunicaciones punto a punto por comunicaciones colectivas, en aquellos puntos del programa donde sea posible. En el código estos puntos están marcados con un comentario previo con la palabra `COMMUNICATIONS`.

Una vez que se tengan las versiones modificades de `mxv1.c` y `mxv2.c`, es interesante comparar los tiempos respecto a las versiones originales. Aunque es probable que no se encuentren muchas diferencias en este problema en concreto.

4. Sistemas de ecuaciones lineales

Esta última sesión se centra en la implementación mediante MPI de un algoritmo paralelo de mayor complejidad, concretamente la resolución de sistemas de ecuaciones lineales. El objetivo principal de la práctica es manejar matrices distribuidas, tanto por bloques como de forma cíclica.

El material de partida para realizar la práctica consiste en el fichero `sistbf.c`, que implementa una versión paralela donde la matriz del problema se distribuye por bloques de filas. Deberá modificarse para que utilice en su lugar una **distribución cíclica por filas**.

Se sugiere que hagas una copia del fichero `sistbf.c`, por ejemplo con el nombre `sistcf.c`, que contendrá la implementación cíclica por filas.

4.1. Resolución de sistemas de ecuaciones lineales

Pretendemos resolver un sistema de ecuaciones lineales, descrito en notación matricial como $Ax = b$, donde A es una matriz, b es el vector "parte derecha" (o de términos independientes), y x es el vector solución. Nos

centramos en el caso en que A es cuadrada, y denotamos por n su dimensión:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}, \qquad x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}, \qquad b = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}. \tag{1}$$

Así, por ejemplo, la primera ecuación sería

$$a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} = b_0. \tag{2}$$

Se quiere calcular el vector x que satisface las n ecuaciones simultáneamente. El sistema tiene solución única (es compatible y determinado) cuando la matriz A tiene un determinante distinto de cero. Existen diversas técnicas directas e iterativas para la resolución de un sistema de ecuaciones, destacando entre las técnicas directas la factorización LU, que es la utilizada en nuestro programa.

La factorización LU consiste en la obtención de un par de matrices, una triangular inferior unidad (todos los elementos por encima de la diagonal principal a cero y la propia diagonal principal a uno) y otra triangular superior (todos los elementos por debajo de la diagonal principal a cero), ambas de la misma dimensión que la matriz de partida, tales que su producto es igual a A.

De esta forma, la resolución del sistema de ecuaciones equivale a la resolución de dos sistemas triangulares:

$$\left.\begin{array}{c} A = LU \\ Ax = b \end{array}\right\} \quad \longrightarrow \quad LUx = b \quad \longrightarrow \quad \left\{\begin{array}{l} Ly = b \\ Ux = y \end{array}\right. . \tag{3}$$

El cálculo de la descomposición LU se puede realizar mediante la eliminación Gaussiana, utilizando los elementos diagonales de la matriz como pivotes y haciendo ceros por debajo de la diagonal en cada columna.

Los sistemas triangulares pueden resolverse aplicando los algoritmos de eliminación progresiva (para el caso de la matriz triangular inferior unidad) y sustitución regresiva (para el caso de la matriz triangular superior).

4.2. Programa paralelo proporcionado

El programa proporcionado (fichero `sistbf.c`) genera un sistema lineal $Ax = b$ y lo resuelve, realizando para ello los siguientes pasos, marcados en el código con comentarios con el texto "STEP":

1. **Generar los datos**. El proceso 0 genera la matriz (`A`) y el vector (`b`) completos. Todos los procesos (incluido el 0) reservan memoria para su matriz local (`Aloc`).

2. **Distribuir los datos**. La matriz se distribuye entre los procesos por bloques de `mb` filas consecutivas. El vector `b` se replica en todos los procesos.

3. **Descomposición LU**. En esta fase la matriz `A` se sobreescribe por L y U. Los elementos de L quedan en la parte inferior estricta de `A` y los de U en la parte superior.

4. **Resolver el sistema triangular inferior** $Ly = b$. El vector y se almacena sobre `b`, sobreescribiéndolo.

5. **Resolver el sistema triangular superior** $Ux = y$. El vector y se encuentra almacenado en la variable `b`, y en esta fase se sobreescribe con el vector x.

Ejercicio 1: Compila y ejecuta el programa. Se pueden ejecutar pruebas cortas directamente en el *front-end* de `kahan`. Por ejemplo, para resolver un sistema de 5 ecuaciones usando 3 procesos:

```
$ mpiexec -n 3 sistbf 5
```

En este caso el sistema que se resuelve es:

$$\begin{bmatrix} 25 & 4 & 3 & 2 & 1 \\ 4 & 25 & 4 & 3 & 2 \\ 3 & 4 & 25 & 4 & 3 \\ 2 & 3 & 4 & 25 & 4 \\ 1 & 2 & 3 & 4 & 25 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 35 \\ 38 \\ 39 \\ 38 \\ 35 \end{bmatrix}.$$

En la terminal aparecerá el contenido de la matriz `A` y el vector `b` en cada proceso en distintos puntos del programa. Por ejemplo, tras realizar el reparto inicial, tenemos:

```
Matrix A:
---- proc. 0 ----
 25.000   4.000   3.000   2.000   1.000
  4.000  25.000   4.000   3.000   2.000
---- proc. 1 ----
  3.000   4.000  25.000   4.000   3.000
  2.000   3.000   4.000  25.000   4.000
---- proc. 2 ----
  1.000   2.000   3.000   4.000  25.000

Vector b:
---- proc. 0 ----
 35.000  38.000  39.000  38.000  35.000
---- proc. 1 ----
 35.000  38.000  39.000  38.000  35.000
---- proc. 2 ----
 35.000  38.000  39.000  38.000  35.000
```

Al final, el vector `b` contendrá la solución del sistema, que es un vector de unos. El sistema generado está preparado para que, independientemente del tamaño del sistema, la solución sea siempre un vector de unos. Finalmente, el programa informa del error de la solución obtenida, que debería ser cero.

Se debe cambiar el código del programa para que utilice una distribución cíclica por filas, en vez de una distribución por bloques de filas. Esto requiere introducir cambios por una parte en el paso 2 (distribución de datos), y por otra parte en los pasos 3, 4 y 5 (descomposición LU y sistemas triangulares). A continuación se dan más detalles sobre estos pasos.

4.3. Fase de distribución de datos

Como hemos dicho, en esta fase (paso 2) se realiza una distribución de la matriz por bloques de filas consecutivas. Para ello se utiliza la función `MPI_Scatter`.

Ejercicio 2: Cambia la forma de distribuir la matriz, para que se haga cíclicamente por filas. Hay distintas formas de implementar este reparto, una de las cuales consiste en hacer múltiples operaciones tipo *scatter*, como a continuación se explica.

En una distribución cíclica entre p procesos, las primeras p filas de la matriz A van cada una a un proceso, lo cual corresponde a una operación *scatter*. Lo mismo ocurre con las siguientes p filas, y así sucesivamente. Por tanto, la distribución de la matriz entera se puede hacer mediante un bucle en el que cada iteración corresponde a una operación *scatter*. Hay que prestar atención a:

- La posición (sobre la matriz global `A`) donde empiezan los datos a enviar en cada *scatter*.
- La posición (sobre la matriz local `Aloc`) donde deben recibirse los datos en cada *scatter*.

Una vez hayas cambiado esta fase, ejecuta el programa y comprueba que la distribución se realiza correctamente. Por ejemplo, al ejecutar:

```
Para k = 0, ..., n-2
  si A(k,k) = 0 entonces abandona
  Para i = k+1, ..., n-1
    /* Modificar fila i (elementos de la columna k a la n-1) */
    A(i,k) = A(i,k)/A(k,k)
    Para j = k+1, ..., n-1
      A(i,j) = A(i,j) - A(i,k)*A(k,j)
    Fin_para
  Fin_para
Fin_para
```

Figura 9: Algoritmo secuencial de descomposición LU.

```
$ mpiexec -n 3 sistcf 5
```

La matriz `A` debería haberse distribuido de la siguiente manera:

```
Matrix A:
---- proc. 0 ----
 25.000   4.000   3.000   2.000   1.000
  2.000   3.000   4.000  25.000   4.000
---- proc. 1 ----
  4.000  25.000   4.000   3.000   2.000
  1.000   2.000   3.000   4.000  25.000
---- proc. 2 ----
  3.000   4.000  25.000   4.000   3.000
```

Los siguientes resultados que aparecen por pantalla no son correctos, porque todavía falta modificar los algoritmos de descomposición LU y sistemas triangulares.

4.4. Fases de descomposición LU y sistemas triangulares

En este apartado se describen los algoritmos de descomposición LU y resolución de sistemas triangulares, así como su paralelización. A continuación se indica qué hay que modificar para adaptar los algoritmos a la nueva distribución (cíclica por filas).

Descomposición LU (función `lu`)

El algoritmo secuencial de la descomposición LU, mostrado en la Figura 9, realiza $n-1$ etapas (bucle k), en cada una de las cuales se modifica el bloque de la matriz que corresponde a las filas $k+1, \ldots, n-1$, columnas $k, \ldots, n-1$ (ver Figura 10). La modificación de ese bloque involucra la fila k (denominada fila pivote).

El algoritmo paralelo de la descomposición LU se basa en paralelizar el bucle i del algoritmo secuencial, teniendo en cuenta que cada iteración actualiza una fila (la fila i), y que la actualización de cada fila es independiente. En el algoritmo paralelo cada proceso actualiza aquellas filas (entre $k+1$ y $n-1$) que posee, de forma que entre todos los procesos se completa la actualización.

Sin embargo, recordemos que la actualización de esas filas requiere utilizar la fila pivote (fila k), la cual está solo en uno de los procesos. Por tanto, la fila pivote debe ser enviada a todos los procesos (operación de difusión) antes de empezar la actualización de las filas.

El correspondiente algoritmo paralelo es el de la Figura 11, donde `propietario(i)` representa el proceso propietario de la fila `i`, e `iloc(i)` representa el índice local (en `Aloc`) de la fila `i` de `A`.

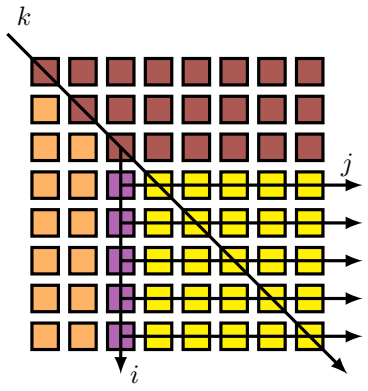


Figura 10: Esquema de la descomposición LU donde se ilustra el recorrido de los tres bucles.

```
Para k = 0, ..., n-2
  Si propietario(k) = yo
     si A(iloc(k),k) = 0 entonces abandona
  Fin_si
  difundir fila k
  Para i = k+1, ..., n-1
    /* Modificar fila i (elementos de la columna k a la n-1) */
    Si propietario(i) = yo
      A(iloc(i),k) = A(iloc(i),k)/A(k,k)
      Para j = k+1, ..., n-1
        A(iloc(i),j) = A(iloc(i),j) - A(iloc(i),k)*A(k,j)
      Fin_para
    Fin_si
  Fin_para
Fin_para
```

Figura 11: Algoritmo paralelo de descomposición LU.

Triangular Inferior

```
Para i = 0, 1, ..., n-1
  Para j = i+1, ..., n-1
    b(j) = b(j) - L(j,i)*b(i)
  Fin_para
Fin_para
```

Triangular Superior

```
Para i = n-1, ..., 0
  b(i) = b(i)/U(i,i)
  Para j = i-1, ..., 0
    b(j) = b(j) - U(j,i)*b(i)
  Fin_para
Fin_para
```

Figura 12: Algoritmos secuenciales de resolución de sistemas triangulares.

Triangular Inferior

```
Para i = 0, 1, ..., n-1
  difundir b(i)
  Para j = i+1, ..., n-1
    Si propietario(j) = yo
      b(j) = b(j) - L(iloc(j),i)*b(i)
    Fin_si
  Fin_para
Fin_para
```

Triangular Superior

```
Para i = n-1, ..., 0
  Si propietario(i) = yo
    b(i) = b(i)/U(iloc(i),i)
  Fin_si
  difundir b(i)
  Para j = i-1, ..., 0
    Si propietario(j) = yo
      b(j) = b(j) - U(iloc(j),i)*b(i)
    Fin_si
  Fin_para
Fin_para
```

Figura 13: Algoritmos paralelos de resolución de sistemas triangulares.

Resolución de sistemas triangulares (funciones `triInf` y `triSup`)

Tras la factorización, se deberá realizar la resolución de los sistemas triangulares, de acuerdo con los algoritmos que aparecen en la Figura 12. El primero de los algoritmos resuelve un sistema $Lx = b$, siendo L una matriz triangular inferior unidad, mientras que el segundo algoritmo resuelve un sistema $Ux = b$, siendo U una matriz triangular superior. En ambos casos, el vector b se sobreescribe con la solución del sistema.

Ambos algoritmos son muy similares. En cada iteración del bucle `i` se actualizan, mediante el bucle `j`, los elementos del vector `b` que hay por debajo del elemento `i` (en el sistema triangular inferior) o por encima (en el sistema triangular superior). Esta actualización requiere usar el elemento `b(i)`.

La paralelización de estos algoritmos (Figura 13) se basa en paralelizar el bucle `j`, de forma que cada proceso actualice los elementos de las filas que posee. Para ello el valor de `b(i)` deberá ser propagado previamente a todos los procesos. Al final, todos los procesos acaban con una copia del vector de incógnitas completo.

Modificación a realizar

Los algoritmos paralelos descritos anteriormente son válidos para cualquier forma de repartir las filas de la matriz entre los procesos. Lo único que hay que cambiar para trabajar con una u otra distribución es la definición de las funciones `propietario` e `iloc`.

Ejercicio 3: Modifica las funciones `propietario` e `iloc` (funciones `owner` y `localIndex` en el código proporcionado, respectivamente) para que correspondan a una distribución cíclica en vez de a una distribución por bloques. El comportamiento de estas funciones debe ser el siguiente:

- Dado un índice de fila `i` de la matriz global `A`, la función `owner` debe devolver el índice del proceso que tiene esa fila en su matriz local `Aloc`.
- Dado un índice de fila `i` de la matriz global `A`, la función `localIndex` debe devolver el índice de dicha fila en la matriz local `Aloc` del proceso propietario de la fila.

También es necesario modificar la función `numLocalRows`, que devuelve el número de filas locales de la matriz en un proceso (ese número puede ser distinto de `mb`, puesto que el número de filas puede no ser

divisible entre el número de procesos). En este caso se facilita el cambio a realizar, de manera que solo hay que descomentar la parte de código que corresponde a la distribución cíclica, y comentar o eliminar la otra parte.

Tras cambiar estas funciones, comprueba que los algoritmos de descomposición LU y resolución de sistemas triangulares funcionan correctamente. Al ejecutar el programa con 3 procesos y un sistema de 5 ecuaciones, el resultado de la descomposición LU y los sistemas triangulares debería ser:

```
Matrix LU:
---- proc. 0 ----
 25.000   4.000   3.000   2.000   1.000
  0.080   0.110   0.140  24.074   3.352
---- proc. 1 ----
  0.160  24.360   3.520   2.680   1.840
  0.040   0.076   0.108   0.139  24.071
---- proc. 2 ----
  0.120   0.144  24.131   3.373   2.614

Vector b after triInf:
---- proc. 0 ----
 35.000  32.400  30.118  27.426  24.071
---- proc. 1 ----
 35.000  32.400  30.118  27.426  24.071
---- proc. 2 ----
 35.000  32.400  30.118  27.426  24.071

Vector b after triSup (system solution):
---- proc. 0 ----
  1.000   1.000   1.000   1.000   1.000
---- proc. 1 ----
  1.000   1.000   1.000   1.000   1.000
---- proc. 2 ----
  1.000   1.000   1.000   1.000   1.000

Total accumulated error: 0.000000
```

Ejercicio 4: Una vez realizada la versión cíclica, obtén resultados experimentales de las dos variantes, comparando las prestaciones de ambas.

Utiliza un tamaño del sistema suficientemente grande (entre 1000 y 2000). Es importante evitar que el programa muestre por pantalla las matrices y vectores, ya que eso hará que tarde considerablemente más. Para ello, basta con comentar la línea:

```
#define VERBOSE
```

Analiza cuál de las dos versiones es más eficiente y trata de razonar a qué puede deberse.