Práctica 1

Implementación distribuida de un algoritmo paralelo de datos usando MPI

En esta práctica se aborda la implementación paralela del algoritmo de Floyd para el cálculo de todos los caminos más cortos en un grafo etiquetado. Se desarrollarán dos versiones paralelas del algoritmo que difieren en el enfoque seguido para distribuir los datos entre los procesos. Por simplicidad, se asume que las etiquetas de las aristas del Grafo de entrada son números enteros.

Los objetivos de esta práctica son:

- Comprender la importancia de la distribución de los datos en la resolución paralela de un problema.
- Adquirir experiencia en el uso de las funciones y mecanismos de la interfaz de paso de mensajes al abordar la paralelización de un problema de análisis de grafos.

1.1. Problema de los caminos más cortos

Sea un *grafo etiquetado* $G = (V, E, long)$, donde:

- $V = \{v_i\}$ es un conjunto de N vértices ($\|V\| = N$).
- $E \subseteq V \times V$ es un conjunto de aristas que conectan vértices de V.
- $long : E \to \mathbb{Z}$ es una función que asigna una etiqueta entera a cada arista de E.

Podemos representar un grafo mediante una *matriz de adyacencia* A tal que:

$$A_{i,j} = \begin{cases} 0 & Si\ i = j \\ long(v_i, v_j) & Si\ (v_i, v_j) \in E \\ \infty & En\ otro\ caso \end{cases}$$

En base a esta representación, podemos definir los siguientes conceptos:

- **Camino** desde un vértice v_i hasta un vértice v_j: secuencia de aristas $(v_i, v_k), (v_k, v_l), \ldots, (v_m, v_j)$ donde ningún vértice aparece más de una vez.
- **Camino más corto** entre dos vértices v_i y v_j: camino entre dicho par de vértices cuya suma de las etiquetas de sus aristas es la menor.

- **Problema del Camino más corto sencillo**: consiste en encontrar el camino más corto desde un único vértice a todos los demás vértices del grafo.
- **Problema de todos los caminos más cortos**: consiste en encontrar los camino más corto desde todos los pares de vértices del grafo.

  El Algoritmo para calcular todos los caminos más cortos toma como entrada la matriz de incidencia del Grafo A y calcula una matriz S de tamaño $N \times N$ donde $S_{i,j}$ es la longitud del camino más corto desde v_i a v_j, o un valor ∞ si no hay camino entre dichos vértices.

1.2. Algoritmo de Floyd

El algoritmo de Floyd deriva la matriz S en N pasos, construyendo en cada paso k una matriz intermedia $I(k)$ con el camino más corto conocido entre cada par de nodos. Inicialmente:

$$I^0_{i,j} = A_{ij}.$$

El k-ésimo paso del algoritmo considera cada I_{ij} y determina si el camino más corto conocido desde v_i a v_j es mayor que las longitudes combinadas de los caminos desde v_i a v_k y desde v_k a v_j, en cuyo caso se actualizará la entrada I_{ij}. La operación de comparación se realiza un total de N^3 veces, por lo que aproximamos el coste secuencial del algoritmo como $t_c N^3$ siendo t_c el coste de una operación de comparación.

```
procedure floyd secuencial
begin
I_{i,j} = A
  for k := 0 to N-1
    for i := 0 to N-1
      for j := 0 to N-1
        I^{k+1}_{i,j} = min{I^k_{i,j}, I^k_{i,k} + I^k_{k,j}}
  end;
```

1.3. Algoritmo de Floyd Paralelo 1. Descomposición unidimensional (por bloques de filas)

Asumimos que el número de vértices N es múltiplo del número de procesos P.

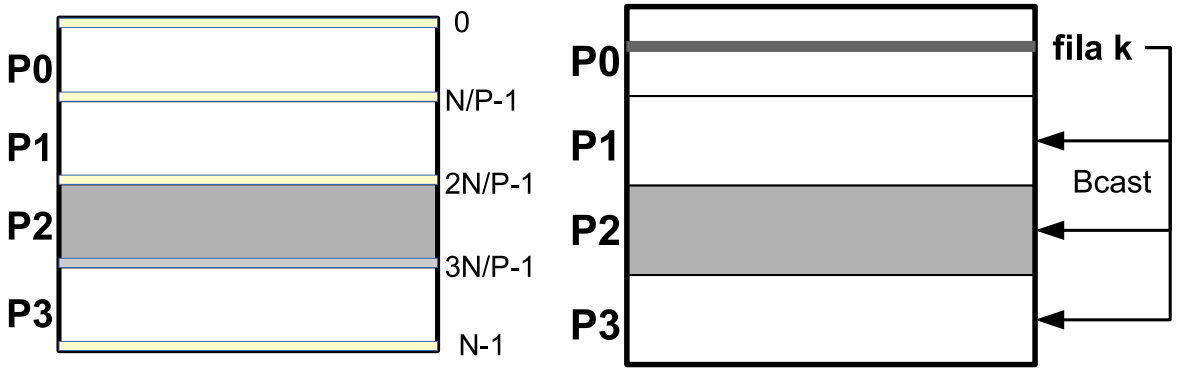
En esta versión, las filas de la matriz I se distribuyen entre los procesos por bloques contiguos de filas, por lo que cada proceso almacena N/P filas de I y de la matriz de salida S.

Se podrán utilizar hasta N procesos como máximo. Cada proceso es responsable de una o más filas adyacentes de I y ejecutará el siguiente algoritmo:

```
procedure floyd paralelo 1
begin
I_{i,j} = A
  for k := 0 to N-1
    for i := local_i_start to local_i_start
      for j := 0 to N-1
        I^{k+1}_{i,j} = min{I^k_{i,j}, I^k_{i,k} + I^k_{k,j}}
```

```
end;
```

En la iteración k, cada proceso, además de sus datos locales, necesita los valores $I_{k,0}, I_{k,1}, \ldots, I_{k,N-1},$, es decir, la fila k de I (véase figura 1.1). Por ello, el proceso que tenga asignada dicha fila k debe difundirla a todos los demás procesos (usando `MPI_Bcast`).



(a) Datos asignados a cada proceso (b) Datos requeridos por un proceso en la etapa k

Figura 1.1: Distribución 1D de la matriz I entre 4 procesos

Para distribuir la matriz A entre los procesos basta con utilizar la operación colectiva `MPI_Scatter`.

1.4. Algoritmo de Floyd Paralelo 2. Descomposición bidimensional (por bloques 2D)

Por simplicidad, se asume que el número de vértices N es múltiplo de la raíz del número de procesos P.

Esta versión del algoritmo de Floyd utiliza una distribución por bloques bidimensionales de la matriz I, pudiendo utilizar hasta N^2 procesos. Suponemos que los procesos se organizan lógicamente como una malla cuadrada con $sqrtP$ procesos en cada fila y $sqrtP$ procesos en cada columna.

Cada proceso trabaja con un bloque de $N/\sqrt{P}$ subfilas alineadas (cubren las mismas columnas contiguas) con $N/\sqrt{P}$ elementos cada uno (véase figura 1.2) y ejecuta el siguiente algoritmo:

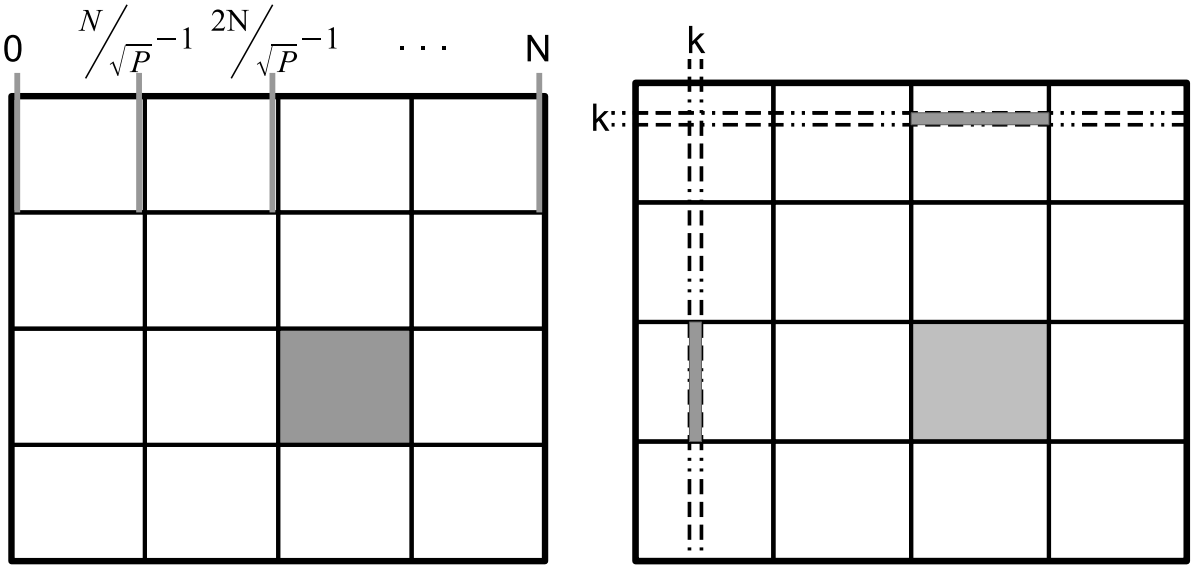
```
procedure floyd paralelo 2
begin
I_{i,j} = A
  for k := 0 to N-1
    for i := local_i_start to local_i_end
      for j := local_j_start to local_j_end
        I^{k+1}_{i,j} = min{I^k_{i,j}, I^k_{i,k} + I^k_{k,j}}
  end;
```

En cada paso, además de los datos locales, cada proceso necesita $N/\sqrt{P}$ valores de dos procesos localizados en la misma fila y columna respectivamente (véase figura 1.2). Por tanto, los requerimientos de comunicación en la etapa k son dos operaciones de broadcast:

- Desde el proceso en cada fila (de la malla de procesos) que contiene parte de la columna k al resto de procesos en dicha fila.

- Desde el proceso en cada columna (de la malla de procesos) que contiene parte de la fila k al resto de procesos en dicha columna.

En cada uno de los N pasos, $N/\sqrt{P}$ valores deben difundirse a los $\sqrt{P}$ procesos en cada fila y en cada columna d ela malla de procesos. Nótese que cada proceso debe servir de origen (`root`) para al menos un broadcast a cada proceso en la misma fila y a cada proceso en la misma columna del malla lógica 2D de procesos.



(a) Datos asignados a cada proceso

(b) Datos requeridos por un proceso en la etapa k

Figura 1.2: Distribución 2D de la matriz I entre 16 procesos

Cada uno de los P procesos almacena una submatriz de I de tamaño $N/\sqrt{P} \times N/\sqrt{P}$ (por simplicidad, supondremos que $\sqrt{P}$ divide a N).

Inicialmente el proceso P_0 contiene la matriz completa, y a cada proceso le corresponderán $N/\sqrt{P}$ elementos de $N/\sqrt{P}$ filas de dicha matriz.

Para permitir que la matriz I pueda ser repartida con una operación colectiva entre los procesos, podemos definir un tipo de datos para especificar submatrices. Para ello, es necesario considerar que los elementos de una matriz bidimensional se almacenan en posiciones consecutivas de memoria por orden de fila, en primer lugar, y por orden de columna en segundo lugar. Así cada submatriz será un conjunto de $N/\sqrt{P}$ bloques de $N/\sqrt{P}$ elementos cada uno, con un desplazamiento de N elementos entre cada bloque.

La función `MPI_Type_vector` permite asociar una submatriz cuadrada de una matriz como un tipo de datos de MPI. De esta manera, el proceso P_0 podrá enviar bloques no contiguos de datos a los demás procesos en un solo mensaje. También es necesario calcular, para cada proceso, la posición de comienzo de la submatriz que le corresponde.

Para poder alojar de forma contigua y ordenada los elementos del nuevo tipo creado (las submatrices cuadradas), con objeto de poder repartirlos con MPI_Scatter entre los procesos, podemos utilizar la función `MPI_Pack`. Utilizando esta función, se empaquetan las submatrices de forma

consecutiva, de tal forma que al repartirlas (usando el tipo `MPI_PACKED`, se le asigna una submatriz cuadrada a cada proceso. A continuación, se muestra la secuencia de operaciones necesarias para empaquetar todos los bloques de una matriz $N \times N$ de forma ordenada y repartirlos con un `MPI_Scatter` entre los procesos:

```
MPI_Datatype MPI_BLOQUE;
......
......

raiz_P=sqrt(P);
tam=N/raiz_P;

/*Creo buffer de envío para almacenar los datos empaquetados*/
buf_envio=reservar_vector(N*N);

if (rank==0)
  {
  /* Obtiene matriz local a repartir*/
  Inicializa_matriz(N,N,matriz_I);
   /*Defino el tipo bloque cuadrado */
   MPI_Type_vector (tam, tam, N, MPI_INT, &MPI_BLOQUE);
   /* Creo el nuevo tipo */
   MPI_Type_commit (&MPI_BLOQUE);

   /* Empaqueta bloque a bloque en el buffer de envío*/
   for (i=0, posicion=0; i<size; i++)
     {
        /* Calculo la posicion de comienzo de cada submatriz */
        fila_P=i/raiz_P;
        columna_P=i%raiz_P;
        comienzo=(columna_P*tam)+(fila_P*tam*tam*raiz_P);
        MPI_Pack (matriz_I(comienzo), 1, MPI_BLOQUE,
                          buf_envio,sizeof(int)*N*N, &posicion, MPI_COMM_WORLD);
     }

     /*Destruye la matriz local*/
     free(matriz_I);
     /* Libero el tipo bloque*/
     MPI_Type_free (&MPI_BLOQUE);
   }

/*Creo un buffer de recepcion*/
buf_recep=reservar_vector(tam*tam);
/* Distribuimos la matriz entre los procesos */
MPI_Scatter (buf_envio, sizeof(int)*tam*tam, MPI_PACKED,
                      buf_recep, tam*tam, MPI_INT, 0, MPI_COMM_WORLD);
```

Para obtener la matriz resultado en el proceso P_0 se utiliza la función `MPI_Gather` seguida de `MPI_Unpack`

1.5. Ejercicios propuestos.

1. Implementar los algoritmos de cálculo de todos los caminos más cortos que han sido descritos previamente. Se proporcionará en la web de documentación de la asignatura una versión secuencial en C++ del algoritmo de Floyd, que se puede utilizar como plantilla para programar los diferentes algoritmos paralelos. Se aconseja realizar cambios sobre la clase `Graph`, que encapsula la gestión del grafo etiquetado, para implementar todas las operaciones de comunicación dentro de dicha clase. Por tanto, la nueva clase `Graph` encapsularía toda la funcionalidad asociada a la gestión de la matriz de incidencia distribuida entre los procesos (por bloques de filas en el caso 1D o por bloques cuadrados en el caso 2D). En la carpeta `input`, se encuentran varios archivos de descripción de grafos que se pueden utilizar como entradas del programa. También se proporciona un programa (`creaejemplo.cpp`) para crear archivos de entrada pasando como argumento el número de vértices del Grafo. Este programa permnitirá realizr pruebas con grafos de diferentes números de vértices (600, 800, 1000, 1200, etc.).

   Se debe crear una carpeta diferente para cada versión paralela (`Floyd-1` y `Floyd-2`). Cada carpeta mantendrá los mismos archivos que se usan en la versión secuencial pero con diferente código. De esa forma el `Makefile` se puede reutilizar y se percibe claramente la diferencia entre las versiones secuencial y paralela.

2. Realizar medidas de tiempo de ejecución sobre los algoritmos implementados. Para medir tiempos de ejecución, podemos utilizar la función `MPI_Wtime`. Para asegurarnos de que todos los procesos comienzan su computación al mismo tiempo podemos utilizar `MPI_Barrier`. Las medidas deberán excluir las fases de e/s. Deberán realizarse las siguientes medidas:

   a) Medidas para el algoritmo secuencial ($P = 1$).

   b) Medidas para el algoritmo paralelo ($P = 4$). Para los algoritmos de multiplicación matriz-vector, las medidas deberán excluir las fases de entrada/salida, así como la fase de distribución inicial de la matriz A desde el proceso P_0 y la fase de reunión de la matriz resultado en P_0.

   Las medidas deberán realizarse para diferentes tamaños de problema, para así poder comprobar el efecto de la granularidad sobre el rendimiento de los algoritmos.

   Se presentará una tabla con el siguiente formato:

Tiempo	$P = 1$ (secuencial)	$P = 4$	Ganancia
$n = 60$			
$n = 240$			
$n = ...$			
$n = 1200$			

   La ganancia en velocidad se calcula dividiendo el tiempo de ejecución del algoritmo secuencial entre el tiempo de ejecución del algoritmo paralelo.