Práctica 2

Cálculo paralelo de los caminos más cortos

En esta práctica se aborda la implementación paralela y el modelado del rendimiento 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 ha supuesto que las etiquetas de las aristas son números enteros.

El objetivo de esta práctica es hacer que el alumno comprenda la importancia de la distribución de datos para la resolución paralela de un problema, mediante la implementación de dos versiones algorítmicas diferentes para resolver el mismo problema. También se persigue que adquiera experiencia en el uso de funciones y mecanismos de la interfaz de paso de mensajes para resolver un problema de análisis de grafos, así como en el modelado teórico del tiempo de ejecución de un algoritmo paralelo.

2.1 Problema de los caminos más cortos

Inicialmente se presenta un algoritmo secuencial para resolver el problema de encontrar los caminos más cortos en un grafo.

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 arcos que conectan vértices de V.
- $long : E \to Z$ es una función que asigna una etiqueta entera a cada arista de E.

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

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

Un *camino* desde el vértice v_i al vértice v_j es una secuencia de vértices $(v_i, v_k), (v_k, v_l), \ldots, (v_m, v_j)$, donde ningún vértice aparece más de una vez. El *camino más corto* entre dos vértices v_i y v_j es el camino cuya suma de las etiquetas en sus aristas es menor.

El *problema del camino más corto sencillo* requiere encontrar el camino más corto desde un único vértice a todos los demás vértices del grafo. El problema de *todos los caminos más cortos*

requiere encontrar los caminos más cortos entre todos los pares de vértices del grafo. El algoritmo para resolver este último problema toma como entrada la matriz de incidencia 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 vertices.

2.2 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:

$$I_{i,j}(0) = A_{i,j}.$$

El k-ésimo paso del algoritmo considera cada $I_{i,j}$ 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 $I_{i,j}$. 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_sequential
begin
  I_{i,j} = 0 si i = j
  I_{i,j} = long(v_i, v_j) si (v_i, v_j) ∈ E
  I_{i,j} = ∞ en otro caso
  for k=0 to N-1
    for i=0 to N-1
      for j=0 to N-1
        I_{i,j}(k+1) = min(I_{i,j}(k), I_{i,k}(k) + I_{k,j}(k))
      endfor
    endfor
  endfor
  S = I(N)
end
```

Figura 2.1: Algoritmo de Floyd Secuencial.

2.3 Algoritmo de Floyd Paralelo-1. Descomposición unidimensional

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

La primera versión paralela del algoritmo de Floyd está basada en una descomposición unidimensional por bloques de filas de la matriz intermedia I y de la matriz de salida S. Podremos utilizar N procesadores como máximo. Cada tarea es responsable de una o más filas adyacentes de I, y ejecutará el siguiente código:

```
for k=0 to N-1
  for i = local_i_start to local_i_end
```

```
    for j=0 to N-1
      I_{i,j}(k+1) = min(I_{i,j}(k), I_{i,k}(k) + I_{k,j}(k))
    endfor
  endfor
endfor
```

En la iteración k, cada tarea, 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 (ver figura 2.2). Por ello, la tarea que tenga asignada la fila k deberá difundirla (`MPI_Broadcast`) a todas las demás.

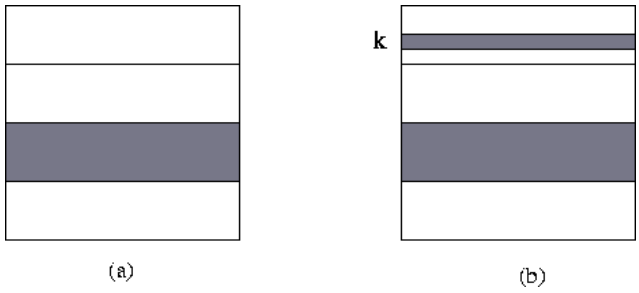

Figura 2.2: Descomposición 1D de la matriz I entre 4 procesadores para la primera versión paralela del algoritmo de Floyd. En (a) se muestran los datos asignados a una tarea. En (b) se muestran los datos necesitados por esta tarea en la etapa k del algoritmo.

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

2.4 Algoritmo de Floyd Paralelo-2. Descomposición bidimensional

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 descomposición bidimensional, pudiendo utilizar hasta N^2 procesadores:

```
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_{i,j}(k+1) = min(I_{i,j}(k), I_{i,k}(k) + I_{k,j}(k))
    endfor
  endfor
endfor
```

En cada paso, además de los datos locales, cada tarea necesita $N/\sqrt{P}$ valores de dos tareas localizadas en la misma fila y columna respectivamente (ver figura 2.3). Por tanto, los requerimientos de comunicación en la etapa k son dos operaciones de broadcast:

- desde la tarea en cada fila que contiene parte de la columna k a todas las demás tareas en esta fila y,

- desde la tarea en cada columna que contiene parte de la fila k a todas las demás tareas en esta columna.

En cada uno de los N pasos, $N/\sqrt{P}$ valores deben ser difundidos a las $\sqrt{P}$ tareas en cada fila y en cada columna. Nótese que cada tarea debe servir como el origen para al menos un broadcast a cada tarea en la misma fila y a cada columna del array 2D.

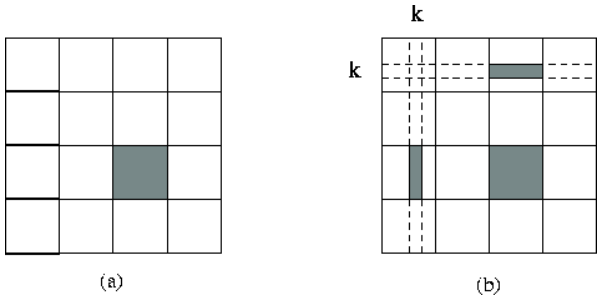

Figura 2.3: Descomposición 2D de la matriz I entre 16 procesadores para la segunda versión paralela del algoritmo de Floyd. En (a) se muestran los datos asignados a una tarea. En (b) se muestran los datos necesitados por esta tarea en la etapa k del algoritmo.

En el caso general, cada uno de los P procesadores almacena una submatriz de A de tamaño $N/\sqrt{P} \times N/\sqrt{P}$ (por simplicidad, supondremos que $\sqrt{P}$ divide a N).

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

Para permitir que la matriz A pueda ser repartida con una operación colectiva entre los procesadores, 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 como un tipo de datos. De esta manera, el procesador P_0 podrá enviar bloques no contiguos de datos a los demás procesadores en un solo mensaje. También es neesario calcular, para cada procesador, 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 procesadores, 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 procesador. A continuación, se muestra una porción de código C con la secuencia de operaciones necesarias para empaquetar todos los bloques de una matriz $N \times N$ (de floats) de forma ordenada, y repartirlos con un MPI_Scatter entre los procesadores:

```
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_A);
   /*Defino el tipo bloque cuadrado */
   MPI_Type_vector (tam, tam, n, MPI_FLOAT, &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_A(comienzo), 1, MPI_BLOQUE,
                          buf_envio,sizeof(float)*n*n, &posicion, MPI_COMM_WORLD);
     }

     /*Destruye la matriz local*/
     free(matriz_A);
     /* 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 procesadores */
MPI_Scatter (buf_envio, sizeof(float)*tam*tam  , MPI_PACKED,
                      buf_recep, tam*tam, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Para obtener la matriz resultado en el procesador P_0, se utiliza un `MPI_Gather` seguido de `MPI_Unpack`.

2.5 Ejercicios propuestos.

1. Implementar los algoritmos de cálculo de todos los caminos más cortos que han sido descritos previamente. En `$MPIDIR/../pract2` se encuentra una versión secuencial en C++ del algoritmo de Floyd, que se puede utilizar como plantilla para programar las diferentes versiones paralelas. 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 clase encapsularía un grafo distribuido entre los procesadores en base a una determinada distribución (por bloques de filas, Floyd-1, o por bloques cuadrados, Floyd-2). En el directorio input, se encuentran dos archivos de descripción de grafos que se pueden utilizar como entrada al programa. Se proporcionará al alumno en la página web un programa para crear más archivos de entrada que representen grafos con un mayor número de vértices (120, 360, 1200, etc.) con objeto de realizar pruebas de mayor envergadura.

Se debe crear un directorio diferente para cada versión paralela, (Floyd-1 y Floyd-2). Cada directorio 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 la versión secuencial y la 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 procesadores 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 ($P = 4$). Las medidas deberán excluir las fases de entrada/salida, así como la fase de distribución inicial de la matriz A desde P_0 y la fase de reunión del 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
$N = 60$			
$N = 240$			
$N = ...$			

3. Suponiendo que las P tareas se ejecutan sobre P procesadores conectados todos con todos, y que las operaciones colectivas se han implementado siguiendo algoritmos que asumen un hipercubo como topología de conexión, obtener fórmulas de tiempo de ejecución para los algoritmos implementados en base a parámetros del problema y a parámetros que caracterizan la arquitectura (t_c=tiempo que lleva una comparación, t_s=tiempo de inicialización de envío, t_w=tiempo de transferencia por palabra).