Práctica 2

Implementación en memoria compartida y en memoria distribuida de un algoritmo paralelo de datos

En esta práctica se aborda la implementación paralela en plataformas multiprocesador de memoeira compartida y de memoria distribuida del algoritmo de Floyd para el cálculo de todos los caminos más cortos en un grafo etiquetado. Se plantearán dos versiones paralelas del algoritmo que difieren en el enfoque seguido para repartir los datos entre las tareas. 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 descomposición de las tareas y los datos en la resolución paralela de un problema.
- Adquirir experiencia en el uso de las directivas y funciones de OpenMP para obtener una versión paralela de un algoritmo.
- Adquirir experiencia en el uso de las funciones y mecanismos de la interfaz de paso de mensajes (MPI) para abordar la implementación distribuida de un algoritmo paralelo de datos.

2.1 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.

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:

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

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;
```

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

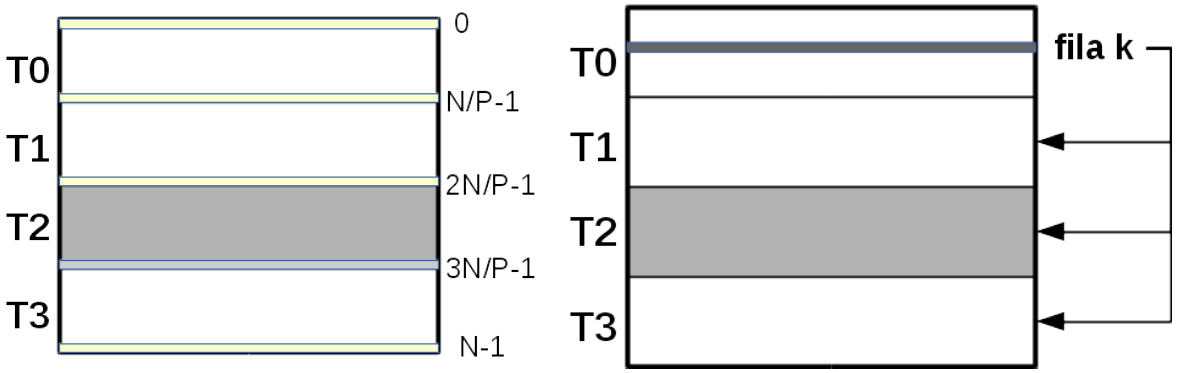
Asumimos que el número de vértices N es múltiplo del número de tareas P.

En esta versión, cada tarea procesa un bloque contiguo de filas de la matriz I por lo que cada tarea procesa N/P filas de I.

Se podrán utilizar hasta N tareas como máximo. Cada tarea 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 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 (véase figura 2.1).



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

Figura 2.1: Dependencias entre los datos gestionados por diferentes tareas para 4 tareas

2.3.1 Implementación en OpenMP

Es bastante sencillo derivar una implementación paralela de este algoritmo paralelo usando la directiva de OpenMP para paralelización de bucles. Basta con repartir equitativamente las iteraciones del segundo bucle del algoritmo secuencial e ntre las hebras. No obstante, se recomienda lo siguiente:

- Para evitar degradación de eficiencia al acceder a datos compartidos y en el uso de las cachés se recomienda que cada hebra privatice la fila k-ésima, es decir, que copie en su espacio privado dicha fila al comienzo de la k-ésima iteración.
- Se aconseja no lanzar una región paralela para cada valor de k sino crear una única región paralela antes del bucle principal y usar una directiva `for` para cada nuevo valor de k. Hay que tener en cuenta que después de cada `#pragma omp for` hay un barrier implícito que nos asegura la coherencia de los datos entre una iteración y la siguiente.

2.4 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 tareas P.

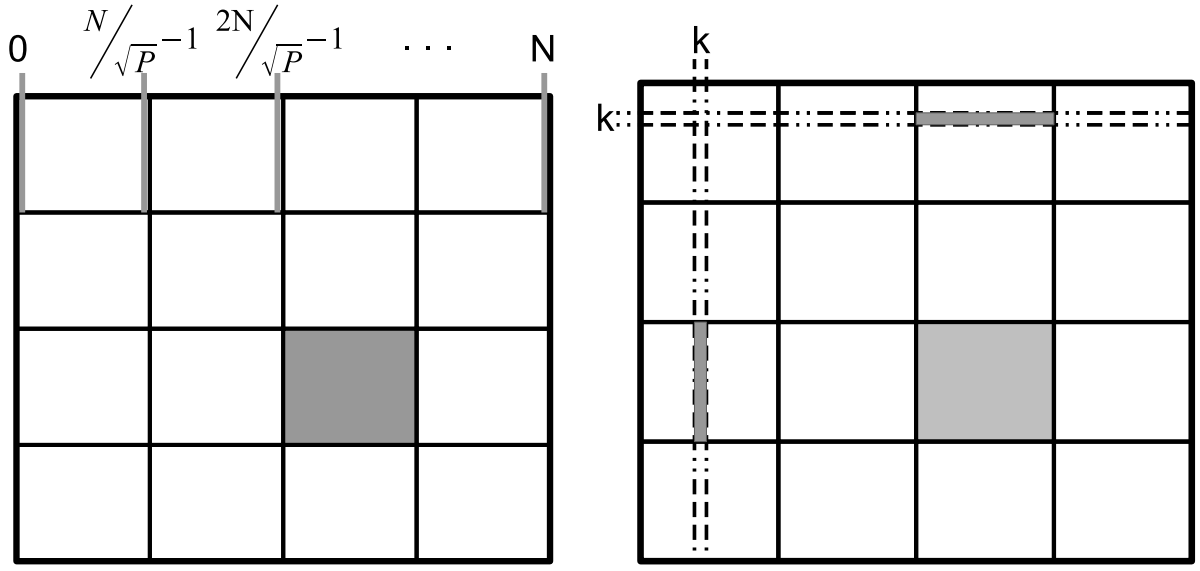
Esta versión del algoritmo de Floyd utiliza un reparto por bloques bidimensionales de la matriz I entre las tareas, pudiendo utilizar hasta N^2 procesos. Suponemos que las tareas se organizan lógicamente como una malla cuadrada con $\sqrt{P}$ tareas en cada fila y $\sqrt{P}$ tareas en cada columna.

Cada tarea trabaja con un bloque de $N/\sqrt{P}$ subfilas alineadas (cubren las mismas columnas contiguas) con $N/\sqrt{P}$ elementos cada uno (véase figura 2.2) 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 tarea necesita $N/\sqrt{P}$ valores de dos procesos localizados en la misma fila y columna respectivamente (véase figura 2.2).

Cada una de las P tareas procesa una submatriz de I de tamaño $N/\sqrt{P} \times N/\sqrt{P}$ (por simplicidad, supondremos que $\sqrt{P}$ divide a N).



(a) Datos asignados a cada tarea

(b) Datos requeridos por una tarea en la etapa k con $P = 4$

Figura 2.2: Distribución 2D de la matriz I entre 16 procesos

2.4.1 Implementación del Algoritmo de Floyd 2 en OpenMP

En este caso no se puede usar la directiva de OpenMP para paralelización de bucles ya que son dos los bucles que se trocean. Por tanto, cada hebra tendrá que identificar explícitamente la parte de la matriz que debe actualizar y sería conveniente lo siguiente:

- Para mejorar la eficiencia se recomienda que cada hebra copie en su espacio privado antes del bucle principal la parte de la matriz con la que trabajará y que, al final del cómputo, todas las hebras cooperen para escribir sus resultados locales en la matriz $N \times N$. También cada hebra debería privatizar la subfila y la subcolumna que necesita al comienzo de la k-ésima iteración.

- Igualmente se aconseja crear una única región paralela. No obstante, debemos tener en cuenta que al final de cada iteración del bucle principal no hay un barrier implícito por lo que debemos hacerlo emplícitamente para asegurar la coherencia de los datos entre iteraciones.

2.4.2 Implementación del Algoritmo de Floyd 2 en MPI

Teniendo en cuenta las dependencias de datos entre procesos, los requerimientos de comunicación en la etapa k requieren de 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 de la 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 de procesos bidimensional.

Distribución inicial de los datos de entrada desde el proceso 0 y recolección de resultados

Inicialmente el proceso P_0 contiene la matriz completa (tras leer el grafo desde un archivo), 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 realiza una llamada a la función `MPI_Gather` seguido de P llamadas a la función `MPI_Unpack` para ir descomprimiendo los paquetes asociados a .

2.5 Ejercicios propuestos.

Se proporcionará en la plataforma PRADO una versión secuencial en C++ del algoritmo de Floyd, que se puede utilizar como plantilla para programar los diferentes algoritmos paralelos. 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.).

1. Implementar los algoritmos paralelos de cálculo de todos los caminos más cortos que han sido descritos previamente usando directivas y funciones de OpenMP.

   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. Implementar usando MPI una versión distribuida del algoritmo de Floyd 2 (bidimensional). 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 cuadrados).

3. Realizar medidas de tiempo de ejecución sobre los algoritmos implementados. Deberán realizarse las siguientes medidas:

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

   (b) Medidas para los algoritmos paralelos ($P = 4, 9$).

   Para la implementación en MPI, las medidas deberán excluir las fases de entrada/salida, así como la fase de distribución inicial de la matriz desde el proceso P_0 y la fase de reunión de la matriz resultado en P_0. En este caso, para medir tiempos de ejecución, podemos utilizar la función `MPI_Wtime` y para asegurarnos de que todos los procesos comienzan su computación al mismo tiempo podemos utilizar `MPI_Barrier`.

   Las medidas deberán realizarse para diferentes tamaños de problema, para así poder comprobar el efecto de la granularidad sobre el rendimiento de los algoritmos y se deberá dar una descripción de la plataforma de ejecución usada.

   Se presentará una tabla con el siguiente formato para cada implementación (Alg-1-OpenMP, Alg-2-OpenMp y Alg-2-MPI):

	$T(P=1)$ (sec.)	$T_P(P=4)$	$S(P=4)$	$T_P(P=9)$	$S(P=9)$
$n = 60$					
$n = 240$					
$n = ...$					
$n = 1200$					

   La ganancia en velocidad o Speedup (S) se calcula dividiendo el tiempo de ejecución del algoritmo secuencial entre el tiempo de ejecución del algoritmo paralelo. Se valorará que se aporte una o varias gráficas que muestren cómo varía el speedup con el tamaño del problema en cada implementación con $P = 4$ y $P = 9$.