Práctica 2

Implementación en memoria distribuida de un algoritmo paralelo de datos

En esta práctica se aborda la implementación paralela y distribuida, usando MPI, 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 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 descomposición de las tareas y 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 (MPI) para abordar la implementación distribuida de un algoritmo paralelo de datos.

Para orientaros sobre el funcionamiento de los algoritmos paralelos podéis visitar la página: `https://lsi.ugr.es/jmantas/ppr/practicas/practicas.php?prac=prac02`.
que presenta animaciones que ilustran gráficamente el comportanmiento de los dos algoritmos distribuidos.

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 con el camino más corto conocido entre cada par de nodos. Inicialmente se parte de la matriz A y se va actualizando en cada paso del algortimo.

El k-ésimo paso del algoritmo considera cada A_{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 A_{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.

```
Algoritmo floyd secuencial
begin
  for k := 0 to N-1
    for i := 0 to N-1
      for j := 0 to N-1
        A_{i,j} = min{A_{i,j}, A_{i,k} + A_{k,j}}
end
```

En los códigos que se proporcionan con la práctica, se incluye una implementación secuencial del algoritmo en el archivo `floyd_seq.cc` que usa una clase `Graph` (que se implementa en los archivos `Graph.cc` y `Graph.h`).

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 procesos P.

En esta versión, cada tarea procesa un bloque contiguo de filas de la matriz A (que aquí denominamos $Bloque\_A$) por lo que cada tarea procesa N/P filas de A (con N^2/P elementos).

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

```
Algoritmo Floyd paralelo 1
begin
  for k := 0 to N-1
    for i := local_i_start to local_i_end − 1
      for j := 0 to N-1
        Bloque_A[i, j] = min{Bloque_A[i, j], Bloque_A[i, k] + A[k, j]}
end
```

Los valores $local\_i\_start$ y $local\_i\_end$ designan los índices de inicio y fin, en la dimensión de las filas (i), para el bloque de filas contiguas ($Bloque\_A$) que maneja cada proceso. Por tanto, se verifica que:

$$local\_i\_start = 0, \qquad local\_i\_end = N/P.$$

Estos índices de inicio y fin "locales" se corresponden con índices globales de fila en la matriz A, en función del rango (`rank`) de cada proceso en `MPI_COMM_WORLD`:

$$global\_i\_start = rank * N/P, \qquad global\_i\_end = global\_i\_start + N/P.$$

2.3.1 Operaciones de comunicación necesarias

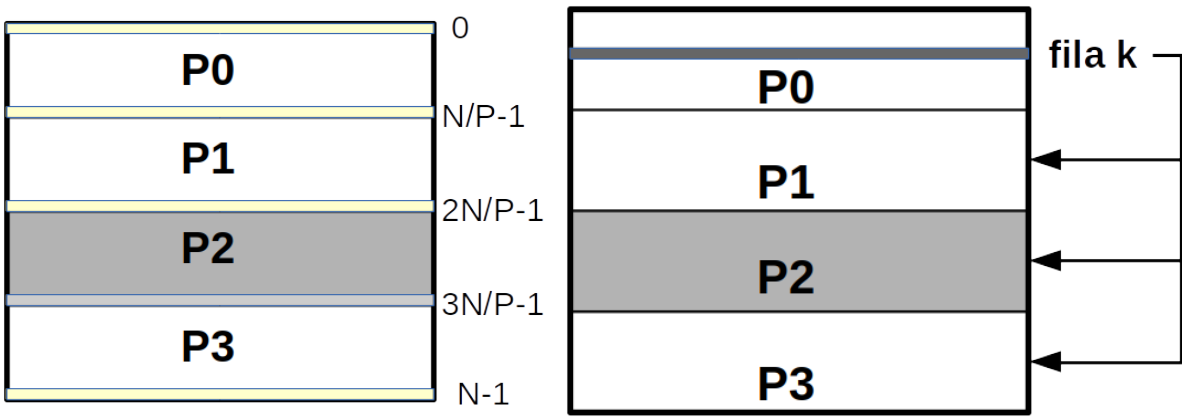
1. Como la matriz de entrada A se lee en el proceso 0, habrá que repartir esta matriz desde el proceso 0 a todos los procesos, para que cada proceso obtenga sus valores iniciales en su bloque local de N/P filas ($Bloque\_A$). Para ello, todos los procesos deben ejecutar una operación de `Scatter` con bloques de N/P filas (N^2/P enteros), siendo raíz el proceso 0.

2. En la iteración k, cada tarea, además de sus datos locales, necesitará la fila k de A, es decir, los valores $A_{k,0}, \ldots, A_{k,N-1},$. (véase figura 2.1 b). Esto se realiza mediante una operación de `Broadcast` de un vector de N elementos entre todos los procesos de `MPI_COMM_WORLD`.

3. Cuando finalizan los bucles, cada proceso mantiene en su submatriz local con N/P filas ($Bloque\_A$) una porción de la matriz resultado. Para poder imprimir el resultado, el proceso 0 debe obtener los bloques locales de cada proceso y componer así la matriz resultado A. Esta operación se realiza mediante un `Gather` con bloques de N/P filas (N^2/P enteros), siendo raíz el proceso 0.

Como resultado, el Algoritmo distribuido, insertando la operación de reparto (Scatter), la operación para difundir la fila k-ésima en cada iteración (Broadcast) y la operación final de recogida de datos (Gather), queda como sigue:

```
Algoritmo Floyd paralelo 1
begin
  Scatter de la matriz de entrada A en Bloque_A de cada proceso
  for k := 0 to N-1
    Broadcast fila_k de A desde proceso propietario a todos
    for i := local_i_start to local_i_end − 1
      for j := 0 to N-1
        Bloque_A[i, j] = min{Bloque_A[i, j], Bloque_A[i, k] + fila_k[j]}
  Gather de Bloque_A de cada proceso para componer matriz resultado en proceso 0
end
```



(a) Datos asignados a cada proceso

(b) Datos requeridos por un proceso en etapa k

Figura 2.1: Datos gestionados por diferentes procesos para $P = 4$

En los códigos que se proporcionan con la práctica, se incluye una implementación de este algoritmo paralelo usando MPI en el archivo `floyd_par1d.cc`. Se han de entender todos los detalles de esta implementación como paso necesario para poder abordar la práctica.

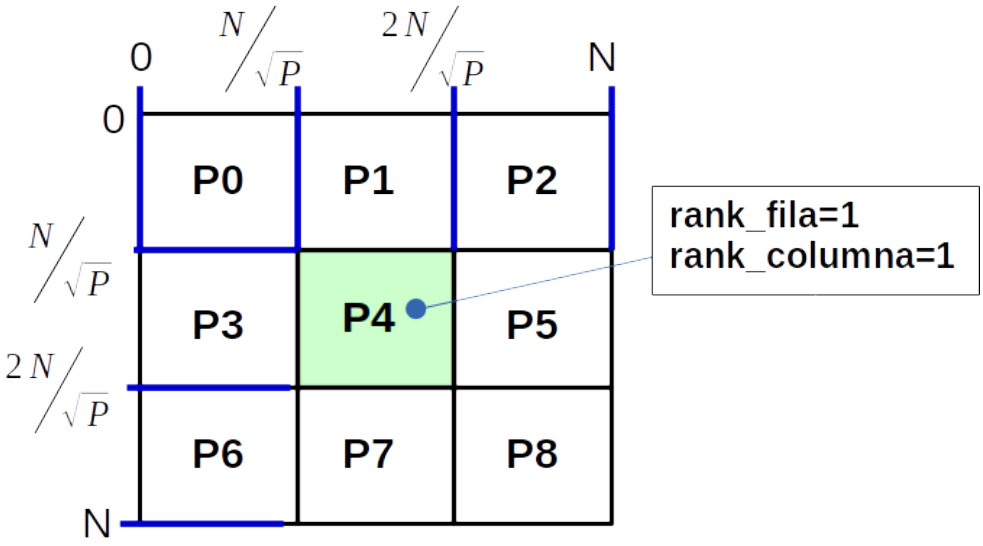


Figura 2.2: Distribución 2D de la matriz A entre $P = 9$ procesos

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

En este caso, asumiremos, por simplicidad, que el número de vértices N del Grafo es múltiplo de la raíz del número de procesos P, es decir, que $N \mod \sqrt{P} = 0$.

Esta versión del algoritmo de Floyd utiliza un reparto por bloques bidimensionales de la matriz A entre los procesos, pudiéndose utilizar hasta N^2 procesos. Suponemos que los procesos se organizan lógicamente como una malla cuadrada con $\sqrt{P} \times \sqrt{P}$ procesos, es decir, con $\sqrt{P}$ filas de procesos y $\sqrt{P}$ columnas de procesos (véase figura 2.2 para $P = 3 \times 3$).

Cada proceso se encarga de actualizar su correspondiente submatriz de la matriz A, $Bloque\_A$, que tiene $N/\sqrt{P}$ subfilas alineadas (cubren las mismas columnas contiguas), donde cada subfila tiene $N/\sqrt{P}$ elementos (véase figura 2.2). Por tanto, el array $Bloque\_A$ que maneja cada proceso mantiene N^2/P elementos. Cada proceso ejecuta el siguiente algoritmo:

```
Algoritmo floyd paralelo 2
begin
  for k := 0 to N-1
    for i := local_i_start to local_i_end − 1
      for j := local_j_start to local_j_end − 1
        Bloque_A[i, j] = min{Bloque_A[i, j], A[i, k] + A[k, j]}
end
```

Los valores $local\_i\_start$ y $local\_i\_end$, designan los índices de inicio y fin, en la dimensión de las filas (i), para la submatriz local ($Bloque\_A$) que maneja cada proceso. Los valores $local\_j\_start$ y $local\_j\_end$, designan los índices de inicio y fin, en la dimensión de las columnas (j), para la submatriz local ($Bloque\_A$) que maneja cada proceso. Como $Bloque\_A$ es una submatriz de A con dimensión $N/\sqrt{P} \times N/\sqrt{P}$, se verifica que:

$$local\_i\_start = local\_j\_start = 0, \qquad local\_i\_end = local\_j\_end = N/\sqrt{P}.$$

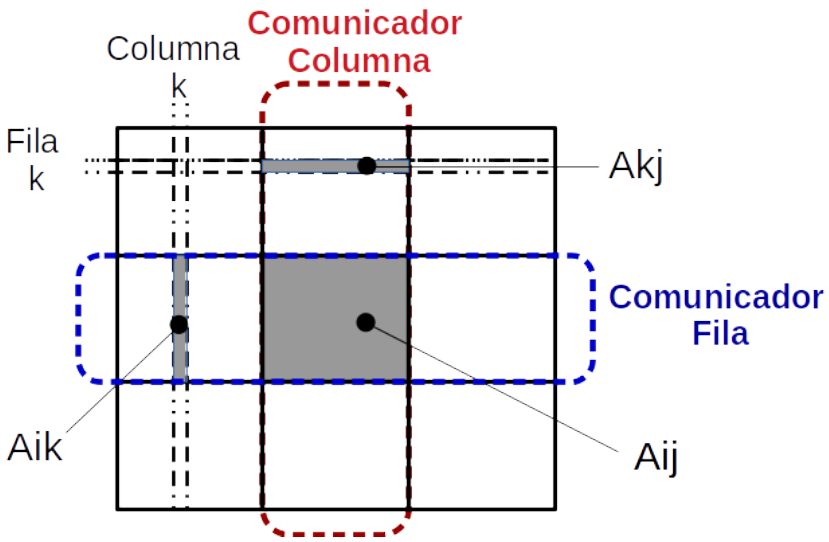


Figura 2.3: Datos requeridos por cada proceso en la iteración k con $P = 9$ procesos

Estos índices de inicio y fin "locales" se corresponden con índices globales en la matriz A. Los índices globales se pueden calcular fácilmente en función del índice de fila, $rank\_fila$, y el índice de columna, $rank\_columna$, que ocupa el proceso en la malla lógica de procesos:

$$global\_i\_start = rank\_fila * N/\sqrt{P}, \qquad global\_i\_end = global\_i\_start + N/\sqrt{P}.$$
$$global\_j\_start = rank\_columna * N/\sqrt{P}, \qquad global\_j\_end = global\_j\_start + N/\sqrt{P}.$$

donde los índices de fila y columna se pueden calcular fácilmente, a partir del del rango (`rank`) de cada proceso en `MPI_COMM_WORLD`:

$$rank\_fila = rank \div \sqrt{P}, \qquad rank\_columna = rank - rank\_fila * \sqrt{P}.$$

donde $\div$ denota la división entera.

2.4.1 Requisitos de comunicación dentro del bucle principal

En la iteración k-ésima, para que cada proceso pueda realizar la actualización de cada elemento de su submatriz local, el proceso proceso $rank$ necesita los siguientes elementos de la matriz global:

$$\mathbf{A}[\mathbf{i}, \mathbf{k}], \qquad \mathbf{i} = global\_i\_start, \ldots, global\_i\_end,$$
$$\mathbf{A}[\mathbf{k}, \mathbf{j}], \qquad \mathbf{j} = global\_j\_start, \ldots, global\_j\_end.$$

Por consiguiente, en cada iteración del bucle principal, cada proceso necesita una subfila (con $N/\sqrt{P}$ valores) de la fila k-ésima de la matriz global (que residirá en alguno de los procesos en su columna de la malla de procesos) y una subcolumna (con $N/\sqrt{P}$ valores) de la columna k-ésima de la matriz global (que residirá en alguno de los procesos en su fila de la malla de procesos). La Figura 2.3 ilustra gráficamente estas dependencias de datos para el caso de $rank = 4$ y $P = 9$ procesos.

Teniendo en cuenta estas dependencias de datos entre procesos, los requerimientos de comunicación en la etapa k exigen la participación de cada proceso en dos operaciones de broadcast:

- Desde el proceso en cada fila (de la malla de procesos) que contiene una subcolumna de la columna k (con $N/\sqrt{P}$ elementos) al resto de procesos en dicha fila.
- Desde el proceso en cada columna (de la malla de procesos) que contiene una subfila de la fila k (con $N/\sqrt{P}$ elementos) al resto de procesos en dicha columna de la malla de procesos.

Para que estos broadcasts sean posibles, cada proceso debe haber participado previamente en la creación de un comunicador para el grupo de procesos en su fila de la malla de procesos (`comm_fila`) y de otro para el grupo de procesos en su columna de la malla de procesos (`comm_columna`). Téngase en cuenta que, aunque cada proceso solo participará en la creación de dos comunicadores (el de su fila y el de su columna en la malla de procesos), a nivel global se crearán $2\sqrt{P}$ comunicadores (uno por cada fila y por cada columna en la malla de procesos).

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 de la malla lógica de procesos bidimensional.

Con estas operaciones colectivas, el algoritmo presenta la formulación que se presenta a continuación, sin tener en cuenta el reparto de la matriz por bloques 2D entre los procesos y la recolección del resultado.

```
Algoritmo floyd paralelo 2
begin
  Crear comunicadores comm_fila y comm_columna
  for k := 0 to N-1
    Broadcast en comm_columna de subfila_k de A desde proceso propietario
    Broadcast en comm_fila de subcolumna_k de A desde proceso propietario
    for i := local_i_start to local_i_end − 1
      for j := local_j_start to local_j_end − 1
        Bloque_A[i, j] = min{Bloque_A[i, j], subfila_k[i] + subcolumna_k[j]}
end
```

Distribución inicial de los datos de entrada desde el proceso 0

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 A 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_A);
   /*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_A(comienzo), 1, MPI_BLOQUE,
                          buf_envio,sizeof(int)*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 procesos */
MPI_Scatter (buf_envio, sizeof(int)*tam*tam, MPI_PACKED,
                      buf_recep, tam*tam, MPI_INT, 0, MPI_COMM_WORLD);
```

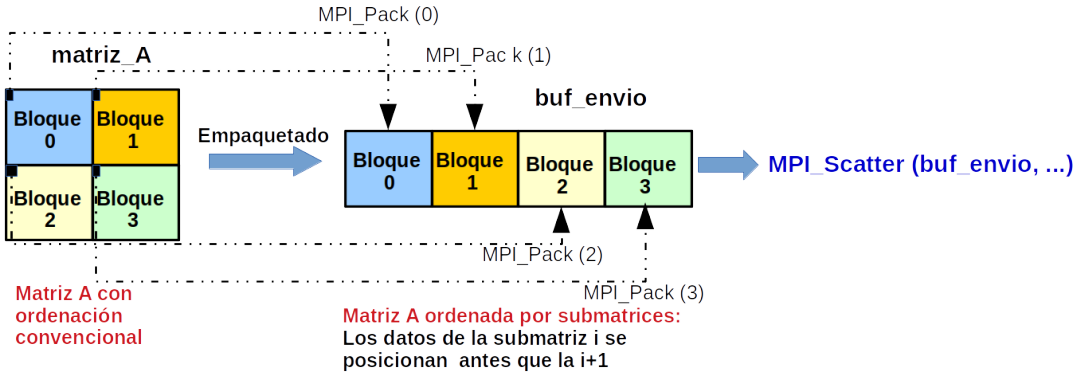


Figura 2.4: Empaquetado de los bloques requeridos por cada proceso para $P = 4$

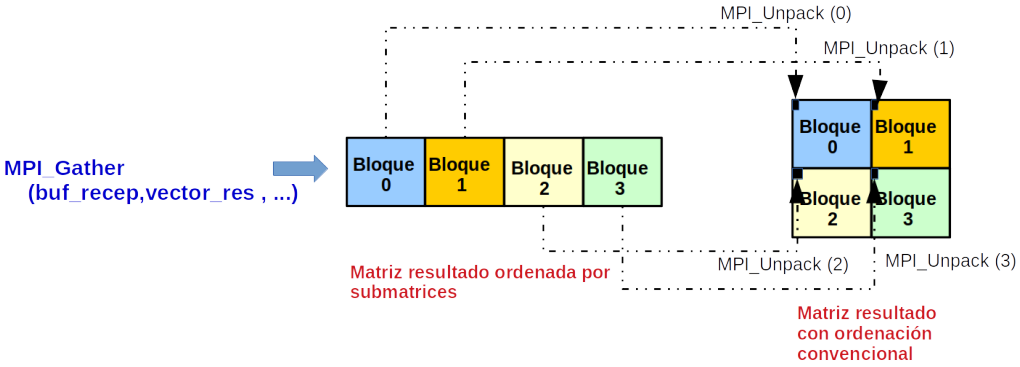


Figura 2.5: Desempaquetado del vector resultado para $P = 4$

La figura 2.4 ilustra este procedimiento de empaquetado de las submatrices requeridas por cada proceso. En cierto modo, este procedimiento es una forma de conseguir copiar la matriz original en un vector (`buf_recep`) donde los datos se encuentran reordenados de forma que los elementos de la submatriz destinada al proceso $rank$ ($rank = 0, 1, ..., P-1$) se ubican justo antes de los elementos de la submatriz destinada al proceso $rank+1$. Esto permite que un `MPI_Scatter` de `buf_recep` con tamaño de bloque igual a N^2/P enteros haga que cada proceso mantenga localmente la submatriz de A que le corresponde.

2.4.2 Recolección de resultados en el proceso 0 y desempaquetado para restaurar ordenación convencional

Para obtener la matriz resultado en el proceso 0, primero se realiza una llamada a la función `MPI_Gather` para obtener, en el proceso 0, un vector con los datos de cada bloque ordenados por número de proceso. Lógicamente, esta no es la ordenación por filas que se desea para los valores de la matriz resultado. Para obtener la matriz resultado definitiva, con la ordenación por filas convencional, se han de realizar, en el proceso 0, P llamadas a la función `MPI_Unpack` para ir desempaquetando cada bloque del vector que almacena los bloques y copiar así sus valores adecuadamente en la matriz resultado. El proceso es similar al seguido para empaquetar los bloques, pero usando `MPI_Unpack` para realizar la operación inversa sobre el vector obtenido para restaurar cada submatriz de la matriz resultado. La figura 2.5 ilustra este procedimiento que resulta necesario si deseamos imprimir los valores de la matriz resultado.

2.5 Ejercicios propuestos.

Se proporcionará en la plataforma PRADO una versión secuencial en C++ del algoritmo de Floyd, y una versión paralela basada en MPI del algoritmo de Floyd paralelo 1D. Cualquiera de estas implementaciones se pueden utilizar como plantilla para programar la implementación paralela basada en una descomposición 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.

1. Implementar usando MPI una versión distribuida del algoritmo de Floyd 2 (bidimensional). Se recomienda desarrollar la implementación fase a fase, y no saltar a otra fase hasta que la fase en desarrollo haya sido probada y chequeada adecuadamente. Si intentáis hacer todo de golpe, fracasaréis o tardaréis mucho más tiempo en tener una versión final correcta. Por ejemplo, se puede empezar realizando la distribución inicial de la matriz de entrada por bloques entre los procesos (lo cual prácticamente se da hecho). Una vez hecho esto y probado con diferentes grafos y número de procesos, se puede abordar la recogida de las submatrices resultado de cada proceso en el proceso 0 y su desempaquetado. Una vez chequeada esta fase, podemos abordar la creación de los comunicadores de fila y columna de la malla de procesos y la difusión de la subfila y la subcolumna para cada valor de k. Cuando se ha comprobado que cada broadcast se realiza correctamente, se puede abordar la actualización de cada celda de la matriz para cada valor de k, i y j.

   Es fundamental entender perfectamente cómo funciona la implementación del Algoritmo 1 (que se proporciona) para poder abordar la implementación del Algoritmo 2.

2. Realizar medidas de tiempo de ejecución sobre los algoritmos disponibles. Deberán realizarse las siguientes medidas:

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

   (b) Medidas para los algoritmos paralelos 1D y 2D ($P = 4, 9$).

   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 han llegado a una determinada fase de cómputo, podemos utilizar `MPI_Barrier`.

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

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

	$T(P=1)$ (sec.)	$T_P(P=4)$	$S(P=4)$	$T_P(P=9)$	$S(P=9)$
$N = 360$					
$N = 720$					
$N = 900$					
$N = 1080$					
$N = 1800$					

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 $P = 9$.

Todos los datos, tablas y gráficas, junto con un análisis con sentido crítico de los mismos, se presentarán en un achivo pdf que acompañará a una carpeta con el código fuente. La carpeta con el código debe incluir un `Makefile` que automatice la obtención del ejecutable.