

# Práctica 2: Algoritmo de Floyd

*Programación Distribuida y Paralela*

---

Juan Pedro Bolívar Puente

15 de diciembre de 2010

## Resumen

El siguiente documento es la memoria de la segunda práctica de la asignatura Programación Distribuida y Paralela, impartida en 4º de Ingeniería Informática en la Universidad de Granada.

En ella se desarrollan tres implementaciones diferentes del algoritmo de Floyd con diferentes grados de paralelismo en un sistema distribuido con MPI. Finalmente, se realiza una comprobación empírica de la eficiencia de dichas implementaciones y se apoya con un modelo teórico del mismo.

## 1. Diseño e implementación

En nuestra implementación de la práctica se ha tomado una aproximación diferente de la propuesta en el guión de prácticas.

Mientras que en éste se nos invita a separar la implementación en tres carpetas diferentes que no comparten código en absoluto, nosotros hemos decidido realizar las tres implementaciones en la misma carpeta, de tal forma que puedan compartir código. La razón fundamental para realizar esto es que, desde el punto de vista didáctico, es mucho mejor ejercitarse en los principios fundamentales del desarrollo de software aplicando la abstracción y evitar la redundancia en el código hasta dónde nos permitan los mecanismos del lenguaje de implementación. En nuestro caso, C++ es un lenguaje muy potente con mecanismos de abstracción muy adecuados para ofrecer una implementación eficiente y abstraída.

En este proceso de abstracción nos hemos percatado de que en realidad los tres algoritmos pueden generalizarse, de tal forma que el algoritmo secuencial y el unidimensional son en realidad especializaciones del algoritmo distribuido bidimensionalmente. El código que implementa el algoritmo de Floyd, generalizado sobre el parámetro plantilla `DistGraph`, es como sigue:

```

1  static void apply (DistGraph& g)
2  {
3      for (size_t k = 0; k < g.size (); k++)
4      {
5          g.bcast_row (k);
6          g.bcast_col (k);
7
8          for (size_t i = g.local_row_begin (); i < g.local_row_end (); ++i)
9              for (size_t j = g.local_col_begin (); j < g.local_col_end (); j++)
10                 g (i, j) = std::min (g (i, j), g (i, k) + g (k, j));
11      }
12 }

```

El código es bastante elegante –muy similar al pseudo-código– y no necesita mayor explicación. No obstante, probablemente esta implementación invita a pensar que esta implementación es más ineficiente que una concreta, especialmente para el algoritmo secuencial y el unidimensionalmente distribuido, ya que se incluyen llamadas a funciones que no son necesarias. Sin embargo, hay que tener en cuenta que el sistema de plantillas de C++ usa *polimorfismo estático*, por lo que, por un lado, se evita la sobrecarga adicional que tendría un diseño orientado a objetos con despacho dinámico<sup>1</sup> y, por otro, el compilador puede expandir las llamadas a funciones, tal forma que no hay que temer que se creen llamadas a funciones triviales como `local_col_begin` que en el caso unidimensional puede substituirse por una constante o `bcast_row` que en el secuencial y el unidimensional debe omitirse. Esta confianza en que la generalización mediante polimorfismo estático no produce pérdida de rendimiento es ampliamente utilizada y es utilizada ampliamente en el *diseño por políticas* ortogonales[1].

Aunque el código está documentado para ser auto-comprensible, vamos a describir las clases principales y las técnicas que hemos aplicado en su implementación.

Por facilitar la reusabilidad, todo el código está en el espacio de nombres `pdp`. Por un lado, la clase `pdp::graph<T>` implementa el grafo como matriz de adyacencia usando `T` como tipo para las etiquetas de las aristas, se encarga de reservar y liberar la memoria y encapsular el acceso a ellas. En el fichero `graph.hpp` también se incluye una clase `pdp::graph_base<T>` que sirve para implementar clases con interfaces similares sin reescribir código. En concreto, estas clases similares son los diferentes grafos distribuidos.

La distribución del grafo entre los diferentes ordenadores se realiza en

---

<sup>1</sup>Una dereferencia y una suma adicional en el caso de la implementación típica con `vtables` en C++ cuando se usa polimorfismo dinámico.

las clases `pdp::full_dist_graph<T>`, que distribuye el grafo a todos los procesos de un comunicador dado, `pdp::uni_dist_graph<T>`, que distribuye el grafo partido unidimensionalmente y `pdp::bi_dist_graph<T>` que distribuye el grafo particionado bidimensionalmente. Estas clases utilizan el patrón RAII (*Resource Acquisition Is Initialization*), esto es, todos los recursos e inicialización se realizan en el constructor y todos se liberan en el destructor. Esto significa que la distribución del grafo se realiza al entrar el ámbito (o *scope*) del grafo distribuido y se reconstruye en el grafo original. Esto nos permite establecer una semántica elegante: el grafo distribuido recibe como parámetro un grafo normal. Mientras el grafo distribuido esté vivo, el grafo normal se considera inválido, ya que sus recursos están siendo utilizados por el grafo distribuido. Cuando finalmente el objeto de grafo distribuido es destruido, el grafo normal es reconstruido en el proceso raíz.

El realizar operaciones colectivas en el destructor puede ocasionar sorpresas, especialmente en presencia de excepciones. Esto explica por qué la API de referencia para C++ de MPI no invoca a `Free()` en el destructor de los diferentes objetos que tienen el método, ya que normalmente desatan operaciones colectivas[2]. No obstante estas situaciones son controlables y revisiones modernas de la API de C++ para MPI como `Boost.MPI`[3] sí utilizan RAII.<sup>2</sup>

Otro aspecto a comentar sobre nuestra implementación es la forma en que distribuimos el grafo. En los dos primeros casos es sencillos y utilizamos mecanismos similares a los propuestos en el guión. No obstante, en el caso distribuido bidimensionalmente usamos una aproximación diferente que tiene algunas ventajas para nuestro caso particular –a parte de que siempre es didáctico investigar por cuenta propia ☺.

En el caso bidimensional se propone empaquetar los datos creando para ello un tipo de dato MPI para la sub-región  $\sqrt{P} \times \sqrt{P}$  que debe recibir cada nodo. Una vez compactados en un búfer, se distribuyen a cada nodo. Esto implica que cada nodo tiene, al menos, dos búferes vivos durante la ejecución del algoritmo, uno para la sub-región propia y otro para el resto del grafo.

En nuestro caso, hemos decidido usar el mismo búfer para todos los datos. Para hacer esto, podríamos realizar el envío de los datos empaquetados

---

<sup>2</sup>Aprovechamos para promocionar la biblioteca `Boost.MPI` para su uso en las prácticas de PDP. Esta biblioteca aporta una interfaz más sencilla, abstraída y moderna para el lenguaje C++. Además, el conjunto de bibliotecas `Boost` están desarrolladas por expertos en C++ muchos de los cuales trabajan en el desarrollo del estándar; muchas de las bibliotecas de éste paquete pasarán a formar parte de la biblioteca estándar de la próxima revisión de C++ que se espera para dentro de unos meses.

y luego desempaquetarlos de nuevo en el búfer principal. Sin embargo, no queríamos usar un búfer temporal y realizar el doble trabajo de empaquetar y desempaquetar los datos. La única alternativa es utilizar el tipo de dato MPI para la subregión, pero como no podemos especificar el entrelazado de las subregiones en el búfer principal al invocar `MPI::Comm::Scatter`, enviamos a cada nodo su trozo individualmente utilizando `MPI::Comm::ISend`. Este método realiza en el envío asíncronamente, con lo cual en muchos casos la eficiencia de nuestro procedimiento es análoga a la de `Scatter` en la mayoría de los casos, aunque existen topologías en las que una implementación de `Scatter` puede proporcionar mejor rendimiento[4]. Antes de terminar la operación de envío se realiza una operación `MPI::Waitall ()` para esperar sobre el conjunto de `MPI::Request` que han devuelto las operaciones asíncronas y evitar que algún nodo reciba información manipulada en las iteraciones de Floyd en el raíz. Al final, en lugar de usar `MPI::Comm::Gather` y desempaquetar luego los datos, realizamos la operación inversa, esto es, una serie de `IRecv` individuales dirigidos a cada nodo del comunicador.

El realizar las cosas así tiene algunas ventajas:

- Aunque puede que las operaciones `Scatter` y `Gather` sean más eficientes en algunas topologías específicas, algunas pruebas informales nos aportan evidencias de que nuestro mecanismo no aumenta especialmente el tiempo de comunicación. Además, el tiempo de computación queda bastante reducido al ahorrarnos las operaciones de empaquetado y desempaquetado.
- El código queda más claro no hay que usar dos búferes diferentes para los datos distribuidos inicialmente y el resto, con lo que eso conlleva en casos esquina a tratar en el código.
- Además, el ahorrarnos un búfer disminuye el consumo espacial de la implementación. Además, hay que tener en cuenta que la localidad espacial de dos búferes diferentes reservados con `new` y `malloc` suele ser muy baja. En nuestra implementación, todos los accesos a memoria en los bucles del algoritmo de Floyd se realizan en la pila o en el último bloque de memoria del grafo, poniéndole la tarea muy sencilla a la caché del procesador.<sup>3</sup>

---

<sup>3</sup>Todo esto depende de cómo se implemente la matriz también. En la matriz propuesta por el profesor se usa un sólo vector y se realiza el indexado bidimensional a mano. En la nuestra, para permitir un indexado más cómodo estilo `_data [i][j]` se inicializan las columnas con el puntero a cada bucle. Sin embargo, usamos un truco para que el vector a `T*` y a `T**` esté en el mismo bloque de memoria.

También, nuestra implementación intenta aprovechar al máximo los recursos de MPI y no empaqueta las columnas al enviarlas, sino que simplemente hace una operación de tipo `Bcast` usando un vector de  $N$  elementos separados con un tamaño de bloque 1 y un *stride* de  $N$ .

También, creo que es interesante mencionar cómo hemos parametrizado los tipos de grafo en función del tipo para las etiquetas `T`. En concreto, existen dos problemas: (a) cada tipo tiene un valor diferente que debe representar el infinito, algunos ni tienen y (b) cada tipo tiene que usar un `MPI::Datatype` diferente en las operaciones con MPI. Una primera opción sería añadir parámetros adicionales a la plantilla, pero esto tiene varios problemas. Por ejemplo, expresiones como `MPI::INT` no pueden ser usadas en un contexto constante. Además, las plantillas de C++ no pueden instanciarse con valores de tipo `float` o `double`<sup>4</sup>.

Sin embargo, existe una técnica, ampliamente usada por la implementación de la STL, que se llama *traits*, que permite asociar constantes y tipos a un tipo [5]. Para poder usar cualquier tipo como etiqueta para los nodos, sólo tenemos que realizar una *especialización* de la clase `pdp::label_traits<T>` y definir los métodos estáticos `infinity()` y `mpi_type()`. Nosotros proporcionamos especializaciones para `int`, `float` y `double`, aunque es trivial extender la biblioteca con otras, y una herramienta para definir automáticamente `infinity()` en función de `std::numeric_limits`.

## 2. Medidas experimentales

A continuación presentamos los resultados de la ejecución de las distintas implementaciones del algoritmo de Floyd propuestas en la práctica, es decir, la ejecución secuencial, la ejecución paralela realizando una descomposición unidimensional y la ejecución paralela realizando una descomposición bidimensional.

Las pruebas fueron realizadas en dos máquinas con procesador *Intel Core Duo* conectadas por un cable cruzado *Ethernet*.

En el Cuadro 1 se muestra un extracto de las mediciones que se realizaron a las tres implementaciones y la ganancia de las soluciones paralelas,  $P = 4$ , con respecto a la secuencial,  $P = 1$ . En la Figura 1 representamos el tiempo de ejecución en función del número de nodos.

---

<sup>4</sup>La lógica detrás de esto, es que el conjunto de tipos diferentes que habría en el sistema depende del redondeo utilizado en tiempo de compilación.

<b>Tiempo</b>	Secuencial	Floyd-1	<i>Ganancia</i>	Floyd-2	<i>Ganancia</i>
$N = 20$	0.00029	0.079921	0.00363	0.00066	0.43939
$N = 40$	0.002404	0.041381	0.05809	0.078056	0.03080
$N = 100$	0.038093	0.054997	0.69263	0.095201	0.40013
$N = 400$	2.43409	0.800081	3,04230	0.766976	3,17362
$N = 800$	19.6013	6.00084	3.26643	6.04182	3.24427
$N = 1000$	38.4677	11.7327	3.27867	11.7789	3.26581
$N = 1500$	129.66	38.9468	3.32916	39.1306	3.31351

Cuadro 1: Extracto de la medición de las implementaciones

Como podemos observar, cuando realizamos la ejecución con un número pequeño de nodos, el algoritmo secuencial realiza los cálculos de manera mucho más rápida que las soluciones paralelas. Sin embargo a medida que se aumenta el número de nodos la ganancia de las soluciones paralelas avanza considerablemente, superando claramente 3 a partir de  $N = 400$ .

Si comparamos los resultados obtenidos por las soluciones paralelas, hasta que no nos encontramos con un número considerable de nodos, la ganancia de la solución unidimensional es ligeramente menor que la bidimensional. Esto se debe a que los tiempos de comunicación son mayores, no obstante, esta segunda solución tiene la ventaja de ser más escalable que la solución unidimensional, ya que mientras que el requisito en la solución unidimensional es  $N$  debe ser múltiplo de  $P$ , en la solución bidimensional,  $N$  debe ser múltiplo de  $\sqrt{P}$ .

### 3. Ejecución Teórica

Como último punto de la práctica se pide el análisis teórico del algoritmo de Floyd suponiendo que las  $P$  tareas se ejecutan sobre  $P$  procesadores conectados todos con todos, asumiendo un hipercubo como topología de conexión. Para cada algoritmo calcularemos el tiempo de **broadcast** y el tiempo de procesamiento.

#### 3.1. Floyd-1, Descomposición Unidimensional

El tiempo de **broadcast** se calcula en función de  $N$ , dimensión de la matriz;  $t_s$ , tiempo de inicialización de envío;  $t_w$ , tiempo de transferencia por entero; y  $P$ , número de procesadores.

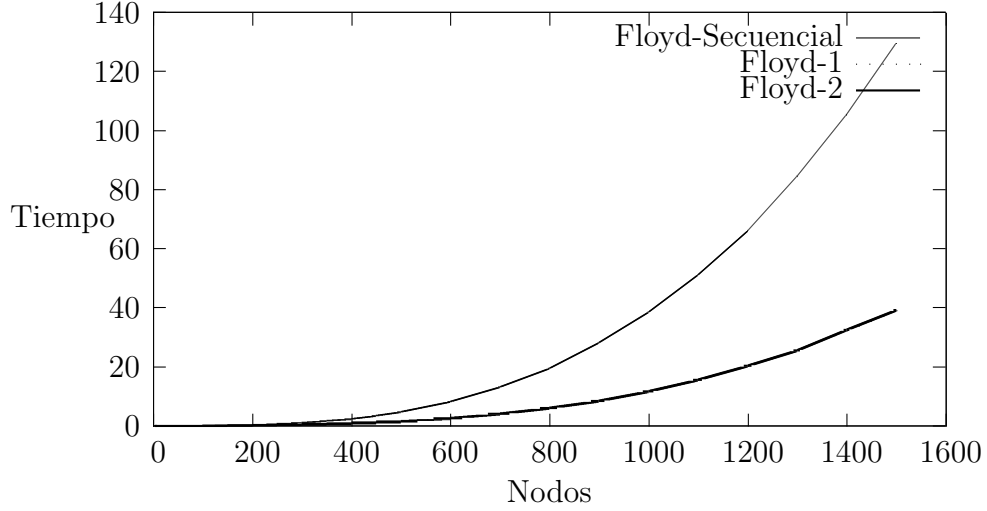


Figura 1: Ejecuciones de las implementaciones del Algoritmo de Floyd

$$T_{Broadcast} = N \cdot (\log_2 P \cdot t_s + \log_2 P \cdot N \cdot t_w)$$

El tiempo de cálculo tiene en cuenta los parámetros  $N$ , dimensión de la matriz;  $C_1$ , tiempo de realización de una operación de mínimo entre dos valores y posterior asignación; y  $P$ , número de procesadores.

$$T_{Calculo} = \frac{N^2}{P} \cdot C_1$$

### 3.2. Floyd-2, Descomposición Bidimensional

En la solución con descomposición bidimensional el tiempo de *broadcast* se reparte entre `bcast_row ()` y `bcast_col ()`. La duración de ambas operaciones *broadcast* es la misma que en el ejemplo anterior, luego siguiendo el razonamiento utilizado en el apartado 3.1, se calculará como,

$$T_{Broadcast} = 2 \cdot N \cdot (\log_2 P \cdot t_s + \log_2 P \cdot \frac{N}{\sqrt{P}} \cdot t_w)$$

Por último el tiempo de cálculo será idéntico al del apartado 3.1. Esto se debe a que cada procesador debe operar la misma cantidad de datos, por ello,

$$T_{Calculo} = \frac{N^2}{P} \cdot C_1$$

## Referencias

- [1] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [2] Jeff Squyres Bill, Bill Saphir Y, and Andrew Lumsdaine Z. The design and evolution of the MPI-2 C++ interface. In *In Proceedings, 1997 International Conference on Scientific Computing in Object-Oriented Parallel Computing, Lecture Notes in Computer Science*. Springer-Verlag, 1997.
- [3] Douglas Gregor and Matthias Troyer. The Boost.MPI library. Web page: [http://www.boost.org/doc/libs/1\\_45\\_0/doc/html/mpi.html](http://www.boost.org/doc/libs/1_45_0/doc/html/mpi.html), 2009.
- [4] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2 edition, January 2003.
- [5] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.