Práctica 1

Implementación del algoritmo de Floyd para obtener el camino más corto.

Jesús Ángel González Novez
76440070F

Tabla de contenidos

Descripción del algoritmo

Este algoritmo nos sirve para obtener el camino más corto entre diferentes puntos. Aquí hay que especificar algunas cosas, consideraremos un punto como un elemento de un grafo, y los caminos que unen dichos puntos son las “líneas” que unen dichos puntos. Estas líneas son de una sola dirección y tienen un coste. Por tanto cuando decimos el camino más corto en realidad estamos refiriéndonos al camino más barato.
Si un punto no tiene camino directo a otro punto, se considera que el coste de ir directamente de ese punto al otro punto es “infinito”, pudiendo existir claro está rutas alternativas para llegar, pero ya pasando por otros puntos.

Solución 1D

Descripción del código

Para inicializar el entorno lo primero que debemos realizar MPI_Init y justo a continuación usar las funciones MPI_Comm_size y MPI_Comm_rank relativas al número de procesos, el comunicador y la id de cada proceso.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
```

A continuación tenemos una serie de variables que nos servirán para indicar el número de vértices, el donde debe comenzar cada proceso en el bucle, el número de filas, y el tamaño del bloque. Además se crea un array para hacer copia del array que tiene internamente el tipo Graph.

```
int nverts,mystart,nfilas,blockSize;;
int * matrizGlobal;
```

Estas inicializaciones las realiza el proceso 0.

```
if(id==0){
   G.lee(argv[1]);
   nverts=G.vertices;
   cout << "EL Grafo de entrada es:"<<endl;
   G.imprime(false);
   matrizGlobal = G.getMatriz();
}
```

Es necesario que todos los procesos conozcan el número de vértices(nverts) por ello hacemos Broadcast.

```
MPI_Bcast(&nverts, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Una vez conocen ese valor ya podemos asignarle a cada uno los valores mystart, nfilas,blocksize.

```
void calculaFilas(int & mystart,int & nfilas,const int & nverts,const int & nprocs,int &
blockSize,const int & id){
    mystart = 0;
    nfilas = nverts / nprocs;
    blockSize = nverts%nprocs;

    if(blockSize>id){
        nfilas++;
        mystart = id*nfilas;
    }
    else{
        mystart = id*nfilas+blockSize;
    }
}
```

El siguiente paso es repartir esa matrizGlobal en pedazos equitativos para cada proceso, de forma que usaremos la función MPI_Scatter que sirve justamente para eso, también necesitamos declarar un array matrizLocal para cada proceso.

```
int *matrizLocal = new int[nverts*nfilas];
MPI_Scatter(matrizGlobal, nfilas*nverts, MPI_INT, matrizLocal, nfilas*nverts, MPI_INT, 0,
MPI_COMM_WORLD);
```

Dado que necesitamos asegurarnos que cada proceso recibió su "porción" pondremos una barrera con MPI_Barrier.

```
MPI_Barrier(MPI_COMM_WORLD);
```

Lo siguiente es ya ejecutar el algoritmo principal, del cual mediremos su tiempo de ejecución para lo que usaremos la función MPI_Wtime.

```
double t=MPI_Wtime();
floyd(nverts,id,nfilas,mystart,matrizLocal);
t=MPI_Wtime()-t;
```

Función floyd

En cada iteración k del algoritmo necesitamos que el proceso actual difunda la fila k, por ello haremos un Broadcast, tras acabar el algoritmo (que por el resto no difiere del original) los datos se irán guardando en una matriz local a cada proceso.

```
void floyd(const int & nverts, const int & id, const int & nfilas, const int & mystart, int
* matrizLocal){
    int *filak = new int[nverts];
    int vijk;
    for(int k=0; k<nverts; k++){
```

```
        if (id == k/nfilas)
          for (int i=0; i<nverts; i++)
            filak[i] = matrizLocal[(k%nfilas)*nverts+i];
        MPI_Bcast(filak, nverts, MPI_INT, k/nfilas, MPI_COMM_WORLD);
        for(int i=0; i<nfilas; i++){
            for (int j=0; j<nverts; j++){
                if (mystart+i!=j && mystart+i!=k && j!=k){
                    vijk = matrizLocal[i*nverts+k] + filak[j];
                    vijk = min(vijk,matrizLocal[i*nverts+j]);
                    matrizLocal[i*nverts+j] = vijk;
                }
            }
        }
    }
}
```

Tras acabar el algoritmo es necesario recolectar todos los datos en la matriz global que se declaró al principio del código, para ello tenemos la función MPI_Gather.

```
MPI_Gather(matrizLocal, nfilas*nverts, MPI_INT, matrizGlobal, nfilas*nverts, MPI_INT, 0,
MPI_COMM_WORLD);
```

Hecho esto ya tendríamos calculado todo, ahora solo queda mostrar los resultados, por comodidad se ha ampliado la funcionalidad de la clase Graph permitiéndonos copiar un array sobre su array interno y así poder usar el método imprime()

```
if(id==0){
        cout << endl<<"EL Grafo con las distancias de los caminos más cortos
es:"<<endl<<endl;
        G.setMatriz(matrizGlobal);
        G.imprime(true);
        cout<< "\nTiempo gastado = " << t << endl;
}
```

Finalmente debemos llamar a la función MPI_Finalize como en cualquier software que use la librería MPI.

```
MPI_Finalize();
```

Modificaciones sobre la clase Graph

Se han añadido dos métodos nuevos:

```
int * Graph::getMatriz(){
    return &A[0];
}
void Graph::setMatriz(int * m){
  A = m;
}
```

Dichos métodos sirven para obtener el array interno de la clase Graph o para sustituirlo por

otro externo a la clase.
Además se ha añadido otro método imprime por comodidad para mi uso, en el que mostrará una pequeña anotación dependiendo de si se trata de mostrar el grafo de entrada o bien el resultado final, para alterna entre uno y otro se usa un parámetro booleano y un if/else simple.

```
void Graph::imprime(bool final){
  if(!final){
    cout << "El coste de ir del nodo i al mismo nodo i es 0" << endl;
    cout << "Si un nodo no puede ir directo a otro nodo su coste es INF" << endl;
    cout << "En esta matriz se indica el coste del viaje directo de un nodo a otros" <<
endl;
    imprime();
  }else{
    cout << "En esta matriz se indica el menor coste posible para ir de un" << endl;
    cout << "nodo a otros" << endl;
    imprime();
  }
}
```

Medidas de tiempo para 1D

El equipo de pruebas ha sido un portatil Intel Dual Core(2xT1600) con Ubuntu 14.04 LTS. Al tener 2 núcleos es normal que con P>2 salgan peores resultados que con P=2, es por ello que se tendrá en cuenta esa columna(P=2) y no la P=4 para medir la ganancia.

	P = 1	**P = 2**	**P = 4**	**Ganancia**
N = 4	0.000004053	0.000016928	0.000183821	**0,24**
N = 60	0.0028398	0.00089407	0.00318003	**3,18**
N = 240	0.25511	0.0511582	0.064996	**3,93**
N = 1024	6.2207	3.38492	3.78931	**1,84**
N = 2048	49.8997	26.006	27.4473	**1,92**

Como vemos para valores muy pequeños como puede ser N=4 tarda más el despachador interno de la librería MPI en gestionar los subprocesos que en lo que tarda en sí mismo el algoritmo, sin embargo conforme crece un poco el valor de N vemos que ya si va siendo una ganancia muy a tener en cuenta.

Solución 2D

Descripción del código

Lo primero es realizar un broadcast del número de vértices a todos los procesos a través del comunicador global.

```
MPI_Bcast(&N,1,MPI_INT, 0,MPI_COMM_WORLD);
```

Lo siguiente que debemos hacer es empaquetar los datos a repartir. Ahora vamos a coger datos no contiguos y agruparlos en un array que si sea contiguo para que los demás procesos puedan trabajar con ellos. Esta tarea debe realizarla el proceso 0.

```
MPI_Datatype MPI_BLOQUE;
int *buf_envio = new int [N*N];
if(rank == 0){
	matriz_I = G.getMatriz();
	MPI_Type_vector(tam , tam, N , MPI_INT, &MPI_BLOQUE);
	MPI_Type_commit(&MPI_BLOQUE);
	for (int i = 0, posicion=0 ;i< P; i++){
		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);
	}
	MPI_Type_free(&MPI_BLOQUE);
}
```

Tras empaquetar la siguiente tarea es la repartición en trozos iguales de los datos a través también del comunicador global.

```
int *buf_recep = new int[tam*tam];
MPI_Scatter(buf_envio, sizeof(int)*tam*tam, MPI_PACKED, buf_recep, tam*tam, MPI_INT, 0,
MPI_COMM_WORLD);
```

Ahora necesitaremos comunicadores para las columnas y comunicadores para las filas, en concreto tendremos tantos como columnas/filas tengan los “paquetes” que antes hemos distribuido.

```
int c_fil, c_col, id_fil, id_col;
c_fil = rank / raiz_P;
c_col = rank % raiz_P;

MPI_Comm_split(MPI_COMM_WORLD, c_col, rank, &comVertical);
MPI_Comm_split(MPI_COMM_WORLD , c_fil, rank, &comHorizontal);
MPI_Comm_rank(comVertical, &id_col);
MPI_Comm_rank(comHorizontal, &id_fil);
```

Descripción Floyd 2D

Ahora ya si pasamos al algoritmo en sí mismo. Ahora trabajaremos con filas locales y columnas locales, ambas deben ser repartidas, pero además como se ha comentado, debemos usar los comunicadores pertinentes en cada caso, bien los verticales bien los horizontales.

```
int *col = new int[tam];
int *fil = new int[tam];

int i, j, k;

MPI_Barrier(MPI_COMM_WORLD);
double t = MPI::Wtime();
for(k=0 ; k < N; k++){
	if (k >= (c_fil * tam) && k < ((c_fil + 1) * tam)){
		for(int jL = 0; jL < tam; jL++){
			fil[jL]= buf_recep[(k % tam)* tam + jL];
		}
	}
	MPI_Bcast( fil, tam, MPI_INT, k/tam, comVertical );
	if (k >= (c_col * tam) && k < ((c_col + 1 )* tam)){
		for(int jL = 0; jL < tam; jL++){
			col[jL]= buf_recep[ jL* tam + (k % tam) ];
		}
	}
	MPI_Bcast( col, tam, MPI_INT, k/tam, comHorizontal );
	for (i=0 ; i < tam; i++){
		for (j= 0; j < tam; j++){
			if (((c_fil * tam) + i) != ((c_col * tam) + j) &&
			((c_fil * tam) + i) != k &&
			((c_col * tam) + j) != k){
				cout << "c_fil*tam+ i:" << c_fil*tam+ i << "
				c_col*tam+j:" << c_col*tam+j <<endl;
				buf_recep[ i*tam +j ] = min(buf_recep[ i*tam+j ],
				                            col[i] + fil[j]);
			}
		}
	}
}

t = MPI_Wtime() - t;
```

Tras acabar el algoritmo debemos recoger los datos como realizamos en la versión unidimensional, esta vez ya volviendo al comunicador global, pero con la diferencia de que recibiremos datos empaquetados.

```
MPI_Gather(buf_recep, tam*tam, MPI_INT, buf_envio, sizeof(int)*tam*tam, MPI_PACKED, 0,
MPI_COMM_WORLD);
```

Como hemos recibido datos empaquetados procedemos a desempaquetar, esta tarea de nuevo debe realizarla el proceso 0 solamente.

```
if(rank == 0){
```

```
        MPI_Type_vector(tam , tam, N , MPI_INT, &MPI_BLOQUE);
        MPI_Type_commit(&MPI_BLOQUE);
        for (int ii = 0, posicion = 0; ii< P; ii++){
                fila_P = ii / raiz_P;
                columna_P = ii % raiz_P;
                comienzo = (columna_P * tam) + (fila_P * tam*tam * raiz_P);
                MPI_Unpack( buf_envio, sizeof(int)*N*N, &posicion, matriz_I + comienzo, 1,
                        MPI_BLOQUE, MPI_COMM_WORLD);
        }
        MPI_Type_free(&MPI_BLOQUE);
}
```

Tras esto ya solo nos queda hacer la llamada a Finalize, mostrar los datos, liberar memoria(free(..)) y fin del código.

Clase Graph

En este caso se mantiene la misma implementación usada en la versión unidimensional, por tanto no hay más que decir respecto a Graph.h y Graph.cc

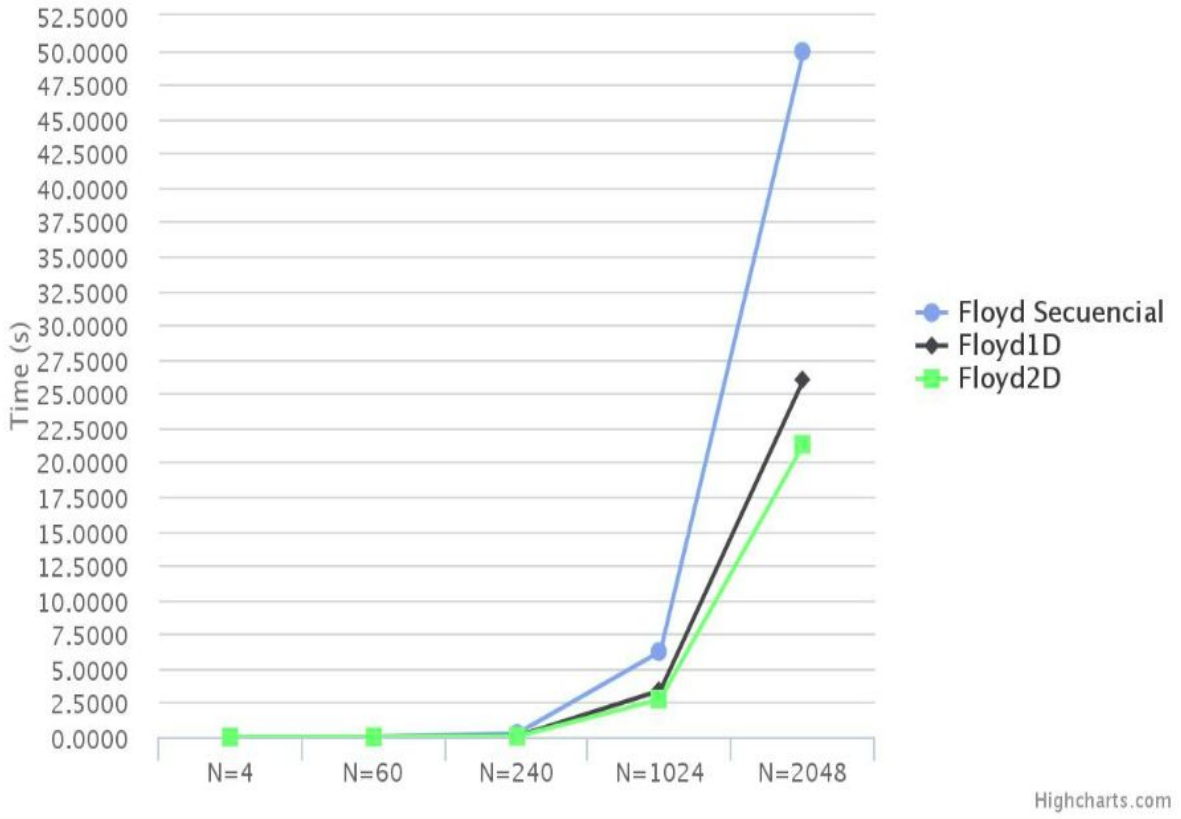
Medidas de tiempo para 2D

En este caso debemos cumplir la regla de que √P sea divisor del número de vértices, esto quiere decir que (*N mod √P == 0*)

	P = 1	**P = 4**	**Ganancia**
N = 4	0.000004053	0.00028586	**0,014**
N = 60	0.0028398	0.00327706	**0,87**
N = 240	0.25511	0.0437021	**5,84**
N = 1024	6.2207	2.75151	**2,26**
N = 2048	49.8997	21.2935	**2,34**

Floyd Secuencial vs Floyd1D vs Floyd2D con gráfica

	Floyd Secuencial	**Floy1D con P=2**	**Floyd2D con P=4**
N = 4	0.000004053	0.000016928	0.00028586
N = 60	0.0028398	0.00089407	0.00327706
N = 240	0.25511	0.0511582	0.0437021
N = 1024	6.2207	3.38492	2.75151
N = 2048	49.8997	26.006	21.2935



Glosario de funciones y tipos MPI usados

Funciones

- Funciones vistas en Floyd 1D
  - `int MPI_Init (int *argc, char ***argv)`
  - `int MPI_Comm_size (MPI_Comm comm, int *size)`
  - `int MPI_Comm_rank (MPI_Comm comm, int *rank)`
  - `int MPI_Finalize()`
  - `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
  - `int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  - `int MPI_Barrier(MPI_Comm comm)`
  - `int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  - `double MPI_Wtime()`
- Funciones vistas en Floyd 2D
  - `int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`
  - `int MPI_Type_commit(MPI_Datatype *datatype)`
  - `int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)`
  - `int MPI_Type_free(MPI_Datatype *datatype)`

- `int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)`
- `int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)`

Tipos de datos

- `MPI_Comm`
  - Este tipo de dato guarda toda la información relevante sobre un comunicador específico. Se utiliza para especificar el comunicador por el que se desea realizar las operaciones de transmisión o recepción. Nótese que hay una constante que define al comunicador global, que es MPI_COMM_WORLD, este no se declara.
  - En el caso de la solución unidimensional sólo usaremos el comunicador global, denominado *MPI_COMM_WORLD*
- `MPI_Datatype`
  - Se utiliza para especificar el tipo de dato de cada elemento que participa en la operación de envío o recepción.