Master Oficial en Sistemas Ingeligentes y Aplicaciones Numéricas en la Ingeniería

Universidad de las Palmas de Gran Canaria

Trabajo para la asignatura: Computacion Paralela

Tareas de paralelización en MPI

Autor: Sergio Marrero Marrero

Tutor de la asignatura:
Eduardo Miguel Rodríguez Barrera

Índice

1. Introducción

El contexto de este documento es la asignatura Computación Paralela del Máster Oficial de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (SIANI)(año académico 2015/2016). Se han realizado como tareas para superar la segunda parte de la asignatura las que se comentan a continuacion.

2. Tarea 1: Ejercicio 3.6 del libro: An Introduction to Parallel Programming.

2.1. Introducción

Lo que se nos pide queda ejemplificado con la imagen 1. En esta se observa que el vector $v = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$ ha sido distribuido de las tres procesos, utilizando tres técnicas de reparto diferentes.

Table 3.4 Different Partitions of a 12-Component Vector among Three Processes

	Components											
									Block-Cyclic			
Process	Block				Cyclic				Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	+6'	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Figura 1: Distintas formas de paralelizar un vector (imagen tomada del libro: *An introduction to Parallel Programming. Autor: Peter S. Pacheco*)

Para resolver estos ejercicios se utilizará la función de MPI: *MPI_Scatterv()*. La interfaz de esta función de MPI es la siguiente:

Listing 1: Interfaz de MPI Scatterv(...)

```
int MPI_Scatterv(const void *sendbuf,
                 const int *sendcounts,
                 const int *displs,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm)
```

La estrategia que se seguirá para los tres apartados será la de componer un vector, cuyas componentes estén colocadas de la manera apropiada.

El objetivo en cada apartado será realizar los procesos pertinentes que nos permita obtener como salida las siguientes variables:

- **sendbuf**: Vector de componentes enteras continente de la información a enviar.
- **sendcounts**: Vector de componentes enteras continente de el tamaño de cada paquete que se le enviará a cada proceso.
- **displs**: Vector de componentes enteras continente de las posiciones de inicio de lectura del vector *sendbuf*, para realizar el envío a cada proceso.

2.2. Apartado a).

En este caso, el vector que se desea repartir se asignará a la variable *sendbuf*, es decir no necesitará ningún tipo de preprocesado. La estrategia que se siguió para calcular las variables *sendcounts* y *displs* queda reflejada en el pseudocódigo 1.

Algorithm 1 Preparacion de vector para Scatterv()

```
1: function PREPARATION(sendcounts(out),displs(out),...)
2:     rem ← dimVector mód nprocs
3:     for i ← 0, nprocs do                          ▷ Iniciamos el reparto
4:         sendcount[i] ← dimVector/nprocs           ▷ Parte entera a cada proceso
5:         if rem > 0 then                           ▷ Solo entramos si queda resto
6:             sendcounts[i] + +;                    ▷ Aumentamos uno a sendcount[i]
7:             rem − −;                              ▷ Restamos uno al resto
8:         end if
9:         displs[i] = sum;                          ▷ Establecemos un comienzo de lectura
10:        sum+ = sendcounts[i];      ▷ Vamos acumulando en sum los que hemos mandado
11:    end for
12: end function
```

2.3. Apartado b).

Se creará un nuevo vector llamado *vectorCiclyc* que recibirá el nuevo vector ordenado. Este nuevo vector se asignará a la variable *sendbuf*. Una vez se tenga este nuevo vector simplemente se seguirán los pasos indicados en el apartado a).

La forma de obtener este vector será la siguiente:

1. Preproceso 1: En esta etapa se creará una matriz de paso que nos facilitará la construcción del vector *vectorCiclyc*. La dimensión del vector en cuestión es *dimVector* , y *nprocs* el

número de procesos que intervengan en el reparto. Para expresar cuantos elementos le tocará a cada proceso simplemente se ordenarán en una matriz, cuyas filas representarán a los procesos, y columnas a los elementos repartidos.

2. Preproceso 2: Este preproceso consistirá en vectorizar la matriz intermedia, de tal forma que se obtenga el vector *vectorCiclyc*.

A continuación se muestran los dos procesos que se acaban de describir. Se ha utilizado como ejemplo un vector de 14 elementos y cuatro procesos. Se puede observar que los dos últimos elementos de la columna cuatro no existen. Esto último habrá que tenerlo en cuenta a la hora de diseñar el algoritmo de la segunda etapa.

$$vector = [v_1, v_2, v_3, ..., v_{14}] \Longrightarrow \begin{bmatrix} v_1 & v_5 & v_9 & v_{13} \\ v_2 & v_6 & v_{10} & v_{14} \\ v_3 & v_7 & v_{11} & \square \\ v_4 & v_8 & v_{12} & \square \end{bmatrix} \Longrightarrow vectorCiclyc = [v_1, v_5, v_9, v_{13}, ...v_4, v_8, v_{12}] \tag{1}$$

Para resolver este problema se propone el pseudocódigo 2. Este pseudocódigo está compuesto por la funcion *vectorCiclycGeneration(...)* que a su vez se divide en los dos procesos (*generateCiclicMatrix(...) y generateVectorCiclyc(...)*) explicados más arriba.

Una vez se ha pasado por este proceso, se puede utilizar el algoritmo principal 1.

2.4. Apartado c).

Este apartado es una combinación de los anteriores. El ejemplo de la imagen 1 muestra como la combinación consiste en agrupar en bloques, y esos bloques distribuirlos de forma cíclica.

La estrategia que se ha seguido para realizar esta distribución es la siguiente:

1. Se construirá una matriz agrupando las filas por bloques. Tendrá tantas columnas como bloques y tantas filas como bloques a repartir. Algunos bloques (filas) podrán quedar incompletas. Ver proceso 2

2. Una vez se ha realizado esto se repartirán estos bloques en la matriz, siguiendo el método cíclico. La matriz resultante quedará como la matriz que aparece en el proceso 3. Hay que tener en cuenta que esta matriz se va rellenando de bloque en bloque, por lo que

Algorithm 2 Cylic Distribution

```
1: function vectorCiclycGeneration(vectorCyclic(out),...)
2:     procedure generateCiclicMatrix(ciclycMatrix(out),...)
3:         for j ← 0, dimVector/nprocs + 1 do                ▷ Columnas de ciclycMatrix
4:             for i ← 0, nprocs do                          ▷ Filas de ciclycMatrix
5:                 if elementosRepartido < dimVector then
6:                     vectorMatrix[i][j] = vector[i + nprocs ∗ j]   ▷ Creamos matriz
7:                 end if
8:             end for
9:         end for
10:    end procedure
11:
12:    procedure generateVectorCiclyc(generateMatrix(in),vectorCiclyc(out),...)
13:        rem ← dimVector mód nprocs
14:        int vectorCiclyc[dimVector];
15:        int catch;                                        ▷ Soluciona el distinto tamaño de filas
16:        if rem > 0 then
17:            catch ← dimVector/nprocs + 1
18:        else
19:            catch ← dimVector/nprocs
20:        end if
21:        for i ← 0, nprocs do
22:            for j ← 0, catch do
23:                vectorCiclyc[posVector] = vectorMatrix[i][j]   ▷ Creamos vectorCiclyc
24:            end for
25:        end for
26:    end procedure
27: end function
```

puede contener espacios en blanco de una forma un tanto más compleja que en el método anterior.

3. Finalmente se vectoriza esta matriz, teniendo cuidado de no coger los huecos en blanco.3

$$vector = [v_1, v_2, v_3..., v_{13}] \Longrightarrow \begin{bmatrix} v_1 & v_2 \\ v_3 & v_4 \\ v_5 & v_6 \\ v_7 & v_8 \\ v_9 & v_{10} \\ v_{11} & v_{12} \\ v_{13} & \square \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} v_1 & v_2 & v_9 & v_{10} \\ v_3 & v_4 & v_{11} & v_{12} \\ v_5 & v_6 & v_{13} & \square \\ v_7 & v_8 & \square & \square \end{bmatrix} \Longrightarrow vectorCiclyc = [v_1, v_2, v_9, v_{10}, ...v_{13}, v_7, v_8] \tag{3}$$

El pseudocódigo 3 muestra como se debería de implementar dicha función. El nombre de esta función es *BlockCiclyc(...)* y a su vez se divide en los dos subprocesos explicados (*generateBlockCiclycMatrix(...) y generateBlockMatrix(...)*). Esta función necesitaría continuar con el subproceso *generateVectorCiclyc(...)* de la función *vectorCiclycGeneration(...)* (ver algoritmo 2) para transformar dicha matriz en un vector y finalmente utilizar la funcion 1.

Algorithm 3 Block-Cylic Distribution

```
1: function BlockCiclyc(...)
2:     procedure generateBlockMatrix(...)
3:         for I ← 0, dimVector/block + 1 do
4:             for j ← 0, j < block do
5:                 if elementosRepartidoS < dimVector then
6:                     blockMatrix[i][j] = vector[j + block ∗ i];          ▷ Creamos matriz
7:                     elementosRepartidos + +;
8:                 end if
9:             end for
10:        end for
11:    end procedure
12:
13:    procedure generateBlockCiclycMatrix(...)
14:        for package ← 0, packageTotal do
15:            for i ← 0, nproc do
16:                for j ← 0, block do
17:                    if elementosRepartido < dimVector then
18:                        BlockCiclycMatrix[i][j+package∗block] = blockMatrix[i+package∗
    nprocs][j];
19:                    end if
20:                end for
21:            end for
22:        end for
23:    end procedure
24: end function
```

3. Tarea 2: Ejercicio 3.13 del libro: An Introduction to Parallel Programming.

El código de esta tarea se adjunta dentro de la carpeta de la entrega. Es el archivo: *tarea2.c*.

3.1. Experimentos realizados con el código paralelizado

Se realizó el siguiente experimento:

Se calcularon los tiempos de trabajo usando la función *MPI_Wtime()*. Para ello, se varió el tamaño del vector tal que: $n = [100, 1000, 10000, 100000]$.

Para cada tamaño del vector, se tomaron los tiempos en realizar el producto escalar de ambos vectores, desde el momento en el que comienza la paralelización hasta que se obtiene el producto escalar. Esta tarea se llevó a cabo con distintos números de procesos, con el fin de compararlos. El número de procesos con que el que se llevó a cabo fue: $nprocs = [1, 2, 4, 8, 16]$. Con el fin de calcular la media, cada experimento se realizó tres veces para cada proceso y cada tamaño del vector. Los resultados de estos experimentos se muestran en las tabla 1 , tabla 2,tabla 3 y tabla 4.

Finalmente se muestran estos resultados en la figura 2. Para una mejor visualización de los resultados, la imagen fue realizada con ejes logarítmicos.

Tablas con los experimentos.

dimVector	Procesos				
100	1	2	4	8	16
100	0.0041	0.1	0.28	1.2	2.09
100	0.0041	0.107	0.22	0.889	2.5
100	0.005	0.115	0.31	1.185	2.363
media	**0.0045**	**0.107**	**0.27**	**1.092**	**2.317**

Tabla 1: Resultados. Número de elementos: 100. Tiempo en milésimas de segundo

dimVector	Procesos				
1000	1	2	4	8	16
1000	0.01	0.106	0.116	0.218	2.168
1000	0.005	0.094	0.129	0.359	3.513
1000	0.0078	0.133	1.144	0.295	2.092
media	**0.0076**	**0.111**	**0.13**	**0.29**	**2.59**

Tabla 2: Resultados. Número de elementos: 1000. Tiempo en milésimas de segundo

dimVector	Procesos				
10000	1	2	4	8	16
10000	0.0569	0.240	0.268	0.66	1.95
10000	0.063	0.16	0.403	0.65	1.67
10000	0.065	0.258	0.222	0.643	1.41
media	**0.062**	**0.22**	**0.298**	**0.651**	**1.68**

Tabla 3: Resultados. Número de elementos: 10000. Tiempo en milésimas de segundo

dimVector	Procesos				
100000	1	2	4	8	16
100000	0.59	1.047	0.967	1.694	13.62
100000	0.594	1.07	1.01	1.49	10.006
100000	0.66	0.945	1.051	1.053	8.905
media	**0.615**	**1.02**	**1.01**	**1.41**	**10.84**

Tabla 4: Resultados. Número de elementos: 100000. Tiempo en milésimas de segundo

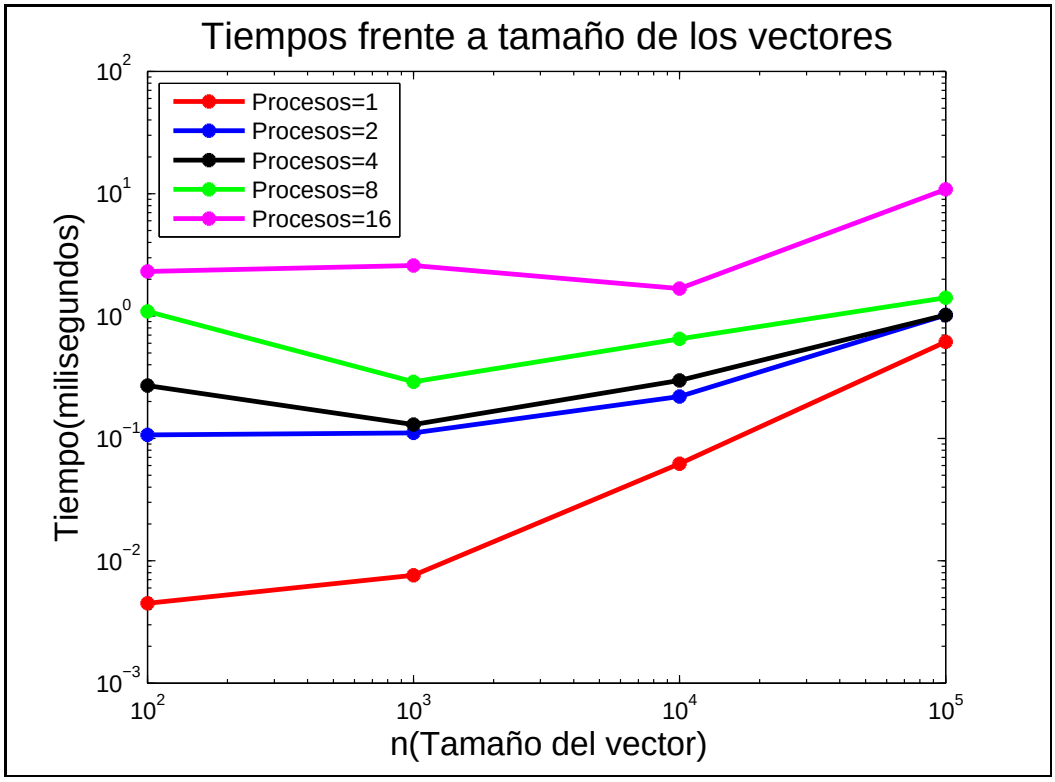

Figura 2: Resultados