Arquitectura de Computadores
Grado en Ingeniería Informática
Práctica 4

Jose Francisco Romero Rodríguez

Sara Marcos Cornejo

UNIVERSIDAD NEBRIJA

Resolución de problemas mediante el modelo de comunicación colectivo

Introducción

La presente práctica continúa profundizando en el modelo de comunicación colectivo que fue presentado en la práctica anterior. Algunas de las ventajas que supondría usar este tipo de comunicación frente a la comunicación punto a punto serían:

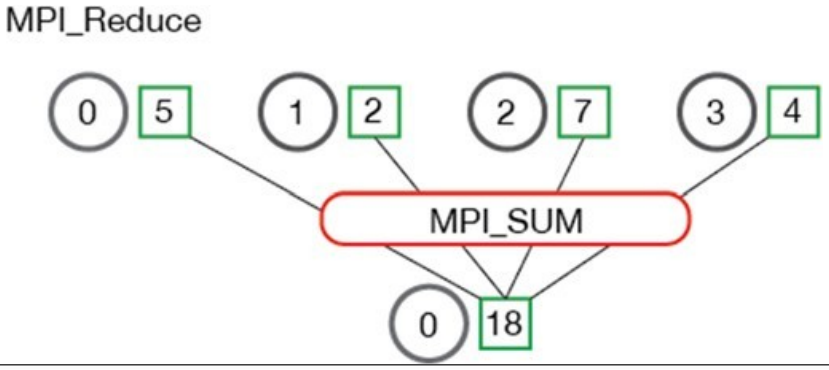
- Se reduce la posibilidad de error
- Código fuente más fácil de leer
- Suele ser más rápida (está más optimizada que su equivalente usando rutinas punto a punto)

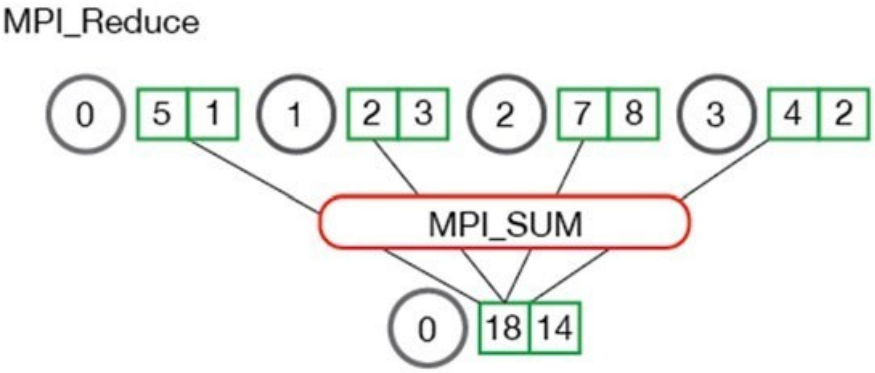
Se trabajará con las funciones: MPI_Reduce(), MPI_Allgather y MPI_Alltoall(), explicadas a continuación.

- Reduce

Realiza una operación de reducción global (como puede ser calcular el máximo, la suma, hacer un AND lógico, etc) sobre cada uno de los miembros del grupo. Se puede notar que únicamente se especifica una vez el número de datos y el tipo de datos, esto es porque el número de datos es el mismo tanto en la emisión como en la recepción, al igual que el tipo. Si el número de datos es mayor que uno, la operación de reducción se realiza uno a uno cada elemento con su posición correspondiente (es decir, el elemento 1 de envío se reduce con cada elemento en la posición 1 de cada sendbuf).

```
MPI_Reduce(void* send_data, void* recv_data, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
communicator)
```

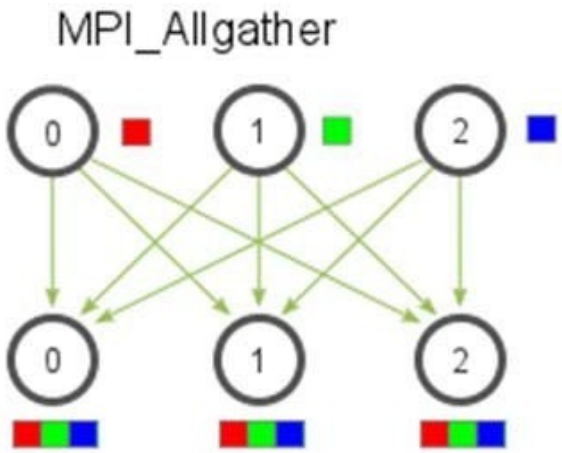




- MPI_Allgather

Dado un conjunto de datos distribuidos entre los diferentes nodos del comunicador, todos los datos se recogerán en todos los nodos. MPI_Allgather es un MPI_Gather seguido por un MPI_Bcast.

```
MPI_Allgather( void* send_data, int send_count, MPI_Datatype
send_datatype, void* recv_data, int recv_count, MPI_Datatype
recv_datatype, MPI_Comm communicator)
```




```
rank    send buf                        recv buf
----    --------                        --------
 0      a,b,c          MPI_Allgather    a,b,c,A,B,C,#,@,%
 1      A,B,C         ---------------->  a,b,c,A,B,C,#,@,%
 2      #,@,%                           a,b,c,A,B,C,#,@,%
```

- MPI_Alltoall

Funciona como MPI_Scatter y MPI_Gather combinados. El búfer de envío en cada proceso se divide como en MPI_Scatter y luego cada columna de fragmentos se recopila mediante el proceso respectivo, cuyo rango coincide con el número de la columna de fragmentos.

```
int MPI_Alltoall(const void *send_data, int send_count, MPI_Datatype
send_datatype, void *recv_data, int recv_count, MPI_Datatype
recv_datatype, MPI_Comm communicator)
```

```
rank    send buf                        recv buf
----    --------                        --------
 0      a,b,c          MPI_Alltoall     a,A,#
 1      A,B,C         ---------------->  b,B,@
 2      #,@,%                           c,C,%

(a more elaborate case with two elements per process)

rank    send buf                        recv buf
----    --------                        --------
 0      a,b,c,d,e,f    MPI_Alltoall     a,b,A,B,#,@
 1      A,B,C,D,E,F   ---------------->  c,d,C,D,%,$
 2      #,@,%,$,&,*                     e,f,E,F,&,*
```

Objetivos

En esta práctica se aprenderá el concepto de comunicación colectivo entre procesos de MPI.

Los objetivos fijados son los siguientes:

- Compilación de programas MPI.
- Ejecución de programas en varios procesos de forma paralela.
- Estructura de un programa MPI.
    - Iniciar y finalizar el entorno MPI con MPI_Init y MPI_Finalize.
    - Identificador (rango) del proceso con MPI_Comm_rank.
    - Consultar el número de procesos lanzados con MPI_Comm_size.
- Aprender operaciones de comunicación colectivas:
    - MPI_Allgather
    - MPI_Alltoall

Compilación y ejecución de programas MPI

- Compilación: `mpicc codigo_fuente.c -o ejecutable`
- Ejecución: `mpirun –np <number> ejecutable`

Ejercicio 1 (1 punto)

Crear un ejemplo sencillo de uso de cada primitiva explicada en esta práctica (MPI_Reduce, MPI_Allgather y MPI_Alltoall) y utilizarlo para explicar el funcionamiento de cada una de ellas. Estos ejemplos deben funcionar correctamente con cualquier número de procesos.

MPI_Reduce:

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{

    int my_rank,size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Cada proceso manda su rank al proceso 0
    int reduction_result = 0;
    MPI_Reduce(&my_rank, &reduction_result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if(my_rank ==0){
        printf("La suma de todos los ranks es %d.\n", reduction_result);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Para este ejemplo se utiliza la función MPI_reduce para realizar la suma del rank de cada proceso.
En primer lugar se definen las funciones MPI_Init(), MPI_Comm_rank() y MPI_Comm_size() necesarias para empezar a desarrollar el código.

Con la función MPI_reduce cada proceso envía su respectivo rank al proceso 0, que ejecuta la suma y por último lo imprime por pantalla.
De tal forma, que si se desean crear 4 procesos:

(Proceso 0: envía rank 0)
(Proceso 1: envía rank 1)
(Proceso 2: envía rank 2)
(Proceso 3: envía rank 3)

La suma sería: 0+1+2+3 = 6

```
sara@sara-vivobook:~/Downloads$ mpicc pract4ej1_AC.c -o ejecutable_ej1
sara@sara-vivobook:~/Downloads$ mpirun -np 4 ./ejecutable_ej1
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
La suma de todos los ranks es 6.
sara@sara-vivobook:~/Downloads$
```

MPI_Allgather:

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int size, my_rank;
    int buffer[size];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);


    printf("Proceso %d, valor: %d\n", my_rank, my_rank);

    MPI_Allgather(&my_rank, 1, MPI_INT, buffer, 1, MPI_INT, MPI_COMM_WORLD);
    printf("El proceso %d tiene los valores:\n", my_rank);
    for (int i = 0; i < size; i++) {
        printf("%d\n", buffer[i]);
    }

    MPI_Finalize();

    return 0;
}
```

Para la realización de este ejemplo se emplea la función MPI_Allgather, mediante la cual se distribuye el rank de los respectivos procesos entre todos los nodos del comunicador.
En primer lugar se definen las funciones MPI_Init(), MPI_Comm_rank() y MPI_Comm_size() necesarias para empezar a desarrollar el código.
Seguidamente, se hace uso de MPI_Allgather, enviando el rank de cada proceso creado a todos los nodos del comunicador, de esta forma:

```
sara@sara-vivobook:~/Downloads$ mpicc pract4ej1b_AC.c -o ejecutable_ej1
sara@sara-vivobook:~/Downloads$ mpirun -np 4 ./ejecutable_ej1
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
Proceso 0, valor: 0
Proceso 1, valor: 1
Proceso 2, valor: 2
Proceso 3, valor: 3
El proceso 0 tiene los valores:
0
1
2
3
El proceso 2 tiene los valores:
0
1
2
3
El proceso 1 tiene los valores:
0
1
2
3
El proceso 3 tiene los valores:
0
1
2
3
sara@sara-vivobook:~/Downloads$
```

MPI_Alltoall:

```c
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int size,my_rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);


    int my_values[size];
    for(int i = 0; i < size; i++)
    {
        my_values[i] = my_rank + i * 100;
    }
    printf("Valores del proceso %d:\n", my_rank);
    for(int i = 0; i < size; i++)
    {
        printf("%d\n", my_values[i]);
    }


    int buffer_recv[size];
    MPI_Alltoall(&my_values, 1, MPI_INT, buffer_recv, 1, MPI_INT, MPI_COMM_WORLD)
    printf("Valores recogidos en el proceso %d:\n", my_rank);
    for(int i = 0; i < size; i++)
    {
        printf("%d,", buffer_recv[i]);
    }
    printf("\n");

    MPI_Finalize();

    return 0;
}
```

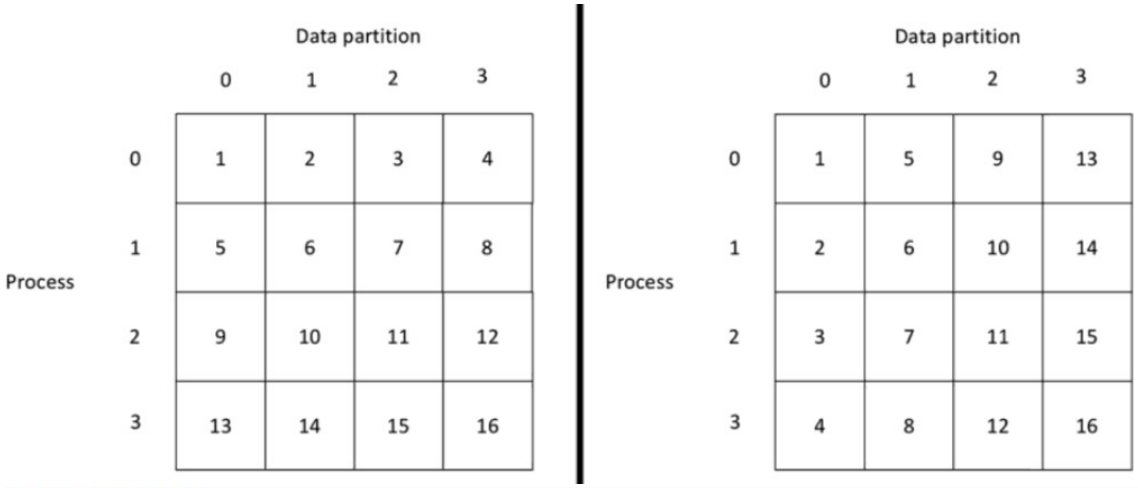
Para este ejemplo se utiliza MPI_Alltoall enviando datos de todos los procesos a todos los procesos.
En primer lugar se definen las funciones MPI_Init(), MPI_Comm_rank() y MPI_Comm_size() necesarias para empezar a desarrollar el código.
Después se asigna los valores deseados al buffer de envío medante un bucle for.
Por último, con MPI_Alltoall, el búfer de envío en cada proceso se divide como en MPI_Scatter y luego cada columna de fragmentos es recopilada por el proceso respectivo, cuyo rank coincide con el número de la columna de fragmentos.

De tal forma que la salida por consola sería la siguiente:

```
Valores del proceso 0:
0
100
200
300
400
Valores del proceso 1:
1
101
201
301
401
Valores del proceso 3:
3
103
203
303
403
Valores del proceso 2:
2
102
202
302
402
Valores del proceso 4:
4
104
204
304
404
Valores recogidos en el proceso 0:
0,1,2,3,4,
Valores recogidos en el proceso 1:
100,101,102,103,104,
Valores recogidos en el proceso 3:
300,301,302,303,304,
Valores recogidos en el proceso 2:
200,201,202,203,204,
Valores recogidos en el proceso 4:
400,401,402,403,404,
sara@sara-vivobook:~/Downloads$
```

Ejercicio 2 (3 puntos)

Implementar un programa que realice la transposición de la matriz inicial mostrada en la figura (parte izquierda). Realizar el proceso en paralelo, distribuyendo la matriz entre 4 procesos.




```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>


int main(int argc, char *argv[]) {

    int rank,numProcs;
    int sendbuf[4], recvbuf[4];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (numProcs != 4) {
    printf("El numero de procesos tiene que ser 4.\n");
    }else{

        for (int i = 0; i < numProcs; i++) {
                    sendbuf[i] = (i + 1) + rank * numProcs;
            }
           printf("Proceso %d: ", rank);
              for (int i = 0; i < numProcs; i++) {
                printf("%d, ", sendbuf[i]);
            }
            printf("\n");

            MPI_Alltoall(&sendbuf, 1, MPI_FLOAT, &recvbuf, 1, MPI_INT, MPI_COMM_WORLD);

            printf("(Transposición) Proceso %d: ", rank);
            for (int i = 0; i < numProcs; i++) {
                printf("%d, ", recvbuf[i]);
            }
            printf("\n");

        }
    MPI_Finalize();
    return 0;
    }
```

Para la realización de este ejercicio se hace uso de la primitiva **MPI_Alltoall.**

En primer lugar se definen las funciones **MPI_Init(), MPI_Comm_rank() y MPI_Comm_size()** necesarias para empezar a desarrollar el código.

Se gestiona el caso en el que no se haya introducido 4 como número de procesos a crear.

A continuación se realiza la asignación de los valores del array mediante un bucle

for, de acuerdo con la imagen propuesta, concordando con la siguiente fórmula:

sendbuf[i]=(i+1)+rank * numProcs

Se imprimen estos valores iniciales, y a continuación se emplea MPI_Alltoall.

Como se explicó antes, esta primitiva también puede verse como una operación de transposición global, que actúa sobre fragmentos de datos, intercambiando los datos de todos los procesos con todos los demás procesos.

Por último se imprime por procesos la matriz final,finalizando la ejecución con **MPI_Finalize().**

La salida por consola sería la siguiente:

```
sara@sara-vivobook:~/Downloads$ mpicc pract4ej2_AC.c -o ejecutable_ej1
sara@sara-vivobook:~/Downloads$ mpirun -np 4 ./ejecutable_ej1
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
Proceso 0: 1, 2, 3, 4,
(Transposición) Proceso 0: 1, 5, 9, 13,
Proceso 1: 5, 6, 7, 8,
(Transposición) Proceso 1: 2, 6, 10, 14,
Proceso 2: 9, 10, 11, 12,
(Transposición) Proceso 2: 3, 7, 11, 15,
Proceso 3: 13, 14, 15, 16,
(Transposición) Proceso 3: 4, 8, 12, 16,
sara@sara-vivobook:~/Downloads$
```

Ejercicio 3 (6 puntos)

Conversión a decimal un número binario.
Haciendo uso de las funciones explicadas en este guion, implementar un programa donde los procesos calculan en paralelo la representación decimal de un número binario:

- El número de procesos con el que se lance la aplicación será igual que el número de bits del número binario.

- Al iniciar la aplicación cada proceso genera un valor aleatorio (0 o 1) y se lo envía al resto. Uno de los procesos deberá imprimir por pantalla el número que se va a pasar a decimal.

- Una vez que todos los procesos tienen el dato, hacen las operaciones necesarias para calcular el valor decimal de forma paralela.

- Solo un proceso conoce el resultado final en decimal y lo imprime por pantalla. Ejemplo de salida por consola con 2, 6 y 10 procesos:

Fig 1: Salida por consola
ejercicio 2

```
nebrija@nebrija:~/Extraordinaria/practica4$ mpicc e1.c -o e1 -lm
nebrija@nebrija:~/Extraordinaria/practica4$ mpirun -np 2 ./e1
Proceso 1, dato: 1
Proceso 0, dato: 0
Número binario: 01
Soy el proceso 0 y el número en decimal es: 1
nebrija@nebrija:~/Extraordinaria/practica4$ mpirun -np 6 ./e1
Proceso 5, dato: 1
Proceso 4, dato: 0
Proceso 0, dato: 1
Proceso 1, dato: 1
Proceso 2, dato: 0
Proceso 3, dato: 1
Número binario: 110101
Soy el proceso 0 y el número en decimal es: 53
nebrija@nebrija:~/Extraordinaria/practica4$ mpirun -np 10 ./e1
Proceso 0, dato: 1
Proceso 7, dato: 1
Proceso 6, dato: 0
Proceso 5, dato: 1
Proceso 1, dato: 1
Proceso 4, dato: 0
Proceso 9, dato: 1
Proceso 8, dato: 1
Proceso 2, dato: 0
Proceso 3, dato: 0
Número binario: 1100010111
Soy el proceso 0 y el número en decimal es: 791
nebrija@nebrija:~/Extraordinaria/practica4$
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<math.h>

int main(int argc, char *argv[]) {
        int i, numProcs, rank, digito, decimal;
        int array[numProcs];
        MPI_Init(&argc, &argv);

        MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        srand(time(NULL)+rank);
        int aleatorio = rand() % 2;

        //se lo envía al resto de procesos(MPI_Allgather)
        MPI_Allgather(&aleatorio, 1, MPI_INT, array, 1, MPI_INT, MPI_COMM_WORLD);

        for(i=0;i<numProcs;i++){
            if(rank==i){
                printf("Soy el proceso %d y he generado el valor %d\n", rank, aleatorio);
            }s
        }

        if (rank == numProcs-1) {
                printf("El número binario es: ");
        for (i = 0; i < numProcs; i++)
                printf("%d", array[i]);
                printf("\n");
        }

        int potencia = pow(2, rank);
        int multiplicacion = potencia * array[numProcs - rank - 1];
        MPI_Reduce(&multiplicacion, &decimal, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

        if (rank == 0) {
                printf("El número decimal es: %d", decimal);
        }
MPI_Finalize();
return 0;
}
```

Para la realización de este ejercicio se hace uso de la primitiva **MPI_Allgather.**

En primer lugar se definen las funciones **MPI_Init(), MPI_Comm_rank() y MPI_Comm_size()** necesarias para empezar a desarrollar el código.

Para generar los números aleatorios se utiliza la semilla srand(time). Para que no sea siempre el mismo valor y vaya cambiando, la instrucción a utilizar sería: srand(time(NULL)+rank)

Haciendo de módulo de 2 para obtener números del 0 al 1:int num = rand() % 2;

A continuación se realiza un for a través del cual se distribuyen todos los datos en todos los nodos. Esto se consigue gracias a la primitiva **MPI_Allgather,** almacenando estos números aleatorios en el array creado.

Se imprime el número binario generado recorriendo el array.

Para realizar el paso del número binario a decimal se siguen los siguientes pasos:

1. Cada proceso calcula su potencia de 2 correspondiente

2. Cada proceso multiplica su potencia de 2 por el valor que le ha llegado

3. Cada proceso suma el resultado de su multiplicación con el resto de resultados a través de la primitiva **MPI_Reduce**

Por último se muestra por pantalla, finalizando la ejecución con **MPI_Finalize().**

La salida por consola (para 4 procesos) sería la siguiente:

```
El número decimal es: 1sara@sara-vivobook:~/Downloads$ mpirun -np 4 ./ejecutable_ej1
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
Soy el proceso 0 y he generado el valor 1
Soy el proceso 1 y he generado el valor 0
Soy el proceso 2 y he generado el valor 0
Soy el proceso 3 y he generado el valor 0
El número binario es: 1000
sara@sara-vivobook:~/Downloads$
```

Conclusiones:

En esta práctica se ha podido profundizar acerca de las comunicaciones colectivas entre procesos (enfocándose más en las utilidades de las primitivas MPI_Reduce(), MPI_Allgather y MPI_Alltoall()), contrastándolo con las comunicaciones punto a punto trabajadas en prácticas anteriores

Bibliografía:

https://www.cfm.brown.edu/faculty/gk/APMA281A/LECTURES/Lec06.ppt