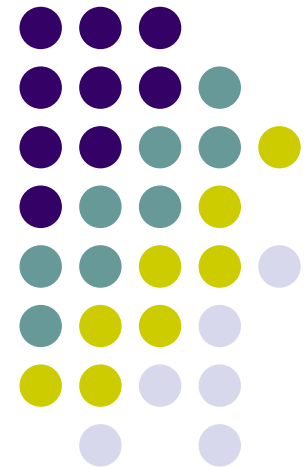


Programación Paralela y Computación de Altas Prestaciones Programación con MPI

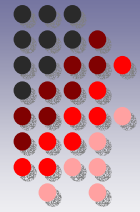
Javier Cuenca



Universidad de Murcia

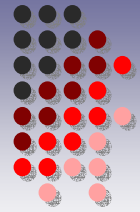


Introducción



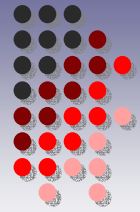
- Previamente PVM: Parallel Virtual Machine
- MPI: Message Passing Interface
- Una especificación para paso de mensajes
- La primera librería de paso de mensajes estándar y portable
- Por consenso MPI Forum. Participantes de unas 40 organizaciones
- Acabado y publicado en mayo 1994. Actualizado en junio 1995
- MPI2, HeteroMPI, FT-MPI

Introducción. ¿Qué ofrece?

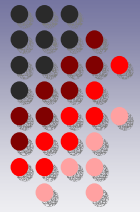


- Estandarización
- Portabilidad: multiprocesadores, multicomputadores, redes, heterogéneos, ...
- Buenas prestaciones, ..., si están disponibles para el sistema
- Amplia funcionalidad
- Implementaciones libres (mpich, lam, ...)

Introducción. Procesos



- Programa MPI: conjunto de procesos autónomos
- Cada proceso puede ejecutar código diferente
- Procesos comunican vía primitivas MPI
- Proceso: secuencial o multithreads
- MPI no provee mecanismos para situar procesos en procesadores. Eso es misión de cada implementación en cada plataforma
- MPI 2.0:
 - Es posible la creación/borrado de procesos dinámicamente durante ejecución
 - Es posible el acceso a memoria remota
 - Es posible la entrada/salida paralela



Introducción.

Ejemplo: hello.c

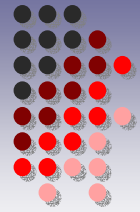
```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char*argv[]) {

    int name, p, source, dest, tag = 0;
    char message[100];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&name)
    ;
    MPI_Comm_size(MPI_COMM_WORLD,&p);
```

Introducción.

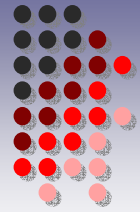
Ejemplo: hello.c



```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
    if (argc != 2)
    {
        printf("Processor %d of %d\n", 0, argc);
        return 0;
    }
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank != 0)
    {
        printf("Processor %d of %d\n", rank, size);
        sprintf(message, "greetings from process %d!", rank);
        dest = 0;
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, 0, MPI_COMM_WORLD);
    }
    else
    {
        printf("processor 0, p = %d \n", 0);
        for(source=1; source < size; source++)
        {
            MPI_Recv(message, 100, MPI_CHAR, source, 0, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }
    MPI_Finalize();
}
```

Introducción.

Ejemplo de uso



- Compilación

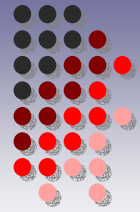
- `mpicc -o ejemplo ejemplo.c -lm -lmpi`

- Ejecución

- `mpirun -np 4 ejemplo`

Introducción.

Ejemplo de uso



- Fichero cabecera:
`#include <mpi.h>`
- Formato de las funciones:
`error=MPI_nombre(parámetros ...)`
- Inicialización:
`int MPI_Init (int *argc , char **argv)`
- Comunicador: Conjunto de procesos en que se hacen comunicaciones
`MPI_COMM_WORLD` , el mundo de los procesos MPI

Introducción.

Ejemplo de uso



- Identificación de procesos:

```
MPI_Comm_rank ( MPI_Comm comm , int *rank)
```

- Procesos en el comunicador:

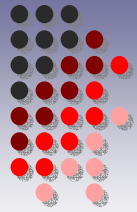
```
MPI_Comm_size ( MPI_Comm comm , int *size)
```

- Finalización:

```
int MPI_Finalize ( )
```

Introducción.

Ejemplo de uso



- MENSAJE: Formado por un cierto número de elementos de un tipo MPI
- Tipos MPI Básicos:

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

- Tipos MPI Derivados: los construye el programador

Introducción.

Ejemplo de uso



- Envío:

```
int MPI_Send ( void *buffer , int contador , MPI_Datatype  
               tipo , int destino , int tag , MPI_Comm comunicador )
```

- Recepción:

```
int MPI_Recv ( void *buffer , int contador , MPI_Datatype  
               tipo , int origen , int tag , MPI_Comm comunicador ,  
               MPI_Status *estado)
```

MPI_ANY_TAG

MPI_ANY_SOURCE



Tipos de comunicación

- Envío:
 - Envío síncrono: `MPI_Ssend`
Acaba cuando la recepción empieza
 - Envío con buffer: `MPI_Bsend`
Acaba siempre, independiente del receptor
 - Envío estándar: `MPI_Send`
Síncrono o con buffer
 - Envío "ready": `MPI_Rsend`
Acaba independiente de que acabe la recepción
- Recepción: `MPI_Recv`
Acaba cuando se ha recibido un mensaje.



Comunicación asíncrona (nonblocking)

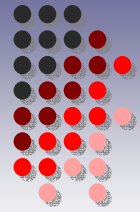
- `MPI_Isend(buf, count, datatype, dest, tag, comm, request)`
- `MPI_Irecv(buf, count, datatype, source, tag, comm, request)`

Parámetro `request` para saber si la operación ha acabado

- `MPI_Wait()`
vuelve si la operación se ha completado. Espera hasta que se completa
- `MPI_Test()`
devuelve un flag diciendo si la operación se ha completado

Comunicación asíncrona (nonblocking).

Ejemplo: hello_nonblocking.c (1/2)



```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&name);
MPI_Comm_size(MPI_COMM_WORLD,&p);

p_requests = (MPI_Request *) malloc ( p * sizeof(MPI_Request) );

if (name != 0)
{
    sprintf(message,"greetings from process %d!",name);
    dest = 0;
    MPI_Isend(message,  strlen(message)+1,  MPI_CHAR,  dest,  tag,
    MPI_COMM_WORLD, &request);
    printf("Procesador %d ya ha hecho el ISEND al procesador
    0\n",name);

    /* ... Código por aquí enmedio ...*/

    MPI_Wait(&request,&status);
    printf("Procesador %d ya ha pasado el WAIT tras
    envio\n",name);
}
```



Comunicación asíncrona (nonblocking). Ejemplo: hello_nonblocking.c (2/2)

```
else
{
    for(source=1; source < p; source++)
    {
        MPI_Irecv(messages[source],100,MPI_CHAR,MPI_ANY_SOURCE,tag,
        MPI_COMM_WORLD, &p_requests[source]);
        printf("Proc. 0 ya ha hecho IRECV para recibir de
        source=%d\n\n",source);
    }

    /* ... Código por aquí enmedio ...*/

    for(source=1; source < p; source++)
    {
        MPI_Wait(&p_requests[source],&status);
        printf("Tras el Wait del Receive: %s\n",messages[source]);
    }
}
free(p_requests);
MPI_Finalize();
}
```

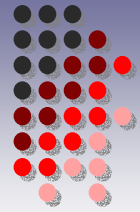


Comunicaciones colectivas

- MPI_Barrier()
bloquea los procesos hasta que la llaman todos
- MPI_Bcast()
broadcast del proceso raíz a todos los demás
- MPI_Gather()
recibe valores de un grupo de procesos
- MPI_Scatter()
distribuye un buffer en partes a un grupo de procesos
- MPI_Alltoall()
envía datos de todos los procesos a todos
- MPI_Reduce()
combina valores de todos los procesos
- MPI_Reduce_scatter()
combina valores de todos los procesos y distribuye
- MPI_Scan()
reducción prefija (0,...,i-1 a i)

Comunicaciones colectivas.

broadcast.c



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

if (my_rank == 0)
{
    printf("Introduce el dato: a (float): "); scanf("%f", &a);
    printf("Introduce el dato: b (float): ");
    scanf("%f", &b);
    printf("Introduce el dato: n (entero): ");
    scanf("%d", &n);
}

MPI_Bcast(&a, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
MPI_Bcast(&b, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, root, MPI_COMM_WORLD);

if (my_rank != 0)
{
    printf("En procesador %d, los datos recibidos son a:%f b:%f\n", my_rank, a, b, n);
}

MPI_Finalize();
```

Comunicaciones colectivas.

broadcast.c



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

```
MPI_Bcast(&a,1,MPI_FLOAT,root,MPI_COMM_WORLD);
```

&a: dirección de comienzo de buffer

1: número de elementos en buffer

MPI_FLOAT: tipo de datos del buffer

root: identif. del root de la operación broadcast

MPI_COMM_WORLD: comunicador

```
}
```

```
MPI_Finalize();
```

Comunicaciones colectivas.

broadcast.c

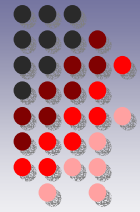


```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);

if(my_rank==0)
{
    printf("Introduce el dato:  a (float): ");scanf("%f",&a);
    printf("make broadcast\n");
    printf("mpicc -O4 broadcast.c -o broadcast -lmpi -llapack -lblas -lm\n");
    printf("$ mpirun -np 4 broadcast\n");
}
MPI_E Introduce el dato:  a (float): 4.2
MPI_E Introduce el dato:  b (float): 5.3
MPI_E Introduce el dato:  n (entero): 6
En procesador 2, los datos recibidos son a:4.200000    b:5.300000    n:6
if(my
{
    pr En procesador 1, los datos recibidos son a:4.200000    b:5.300000    n:6
    pr En procesador 3, los datos recibidos son a:4.200000    b:5.300000    n:6
    n:%d \n",my_rank,a,b,n);
}
MPI_Finalize();
```

Comunicaciones colectivas.

gather.c



```
inicializa(my_rank,mis_datos,TAMA);
if(my_rank==0)
{
    datos=(int*)malloc(sizeof(int)*TAMA*p);
}

MPI_Gather(mis_datos,TAMA,MPI_INT,datos,TAMA,MPI_INT,root,MPI_COMM_WORLD);
if(my_rank==0)
{
    printf("\n TODOS LOS DATOS RECIBIDOS EN PROCESO\n");
    ROOT SON:\n";
    escribe(datos,TAMA*p);
    free(datos);
}
```

Comunicaciones colectivas.

gather.c



```
inicializa(my_rank,mis_datos,TAMA);  
if(my_rank==0)
```

```
MPI_Gather(mis_datos,TAMA,MPI_INT,datos,TAMA,MPI_INT,root,MPI_COMM_WORLD);
```

Mis_datos: dirección buffer de envío (en cada nodo emisor)

TAMA: número de elementos a enviar desde cada proceso

MPI_INT: tipo de datos de los elementos del buffer de envío

datos: dirección buffer de recepción (en nodo receptor)

TAMA: número de elementos de cada recepción individual

MPI_INT: tipo de datos del buffer de recepción

root: identificador del proceso que recibe los datos

MPI_COMM_WORLD: comunicador



Com gath

```
$ mpirun -np 4 gather
```

Datos iniciales en el proceso 1:

```
10000 10001 10002 10003 10004 10005 10006 10007
10008 10009
```

Datos iniciales en el proceso 2:

```
20000 20001 20002 20003 20004 20005 20006 20007 20008
20009
```

Datos iniciales en el proceso 3:

```
30000 30001 30002 30003 30004 30005 30006 30007 30008
30009
```

Datos iniciales en el proceso 0:

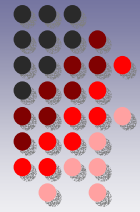
```
0 1 2 3 4 5 6 7 8 9
```

TODOS LOS DATOS RECIBIDOS EN PROCESO ROOT SON:

```
0 1 2 3 4 5 6 7 8 9 10000 10001 10002
10003 10004 10005 10006 10007 10008 10009 20000 20001
20002 20003 20004 20005 20006 20007 20008 20009 30000
30001 30002 30003 30004 30005 30006 30007 30008 30009
```

Comunicaciones colectivas.

scatter.c



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

if (my_rank == 0)
{
    datos = (int*) malloc(sizeof(int) * TAMA * p);
    inicializa(my_rank, datos, TAMA * p);
}

MPI_Scatter(datos, TAMA, MPI_INT, mis_datos, TAMA, MPI_INT, root, MPI_COMM_WORLD);

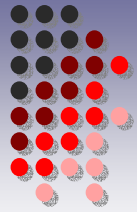
printf("Datos recibidos por proceso %d son:\n", my_rank);
escribe(mis_datos, TAMA);

if (my_rank == 0)
{
    free(datos);
}

MPI_Finalize();
```

Comunicaciones colectivas.

scatter.c



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

```
MPI_Scatter(datos,TAMA,MPI_INT,mis_datos,TAMA,MPI_INT,root,MPI_COMM_WORLD)
```

datos: dirección buffer de envío

TAMA: número de elementos a enviar a cada proceso

MPI_INT: tipo de datos de los elementos del buffer de envío

mis_datos: dirección buffer de recepción

TAMA: número de elementos en buffer de recepción

MPI_INT: tipo de datos del buffer de recepción

root: identificador del proceso que envia datos

MPI_COMM_WORLD: comunicador

```
MPI_Finalize();
```


Comunicaciones colectivas.

scatter.c



```
$ mpirun -np 4 scatter
```

```
Datos iniciales en proceso 0 son:
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
14	15	16	17	18	19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36	37	38	39		

```
Datos recibidos por proceso 1 son:
```

10	11	12	13	14	15	16	17	18	19
----	----	----	----	----	----	----	----	----	----

```
Datos recibidos por proceso 0 son:
```

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

```
Datos recibidos por proceso 2 son:
```

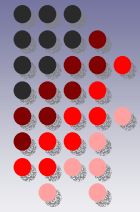
20	21	22	23	24	25	26	27	28	29
----	----	----	----	----	----	----	----	----	----

```
Datos recibidos por proceso 3 son:
```

30	31	32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----	----	----

Comunicaciones colectivas.

reduce.c



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

    inicializa(my_rank,mis_datos,TAMA);

if(my_rank==0)
{
    datos=(int*)malloc(sizeof(int)*TAMA);

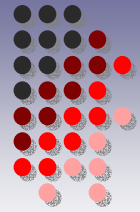
MPI_Reduce(mis_datos,datos,TAMA,MPI_INT,MPI_SUM,root,MPI_COMM_WORLD);

if(my_rank==0)
{
    printf("\n LOS DATOS, TRAS REDUCCION, EN PROCESO ROOT SON:\n");
    escribe(datos,TAMA);
    free(datos);
}

MPI_Finalize();
```

Comunicaciones colectivas.

reduce.c



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

```
MPI_Reduce(mis_datos,datos,TAMA,MPI_INT,MPI_SUM,root,MPI_COMM_WORLD);
```

mis_datos: dirección buffer de envío

datos: dirección buffer de recepción

TAMA: número de elementos en buffer de envío

MPI_INT: tipo de elementos de buffer de envío

MPI_SUM: operación a realizar durante la reducción

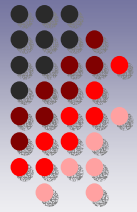
root: identificador del proceso root

MPI_COMM_WORLD:comunicador

```
MPI_Finalize();
```

Comunicaciones colectivas.

reduce.c



```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_C
```

```
$ mpirun -np 3 reduce
```

```
in
```

```
En proceso 2 los datos son:
```

```
if(my_rank == 2)
{
    data[0] = 20000; data[1] = 20001; data[2] = 20002; data[3] = 20003; data[4] = 20004; data[5] = 20005; data[6] = 20006; data[7] = 20007;
    data[8] = 20008; data[9] = 20009;
}
```

```
En proceso 0 los datos son:
```

```
MPI_Reduce(data, root_data, 10, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
if(my_rank == 0)
```

```
En proceso 1 los datos son:
```

```
printf("Proceso 1: ");
for(i=0; i<10; i++)
    printf("%d ", data[i]);
printf("\n");
```

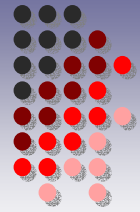
```
MPI_Finalize();
```

```
LOS DATOS, TRAS REDUCCION, EN PROCESO ROOT SON:
```

```
30000 30003 30006 30009 30012 30015 30018 30021
30024 30027
```

Comunicaciones colectivas.

Operaciones de reducción

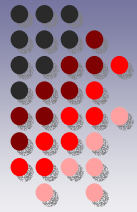


MPI Operator	Operation
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bitwise and
MPI_LOR	logical or
MPI_BOR	bitwise or
MPI_LXOR	logical exclusive or
MPI_BXOR	bitwise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Agrupación de datos



- Con contador en rutinas de envío y recepción: agrupar los datos a enviar
 - Con tipos derivados
 - Con empaquetamiento



Agrupación de datos. Tipos derivados

- Se crean en tiempo de ejecución
- Se especifica la disposición de los datos en el tipo:

```
Int MPI_Type_Struct  
(  
    int                count,  
    int                *array_of_block_lengths,  
    MPI_Aint            *array_of_displacements,  
    MPI_Datatype        *array_of_types,  
    MPI_Datatype        *newtype  
)
```

- Se pueden construir tipos de manera recursiva
- La creación de tipos requiere trabajo adicional



Agrupación de datos.

Tipos derivados: derivados.c (1/2)

```
typedef struct
{
    float    a;
    float    b;
    int      n;
} INDATA_TYPE;
INDATA_TYPE    indata;
MPI_Datatype    message_type;

if(my_rank==0)
{
    printf("Introduce el dato:a (float): ");scanf("%f",&(indata.a));
    printf("Introduce el dato:b (float): ");scanf("%f",&(indata.b));
    printf("Introduce el dato:n(entero): ");scanf("%d",&(indata.n));
}

Build_derived_type(&indata,&message_type);

MPI_Bcast(&indata,count,message_type,root,MPI_COMM_WORLD);

if(my_rank !=0)
    printf("En procesador %d, los datos recibidos son a:%f      b:%f\n",my_rank,indata.a,indata.b,indata.n)
```




Agrupación de datos.

Tipos derivados: derivados.c (2/2)

```
void      Build_derived_type(INDATA_TYPE      *indata,      MPI_Datatype
      *message_type_ptr)
{
    int          block_lengths[3]; /*...continuación...*/
    MPI_Aint      dis[3];
    MPI_Aint      addresses[4];
    MPI_Datatype  typelist[3];

    typelist[0]=MPI_FLOAT;
    typelist[1]=MPI_FLOAT;
    typelist[2]=MPI_INT;

    block_lengths[0]=1;
    block_lengths[1]=1;
    block_lengths[2]=1;

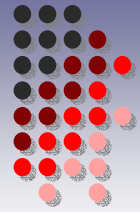
    MPI_Address(indata,&addresses[0]);
    MPI_Address(&(indata->a),&addresses[1]);
    MPI_Address(&(indata->b),&addresses[2]);
    MPI_Address(&(indata->n),&addresses[3]);
    /*...continua...*/

    MPI_Type_struct(3,block_lengths,dis,
    typelist,message_type_ptr);

    MPI_Type_commit(message_type_ptr);
}
```

Agrupación de datos.

Tipos derivados



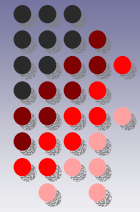
- Si los datos que constituyen el nuevo tipo son un subconjunto de entradas hay mecanismos especiales para construirlos (1/3):

```
int MPI_Type_contiguous
(
    int count,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype
)
```

- Crea un tipo derivado formado por `count` elementos del tipo `oldtype` contiguos en memoria.

Agrupación de datos.

Tipos derivados

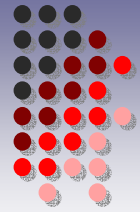


- Si los datos que constituyen el nuevo tipo son un subconjunto de entradas hay mecanismos especiales para construirlos (2/3):

```
int MPI_Type_vector
(
    int count,
    int block_lenght,
    int stride,
    MPI_Datatype element_type,
    MPI_Datatype *newtype
)
```

- Crea un tipo derivado formado por **count** elementos, cada uno de ellos con **block_lenght** elementos del tipo **element_type**.
- **stride** es el número de elementos del tipo **element_type** entre elementos sucesivos del tipo **new_type**.
- De este modo, los elementos pueden ser entradas igualmente espaciadas en un array.

Agrupación de datos. Tipos derivados

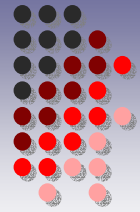


- Si los datos que constituyen el nuevo tipo son un subconjunto de entradas hay mecanismos especiales para construirlos (3/3):

```
int MPI_Type_indexed
(
    int count,
    int *array_of_block_lengths,
    int *array_of_displacements,
    MPI_Datatype element_type,
    MPI_Datatype *newtype
)
```

- Crea un tipo derivado con **count** elementos, habiendo en cada elemento **array_of_block_lengths[i]** entradas de tipo **element_type**, y el desplazamiento **array_of_displacements[i]** unidades de tipo **element_type** desde el comienzo de **newtype**

Agrupación de datos. Empaquetamiento



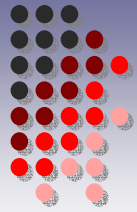
- Los datos se pueden empaquetar para ser enviados y desempaquetarse tras ser recibidos.
 - Se empaquetan **in_count** datos de tipo **datatype**, y **pack_data** referencia los datos a empaquetar en el **buffer**, que debe consistir de **size** bytes (puede ser una cantidad mayor a la que se va a ocupar).
- ```
int MPI_Pack
(
 void *pack_data,
 int in_count,
 MPI_Datatype datatype,
 void *buffer,
 int size,
 int *position_ptr,
 MPI_Comm comm
)
```
- El parámetro **position\_ptr** es de E/S:
    - Como entrada, el dato se copia en la posición **buffer+\*position\_ptr**.
    - Como salida, referencia la siguiente posición en el **buffer** después del dato empaquetado.
    - El cálculo de dónde se sitúa en el **buffer** el siguiente elemento a empaquetar lo hace MPI automáticamente.

# Agrupación de datos. Empaquetamiento



- Copia **count** elementos de tipo **datatype** en **unpack\_data**, tomándolos de la posición **buffer+\*position\_ptr** del buffer.
- El tamaño del **buffer (size)** en bytes, y **position\_ptr** es manejado por MPI de manera similar a como lo hace en MPI\_Pack.

```
int MPI_Unpack
(
 void *buffer,
 int size,
 int *position_ptr,
 Void *unpack_data,
 int count,
 MPI_Datatype datatype,
 MPI_Comm comm
)
```



## Agrupación de datos.

### Empaquetamiento: empaquetados.c

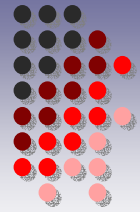
```
if(my_rank==0)
{
printf("Introduce el dato: a (float): ");
scanf("%f",&a);
printf("Introduce el dato: b (float): ");
scanf("%f",&b);
printf("Introduce el dato: n (entero): ");
scanf("%d",&n);
}
position=0;
MPI_Pack(&a,1,MPI_FLOAT,buffer,100,&position,MPI_COMM_WORLD);
MPI_Pack(&b,1,MPI_FLOAT,buffer,100,&position,MPI_COMM_WORLD);
MPI_Pack(&n,1,MPI_INT,buffer,100,&position,MPI_COMM_WORLD);

MPI_Bcast(buffer,position,MPI_PACKED,root,MPI_COMM_WORLD);

position=0;
MPI_Unpack(buffer,100,&position,&a,1,MPI_FLOAT,MPI_COMM_WORLD);
MPI_Unpack(buffer,100,&position,&b,1,MPI_FLOAT,MPI_COMM_WORLD);
MPI_Unpack(buffer,100,&position,&n,1,MPI_INT,MPI_COMM_WORLD);

printf("En procesador %d, los datos recibidos son a:%f b:%f\n",my_rank,a,b,n);
```

# Comunicadores



- MPI\_COMM\_WORLD incluye a todos los procesos
- Se puede definir comunicadores con un número menor de procesos: para comunicar datos en cada fila de procesos en la malla, en cada columna, ...
- Dos tipos de comunicadores:
  1. **intra-comunicadores:** se utilizan para enviar mensajes entre los procesos en ese comunicador,
  2. **inter-comunicadores:** se utilizan para enviar mensajes entre procesos en distintos comunicadores.



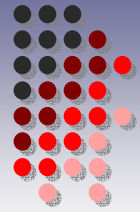
# Comunicadores. Intra-comunicadores



## 1. Intra-comunicador. Consta de:

- **Un grupo**, que es una colección ordenada de procesos a los que se asocia identificadores entre 0 y  $p-1$ .
- **Un contexto**, que es un identificador que asocia el sistema al grupo.
- Adicionalmente, a un comunicador se le puede asociar **una topología virtual**.

# Comunicadores. Intra-comunicadores



- MPI\_COMM\_WORLD consta de  $p=q^2$  procesos agrupados en  $q$  filas y  $q$  columnas. El proceso número  $r$  tiene las coordenadas  $(r \div q, r \bmod q)$ .
- Ejemplo: Creación un comunicador cuyos procesos son los de la primera fila de nuestra malla virtual.

```
MPI_Group MPI_GROUP_WORLD;
MPI_Group first_row_group, first_row_comm;
int row_size;
int *process_ranks;

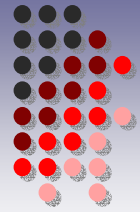
process_ranks=(int *) malloc(q*sizeof(int));

for(proc=0;proc<q;proc++)
 process_ranks[proc]=proc;

MPI_Comm_group(MPI_COMM_WORLD,&MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD,q,process_ranks,&first_row_group);
MPI_Comm_create(MPI_COMM_WORLD,first_row_group,&first_row_comm);
```

# Comunicadores.

## Intra-comunicadores



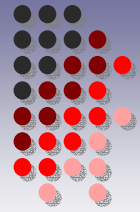
- Para crear varios comunicadores disjuntos

```
int MPI_Comm_split
(
 MPI_Comm old_comm,
 int split_key,
 int rank_key,
 MPI_Comm *new_comm
)
```

- Crea un nuevo comunicador para cada valor de **split\_key**.
- Los procesos con el mismo valor de **split\_key** forman un grupo.
- Si dos procesos **a** y **b** tienen el mismo valor de **split\_key** y el **rank\_key** de **a** es menor que el de **b**, en el nuevo grupo **a** tiene identificador menor que **b**.
- Si los dos procesos tienen el mismo **rank\_key** el sistema asigna los identificadores arbitrariamente.

# Comunicadores.

## Intra-comunicadores



```
int MPI_Comm_split
 (...
)
```

- Es una operación colectiva.
- Todos los procesos en el comunicador deben llamarla.
- Los procesos que no se quiere incluir en ningún nuevo comunicador pueden utilizar el valor `MPI_UNDEFINED` en `rank_key`, con lo que el valor de retorno de `new_comm` es `MPI_COMM_NULL`.
- Ejemplo: crear  $q$  grupos de procesos asociados a las  $q$  filas:

```
MPI_Comm my_row_comm;
int my_row;

my_row=my_rank/q;
MPI_Comm_split (MPI_COMM_WORLD,my_row,my_rank,&my_row_comm);
```



# Comunicadores.

## Intra-comunicadores. Topologías

- A un grupo se le puede asociar una topología virtual:
  - topología de grafo en general
  - de malla o cartesiana

- Una topología cartesiana se crea:

```
int MPI_Cart_create (
 MPI_Comm old_comm,
 int number_of_dims,
 int *dim_sizes,
 int *periods,
 int reorder,
 MPI_Comm *cart_comm)
```

- El número de dimensiones de la malla es **number\_of\_dims**
- El número de procesos en cada dimensión está en **dim\_sizes**
- Con **periods** se indica si cada dimensión es circular o lineal
- Un valor de 1 en **reorder** indica al sistema que se reordenen los procesos para optimizar la relación entre el sistema físico y el lógico.

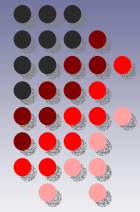


# Comunicadores.

## Intra-comunicadores. Topologías

- Las coordenadas de un proceso conocido su identificador se obtienen con  
`int MPI_Cart_coords( MPI_Comm comm, int rank,  
int number_of_dims, int *coordinates)`
- El identificador, conocidas las coordenadas con  
`int MPI_Cart_rank( MPI_Comm comm, int *coordinates, int *rank)`
- Una malla se puede particionar en mallas de menor dimensión  
`int MPI_Cart_sub(MPI_Comm old_comm, int *varying_coords,  
MPI_Comm *new_comm)`
  - en **varying\_coords** se indica para cada dimensión si pertenece al nuevo comunicador.
  - Si **varying\_coords**[0]=0 y **varying\_coords**[1]=1 para obtener el nuevo comunicador no se varía la primera dimensión pero sí la segunda. Se crean **q** comunicadores, uno por cada fila.
  - Es colectiva.

# MPI-2



- **Corregir errores de MPI-1**
- **Entrada/Salida paralela (MPI-IO)**
  - Aumentar prestaciones E/S
  - Accesos no contiguos a memoria y a ficheros
  - Operaciones colectivas de E/S
  - Punteros a ficheros tantos individuales como colectivos
  - E/S asíncrona
  - Representaciones de datos portables y ajustadas a las necesidades
- **Operaciones remotas de memoria**
  - Proveer elementos del “tipo” de memoria compartida
  - Concepto de “ventana de memoria”: porción de memoria de un proceso que es expuesta explícitamente a accesos de otros procesos
  - Operación remotas son asíncronas → necesidad de sincronizaciones explícitas
- **Gestión dinámica de procesos**
  - Un proceso MPI puede participar en la creación de nuevos procesos: **spawing**
  - Un proceso MPI puede comunicarse con procesos creados separadamente: **connecting**
  - La clave: **intercomunicadores**: comunicadores que contienen 2 grupos de procesos
  - Extensión de comunicaciones colectivas a intercomunicadores



## MPI-2

### Entrada/Salida paralela (MPI-IO): escribiendo en diferentes ficheros: mpi\_io\_1.c

```
int main(int argc, char *argv[])
{
 int i, myrank, buf[BUFSIZE];
 char filename[128];
 MPI_File myfile;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

 for (i=0; i<BUFSIZE; i++)
 buf[i]=myrank*BUFSIZE+i;

 sprintf(filename, "testfile.%d", myrank); //cada proceso → un fichero

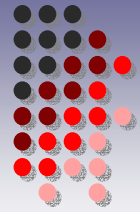
 MPI_File_open(MPI_COMM_SELF, filename, MPI_MODE_WRONLY |
 MPI_MODE_CREATE, MPI_INFO_NULL, &myfile);
 MPI_File_write(myfile, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
 MPI_File_close(&myfile);

 MPI_Finalize();
 return 0;
}
```



# MPI-2

## Entrada/Salida paralela (MPI-IO): escribiendo en diferentes ficheros: mpi\_io\_1.c



```
MPI_File_open(MPI_COMM_SELF,filename, MPI_MODE_WRONLY |
MPI_MODE_CREATE, MPI_INFO_NULL,&myfile);
```

- MPI\_COMM\_SELF: Comunicador de los procesos que abren el fichero. En este caso cada proceso abre el suyo propio.
- Modo apertura
- Campo nulo
- Descriptor del fichero

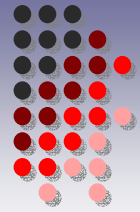
```
MPI_File_write(myfile,buf,BUFSIZE,MPI_INT,MPI_STATUS_IGNORE);
```

- Descriptor del fichero
- Datos a escribir
- Tamaño
- Tipo
- MPI\_STATUS\_IGNORE: para que no me devuelva información en Status, porque no la necesito

```
}
```

# MPI-2

## Entrada/Salida paralela (MPI-IO): escribiendo en un fichero único: mpi\_io\_2.c



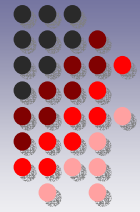
```
int main(int argc, char *argv[])
{
 int i, myrank, buf[BUFSIZE];
 char filename[128];
 MPI_File myfile;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 for (i=0; i<BUFSIZE; i++)
 buf[i]=myrank*BUFSIZE+i;

 sprintf(filename, "testfile"); // todos procesos → un único fichero
 MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_WRONLY |
 MPI_MODE_CREATE, MPI_INFO_NULL, &myfile);
 MPI_File_set_view(myfile, myrank*BUFSIZE*sizeof(int), MPI_INT, MPI_INT, "
 native", MPI_INFO_NULL);
 MPI_File_write(myfile, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
 MPI_File_close(&myfile);

 MPI_Finalize();
 return 0;
}
```

# MPI-2

Entrada/Salida paralela (MPI-IO):  
escribiendo en un fichero único: mpi\_io\_2.c



```
MPI_File_set_view(myfile,myrank*BUFSIZE*sizeof(int),MPI_INT,MPI_INT,"
 native",MPI_INFO_NULL);
```

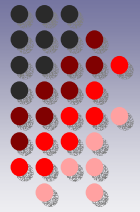
Descriptor del Fichero

Desplazamiento donde escribir

Tipo a escribir

Tipo zona discontinua del fichero

"native"representación del dato a escribir tal como se  
representa en memoria



## MPI-2

### Entrada/Salida paralela (MPI-IO): leyendo de un fichero único: mpi\_io\_3.c

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

 MPI_File_open(MPI_COMM_WORLD, "testfile", MPI_MODE_RDONLY,
 MPI_INFO_NULL, &myfile);

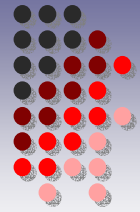
 MPI_File_get_size(myfile, &filesize); //in bytes
 filesize = filesize / sizeof(int); //in number of ints
 bufsize = filesize / numprocs + 1; //local number to read
 buf = (int *) malloc (bufsize * sizeof(int));

 MPI_File_set_view(myfile, myrank * bufsize * sizeof(int), MPI_INT, MPI_
 INT, "native", MPI_INFO_NULL);
 MPI_File_read(myfile, buf, bufsize, MPI_INT, &status);
 MPI_Get_count(&status, MPI_INT, &count);

 printf("process %d read %d ints \n", myrank, count);

 /* ... */
}
```

# Ejercicios



1. Copia a tu directorio y prueba los ejemplos que están en:  
`/home/javiercm/ejemplos_mpi`

```
Conectarse a luna.inf.um.es
ssh luna.inf.um.es
usuario XXXX
clave YYYY
PATH=$PATH:..
lamboot
```

Para obtener los programas de ejemplo:

```
cp /home/javiercm/ejemplos_mpi/*.c .
```

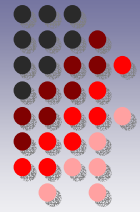
Para compilar el programa `hello.c`:

```
mpicc hello.c -o hello -lm -lmpi
```

Para ejecutarlo con 4 procesos:

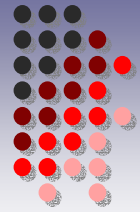
```
mpirun -np 4 hello
```

# Ejercicios

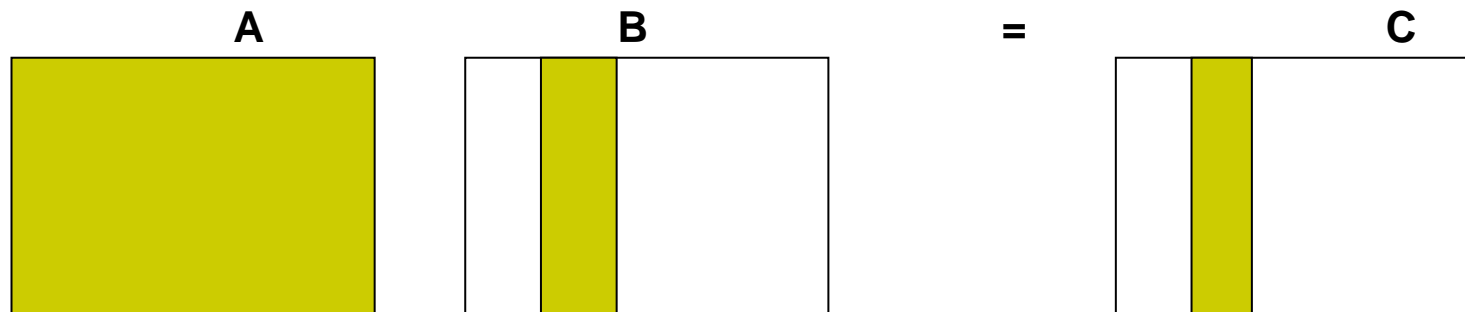


2. Amplia el programa [broadcast.c](#) de manera que además de los datos que se distribuyen en la versión actual también se envíe una cadena de 10 caracteres.
3. Modifica el programa [gather.c](#) para que además del vector **datos**, se recopile en el proceso raíz el vector **resultados**, también formado de enteros.
4. Modifica el programa [scatter.c](#) para que el array **datos** distribuido desde el proceso 0 contenga números reales de doble precisión en lugar de enteros.
5. Amplia el programa [reduce.c](#) para que, además de calcular la suma, calcule el valor máximo relativo a cada posición de los arrays locales de los distintos procesos (el máximo de la posición 0, el de la posición 1,...)
6. Amplia el programa [hello.c](#) para que al final del programa cada proceso **i** mande otro mensaje de saludo al nodo **(i+1)%p**, y tras ello, cada proceso muestre el mensaje recibido en pantalla.

# Ejercicios

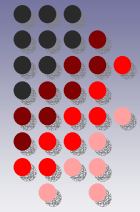


7. Completa el programa de ejemplo [mult\\_mpi.c](#) (calcula el producto de matrices  $C \leftarrow A * B$ ) que conlleva:
- **broadcast:** La matriz A se manda completamente a todos.
  - **scatter:** La matriz B se distribuye por bloques de columnas.
  - **mm:** Cada proceso calcula un bloque de columnas de C multiplicando la matriz A por el bloque de columnas de B que le ha correspondido. Por ejemplo,  $P_1$  calcularía:



- **gather:** Todos los procesos mandan al root sus resultados parciales, es decir, sus bloques de columnas de C. Al finalizar, el proceso root contendrá la matriz C resultado completamente.

# Ejercicios



8. Amplia el programa [derivados.c](#) para que el tipo derivado que se forma y se envía contenga también una cadena de 48 caracteres.
9. Amplia el programa [empaquetados.c](#) para que el paquete de datos que se forma y se envía contenga también un vector de 100 números reales de doble precisión.