

COMPUTACIÓN PARALELA
3º GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA
INFORMACIÓN

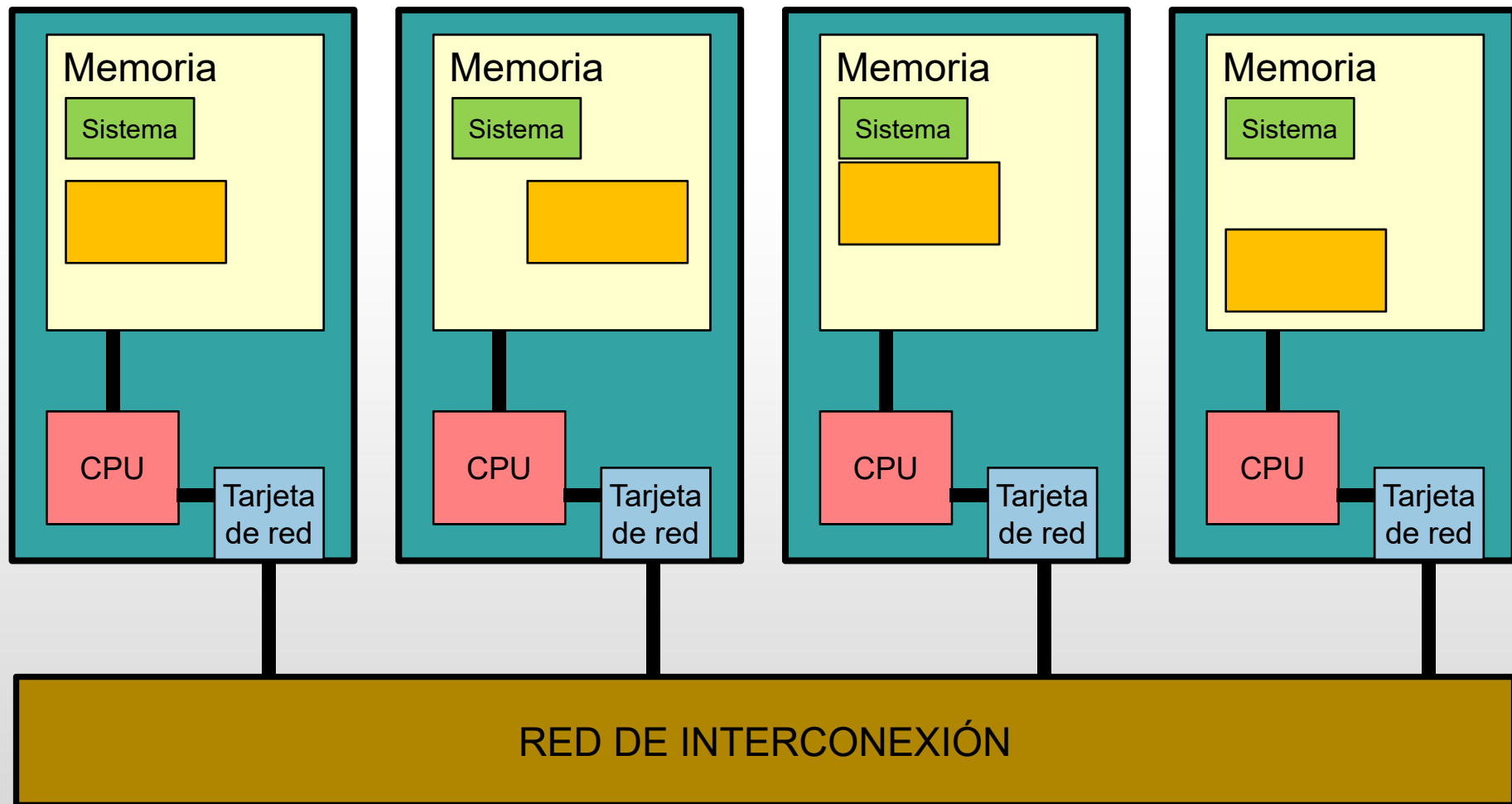
Unidad Didáctica II.
Arquitecturas de memoria
distribuida: MPI

- II.1 Introducción**
- II.2 Introducción a MPI**
- II.3 Funciones MPI básicas**
- II.4 Funciones MPI de comunicación colectivas**
- II.5 Otras funcionalidades MPI**
- II.6 Ejemplos**

II.1. INTRODUCCIÓN

- **Recordatorio arquitectura**
- **Modelos explotación del paralelismo en arquitecturas de memoria distribuida**

II.1 INTRODUCCIÓN MPI – Recordatorio arquitecturas



Modelos: (rápido)

- Paso de mensajes (lo veremos con MPI)
- Datos paralelos
- Plataformas o librerías simulan memoria compartida
- ¿Podríamos utilizar comunicaciones a más bajo nivel, TCP, UDP ...?

Datos Paralelos

- Dirigido a aplicaciones que se centran en operaciones con conjuntos de datos
- Cada tarea trabaja sobre un conjunto de datos ...
- ... cada tarea realiza la misma operación a su sección de datos (¿modelo de descomposición?)

- En memoria compartida (es un modelo por tanto no exclusivo de memoria distribuida) se accede a través de la memoria global
- En memoria distribuida cada tarea debe almacenar su sección en su memoria
- Para trabajar con las estructuras de datos paralelas se utilizan librerías o directivas (un ejemplo es HPF)

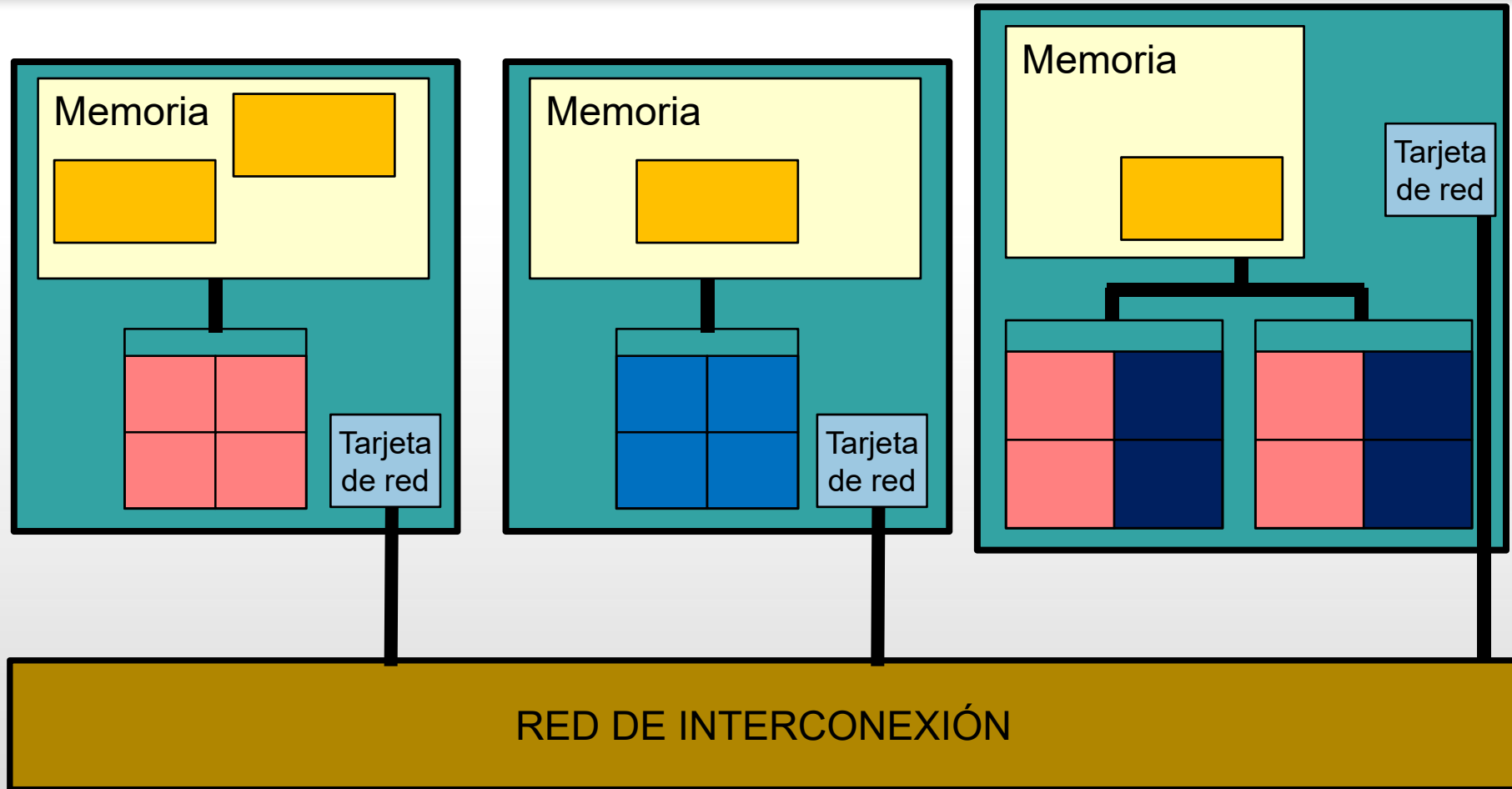
Librerías uso memoria distribuida sin comunicaciones explícitas

- Modela un sistema de memoria distribuida como un modelo con un espacio de memoria único.
- El objetivo normalmente es abstraer arquitecturas híbridas (DM + SM)
- El más utilizado UPC, pero hay varias alternativas disponibles
- ¿Qué puede pasar al abstraer la arquitectura al programador? Pensemos en el patrón de comunicaciones

A considerar

- ¿Hemos hablado de memoria físicamente o lógicamente distribuida?
- Utilizamos herramientas que facilitan la programación ¿Por qué?
- ¿Podríamos utilizar herramientas de más o menos nivel?
 - S.O.
 - Protocolos de comunicación TCP, UDP...
 - OpenCL

II.1 INTRODUCCIÓN MPI – Recordatorio arquitecturas



II.2. INTRODUCCIÓN MPI

- **Características básicas**
- **Distribuciones**
- **Modelo de ejecución**
- **Descomposición de trabajo**
- **Distribución de datos**
- **Aspectos básicos en diseño y ejecución**

¿Por qué MPI? (**Message Passing Interface**)

- Estándar de facto para arquitecturas de memoria distribuida
- Rendimiento: consigue buenos rendimientos ocultando al usuario las características propias de la red siendo eficiente en múltiples arquitecturas.
- Escalable: la escalabilidad depende del algoritmo y su implementación pero también del uso correcto de características avanzadas (grupos, comunicaciones colectivas)
- Formal: comportamiento completamente definido
- Seguro: las comunicaciones se realizan sin duda ni necesidad de controlarlo
- Lenguajes: C, Fortran, Python, ...

Implementaciones MPI (libres):

- MPICH
- LAM/MPI
- OpenMPI
- MPI-MS (.NET)
- ...

MPI está enfocado a arquitecturas de memoria distribuida en las cuales:

- Disponemos de un conjunto de procesadores con su propia memoria
- Cada procesador ejecuta su propio programa
- Un proceso se ejecuta en un procesador. Un procesador debe ejecutar sólo un proceso
- Los procesadores están conectados mediante una red de interconexión

Que es **EL MISMO**
para **TODOS** los
procesos.

**PERO NO HACE LO
MISMO EN TODOS**

¿Y si el
procesador es
un multicore?

**NO ES ESTE
EL MODELO A
UTILIZAR**

Los programas (o procesos o tareas) cooperan
para conseguir un objetivo común.
Coordinación → Comunicación

MPI es un modelo de **Paso de mensajes**:

- Cada una de las tareas (software) tiene su propia memoria.
- Los datos se intercambian enviando y recibiendo **mensajes**
- El intercambio de datos **normalmente** requiere trabajo cooperativo (emisor y receptor)

¿Dónde residen los datos que procesamos?

Mensaje → Datos
¿Dónde los guardamos?

Emisor y receptor
no pueden estar
haciendo otra cosa

... ¿Y si están
haciendo otra cosa?
SE ESPERAN

Elemento básico: proceso

- Un proceso es la ejecución de un programa (compilado y linkado) desarrollado en lenguaje secuencial (en nuestro caso usaremos C)
- Los procesos en ejecución hacen llamadas a rutinas de la librería de paso de mensajes (MPI)
- Un proceso se ejecuta en un procesador. Un procesador debe ejecutar sólo un proceso
- El proceso se comunica y sincroniza con el resto de procesos mediante el envío/recepción de mensajes

¿Y si el
procesador es
un multicore?

**NO ES ESTE
EL MODELO A
UTILIZAR**

El intercambio de datos puede ser:

Elementos:
emisor y receptor

¿Puede haber más
de un emisor y más
de un receptor?

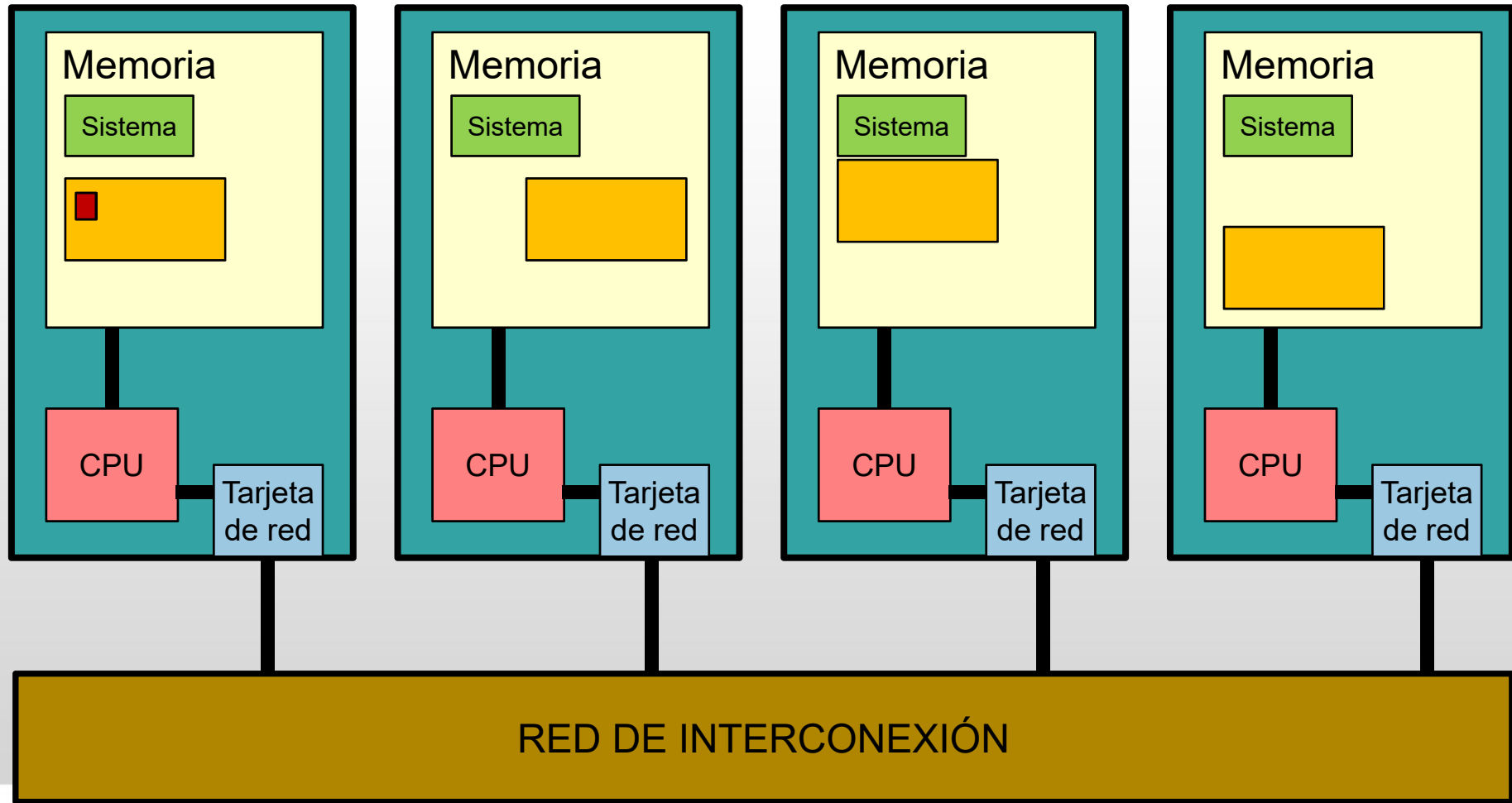
- Cooperativo: intervienen todos los elementos en el proceso de intercambio de datos (MPI-1).
- One sided (MPI-2): solo interviene uno de los elementos que intervienen en la comunicación

Elementos:
emisor y receptor

Si hay emisor y no hay
receptor ¿Qué proceso
recibe? ¿Cómo recibe?
¿Qué recibe?

II.2 INTRODUCCIÓN MPI – Modelo de ejecución

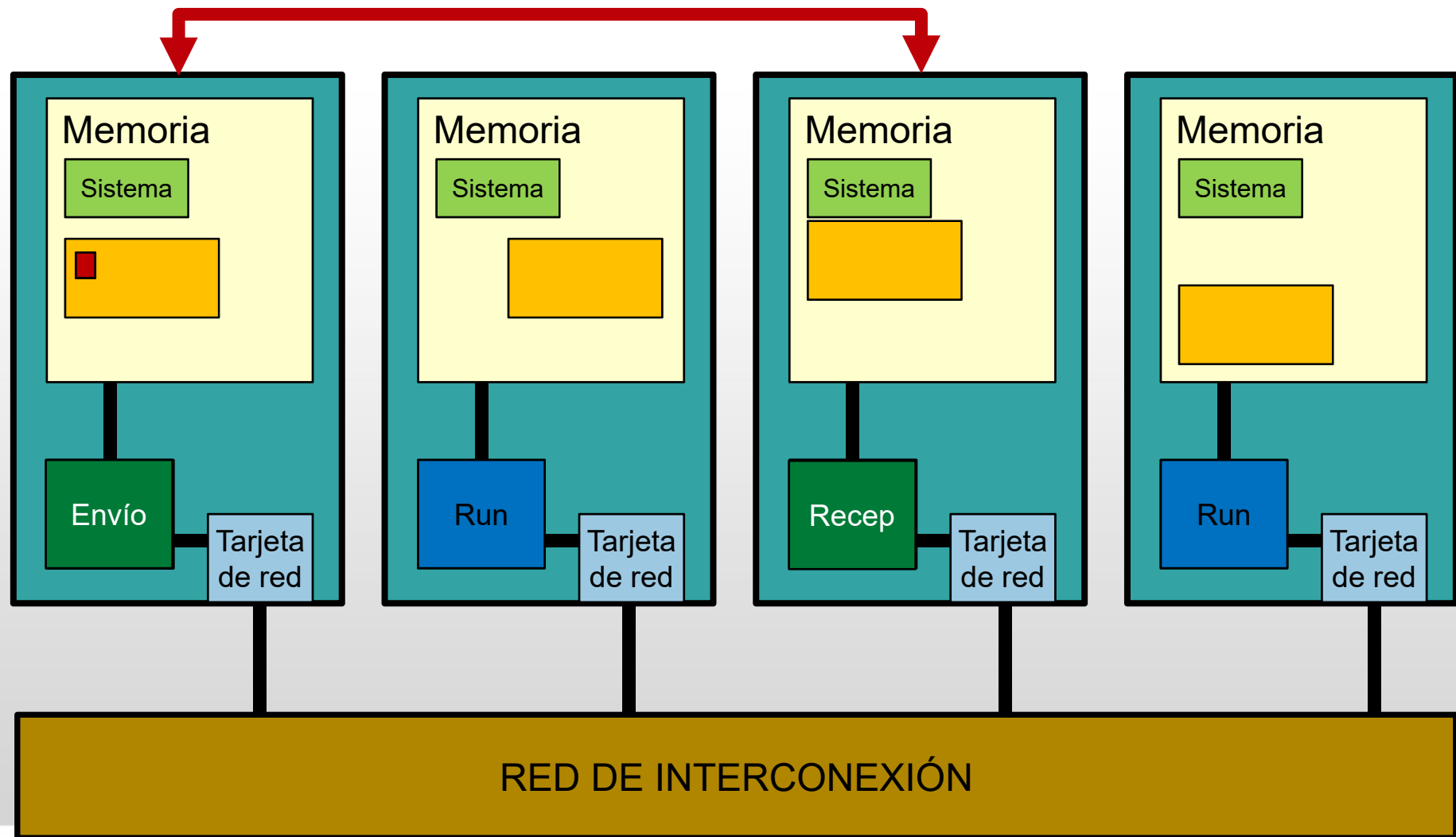
Los datos a intercambiar con un mensaje residen en memoria



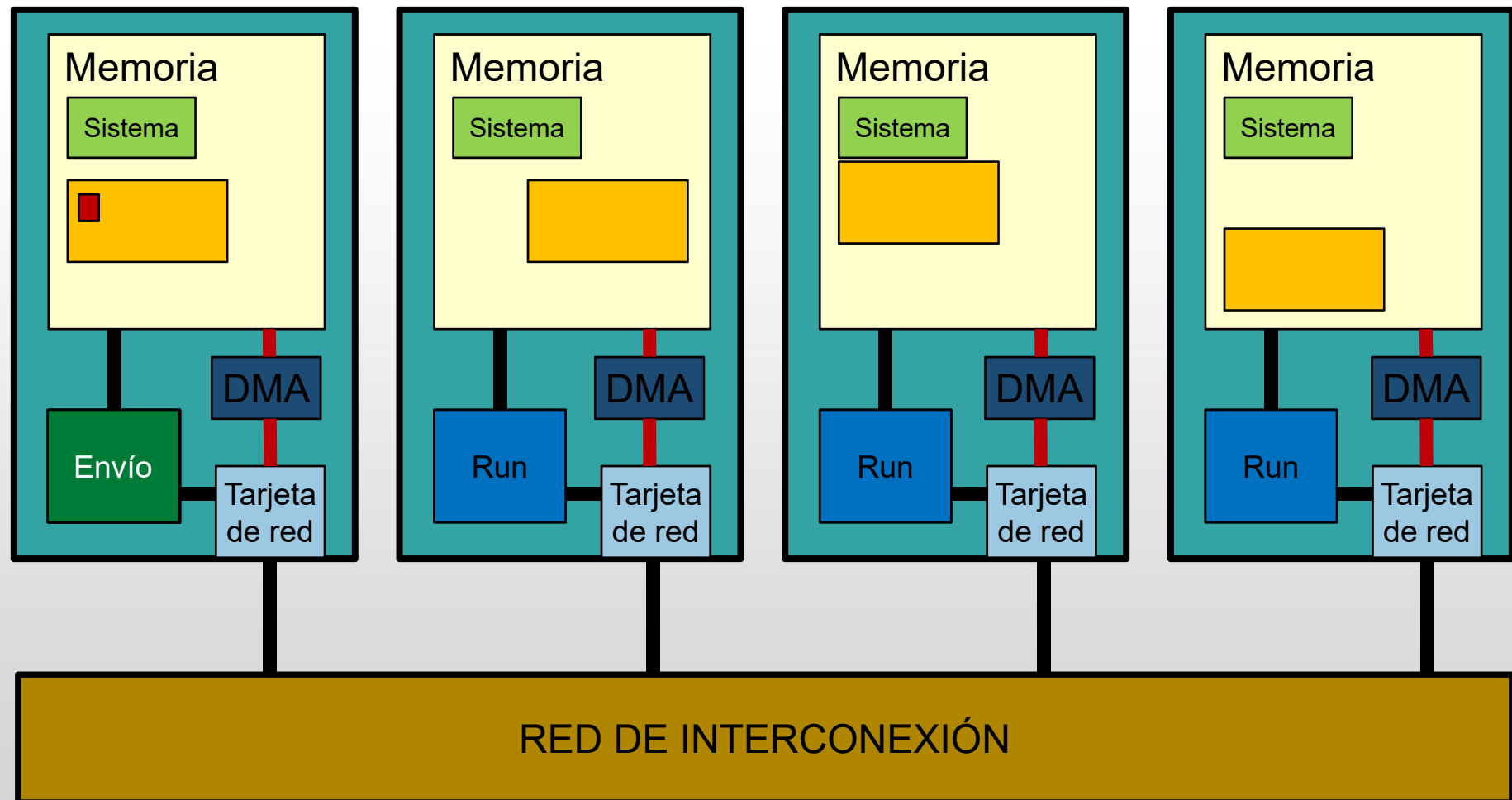
II.2 INTRODUCCIÓN MPI – Modelo de ejecución

Cooperativo

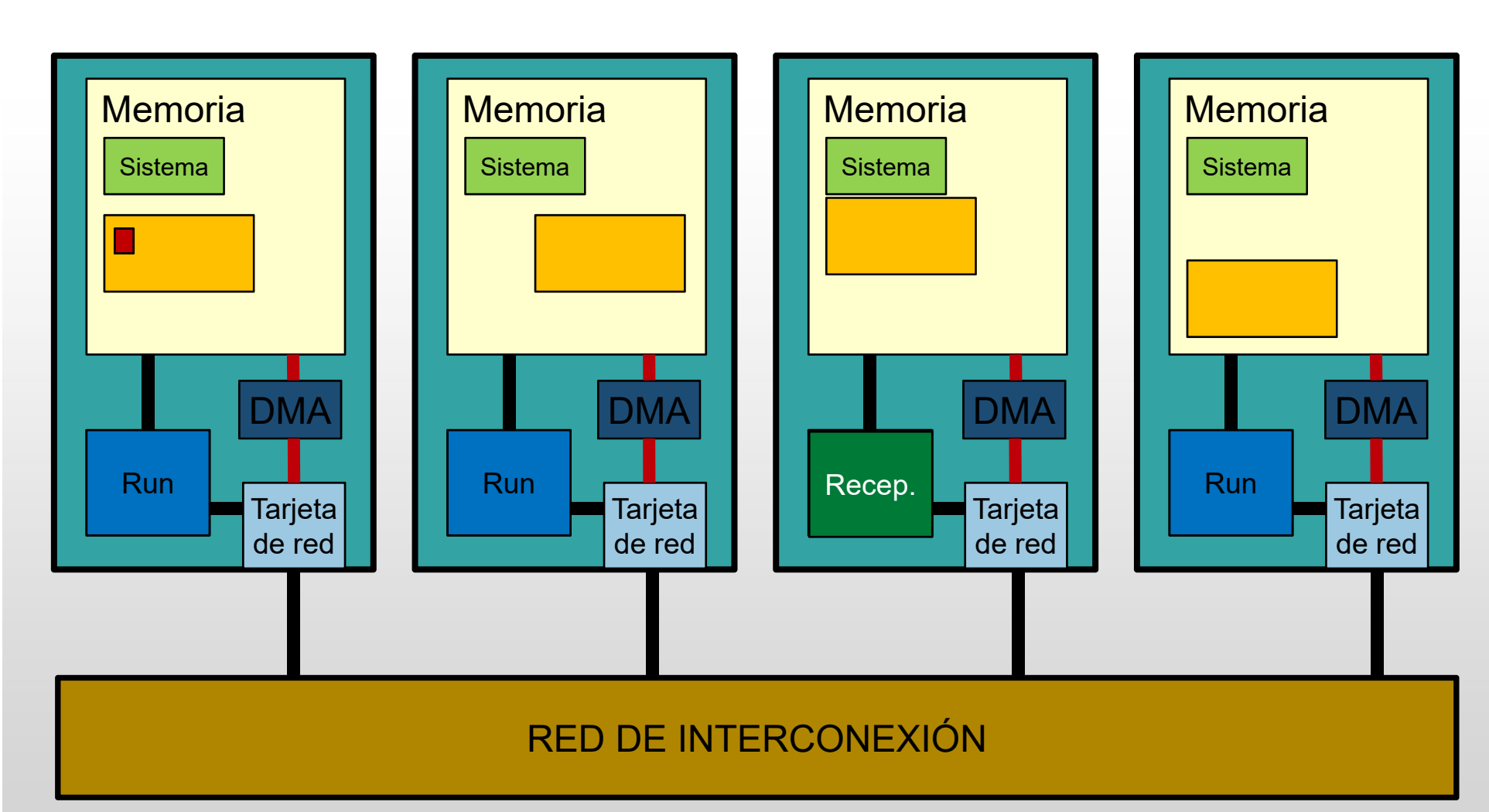
Sincronizados



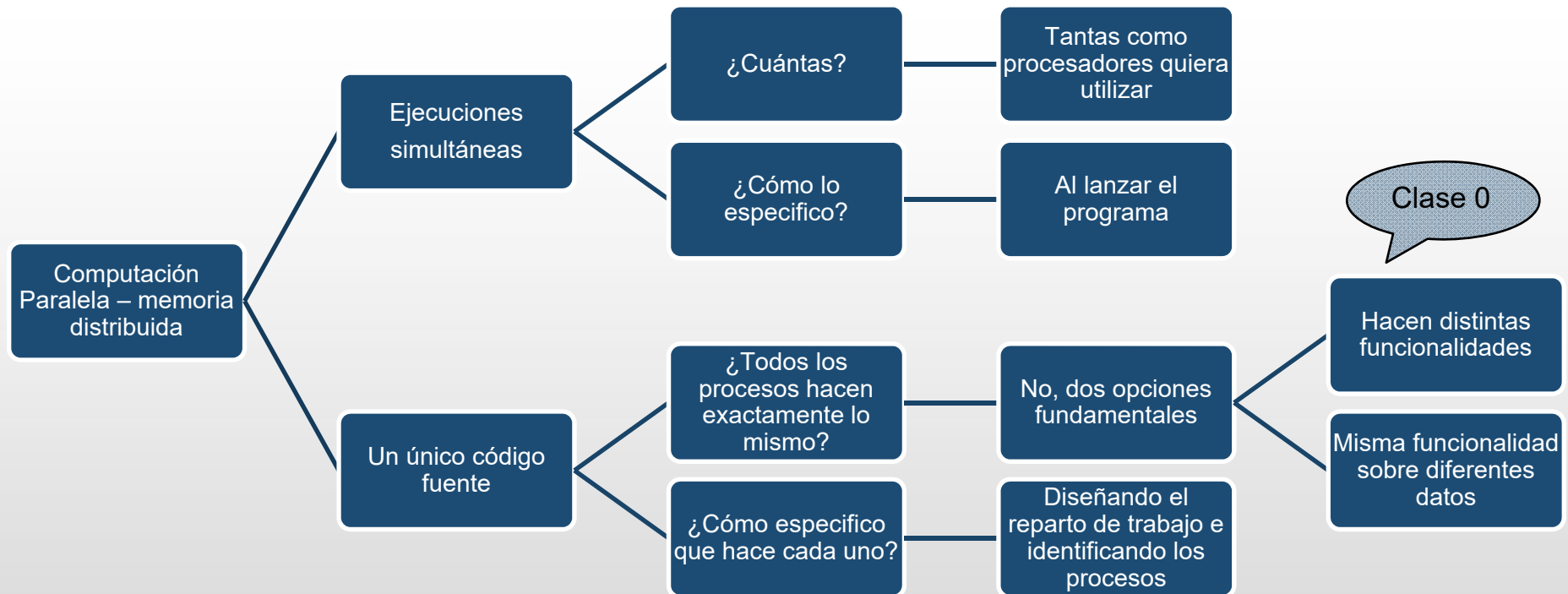
One-sided



One-sided



II.2 INTRODUCCIÓN MPI – Modelo de ejecución



II.2 INTRODUCCIÓN MPI – Descomposición de trabajo

Descomposición de dominio

Misma funcionalidad
sobre diferentes
datos

Descomposición funcional

Hacen distintas
funcionalidades

Clase 0:

Para todos los elementos de una matriz contar:

1. Elementos pares
2. Elementos impares
3. Números cuyas cifras sumen 15

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

II.2 INTRODUCCIÓN MPI – Descomposición de trabajo

Descomposición de dominio

Misma funcionalidad
sobre diferentes
datos

Proceso 0 **Proceso 1** **Proceso 2**
Todos tareas 1, 2 y 3

Clase 0:
Para todos lo
contar:

```
if dato == 0 { // nada}  
else { // trabajo}
```

1. Elementos pares
2. Elementos impares
3. Números cuyas cifras sumen 15

Bloque

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
271	124	48	261	190	167	283	113	146	114	302	352
0	0	0	0	0	0	0	0	0	0	302	0
0	0	464	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	277	0	0	8
414	0	0	0	0	0	0	0	0	0	0	9

Bloque

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

Cíclico

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

Cíclico 2D

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

II.2 INTRODUCCIÓN MPI – Distribución de datos

Descomposición de dominio

Misma funcionalidad
sobre diferentes
datos

Proceso 0 Proceso 1 Proceso 2

Todos tareas 1, 2 y 3

Clase 0:

Para todos los elementos de una matriz
contar:

1. Elementos pares
2. Elementos impares
3. Números cuyas cifras sumen 15

Proceso 0

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

24	24	4
----	----	---

--	--	--

--	--	--

Proceso 1

24	24	1
----	----	---

Proceso 2

28	20	1
----	----	---

II.2 INTRODUCCIÓN MPI – Distribución de datos

Descomposición de dominio

Misma funcionalidad
sobre diferentes
datos

Proceso 0 **Proceso 1** **Proceso 2**
Todos tareas 1, 2 y 3

Clase 0:

Para todos los elementos de una matriz
contar:

1. Elementos pares
2. Elementos impares
3. Números cuyas cifras sumen 15

Proceso 0

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

Proceso 1

327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417

¿Cuántas comunicaciones son necesarias?

Dependiendo del almacenamiento elegido (4,12,1)
para comunicar el mismo número de datos

Proceso 2

RECORDAD:

Si la reserva es dinámica el programador decide el almacenamiento, si es estática lo decide el compilador
Dos procesos de reserva de memoria consecutivos (malloc) no aseguran memoria contigua

II.2 INTRODUCCIÓN MPI – Distribución de datos

Descomposición funcional


Hacen distintas funcionalidades

Clase 0:


Para todos los elementos de una matriz contar:

1. Elementos pares
2. Elementos impares
3. Números cuyas cifras sumen 15


Proceso 0 Proceso 1 Proceso 2



115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439



115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439



115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

R
u
n

Trabajando

Ocioso (listo comunicar)

Recordatorio:

- Decidir reparto de trabajo
 - Analizar balanceo de carga
 - Analizar uso de recursos
 - Analizar el patrón de comunicaciones
- **Implementar** (codificar para cualquier número de procesos)
- Depurar
- Analizar resultados
- Optimizar ¿?

Recomendación:

- Codificar código secuencial
- **Implementar** (codificar para cualquier número de procesos probablemente partiendo del código secuencial)
- Depurar
- Analizar resultados
- Optimizar ¿?

Aspectos básicos iniciales en la codificación con MPI:

- Al lanzar indicamos el número de procesos a utilizar (un único ejecutable)
- Inicialización (y de MPI)
- Identificación de los procesos
- Finalización (y de MPI)

Recordatorio:

Al ejecutar una aplicación desarrollada con MPI:

- Se crea una copia del ejecutable en cada procesador
- Cada procesador, por tanto, ejecuta el mismo programa
- La ejecución real debe depender tanto del número de procesos totales como del proceso en particular

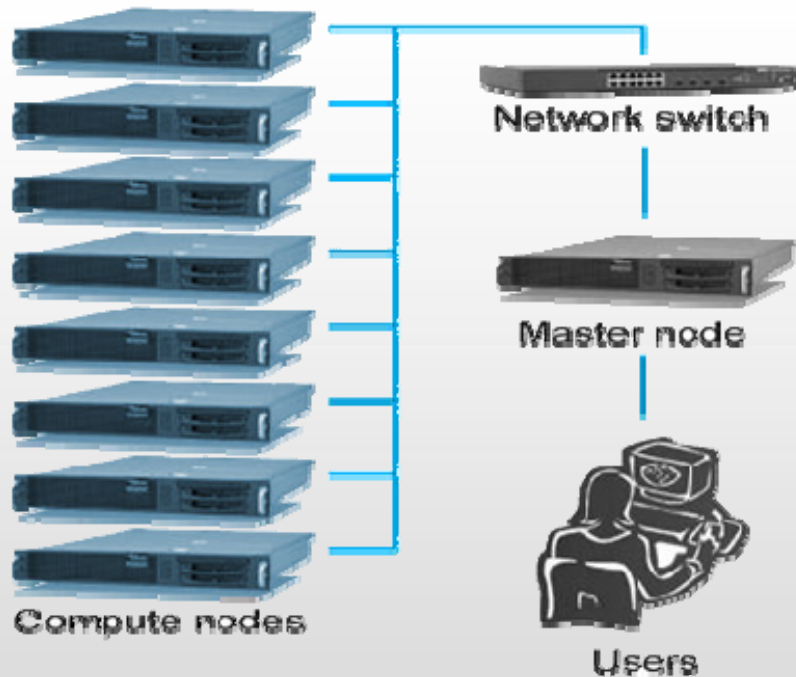
Sistema cluster para computación de altas prestaciones



Sistema cluster para computación de altas prestaciones



Simbólicamente nuestro modelo es:



- Conexión remota a nodo de acceso
- En el nodo de acceso compilamos y lanzamos
- Nuestro programa se ejecuta en los nodos de cómputo

Analicemos:

Sistema de colas: solicitar recursos y ejecutar

¿Puedo usar mi programa los dispositivos de interfaz humana comunes (monitor y teclado)?

II.3. Funciones MPI básicas

- **Inicialización y finalización**
- **Identificación de procesos**
- **Envío y recepción**

Directiva necesaria para trabajar con mpi

```
#include "mpi.h"
```

Incluye todas las definiciones, macros y prototipos de programas necesarios para la aplicación con MPI

Todavía NO se puede hacer llamadas a funciones de MPI (.. y esperar un funcionamiento correcto)

Antes de utilizar funciones MPI - INICIALIZACIÓN

```
ierr = MPI_Init (&argc,&argv)
```

Antes de finalizar el programa hay que finalizar MPI:

```
ierr = MPI_Finalize ()
```

Muchas funciones MPI devuelven un valor entero que retorna el código de error obtenido en la llamada a la función

```
int ierr;
```

```
ierr = MPI_Init (&argc,&argv)
```

Recordemos que argc y argv hacen referencia a los parámetros pasados por línea de comando:

- argc → número de parámetros
- argv → array que contiene los parámetros

Pregunta: ¿En la llamada a MPI_Init estás dos variables se pasan por valor o por referencia?

II.3 FUNCIONES MPI BÁSICAS – Inicialización y Finalización

```
#include <mpi.h>

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    //
    //  PROGRAMA A DESARROLLAR CON
    //  LLAMADAS A RUTINAS MPI
    //
    MPI_Finalize();
}
```

Compilación:

- La distribución de MPI instalada proporcionará los compiladores para los diferentes lenguajes soportados (por ejemplo mpif77, mpicc)
- Estos compiladores añaden una capa a los compiladores instalados en el sistema
- La ejecución tampoco será el simple lanzamiento de un ejecutable (por ejemplo: un lanzamiento implica varias ejecuciones, los demonios que permiten la comunicación deben iniciarse)

COMPILACIÓN (sentencia más simple)

```
mpicc -o ejemplo ejemplo.c
```

Ejecutable → ejemplo

EJECUCIÓN

```
mpirun -np 4 ejemplo
```


Con “mpirun” se crean un grupo de procesos (especificados por el parámetro asociado a `-np`)

Entre los procesos que se crean podemos diferenciar entre el proceso “root” y el resto de procesos.

Los procesos vienen identificados por un número.

El proceso root (o raíz) es el proceso 0

¿Cuántos procesos se están ejecutando?

```
int ierr;
```

```
int nproces;
```

```
ierr = MPI_Comm_size(MPI_COMM_WORLD, &nproces);
```

```
//nproces se pasa por referencia para que devuelva el número  
total de procesos ejecutados
```

```
//nproces coincide con el valor especificado en “-np 4”
```

¿Qué proceso soy “yo” de todos esos procesos?

```
int myrank;
```

```
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
```

```
// myrank se pasa por referencia para que devuelva el identificador de  
proceso
```

```
//IDENTIFICADOR DE PROCESO → entero entre 0 y (nproces-1)
```

¿Qué es `MPI_COMM_WORLD`?

Es un *comunicador* predefinido para el grupo de procesos global.

Pueden crearse otros grupos de procesos (sin crear procesos nuevos) que contengan sólo parte de los procesos.

Cada grupo tendrá un *comunicador* para referenciarlo. En la gran mayoría de funciones MPI un parámetro es el comunicador que selecciona el grupo sobre el cual se va a realizar dicha función.

II.3 FUNCIONES MPI BÁSICAS – Identificación de procesos

```
int main(int argc, char *argv[])
{
    int nproces, myrank, err;
    err = MPI_Init(&argc, &argv);

    err = MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    err = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    printf("Soy el proceso %d de %d procesos.\n", myrank, nproces);

    err = MPI_Finalize();
}
```

Recordatorio: en un sistema HPC con gestor de colas la salida de pantalla se redirecciona a fichero

Dado el ejemplo anterior cuyo ejecutable al compilar es “ejemplo”:

¿cuál será la salida si la ejecución es “**mpirun -np 4 ejemplo**”?

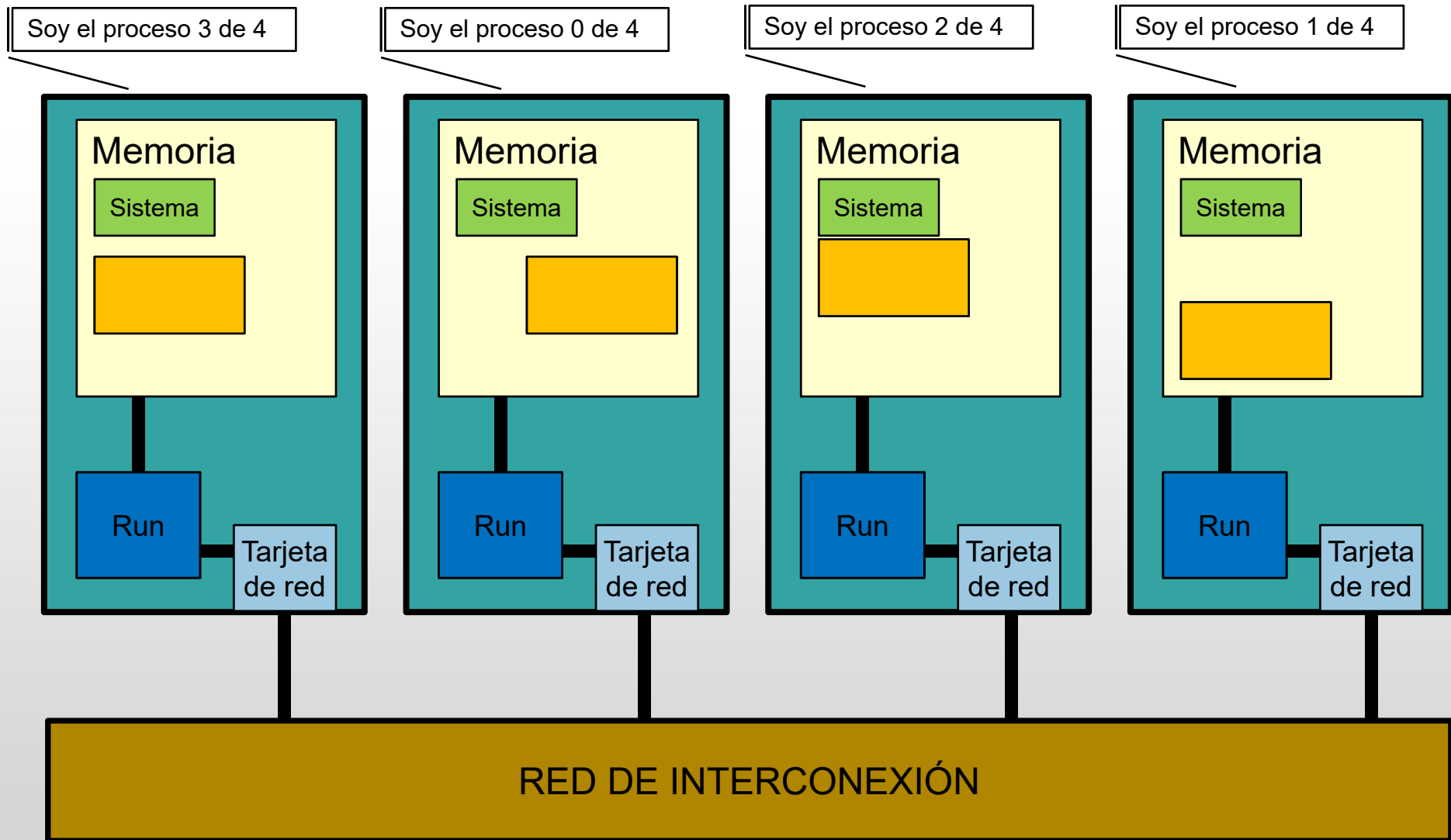
Antes de contestar sabemos:

- La salida será una línea de texto.
- La línea de texto incluye el valor actual de dos variables (myrank y nproces)
- El valor de las variables myrank y nproces se ha actualizado en las llamadas a las funciones MPI

Antes de contestar respondemos:

- ¿Cuántas variables myrank y nproces existen?
- ¿Cuántas líneas de texto hay en la salida?

II.3 FUNCIONES MPI BÁSICAS – Identificación de procesos



Salida completa:

Soy el proceso 0 de 4
Soy el proceso 1 de 4
Soy el proceso 2 de 4
Soy el proceso 3 de 4

Soy el proceso 3 de 4
Soy el proceso 0 de 4
Soy el proceso 2 de 4
Soy el proceso 1 de 4

Soy el proceso 3 de 4
Soy el proceso 2 de 4
Soy el proceso 1 de 4
Soy el proceso 0 de 4

Soy el proceso 3 de 4
Soy el proceso 1 de 4
Soy el proceso 2 de 4
Soy el proceso 0 de 4

Soy el proceso 2 de 4
Soy el proceso 1 de 4
Soy el proceso 3 de 4
Soy el proceso 0 de 4

Soy el proceso 1 de 4
Soy el proceso 0 de 4
Soy el proceso 3 de 4
Soy el proceso 2 de 4

Ejecución SIMULTÁNEA

II.3 FUNCIONES MPI BÁSICAS – Identificación de procesos

```
int main(int argc, char *argv[])
{
    int nproces, myrank, err, i=0;
    MPI_Init(&argc, &argv);

    err = MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    err = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    i++;
    printf("Proceso %d de %d. Mi variable i vale %d\n", myrank, nproces, i);

    MPI_Finalize();
}
```

- La salida será una línea de texto por proceso
- La línea de texto incluye el valor actual de 3 variables (myrank, nproces, i)
- ¿Cuántos incrementos de “i” se producen (mpirun -np 6 ejemplo)?

Salida completa:

Proceso 0 de 6. Mi variable i vale 1
Proceso 1 de 6. Mi variable i vale 1
Proceso 2 de 6. Mi variable i vale 1
Proceso 3 de 6. Mi variable i vale 1
Proceso 4 de 6. Mi variable i vale 1
Proceso 5 de 6. Mi variable i vale 1

Proceso 5 de 6. Mi variable i vale 1
Proceso 2 de 6. Mi variable i vale 1
Proceso 1 de 6. Mi variable i vale 1
Proceso 4 de 6. Mi variable i vale 1
Proceso 0 de 6. Mi variable i vale 1
Proceso 3 de 6. Mi variable i vale 1

Proceso 2 de 6. Mi variable i vale 1
Proceso 3 de 6. Mi variable i vale 1
Proceso 1 de 6. Mi variable i vale 1
Proceso 0 de 6. Mi variable i vale 1
Proceso 5 de 6. Mi variable i vale 1
Proceso 4 de 6. Mi variable i vale 1

Proceso 5 de 6. Mi variable i vale 1
Proceso 0 de 6. Mi variable i vale 1
Proceso 4 de 6. Mi variable i vale 1
Proceso 1 de 6. Mi variable i vale 1
Proceso 3 de 6. Mi variable i vale 1
Proceso 2 de 6. Mi variable i vale 1

Hasta aquí hemos visto 4 de las 6 funciones básicas (2 de inicialización y finalización, 2 de información), con las 2 siguientes de envío y recepción se completan las básicas.

Envío:

```
err = MPI_Send(DatosEnv, NumDatos, TipoDatos, Destino,  
tag, comunicador)
```

Preguntas

¿Qué envío?	Un mensaje
¿Quién envía?	Un proceso
¿Quién recibe?	Otro proceso

Preguntas

¿Qué es un mensaje?
¿Cómo se identifica?
¿Cómo se identifica?

¿Qué es un mensaje?

Es el conjunto de datos almacenados en una zona contigua de memoria.

¿Cómo especifico dichos datos?

Especificando el inicio de la zona de memoria a enviar y el tamaño de la misma en bytes.


¿Cómo especifico el “inicio de la memoria ...”?

Con un puntero, es decir con la dirección de memoria del primer dato a enviar

¿Cómo especifico el “inicio de la memoria ...”?

Con un puntero, es decir con la dirección de memoria del primer dato a enviar

EJEMPLOS:

<code>int ivar;</code>	<code>→ &ivar</code>	<div># es un entero entre 0 y 14</div> 
<code>double dvar;</code>	<code>→ &dvar</code>	
<code>int v_ivar[15];</code>	<code>→ v_ivar ó &v_ivar[0] ó &v_ivar[#]</code>	
<code>double v_dvar[15];</code>	<code>→ v_dvar ó &v_dvar[0] ó &v_dvar[#]</code>	
<code>int *mdi_var;</code>	<code>→ mdi_var ó &mdi_var[0] ó &mdi_var[#]</code>	
<code>double *mdd_var;</code>	<code>→ mdd_var ó &mdd_var[0] ó &mdd_var[#]</code>	

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

`int *mdi_var;` \rightarrow `mdi_var` ó `&mdi_var[0]` ó `&mdi_var[#]`
`double *mdd_var;` \rightarrow `mdd_var` ó `&mdd_var[0]` ó `&mdd_var[#]`

`mdi_var = (int*) malloc (200 * sizeof(int));` // # entero entre 0 y 199

`mdd_var = (double *)malloc (300 * sizeof(double));` // # entero entre 0 y 299

// RECORDEMOS:

//

// `void* malloc (unsigned numero_de_bytes);`

Función de envío:

```
MPI_Send(  
void* DatosEnv,  
int NumDatos,  
MPI_Datatype TipoDatos,  
int Destino,  
int tag,  
MPI_Comm comunicador)
```

¿Cuál es el proceso origen, el que envía los datos? Aquel proceso que ejecuta la función

Mensaje: conjunto de datos almacenados en una zona contigua de memoria

Inicio: DatosEnv

Tamaño: (NumDatos, TipoDatos)

Destino: Identificador (número entero) del proceso que va a recibir los datos.

CUIDADO: Obligatoriamente los tiene que recibir con función de recibir

tag: es una etiqueta (número entero) que debe ser el mismo en la función de envío que en la función de recepción.

comunicador: MPI_COMM_WORLD

MPI_Datatype TipoDatos

MPI_CHAR
MPI_SHORT
MPI_INT
MPI_INTEGER
MPI_INT4
MPI_LONG
MPI_LONG_LONG
MPI_SIGNED_CHAR
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED_LONG
MPI_UNSIGNED
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
...

Recepción:

```
err = MPI_Recv(DatosRec, NumDatos, TipoDatos, Origen,  
              tag, comunicador, status)
```

Preguntas

- | | |
|---------------------|---|
| ¿Qué recibo? | Un mensaje |
| ¿Dónde lo almaceno? | En memoria a partir del puntero <i>DatosRec</i> |
| ¿De qué tamaño? | Se obtiene (NumDatos, TipoDatos) |
| ¿Quién envía? | El proceso con <i>rango</i> Origen |
| ¿Quién recibe? | El proceso que ejecuta la función |

Función de recepción:

```
MPI_Recv(  
void* DatosRec,  
int NumDatos,  
MPI_Datatype TipoDatos,  
int Origen,  
int tag,  
MPI_Comm comunicador,  
MPI_Status* status)
```

status: variable de tipo MPI_Status, pasada por referencia (es puntero a MPI_Status) que almacena información adicional de la recepción que se acaba de realizar

Mensaje: conjunto de datos *que van a ser almacenados* en una zona contigua de memoria

Inicio: DatosRec

Tamaño: (NumDatos, TipoDatos)

Origen: Identificador (número entero) del proceso que envió *o va a enviar* los datos.

CUIDADO: ¿Qué pasa si no los envía nunca?

tag: es una etiqueta (número entero) que debe ser el mismo en la función de recepción que en la función de envío.

comunicador: MPI_COMM_WORLD

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

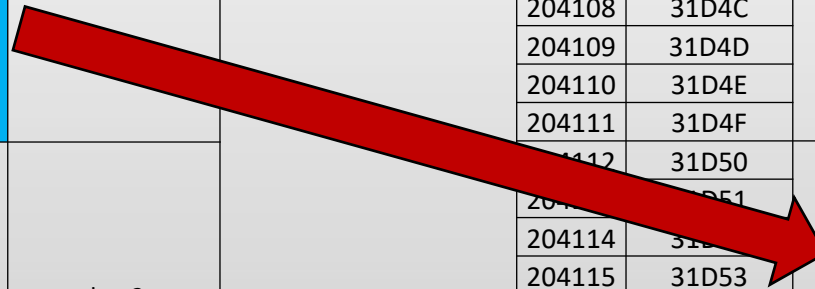
Ejemplo de envío y recepción de un escalar de un proceso dado (0) a otro (3)

Proceso 0

Dirección de memoria			Nombre variable
Decimal	Hexadecimal		
102464	19040	0	myrank
102465	19041		
102466	19042		
102467	19043		
102468	19044	4	nproces
102469	19045		
102470	19046		
102471	19047		
102472	19048	4765,3	data1
102473	19049		
102474	1904A		
102475	1904B		
102476	1904C		
102477	1904D		
102478	1904E		
102479	1904F		
102480	19050	0	data2
102481	19051		
102482	19052		
102483	19053		
102484	19054		
102485	19055		
102486	19056		
102487	19057		
102488	19058		
102489	19059		
102490	1905A		

Proceso 3

Dirección de memoria			Nombre variable
Decimal	Hexadecimal		
204096	31D40	3	myrank
204097	31D41		
204098	31D42		
204099	31D43		
204100	31D44	4	nproces
204101	31D45		
204102	31D46		
204103	31D47		
204104	31D48	0	data1
204105	31D49		
204106	31D4A		
204107	31D4B		
204108	31D4C		
204109	31D4D		
204110	31D4E		
204111	31D4F		
204112	31D50	0	data2
204113	31D51		
204114	31D52		
204115	31D53		
204116	31D54		
204117	31D55		
204118	31D56		
204119	31D57		
204120	31D58		
204121	31D59		
204122	31D5A		



II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de envío y recepción de un escalar de un proceso dado (0) a otro (3)

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    double data1=0, data2=0;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0){
        data1 = 4765.3;
        MPI_Send(&data1, 1, MPI_DOUBLE, 3, 99, MPI_COMM_WORLD);
    }
    else{
        if (myrank == 3)
            MPI_Recv(&data2, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de envío y recepción de un escalar de un proceso dado (0) a otro (3)

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    double data1=0, data2=0;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0){
        data1 = 4765.3;
        MPI_Send(&data1, 1, MPI_DOUBLE, 3, 99, MPI_COMM_WORLD);
    }
    else{
        if (myrank == 3)
            MPI_Recv(&data2, 1, MPI_DOUBLE, 0, 88, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

¿Qué pasa si ... ?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de envío y recepción de un escalar de un proceso dado (0) a otro (3)

```
int main(int argc, char *argv[])
{
    int nprocs, myrank;
    double data1=0, data2=0;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0){
        data1 = 4765.3;
        MPI_Send(&data1, 1, MPI_DOUBLE, 2, 99, MPI_COMM_WORLD);
    }
    else{
        if (myrank == 3)
            MPI_Recv(&data2, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

¿Qué pasa si ... ?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de envío y recepción de un escalar de un proceso dado (0) a otro (3)

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    double data1=0, data2=0;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0){
        data1 = 4765.3;
        MPI_Send(&data1, 1, MPI_DOUBLE, 3, 99, MPI_COMM_WORLD);
    }
    else{
        if (myrank == 3)
            MPI_Recv(&data2, 1, MPI_DOUBLE, 1, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

¿Qué pasa si ... ?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de broadcast de un vector. Proceso que envía: 1

Proceso 1

i_vector1

?	?	?	?	?	?	?	?	3	44
7	27	97	54	43	19	65	73	27	59
87	52	44	13	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

Resto de procesos

i_vector1

82	49	25	2	85	77	67	76	58	58
33	67	21	2	29	49	64	67	53	71
82	15	92	28	89	66	2	93	18	18
1	47	23	28	30	65	27	75	4	56
3	61	56	12	22	44	62	50	11	43
63	79	46	63	82	66	67	47	70	61
89	52	73	62	63	34	43	6	46	93

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de broadcast de un vector. Proceso que envía: 1

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    int i_vector1[16];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //Vector inicializado
    if (myrank == 1){

        MPI_Send(i_vector1, 16, MPI_INTEGER, 0, 99, MPI_COMM_WORLD);

    }
    else{
        MPI_Recv(&i_vector1[0], 16, MPI_INTEGER, 1, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de broadcast de un vector. Proceso que envía: 1

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    int i_vector1[16];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //Vector inicializado
    if (myrank == 1){
        for(int i=0; i<nproces; i++){
            MPI_Send(i_vector1, 16, MPI_INTEGER, i, 99, MPI_COMM_WORLD); }
        }
    else{
        MPI_Recv(&i_vector1[0], 16, MPI_INTEGER, 1, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de broadcast de un vector. Proceso que envía: 1

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    int i_vector1[16];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //Vector inicializado
    if (myrank == 1){
        for(int i=0; i<nproces; i++){
            if (i != 1)
                MPI_Send(i_vector1, 16, MPI_INTEGER, i, 99, MPI_COMM_WORLD);
        }
    }
    else{
        MPI_Recv(&i_vector1[0], 16, MPI_INTEGER, 1, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de broadcast de un vector. Proceso que envía: 1

Proceso 1

i_vector1

?	?	?	?	?	?	?	?	3	44
7	27	97	54	43	19	65	73	27	59
87	52	44	13	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

Resto de procesos

i_vector1

82	49	25	2	85	77	67	76	58	58						
33	67	21	2	29	49	64	67	53	71						
82	15	92	28	89	66	2	93	18	18						
3	44	7	27	97	54	43	19	65	73	30	65	27	75	4	56
27	59	87	52	44	13	62	50	11	43	22	44	62	50	11	43
63	79	46	63	82	66	67	47	70	61	82	66	67	47	70	61
89	52	73	62	63	34	43	6	46	93	63	34	43	6	46	93

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de broadcast de un vector. Proceso que envía: 1

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

```
    int nproces, myrank;
```

TODOS LOS PROCESOS

```
    int i_vector1[16];
```

```
    MPI_Status status;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
    //Vector inicializado
```

```
    if (myrank == 1){
```

```
        for(int i=0; i<nproces; i++){
```

SOLO UN PROCESO

```
            if (i != 1)
```

¿Cuántos envíos?

```
                MPI_Send(i_vector1, 16, MPI_INTEGER, i, 99, MPI_COMM_WORLD); } }
```

```
    }
```

```
    else{
```

```
        MPI_Recv(&i_vector1[0], 16, MPI_INTEGER, 1, 99, MPI_COMM_WORLD, &status);
```

```
    }
```

```
    MPI_Finalize();
```

```
}
```

¿Cuántos procesos? ¿Cuántas recepciones por proceso? ¿Y en total?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de broadcast de un vector. Proceso que envía: 1

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    int i_vector1[16];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //Vector inicializado
    if (myrank == 1){
        for(int i=0; i<nproces; i++){
            if (i != 1)
                MPI_Send(i_vector1, 10, MPI_INTEGER, i, 99, MPI_COMM_WORLD);
        }
    }
    else{
        MPI_Recv(&i_vector1[0], 16, MPI_INTEGER, 1, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

¿Qué pasa si ... ?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de broadcast de un vector. Proceso que envía: 1

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    int i_vector1[16];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //Vector inicializado
    if (myrank == 1){
        for(int i=0; i<nproces; i++){
            if (i != 1)
                MPI_Send(i_vector1, 16, MPI_INTEGER, i, 99, MPI_COMM_WORLD);
        }
    }
    else{
        MPI_Recv(&i_vector1[0], 10, MPI_INTEGER, 1, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

¿Qué pasa si ... ?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de broadcast de un vector. Proceso que envía: 1

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    int i_vector1[16];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //Vector inicializado
    if (myrank == 1){
        for(int i=0; i<nproces; i++){
            if (i != 1)
                MPI_Send(i_vector1, 17, MPI_INTEGER, i, 99, MPI_COMM_WORLD);
        }
    }
    else{
        MPI_Recv(&i_vector1[0], 17, MPI_INTEGER, 1, 99, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

¿Qué pasa si ... ?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de “reducción”. Proceso 0 suma un entero que aportan todos los procesos

```
int main(int argc, char *argv[])
{
    int nproces, myrank;
    MPI_Status status;
    int i_data; //Almacena el dato que se tiene que sumar
    int i_suma; //Almacena la suma de i_data de todos los procesos

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

    .....
```

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplo de “reducción”. Proceso 0 suma un entero que aportan todos los procesos

....

```
if (myrank != 0){  
    MPI_Send(&i_data,1,MPI_INTEGER,0,88,MPI_COMM_WORLD);}  
}  
else{  
    for(int i=1;i<nproces;i++){  
        MPI_Recv(&i_data,1,MPI_INTEGER,i,88,MPI_COMM_WORLD,&status);  
        i_suma += i_data;  
    }  
}  
MPI_Finalize();  
}
```

Hay un error al programar la funcionalidad. No hay errores ni sintácticos ni de MPI. ¿Cuál?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplos consolidación 1

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double data1, data2;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = 0;
    if (myrank == 0) {
        for (i = 1; i < nproces; i++) {
            data1 = i * 10;
            MPI_Send(&data1, 1, MPI_DOUBLE, i, 5, MPI_COMM_WORLD);
        }
    }
    else {
        MPI_Recv(&data2, 1, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD, &status);
    }
    printf("Proceso %d. d1: %f - d2: %f.\n", myrank, data1, data2);
    MPI_Finalize();
}
```

Proceso	data1	data2
0	30	¿?
1	0	10
2	0	20
3	0	30

Proceso	data1	data2
0	50	¿?
1	0	10
2	0	20
3	0	30
4	0	40
5	0	50

¿Cuál es la salida del programa usando 4 y 6 procesos?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplos consolidación 2

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double data1, data2;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = 0;
    if(myrank > 0){
        data1 = myrank * 10;
        MPI_Send(&data1, 1, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD);
    }
    else{
        for (i=1; i<nproces; i++){
            MPI_Recv(&data2, 1, MPI_DOUBLE, i, 5, MPI_COMM_WORLD, &status);
            data1 = data1 + data2;
        }
    }
    printf("Proceso %d. d1: %f - d2: %f.\n", myrank, data1, data2);
    MPI_Finalize();
}
```

Proceso	data1	data2
0	60	30
1	10	¿?
2	20	¿?
3	30	¿?

Proceso	data1	data2
0	150	50
1	10	¿?
2	20	¿?
3	30	¿?
4	40	¿?
5	50	¿?

¿Cuál es la salida del programa usando 4 y 6 procesos?

II.3 FUNCIONES MPI BÁSICAS – E

Ejemplos consolidación 3a

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double *data1;
    double *data2;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = (double*) malloc(20 * sizeof(double));
    data2 = (double*) malloc(20 * sizeof(double));
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
    if (myrank == 2) {
        for (i=0; i<nproces; i++) {
            if (i!=2) MPI_Send(data1, 2, MPI_DOUBLE, i, 5, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(data2, 2, MPI_DOUBLE, 2, 5, MPI_COMM_WORLD, &status);
    }
    //¿Valores de data1 y data2 en cada proceso en una ejecución con 4 y 6 procesos?
    MPI_Finalize();
}
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	2	4	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	2	4	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	2	4	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	2	4	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	2	4	62	68	74	80	86	92	98	104	110

II.3 FUNCIONES MPI BÁSICAS – E

Ejemplos consolidación 3

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double *data1;
    double *data2;;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = (double*) malloc(20 * sizeof(double));
    data2 = (double*) malloc(20 * sizeof(double));
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
    if(myrank == 2){
        for (i=0; i<nproces; i++){
            if (i!=2) MPI_Send(data1, 2, MPI_DOUBLE, i, 5, MPI_COMM_WORLD);
        }
    }
    else{
        MPI_Recv(&data2[myrank], 2, MPI_DOUBLE, 2, 5, MPI_COMM_WORLD, &status);
    }
    //¿Valores de data1 y data2 en cada proceso en una ejecución con 4 y 6 procesos?
    MPI_Finalize();
}
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	2	4	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	2	4	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	2	4	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	40	45	50	55	2	4	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	50	56	62	68	74	2	4	92	98	104	110

II.3 FUNCIONES MPI BÁSICAS – Envío

Ejemplos consolidación 4

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double *data1;
    double *data2;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = (double*) malloc(20 * sizeof(double));
    data2 = (double*) malloc(20 * sizeof(double));
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
    if (myrank == 2) {
        for (i=0; i<nproces; i++) {
            if (i!=2) MPI_Send(&data1[i+1], 2, MPI_DOUBLE, i, 5,
                               MPI_COMM_WORLD);
        }
    }
    else {
        MPI_Recv(data2, 2, MPI_DOUBLE, 2, 5,
                 MPI_COMM_WORLD, &status);
    }
    //¿Valores de data1 y data2 en cada proceso en una ejecución con 4 y 6 procesos?
    MPI_Finalize();
}
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	4	6	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	6	8	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	10	12	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	12	14	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	14	16	62	68	74	80	86	92	98	104	110

II.3 FUNCIONES MPI BÁSICAS – E

Ejemplos consolidación 5

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

```
    int nproces, myrank, i;
```

```
    double *data1;
```

```
    double *data2;;
```

```
    MPI_Status status;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
    data1 = (double*) malloc(20 * sizeof(double));
```

```
    data2 = (double*) malloc(20 * sizeof(double));
```

```
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
```

```
    if(myrank > 0){
```

```
        MPI_Send(&data1[2*myrank], 2, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD);}
```

```
    }
```

```
    else{
```

```
        for (i=1; i<nproces; i++){
```

```
            MPI_Recv(&data2[2*i], 2, MPI_DOUBLE, i, 5, MPI_COMM_WORLD, &status);
```

```
        }
```

```
        //¿Valores de data1 y data2 en cada proceso en una ejecución con 4 y 6 procesos?
```

```
    MPI_Finalize();
```

```
}
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	3	4	10	12	21	24	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplos consolidación 6

Proceso 0 **Proceso 1** **Proceso 2**
Todos tareas 1, 2 y 3

Bloque

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

Clase 0:

Para todos los elementos de una matriz contar:

1. Elementos pares
2. Elementos impares
3. Números cuyas cifras sumen 15

Implementar:

1. Matriz es un vector 144 elementos
2. Opcional: Matriz es bidimensional 12 x 12 (puntero doble).
3. La salida debe ser realizada SOLO por el proceso 0, el cual imprimirá los valores parciales obtenidos por cada proceso y los valores finales

El envío y recepción (que acabamos de ver) son bloqueantes (**trabajo cooperativo**), esto hace que cada envío deba recepcionarse en su totalidad, en caso contrario el programa no “evoluciona”, es decir está mal programado.

Tal y como lo hemos programado la recepción debe coincidir el proceso que envía y el “tag”, en caso contrario está mal programado.

RECORDEMOS: tenemos un mismo programa que ejecuta un conjunto de instrucciones diferente en función del proceso.

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

Ejemplos consolidación 2

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double data1, data2;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = 0;
    if(myrank > 0){
        data1 = myrank * 10;
        MPI_Send(&data1, 1, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD);
    }
    else{
        for (i=1; i<nproces; i++){
            MPI_Recv(&data2, 1, MPI_DOUBLE, i, 5, MPI_COMM_WORLD, &status);
            data1 = data1 + data2;
        }
    }
    MPI_Finalize();
}
```

En el programa anterior el root debe recibir los datos enviados por los “slaves” o esclavos (es decir cualquier proceso que no es el root) en order de “rango” (es decir de myrank)

Esto puede provocar un overhead (añadido al de las comunicaciones bloqueantes) ¿Cuál?

¿Cómo podríamos solucionarlo?

Vamos a usar: **MPI_ANY_SOURCE**

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double data1, data2;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = 0;
    if(myrank > 0){
        data1 = myrank * 10;
        MPI_Send(&data1, 1, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD);
    }
    else{
        for (i=1; i<nproces; i++){
            MPI_Recv(&data2, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 5, MPI_COMM_WORLD, &status);
            data1 = data1 + data2;
        }
    }
    MPI_Finalize();
}
```

¿Qué cambiaría por usar MPI_ANY_SOURCE?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double data1, data2;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = 0;
    if(myrank > 0){
        data1 = myrank * 10;
        MPI_Send(&data1, 1, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD);
    }
    else{
        for (i=1; i<nproces; i++){
            MPI_Recv(&data2, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 5, MPI_COMM_WORLD, &status);
            data1 = data1 + data2;
        }
    }
    ...
    // la línea siguiente es ejecutada por el proceso 1
    MPI_Send(&data1, 1, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
}
```

Posible ERROR no
DETERMINISTICO.
¿Cuál?

II.3 FUNCIONES MPI BÁSICAS – Envío y recepción

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double data1, data2;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = 0;
    if(myrank > 0){
        data1 = myrank * 10;
        MPI_Send(&data1, 1, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD);
    }
    else{
        for (i=1; i<nproces; i++){
            MPI_Recv(&data2, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 5, MPI_COMM_WORLD, &status);
            data1 = data1 + data2;
        }
    }
    ...
    // la línea siguiente es ejecutada por el proceso 1
    MPI_Send(&data1, 1, MPI_DOUBLE, 0, 101, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
}
```

Similar al funcionamiento de MPI_ANY_SOURCE, podemos usar MPI_ANY_TAG

En ambos casos SÓLO puede utilizarse en las funciones de recepción

En ambos casos debe comprobarse que el resultado es SIEMPRE correcto, no podemos realizar una única prueba y si el resultado es correcto dar el código cómo bueno.

“status” variable de tipo MPI_STATUS es una estructura con los siguientes campos

MPI_SOURCE

MPI_TAG

MPI_ERROR

Que contienen información de la recepción que acaba de producirse y que pueden ser desconocidos

TODO LO NECESARIO PARA TRABAJAR CON MPI ESTÁ VISTO

No obstante no sólo es importante la velocidad del código sino que en ocasiones también es importante la velocidad de desarrollo y la velocidad de depuración y/o actualización

II.4 FUNCIONES MPI COLECTIVAS

Aspectos claves de las comunicaciones colectivas:

- Participan todos los procesos (del grupo)
- Todos los procesos llaman a la misma función
- Todo mensaje enviado debe ser recibido
- Están basadas en las funciones Send y Receive
- Facilitan el desarrollo y claridad de los códigos

¿Qué identifica a un grupo?

Para unos implica envío y para otros recepción

¿Qué sucede si no se cumple?

Entonces ¿Aceleran?

Aceleran desarrollo y mantenimiento

II.4 FUNCIONES MPI COLECTIVAS

La función broadcast (como cualquier broadcast) envía un mensaje de un proceso al resto de procesos.

Un broadcast implica un envío para uno de los procesos y una recepción para el resto.

El envío realizado con un broadcast ha de recibirse con un broadcast (no con un receive)

II.4 FUNCIONES MPI COLECTIVAS - BCAST

```
err = MPI_Bcast(  
void* Datos,  
int NumDatos,  
MPI_Datatype TipoDatos,  
int Raíz,  
MPI_Comm comunicador)
```

Recordamos en envío “¿Cuál es el proceso origen, el que envía los datos? Aquel proceso que ejecuta la función”

Aquí todos ejecutan la misma función

Mensaje: conjunto de datos de zona contigua de memoria
Inicio: Datos
Tamaño: (NumDatos, TipoDatos)

Raíz: Identificador (número entero) del proceso propietario de los datos.
CUIDADO: Obligatoriamente se tienen que recibir con la misma función

tag: ¿Qué ha pasado con el tag?.

comunicador: MPI_COMM_WORLD

II.4 FUNCIONES MPI COLECTIVAS - BCAST

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    int data1, data2, err;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    data1 = 0;
    data2 = 0;
    if(myrank > 0){
        MPI_Recv(&data1, 1, MPI_INT, 0, 5, MPI_COMM_WORLD, &status);
    }
    else{
        data1 = 33;
        for (i=1; i<nproces; i++){
            MPI_Send(&data1, 1, MPI_INT, i, 5, MPI_COMM_WORLD);
        }
    }
    printf("Proceso %d. d1: %d - d2: %d.\n", myrank, data1, data2);
    MPI_Finalize();
}
```

El mismo código usando Broadcast

II.4 FUNCIONES MPI COLECTIVAS - BCAST

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
data1 = 0;
data2 = 0;
if(myrank > 0){
    err = MPI_Bcast(&data1,1,MPI_INTEGER,0,MPI_COMM_WORLD);
}
else{
    data1 = 33;
    err = MPI_Bcast(&data1,1,MPI_INTEGER,0,MPI_COMM_WORLD);
}
printf("Proceso %d. d1: %d - d2: %d.\n",myrank,data1,data2);
MPI_Finalize();
}
```

II.4 FUNCIONES MPI COLECTIVAS - BCAST

O más compacto

```
data1 = 0;  
data2 = 0;  
if (myrank == 0) data1 = 33;  
err = MPI_Bcast(&data1,1,MPI_INTEGER,0,MPI_COMM_WORLD);  
printf("Proceso %d. d1: %d - d2: %d.\n",myrank,data1,data2);  
MPI_Finalize();
```


II.4 FUNCIONES MPI COLECTIVAS - BCAST

```
int main(int argc, char *argv[])
{
    int nprocs, myrank, i;
    double *data1;
    double *data2;

    ...
    data1 = (double*) malloc(20 * sizeof(double));
    data2 = (double*) malloc(20 * sizeof(double));
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
    MPI_Bcast(data1, 3, MPI_DOUBLE, 1, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
}
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	1	2	3	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	1	2	3	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	1	2	3	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	1	2	3	16	20	24	28	32	36	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	1	2	3	20	25	30	35	40	45	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

II.4 FUNCIONES MPI COLECTIVAS - BCAST

```
int main(int argc, char *argv[])
{
    int nproc, myrank, i;
    double *data1;
    double *data2;

    ...
    data1 = (double*) malloc(20 * sizeof(double));
    data2 = (double*) malloc(20 * sizeof(double));
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
    MPI_Bcast(&data1[4], 5, MPI_DOUBLE, 3, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
}
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	15	18	21	24	27	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	15	18	21	24	27	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	15	18	21	24	27	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	15	18	21	24	27	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	15	18	21	24	27	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

II.4 FUNCIONES MPI COLECTIVAS - BCAST

```
int main(int argc, char *argv[])
{
    int nprocs, myrank, i;
    double *data1;
    double *data2;

    ...
    data1 = (double*) malloc(20 * sizeof(double));
    data2 = (double*) malloc(20 * sizeof(double));
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
    MPI_Bcast(&data1[4], 5, MPI_DOUBLE, 3, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
}
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	15	18	21	24	27	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	15	18	21	24	27	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	15	18	21	24	27	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	15	18	21	24	27	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	15	18	21	24	27	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

II.4 FUNCIONES MPI COLECTIVAS - BCAST

```
int main(int argc, char *argv[])
{
    int nproc, myrank, i;
    double *data1;
    double *data2;

    ...
    data1 = (double*) malloc(20 * sizeof(double));
    data2 = (double*) malloc(20 * sizeof(double));
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
    if (myrank == 0)
        MPI_Bcast(data1, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    else
        MPI_Bcast(&data1[2], 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
}
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	3	4	5	6	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	6	8	10	12	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	9	12	15	18	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	12	16	20	24	28	32	36	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	15	20	25	30	35	40	45	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	0	0	0	0	0	0	0	0	0	0	0
	data2	0	1	2	3	4	5	6	7	8	9	10
1	data1	1	2	0	0	0	0	7	8	9	10	11
	data2	10	12	14	16	18	20	22	24	26	28	30
2	data1	2	4	0	0	0	0	14	16	18	20	22
	data2	20	23	26	29	32	35	38	41	44	47	50
3	data1	3	6	0	0	0	0	21	24	27	30	33
	data2	30	34	38	42	46	50	54	58	62	66	70
4	data1	4	8	0	0	0	0	28	32	36	40	44
	data2	40	45	50	55	60	65	70	75	80	85	90
5	data1	5	10	0	0	0	0	35	40	45	50	55
	data2	50	56	62	68	74	80	86	92	98	104	110

II.4 FUNCIONES MPI COLECTIVAS - BCAST

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double *data1;
    double *data2;
    ...
    data1 = (double*) malloc(20 * sizeof(double));
    data2 = (double*) malloc(20 * sizeof(double));
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
    if (myrank == 0)
        MPI_Bcast(data1, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    else
        MPI_Bcast(data2, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
}
```

**ERROR: Obligatoriamente los buffers
tienen que tener el mismo nombre**

II.4 FUNCIONES MPI COLECTIVAS - BCAST

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i;
    double *data1;
    double *data2;
    ...
    data1 = (double*) malloc(20 * sizeof(double));
    data2 = (double*) malloc(20 * sizeof(double));
    //data1 → (i+1)*myrank ; data2 → (i+10)*myrank + i
    if (myrank != 3)
        MPI_Bcast(data1, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
}
```

ERROR de comunicaciones incompletas
¿Cómo se podría implementar?

II.4 FUNCIONES MPI COLECTIVAS - REDUCE

La función REDUCE implica un envío de todos los procesadores a uno.

El proceso que recibe realiza una operación con los datos y lo guarda en una variable que le pertenece.

En la llamada especificamos la operación a realizar entre un conjunto de operaciones disponibles

II.4 FUNCIONES MPI COLECTIVAS - REDUCE

```
MPI_Reduce(  
void* Operando,  
void* Resultado,  
int NumDatos,  
MPI_Datatype TipoDatos,  
MPI_Op Operación,  
int Raíz,  
MPI_Comm comunicador)
```

¿Qué procesos envían datos?

¿Qué proceso recibe datos?

¿Qué datos recibe?

Mensaje: conjunto de datos almacenados en una zona contigua de memoria

Inicio: Operando y Resultado

Tamaño: (NumDatos, TipoDatos)

Raíz: Identificador (número entero) del proceso que almacenará el resultado.

tag: ¿Qué ha pasado con el tag?

comunicador: MPI_COMM_WORLD

El resultado en un determinado grupo, con unos determinados valores ¿depende de que proceso sea el raíz?

II.4 FUNCIONES MPI COLECTIVAS - REDUCE

Operaciones:

MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_LAND
MPI_BAND
MPI_LOR
MPI_BOR
MPI_LXOR
MPI_BXOR
MPI_MAXLOC (máximo y localización)*
MPI_MINLOC (máximo y localización)*

***Trabajan con pares de valores**

Para usar MINLOC o MAXLOC se debe generar una estructura:

```
struct {  
    double datos;  
    int     rango;  
} est1[30], est2[30];
```

datos: que se comparan

rango: se asigna igual al rango

II.4 FUNCIONES MPI COLECTIVAS - REDUCE

```

int main(int argc, char *argv[])
{
    int nprocs, myrank, i, err, len;
    double data1[10], data2[10];
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    //Obtención de datos de la ejecución paralela
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    //No lo hemos visto hasta ahora
    err = MPI_Get_processor_name(name, &len);
    printf("Proceso %d en procesador %s\n", myrank, name);
    //Inicialización de datos
    for (i=0; i<10; i++){
        data1[i] = 100 * (myrank + 1) + 10*i;
        data2[i] = 0;
    }
    //Función MPI
    err = MPI_Reduce(data1, data2, 2, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    //Impresión en pantalla (archivo de salida en nuestro caso)
    for (i=0; i<6; i++){
        printf("Proceso %d. d1: %f - d2: %f.\n", myrank, data1[i], data2[i]);
    }
    MPI_Finalize();
}

```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	100	110	120	130	140	150	160	170	180	190	200
	data2	0	0	0	0	0	0	0	0	0	0	0
1	data1	200	210	220	230	240	250	260	270	280	290	300
	data2	0	0	0	0	0	0	0	0	0	0	0
2	data1	300	310	320	330	340	350	360	370	380	390	400
	data2	0	0	0	0	0	0	0	0	0	0	0
3	data1	400	410	420	430	440	450	460	470	480	490	500
	data2	0	0	0	0	0	0	0	0	0	0	0

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	100	110	120	130	140	150	160	170	180	190	200
	data2	1000	1040	0	0	0	0	0	0	0	0	0
1	data1	200	210	220	230	240	250	260	270	280	290	300
	data2	0	0	0	0	0	0	0	0	0	0	0
2	data1	300	310	320	330	340	350	360	370	380	390	400
	data2	0	0	0	0	0	0	0	0	0	0	0
3	data1	400	410	420	430	440	450	460	470	480	490	500
	data2	0	0	0	0	0	0	0	0	0	0	0

Escribe la salida si fueran 4 procesos

II.4 FUNCIONES MPI COLECTIVAS - ALLREDUCE

La función ALLREDUCE implica un envío de todos los procesadores a uno inicial para realizar un REDUCE.

El funcionamiento simula que:

1. El proceso que recibe realiza una operación con los datos y lo guarda en una variable que le pertenece.
2. El proceso acaba con un broadcast

II.4 FUNCIONES MPI COLECTIVAS - ALLREDUCE

```
MPI_Allreduce(  
void* Operando,  
void* Resultado,  
int NumDatos,  
MPI_Datatype TipoDatos,  
MPI_Op Operación,  
MPI_Comm comunicador)
```

Mensaje: conjunto de datos almacenados
en una zona contigua de memoria

Inicio: Operando y Resultado

Tamaño: (NumDatos, TipoDatos)

Raíz: ¿Qué ha pasado con Raíz?

¿Qué procesos envían datos?

¿Qué procesos reciben datos?

¿Qué datos reciben?

II.4 FUNCIONES MPI COLECTIVAS - ALLREDUCE

```
int main(int argc, char *argv[])
{
    int nproces, myrank, i, err, len;
    double data1[10], data2[10];
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproces);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    err = MPI_Get_processor_name(name, &len);
    printf("Proceso %d en procesador %s\n", myrank, name);
    //Iniciación de datos
    for (i=0; i<10; i++){
        data1[i] = 100 * (myrank + 1) + 10*i;
        data2[i] = 0;
    }
    //Llamada a la función
    err = MPI_Allreduce(data1, data2, 10, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    //Salida
    for (i=0; i<10; i++){
        printf("Proceso %d. d1: %f - d2: %f.\n", myrank, data1[i], data2[i]);
    }

    MPI_Finalize();
}
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	100	110	120	130	140	150	160	170	180	190	200
	data2	0	0	0	0	0	0	0	0	0	0	0
1	data1	200	210	220	230	240	250	260	270	280	290	300
	data2	0	0	0	0	0	0	0	0	0	0	0
2	data1	300	310	320	330	340	350	360	370	380	390	400
	data2	0	0	0	0	0	0	0	0	0	0	0
3	data1	400	410	420	430	440	450	460	470	480	490	500
	data2	0	0	0	0	0	0	0	0	0	0	0
4	data1	500	510	520	530	540	550	560	570	580	590	600
	data2	0	0	0	0	0	0	0	0	0	0	0

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	100	110	120	130	140	150	160	170	180	190	200
	data2	1500	1550	1600	0	0	0	0	0	0	0	0
1	data1	200	210	220	230	240	250	260	270	280	290	300
	data2	1500	1550	1600	0	0	0	0	0	0	0	0
2	data1	300	310	320	330	340	350	360	370	380	390	400
	data2	1500	1550	1600	0	0	0	0	0	0	0	0
3	data1	400	410	420	430	440	450	460	470	480	490	500
	data2	1500	1550	1600	0	0	0	0	0	0	0	0
4	data1	500	510	520	530	540	550	560	570	580	590	600
	data2	1500	1550	1600	0	0	0	0	0	0	0	0

Escribe la salida 5 procesos

II.4 FUNCIONES MPI COLECTIVAS - SCAN

La función SCAN implica un envío de cada proceso a un conjunto de procesos.

Todos los procesos realizan una operación (de reducción) con un conjunto de datos (que es diferente para cada proceso) y lo guarda en una variable que le pertenece.

El conjunto de datos recibidos y sobre los que se opera es de 0 ... myrank

II.4 FUNCIONES MPI COLECTIVAS - SCAN

```
err = MPI_Scan(  
void* Operando,  
void* Resultado,  
int NumDatos,  
MPI_Datatype TipoDatos,  
MPI_Op Operación,  
MPI_Comm comunicador)
```

Mensaje: conjunto de datos almacenados
en una zona contigua de memoria
Inicio: Operando y Resultado
Tamaño: (NumDatos, TipoDatos)

Raíz: ¿Qué ha pasado con Raíz?

¿Qué procesos envían datos?

¿Qué procesos reciben datos?

¿Cuántos datos reciben?

II.4 FUNCIONES MPI COLECTIVAS - SCAN

```
MPI_Comm_size(MPI_COMM_WORLD,&nproces);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
//Inicialización de datos
for (i=0;i<10;i++){
    data1[i] = 100 * (myrank +1) + 10*i;
    data2[i] = 0;
}
err = MPI_Scan(data1,data2,3,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
printf("Proceso %d. d1: %f - d2: %f.\n",myrank,data1,data2);
MPI_Finalize();
```

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	100	110	120	130	140	150	160	170	180	190	200
	data2	0	0	0	0	0	0	0	0	0	0	0
1	data1	200	210	220	230	240	250	260	270	280	290	300
	data2	0	0	0	0	0	0	0	0	0	0	0
2	data1	300	310	320	330	340	350	360	370	380	390	400
	data2	0	0	0	0	0	0	0	0	0	0	0
3	data1	400	410	420	430	440	450	460	470	480	490	500
	data2	0	0	0	0	0	0	0	0	0	0	0

		i										
		0	1	2	3	4	5	6	7	8	9	10
0	data1	100	110	120	130	140	150	160	170	180	190	200
	data2	100	110	120	0	0	0	0	0	0	0	0
1	data1	200	210	220	230	240	250	260	270	280	290	300
	data2	300	320	340	0	0	0	0	0	0	0	0
2	data1	300	310	320	330	340	350	360	370	380	390	400
	data2	600	630	660	0	0	0	0	0	0	0	0
3	data1	400	410	420	430	440	450	460	470	480	490	500
	data2	1000	1040	1080	0	0	0	0	0	0	0	0

Escribe la salida si fueran 4 procesos

II.4 FUNCIONES MPI COLECTIVAS - GATHER

La función GATHER es una “recolección”, por tanto implica el envío de datos de todos los procesos a uno dado.

Cada proceso envía un conjunto de datos (diferentes porque pertenecen a procesos distintos), que son almacenados en el raíz en posiciones diferentes (lógico para no sobrescribir datos).

Los datos enviados y posición de almacenamiento depende del rango del proceso

No hay proceso de reducción, el volumen de datos recibidos depende del número de procesos de la ejecución

II.4 FUNCIONES MPI COLECTIVAS - GATHER

```
err = MPI_Gather(  
void*  
int  
MPI_Datatype  
void*  
int  
MPI_Datatype  
int  
MPI_Comm  
DatoEnvio,  
NumDatoEnvio,  
TipoDatoEnvio,  
DatoRecepcion,  
NumDatoRecepcion,  
Tipodatosrecepcion,  
Raíz,  
comunicador)
```

Mensaje: Dos mensajes en uno residen los datos a enviar y en el otro se almacenarán los datos recibidos

Inicio: DatosEnvio y DatosRecepción

Tamaño: (NumDatos____, TipoDatos____ y)

¿Qué procesos envían datos?

¿Qué proceso recibe datos?

¿Cuántos datos recibe?

Raíz: Identificador (número entero) del proceso que almacenará el conjunto de datos APORTADOS por TODOS los procesos.

Todos los procesos APORTAN la misma cantidad de datos.

II.4 FUNCIONES MPI COLECTIVAS - GATHER

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nproces);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
// send y receive son punteros con memoria reservada para enteros
for (i=0;i<20;i++){
    send[i] = 10 * (myrank +1) + i;
    receive[i]=0;
}
err = MPI_Gather(send,2,MPI_INTEGER,receive,2,MPI_INTEGER,0,MPI_COMM_WORLD);
for (i=0;i<8;i++){
    printf("Proceso %d. send: %d - receive: %d.\n",myrank,send[i],receive[i]);
}
MPI_Finalize();
```

Escribe la salida si fueran 4 procesos

Escribe la salida si el “raíz” fuera el proceso 1

II.4 FUNCIONES MPI COLECTIVAS - GATHER

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

Todos los procesos
APORTAN la misma
cantidad de datos

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

La cantidad total
depende del número
de procesos

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	¿?	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	¿?	10	11	20	21	30	31	40	41	0	0	0	0	0	0	0	0	0	0	0	0

¿Cuántas
comunicaciones?

En el ejemplo el
proceso raíz es el 0

II.4 FUNCIONES MPI COLECTIVAS - GATHERV

Antes de ver el Gatherv

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	¿?	10	11	12	13	20	21	30	31	32	33	34	35	40	41	42	0	0	0	0	0

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	¿?	10	11	0	0	20	21	30	31	0	0	0	0	40	41	0	0	0	0	0	0

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	¿?	0	0	10	11	12	0	0	20	21	0	30	31	0	0	0	40	0	0	0	0

II.4 FUNCIONES MPI COLECTIVAS - GATHERV

La función GATHERV es una “recolección” similar a GATHER, pero ...

... los datos NO tienen porque ser recogidos en posiciones contiguas, y ...

... NO todos los procesos aportan la misma cantidad de datos.

Entonces, ¿Cómo se especifica esta nueva estrategia?

O inicialmente ¿Cuántos datos necesito especificar

II.4 FUNCIONES MPI COLECTIVAS - GATHERV

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	¿?	0	0	10	11	12	0	0	20	21	0	30	31	0	0	0	40	0	0	0	0

Tengo que especificar la cantidad de datos que aporta cada proceso
(si son 4 especifico 4 cantidades, si son 6 especifico 6,)

Tengo que especificar dónde se almacena cada aporte
(si son 4 especifico 4 cantidades, si son 6 especifico 6,)

- ¿Qué necesitan saber los procesos que aportan?
- ¿Cuántos procesos aportan?
- ¿Qué necesita saber el proceso que recibe?

II.4 FUNCIONES MPI COLECTIVAS - GATHERV

```
err = MPI_Gatherv(  
void*          DatosEnvio,  
int            NumDatosEnvio,  
MPI_Datatype   TipoDatosEnvio,  
void*          DatosRecepcion,  
int*           NumDatosRecepcion,  
int*           Desplazamientos,  
MPI_Datatype   Tipodatosrecepcion,  
int            Raíz,  
MPI_Comm       comunicador)
```

Los datos no son recogidos en posiciones contiguas.

NumDatosRecepcion y Desplazamientos son ARRAYS, que deben especificar los datos necesarios vistos.

II.4 FUNCIONES MPI COLECTIVAS - GATHERV

NumDatosRecepcion

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

	0	1	2	3
NumDatosRecepcion	4	2	6	3

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

	0	1	2	3
NumDatosRecepcion	2	2	2	2

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

	0	1	2	3
NumDatosRecepcion	3	2	2	1

II.4 FUNCIONES MPI COLECTIVAS - GATHERV

Desplazamientos

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

	0	1	2	3
Desplazamientos	0	4	6	12

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	¿?	10	11	12	13	20	21	30	31	32	33	34	35	40	41	42	0	0	0	0	0

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

	0	1	2	3
Desplazamientos	0	4	6	12

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	¿?	10	11	0	0	20	21	30	31	0	0	0	0	40	41	0	0	0	0	0	0

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	¿?	0	0	10	11	12	0	0	20	21	0	30	31	0	0	0	40	0	0	0	0

	0	1	2	3
Desplazamientos	2	7	10	15

II.4 FUNCIONES MPI COLECTIVAS - GATHERV

```
MPI_Comm_size(MPI_COMM_WORLD,&nproces);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
err = MPI_Get_processor_name(name, &len);
printf("Proceso %d en procesador %s\n",myrank,name);
receive_cnt = (int*)malloc(nproces*sizeof(int));
desp_cnt = (int*)malloc(nproces*sizeof(int));
for (i=0;i<20;i++){
    send[i] = 10 * (myrank + 1) + i;
    receive[i]=0; }
for (i=0;i<4;i++){
    receive_cnt[i]=2;
    desp_cnt[i]=2*i;
}
err = MPI_Gatherv(send,2,MPI_INTEGER,receive,receive_cnt,desp_cnt,MPI_INTEGER,3,
    MPI_COMM_WORLD);
MPI_Finalize();
```

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	3	10	11	20	21	30	31	40	41	0	0	0	0	0	0	0	0	0	0	0	0

Escribe la salida si fueran 4 procesos

II.4 FUNCIONES MPI COLECTIVAS - SCATTER

La función SCATTER es una “dispersión”, por tanto implica el envío de datos de un proceso al resto.

El proceso que envía (“raíz”) envía datos diferentes a cada proceso (si no sería un broadcast), los datos enviados dependen del rango del proceso.

Los datos enviados son almacenados en la variable pasada para ello sin depender para nada del rango del proceso receptor

II.4 FUNCIONES MPI COLECTIVAS - SCATTER

```
err = MPI_Scatter(  
void*  
int  
MPI_Datatype  
void*  
int  
MPI_Datatype  
int  
MPI_Comm  
DatosEnvio,  
NumDatosEnvio,  
TipoDatosEnvio,  
DatosRecepcion,  
NumDatosRecepcion,  
Tipodatosrecepcion,  
Raíz,  
comunicador)
```

Mensaje: Dos mensajes en uno residen los datos a enviar y en el otro se almacenarán los datos recibidos

Inicio: DatosEnvio y DatosRecepción

Tamaño: (NumDatos____, TipoDatos____ y)

¿Qué proceso envía datos?

Raíz: Identificador (número entero) del proceso que dispone de los datos en su memoria.

¿Qué procesos reciben datos?

Los datos serán distribuidos entre TODOS los procesos.

¿Cuántos datos se reciben?

II.4 FUNCIONES MPI COLECTIVAS - SCATTER

```
MPI_Comm_size(MPI_COMM_WORLD,&nproces);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
err = MPI_Get_processor_name(name, &len);
printf("Proceso %d en procesador %s\n",myrank,name);
for (i=0;i<20;i++){
    receive[i]=0;
    send[i] = 10 * (myrank +1) + i;
}
```

```
MPI_Scatter(send,2,MPI_INTEGER,receive,2,MPI_INTEGER,0,MPI_COMM_WORLD
```

```
    for (i=0;i<8;i++){
        printf("Proceso %d. send: %d - receive: %d.\n",myrank,send[i],receive[i]);
    }
    MPI_Finalize();
```

Escribe la salida si fueran 4 procesos

Escribe la salida si el “raíz” fuera el proceso 1 y 5 procesos

II.4 FUNCIONES MPI COLECTIVAS - SCATTER

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

Todos los procesos
RECIBEN la misma
cantidad de datos

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

La cantidad total de datos de ORIGEN depende del número de procesos (y de la cantidad enviada a cada proceso)

[illegible][illegible]

II.4 FUNCIONES MPI COLECTIVAS - SCATTER

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
	Proceso 4	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
	Proceso 4	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69

[illegible][illegible]

II.4 FUNCIONES MPI COLECTIVAS - SCATTER

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
	Proceso 4	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	Proceso 0	23	24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Proceso 1	25	26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Proceso 2	27	28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Proceso 3	29	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Proceso 4	31	32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

¿Qué cambio en el código?

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	Proceso 0	23	24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Proceso 1	0	25	26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Proceso 2	0	0	27	28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Proceso 3	0	0	0	29	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Proceso 4	0	0	0	0	31	32	0	0	0	0	0	0	0	0	0	0	0	0	0	0

¿Qué cambio en el código?

II.4 FUNCIONES MPI COLECTIVAS - SCATTERV

Antes de ver el Scatterv

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

[illegible]

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

[illegible]

II.4 FUNCIONES MPI COLECTIVAS - SCATTERV

```
err = MPI_Scatterv(  
void*          DatosEnvio,  
int*          NumDatosEnvio,  
int*          Desplazamientos,  
MPI_Datatype  TipoDatosEnvio,  
void*          DatosRecepcion,  
int           NumDatosRecepcion,  
MPI_Datatype  Tipodatosrecepcion,  
int           Raíz,  
MPI_Comm      comunicador)
```

Los datos no son recogidos en posiciones contiguas.

NumDatosEnvio y Desplazamientos son ARRAYS, que deben especificar los datos necesarios.

II.4 FUNCIONES MPI COLECTIVAS - SCATTERV

```
MPI_Comm_size(MPI_COMM_WORLD,&nproces);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
err = MPI_Get_processor_name(name, &len);
printf("Proceso %d en procesador %s\n",myrank,name);
for (i=0;i<20;i++){
    send[i]=0;receive[i]=0;
}
for (i=0;i<20;i++){
    send[i] = 10 * (myrank +1) + i;
}
for (i=0;i<3;i++){
    send_cnt[i]=2 + i;
    desp_cnt[i]=4*i + 1;
}
err = MPI_Scatterv(send,send_cnt,desp_cnt,MPI_INTEGER,receive,2,MPI_INTEGER,0,MPI_COMM_WORLD);
MPI_Finalize();
```

Escribe la salida si fueran 3 procesos
(y estuviera bien codificado el parámetro NumDatosRecepcion)

II.4 FUNCIONES MPI COLECTIVAS - SCATTERV

```
for (i=0;i<3;i++){
    send_cnt[i]=2 + i;
    desp_cnt[i]=4*i + 1;
}
```

3 procesos

```
MPI_Scatterv(send,send_cnt,desp_cnt,MPI_INTEGER,receive,var_cnt,MPI_INTEGER,0,MPI_COMM_WORLD);
```

[illegible]

II.4 FUNCIONES MPI COLECTIVAS - SCATTERV

```
for (i=0;i<3;i++){
    send_cnt[i]=2 + i;
    desp_cnt[i]=2*i + 1;
}
```

```
MPI_Scatterv(send,send_cnt,desp_cnt,MPI_INTEGER,receive,var_cnt,MPI_INTEGER,0,MPI_COMM_WORLD);
```

[illegible]

II.4 FUNCIONES MPI COLECTIVAS – REDUCE_SCATTER

La función REDUCE_SCATTER

Se realiza la operación de reducción (**Operación**) sobre un número dado de elementos.

Después se dispersa en bloques

Es decir dos operaciones que ya hemos visto en una única instrucción

II.4 FUNCIONES MPI COLECTIVAS – REDUCE_SCATTER

```
err = MPI_Reduce_scatter(  
void* DatosEnvio,  
Void* DatosRecepcion,  
int* NumDatosRecepcion,  
MPI_Datatype TipoDatos,  
MPI_Op Operación,  
MPI_Comm comunicador)
```

¿Qué procesos envían inicialmente datos?

¿Quién es el raíz?

¿Qué procesos aportan inicialmente datos?

¿Cuántos datos “se reducen”?

¿Cuántos datos “se dispersan”?

¿Cuántos datos recibe cada proceso?

II.4 FUNCIONES MPI COLECTIVAS – REDUCE_SCATTER

```
MPI_Comm_size(MPI_COMM_WORLD,&nproces);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
err = MPI_Get_processor_name(name, &len);
printf("Proceso %d en procesador %s\n",myrank,name);
//La información la tienen que tener todos
for (i=0;i<20;i++){
    send[i]=0;receive[i]=0;
}
for (i=0;i<4;i++){
    recv_cnt[i]=2;
}
for (i=0;i<20;i++){
    send[i] = 10 * (myrank +1) + i;
}

err = MPI_Reduce_scatter(send,receive,recv_cnt,MPI_INTEGER,MPI_SUM,MPI_COMM_WORLD);

for (i=0;i<8;i++){
    printf("Proceso %d. send: %d - receive: %d.\n",myrank,send[i],receive[i]);
}
MPI_Finalize();
```

Escribe la salida si fueran 4 procesos

II.4 FUNCIONES MPI COLECTIVAS – REDUCE_SCATTER

```
for (i=0;i<4;i++){
    recv_cnt[i]=2;
}
```

...

```
err = MPI_Reduce_scatter(send, receive, recv_cnt, MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD);
```

		i																					
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19		
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39		
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49		
	Proceso 3	40	41	i																			
				0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	19	19	20	21	22	23	24	25	26	27	28	29		
	Proceso 1	20	21	22	23	24	25	26	27	29	29	30	31	32	33	34	35	36	37	38	39		
	Proceso 2	30	31	32	33	34	35	36	37	39	39	40	41	42	43	44	45	46	47	48	49		
	Proceso 3	40	41	42	43	44	45	46	47	49	49	50	51	52	53	54	55	56	57	58	59		

	0	1	2	3	4	5	6	7
Temporal	100	104	108	112	116	120	124	128

[illegible]

II.4 FUNCIONES MPI COLECTIVAS - ALLGATHERV

```
err = MPI_Allgather(  
void*          DatosEnvio,  
int            NumDatosEnvio,  
MPI_Datatype   TipoDatosEnvio,  
void*          DatosRecepcion,  
int            NumDatosRecepcion,  
MPI_Datatype   Tipodatosrecepcion,  
MPI_Comm       comunicador)
```

¿Qué procesos envían datos?

Todos los procesos **APORTAN** la misma cantidad de datos.

¿Qué procesos reciben datos?

Todos los procesos **RECIBEN** la misma cantidad de datos.

¿Cuántos datos recibe?

II.4 FUNCIONES MPI COLECTIVAS - ALLGATHER

```
err = MPI_Allgather(send,2,MPI_INTEGER,receive,2,MPI_INTEGER,MPI_COMM_WORLD);
```

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

[illegible]

II.4 FUNCIONES MPI COLECTIVAS - ALLGATHERV

```
err = MPI_Allgatherv(  
void*          DatosEnvio,  
int            NumDatosEnvio,  
MPI_Datatype   TipoDatosEnvio,  
void*          DatosRecepcion,  
int*           NumDatosRecepcion,  
int*           Desplazamientos,  
MPI_Datatype   Tipodatosrecepcion,  
MPI_Comm       comunicador)
```

¿Qué procesos envían datos?

Todos los procesos pueden aportar
DISTINTA cantidad de datos.

¿Qué procesos reciben datos?

Todos los procesos RECIBEN (o mejor
dicho almacenan) la misma cantidad de
datos.

¿Cuántos datos recibe?

II.4 FUNCIONES MPI COLECTIVAS - ALLGATHERV

```
for (i=0;i<4;i++){
    recv_cnt[i]=2;
    desp_cnt[i] = 2*i;
}
```

```
MPI_Allgatherv(send,2,MPI_INTEGER,receive,recv_cnt,desp_cnt,MPI_INTEGER,MPI_COMM_WORLD);
```

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

[illegible]

II.4 FUNCIONES MPI COLECTIVAS - ALLGATHERV

```
for (i=0;i<4;i++){  
    recv_cnt[i]=2;  
    desp_cnt[i] = 4*(i+1);  
}
```

```
MPI_Allgatherv(send,2,MPI_INTEGER,receive,recv_cnt,desp_cnt,MPI_INTEGER,MPI_COMM_  
WORLD);
```

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	Proceso 0	0	0	0	0	10	11	0	0	20	21	0	0	30	31	0	0	40	41	0	0
	Proceso 1	0	0	0	0	10	11	0	0	20	21	0	0	30	31	0	0	40	41	0	0
	Proceso 2	0	0	0	0	10	11	0	0	20	21	0	0	30	31	0	0	40	41	0	0
	Proceso 3	0	0	0	0	10	11	0	0	20	21	0	0	30	31	0	0	40	41	0	0

II.4 FUNCIONES MPI COLECTIVAS - ALLTOALL

La función ALLTOALL es una “dispersión” y “una recolección”, realizada por todos y cada uno de los procesos.

Pero un proceso **envía datos diferentes** a cada uno del resto de procesos en función de su rango.

Por tanto los datos **recibidos** son **diferentes**.

NO hay ningún proceso de reducción

II.4 FUNCIONES MPI COLECTIVAS - ALLTOALL

err = **MPI_Alltoall**

void*

int

MPI_Datatype

void*

int

MPI_Datatype

MPI_Comm

DatosEnvio,
NumDatosEnvio,
TipoDatosEnvio,
DatosRecepcion,
NumDatosRecepcion,
Tipodatosrecepcion,
comunicador)

¿Qué procesos envían datos?

Todos los procesos aportan datos distintos a cada proceso.

¿Qué procesos reciben datos?

¿Cuántos datos recibe?

II.4 FUNCIONES MPI COLECTIVAS - ALLTOALL

```
for (i=0;i<20;i++){
    send[i] = 10 * (myrank +1) + i;
}
err = MPI_Alltoall(send,2,MPI_INTEGER,receive,2,MPI_INTEGER,MPI_COMM_WORLD);
```

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

[illegible]

II.4 FUNCIONES MPI COLECTIVAS - ALLTOALLV

```
err = MPI_Alltoallv  
void*  
int*  
int*  
MPI_Datatype  
void*  
int*  
int*  
MPI_Datatype  
MPI_Comm  
DatosEnvio,  
NumDatosEnvio,  
DesplazamientosEnvio,  
TipoDatosEnvio,  
DatosRecepcion,  
NumDatosRecepcion,  
DesplazamientosRecepcion,  
Tipodatosrecepcion,  
comunicador)
```

¿Qué procesos envían datos?

¿Qué procesos reciben datos?

¿Cuántos datos recibe?

Todos los procesos aportan datos distintos a cada proceso. Y además pueden aportar distinta cantidad

II.4 FUNCIONES MPI COLECTIVAS - ALLTOALLV

```
for (i=0;i<20;i++){
    send[i] = 10 * (myrank +1) + i;
}
for (i=0;i<4;i++){
    recv_cnt[i]=2; send_cnt[i]=2;
    rdesp_cnt[i] = 4*i; // 0, 4, 8, 12
    sdesp_cnt[i]= 3*i; // 0, 3, 6, 9
}
```

Los arrays “desp” NO tienen porque ser iguales en todos los procesos.

`MPI_Alltoallv(send,send_cnt,sdesp_cnt,MPI_INTEGER,receive,recv_cnt,rdesp_cnt,MPI_INTEGER,MPI_COMM_WORLD);`

		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31																		
	Proceso 3	40	41																		
		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
send	Proceso 0	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	Proceso 1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	Proceso 2	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
	Proceso 3	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
		i																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
receive	Proceso 0	10	11	0	0	20	21	0	0	30	31	0	0	40	41	0	0	0	0	0	0
	Proceso 1	13	14	0	0	23	24	0	0	33	34	0	0	43	44	0	0	0	0	0	0
	Proceso 2	16	17	0	0	26	27	0	0	36	37	0	0	46	47	0	0	0	0	0	0
	Proceso 3	19	20	0	0	29	30	0	0	39	40	0	0	49	50	0	0	0	0	0	0

II.5 OTRAS FUNCIONALIDADES MPI

Las funciones de envío y recepción vistas eran funciones bloqueantes, hay varias opciones para implementar estas funciones

MPI_Ssend

Envío síncrono, el envío finaliza cuando empieza la recepción (por tanto sé que el buffer de envío se puede reutilizar y sé que el receptor ha llegado a una determinada instrucción de recepción)

MPI_Bsend

Envío basado en buffer, acaba independientemente del receptor cuando la información está en el buffer (pero no se ha completado por lo tanto no se puede reutilizar el buffer de salida y podría provocar errores por el tamaño del buffer que habría que gestionarlo correctamente)

MPI_Send puede utilizar cualquiera de las dos opciones en función de la instalación y la ejecución

MPI_Rsend

Acaba independiente de lo que pasa en recepción, es la función que más errores puede provocar (y por tanto optimizar). (retorna inmediatamente porque la recepción debe estar preparada (R)eady)

MPI_Recv

Cuando se llama a esta función no se retorna hasta que se ha recibido el mensaje (si no llega nunca no retorna nunca)

II.5 OTRAS FUNCIONALIDADES MPI

Una opción para “intentar” simultanear cómputo y comunicaciones son las funciones asíncronas (se intenta evitar que los procesos estén ociosos mientras se espera un proceso de envío o de recepción)

```
MPI_Isend(  
void* DatosEnv,  
int NumDatos,  
MPI_Datatype TipoDatos,  
int Destino,  
int tag,  
MPI_Comm comunicador  
MPI_Request *request )
```

```
MPI_Irecv (... , MPI_Request *request )
```

II.5 OTRAS FUNCIONALIDADES MPI

Como las funciones **MPI_Isend** y **MPI_Irecv** retornan sin asegurar la finalización del proceso se debe trabajar con (MPI_Request)

```
MPI_Wait (  
MPI_Request *request,  
MPI_Status  *status  
)  
(retorna cuando se ha completado)
```

```
MPI_Test (  
MPI_Request *request,  
int         *flag  
MPI_Status  *status  
)
```

(informa si se ha completado en flag con true o false)

II.5 OTRAS FUNCIONALIDADES MPI

Cuando necesitamos enviar tipos de datos complejos podemos optar por dos opciones:

1. Crear un tipo de datos derivado
2. Empaquetar los datos

Si se crea un tipo derivado en la creación se asegura la contigüidad de los datos en memoria

Si no se ha creado el tipo derivado se empaquetan para “prepararlos” para el envío.

Hay que tener en cuenta que mejora el rendimiento realizando envíos de gran tamaño que muchos envíos de pequeño tamaño

II.5 OTRAS FUNCIONALIDADES MPI

Los comunicadores se utilizan para la comunicación de mensaje entre un grupo de procesos.

Grupos por defecto

MPI_COMM_WORLD
MPI_COMM_SELF
MPI_COMM_NULL

Propiedades que se obtienen de un grupo

MPI_Group_size
MPI_Group_rank

Crear comunicador:

`MPI_Comm_group`

Seleccionar procesos para el grupo:

`MPI_Group_union`

`MPI_Group_intersection`

`MPI_Group_difference`

`MPI_Group_incl`

`MPI_Group_excl`

`MPI_Group_range_incl`

`MPI_Group_range_excl`

Constructor:

`MPI_Comm_create`

Destructor:

`MPI_Group_free`

II.5 OTRAS FUNCIONALIDADES MPI

Ejemplo grupos:

```
int rangos1[4]={0,1,2};
int rangos2[4]={7,5};
MPI_Group G_original, G2, G3;
MPI_Comm COMM2, COMM3;

MPI_Comm_group(MPI_COMM_WORLD, &G_original); //Genero un identificador de grupo para MPI_COMM_WORLD

MPI_Group_incl(G_original,3,rangos1,&G2); //Incluyo procesos en los grupos
MPI_Group_incl(G_original,2,rangos2,&G3);

MPI_Comm_create(MPI_COMM_WORLD,G2,&COMM2); //Genero los comunicadores
MPI_Comm_create(MPI_COMM_WORLD,G3,&COMM3);

....

MPI_Group_free(&G2);      //Libero recursos
MPI_Group_free(&G3);
```

MPI_Wtime

MPI_Wtick

MPI_Sendrecv (buffers diferentes)

MPI_Sendrecv_replace (mismo buffer)

MPI2

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 (comunicaciones One-sided)

- 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: spawning
- 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

MPI2 – Comunicaciones ONE SIDED

Hemos visto comunicaciones más o menos bloqueantes en MPI-1 en las cuales el nivel de intervención de emisor y receptor varía pero existe.

En las comunicaciones ONE SIDED de MPI2 sólo uno de los elementos de la comunicación participa

El proceso de comunicación pasa a ser un proceso **RMA** (Remote Memory Access)

Tres funciones principales:

- **MPI_Put**
- **MPI_Get**
- **MPI_Accumulate**

CUIDADO: Para por ejemplo plataformas SM con mecanismos como la DMA que puedan ser utilizados por MPI ...

MPI2 – Comunicaciones ONE SIDED

No se utilizan las funciones anteriores directamente, han de crearse unas “ventanas”

`MPI_Win_create`

`MPI_Win_free`

Problemas de sincronización

`MPI_Win_fence`

Problemas de bloqueos

`MPI_Win_lock`

`MPI_Win_unlock`

MPI2 – Creación y manejo de procesos

El modelo de creación de procesos difiere del modelo fork-join

El modelo es la ejecución de un binario con una serie de atributos, uno de los parámetros es el nombre del ejecutable

`MPI_Comm_spawn`

`MPI_Comm_spawn_multiple`

El nuevo proceso se integra dentro del grupo `MPI_COMM_WORLD` con el rango que se le deba asignar

Existe un nuevo parámetro `MPI_UNIVERSE_SIZE` (tamaño máximo de procesos)

COMPUTACIÓN PARALELA
3º GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA
INFORMACIÓN

Unidad Didáctica 2.
Arquitecturas de memoria
distribuida: MPI

- II.1 Introducción**
- II.2 Introducción a MPI**
- II.3 Funciones MPI básicas**
- II.4 Funciones MPI de comunicación colectivas**
- II.5 Otras funcionalidades MPI**
- II.6 Ejemplos**