

# SISTEMAS PARALELOS

---

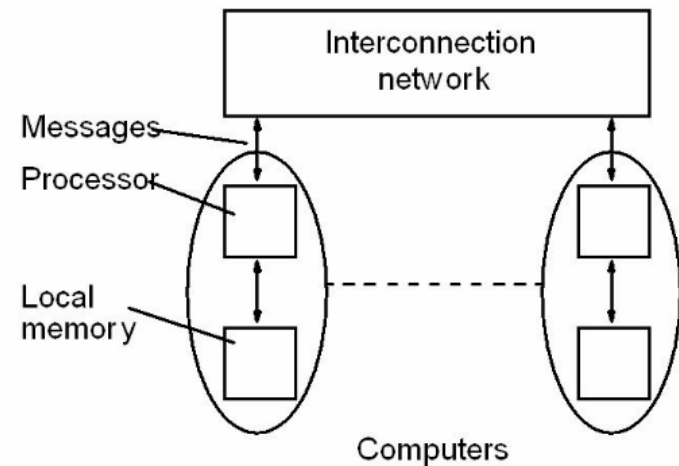
Clase 5 – Programación en memoria distribuida // MPI

# PROGRAMACIÓN EN MEMORIA DISTRIBUIDA // MESSAGE PASSING INTERFACE (MPI)

---

# Plataformas de memoria distribuida

- Consisten de un conjunto de nodos de procesamiento, cada uno con su propio espacio de datos
  - Cada nodo puede contener un procesador único o un multiprocesador (variante actual)
- Los nodos se comunican intercambiando mensajes tanto para sincronización como para comunicación
- ¿Hay problemas de consistencia?
- Modelo de programación asociado: pasaje de mensajes. ¿Memoria compartida?



# Fundamentos del modelo de pasaje de mensajes

- Consiste de  $p$  procesos (eventualmente procesadores), cada uno de ellos con su espacio de direcciones exclusivo.
- Características claves:
  - Espacio de direcciones particionado.
    - Cada dato pertenece a una partición.
    - Toda interacción requiere la cooperación de dos procesos.
  - Solo soporta paralelismo explícito.
- El intercambio de mensajes sirve para varios propósitos:
  - Intercambio explícito de datos (programador).
  - Sincronizar procesos.

# Fundamentos del modelo de pasaje de mensajes

- Programabilidad → En pasaje de mensajes, se trabaja en bajo nivel.
  - El programador es responsable de la distribución de datos y el mapeo de procesos, así como la comunicación entre tareas
  - Difícil de programar, difícil de depurar, difícil de mantener.
  - La mayoría de los programas se escriben siguiendo el modelo *Single Program Multiple Data* (SPMD).
    - No todos ejecutan las mismas instrucciones (sentencias de selección).
    - Los procesos no están sincronizados en la ejecución de cada sentencia.

# Fundamentos del modelo de pasaje de mensajes

- Eficiencia → El ajuste (*tuning*) para mejorar el rendimiento puede ser óptimo.
  - Puede manejar el balance de carga, la redistribución de datos y procesos (dinámicamente), replicar datos, entre otras tareas.
- Portabilidad → Existen numerosas librerías para facilitar la ejecución de código sobre diferentes arquitecturas (estándares)
  - Buena portabilidad de código, no necesariamente portabilidad de rendimiento.

# Fundamentos del modelo de pasaje de mensajes

- Ventajas:

- El programador tiene total control para lograr sistemas más eficientes y escalables.
- Puede implementarse eficientemente en muchas arquitecturas paralelas.
- Más fácil de predecir el rendimiento.



- Desventajas:

- Mayor complejidad al implementar estos algoritmos para lograr alto rendimiento.



# Fundamentos del modelo de pasaje de mensajes: Operaciones Send y Receive

- Los prototipos de las operaciones son:

```
Send (void *sendbuf, int nelems, int dest)
Receive (void *recvbuf, int nelems, int source)
```

- Ejemplo:

|                 |                    |
|-----------------|--------------------|
| P0              | P1                 |
| a = 100;        | receive(&b, 1, 0)  |
| send(&a, 1, 1); | printf("%d\n", b); |
| a = 0;          |                    |

- La semántica del Send requiere que en P1 quede el valor 100 (no 0).
- Existe diferentes protocolos para Send y Receive: bloqueante y no bloqueante.

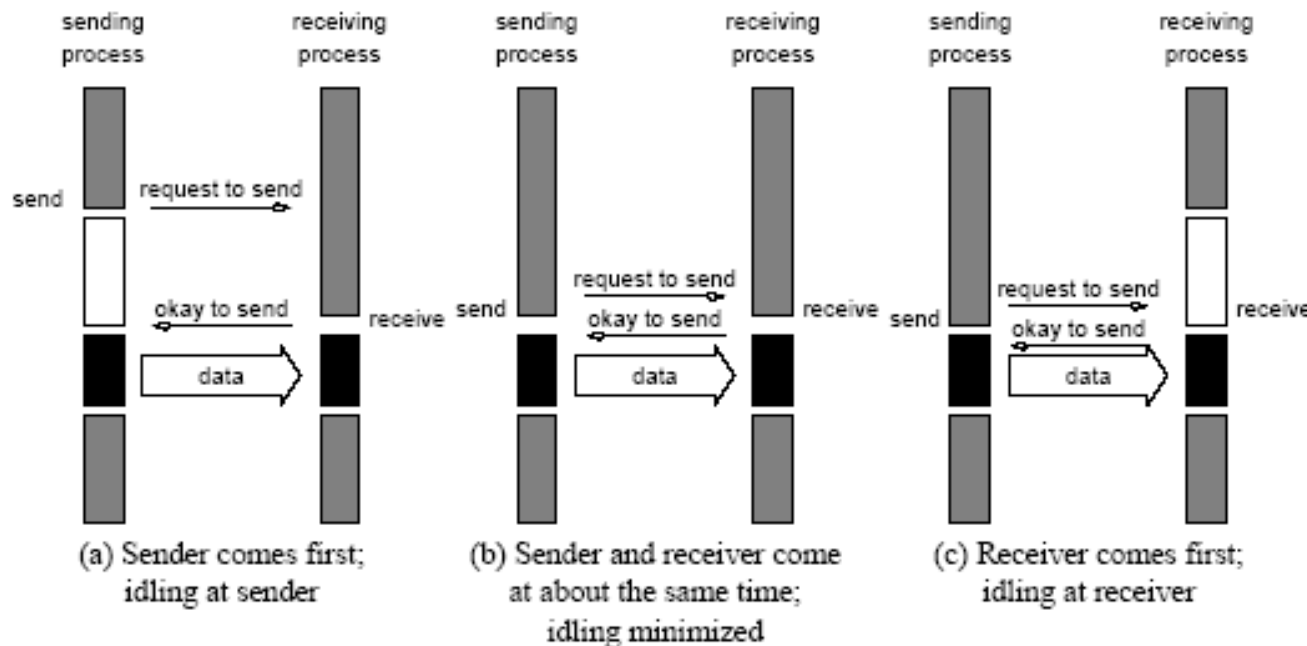


# Operaciones Send y Receive - Bloqueantes

- Send bloqueante: para garantizar la semántica de la operación Send, no devuelve el control hasta que el dato a transmitir esté seguro.
- Ociosidad en los procesos.
- Hay dos alternativas:
  - Sin *buffering*.
  - Con *buffering*.

# Operaciones Send y Receive – Bloqueantes sin *buffering*

- El send se bloquea hasta que el receptor no termine el receive del mensaje.
- Tiempo ocioso de los procesadores.
- Deadlocks si las sentencias de comunicación no coinciden.



**P0**

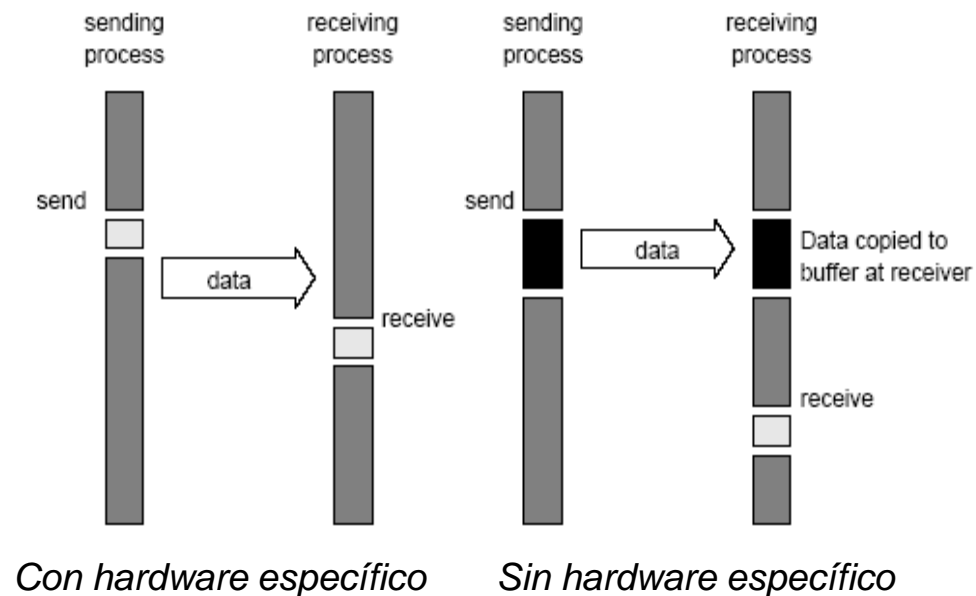
```
send(&a, 1, 1);
receive(&b, 1, 1);
```

**P1**

```
send(&a, 1, 0);
receive(&b, 1, 0);
```

# Operaciones Send y Receive – Bloqueantes con *buffering*

- El send se bloquea hasta que el mensaje llega a un buffer prealocado.
- Transmisión del mensaje:
  - Hardware para comunicación asincrónica (sin intervención de la CPU) → se comienza la transmisión al buffer del receptor.
  - Sin hardware especial → el emisor transmite el mensaje al buffer del receptor y recién ahí se desbloquea.



# Operaciones Send y Receive – Bloqueantes con *buffering*

- Protocolos con *buffering* reducen el tiempo ocioso de los procesadores pero aumentan el costo por manejo de buffers.
- Tamaño limitado de los buffers → bloquea al send hasta que haya lugar.

```
P0
for (i=0; i<1000; i++){
    produce_data(&a);
    send(&a, 1, 1);
}
```

```
P1
for (i=0; i<1000; i++){
    receive(&a, 1, 0);
    consume_data(&a);
}
```

- *Buffering* reduce la ocurrencia de deadlocks pero no los evita completamente

```
P0
receive(&a, 1, 1)
send(&b, 1, 1)
```

```
P1
receive(&a, 1, 0)
send(&b, 1, 0);
```

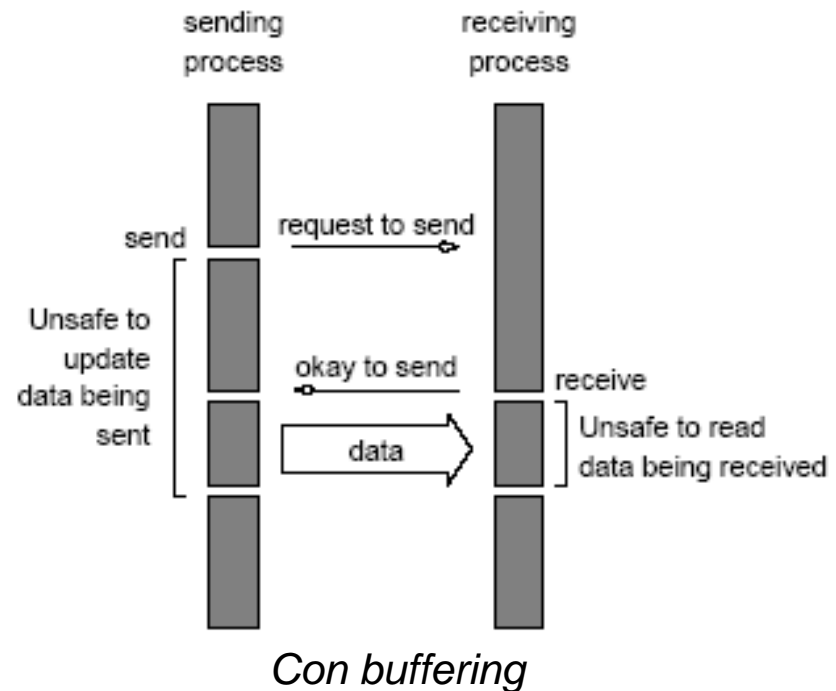
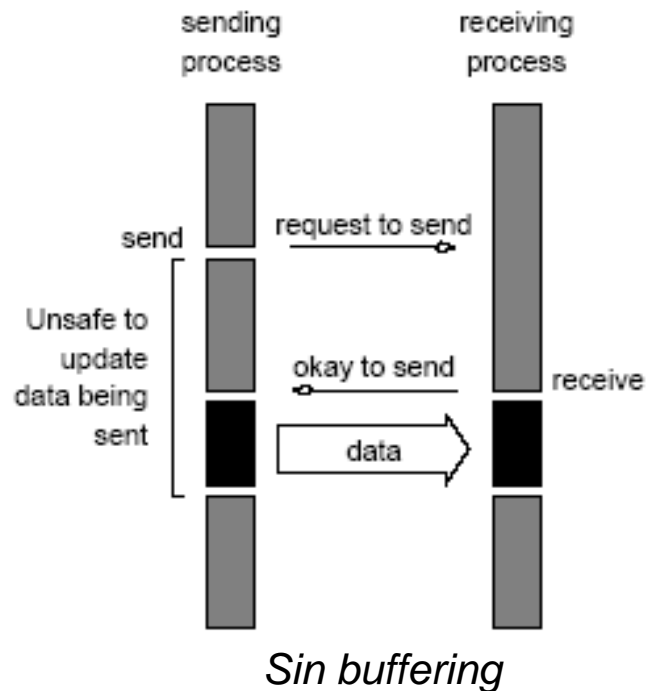
```
P0
send(&a, 1, 1);
receive(&b, 1, 1);
```

```
P1
send(&a, 1, 0);
receive(&b, 1, 0);
```

# Operaciones Send y Receive – No bloqueantes

- Para evitar overhead (ociosidad o manejo de buffer) se devuelve el control de la operación inmediatamente.
- Requiere un posterior chequeo para asegurarse la finalización de la comunicación → Deja en manos del programador asegurar la semántica del Send.
- Hay dos alternativas:
  - Sin *buffering*: inicia comunicación al llegar al receive.
  - Con *buffering*: el emisor utiliza acceso directo a memoria (DMA) para copiar los datos a un buffer prealocado mientras el proceso continúa su computo (reduce el tiempo en que el dato no está seguro).

# Operaciones Send y Receive – No bloqueantes



# Resumen de opciones para operaciones Send y Receive

|               | Operaciones bloqueantes   | Operaciones no bloqueantes  |
|---------------|---|---|
| Con buffering | El emisor retoma el control una vez que los datos han sido copiados al buffer   | El emisor retoma el control una vez que ha iniciado la transferencia DMA al buffer, aun cuando podría no haberse completado |
| Sin buffering | El emisor se bloquea hasta que el receptor alcance el receive correspondiente   | El emisor envía los datos cuando el receptor alcanza el receive   |
|               | La semántica del send y receive está asegurada por la operación correspondiente | El programador debe asegurar explícitamente el cumplimiento de la semántica de las operaciones                              |

# ESTÁNDAR MESSAGE PASSING INTERFACE (MPI)

---



# Estándar MPI

- A principios de la década de 1990, existían numerosas librerías para pasaje de mensaje (no compatibles). Un grupo de representantes de las universidades y de la industria se reunieron para desarrollar un estándar para programación basada en pasaje de mensajes.
- MPI define una librería estándar que puede ser empleada desde C o Fortran (y potencialmente desde otros lenguajes).
- Existen diferentes implementaciones de MPI en la actualidad (OpenMPI, MPICH, Intel MPI, IBM MPI, entre otras), aunque no todas soportan la especificación en forma completa → Aspecto a considerar a la hora de elegir la implementación MPI.
- Modelo Single Program Multiple Data (SPMD).
- El estándar MPI define la sintaxis y la semántica de más de 400 rutinas, aunque básicamente con 6 podemos escribir programas paralelos basados en pasaje de mensajes: *MPI\_Init*, *MPI\_Finalize*, *MPI\_Comm\_size*, *MPI\_Comm\_rank*, *MPI\_Send* y *MPI\_Recv*

# MPI – Inicio y finalización de entorno

- *MPI\_Init*: inicializa el entorno MPI. Debe ser invocada por todos los procesos antes que cualquier otro llamado a rutinas MPI.

```
MPI_Init (int *argc, char **argv)
```

Algunas implementaciones de MPI requieren argc y argv para inicializar el entorno

- *MPI\_Finalize*: cierra el entorno MPI. Debe ser invocado por todos los procesos como último llamado a rutinas MPI.

```
MPI_Finalize ()
```

# MPI - Comunicadores

- Un comunicador define el dominio de comunicación.
- Son variables del tipo `MPI_Comm` → almacena información sobre que procesos pertenecen a él.
- Un proceso puede pertenecer a muchos comunicadores.
- Existe un comunicador que incluye a todos los procesos de la aplicación `MPI_COMM_WORLD`.
- En cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar.

# MPI – Adquisición de información

- *MPI\_Comm\_size*: indica la cantidad de procesos en el comunicador.

`MPI_Comm_size (MPI_Comm comunicador, int *cantidad).`

- *MPI\_Comm\_rank*: indica el “*rank*” (identificador) del proceso dentro de ese comunicador.

`MPI_Comm_rank (MPI_Comm comunicador, int *rank)`

- rank es un valor entre [0..cantidad]
- Cada proceso puede tener un rank diferente en cada comunicador.

# MPI – ¡Hola Mundo!

```
1  #include <mpi.h>
2
3  int main(int argc, char *argv[])
4  {
5      int cantidad, identificador;
6
7      /* iniciar entorno MPI */
8      MPI_Init(&argc, &argv);
9      /* obtener cantidad de procesos*/
10     MPI_Comm_size (MPI_COMM_WORLD, &cantidad);
11     /* obtener rank */
12     MPI_Comm_rank(MPI_COMM_WORLD, &identificador);
13     /* imprimir */
14     printf("¡Hola Mundo! Soy %d de %d \n",
15           identificador, cantidad);
16     /* finalizar entorno MPI */
17     MPI_Finalize();
18 }
19
20
```

# MPI – ¡Hola Mundo!

```
1  #include <mpi.h>
2
3  int main(int argc, char *argv[])
4  {
5      int cantidad, identificador;
6
7      /* iniciar entorno MPI */
8  enzo@hoja3: ~/sp
9  enzo@hoja3:~/sp$ mpicc mpi_hello_world.c -o mpi_hello_world -O3 -Wall
10 mpi_hello_world.c: In function 'main':
11 mpi_hello_world.c:18: warning: control reaches end of non-void function
12 enzo@hoja3:~/sp$
13 enzo@hoja3:~/sp$ mpirun -np 8 mpi_hello_world
14 Hola Mundo! Soy 5 de 8
15 Hola Mundo! Soy 6 de 8
16 Hola Mundo! Soy 7 de 8
17 Hola Mundo! Soy 2 de 8
18 Hola Mundo! Soy 1 de 8
19 Hola Mundo! Soy 3 de 8
20 Hola Mundo! Soy 4 de 8
    Hola Mundo! Soy 0 de 8
    enzo@hoja3:~/sp$
```

# MPI – Comunicaciones punto a punto bloqueantes

- *MPI\_Send*: rutina básica para enviar datos a otro proceso.

```
MPI_Send (void *buf, int cantidad, MPI_Datatype tipoDato,  
          int destino, int tag, MPI_Comm comunicador)
```

- Valor de Tag entre [0..MPI\_TAG\_UB].
- Existen diferentes variantes para *MPI\_Send*:  
MPI\_Send    MPI\_Bsend    MPI\_Ssend    MPI\_Rsend

# MPI – Tipos de datos para comunicaciones

| <b>MPI</b>         | <b>C</b>           |
|--------------------|--------------------|
| MPI_CHAR           | signed char        |
| MPI_SHORT          | signed short int   |
| MPI_INT            | signed int         |
| MPI_LONG           | signed long int    |
| MPI_UNSIGNED_CHAR  | unsigned char      |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED       | unsigned int       |
| MPI_UNSIGNED_LONG  | unsigned long int  |
| MPI_FLOAT          | float              |
| MPI_DOUBLE         | double             |
| MPI_LONG_DOUBLE    | long double        |
| MPI_BYTE           |                    |
| MPI_PACKED         |                    |



# MPI – Comunicaciones punto a punto bloqueantes

- *MPI\_Recv*: rutina básica para recibir datos a otro proceso.

```
MPI_Recv (void *buf, int cantidad, MPI_Datatype tipoDato, int  
origen, int tag, MPI_Comm comunicador, MPI_Status *estado)
```

- Comodines MPI\_ANY\_SOURCE y MPI\_ANY\_TAG.

- Estructura MPI\_Status

```
typedef struct MPI_Status { int MPI_SOURCE;  
                           int MPI_TAG;  
                           int MPI_ERROR; }
```

- *MPI\_Get\_count*: rutina para obtener la cantidad de elementos recibidos

```
MPI_Get_count(MPI_Status *estado,  
              MPI_Datatype tipoDato, int *cantidad)
```

# MPI – Comunicaciones punto a punto – Casos de deadlock

## Ejemplo 1

```
int a[10], b[10], identificador;
MPI_Status estado;
MPI_Comm_rank(MPI_COMM_WORLD, &identificador);
if (identificador == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, estado);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, estado);
}
```

## Ejemplo 2

```
int a[10], b[10], cantProc, id;
MPI_Status estado;
MPI_Comm_size(MPI_COMM_WORLD, &cantProc);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Send(a, 10, MPI_INT, (id+1)%cantProc, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (id-1+cantProc)%cantProc, 1, MPI_COMM_WORLD, estado);
```

# MPI – Comunicaciones punto a punto – Casos de deadlock

Código para evitar el deadlock del segundo ejemplo

```
int a[10], b[10], cantProc, id;
MPI_Status estado;

MPI_Comm_size(MPI_COMM_WORLD, &cantProc);
MPI_Comm_rank(MPI_COMM_WORLD, &id);

if (id%2 == 1) {
    MPI_Send(a, 10, MPI_INT, (id+1)%cantProc, 1, MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (id-1+cantProc)%cantProc, 1, MPI_COMM_WORLD, estado);
}
else {
    MPI_Recv(b, 10, MPI_INT, (id-1+cantProc)%cantProc, 1, MPI_COMM_WORLD, estado);
    MPI_Send(a, 10, MPI_INT, (id+1)%cantProc, 1, MPI_COMM_WORLD);
}
```

# MPI – Comunicaciones punto a punto no bloqueantes

- Comienzan la operación de comunicación e inmediatamente devuelven el control (no garantiza que la operación haya finalizado).

```
MPI_Isend (void *buf, int cantidad, MPI_Datatype tipoDato,  
           int destino, int tag, MPI_Comm comunicador,  
           MPI_Request *solicitud)
```

```
MPI_Irecv (void *buf, int cantidad, MPI_Datatype tipoDato,  
           int origen, int tag, MPI_Comm comunicador,  
           MPI_Request *solicitud)
```

# MPI – Comunicaciones punto a punto no bloqueantes

- *MPI\_Test*: evalúa si la operación de comunicación finalizó.

`MPI_Test (MPI_Request *solicitud, int *flag, MPI_Status *estado)`

- *MPI\_Wait*: bloquea al proceso hasta que la operación indicada en el Request haya finalizado.

`MPI_Wait (MPI_Request *solicitud, MPI_Status *estado)`

- Este tipo de comunicación permite solapar cómputo con comunicación.
- Evita overhead asociado al manejo de buffers.
- Es responsabilidad del programador asegurar que se la comunicación se realice correctamente.

# MPI – Comunicaciones punto a punto no bloqueantes – Ejemplo MPI\_Isend

```
int main (int argc, char *argv[]){
    int cant, id, *dato, i;
    MPI_Status estado;

    dato = (int *) malloc (100 * sizeof(int));
    MPI_Init(&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    if (id == 0)  {
        cant = atoi(argv[1])%100;
        MPI_Send(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD);
        for (i=0; i< 100; i++) dato[i]=0;
    } else {
        MPI_Recv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD, &estado);
        MPI_Get_count(&estado, MPI_INT, &cant);
        // Procesa los datos recibidos
    }
    MPI_Finalize();
}
```

Para usar comunicaciones no bloqueantes, ¿alcanza con cambiar *MPI\_Send* por *MPI\_Isend*?

# MPI – Comunicaciones punto a punto no bloqueantes – Ejemplo MPI\_Isend

```
int main (int argc, char *argv[]){
    int cant, id, *dato, i;
    MPI_Status estado; MPI_Request req;

    dato = (int *) malloc (100 * sizeof(int));
    MPI_Init(&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    if (id == 0)  {
        cant = atoi(argv[1])%100;
        MPI_Isend(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD, &req);
        // Otro cómputo
        MPI_Wait(&req,&estado);
        for (i=0; i< 100; i++) dato[i]=0;
    } else {
        MPI_Recv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD, &estado);
        MPI_Get_count(&estado, MPI_INT, &cant);
        // Procesa los datos recibidos
    }
    MPI_Finalize();
}
```

# MPI – Comunicaciones punto a punto no bloqueantes – Ejemplo MPI\_Irecv

```
int main (int argc, char *argv[]){
    int id, *dato, i, flag;
    MPI_Status estado;
    MPI_Request req;

    dato = (int *) malloc (100 * sizeof(int));
    MPI_Init(&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    if (id == 0)  {
        // Inicializa datos a enviar
        MPI_Send(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD);
    } else {
        MPI_Irecv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD ,&req);
        MPI_Test(&req, &flag,&estado);
        while (!flag){
            //Trabaja mientras espera
            MPI_Test(&req, &flag,&estado);
        }
        // Procesa los datos recibidos
    }
    MPI_Finalize();
}
```



# MPI – Comunicaciones punto a punto –

## Orden y *fairness*

- Sobre orden:
  - MPI asegura que los mensajes no se *sobrepasarán* entre ellos.
  - Si un proceso envía 2 mensajes seguidos a un mismo receptor (M1 y M2), y ambos coinciden con el mismo receive, el orden de recepción será: M1, M2.
  - Si un proceso ejecuta 2 receive seguidos (R1 y R2), y hay un mensaje pendiente que coincide con ambos, R1 recibirá antes que R2.
- Sobre *fairness* (justicia):
  - MPI no asegura fairness → es responsabilidad del programador que un proceso no sufra *inanición*
  - Ejemplo: P0 le envía un mensaje a P2. Sin embargo, P1 envía otro mensaje a P2 que compite con el de P0 (coincide con el receive). P2 sólo recibirá uno de los 2 mensajes.

# MPI – Consulta de comunicaciones pendientes

- Es posible consultar si hay comunicaciones pendiente y algunos de sus datos
- *MPI\_Probe*: bloquea al proceso hasta que llegue un mensaje que cumpla con el origen y el tag.

```
MPI_Probe (int origen, int tag, MPI_Comm comunicador,  
           MPI_Status *estado)
```

- *MPI\_Iprobe*: chequea por el arribo de un mensaje que cumpla con el origen y tag.

```
MPI_Iprobe (int origen, int tag, MPI_Comm comunicador,  
            int *flag, MPI_Status *estado)
```

- Comodines en Origen y Tag.

¿Cuándo y para qué usar cada uno?

# MPI – Comunicaciones colectivas

- MPI provee un conjunto de funciones para realizar operaciones colectivas, sobre un grupo de procesos asociado con un comunicador. Todos los procesos del comunicador deben llamar a la rutina colectiva.
- Tipos de operaciones colectivas:
  - Sincronización: los procesos se bloquean hasta que todos hayan llegado a determinado punto del programa
  - Transferencia de datos: *broadcast*, *scatter*, *gather* y sus variantes
  - Computaciones colectivas: operaciones de reducción

# MPI – Comunicaciones colectivas – Sincronización por barrera

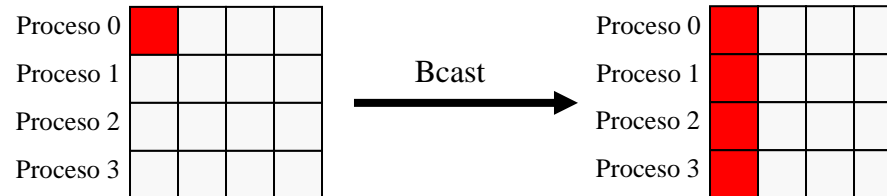
- Sincronización en una barrera.

```
MPI_Barrier(MPI_Comm comunicador)
```

# MPI – Comunicaciones colectivas - Broadcast

- *Broadcast*: un proceso envía el mismo mensaje a todos los otros procesos del comunicador (incluido a sí mismo).

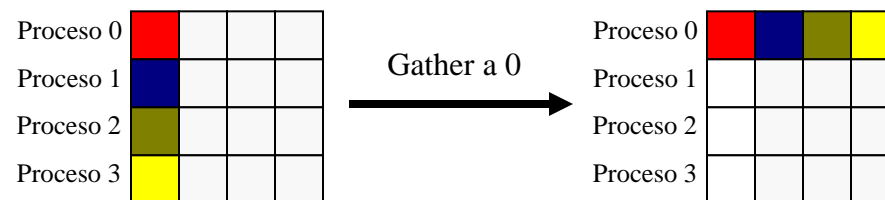
```
MPI_Bcast (void *buf, int cantidad, MPI_Datatype tipoDato,  
           int origen, MPI_Comm comunicador)
```



# MPI – Comunicaciones colectivas - Gather

- *Gather*: recolecta un vector de datos de cada proceso del comunicador (inclusive el destino) y los concatena en orden para dejar el resultado en un único proceso.

```
MPI_Gather (void *sendbuf, int cantEnvio,  
           MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec,  
           MPI_Datatype tipoDatoRec, int destino,  
           MPI_Comm comunicador)
```



# MPI – Comunicaciones colectivas - Gatherv

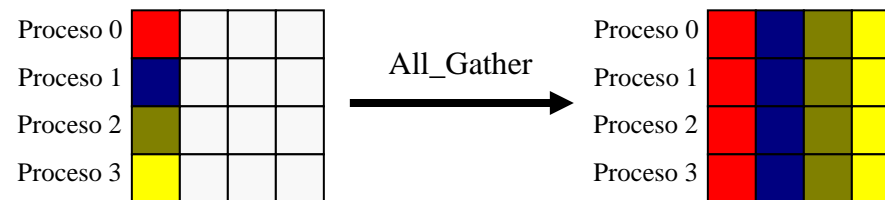
- Gather requiere que todos los procesos aporten la misma cantidad de datos. Para casos en los que cada proceso envía una cantidad diferente, MPI cuenta con la variante Gatherv

```
MPI_Gatherv (void *sendbuf, int cantEnvio,  
             MPI_Datatype tipoDatoEnvio, void*recvbuf, int *cantsRec,  
             int *desplazamientos, MPI_Datatype tipoDatoRec,  
             int destino, MPI_Comm comunicador)
```

# MPI – Comunicaciones colectivas - Allgather

- *Allgather*: funciona como el Gather sólo que el resultado es enviado a todos los procesos.

```
MPI_AllGather (void *sendbuf, int cantEnvio,  
              MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec,  
              MPI_Datatype tipoDatoRec, MPI_Comm comunicador)
```





# MPI – Comunicaciones colectivas - Allgatherv

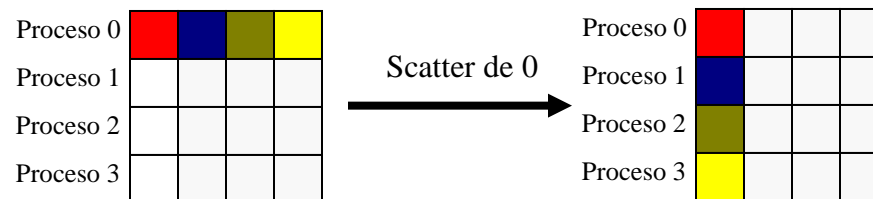
- MPI ofrece una variante de *Allgather* para aquellos casos en cada proceso puede enviar una cantidad diferente de datos.

```
MPI_Allgatherv (void *sendbuf, int cantEnvio,  
                MPI_Datatype tipoDatoEnvio, void*recvbuf,  
                int *cantsRec, int *desplazamientos,  
                MPI_Datatype tipoDatoRec, MPI_Comm comunicador)
```

# MPI – Comunicaciones colectivas - Scatter

- Scatter: reparte un vector de datos entre todos los procesos (inclusive el mismo dueño del vector) de forma equitativa.

```
MPI_Scatter (void *sendbuf, int cantEnvio,  
             MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec,  
             MPI_Datatype tipoDatoRec, int origen, MPI_Comm comunicador)
```



# MPI – Comunicaciones colectivas - Scatterv

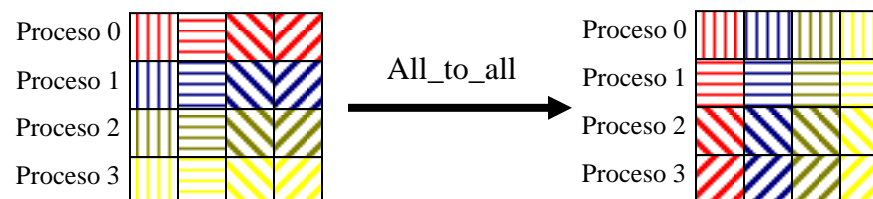
- Al igual que con Gather, MPI ofrece una variante de Scatter para aquellos casos en que hay que repartir una cantidad diferente a cada proceso.

```
MPI_Scatterv (void *sendbuf, int *cantsEnvio,  
             int *desplazamientos, MPI_Datatype tipoDatoEnvio,  
             void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec,  
             int origen, MPI_Comm comunicador)
```

# MPI – Comunicaciones colectivas – All to all

- All to all: cada proceso envía una parte de sus datos a cada uno de los otros procesos (incluso a él mismo) y recibe de ellos una parte.
  - Todas las porciones son del mismo tamaño
  - Es equivalente a realizar un Scatter + Gather.

```
MPI_Alltoall (void *sendbuf, int cantEnvio,  
              MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec,  
              MPI_Datatype tipoDatoRec, MPI_Comm comunicador)
```



# MPI – Comunicaciones colectivas – All to all

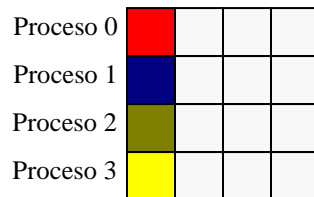
- MPI cuenta con la variante Alltoallv para los casos en que los procesos pueden enviar y recibir porciones de tamaño diferente.

```
MPI_Alltoallv (void *sendbuf, int *cantsEnvio, int *despEnvio,  
               MPI_Datatype tipoDatoEnvio, void*recvbuf, int *cantsRec,  
               int *despRec, MPI_Datatype tipoDatoRec,  
               MPI_Comm comunicador)
```

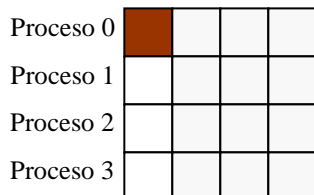
# MPI – Comunicaciones colectivas – Reducciones

- Reducción de todos a uno: combina los elementos enviados por cada uno de los procesos (inclusive el destino) aplicando una cierta operación.

`MPI_Reduce (void *sendbuf, void *recvbuf, int cantidad,  
MPI_Datatype tipoDato, MPI_Op operación,  
int destino , MPI_Comm comunicador)`



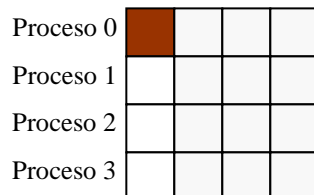
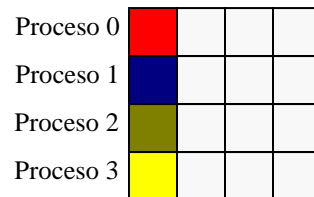
Reduce a 0



# MPI – Comunicaciones colectivas – Reducciones

- Reducción de todos a uno: combina los elementos enviados por cada uno de los procesos (inclusive el destino) aplicando una cierta operación.

MPI\_Reduce (voice

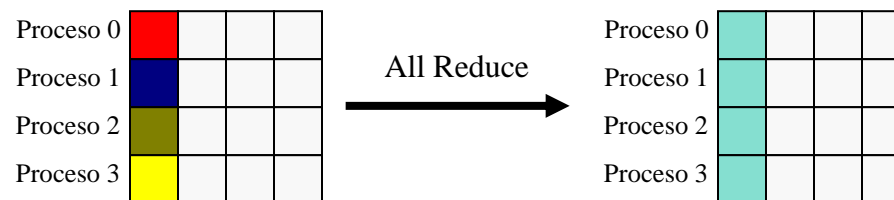


| Operation  | Meaning                | Datatypes                     |
|------------|------------------------|-------------------------------|
| MPI_MAX    | Maximum                | C integers and floating point |
| MPI_MIN    | Minimum                | C integers and floating point |
| MPI_SUM    | Sum                    | C integers and floating point |
| MPI_PROD   | Product                | C integers and floating point |
| MPI_LAND   | Logical AND            | C integers                    |
| MPI_BAND   | Bit-wise AND           | C integers and byte           |
| MPI_LOR    | Logical OR             | C integers                    |
| MPI_BOR    | Bit-wise OR            | C integers and byte           |
| MPI_LXOR   | Logical XOR            | C integers                    |
| MPI_BXOR   | Bit-wise XOR           | C integers and byte           |
| MPI_MAXLOC | max-min value-location | Data-pairs                    |
| MPI_MINLOC | min-min value-location | Data-pairs                    |

# MPI – Comunicaciones colectivas – Reducciones

- Reducción de todos a todos: el resultado de la operación de reducción es enviado a todos los procesos.

`MPI_Allreduce (void *sendbuf, void *recvbuf, int cantidad, MPI_Datatype tipoDato, MPI_Op operación, MPI_Comm comunicador)`





# MPI – Ejemplo: reducción a suma de vector con comunicaciones punto a punto

---

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  #define MAX_SIZE 2000
6  #define COORDINATOR 0
7
8  int main(int argc, char* argv[]){
9      int i, numProcs, rank, size, stripSize, localSum=0, sum=0;
10     int array[MAX_SIZE];
11     MPI_Status status;
12
13     /* Lee parámetros de la línea de comando */
14     size = atoi(argv[1]);
15     size = (size < MAX_SIZE ? size : MAX_SIZE);
16
17     MPI_Init(&argc, &argv);
18
19     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
20     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
21
22     if (rank == COORDINATOR)
23         for (i=0; i<size ; i++)
24             array[i] = i+1;
```

# MPI – Ejemplo: reducción a suma de vector con comunicaciones punto a punto

```
26     stripSize = size / numProcs;
27
28     /* distribuir datos */
29     if (rank==COORDINATOR){
30         for (i=1; i<numProcs; i++)
31             MPI_Send(array+i*stripSize, stripSize, MPI_INT, i, 0, MPI_COMM_WORLD);
32     } else
33         MPI_Recv(array, stripSize, MPI_INT, COORDINATOR, 0, MPI_COMM_WORLD, &status);
34
35     /* computar suma parcial */
36     for (i=0; i<stripSize; i++)
37         localSum += array[i];
38
39     /* recolectar resultados parciales */
40     if (rank==COORDINATOR){
41         sum = localSum;
42         for (i=1; i<numProcs; i++) {
43             MPI_Recv(&localSum, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
44             sum += localSum;
45         }
46     } else
47         MPI_Send(&localSum, 1, MPI_INT, COORDINATOR, 1, MPI_COMM_WORLD);
48
49     MPI_Finalize();
50
51     if (rank==COORDINATOR){
52         printf("Sum=%d\n", sum);
53     }
```

# MPI – Ejemplo: reducción a suma de vector con comunicaciones colectivas

---

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  #define MAX_SIZE 2000
6  #define COORDINATOR 0
7
8  int main(int argc, char* argv[]){
9      int i, numProcs, rank, size, stripSize, localSum=0, sum=0;
10     int array[MAX_SIZE];
11     MPI_Status status;
12
13     /* Lee parámetros de la línea de comando */
14     size = atoi(argv[1]);
15     size = (size < MAX_SIZE ? size : MAX_SIZE);
16
17     MPI_Init(&argc,&argv);
18
19     MPI_Comm_size(MPI_COMM_WORLD,&numProcs);
20     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
21
22     if (rank == COORDINATOR)
23         for (i=0; i<size ; i++)
24             array[i] = i+1;
```

# MPI – Ejemplo: reducción a suma de vector con comunicaciones colectivas

```
26     stripSize = size / numProcs;
27
28     /* distribuir datos */
29     if (rank==COORDINATOR){
30         for (i=1; i<numProcs; i++)
31             MPI_Send(array+i*stripSize, stripSize, MPI_INT, i, 0, MPI_COMM_WORLD);
32     } else
33         MPI_Recv(array, stripSize, MPI_INT, COORDINATOR, 0, MPI_COMM_WORLD, &status);
34
35     /* computar suma parcial */
36     for (i=0; i<stripSize; i++)
37         localSum += array[i];
38
39     /* recolectar resultados parciales */
40     if (rank==COORDINATOR){
41         sum = localSum;
42         for (i=1; i<numProcs; i++) {
43             MPI_Recv(&localSum, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
44             sum += localSum;
45         }
46     } else
47         MPI_Send(&localSum, 1, MPI_INT, COORDINATOR, 1, MPI_COMM_WORLD);
48
49     MPI_Finalize();
50
51     if (rank==COORDINATOR){
52         printf("Sum=%d\n", sum);
53     }
```

Scatter

Reduce

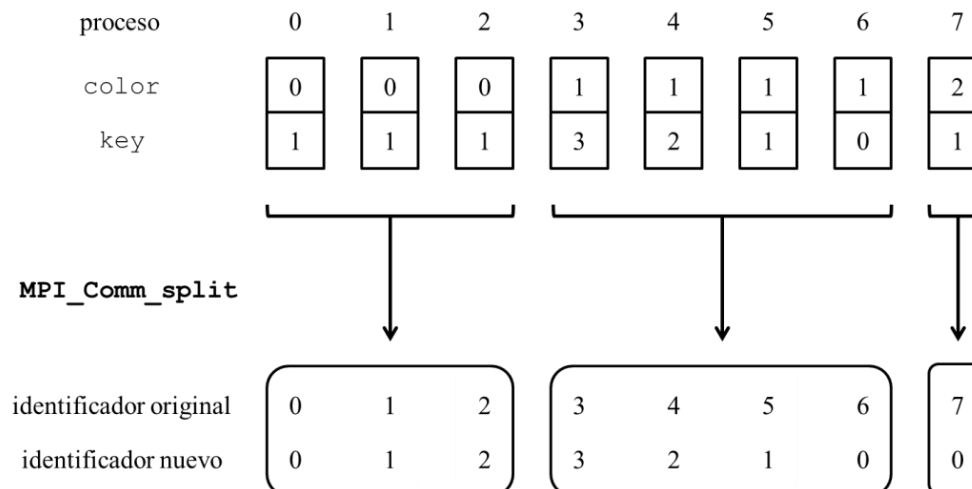
# MPI – Ejemplo: reducción a suma de vector con comunicaciones colectivas

```
25     stripSize = size / numProcs;
26
27     MPI_Scatter(array, stripSize, MPI_INT, array, \
28               stripSize, MPI_INT, COORDINATOR, MPI_COMM_WORLD);
29
30     for (i=0; i<stripSize; i++)
31         localSum += array[i];
32
33     MPI_Reduce(&localSum, &sum, 1, MPI_INT, MPI_SUM, COORDINATOR, MPI_COMM_WORLD);
34
35     MPI_Finalize();
--
```

# MPI – Grupos y comunicadores

- En ocasiones, las operaciones de comunicación entre los procesos de un programa se realizan entre subconjunto de ellos → MPI provee mecanismos para dividir el grupo de procesos asociado a un comunicador en varios subgrupos (cada uno con su correspondiente comunicador)

```
MPI_Comm_split(MPI_Comm comm, int color, int key,  
               MPI_Comm *newcomm)
```



# MPI-2 y MPI-3

- MPI-1 fue introducida en 1994, aunque el foro de MPI continuó trabajando en correcciones y extensiones de esa primera versión.
- MPI-2 fue publicado en 1998 como una versión superadora de la primera. Incluyó funcionalidad para:
  - Generación dinámica de procesos
  - Comunicaciones one-sided
  - Soporte para comunicaciones colectivas de a grupos
  - Soporte para C++
  - E/S paralela
- MPI-3 fue aprobado en 2012. Incluyó funcionalidad para:
  - Comunicaciones colectivas no bloqueantes
  - Más soporte para comunicaciones one-sided
  - Mas soporte para comunicaciones colectivas de a grupos
  - Soporte para Fortran 2008

# Bibliografía usada para esta clase

- MPI tutorial. Blaise Barney, Lawrence Livermore National Laboratory. <https://computing.llnl.gov/tutorials/mpi/>
- Capítulo 6, An Introduction to Parallel Computing. Design and Analysis of Algorithms (2da Edition). Grama A., Gupta A., Karypis G. & Kumar V. (2003) Inglaterra: Pearson Addison Wesley.
- Capítulo 5, Parallel Programming for Multicore and Cluster Systems. Rauber, T. & Rünger, G. (2010). EEUU: Springer-Verlag Berlin Heidelberg.
- Capítulo 10 y 11, Introduction to HPC for Scientists and Engineers. Hager, G. & Wellein, G. (2011) EEUU: CRC Press.