

# MPI

Metodología de la Programación Paralela

Jesús Sánchez Cuadrado (jesusc@um.es)

Curso 2019/20

# ¿Qué es MPI?

- Previamente PVM: Parallel Virtual Machine (aprox. 1993).
- MPI: Message Passing Interface.
- Una especificación para paso de mensajes.
- La primera librería de paso de mensajes estándar y portable.
- Por consenso del MPI Forum. Participantes de unas 40 organizaciones.
- Acabado y publicado en mayo 1994. Actualizado en junio 1995.
- Versiones y variantes: MPI2, HMPI, FT-MPI...

# ¿Qué es MPI?

- MPI es una especificación, no una implementación
  - La especificación indica los nombres, la signatura de las funciones y su semántica
  - Cada fabricante de computadores paralelos ofrece una implementación adaptada a su hardware
  - Un programa MPI correcto se ejecutará en máquina diferentes sin necesidad de cambios
- MPI es un librería, no un lenguaje
  - Los programas en C, C++ o Fortran se compilan utilizando el compilador estándar, y luego se enlazan con la librería MPI

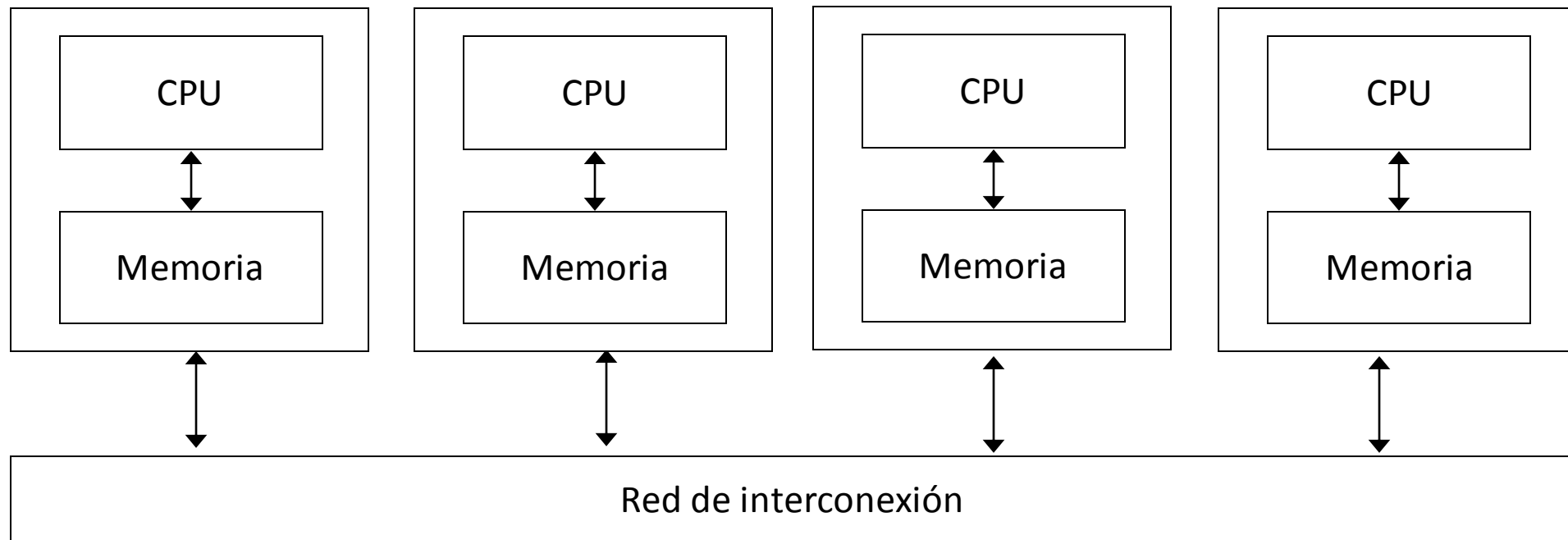
¿Cuál es la diferencia con OpenMP?

# ¿Qué ofrece MPI?

- Estandarización.
  - <http://www.mpi-forum.org>
- Portabilidad: multiprocesadores de memoria compartida, multicomputadores de paso de mensajes, clusters, sistemas heterogéneos, ...
- Buenas prestaciones si está disponible para el sistema con una implementación eficiente.
- Amplia funcionalidad. Del orden de 140 funciones para las operaciones más comunes de paso de mensajes.
- Implementaciones de empresas y libres (mpich, lam/mpi, OpenMPI...)

# Conceptos básicos

- Sistemas de memoria distribuida



# Conceptos básicos

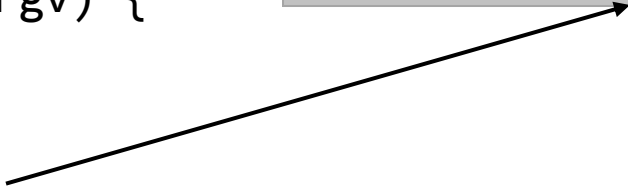
- Procesos
- Modelo SPMD (Single-Program Multiple-Data)
  - El mismo programa se ejecuta igual en todos los procesos
  - Se usa el id del proceso para tener comportamientos diferentes según el proceso
- Mensajes
  - Envío
    - Síncrono
    - Asíncrono
  - Recepción
    - Síncrono
    - Asíncrono
- Comunicadores
  - Universo de procesos

# Estructura básica

- `MPI_Init(int *argc, char **argv);`
  - Inicializa las estructuras de MPI.
  - Configura los procesos
  - Antes de este punto no puede haber llamadas a funciones MPI
  - `MPI_Init(NULL, NULL)` para inicialización por defecto
- `MPI_Finalize();`
  - Libera los recursos
  - A partir de este punto no puede haber llamadas a funciones MPI

# Ejemplo

Comunicador por defecto.  
Incluye todos los procesos.



Número de procesos en el  
comunicador

Identificador del proceso  
actual en el comunicador

Nombre del proceso actual

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    printf("Hola! Procesador %s, rango %d de %d procs.\n",
           processor_name, world_rank, world_size);

    MPI_Finalize();
}
```



# Comunicadores

- Un **comunicador** es una colección de procesos que pueden enviarse mensajes entre sí.
- `MPI_COMM_WORLD` es el comunicador principal
  - Incluye todos los procesos creados al inicio de la ejecución

# Funciones para obtener información

- `int MPI_Comm_size(MPI_Comm comm, int *size);`
  - Número de procesos en el comunicador
    - El resultado en el parámetro `size`
  - El fijo durante toda la ejecución
  - Se fija al inicio de la ejecución
- `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
  - Identificador del proceso en el comunicador
    - El resultado en el parámetro `rank`
  - El valor de `rank` será entre  $0 \dots \text{procesos} - 1$

# Envío y recepción de mensajes

```
...
/* Obtener el ID del proceso */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank != 0) {
    /* Crear el mensaje */
    sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
    /* Enviar mensaje al proceso 0 */
    MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
} else {
    printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
    for (int q = 1; q < comm_sz; q++) {
        /* Recibir mensaje enviado desde el proceso q */
        MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        printf("%s\n", greeting);
    }
}
```

# Envío y recepción de mensajes

- Funciones básicas de comunicación: `MPI_Send` y `MPI_Recv`
- `int MPI_Send (void *buffer , int tam, MPI_Datatype tipo , int destino , int tag , MPI_Comm comunicador)`
- `int MPI_Recv (void *buffer, int tam, MPI_Datatype tipo, int origen, int tag, MPI_Comm comunicador, MPI_Status *estado)`

# Envío y recepción de mensajes

- Funciones básicas de comunicación: `MPI_Send` y `MPI_Recv`
- Buffer es un array con el contenido a enviar o para recibir
- `MPI_Datatype` permite indicar el tipo de datos
- **destino** indica el proceso al que se envía
- **origen** indica el proceso desde el que se desea recibir
  - `MPI_ANY_SOURCE` – Para indicar que no importa quién lo envíe
- **tag** permite distinguir mensajes
  - `MPI_ANY_TAG` – Para indicar que es irrelevante
- **status** proporciona información sobre el mensaje (ej., ¿quién lo envía?)

# Mensajes – Tipos

- Tipos básicos
  - Mapping a C
- Tipos derivados
  - Definidos por el programador

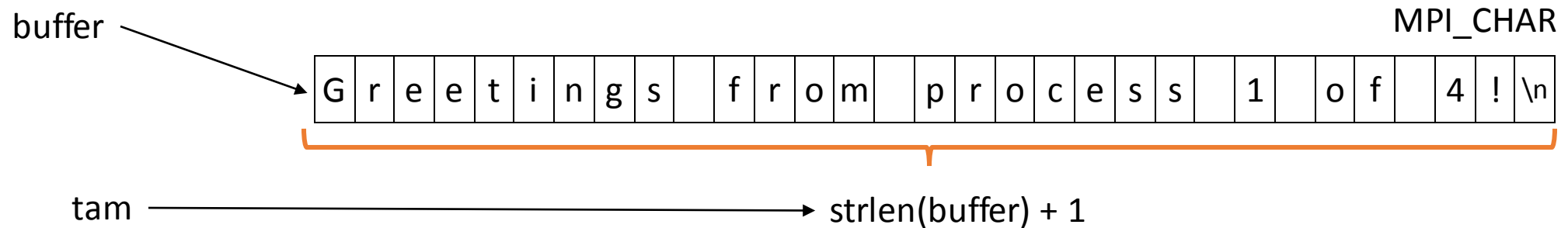
Tipo MPI	Tipo C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double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

# Envío de mensajes

```
int MPI_Send (  
    void *buffer,          /* In */  
    int tam,               /* In */  
    MPI_Datatype tipo,     /* In */  
    int destino,           /* In */  
    int tag,               /* In */  
    MPI_Comm comunicador /* In */  
)
```

Contenido del mensaje

Destino del mensaje

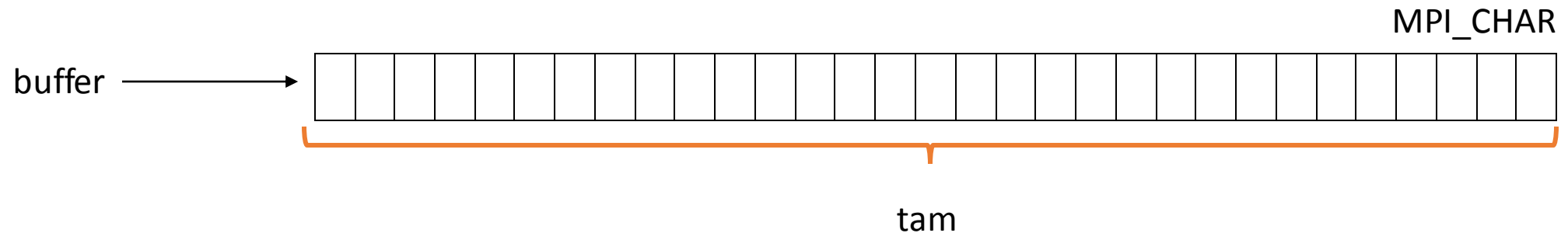


# Recepción de mensajes

```
int MPI_Recv (  
    void *buffer,          /* Out */  
    int tam,               /* In */  
    MPI_Datatype tipo,     /* In */  
    int origen,            /* In */  
    int tag,               /* In */  
    MPI_Comm comunicador, /* In */  
    MPI_Status *status  
)
```

Contenido del mensaje

Origen del mensaje



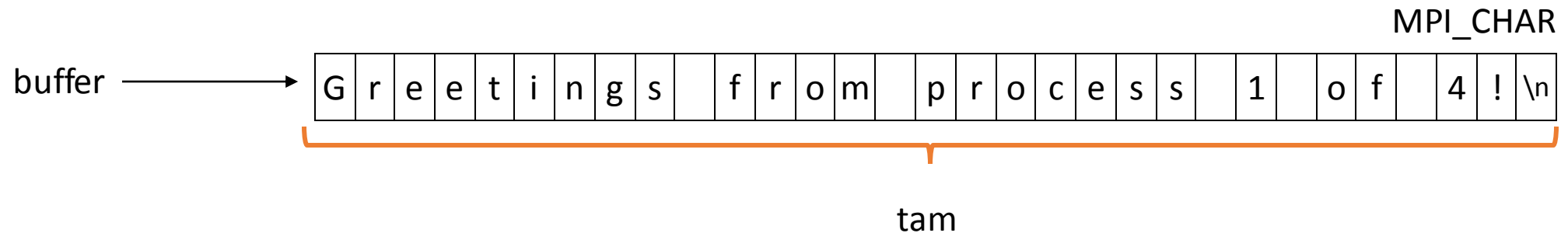


# Recepción de mensajes

```
int MPI_Recv (  
    void *buffer,          /* Out */  
    int tam,               /* In */  
    MPI_Datatype tipo,     /* In */  
    int origen,            /* In */  
    int tag,               /* In */  
    MPI_Comm comunicador, /* In */  
    MPI_Status *status  
)
```

Contenido del mensaje

Origen del mensaje



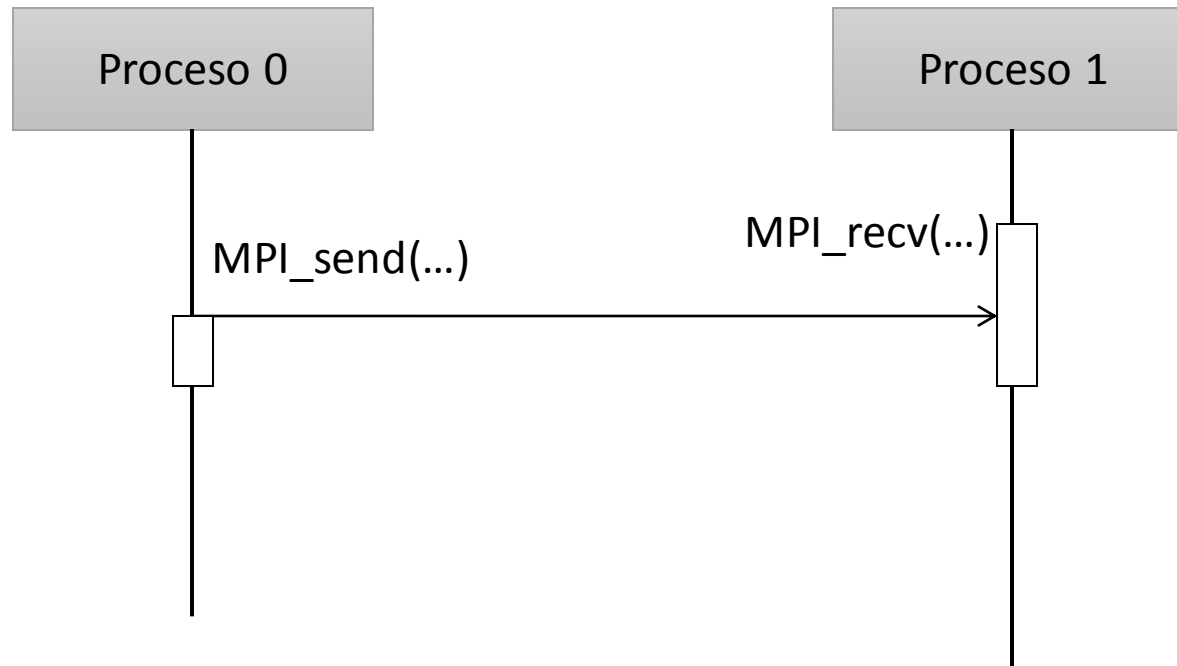
# Envío y recepción de mensajes

- Condiciones:
  - Número de proceso del MPI\_Send debe corresponder con el número de proceso MPI\_Recv o que se use MPI\_ANY\_SOURCE
  - Tag debe ser equivalente o MPI\_ANY\_TAG
  - Sólo se reciben los mensajes los procesos del mismo comunicador
  - Tipo de datos origen debe ser igual al tipo de datos destino
  - Tamaño del buffer destino debe ser mayor o igual que el buffer enviado

¿Qué hace esto?

```
for(i = 1; i < comm_sz; i++) {  
    MPI_Recv(buffer, tam, MPI_CHAR,  
             MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
    procesar(buffer);  
}
```

# Envío y recepción de mensajes



- Proceso emisor queda bloqueado hasta que se ha enviado el mensaje.
- Proceso receptor queda bloqueado hasta que le llega un mensaje
- Recepción con `MPI_Recv`
  - Bloqueante
- Recepción con `MPI_Send`
  - Bloqueante hasta que la red ha procesado el mensaje
  - No significa que haya llegado
  - Non-overtaking

# Envío y recepción de mensajes

- Preguntas
  - ¿Por qué puede interesar tener diferentes tags?
  - ¿Por qué puede interesar usar diferentes comunicadores?
  - ¿Cuándo utilizar MPI\_Status?

# Envío y recepción de mensajes

- El argumento **status** sirve para que el receptor recoja información sobre el mensaje recibido.
  - Tamaño real del mensaje
  - El emisor del mensaje
    - Si hemos usado MPI\_ANY\_SOURCE no se conoce a priori
  - El tag del mensaje
    - Se hemos usado MPI\_ANY\_TAG no se conoce a priori

```
status.MPI_SOURCE  
status.MPI_TAG
```

```
int MPI_Get_count(  
    MPI_Status * status_p, /* in */  
    MPI_Datatype type, /* in */  
    int * count_p /* out */  
)
```

# Envío y recepción de mensajes

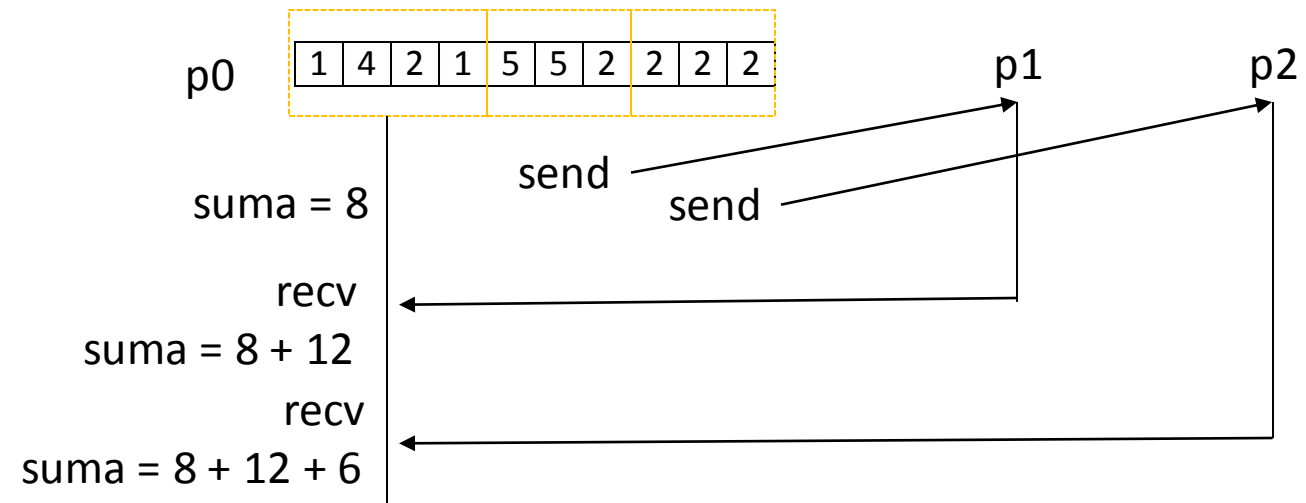
```
const int MAX_NUMBERS = 100;
int numbers[MAX_NUMBERS];
int number_amount;
if (rank == 0) {
    // Número aleatorio de elementos
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
    MPI_Status status;
    MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

    // ¿Cuántos se han recibido realmente?
    MPI_Get_count(&status, MPI_INT, &number_amount);
    printf("1 received %d numbers from 0. Message source = %d, "tag = %d\n",
           number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```

# Comunicaciones colectivas

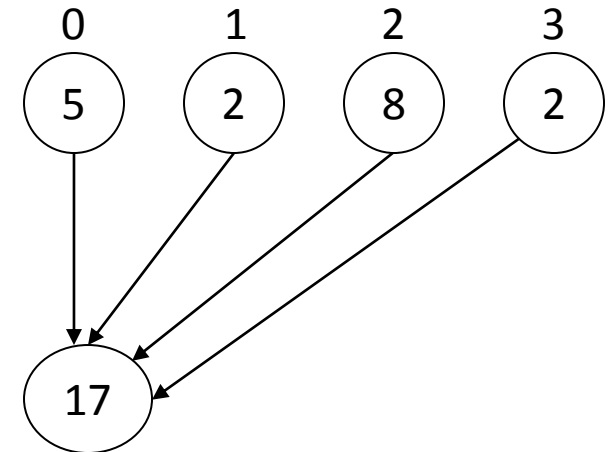
suma\_array\_no\_colectiva.c

- Las comunicaciones punto a punto pueden suponer un cuello de botella
  - Particionar los datos y enviar trabajos a los esclavos
  - Recoger el resultado por parte del proceso maestro
- Ejemplo
  - Suma de los valores de un array
  - p0 reparte el trabajo
  - p0 hace su parte de la suma
  - p1 send del resultado y p0 recv
  - p2 send del resultado y p0 recv



# Comunicaciones colectivas

- Funciones de comunicación que implican a todos los procesos de un comunicador
  - Recuerda que `MPI_Send` y `MPI_Recv` son punto a punto
  - Problemas por el cuello de botella en el maestro

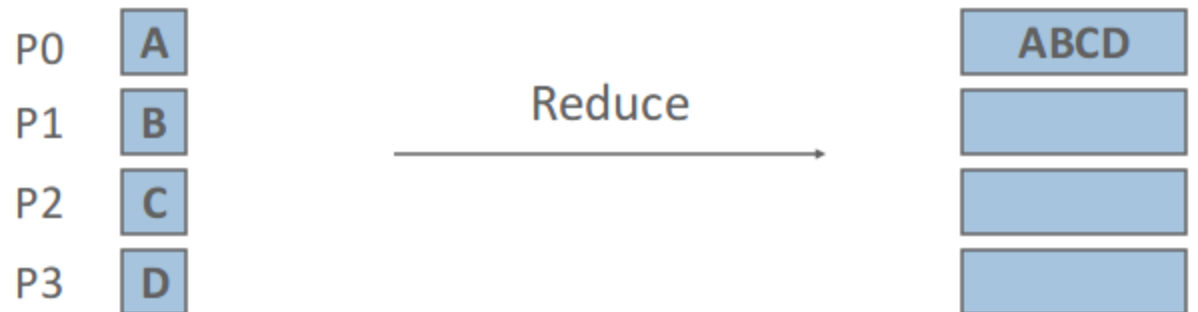
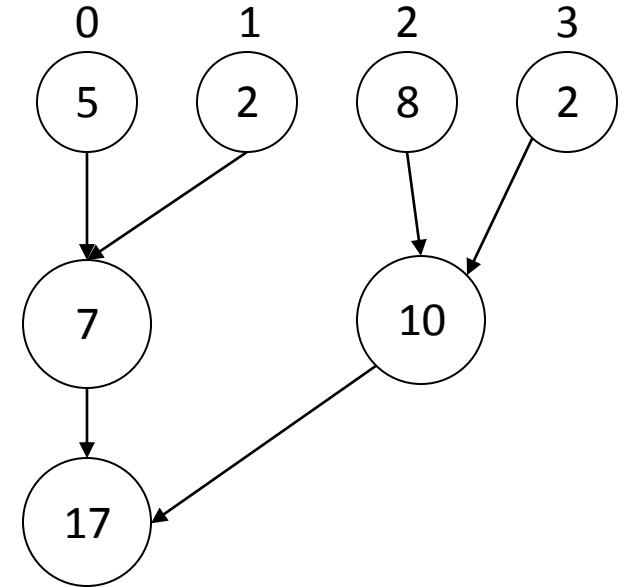




# Comunicaciones colectivas

- MPI\_Reduce

```
int MPI_Reduce(  
    void * resultado_local    /* in */,  
    void * resultado_global  /* out */,  
    int    count             /* in */,  
    MPI_Datatype datatype    /* in */,  
    MPI_Op operator          /* in */,  
    int    dest_process      /* in */,  
    MPI_Comm comm            /* in */)
```



# Comunicaciones colectivas

- Operadores de reducción

Enumerado	Operador
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_Prod	Producto
MPI_LAND	And lógico
MPI_BAND	And a nivel de bits
MPI_LOR	Or lógico
MPI_BOR	Or a nivel de bits
MPI_LXOR	Or exclusivo (XOR) lógico
MPI_BXOR	Or exclusivo a nivel de bits
MPI_MAXLOC	Máximo y localización del máximo
MPI_MINLOC	Mínimo y localización del mínimo

# Comunicaciones colectivas

- Uso de reduce
  - Todos los procesos invocan a MPI\_Reduce
  - Importante especifica el número del proceso que recoge el resultado
  - El valor del parámetro “resultado\_global” sólo es necesario en la invocación del proceso receptor del resultado

# Comunicaciones colectivas

## Punto a punto

- Send y recv puede invocarse por procesos diferentes
- El emparejamiento de send/recv se hace utilizando número de proceso, tag y comunicador

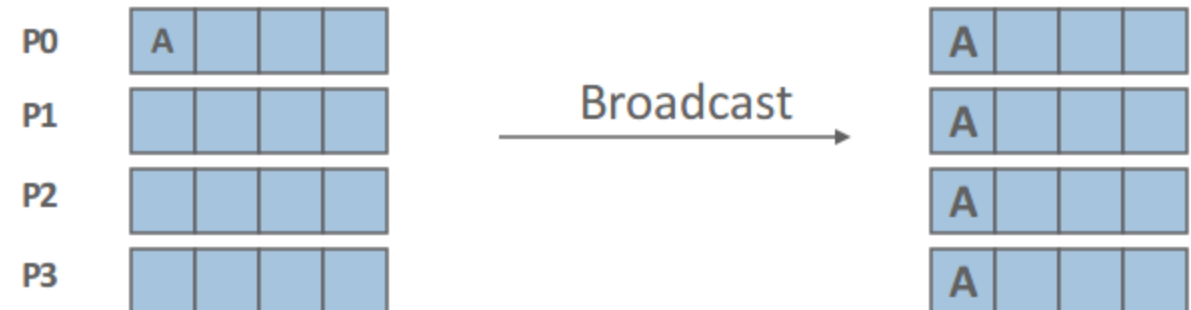
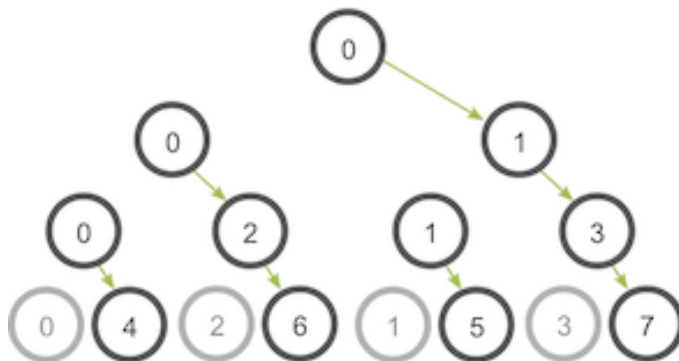
## Colectivas

- Todos los procesos deben invocar la operación colectiva
- El argumento `output_data_p` sólo lo debe usar el proceso destino. Los otros procesos pasan NULL.
- Todos los procesos del comunicador reciben el mensaje

# Comunicaciones colectivas

- Broadcast
  - Un proceso dispone de un dato que quiere distribuir al resto de procesos

```
int MPI_Broadcast(  
    void * datos          /* in/out */,  
    int    count          /* in */,  
    MPI_Datatype datatype /* in */,  
    int    origen         /* in */,  
    MPI_Comm comm         /* in */)
```



# Comunicaciones colectivas

- Broadcast
  - Ejemplo: Distribución de tres datos desde el proceso master

```
void get_data(int my_rank, float *a_ptr, float *b_ptr, int *n_ptr)
{
    int root=0;
    int count=1;
    if(my_rank==0) {
        printf("Enter a, b y n\n");
        scanf("%f %f %d",a_ptr,b_ptr,n_ptr);
    }
    MPI_Bcast(a_ptr,1,MPI_FLOAT,root,MPI_COMM_WORLD);
    MPI_Bcast(b_ptr,1,MPI_FLOAT,root,MPI_COMM_WORLD);
    MPI_Bcast(n_ptr,1,MPI_FLOAT,root,MPI_COMM_WORLD);
}
```

# Comunicaciones colectivas

- Broadcast
  - ¿Qué es más rápido Send/Recv o MPI\_Bcast?

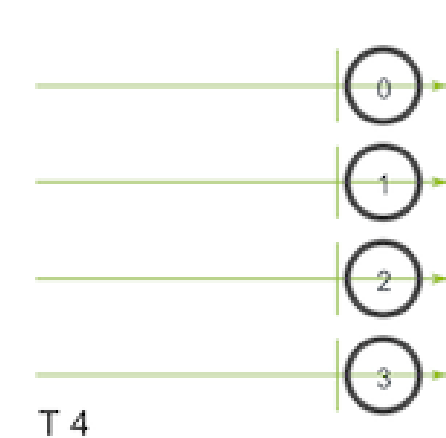
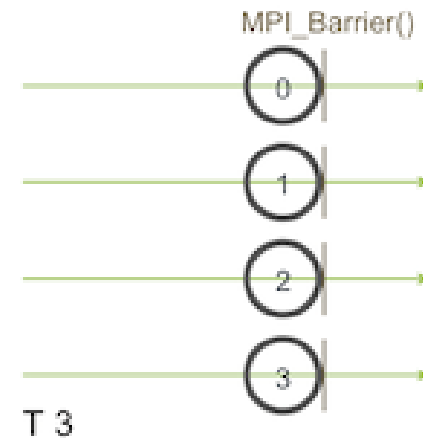
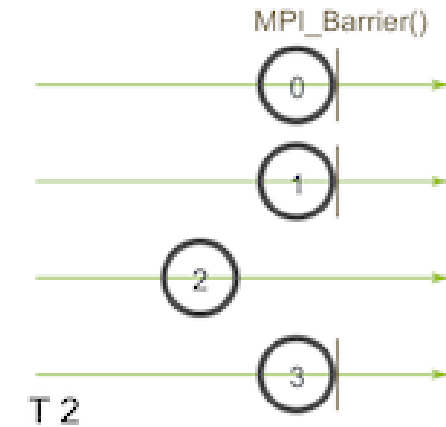
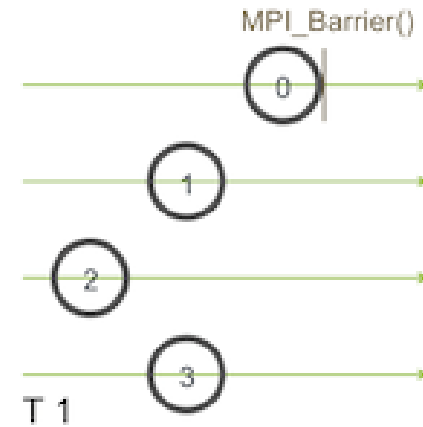
```
void my_bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm comm) {
    int world_rank;
    MPI_Comm_rank(comm, &world_rank);
    int world_size;
    MPI_Comm_size(comm, &world_size);

    if (world_rank == root) {
        for (int i = 0; i < world_size; i++) {
            if (i != world_rank) {
                MPI_Send(data, count, datatype, i, 0, comm);
            }
        }
    } else {
        MPI_Recv(data, count, datatype, root, 0, comm, MPI_STATUS_IGNORE);
    }
}
```

# Comunicaciones colectivas

- Barrier

```
MPI_Barrier(  
    MPI_Comm communicator)
```





# Comunicaciones colectivas

- Cálculo de PI utilizando MPI
  - ¿Cómo se implementaría el cálculo del número PI en MPI?

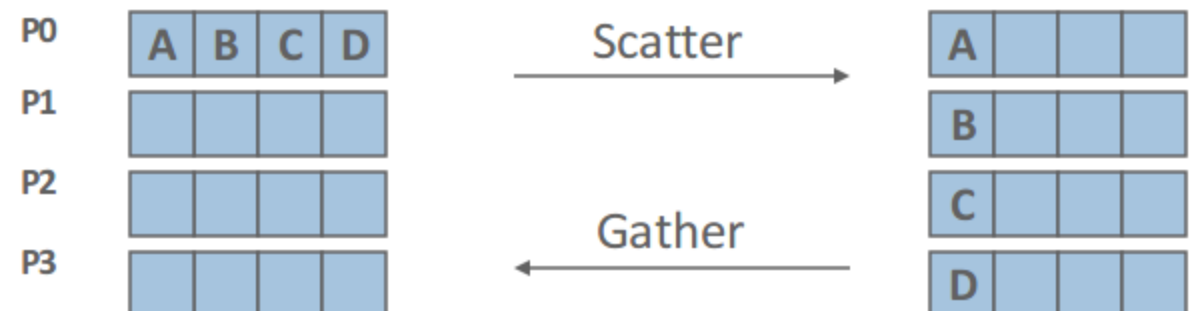
```
const long n = 1000000000;  
  
double PI25DT = 3.141592653589793238462643;  
double h = 1.0 / (double) n;  
double sum = 0.0;  
  
for (long i = 1; i <= n; i++) {  
    double x = h * ((double)i - 0.5);  
    sum += (4.0 / (1.0 + x*x));  
}  
double pi = sum * h;  
...
```

# Comunicaciones colectivas

- Scatter

- Divide un array en trozos y lo envía en partes iguales a todos los procesos
- send\_data, send\_count y send\_type definen el array
- recv\_count cuántos elementos del array recibe cada proceso
  - Normalmente tamaño(array) / N
- recv\_data el buffer de tamaño recv\_count

```
MPI_Scatter(  
    void* send_data,           /*in*/  
    int send_count,           /*in*/  
    MPI_Datatype send_type,    /*in*/  
    void* recv_data,          /*out*/  
    int recv_count,           /*in*/  
    MPI_Datatype recv_datatype, /*in*/  
    int root,                 /*in*/  
    MPI_Comm communicator     /*in*/) 
```



# Comunicaciones colectivas

vector\_add.c

- Scatter

- El proceso emisor configura los datos y los envía
- El proceso receptor recibe un fragmento
- El fragmento tiene que ser divisible entre el número de procesos – ¿y si no?

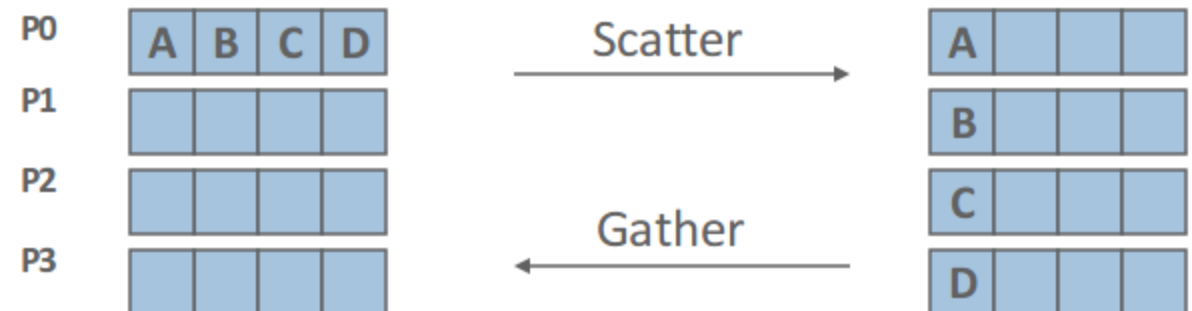
```
double *a;
int local_n = n / num_procesos;
double local_a = malloc(local_n*sizeof(double));
if (my_rank == 0) {
    a = malloc(n*sizeof(double));
    ... Llenar el vector ...
    MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
    free(a);
} else {
    MPI_Scatter(NULL, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
}
... Usar local_a ...
```

# Comunicaciones colectivas

vector\_add.c

- Gather
  - Operación “inversa” a scatter

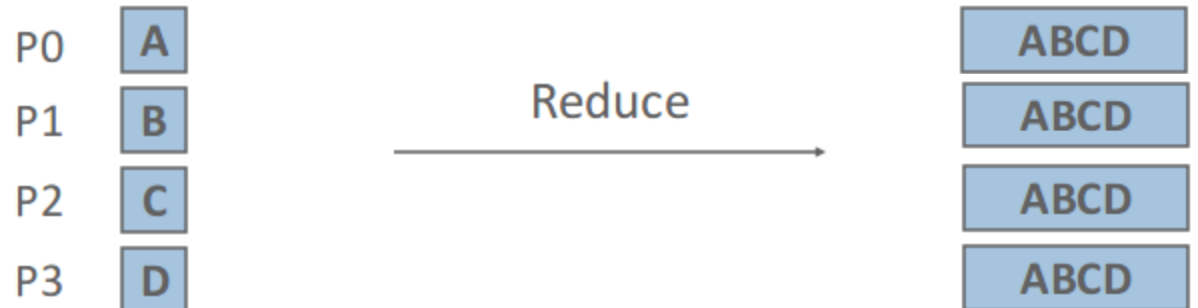
```
MPI_Gather(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    int root,  
    MPI_Comm communicator)
```



# Comunicaciones colectivas

- All reduce
  - Similar a reduce, pero todos los procesos reciben el resultado.
  - No requiere indicar el proceso destino (todos son el destino)
  - ¿Cómo se podría implementar Allreduce utilizando las primitivas que hemos visto?

```
int MPI_Allreduce(  
    void * resultado_local    /* in */,  
    void * resultado_global   /* out */,  
    int    count              /* in */,  
    MPI_Datatype datatype     /* in */,  
    MPI_Op operator           /* in */,  
    MPI_Comm comm             /* in */)
```



# Comunicaciones colectivas

- Ejemplo
  - Cálculo de la desviación estándar
  - Cada proceso genera una parte de la muestra (con números aleatorios)
  - La media la necesitan todos los procesos
    - Se distribuye la suma de los valores
  - Cada proceso calcula las diferencias
  - El proceso raíz agrega la suma de las diferencias

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

N = Tamaño de la muestra

$\mu$  = Media aritmética de la muestra

# Comunicaciones colectivas

- All gather
  - Útil cuando todos los procesos tienen que hacer operaciones en cadena con el resultado

```
MPI_Allgather(  
    void* send_data,          /* in */  
    int send_count,          /* in */  
    MPI_Datatype send_type,  /* in */  
    void* recv_data,         /* out */  
    int recv_count,          /* in */  
    MPI_Datatype recv_type,  /* in */  
    MPI_Comm Communicator    /* in */)
```



# Variantes de send

- `MPI_Send`
  - Se bloquea hasta que el buffer de envío está libera.
- `MPI_Bsend`
  - Se proporciona un buffer propio para que la llamada retorne inmediatamente.
- `MPI_Ssend`
  - Bloqueante hasta que se empareje con un `recv`.
- `MPI_Rsend`
  - Puede usarse solo si hay un `recv` que ya está listo para recibir. Debe usarse con extremo cuidado.
- `MPI_Isend`
  - Envío no bloqueante. No se puede reutilizar el buffer de envío hasta que se sepa que el mensaje ha sido recibido (comprobar con `wait/test`).



# Comunicación asíncrona

- Envío:
  - `MPI_Isend(buf, count, datatype, dest, tag, comm, request)`
- Recepción:
  - `MPI_Irecv(buf, count, datatype, source, tag, comm, request)`
- request se usa para saber si la operación ha acabado:
  - `MPI_Wait( )`
    - vuelve si la operación se ha completado, espera hasta que se completa.
  - `MPI_Test( )`
    - devuelve un flag diciendo si la operación se ha completado.

# Comunicación asíncrona

Proceso 0 envía,  
Proceso 1 recibe

```
MPI_Request request;
int request_complete = 0;

if (rank == 0) {
    MPI_Isend(buffer, buffer_count, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
    while (has_work) {
        do_work(); // ... Trabajo mientras que esperamos a que proceso 1 reciba

        if (!request_complete)
            MPI_Test(&request, &request_complete, &status);
    }
    // No queda trabajo, esperamos de manera inactiva
    if (!request_complete)
        MPI_Wait(&request, &status);
}
else {
    MPI_Irecv(buffer, buffer_count, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
    // ... Trabajo intermedio y al terminar esperamos el mensaje
    MPI_Wait(&request, &status);
}
```

# Empaquetamiento y tipos derivados

- Problema:
  - La comunicación puede consumir mucho más tiempo que la computación
  - Se debe minimizar el número de mensajes
  - ¿Cuántos mensajes hacen falta para enviar 1, “a” y 2.2?

# Empaquetamiento

- Escenario de uso: para enviar mensajes que contienen:
  - Datos no contiguos de un único tipo (ej., un sub-bloque de una matriz)
  - Datos contiguos de tipos diferentes (ej., un entero seguido de una secuencia de números reales)
  - Datos no contiguos de diferentes tipos
- Dos soluciones:
  - Realizar tantos send (y los recv correspondientes) como elementos de datos se tengan.
  - Se empaquetan los datos en un buffer contiguo antes de enviarlos, y se desempaquetan al recibirlos. Utilizar MPI\_Pack y MPI\_Unpack para empaquetar/dempaquetar los datos. Corresponde al tipo de datos MPI\_Packed.

# Empaquetamiento

- **Empaqueta** in\_count datos de tipo datatype .
- pack data referencia los datos a empaquetar en el buffer, que debe consistir de size bytes (puede ser una cantidad mayor a la que se va a ocupar).
- El parámetro position\_ptr es de entrada y salida. Como entrada, el dato se copia en la posición buffer+\*position\_ptr . Como salida, referencia la siguiente posición en el buffer después del dato empaquetado.

```
int MPI_Pack(void *pack_data,
             int in_count,
             MPI_Datatype datatype,
             void *buffer,
             int size,
             int *position_ptr /*inout*/,
             MPI_Comm comm)
```

# Empaquetamiento

- **Desempaqueta** count elementos de tipo datatype en unpack data , tomándolos de la posición buffer+\*position ptr del buffer .
- Hay que decir el tamaño del buffer ( size ) en bytes, y position ptr es manejado por MPI.

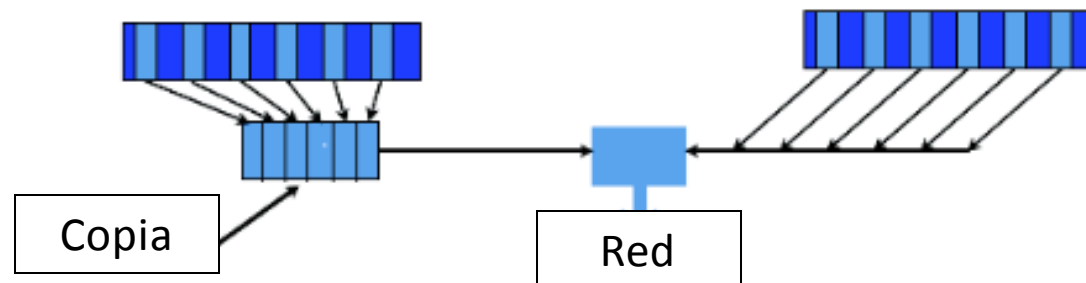
```
int MPI_Unpack(void *buffer,  
               int size,  
               int *position_ptr,  
               void *unpack_data,  
               int count,  
               MPI_Datatype datatype,  
               MPI_Comm comm)
```

# Empaquetamiento

```
void Get_data(int my_rank, float *a_ptr, float *b_ptr, int *n_ptr) {
    int root=0 ; char buffer[100] ; int position = 0;
    if(my_rank==0) {
        printf("Enter a, b y n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
        MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);
        MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);
        MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100, &position, MPI_COMM_WORLD);
        MPI_Bcast(buffer, 100, MPI_PACKED, root, MPI_COMM_WORLD);
    } else {
        MPI_Bcast(buffer, 100, MPI_PACKED, root, MPI_COMM_WORLD);
        MPI_Unpack(buffer, 100, &position, a_ptr, 1, MPI_FLOAT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, 100, &position, b_ptr, 1, MPI_FLOAT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, 100, &position, n_ptr, 1, MPI_INT, MPI_COMM_WORLD);
    }
}
```

# Tipos derivados

- En MPI los tipos derivados son aquellos que se construyen a partir de los tipos básicos (ej. MPI\_INT, MPI\_CHAR).
  - Los define el programador utilizando funciones proporcionadas por MPI
- Los tipos derivados de MPI proporcionan un manera portable de comunicar datos no-contiguos y con tipos diferentes en un mismo mensaje.
- Durante la comunicación, MPI se encarga de recolectar los datos para ponerlos en el buffer de envío y de re-codificarlos en la recepción.



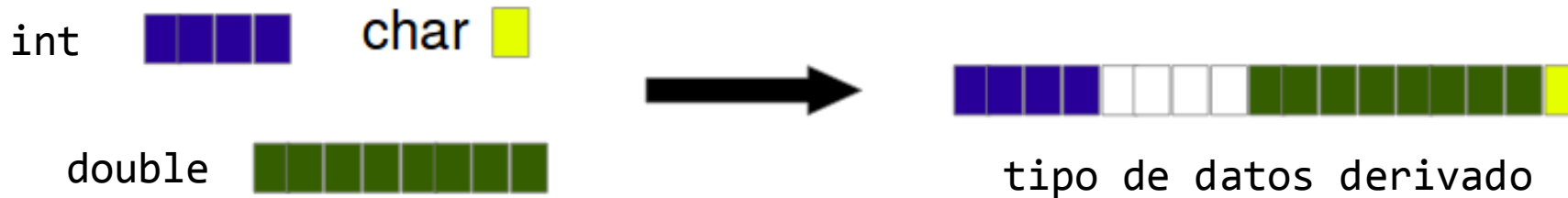


# Tipos derivados

- Un tipo de datos es un objeto opaco que describe la organización de un buffer de memoria especificando:
  - Una secuencia de tipos de datos básicos
  - Una secuencia de desplazamientos
- Typemap = {(tipo 0, desplazamiento 0), ... (tipo n-1, desplazamiento-1)}
  - Pares de tipos básicos y desplazamientos (en bytes)
- Signatura del tipo= {tipo 0, tipo 1, ... tipo n-1}
  - Lista de tipos
  - Permite a MPI saber el tamaño de cada uno de los elementos para saber cómo procesar los envíos y recepciones
- Desplazamiento:
  - Le dice a MPI de donde obtener los datos para enviar o dónde ponerlos al recibir

# Tipos derivados

- Los tipos básicos son un caso particular:
  - `MPI_INT = {(int ,0)}`
- Ejemplo de tipo de datos derivado
  - `Tipo = {(int,0), (double,8), (char,16)}`



# Tipos derivados

Tipo
MPI_TYPE_CONTIGUOUS
MPI_TYPE_VECTOR
MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_INDEXED
MPI_TYPE_CREATE_INDEXED_BLOCK
MPI_TYPE_CREATE_SUBARRAY
MPI_TYPE_CREATE_DARRAY
MPI_TYPE_CREATE_STRUCT

# Tipos derivados

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

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

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

# Tipos derivados

```
void Build_derived_type(MYDATA_TYPE *indata, MPI_Datatype *message_type_ptr) {
    int block_lenghts[3] = {1, 1, 1};
    MPI_Aint displacements[3];
    MPI_Aint addresses[4];
    MPI_Datatype typelist[3] = {MPI_FLOAT, MPI_FLOAT, MPI_INT};
    MPI_Get_address(indata, &addresses[0]);
    MPI_Get_address(&(indata->a), &addresses[1]);
    MPI_Get_address(&(indata->b), &addresses[2]);
    MPI_Get_address(&(indata->n), &addresses[3]);
    displacements[0]=addresses[1]-addresses[0];
    displacements[0]=0;
    displacements[1]=addresses[1]-addresses[0];
    displacements[2]=addresses[2]-addresses[0];
    MPI_Type_struct(3, block_lenghts, displacements, typelist, message, type_ptr);
    MPI_Type_commit(message_type_ptr);
}
```

# Tipos derivados

```
void get_data(MYDATA_TYPE *indata , int my_rank) {  
    MPI_Datatype message_type;  
    int root=0;  
    int count=1;  
    if(my_rank==0) {  
        printf("Enter a, b y n\n");  
        scanf("%f %f %d",&(indata->a),&(indata->b), &(indata->n));  
    }  
    Build_derived_type(indata, &message_type);  
    MPI_Bcast(indata, count, message_type, root, MPI_COMM_WORLD);  
}
```

# Compilación y ejecución de programas MPI

- Wrapper sobre GCC
  - `mpicc mi_programa.c`
- Obtener la línea de comandos que hay que escribir
  - `mpicc --showme:compile`
  - Para obtener la información de las rutas de las librerías (.so y .h)
- Ejecución
  - `mpirun -np [num_procesos] ./mi_programa`

\* <https://www.open-mpi.org/faq/?category=mpi-apps>

# Tomar tiempos

- Se toman en el proceso maestro

```
double start = MPI_Wtime;  
    ... código paralelo ...  
double end = MPI_Wtime;  
printf("Tiempo ejecución = %2.4f segundos\n",  
       finish - start);
```



# Ejemplos

- <https://computing.llnl.gov/tutorials/mpi/exercise.html>
  - Ejercicios de un tutorial
- [https://people.sc.fsu.edu/~jburkardt/c\\_src/mpi/mpi.html](https://people.sc.fsu.edu/~jburkardt/c_src/mpi/mpi.html)
  - Ejemplos de algunos problemas interesantes (ej., cálculo de primos).
- <https://mpitutorial.com/tutorials/>
  - Tutoriales avanzados de MPI

# Bibliografía

- An Introduction to Parallel Programming, Peter Pacheco
  - Disponible en el Aula Virtual
- Introducción a la Programación Paralela, Domingo Giménez Cánovas
  - Disponible en la biblioteca
- MPI Tutorials
  - <https://mpitutorial.com/tutorials/>
- MPI for Dummies
  - [https://hlor.inf.ethz.ch/teaching/mpl\\_tutorials/ppopp13/](https://hlor.inf.ethz.ch/teaching/mpl_tutorials/ppopp13/)
- Parallel Programming in MPI and OpenMP (Victor Eijkhout)
  - <http://pages.tacc.utexas.edu/~eijkhout/pcse/html/>