

Sistemas Distribuidos y Paralelos

Ingeniería en Computación



Modelo de programación sobre memoria distribuida

Universidad Nacional de La Plata



Facultad de Informática



Agenda

2

- I. Modelo de programación sobre memoria distribuida: Introducción
- II. Parallel Virtual Machine (PVM)
- III. Message Passing Interface (MPI)
 - i. Funcionamiento, compilación y ejecución
 - ii. Estructura de programa
 - iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
 - iv. Ocultamiento de la latencia

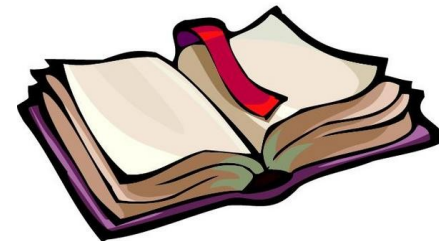


I. Modelo de programación sobre memoria distribuida: Introducción

II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

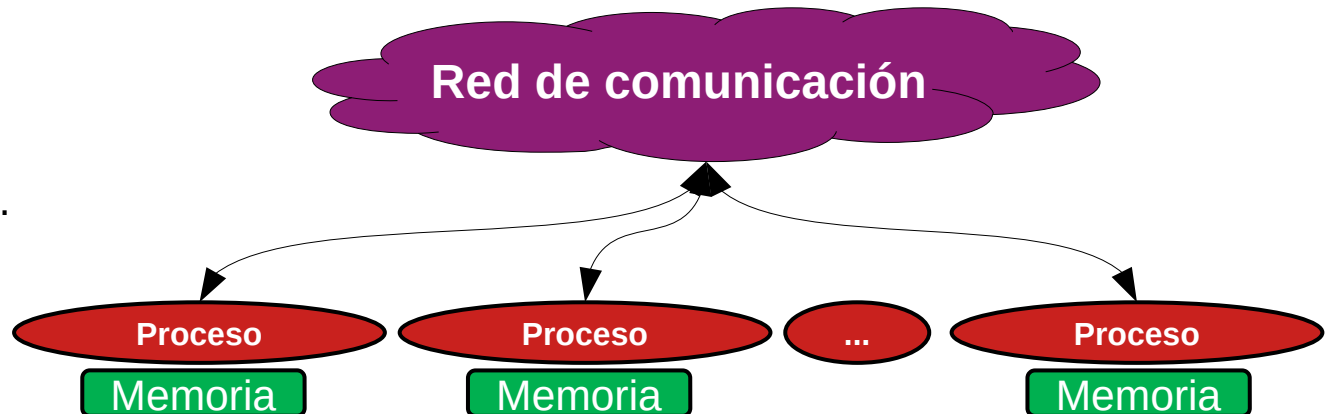
- i. Funcionamiento, compilación y ejecución
- ii. Estructura de programa
- iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
- iv. Ocultamiento de la latencia



Modelo de programación sobre memoria distribuida

4

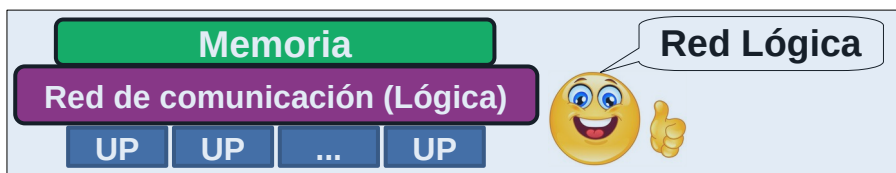
- En el modelo de programación sobre memoria distribuida cada proceso tiene su propia memoria local/privada.
- Un proceso no puede acceder al espacio de memoria de otro proceso.
- Los procesos se comunican y sincronizan mediante el envío y recepción de mensajes a través de una red de comunicación.
- Existen diversas herramientas para el desarrollo de aplicaciones utilizando este modelo:
 - PVM
 - MPI
 - Otras: Sockets, RMI, etc.



Modelo de programación sobre memoria distribuida

Modelos de programación paralela – Modelos de arquitectura paralela

- El modelo de programación sobre memoria distribuida puede utilizarse en arquitecturas de memoria compartida, arquitecturas de memoria distribuida o híbridos.



		Software		
		Memoria Compartida (Ej: OpenMP)	Memoria Distribuida (Ej: MPI)	Híbrido (EJ: MPI + OpenMP)
Hardware	Memoria Compartida (Ej: multicore)	Trivial (Ej: OpenMP sobre Multicore)	Posible (Ej: MPI sobre multicore)	Posible (extraño) (Ej: MPI + OpenMP sobre multicore)
	Memoria Distribuida (Ej: cluster moncore ¹)	(NO ADECUADO) Single System Image(SSI) Overhead	Trivial (Ej: MPI sobre cluster)	Posible (muy extraño) (Ej: MPI + OpenMP sobre cluster moncore)
	Híbrido (Ej: cluster multicore)	(NO ADECUADO) Single System Image(SSI) Overhead	Posible (Ej: MPI sobre cluster multicore)	Trivial (Ej: MPI + OpenMP sobre cluster multicore)

¹Actualmente, los clusters moncore cayeron en desuso pero se mencionan para ejemplificar.

I. Modelo de programación sobre memoria distribuida: Introducción

II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

- i. Funcionamiento, compilación y ejecución
- ii. Estructura de programa
- iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
- iv. Ocultamiento de la latencia



- **PVM** (Parallel Virtual Machine): desarrollada por la Universidad de Tennessee. Fue la herramienta que **impulsó el uso de clusters**.
 - Versión 1 – 1989: escrita en ORNL.
 - Versión 2 – 1991: escrita para C, C++ y Fortran
 - Versión 3 – 1993: mejoras en la tolerancia a fallas y portabilidad
 - **Última versión 2009**

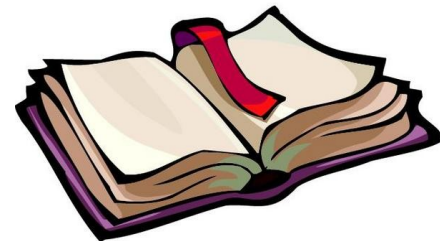


I. Modelo de programación sobre memoria distribuida: Introducción

II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

- i. Funcionamiento, compilación y ejecución
- ii. Estructura de programa
- iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
- iv. Ocultamiento de la latencia



I. Modelo de programación sobre memoria distribuida: Introducción

II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

- i. Funcionamiento, compilación y ejecución
- ii. Estructura de programa
- iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
- iv. Ocultamiento de la latencia



- **Message Passing Interface (MPI)** es un estándar¹ que define la sintaxis y la semántica de funciones para pasaje de mensajes. **No define los comandos.**
- Existen varias distribuciones que implementan el estándar:
 - OpenMPI
 - Mpich
 - Otros...
- Implementado para los lenguajes:
 - C
 - C++
 - Fortran
- Para otros lenguajes (**Java** o **Python**) son **wrappers** bajo lenguaje C.

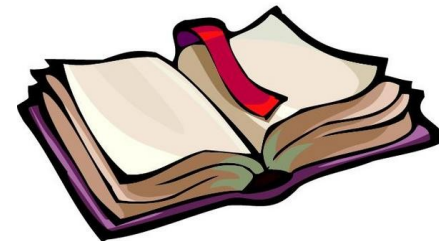
¹<http://www.mpi-forum.org>

I. Modelo de programación sobre memoria distribuida: Introducción

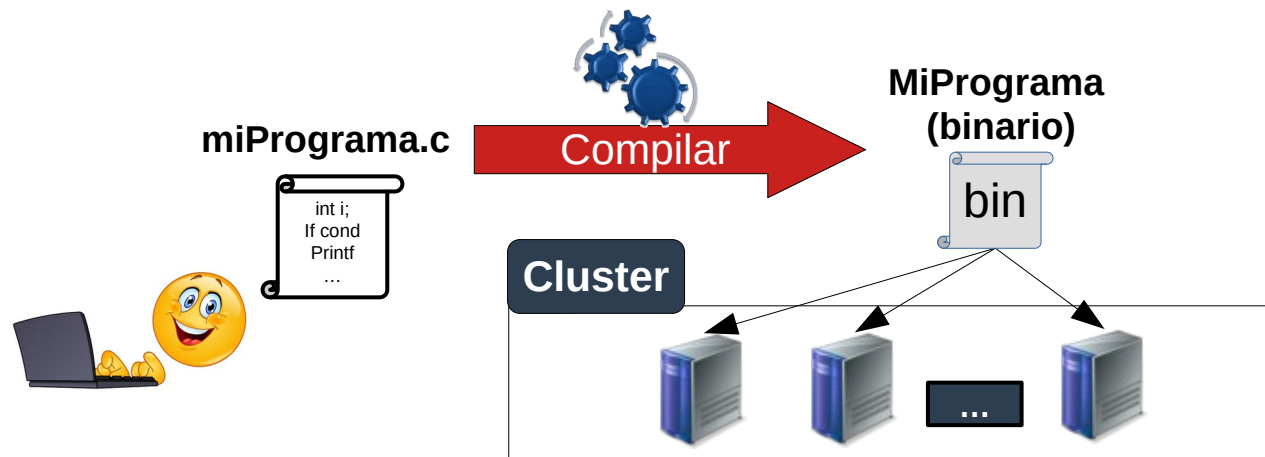
II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

- i. Funcionamiento, compilación y ejecución
- ii. Estructura de programa
- iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
- iv. Ocultamiento de la latencia



- Para ejemplificar el funcionamiento de MPI, suponemos un **cluster** con varias máquinas conectas a una red de comunicación física y que usamos **OpenMPI**.
- Cada máquina debe conocer el **archivo ejecutable (binario)**.
- MPI no compila automáticamente el archivo fuente ni distribuye el binario.
- El programador debe compilar el código fuente y distribuir el binario.



MPI

Compilación

13

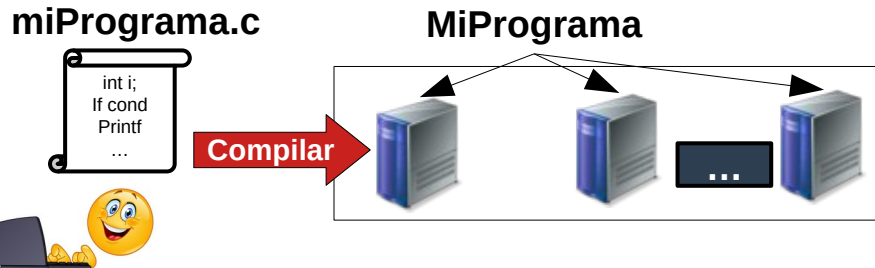
- Un programa MPI se compila de la siguiente forma:

```
mpicc -o miPrograma miPrograma.c
```

- La compilación depende de las características del cluster:

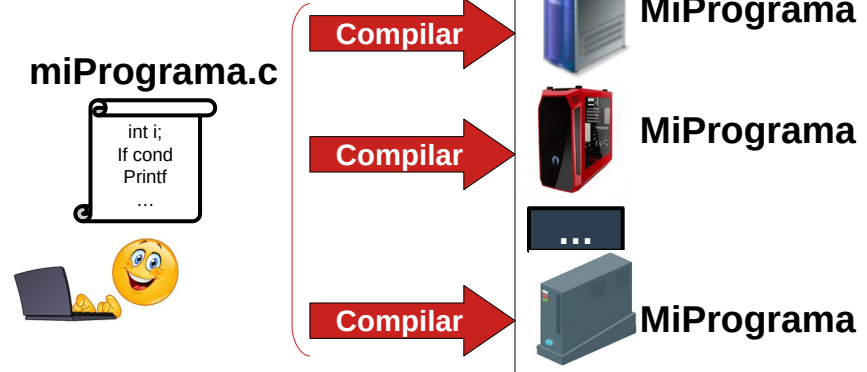
Cluster Homogéneo

- Todas las máquinas iguales.
- Compilar el código fuente en una de las máquinas y luego distribuir el binario o compartirlo con un sistema de archivos distribuido (Por ejemplo:NFS).



Cluster Heterogéneo

- Las máquinas pueden ser diferentes.
- Compilar el código fuente en cada máquina. Cada máquina tendrá su archivo binario.



MPI

Ejecución

14

- Un programa MPI se ejecuta de la siguiente forma:

```
mpirun -np NrProcesos -machinefile maquinas miPrograma
```

- **NrProcesos:** es el número de procesos a crear
- **maquinas:** es un archivo que contiene el nombre de las máquinas a utilizar. Su composición varía dependiendo de la distribución de MPI. En **OpenMPI**:

Cluster



Maquina A



Maquina B



Maquina C

Archivo “maquinas”

maquinaA slots=2
maquinaB slots=1
maquinaC slots=1

slots indica el número de procesos que se le enviarán a cada máquina

MPI

Ejecución

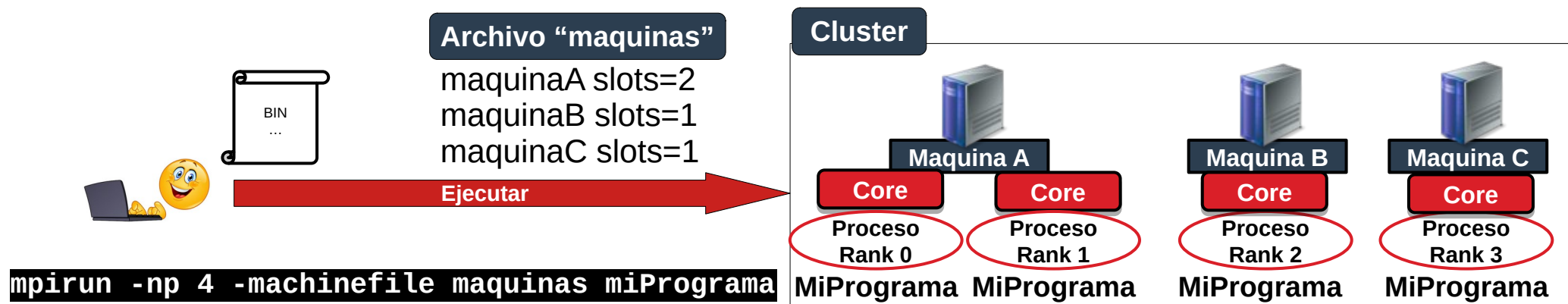
15

- Se envía a ejecutar desde una máquina (no necesariamente del cluster) y el **Runtime System de MPI** automáticamente desencadena la ejecución de una copia del programa en cada unidad de procesamiento



Todas las unidades de procesamiento ejecutan una copia del mismo programa

- El **Runtime System de MPI** automáticamente asigna un identificador único (**rank**) a cada **proceso** creado. Generalmente, lo asigna en el orden del archivo de máquinas.

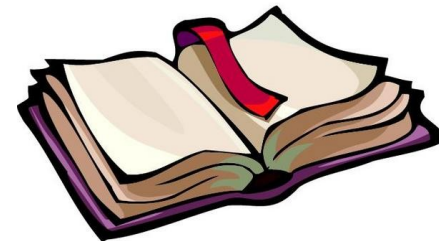


I. Modelo de programación sobre memoria distribuida: Introducción



II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

- i. Funcionamiento, compilación y ejecución
- ii. Estructura de programa
- iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
- iv. Ocultamiento de la latencia



Estructura de programa – Funciones principales

- Todas las funciones MPI llevan el prefijo **MPI_** delante.
- Las cuatro **funciones** principales **de control** son:
 - **MPI_init**: inicializa el ambiente de ejecución MPI. Recibe como parámetros los argumentos que recibió el programa C. **Es la primera sentencia que debe ejecutarse** después de la definición de variables.
 - **MPI_Comm_rank**: permite obtener el identificador de proceso (0..P) asignado por el Runtime System de MPI.
 - **MPI_Comm_size**: permite obtener el número de procesos creados.
 - **MPI_Finalize**: termina la ejecución del ambiente MPI. **Es la última función que debe ejecutarse** antes de la sentencia return.

- Varias funciones MPI reciben como parámetro el nombre de un **comunicador**.
- Básicamente, un comunicador se utiliza para agrupar procesos.
- El uso de comunicadores en MPI ofrece ciertas ventajas:
 - Organizar procesos y armar topologías virtuales.
 - Comunicar procesos en un grupo de manera independiente del resto.
- **MPI_COMM_WORLD** es el comunicador por defecto que incluye a todos los procesos en la ejecución.

No utilizaremos comunicadores especiales y usaremos **MPI_COMM_WORLD** en todos los casos.



```
#include<mpi.h> //Se debe incluir la cabecera

int main(int argc, char** argv){
    int miID;
    int nrProcesos;

    MPI_Init(&argc, &argv); // Inicializa el ambiente. No debe haber sentencias antes
    MPI_Comm_rank(MPI_COMM_WORLD,&miID); // Obtiene el identificador de cada proceso (rank)
    MPI_Comm_size(MPI_COMM_WORLD,&nrProcesos); // Obtiene el número de procesos
    ...
    if(miID == 0)
        funcionProcesoTipoA(); // Función que implementa los procesos de tipo A
    else if (miID >= 1 && miID <= K)
        funcionProcesoTipoB(); // Función que implementa los procesos de tipo B
    else if
        ...
    else
        funcionProcesoTipoZ(); // Función que implementa los procesos de tipo Z

    MPI_Finalize(); // Finaliza el ambiente MPI. No debe haber sentencias después
    return(0); // Luego de MPI_Finalize()
}
```

```
tipo_t miVariable;
```

```
int main(int argc, char** argv){  
    int miID;  
    int nrProcesos;  
  
    MPI_Init(&argc, &argv);  
    MPI_Comm_rank(MPI_COMM_WORLD, &miID);  
    MPI_Comm_size(MPI_COMM_WORLD, &nrProcesos);  
    ...  
    if(miID == 0)  
        funcionProcesoTipoA();  
    else if (miID >= 1 && miID <= K)  
        funcionProcesoTipoB();  
    else if  
        ...  
    else  
        funcionProcesoTipoZ();  
  
    MPI_Finalize();  
    return(0);  
}
```



Una variable definida en ésta área...

NO ES UNA VARIABLE COMPARTIDA

...entre los procesos.

Estamos en un modelo de memoria distribuida

NO HAY VARIABLES COMPARTIDAS

Recordar:



Todas las unidades de procesamiento ejecutan una copia del mismo programa.

Existirá una copia del programa para cada proceso. Por lo tanto, existirá una copia de la variable para cada proceso.

VARIABLE LOCAL Y PRIVADA

I. Modelo de programación sobre memoria distribuida: Introducción

II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

i. Funcionamiento, compilación y ejecución

ii. Estructura de programa

iii. Comunicación

i. Operaciones punto a punto

ii. Operaciones colectivas

iv. Ocultamiento de la latencia



MPI

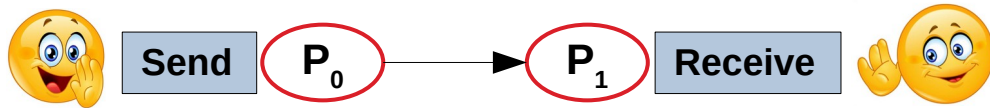
Comunicación

22

- Los procesos MPI se **comunican** mediante pasaje de mensajes.
- Las operaciones de comunicación pueden clasificarse en:

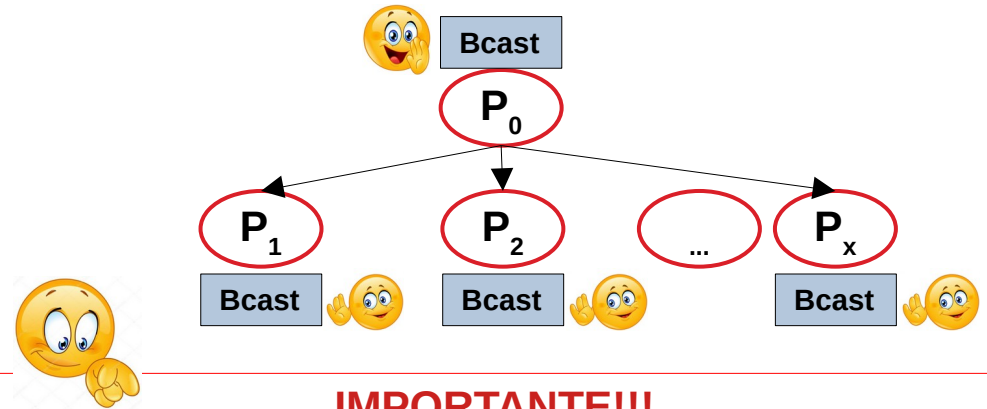
Punto a Punto

Comunican sólo **dos procesos**.
Un proceso hace un “**tipo**” de **send** y otro proceso hace un “**tipo**” de **receive**.



Colectivas

Comunican **varios procesos**:



IMPORTANTE!!!

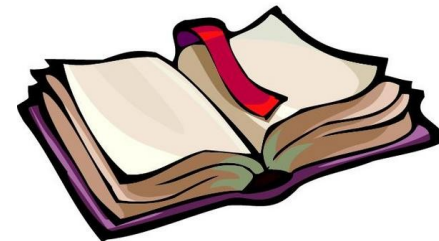
El uso de operaciones **Colectivas** requiere que todos los procesos ejecuten la **misma función de comunicación**.

I. Modelo de programación sobre memoria distribuida: Introducción

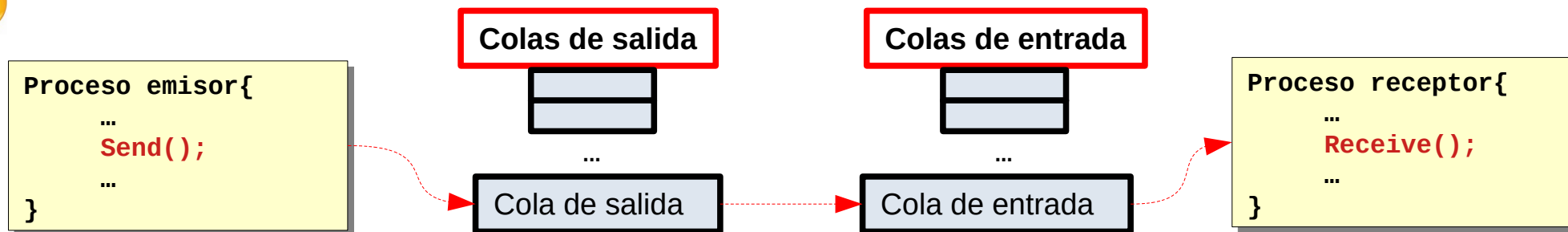
II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

- i. Funcionamiento, compilación y ejecución
- ii. Estructura de programa
- iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
- iv. Ocultamiento de la latencia



- Las operaciones básicas de comunicación punto a punto entre procesos MPI son:
 - Envío de mensajes: **MPI_Send**
 - Recepción de mensajes: **MPI_Recv**
- Se debe tener en cuenta el siguiente diagrama:



Una operación puede ser:

- **Bloqueante:** si por alguna razón la sentencia (send o receive) no retorna el control al programa.
- **No Bloqueante:** si la sentencia retorna el control de inmediato independientemente de lo que ocurra con el mensaje.
- **Sincrónica:** si el programa que envía no puede continuar hasta que no haya una recepción.

Comunicación punto a punto – Parámetros comunes

- Las funciones que envían o reciben mensajes tienen parámetros comunes:
 - Buffer:** representa el mensaje a enviar o recibir
 - Tipo:** el tipo de datos de los elementos del buffer. MPI define los tipos:

MPI_BYTE	MPI_SHORT	MPI_INT	MPI_LONG
MPI_FLOAT	MPI_DOUBLE	MPI_UNSIGNED_CHAR	MPI_UNSIGNED_SHORT
MPI_UNSIGNED	MPI_UNSIGNED_LONG	MPI_LONG_DOUBLE	MPI_LONG_LONG_INT

- Tamaño del mensaje:** cantidad de elementos del tipo anterior que contiene el buffer
- Comunicador:** el comunicador que agrupa a los procesos
- Source:** identificador de proceso a quien se enviará o de quién se recibirá el mensaje
- Tag:** cada proceso puede enviar o recibir por distintos tag (símil canales)

- **MPI_Send**: es un envío bloqueante.

```
int MPI_Send( const void *buffer,  
              int count,  
              MPI_Datatype dtype,  
              int source,  
              int tag,  
              MPI_Comm comm)
```

- La sentencia no retorna el control al programa hasta que el mensaje haya sido almacenado de forma segura, de manera que el emisor pueda modificar libremente el mensaje enviado.
- El mensaje puede ser copiado en una **cola de entrada** del receptor o en una **cola de salida** temporal (del emisor).
- En **source** debe indicarse el **rank o Id del receptor**.

- **MPI_Recv**: es una recepción bloqueante.
- En **source** debe indicarse:
 - El **rank o Id del emisor** si se conoce el emisor.
 - La constante **MPI_ANY_SOURCE** si se desconoce el rank o Id emisor.
- En **tag** debe indicarse:
 - El **número de tag** por el que el emisor envió, si se conoce.
 - La constante **MPI_ANY_TAG** si se desconoce el tag por el que el emisor envió.
- Si se utilizan las opciones **_ANY_** se necesita el **argumento status** que es una estructura que permite obtener el source y el tag. (Puede ignorarse utilizando MPI_STATUS_IGNORE)

```
int MPI_Recv(void *buffer,
             int count,
             MPI_Datatype dtype,
             int source,
             int tag,
             MPI_Comm comm,
             MPI_Status *status)
```

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

Comunicación punto a punto – Ejemplo simple

```
#include<mpi.h>

int main( int argc, char *argv[]){
char message[20];
int myrank;
MPI_Status status;

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

if (myrank == 0) { // código del proceso 0
    strcpy(message,"Hello, there");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
}else if (myrank == 1){ // código del proceso 1
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("received :%s:\n", message);
}

MPI_Finalize();
return 0;
}
```

```
#include<mpi.h>

int main( int argc, char *argv[]){
char message[20];
int myrank;
MPI_Status status;

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

if (myrank == 0) { // código del proceso 0
    strcpy(message,"Hello, there");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
}else if (myrank == 1){ // código del proceso 1
    MPI_Recv(message, 20, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    printf("received :%s: source:%d tag:%d\n", message, status.MPI_SOURCE, status.MPI_TAG);
}

MPI_Finalize();
return 0;
}
```

Comunicación punto a punto - Modos

- MPI provee distintos **modos** de comunicación punto a punto que se indican con prefijos:

Prefijo B

Buffered
(Bloqueante)

Prefijo S

Synchronous
(Bloqueante)

Prefijo R

Ready
(Bloqueante)

Prefijo I

Immediate
(NO Bloqueante)



- **MPI_Bsend:** buffered (Bloqueante).

```
int MPI_Bsend(const void *buffer,  
              int count,  
              MPI_Datatype dtype,  
              int dest,  
              int tag,  
              MPI_Comm comm)
```

- **Si hay un receive:** el mensaje se envía y el emisor continua.
- **Si no hay un receive:** el mensaje se almacena en una **cola de salida** local (del emisor) y luego el emisor continua:
 - Si no hay espacio suficiente en la cola de salida **MPI_Bsend** retorna un **error**.
 - La cantidad de espacio disponible en la cola de salida es controlada por el usuario.

- **MPI_Ssend:** synchronous (Bloqueante).
- **Si hay un receive:** el emisor continua.
- **Si no hay un receive:** el emisor espera.
- La finalización de un synchronous send indica que el buffer puede ser usado y también que el receptor ha alcanzado cierto punto en su ejecución.

```
int MPI_Ssend(const void *buffer,  
              int count,  
              MPI_Datatype dtype,  
              int dest,  
              int tag,  
              MPI_Comm comm)
```


- **MPI_Rsend:** ready (Bloqueante).

```
int MPI_Rsend(const void *buffer,
              int count,
              MPI_Datatype dtype,
              int dest,
              int tag,
              MPI_Comm comm)
```

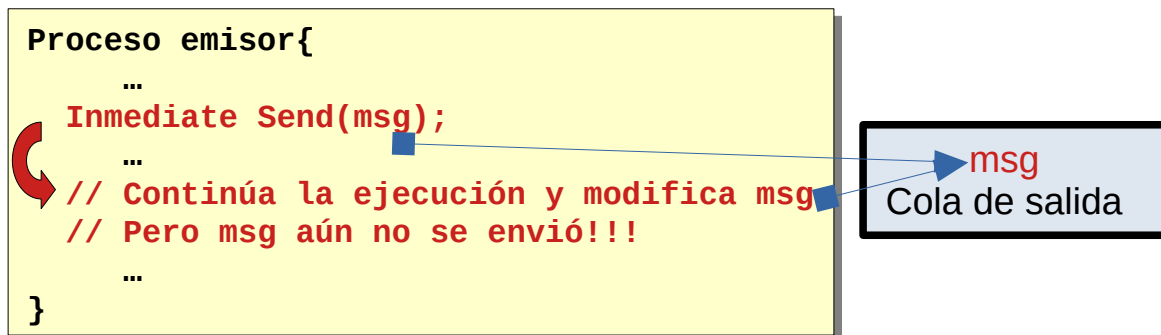
- Si hay un receive: el emisor continua.
- Si no hay un receive: **MPI_Rsend** retorna un **error**.

Comunicación punto a punto – Modo Inmediato

- El modo I (immediate) es "No bloqueante" .
- No bloqueante se refiere a que se retorna el control al programa del emisor, este continúa la ejecución y el **buffer de envío** está disponible para reusarlo.

IMPORTANTE!!!

Podría modificarse un mensaje que aún no fue enviado !!!



- **MPI_Isend**: immediate send (NO bloqueante).

```
int MPI_Isend(const void *buffer,
              int count,
              MPI_Datatype dtype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *req)
```

- El proceso que inicia el Isend continúa su ejecución antes que el mensaje sea almacenado en la cola de salida local (del emisor).
- Un send no bloqueante inicia el envío pero no lo completa.
- El campo **request** almacena el estado de la operación no bloqueante. Se utiliza para poder consultar si la operación ha finalizado.

- **MPI_Irecv**: immediate receive (NO bloqueante).

```
int MPI_Irecv(const void *buf,
              int count,
              MPI_Datatype dtype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *req)
```

- Bloquea el proceso que hace el receive hasta que sea **notificado** de la llegada del mensaje.
- No quiere decir que el mensaje este completo, sólo que hay un mensaje.
- Luego, puede usarse el campo **request** para comprobar si el mensaje fue recibido completamente.



Comunicación punto a punto – MPI_Test / MPI_Wait

- Una vez ejecutada una operación inmediata (Isend o Irecv) hay dos operaciones **para verificar el campo request**:

- **MPI_Test**: chequea si la comunicación finalizó.

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

- Si la comunicación finalizó: **flag = 1**
- Si la comunicación no finalizó: **flag = 0**

- **MPI_Wait**: espera hasta que la comunicación finalice.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- En ambos casos el campo **status** retorna información de la comunicación.

- El modo "I immediate" puede combinarse con los otros modos:
 - IB send
 - IS send
 - IR send
- En todos los casos el emisor continua inmediatamente.
- El **Runtime System** de **MPI** es quien actúa en nombre del emisor para gestionar la comunicación.



- MPI_IBsend:

```
int MPI_IBsend(const void *buffer,
               int count,
               MPI_Datatype dtype,
               int dest, int tag,
               MPI_Comm comm,
               MPI_Request *req)
```

- El emisor continua inmediatamente, luego el **Runtime System de MPI** se comporta como en MPI_Bsend:
 - **Si hay un receive:** el mensaje se envía.
 - **Si no hay un receive:** entonces el mensaje se almacena en una cola de salida local (del emisor) hasta que el **Runtime System de MPI** pueda enviarlo.
- Se requiere un **MPI_Wait** o **MPI_Test** para verificar el resultado de la operación.

- **MPI_Isend:**

```
int MPI_Isend(const void *buffer,
              int count,
              MPI_Datatype dtype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *req)
```

- El emisor continua inmediatamente, luego el **Runtime System de MPI** se comporta como en MPI_Ssend:
 - **Si hay un receive:** la comunicación será exitosa.
 - **Si no hay un receive:** el **Runtime System de MPI** espera.
- Se requiere un **MPI_Wait** o **MPI_Test** para verificar el resultado de la operación.

- MPI_IRequest:

```
int MPI_IRequest(const void *buffer,
                int count,
                MPI_Datatype dtype,
                int dest,
                int tag,
                MPI_Comm comm,
                MPI_Request *req)
```

- El emisor continua inmediatamente, luego el **Runtime System de MPI** se comporta como MPI_Rsend:
 - Si hay un receive: la comunicación será exitosa.
 - Si no hay un receive: MPI_IRequest retorna un error.
- Se requiere un MPI_Wait o MPI_Test para verificar el resultado de la operación.

MPI_Send	El emisor no continua hasta que el mensaje pueda modificarse de forma segura. Se almacena en una cola del receptor o en una cola del emisor.
MPI_Bsend	Si no hay un receive almacena el mensaje en una cola. Luego el emisor continua.
MPI_Ssend	El emisor no continua hasta que no haya un receive.
MPI_Rsend	El emisor continua solo si hay un receive sino retorna un error.
MPI_Isend	El emisor continua inmediatamente.
MPI_IBsend	Igual a MPI_Bsend pero el emisor continua inmediatamente.
MPI_ISsend	Igual a MPI_Ssend pero el emisor continua inmediatamente.
MPI_IRsend	Igual a MPI_Rsend pero el emisor continua inmediatamente.

Comunicación punto a punto - Comprobaciones

- MPI permite comprobar si a un proceso le enviaron un mensaje sin necesidad de recibirlo:

- **MPI_Probe:** comprobación bloqueante. Devuelve el control al programa si existe algún mensaje.

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- **MPI_Iprobe:** comprobación no bloqueante. Devuelve el control al programa inmediatamente.

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
```

- La comprobación se verifica con el argumento **flag**:

- **flag = 0:** no hay mensajes.
- **flag = 1:** hay al menos un mensaje.

- En ambos casos se utiliza el campo **status** para obtener la información del emisor.

- Las comprobaciones `probe` e `iprobe` también suelen ser útiles para recibir mensajes de los cuales se desconocen sus longitudes.
- En ambos casos se utiliza el campo **status** y la función **MPI_Get_count** para obtener la información del emisor.

```
...
MPI_Status status_P;
MPI_Status status;
int number_amount;
int* message;

...
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status_P)
MPI_Get_count(&status_P, MPI_INT, &number_amount);

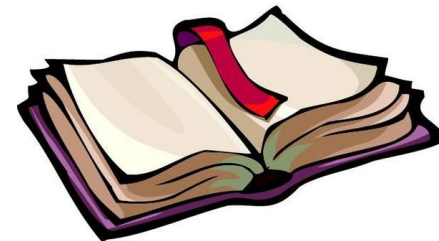
MPI_Recv(message, number_amount, MPI_INT, status_P.MPI_SOURCE, status_P.MPI_TAG, MPI_COMM_WORLD, &status);
...
```

I. Modelo de programación sobre memoria distribuida: Introducción

II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

- i. Funcionamiento, compilación y ejecución
- ii. Estructura de programa
- iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
- iv. Ocultamiento de la latencia

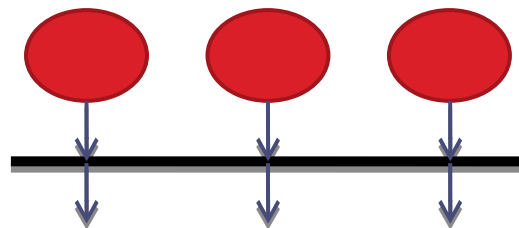


- Las operaciones colectivas son **bloqueantes** (aunque existen los modos l) pero **no** necesariamente **sincrónicas**.
- La finalización de una operación colectiva indica que el proceso que invocó a la función es libre de modificar el búffer de comunicación. No indica que otros procesos del grupo hayan completado o incluso iniciado la operación (a menos que la descripción de la operación indique lo contrario).
- Por lo tanto, una operación de comunicación colectiva puede, o no, tener el efecto de sincronizar todos los procesos MPI participantes.

- **MPI_Barrier:** implementa una barrera distribuida.

```
int MPI_Barrier(MPI_Comm comm)
```

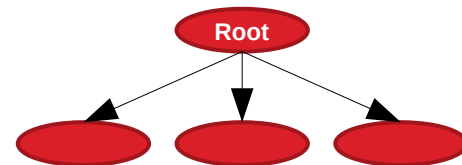
- Cada proceso se bloquea hasta que **MPI_Barrier** sea ejecutada por todos los procesos pertenecientes al comunicador especificado.



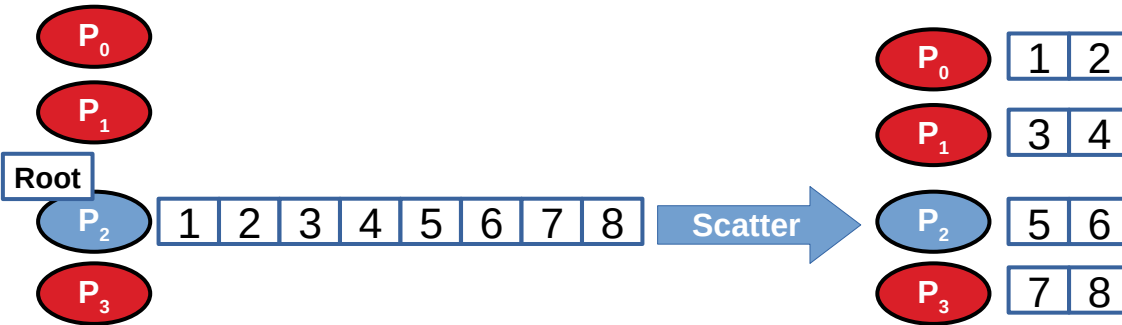
- **MPI_Bcast:**

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype dtype, int root, MPI_Comm comm)
```

- Envía un mensaje desde un proceso origen (**root**) al resto de los procesos del comunicador (**comm**).



- **MPI_Scatter:** Un proceso (**root**) distribuye un conjunto de datos entre los procesos (incluyéndose a si mismo) de un comunicador. Esta distribución se hace “proporcionalmente” y según el rank.



```
int MPI_Scatter(void *sendbuf,  
               int sendcount,  
               MPI_Datatype sendtype,  
               void *recvbuf,  
               int recvcount,  
               MPI_Datatype recvtype,  
               int rootRank,  
               MPI_Comm comm)
```

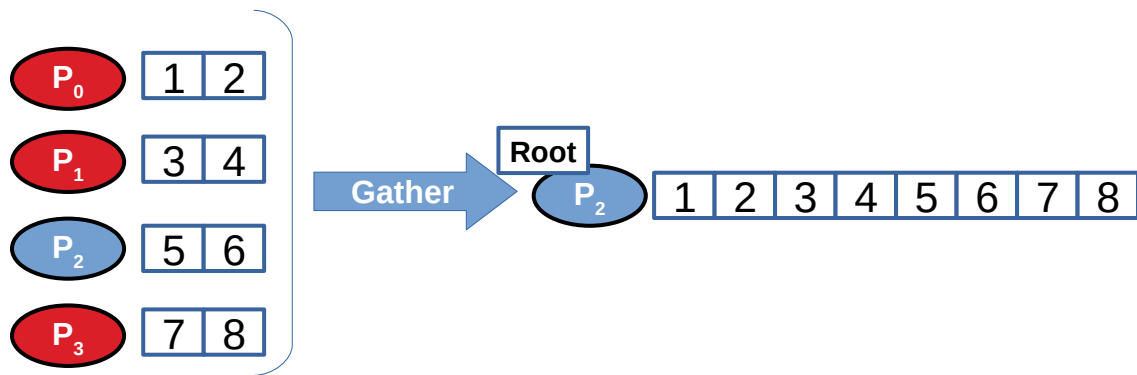
Algunos Parámetros:

- **sendbuf:** Mensaje con los datos a enviar.
- **sendcount:** Cantidad de elementos enviados a cada proceso
- **sendtype:** Tipo de datos de esos elementos.
- **recvbuf:** buffer donde cada proceso recibe su parte de los datos.
- **recvcount:** cantidad de elementos que espera recibir.

*sólo tienen sentido para el proceso **root***



- **MPI_Gather:** inversa de MPI_Scatter. Cada proceso (incluyendo a root) envía los datos al proceso **root**.
- El proceso **root** organiza los datos recibidos según el rank de los procesos.



```
int MPI_Gather(void *sendbuf,  
              int sendcount,  
              MPI_Datatype sendtype,  
              void *recvbuf,  
              int recvcount,  
              MPI_Datatype recvtype,  
              int rootRank,  
              MPI_Comm comm)
```

Algunos Parámetros:

- **recvbuf:** buffer donde el proceso root recibe los datos completos. Los datos se almacenan en orden de los ranks o ids de los procesos.
- **recvcount:** cantidad de elementos que espera recibir de cada proceso.

*sólo tienen sentido para el proceso **root***



Comunicación colectiva – Ejemplo Scatter/Gather

```
int main( int argc, char *argv[]){
    char *message;
    char *part;
    int ID;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &ID );
    MPI_Comm_size(MPI_COMM_WORLD,&nProcs);

    if (ID == 0)  message = (char*)malloc(sizeof(char)*N); //Sólo root aloca

    part = (char*)malloc(sizeof(char)*N/nProcs); //Todo proceso (root inclusive) aloca su parte

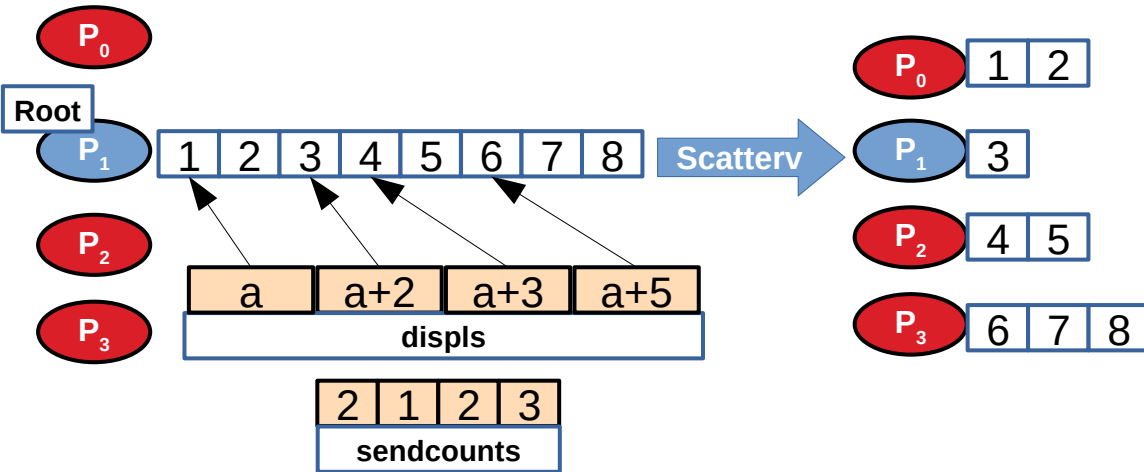
    MPI_Scatter(message, N/nProcs, MPI_CHAR, part, N/nProcs, MPI_CHAR, 0, MPI_COMM_WORLD);

    //Todos los procesos trabajan en esta parte del código con los datos recibidos en part

    MPI_Gather(part, N/nProcs, MPI_CHAR, message, N/nProcs, MPI_CHAR, 0, MPI_COMM_WORLD);
    ...

    MPI_Finalize();
    return 0;
}
```

- **MPI_Scatterv**: es una **variante de MPI_Scatter** que permite distribuir un conjunto de datos en partes de tamaño variable.

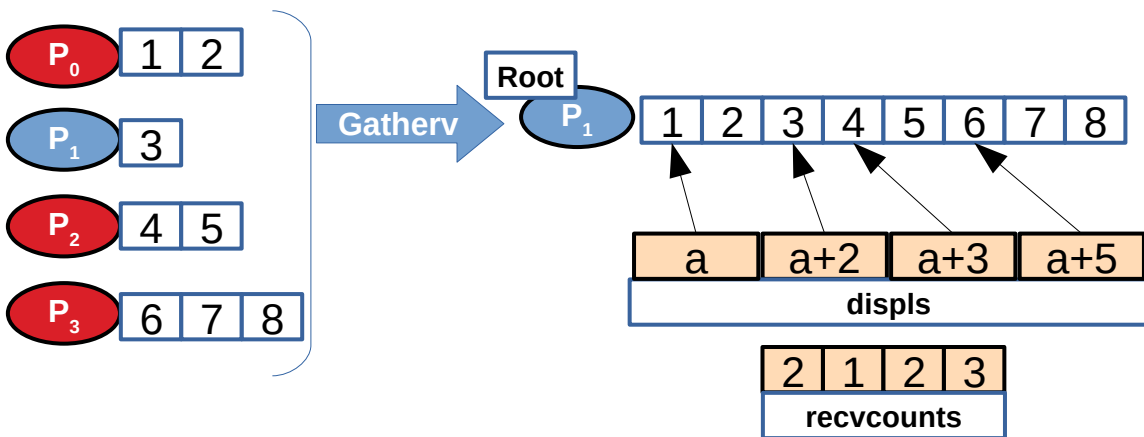


```
int MPI_Scatterv(const void *sendbuf,
                 const int *sendcounts,
                 const int *displs,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int rootRank,
                 MPI_Comm comm)
```

Algunos Parámetros:

- **sendcounts**: ahora es un arreglo que indica en la entrada **i-esima** la cantidad de datos que se envían al proceso con **rank i**.
- **displs**: es un arreglo que indica en la entrada **i-esima** el desplazamiento relativo a sendbuf desde el cual el proceso con **rank i** tomará **sendcounts[i]** datos.

- **MPI_Gatherv**: es una **variante de MPI_Gather** que permite recolectar conjuntos de datos de tamaño variable.



```
int MPI_Gatherv(const void *sendbuf,  
               const int *sendcounts,  
               MPI_Datatype sendtype,  
               void *recvbuf,  
               int *recvcounts,  
               const int *displs,  
               MPI_Datatype recvtype,  
               int rootRank,  
               MPI_Comm comm)
```

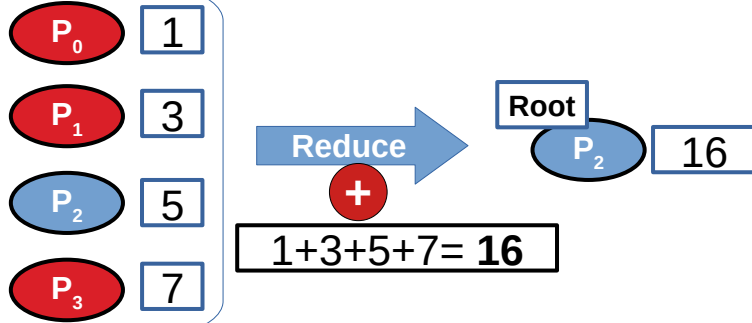
Algunos Parámetros:

- **recvcounts**: ahora es un arreglo que indica en la entrada **i-esima** la cantidad de datos que se reciben del proceso con **rank i**.
- **displs**: es un arreglo que indica en la entrada **i-esima** el desplazamiento relativo a recvbuf desde el cual el proceso con **rank i** recibirán **recvcounts[i]** datos.

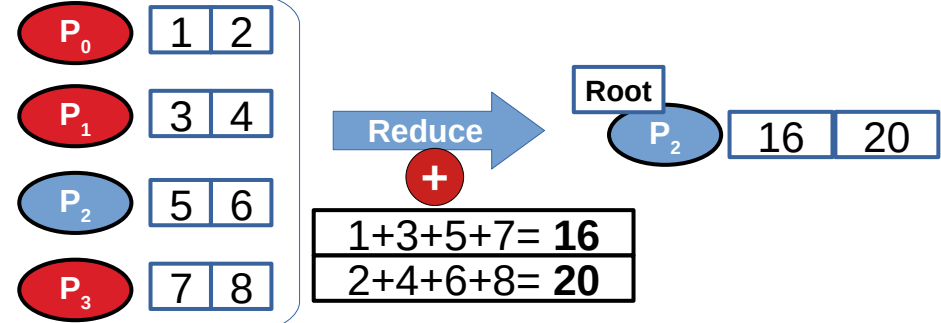
- **MPI_Reduce:** se utiliza cuando cada proceso tiene uno o varios valores y quieren reducirse, mediante alguna operación, a un único valor o un único conjunto de valores. Los valores finales los recibirá un único proceso (**root**)

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int rootRank, MPI_Comm comm)
```

Un único valor



Varios valores



Algunos Parámetros:

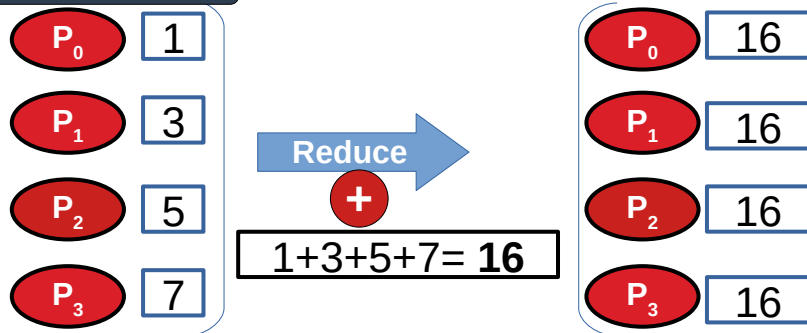
- **sendbuf:** valores a enviar para la reducción.
- **recvbuf:** (sólo tienen sentido para el proceso **root**) buffer donde se reciben los valores finales. 🙌
- **op:** es la operación a realizar sobre los datos recibidos.

- **MPI_MAX**: Máximo entre los elementos
- **MPI_MIN**: Mínimo entre los elementos
- **MPI_SUM**: Suma
- **MPI_PROD**: Producto
- **MPI_LAND**: AND lógico (devuelve 1 o 0, verdadero o falso)
- **MPI_BAND**: AND a nivel de bits
- **MPI_LOR**: OR lógico
- **MPI_BOR**: OR a nivel de bits
- **MPI_LXOR**: XOR lógico
- **MPI_BXOR**: XOR a nivel de bits
- **MPI_MAXLOC**: Valor máximo entre los elementos y el rango del proceso que lo tenía
- **MPI_MINLOC**: Valor mínimo entre los elementos y el rango del proceso que lo tenía

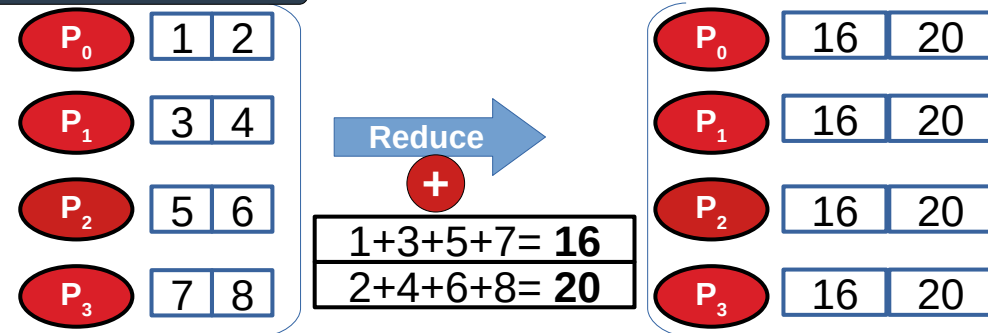
- **MPI_Allreduce: variante de MPI_Reduce** donde los valores finales quedan distribuidos en todos los procesos (combinación de reduction + broadcast).

```
int MPI_Allreduce(const void *sendbuf,  
                 void *recvbuf,  
                 int count,  
                 MPI_Datatype datatype,  
                 MPI_Op op,  
                 MPI_Comm comm)
```

Un único valor



Varios valores

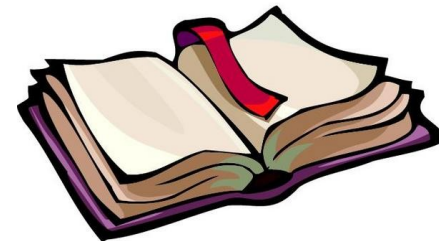





I. Modelo de programación sobre memoria distribuida: Introducción

II. Parallel Virtual Machine (PVM)

III. Message Passing Interface (MPI)

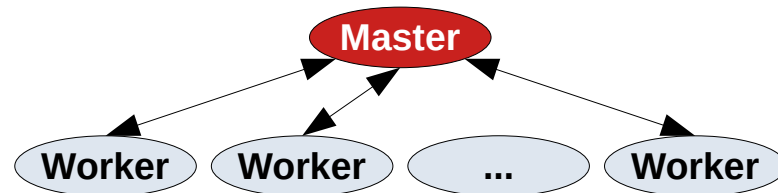
- i. Funcionamiento, compilación y ejecución
- ii. Estructura de programa
- iii. Comunicación
 - i. Operaciones punto a punto
 - ii. Operaciones colectivas
- iv. Ocultamiento de la latencia



- Una situación común es encontrar procesos inactivos esperando recibir un mensaje o esperando para enviar un mensaje (comunicación bloqueante).
- El hecho que los procesos esperen ociosos por mucho tiempo puede afectar el rendimiento. 
-  En ocasiones, puede evitarse esta espera reestructurando los procesos. En particular, un proceso puede “avisar” que quiere recibir un mensaje y luego continuar con el cómputo. Esto permite al programador superponer la comunicación y el cálculo.
- Estas técnicas se conocen como **ocultamiento de la latencia** (Latency hiding) o superposición (Overlapping) entre cómputo y comunicación: 
 - Se utilizan operaciones punto a punto mediante **Sends/Receives no bloqueantes**.
 - Se utilizan técnicas de **prefetching** o carga anticipada: se solicitan datos para el próximo cálculo.

- Ejemplo de un modelo Master-Worker:

- Los Workers piden trabajo al Master.
- El Master se demora un tiempo en preparar el trabajo para cada Worker.
- Los Workers reciben trabajo hasta que el Master les indique que ya no hay trabajo disponible.



Master

```
while hayTrabajo
    "Prepara trabajo para enviar"
    Receive(pedidoTrabajo,w)
    Send(trabajo,w)
end while

for worker = 1 a numWorkers
    Receive(pedidoTrabajo,w)
    Send("NoTrabajo",w)
end for
```

End master

Sin ocultamiento



```
Worker [w:1..numWorkers]
hayTrabajo=true
```

```
Send(pedidoTrabajo,master)
Receive(trabajo,master)
while hayTrabajo(trabajo)
    "Trabaja"
    Send(pedidoTrabajo,master)
    Receive(trabajo,master)
end while
```

End worker

- Podemos ocultar la latencia de comunicación en los workers utilizando una combinación de operaciones no bloqueantes y prefetching.
 - El Master no cambia 🤔
 - Los Workers piden trabajo al Master. Una vez que lo reciben, piden trabajo para la próxima iteración mediante operaciones no bloqueantes.

Master

```

while hayTrabajo
    "Prepara trabajo para enviar"
    Receive(pedidoTrabajo,w)
    Send(trabajo,w)
end while

for worker = 1 a numWorkers
    Receive(pedidoTrabajo,w)
    Send("NoTrabajo",w)
end for

```

End master



Con ocultamiento y prefetching

Worker [w:1..numWorkers]

```

Send(pedidoTrabajo,Master)
Receive(trabajo,Master)
while hayTrabajo(trabajo)
    ISend(pedidoTrabajo2,Master)
    IReceive(trabajo2,Master)
    "Trabaja sobre trabajo"
    Wait(IReceive)
    swap(trabajo,trabajo2)
end while

```

End Worker

Mientras el worker trabaja se va recibiendo (IReceive) trabajo para la próxima iteración.

Para continuar se debe esperar que Ireceive se complete.

Se intercambian los punteros a los buffers.

