SISTEMAS DISTRIBUIDOS Y PARALELOS

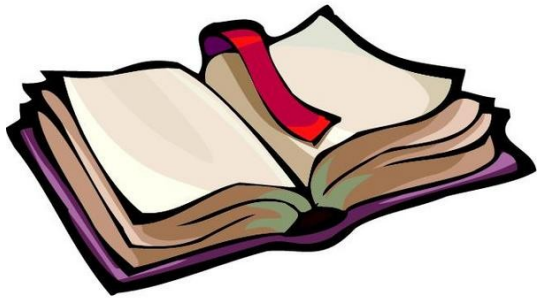
Carrera: Ingeniería en computación
Facultad de Informática – Universidad Nacional de La Plata

Dr. Adrián Pousa

Modelo de programación sobre memoria distribuida

Agenda

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*



Agenda

I. ***Modelo de programación sobre memoria distribuida: Introducción***

II. *Parallel Virtual Machine (PVM)*

III. *Message Passing Interface (MPI)*

   I. *Funcionamiento, compilación y ejecución*

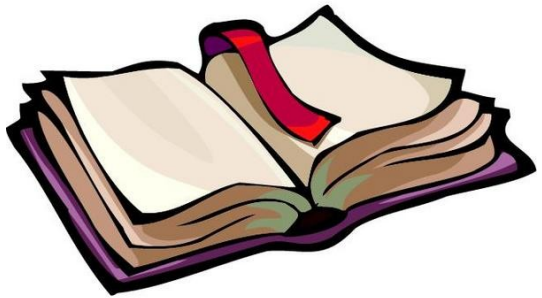
   II. *Estructura de programa*

   III. *Comunicación*

      I. *Operaciones punto a punto*

      II. *Operaciones colectivas*

   IV. *Ocultamiento de la latencia*



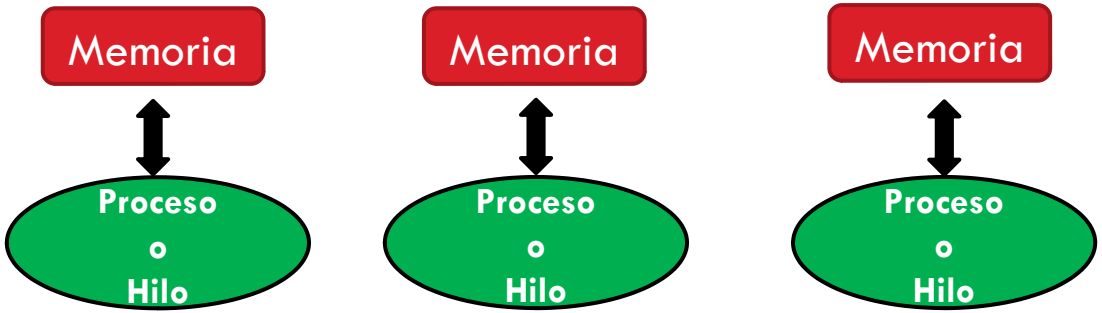
Modelo de programación

- Paralelismo Implícito

- Paralelismo Explicito:
  - Modelo de programación sobre memoria compartida
  - Modelo de programación sobre memoria distribuida
  - Modelos híbridos

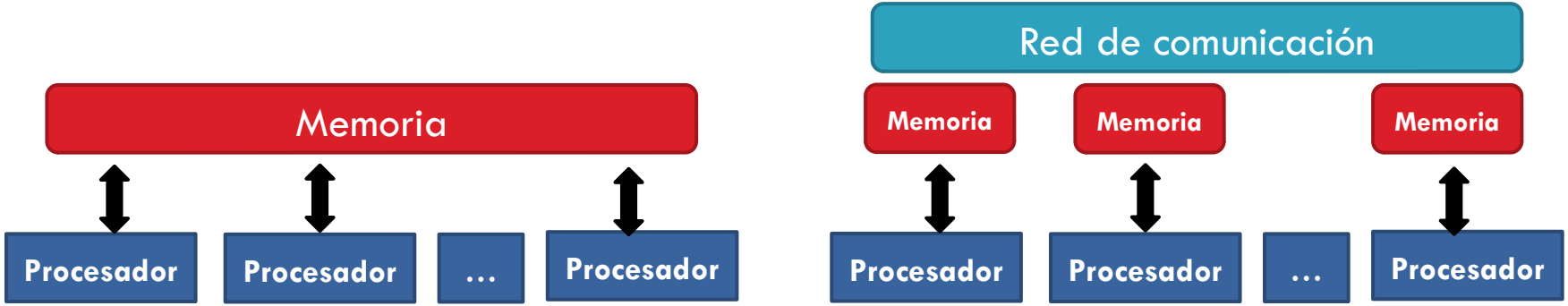
Modelo de programación sobre memoria distribuida

- 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 envío y recepción de mensajes.



Modelo de programación sobre memoria distribuida

- Los procesos se pueden ejecutar tanto en arquitecturas de memoria compartida como en arquitecturas de memoria distribuida.
- En arquitecturas de memoria distribuida los procesos se comunican por mensajes a través de una red de comunicación física.
- En arquitecturas de memoria compartida la red es virtual.



Modelo de programación sobre memoria distribuida

- Las herramientas más utilizadas que siguen el modelo de programación sobre memoria distribuida son:
  - PVM
  - MPI

Agenda

I. *Modelo de programación sobre memoria distribuida: Introducción*

II. ***Parallel Virtual Machine (PVM)***

III. *Message Passing Interface (MPI)*

   I. *Funcionamiento, compilación y ejecución*

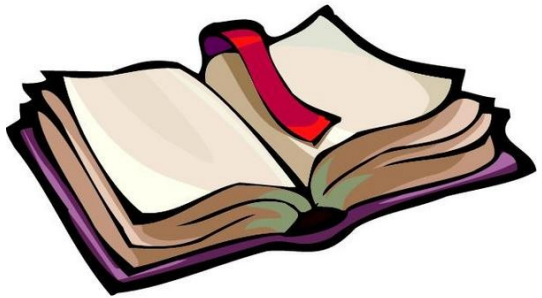
   II. *Estructura de programa*

   III. *Comunicación*

      I. *Operaciones punto a punto*

      II. *Operaciones colectivas*

   IV. *Ocultamiento de la latencia*



PVM

- **PVM** (Parallel Virtual Machine) fue desarrollada por la Universidad de Tennessee
- La primera versión fue escrita en ORNL en 1989
- La versión 2 escrita para C, C++ y Fortran en marzo de 1991
- La versión 3 fue lanzada en marzo de 1993 con mejoras en la tolerancia a fallas y portabilidad
- Última versión 2009
- Fue la herramienta que impulsó el uso de clusters

Agenda

I. *Modelo de programación sobre memoria distribuida: Introducción*

II. *Parallel Virtual Machine (PVM)*

III. ***Message Passing Interface (MPI)***

   I. ***Funcionamiento, compilación y ejecución***

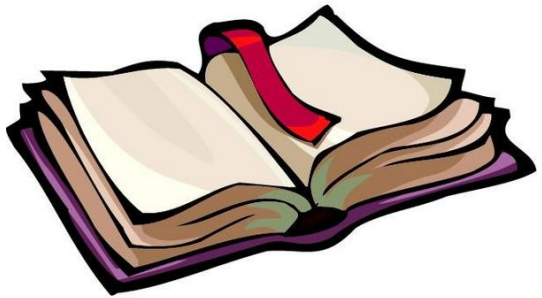
   II. ***Estructura de programa***

   III. ***Comunicación***

      I. ***Operaciones punto a punto***

      II. ***Operaciones colectivas***

   IV. ***Ocultamiento de la latencia***



Agenda

I. *Modelo de programación sobre memoria distribuida: Introducción*

II. *Parallel Virtual Machine (PVM)*

III. ***Message Passing Interface (MPI)***

   I. ***Funcionamiento, compilación y ejecución***

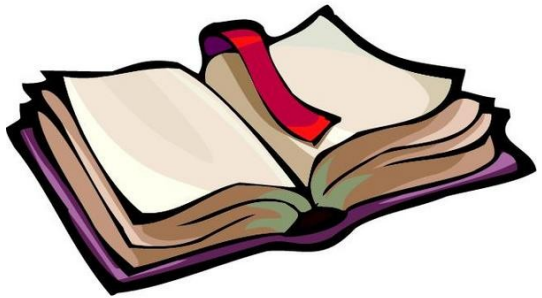
   II. *Estructura de programa*

   III. *Comunicación*

      I. *Operaciones punto a punto*

      II. *Operaciones colectivas*

   IV. *Ocultamiento de la latencia*

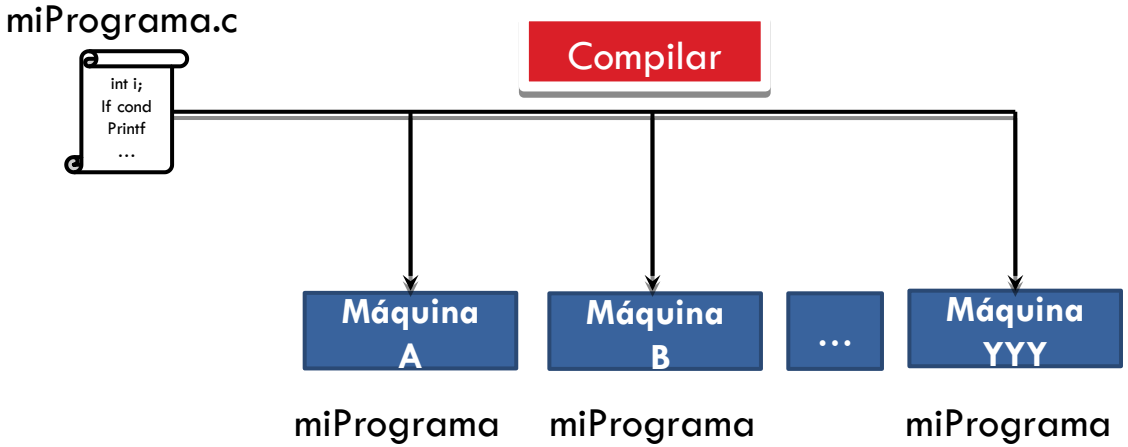


MPI

- **M**essage **P**assing **I**nterface (MPI) es un estándar que define la sintaxis y la semántica de funciones para pasaje de mensajes
- Estándar: http://www.mpi-forum.org
- Existen varias implementaciones:
  - OpenMPI, Mpich, LamMPI Etc.
  - Para lenguajes: C, C++ y Fortran.
  - Existen para otros lenguajes (Java o Python) pero son wrappers bajo lenguaje C.

MPI – Funcionamiento

- Cada máquina debe conocer el archivo ejecutable (binario)
- MPI no compila automáticamente el archivo fuente ni lo distribuye
- La responsabilidad es del programador



MPI – Compilar un programa MPI

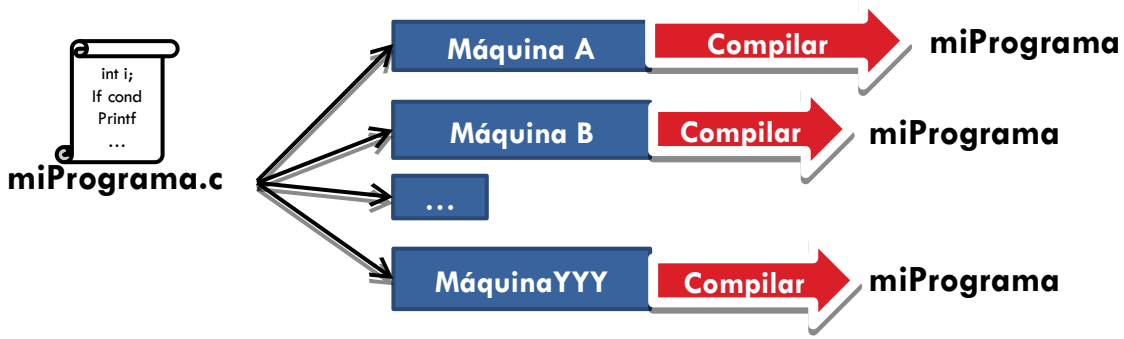
mpicc -o miPrograma miPrograma.c

- **Cluster homogéneo:** Alcanza con generar el binario compilando el código en una de las máquinas del cluster
- Luego, el binario deben conocerlo todas las máquinas del cluster. Dos opciones:
  - Copiar el binario en cada máquina
  - Compartir el binario mediante sistemas de archivos distribuidos (ej: NFS)

MPI – Compilar un programa MPI

mpicc -o miPrograma miPrograma.c

- **Cluster heterogéneo:** se debe compilar el código en cada una de las máquinas
- No se puede compartir el binario porque son incompatibles



MPI – Ejecución de un programa MPI

mpirun -np **NrProcesos** –machinefile **maquinas** miPrograma

- NrProcesos: es la cantidad de procesos a crear
- El archivo "maquinas" contiene el nombre de las máquinas a utilizar. Su composición varía dependiendo de la distribución de MPI
- Archivo máquinas en OpenMPI:

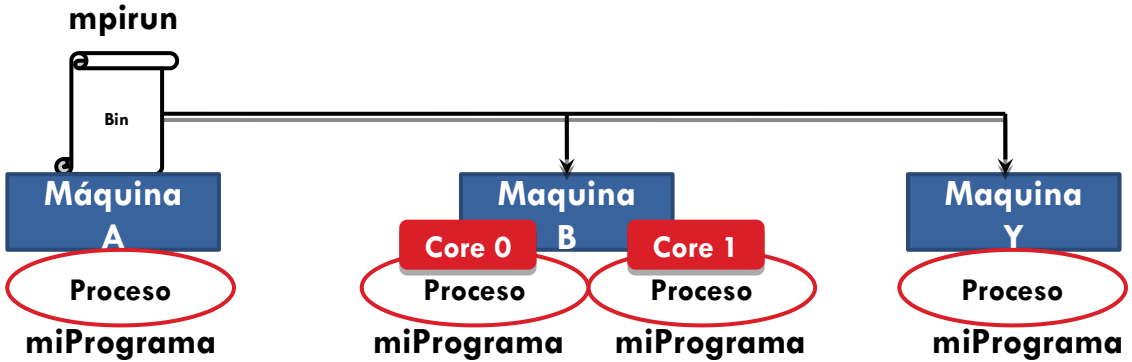
```
maquinaA slots=1
maquinaB slots=2
maquinaY slots=1
```

slots indica la cantidad de procesos que se le enviarán a esa máquina

MPI – Ejecución de un programa MPI

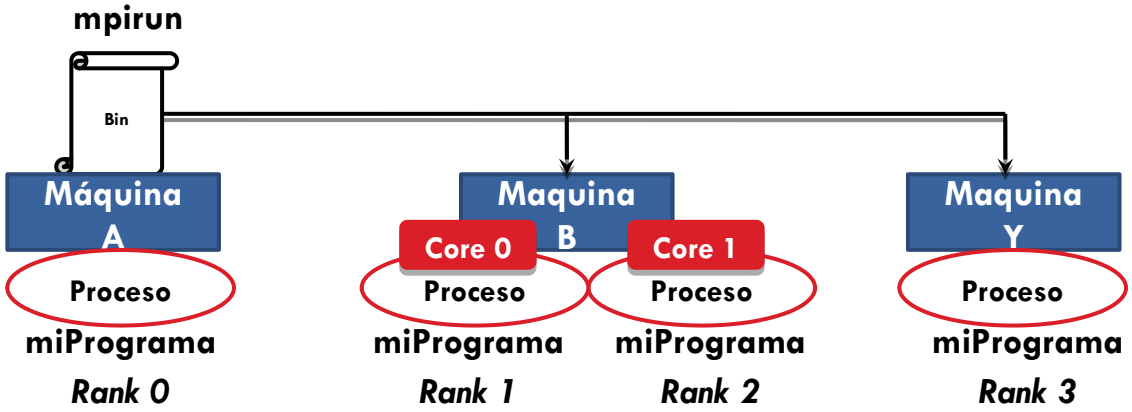
mpirun -np **NrProcesos** –machinefile **maquinas** miPrograma

- El comando se corre en una máquina y MPI automáticamente desencadena la ejecución de una copia del programa en cada unidad de procesamiento (procesador/core)
- Todas las unidades de procesamiento ejecutan el mismo programa



MPI – Ejecución de un programa MPI

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



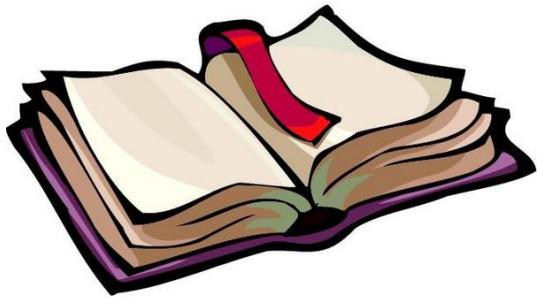
Agenda

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 – Estructura típica de programa MPI en C


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

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&miID);
 MPI_Comm_size(MPI_COMM_WORLD,&cantidadDeProcesos);
 …
 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
    funcionProcesoTipoC();  // Función que implementa los procesos de tipo C
 MPI_Finalize();
  return(0);
}
```

MPI

- Todas las funciones MPI llevan el prefijo **MPI_** delante.
- Las cuatro funciones en la estructura de programa anterior son funciones básicas de control:

  - **MPI_init**: inicializa el ambiente de ejecución MPI, recibe como parámetros los argumentos que recibió el programa C.
  - **MPI_Comm_rank**: permite obtener el identificador de proceso (0..P).
  - **MPI_Comm_size**: permite obtener la cantidad de procesos creados.
  - **MPI_Finalize**: termina la ejecución del ambiente MPI.

MPI - Comunicadores

- Algunas de las funciones reciben como parámetro el nombre de un comunicador **MPI_COMM_WORLD**.
- Un comunicador agrupa procesos que pueden comunicarse entre si.
- MPI permite crear comunicadores y ofrece distintas ventajas:
  - Organizar sus procesos y armar topologías virtuales.
  - Permite la comunicación entre un grupo de procesos.
- **MPI_COMM_WORLD** es el comunicador por defecto que incluye a todos los procesos en la ejecución.

Agenda

I. *Modelo de programación sobre memoria distribuida: Introducción*

II. *Parallel Virtual Machine (PVM)*

III. ***Message Passing Interface (MPI)***

   I. *Funcionamiento, compilación y ejecución*

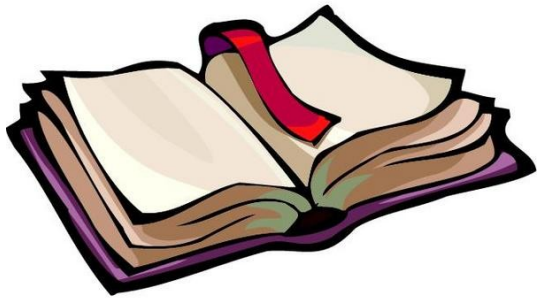
   II. *Estructura de programa*

   III. ***Comunicación***

      I. ***Operaciones punto a punto***

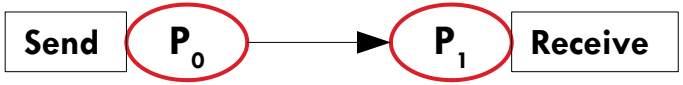
      II. ***Operaciones colectivas***

   IV. *Ocultamiento de la latencia*

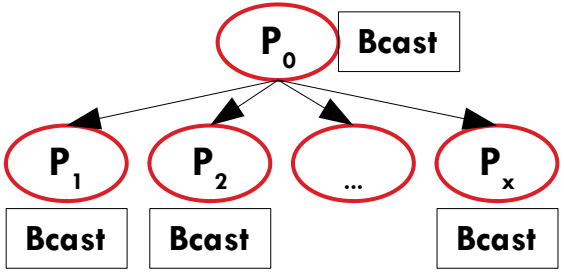


MPI – Comunicación entre procesos

- 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 al mismo tiempo.



IMPORTANTE Colectivas
Todos los procesos deben ejecutar la **misma función de comunicación.**

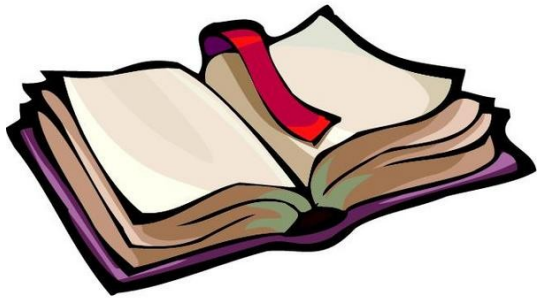
Agenda

I. *Modelo de programación sobre memoria distribuida: Introducción*

II. *Parallel Virtual Machine (PVM)*

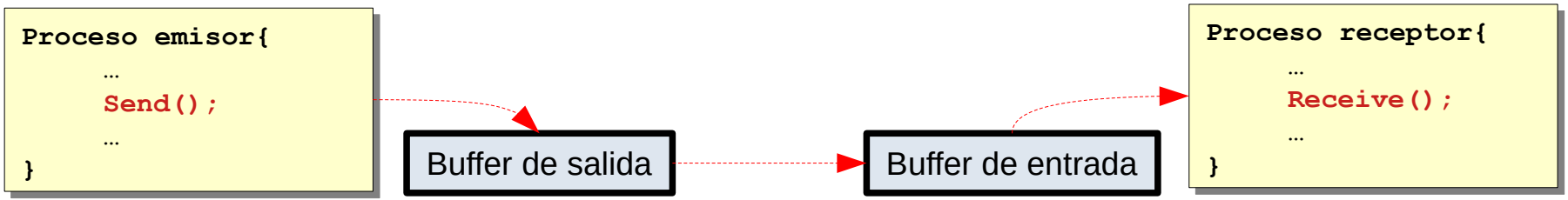
III. ***Message Passing Interface (MPI)***

- 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 punto a punto

- Las operaciones básicas de comunicación punto a punto entre procesos MPI son:
  - Envío de mensajes: MPI_Send
  - Recepción de mensajes: MPI_Recv
- Para comprender la comunicación punto a punto se debe considerar el siguiente diagrama:



- La operación es **bloqueante** si por alguna razón la sentencia (send o receive) no retorna el control al programa. Es **No bloqueante** si la sentencia retorna el control inmediatamente sin importar lo que ocurra con el mensaje.
- La operación de envío es **sincrónica** si el programa no puede continuar hasta que no haya una recepción.

MPI – Comunicación punto a punto

- Todas las funciones que envían o reciben mensajes tienen parámetros en común:
  - **Buffer**: representa el mensaje a enviar o recibir
  - **Tipo**: el tipo de datos de los elementos del buffer
  - **Cantidad**: cantidad de elementos del tipo anterior que contiene el buffer
  - **Comunicador**: el comunicador que agrupa a los procesos comunicándose
  - **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 – Tipos MPI

- MPI define palabras clave para cada tipo de dato pasado como argumento en las funciones:

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

MPI – Comunicación punto a punto - MPI_Send

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

```
int MPI_Send( const void *buf,
              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 un buffer del receptor o en un buffer temporal (del emisor).
  - En **source** se debe indicar el ID del receptor.

MPI – Comunicación punto a punto - MPI_Recv

- **MPI_Recv**: es una recepción bloqueante.

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

  - **source** indica el ID del emisor.
  - Si se desconoce el ID emisor se utiliza **MPI_ANY_SOURCE**.
  - Si se desconoce el tag del emisor se utiliza **MPI_ANY_TAG**.
  - 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)

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

MPI – Comunicación punto a punto - Ejemplo


```
#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, &status);
    printf("received :%s:\n", message);
 }
 MPI_Finalize();
 return 0;
}
```

MPI – Comunicación punto a punto - Modos

- MPI provee distintos modos de comunicación punto a punto que se indican con los prefijos:
  - B buffered. (Bloqueante)
  - S synchronous. (Bloqueante)
  - R ready. (Bloqueante)
  - I inmediate. (No bloqueante)

MPI – Comunicación punto a punto - BSend

- B buffered (Bloqueante).

```
int MPI_BSend(const void *buf,
              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 entonces el mensaje debe ser almacenado en un buffer local (del emisor) y luego el emisor continua.
  - Puede ocurrir un error si no hay espacio suficiente en el buffer.
  - La cantidad de espacio disponible en el buffer es controlada por el usuario.

MPI – Comunicación punto a punto - SSend

- S synchronous (Bloqueante).

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

  - El emisor continua sólo si existe un receive.
  - Si no existe un receive el emisor espera.
  - La finalización de un send synchronous indica que el buffer de envío puede ser usado y también que el receptor ha alcanzado cierto punto en su ejecución.

MPI – Comunicación punto a punto - RSend

- R ready (Bloqueante).

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

  - El emisor continua sólo si existe un receive.
  - Si no hay un receive MPI_RSend retorna un error.

MPI – Comunicación punto a punto – Modo inmediato

- El modo I (inmediate) es "No bloqueante" .
- "No bloqueante" se refiere a si el buffer de envío está disponible para reusarlo.
- Al ejecutar una operación en modo inmediato el emisor continua y el buffer de envío se puede reutilizar.

Pero puede modificarse un mensaje que aún no fue enviado !!!

MPI – Comunicación punto a punto - Isend

- I inmediate send (NO bloqueante).

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

  - El proceso que inicia el send continúa su ejecución antes que el mensaje sea almacenado en el buffer 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 – Comunicación punto a punto - MPI_Irecv

- I inmediate 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, se puede usar el campo **request** para comprobar si el mensaje fue recibido completamente.

MPI – Comunicación punto a punto - MPI_Test-MPI_Wait

- Una vez ejecutada una operación inmediate (Isend o Irecv) hay dos opciones para usar el campo **request**:
  - MPI_Test: chequea si la comunicación finalizó.

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

  - MPI_Wait: espera hasta que la comunicación finalice.

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

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

MPI – Comunicación punto a punto – Modo inmediato

- El modo "I inmediate" puede combinarse con los otros modos:
  - IB send.
  - IS send.
  - IR send.
- En todos los casos el emisor continua inmediatamente.

MPI – Comunicación punto a punto - MPI_IBsend

- **MPI_IBsend:**

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

  - El emisor continua inmediatamente, luego MPI se comporta como Bsend:
    - Si hay un receive el mensaje se envía.
    - Si no hay un receive entonces el mensaje debe ser almacenado en un buffer local (del emisor).
  - Se requiere un Wait o Test que se completarán cuando alguna de las dos condiciones anteriores ocurran.

MPI – Comunicación punto a punto - MPI_ISsend

- **MPI_ISsend:**

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

  - El emisor continua inmediatamente, luego MPI se comporta como Ssend:
    - La operación será exitosa si había un receive.
  - Se requiere un Wait o Test que se completarán cuando la condición anterior ocurra.

MPI – Comunicación punto a punto - MPI_IRsend

- **MPI_IRsend:**

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

  - El emisor continua inmediatamente, luego MPI se comporta como Rsend:
    - La operación será exitosa si había un receive sino retorna error.
  - Se requiere un Wait o Test que se completarán cuando la condición anterior sea exitosa o emitirán un error.

MPI – Comunicación punto a punto - Resumen

MPI_Send	El emisor no continua hasta que el mensaje pueda modificarse de forma segura. Se almacena en un buffer del receptor o temporal del emisor (buffer de envío).
MPI_Bsend	Si no hay un receive almacena el mensaje en un buffer. 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.

MPI – Comunicación punto a punto - Comprobaciones

- MPI permite comprobar si existen mensajes.
- **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)
```

  - **flag**: 0 si no existe mensaje, > 1 en caso contrario.

MPI – Comunicación punto a punto - MPI_SendRecv

- **MPI_SendRecv:**

```
int MPI_SendRecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

  - Permite realizar en un solo llamado un send y un receive.
  - Ambos procesos deben ejecutar la misma función.
  - Utilizada para evitar bloqueos en caso de envío y recepción bloqueante.

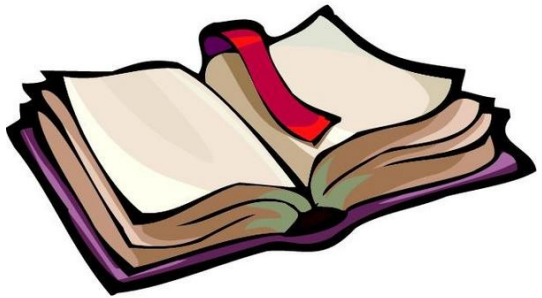
Agenda

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 colectiva - MPI_Barrier

- **MPI_Barrier**: Bloquea al proceso hasta que todos los procesos pertenecientes al comunicador especificado lo ejecuten.

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

- **MPI_Bcast**: Envía un mensaje desde un proceso origen a todos los procesos.

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

  - Todos los procesos deben ejecutar la misma función.
  - El parámetro root indica el source del proceso que envía.

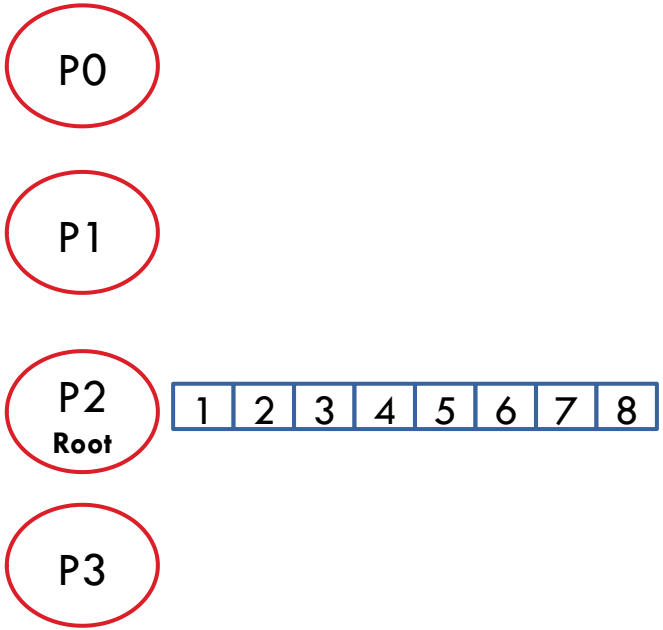
MPI – Comunicación colectiva - MPI_Scatter

- **MPI_Scatter:**

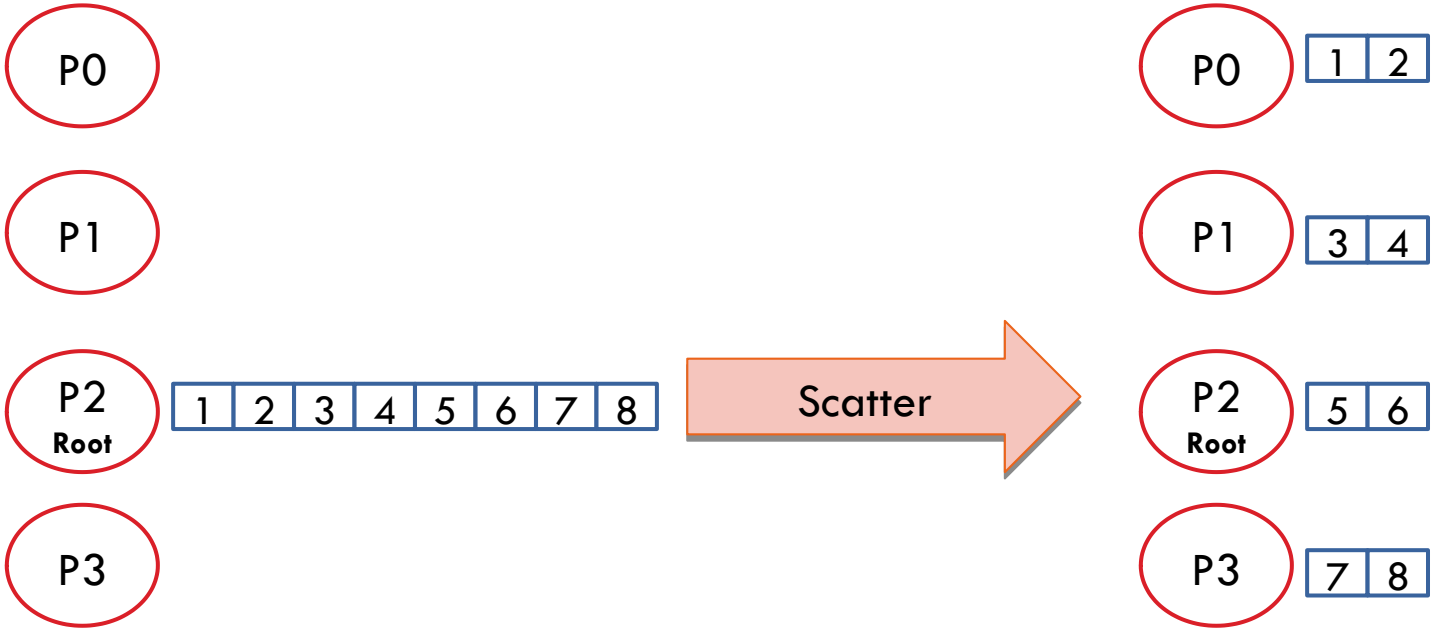
```
int MPI_Scatter(void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm)
```

- Un proceso (**root**) divide un mensaje en partes iguales y las envía al resto de procesos y a sí mismo.
- **sendbuf**: dirección del mensaje de salida.
- **sendcount**: cantidad de elementos enviados a cada proceso.
- **sendbuf, sendcount, sendtype** sólo útiles para **root**.
- **recvcount**: cantidad de elementos que espera recibir.

MPI – Comunicación colectiva - MPI_Scatter



MPI – Comunicación colectiva - MPI_Scatter



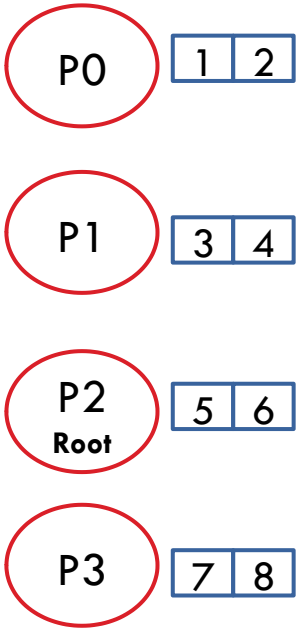
MPI – Comunicación colectiva - MPI_Gather

- **MPI_Gather:**

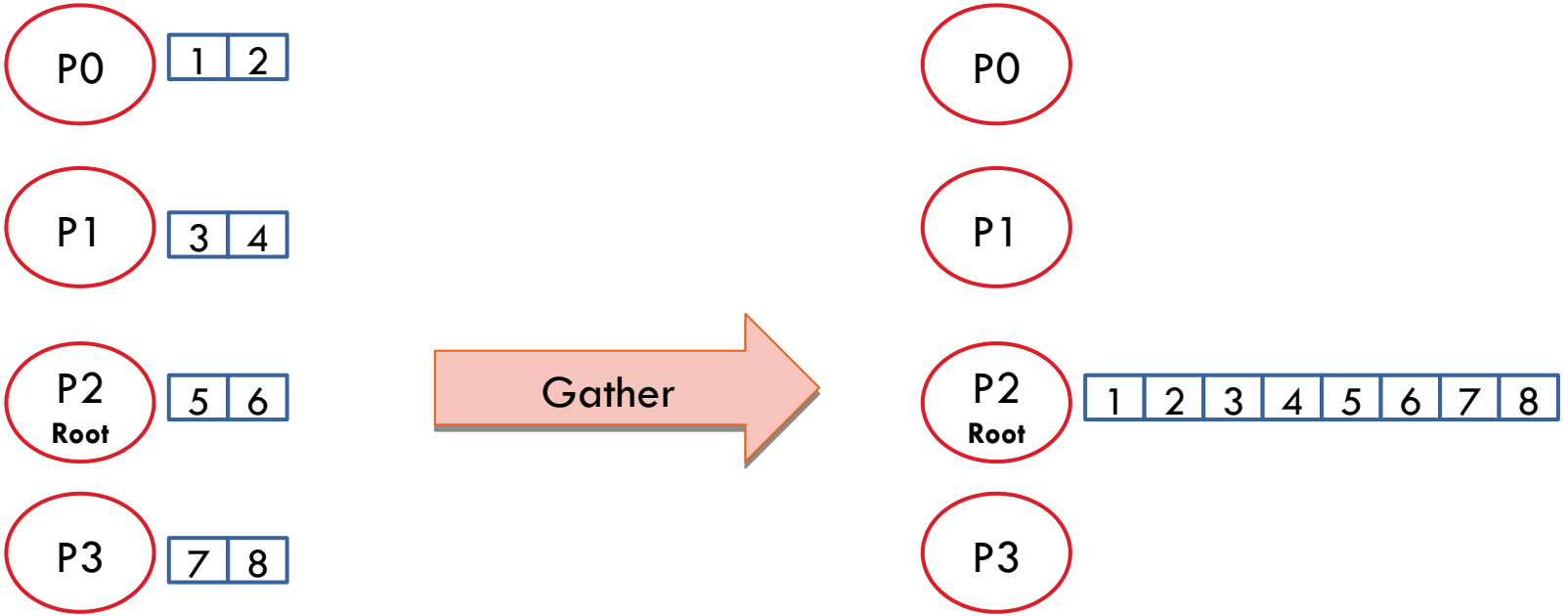
```
int MPI_Gather(void *sendbuf,
               int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf,
               int recvcount,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm)
```

  - Funciona como inversa de **MPI_Scatter**.
  - Cada proceso (incluyendo a **root**) envía los datos al proceso **root**.
  - El proceso **root** recibe los datos y los almacena en orden de los ID de los procesos.
  - **recvcount** (solo útil para **root**): cantidad de elementos que espera recibir de cada proceso.

MPI – Comunicación colectiva - MPI_Gather



MPI – Comunicación colectiva - MPI_Gather



MPI – Comunicación colectiva – Ejemplo Scatter/Gather


```
char *message;
char *part;
int ID;
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(N)*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);

//Trabaja con los datos recibidos en part

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

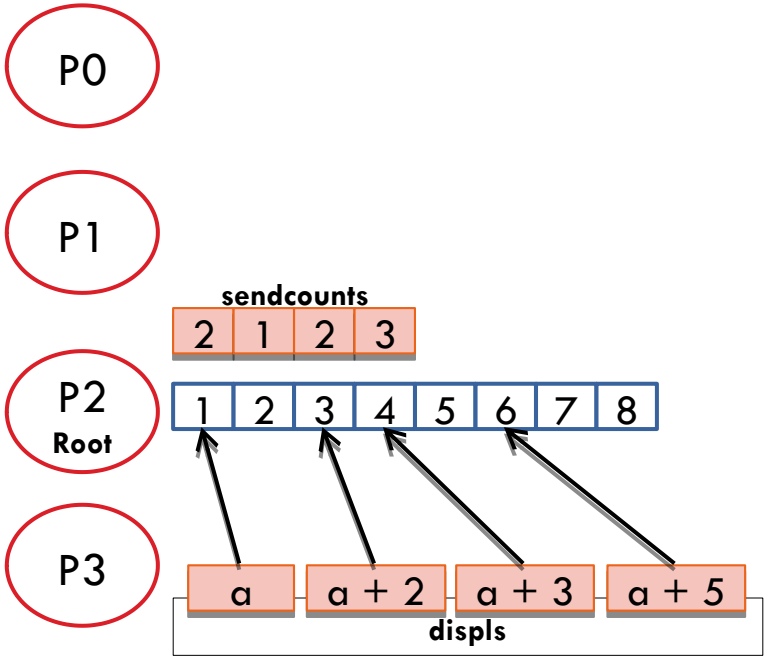
MPI – Comunicación colectiva - MPI_Scatterv

- **MPI_Scatterv**: variante de MPI_Scatter que permite distribuir mensajes de tamaño variable.

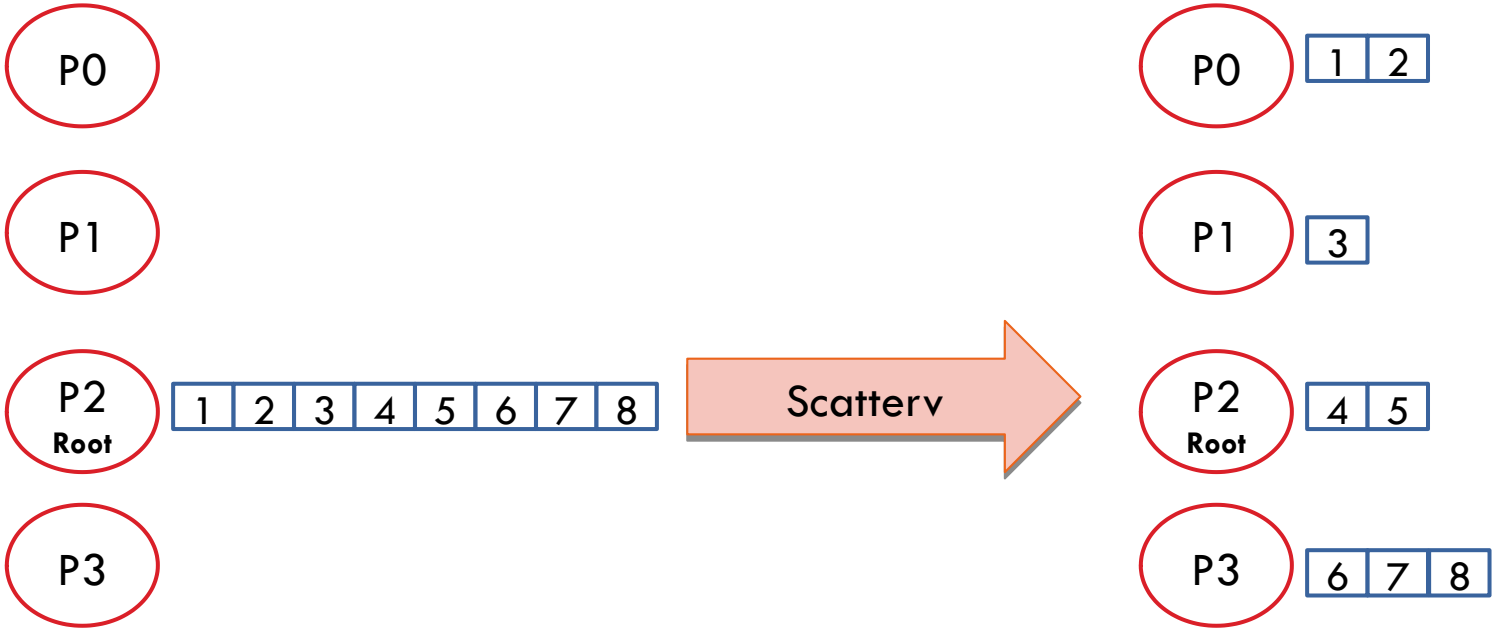
```
int MPI_Scatterv(const void *sendbuf, const int *sendcounts, const
int *displs,  MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

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

MPI – Comunicación colectiva - MPI_Scatterv



MPI – Comunicación colectiva - MPI_Scatterv



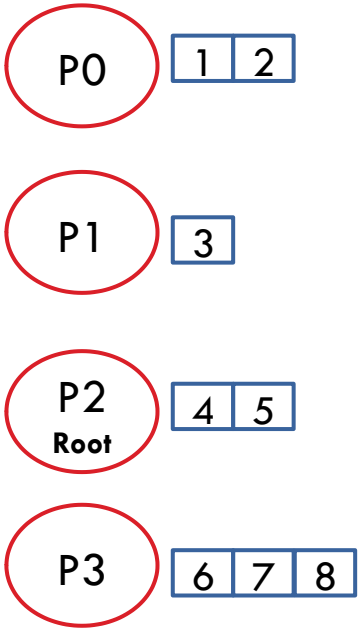
MPI – Comunicación colectiva - MPI_Gatherv

- **MPI_Gatherv**: variante de MPI_Gather que permite recolectar mensajes de tamaño variable.

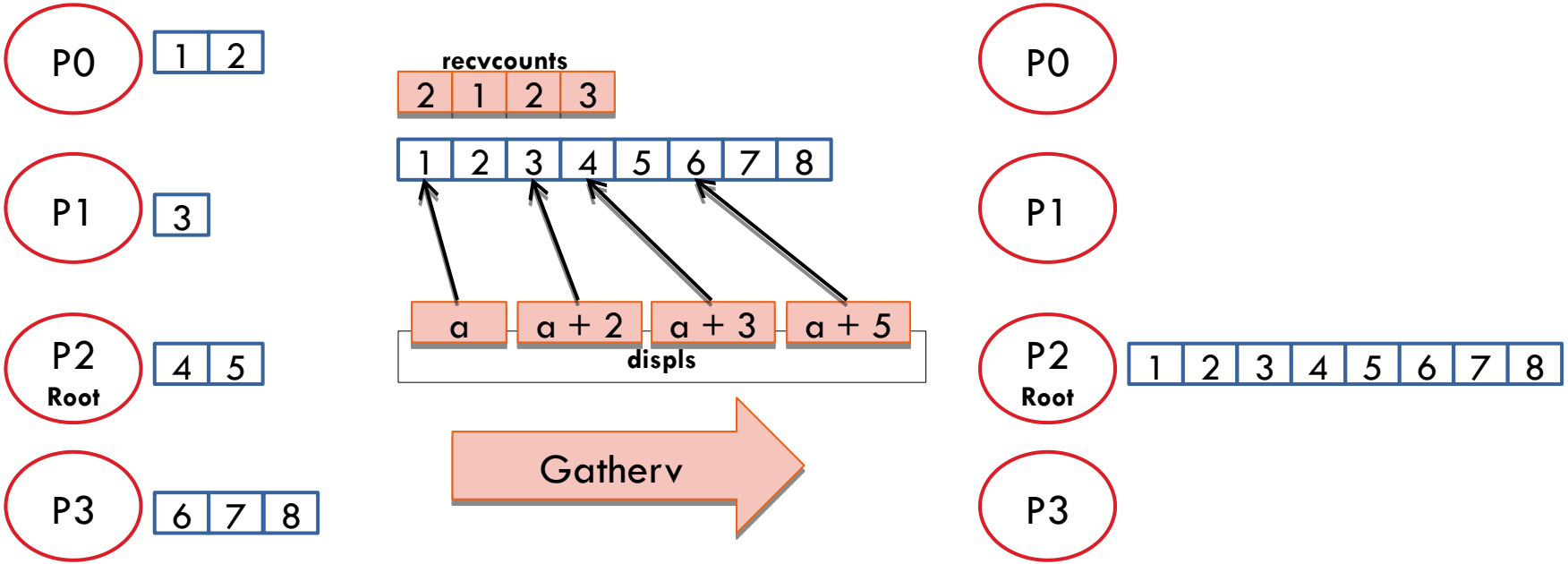
```
int MPI_Gatherv(const void *sendbuf, const int *sendcounts, MPI_Datatype
sendtype, void *recvbuf, int *recvcounts, const int *displs, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

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

MPI – Comunicación colectiva - MPI_Gatherv



MPI – Comunicación colectiva - MPI_Gatherv



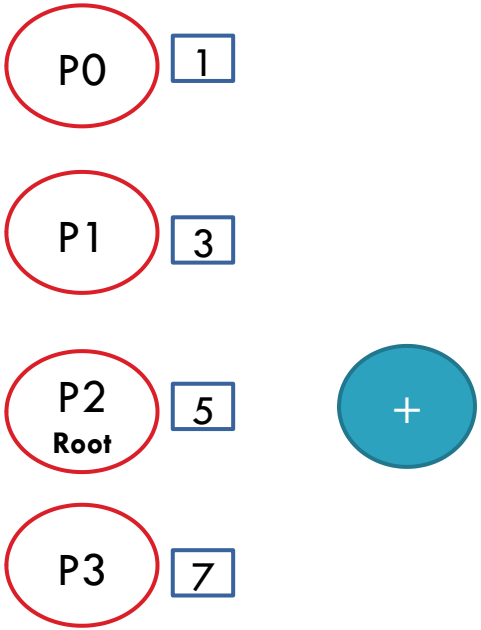
MPI – Comunicación colectiva - MPI_Reduce

- **MPI_Reduce**: reduce un valor que tienen todos los procesos a un único valor mediante una operación.

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

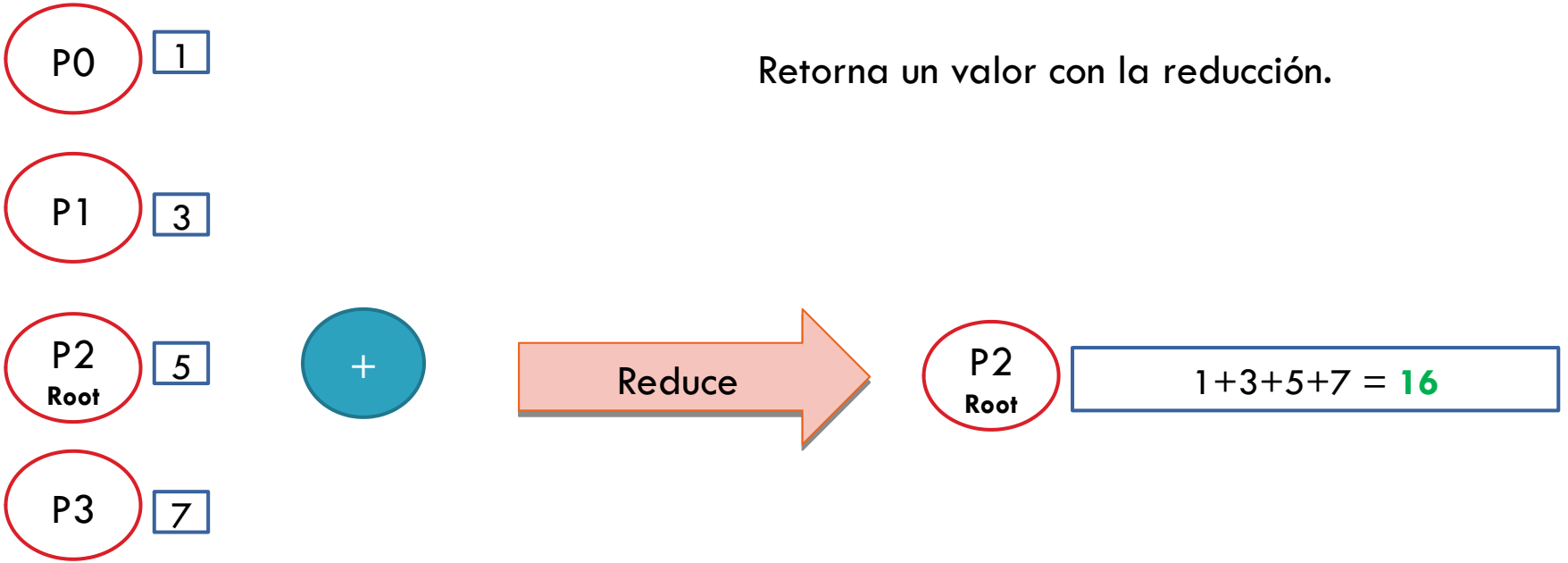
- **sendbuf** son los datos que se van a enviar.
- **recvbuf** es el buffer de recepción que solo tiene sentido para el proceso (**root**).
- **root** indica que proceso va a hacer la operación con los datos recibidos.
- **op** es la operación a realizar sobre los datos recibidos.

MPI – Comunicación colectiva - MPI_Reduce



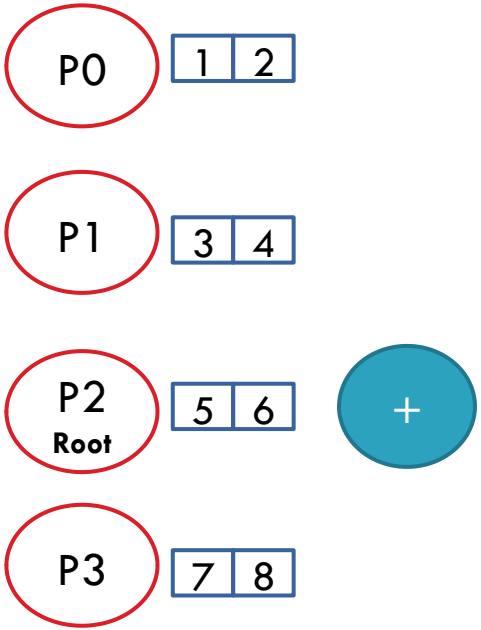
Un elemento por proceso

MPI – Comunicación colectiva - MPI_Reduce



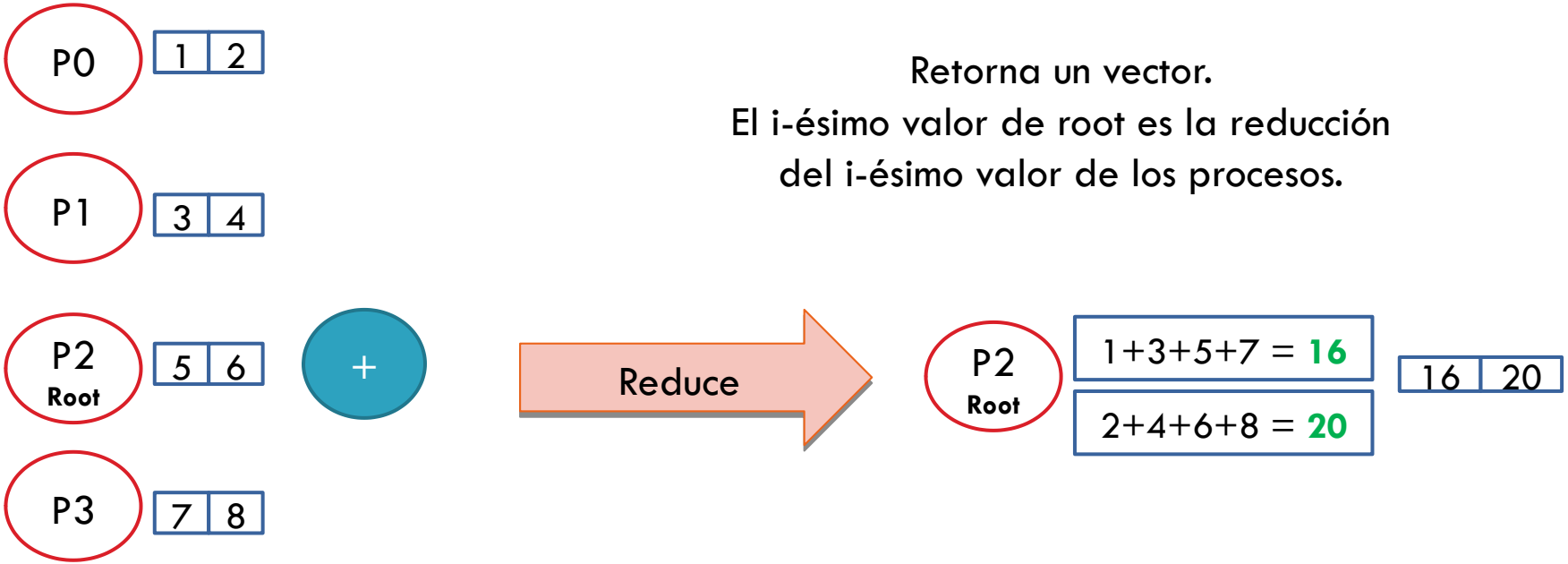
Retorna un valor con la reducción.

MPI – Comunicación colectiva - MPI_Reduce



Más de un elemento por proceso

MPI – Comunicación colectiva - MPI_Reduce



Retorna un vector.
El i-ésimo valor de root es la reducción del i-ésimo valor de los procesos.

MPI – Comunicación colectiva - MPI_Reduce - Ops

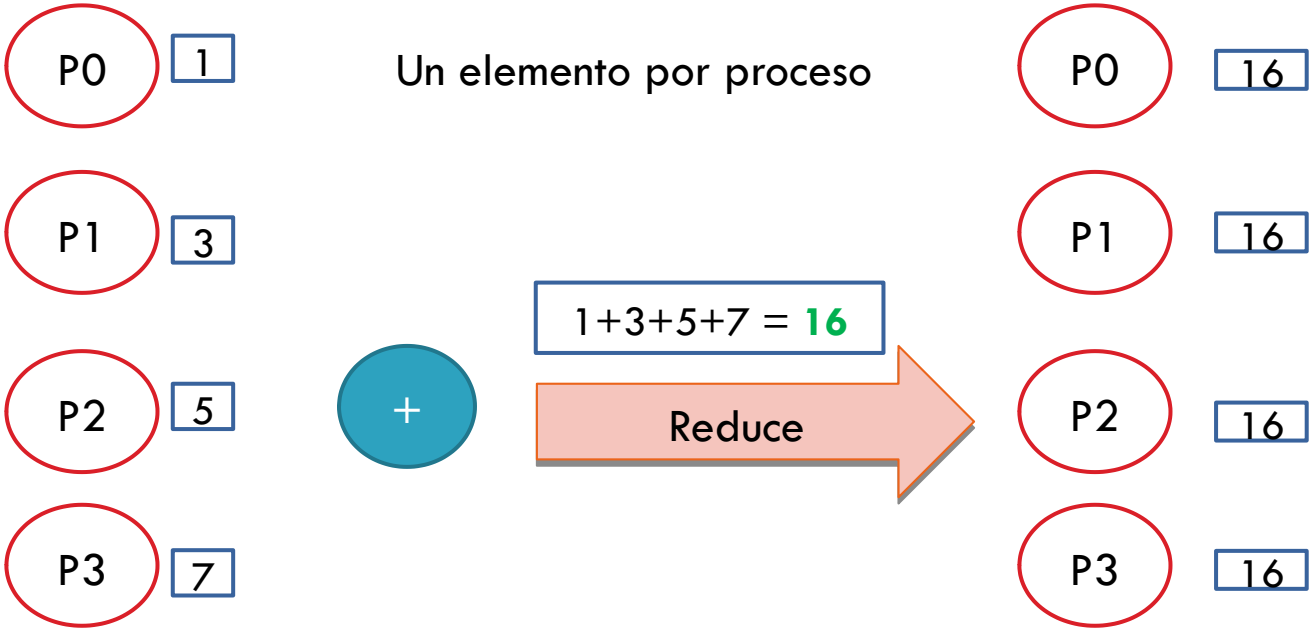
- MPI_MAX Máximo entre los elementos
- MPI_MIN Mínimo entre los elementos
- MPI_SUM Suma
- MPI_PROD Producto
- MPI_LAND AND lógico (devuelve 1 o 0, verdadero o falso)
- MPI_BAND AND a nivel de bits
- MPI_LOR OR lógico
- MPI_BOR OR a nivel de bits
- MPI_LXOR XOR lógico
- MPI_BXOR XOR a nivel de bits
- 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 – Comunicación colectiva - MPI_Allreduce

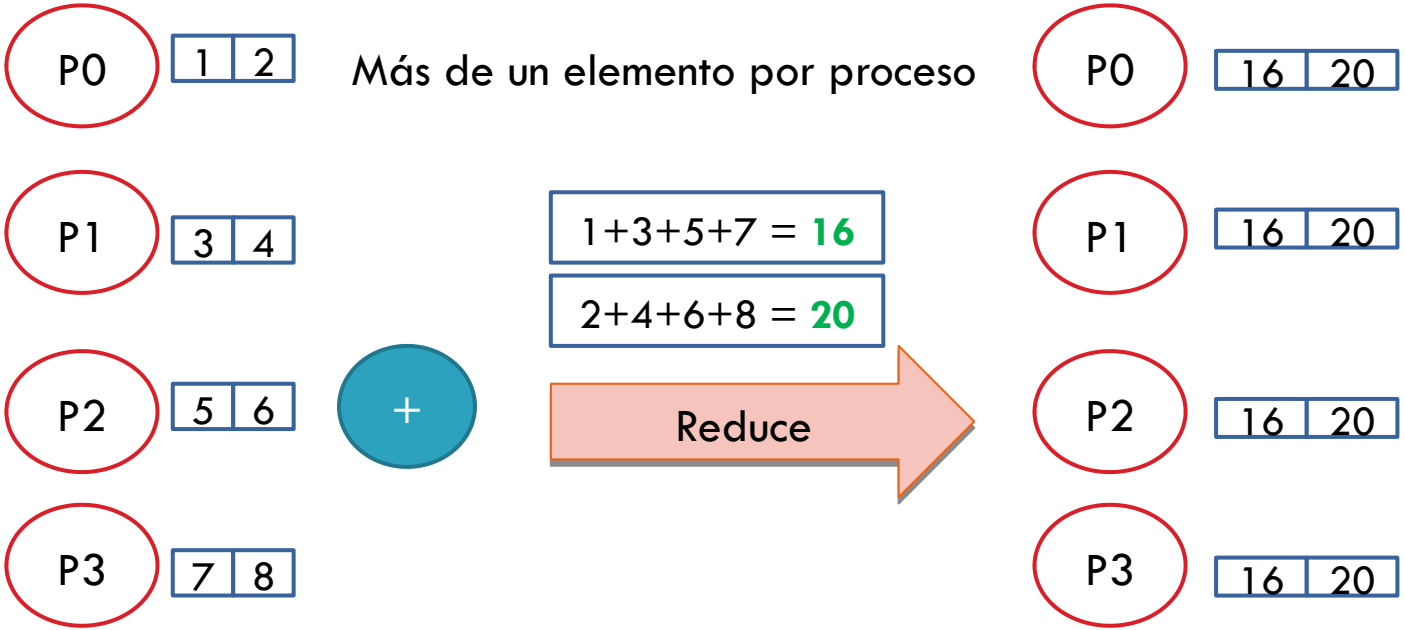
- **MPI_Allreduce**: similar a MPI_Reduce pero no necesita un proceso root. El valor queda distribuido en todos los procesos

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

MPI – Comunicación colectiva - MPI_Allreduce



MPI – Comunicación colectiva - MPI_Reduce



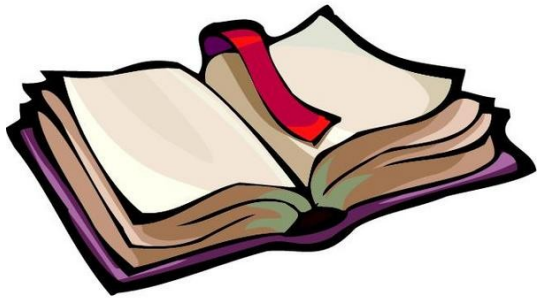
Agenda

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***



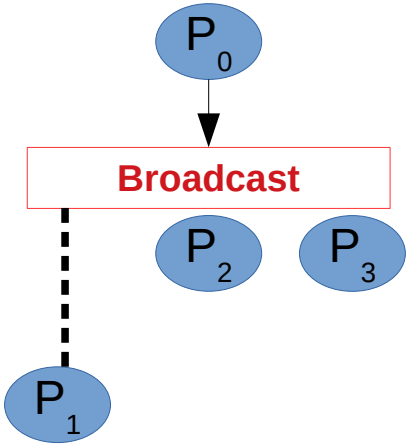
MPI – Ocultamiento de la latencia

- El uso de comunicación colectiva tiene la desventaja que los procesos no comienzan a hacer cómputo efectivo hasta que todos ejecuten la operación.

- Una técnica para evitar esto se conoce como Latency hiding/overlapping de cómputo y comunicación:
  - Utilizando Sends/Receives no bloqueantes
  - Se envían mensajes individuales. Los procesos que van recibiendo los mensajes van trabajando mientras que otros procesos se comunican.

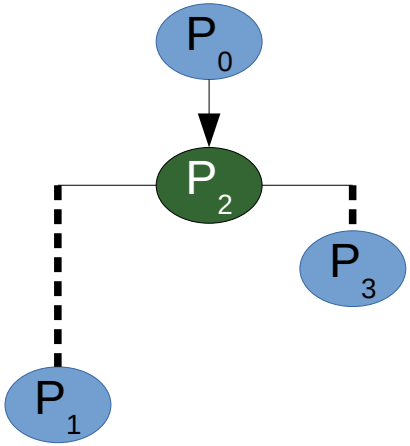
MPI – Ocultamiento de la latencia

Colectiva

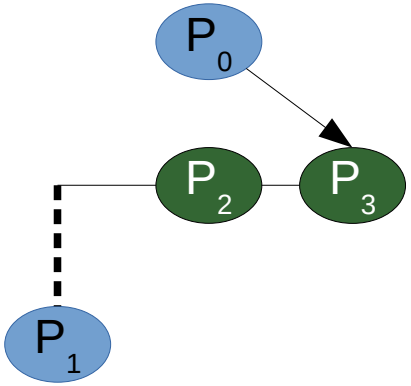


P2 y P3 deben esperar ociosos en un punto del código hasta que llegue P1

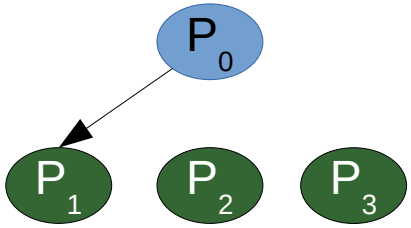
Punto a punto



1) P1 y P3 se retrasan pero P2 está listo, recibe los datos de P0 y comienza a realizar cómputo.



2) P1 sigue retrasado. P3 está listo, recibe los datos de P0 y comienza a realizar cómputo. Eventualmente P2 podría continuar la ejecución.



3) P1 está listo, recibe los datos de P0 y comienza a realizar cómputo. Eventualmente P2 y P3 podrían continuar la ejecución.