Algoritmos Paralelos
(clase 01.09.15)

Prof. J.Fiestas

Métodos de Programación, Message Passing:

- Comunicación (envío de mensajes) entre procesos.
- Utilizado en programación en paralelo (MPI) y orientada a objetos (C,C++,Fortran)
- No utiliza memoria compartida, sino espacio de memoria particionado en p nodos.
- Paralelización es explícita.

→ **Paralelismo del Especialista**

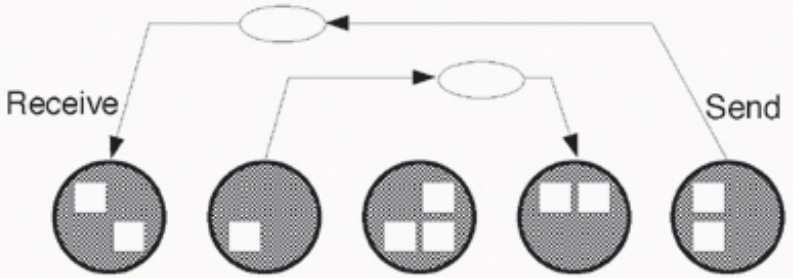


Figure 2.1
Message passing: the process structure - the number of processes and their relationships - determines the prgram structure. A collection of concurrent processes communicate by exchanging messages; every data object is locked inside some process. (Process are round, data objects square, messages oval).

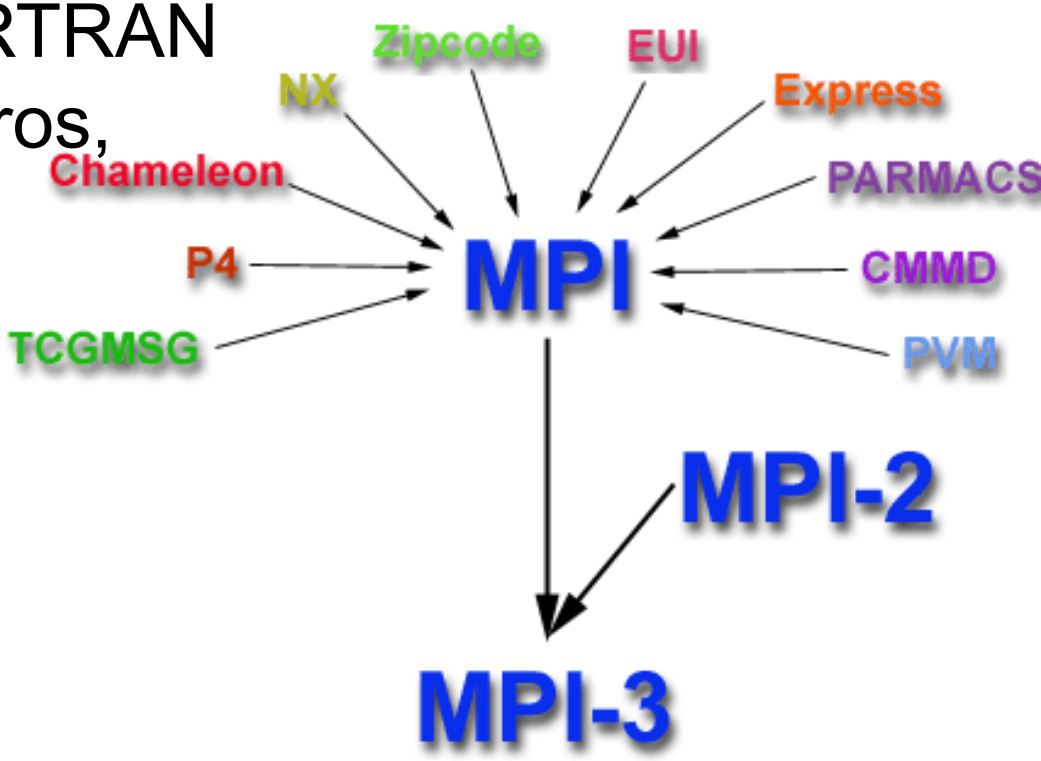
Message Passing Interface (MPI)

Interfase para la programación usando el paradigma de paso de mensajes

Aplicable a computadores de memoria compartida, distribuída e híbridos

Se usa en C, C++, FORTRAN

Utiliza funciones y macros, y paralelismo explícito



La Interfase MPI
(message passing interface)

MPI produce una interfase standard. Fue creada en EEUU y Europa va el 'MPI Forum' Luego de varios años de proposals, meetings y reviews, se creo el standard MPI

La interfase MPI permite portabilidad de código (escrito en C,C++,Fortran) a distintas arquitecturas

La Interfase MPI
(message passing interface)

Ademas de portabilidad e implementación en distintas arquitecturas, soporta arquitecturas híbridas.

Lo que no es explícito en MPI:

- Distribución de tareas en procesadores
- Creación de subprocesos durante ejecución
- Debugging
- Entrada/salida en paralelo

La Interfase MPI
(message passing interface)

MPI es una librería. Un proceso MPI consiste en un código C/Fortran, que se comunica con otros procesos MPI llamando rutinas MPI.
Puede combinarse con otros métodos de comunicación pero no está totalmente definido para ello (e.g OpenMP, problemas de sincronzación, ‘thread unsafe’)

Se utilizan scripts para compilar programas MPI, que incluyen las correspondientes librerías

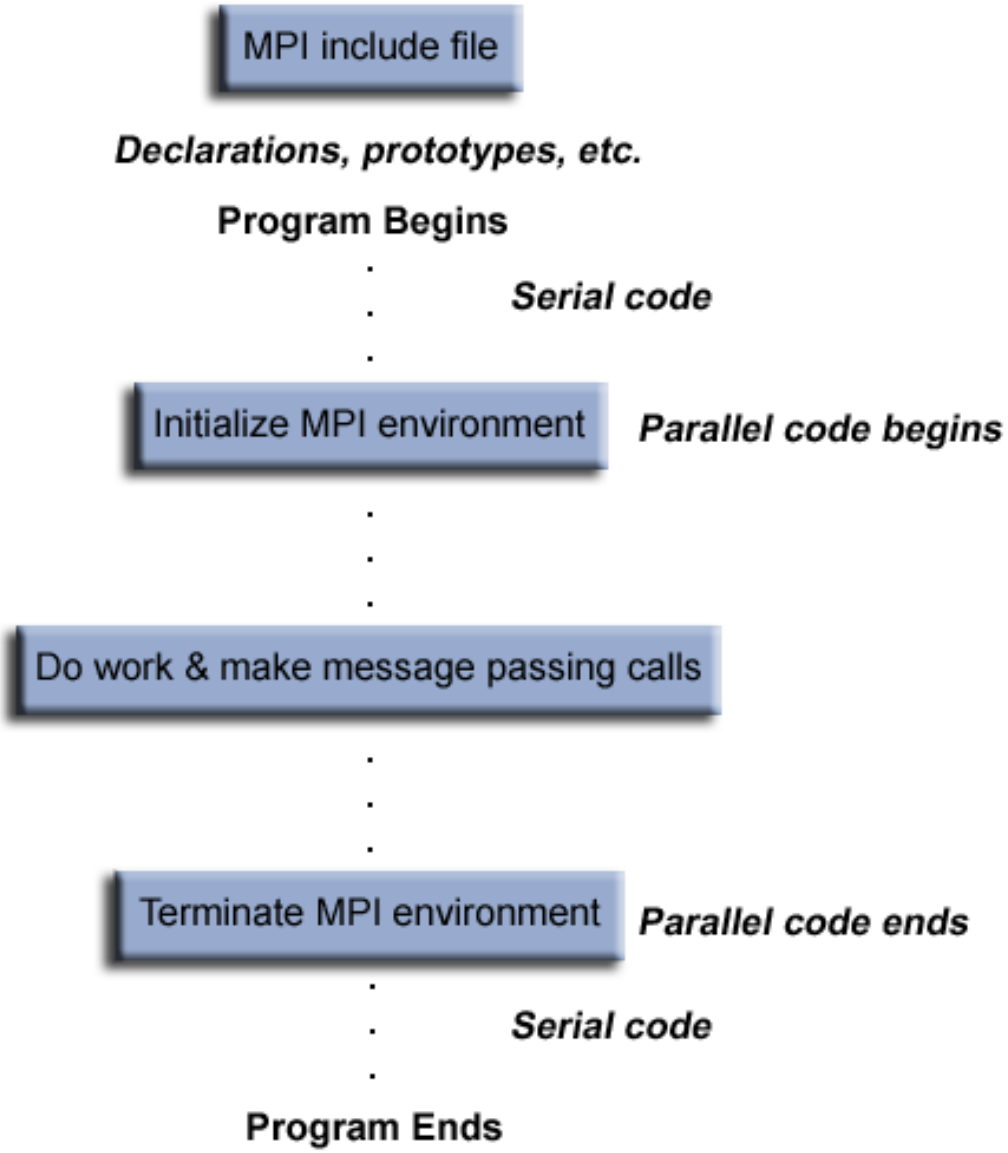
Language	Script Name	Underlying Compiler
C	`mpicc`	gcc
	`mpigcc`	gcc
	`mpiicc`	icc
	`mpipgcc`	pgcc
C++	`mpiCC`	g++
	`mpig++`	g++
	`mpiicpc`	icpc
	`mpipgCC`	pgCC
Fortran	`mpif77`	g77
	`mpigfortran`	gfortran
	`mpiifort`	ifort
	`mpipgf77`	pgf77
	`mpipgf90`	pgf90

Estructura de un programa en MPI

Requiere:

```
#include “mpi.h”
```

Cada proceso esta identificado con un número entero (rank), que se inicia en cero



MPI Handles

Son referencias a estructuras de data en MPI. Son variables de retorno en algunas directivas MPI, y pueden ser argumento de otras

Ejemplos:

MPI_SUCCESS – un entero, usado para detectar errores de código

MPI_COMM_WORLD - en C, un objeto de tipo MPI_Comm (a "communicator"); representa un comunicador pre-definido consistente de todos los procesos

Handles pueden ser copiados con las operaciones estandard de asignación

MPI Errors

En C, rutinas MPI retornan un entero, con un código de error. De ser detectado, el código aborta.

```
int ierr;
...
ierr = MPI_Init(&argc, &argv);
...
```

MPI Errors

Si la rutina es exitosa, el retorno es MPI_SUCCESS
E.g. formas de probar exito en la ejecución:

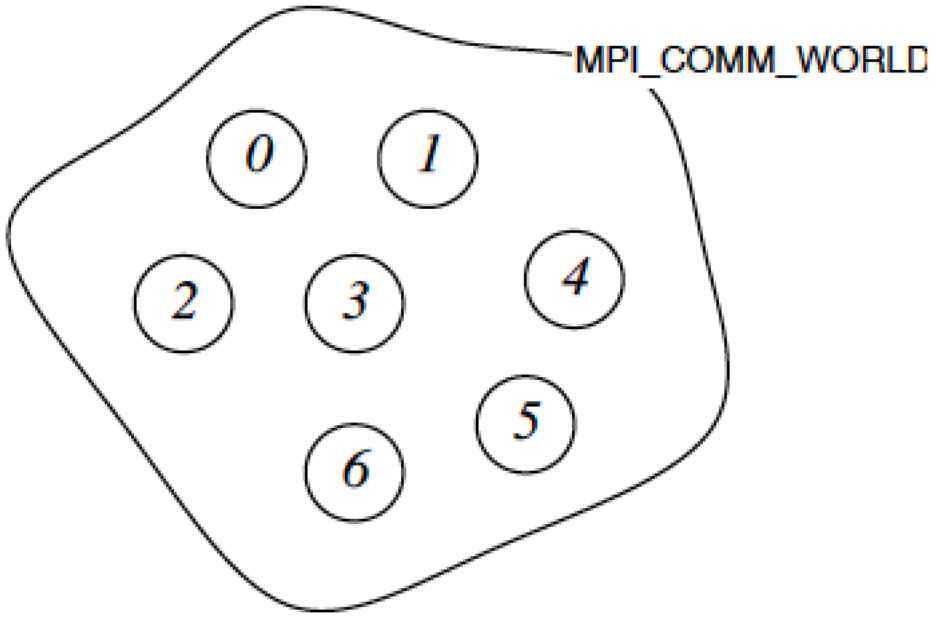
```
if (ierr == MPI_SUCCESS) {
    ...routine ran correctly...
}
```

Las directivas MPI tienen la forma **MPI_**

MPI_COMM_WORLD

MPI_Init define y llama MPI_COMM_WORLD para cada proceso. Que define la forma de comunicación entre procesos.

Este contiene una lista de procesos, que son numerados desde 0, y asignados a la variable **rank**



Funciones básicas MPI

MPI_Init:
Inicializa MPI. Se llama al inicio y una sola vez en el programa

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

MPI_Comm_size:
Retorna el número total de procesos MPI

```
MPI_Comm_size (comm, &size)
```

Funciones básicas MPI

MPI_Comm_rank:
Retorna el rank del proceso MPI

```
MPI_Comm_rank (comm, &rank)
```

MPI_Get_processor_name:
Retorna el nombre del procesador

```
MPI_Get_processor_name (&name, &lenght)
```

Funciones básicas MPI

MPI_Wtime:
Retorna el tiempo de reloj en segundos (double precision) en el proceso

MPI_Wtime ()

MPI_Finalize:
Termina la ejecución MPI y limpia todas sus estructuras

MPI_Finalize ()

MPI_Abort:
Termina todos los procesos MPI

MPI_Abort (comm,error)

Ejemplo: 'Hello World' en MPI,

```
#include "mpi.h"
#include <iostream>
using namespace std;

int main(int argc,char *argv[])
{
int rank, size;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

cout<<"Hola!"<<" soy "<< rank<<" de "<<size<<endl;

MPI_Finalize();
}
```

Ejemplo: ‘Hello World’ en MPI,

Compilación con C++:

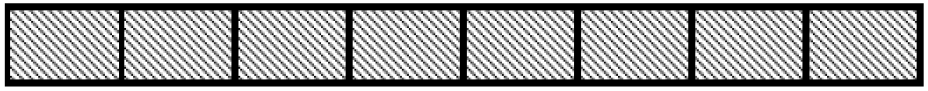
```
mpic++ -o ejemplo.exe ejemplo.cpp
```

Ejecución en 4 procesadores/núcleos:

```
mpiexec –np 4 ./ejemplo.exe
```

Mensajes MPI:

Un mensaje MPI es un array de elementos de un tipo particular MPI



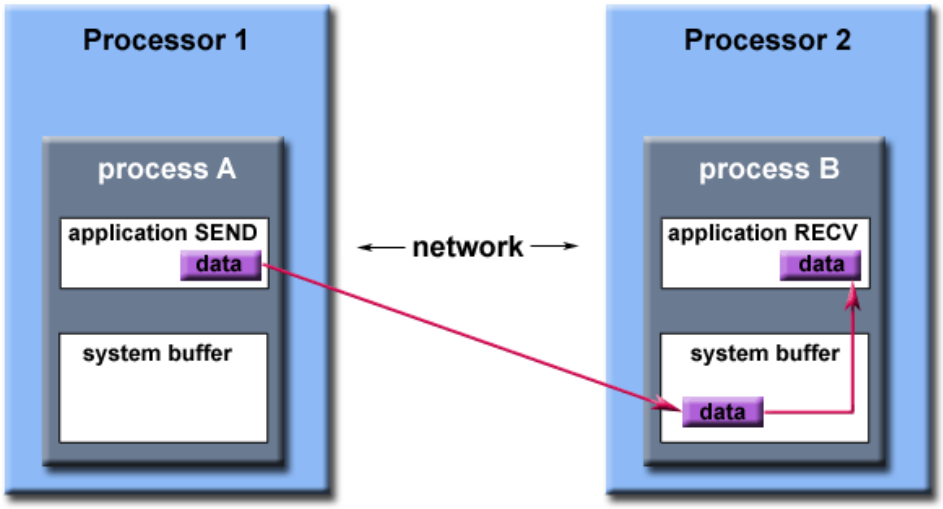
enviado debe ser el mismo que el recibido. Mpi soporta asi arquitecturas heterogéneas

MPI Datatype	C datatype
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	

Comunicación point-to-point (bloqueada)

Tipos de operaciones **point-to-point**, ejecutan envío de mensajes entre dos procesadores: envío y recibo 'seguro' hace que la ejecución del proceso se detenga y no retorne hasta que suceda el envío o recibo.

Envío y recibo pueden no estar sincronizadas, durante lo cual se graba la informacion en memoria (system buffer) reservada para guardar data en tránsito



Path of a message buffered at the receiving process

Modos de comunicación

Se clasifican de acuerdo al tipo de envío de mensajes (send), en

- **Send sincrónico**: solo se completa cuando ha sido recibido
- **Buffered send**: siempre se completa (haya sido recibido o no)
- **Send standard**: sincrónico o buffered
- **Ready send**: siempre se completa (haya sido recibido o no)
- **Receive**: se completa cuando el mensaje ha sido recibido

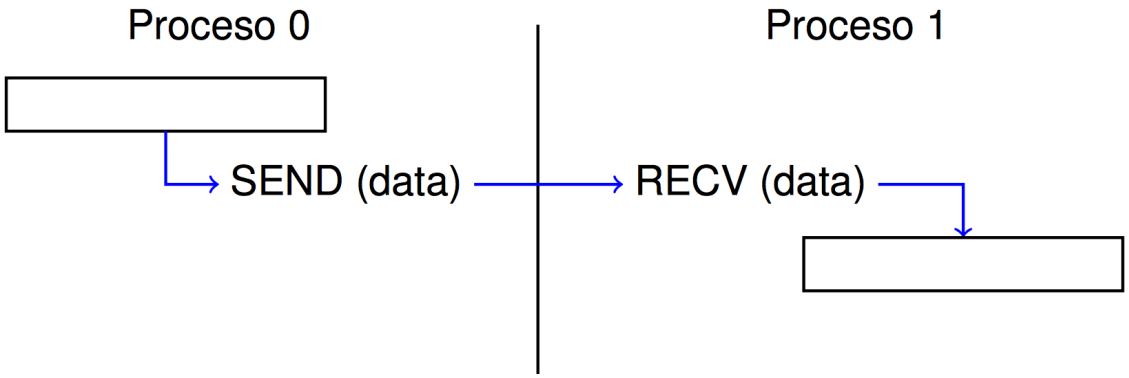
Envío y recibo pueden no estar sincronizadas, durante lo cual se graba la informacion en memoria (system buffer) reservada para guardar data en tránsito

Rutinas MPI de comunicación point-to-point

MPI_Send (standard):

Operación de envío de mensajes. Tener en cuenta que

- No se debe asumir que el envio del mensaje se complete antes que se reciba, o luego que **receive** empieze
- Debe ser sincónico o buffered



Rutinas MPI de comunicación point-to-point

MPI_Send (standard):

Operación de envío de mensajes

MPI_Send (&buffer, count, datatype,dest,tag,comm)

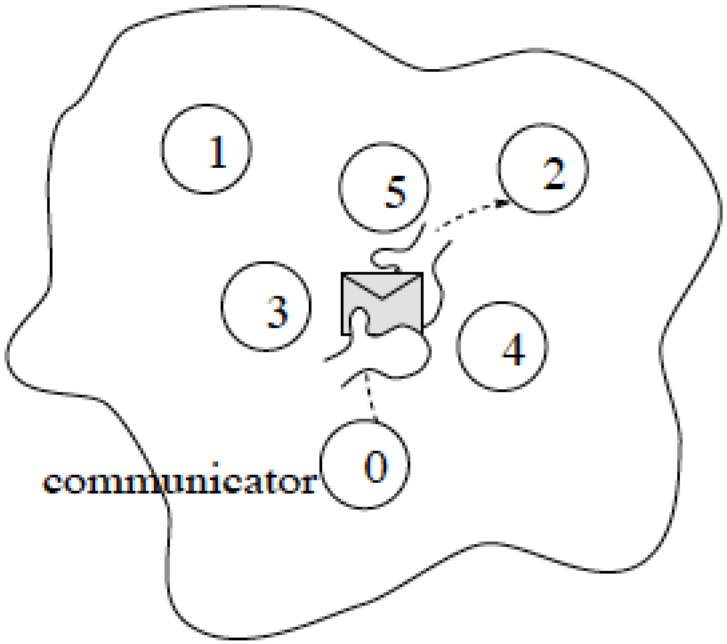
buffer: puntero a la variable que ocupa la data que va a ser enviada o recibida (nombre de la variable)
count: numero de elementos de la data que seran enviados/ recibidos
datatype: tipo de variable MPI
dest: proceso destino del mensaje (rank)
tag: identificador del proceso que envía, útil en caso de envío de múltiples mensajes
comm: comunicador del grupo de procesos activo

Rutinas MPI de comunicación point-to-point

Synchronous Send:

Se utiliza si el proceso que envía debe saber si el mensaje ha sido recibido.

Para ello el enviante manda un mensaje de ‘recibido’ al enviante, momento en el cual se considera el mensaje como enviado



Rutinas MPI de comunicación point-to-point

Synchronous Send:

Operación de envío de mensajes

MPI_SSend (&buffer, count, datatype,dest,tag,comm)

buffer: puntero a la variable que ocupa la data que va a ser enviada o recibida (nombre de la variable)
count: numero de elementos de la data que seran enviados/ recibidos
datatype: tipo de variable MPI
dest: proceso destino del mensaje (rank)
tag: identificador del proceso que envía, útil en caso de envío de múltiples mensajes
comm: comunicador del grupo de procesos activo

Rutinas MPI de comunicación point-to-point

Buffered Send:

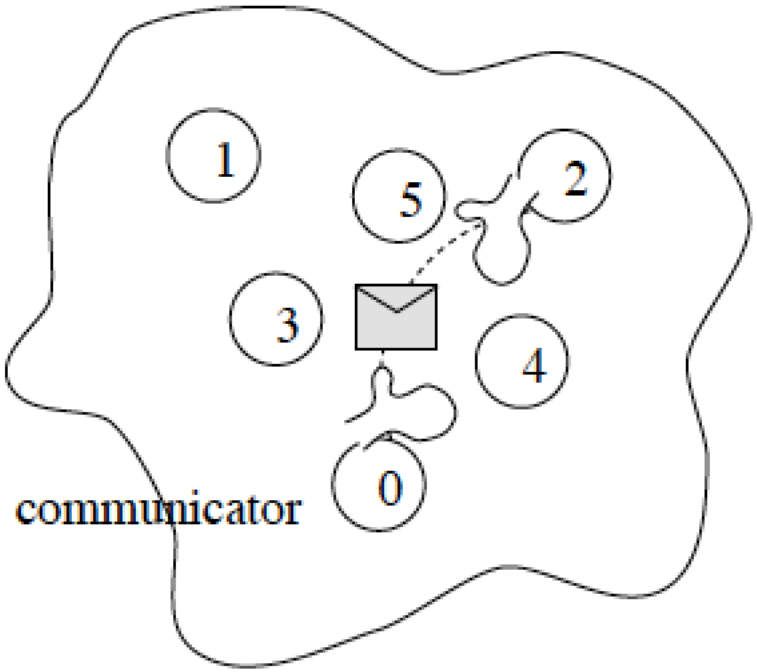
Garantiza un término inmediato, a través del almacenamiento en un buffer para transmisión posterior si es necesario.

Para ello, el usuario debe destinar suficiente memoria para el programa utilizando
MPI_BUFFER_ATTACH (buffer, size)
MPI_BUFFER_DETACH (buffer, size)

Rutinas MPI de comunicación point-to-point

Ready Send:

Se completa inmediatamente, enviando el mensaje al comunicador. Este se recibirá solo si el proceso esta listo para hacerlo, si no ocurrira un error.
El objetivo es mejorar performance.



Rutinas MPI de comunicación point-to-point

Ready Send:

Es un modo dificil para debugging. Solo recomendable si performance es crítica.

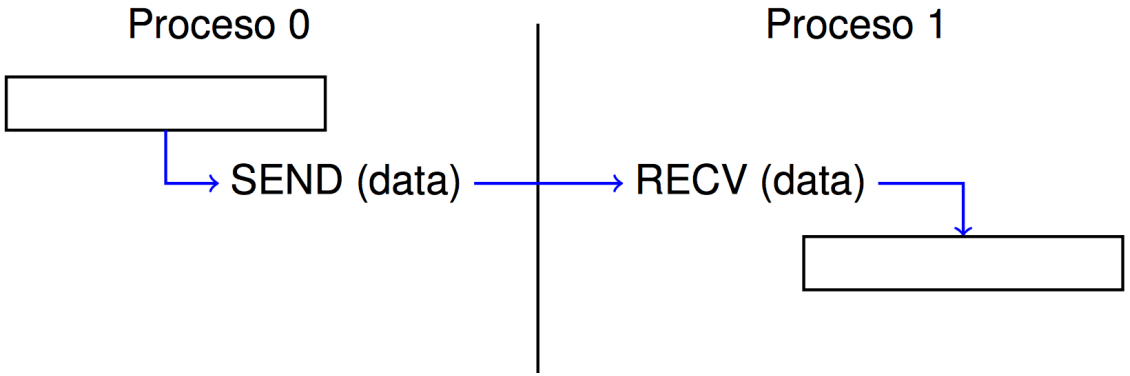
MPI_RSend (&buffer, count, datatype,dest,tag,comm)

buffer: puntero a la variable que ocupa la data que va a ser enviada o recibida (nombre de la variable)
count: numero de elementos de la data que seran enviados/ recibidos
datatype: tipo de variable MPI
dest: proceso destino del mensaje (rank)
tag: identificador del proceso que envía, útil en caso de envío de múltiples mensajes
comm: comunicador del grupo de procesos activo

Rutinas MPI de comunicación point-to-point

MPI_Recv (standard blocking)

Operación de recibo de mensajes. Se completan cuando se confirma por el proceso 1.



Rutinas MPI de comunicación point-to-point

MPI_Recv (standard blocking)

MPI_Recv (&buffer, count, datatype,source,tag,comm,&status)

data Type: tipos de data MPI (el usuario puede crear sus propios tipos de datos)
dest: argumento de envío de rutinas que indica el proceso a donde el mensaje será enviado (rank del proceso receptor)
source: receptor de rutinas indicando el origen del mensaje (rank del proceso enviado)
tag: entero no-negativo asignado por el usuario para identificar el mensaje
comm: contexto de comunicación (usualmente MPI_COMM_WORLD)
status: indica el estado del objeto

Rutinas MPI de comunicación point-to-point

Estado de la comunicación

Contiene información del remitente, conocida como status.

status.MPI_SOURCE contiene información de la fuente del mensaje

status.MPI_TAG contiene información del identificador del mensaje

MPI_GET_COUNT (status, datatype, count) contiene el número de elementos recibidos

Rutinas MPI de comunicación colectiva

Facilita operaciones colectivas de comunicación

- No puede interferir con comunicación point-to-point
- Puede o no sincronizar los procesos
- El buffer se re-utiliza solo cuando el proceso termina
- Todos los procesos en un comunicador participan de comunicación colectiva

Rutinas MPI de comunicación colectiva

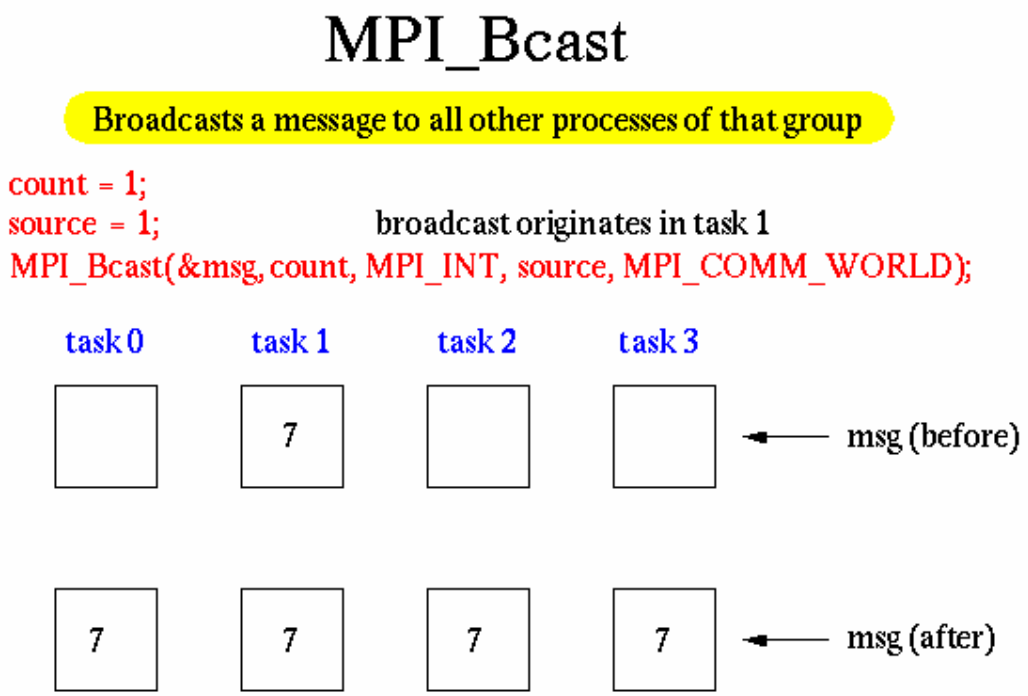
MPI_Barrier:

Operación de sincronización. No utiliza data. Funciona bloqueando el proceso de llamada hasta que todos los miembros del grupo han llamado a la operación

MPI_Barrier (comm)

Rutinas MPI de comunicación colectiva

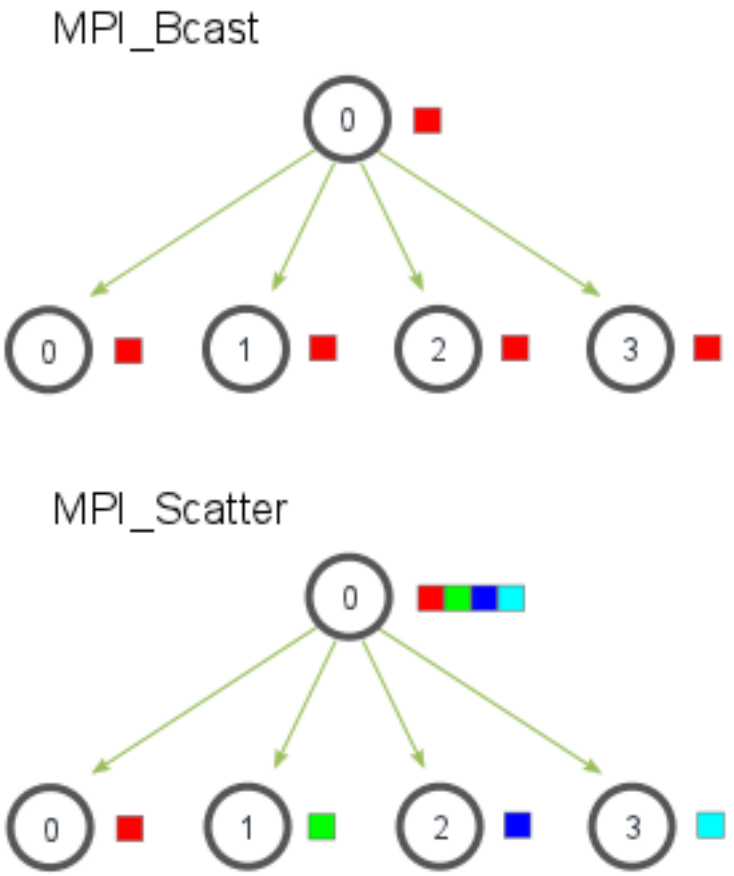
MPI_Bcast:
Se envía mensaje del rank 'root' a todos los procesos en el grupo



MPI_Bcast (&buf, count, datatype,root,comm)

Rutinas MPI de comunicación colectiva

MPI_Scatter:
Distribuye mensajes de una sola fuente a cada proceso en el grupo



Rutinas MPI de comunicación colectiva

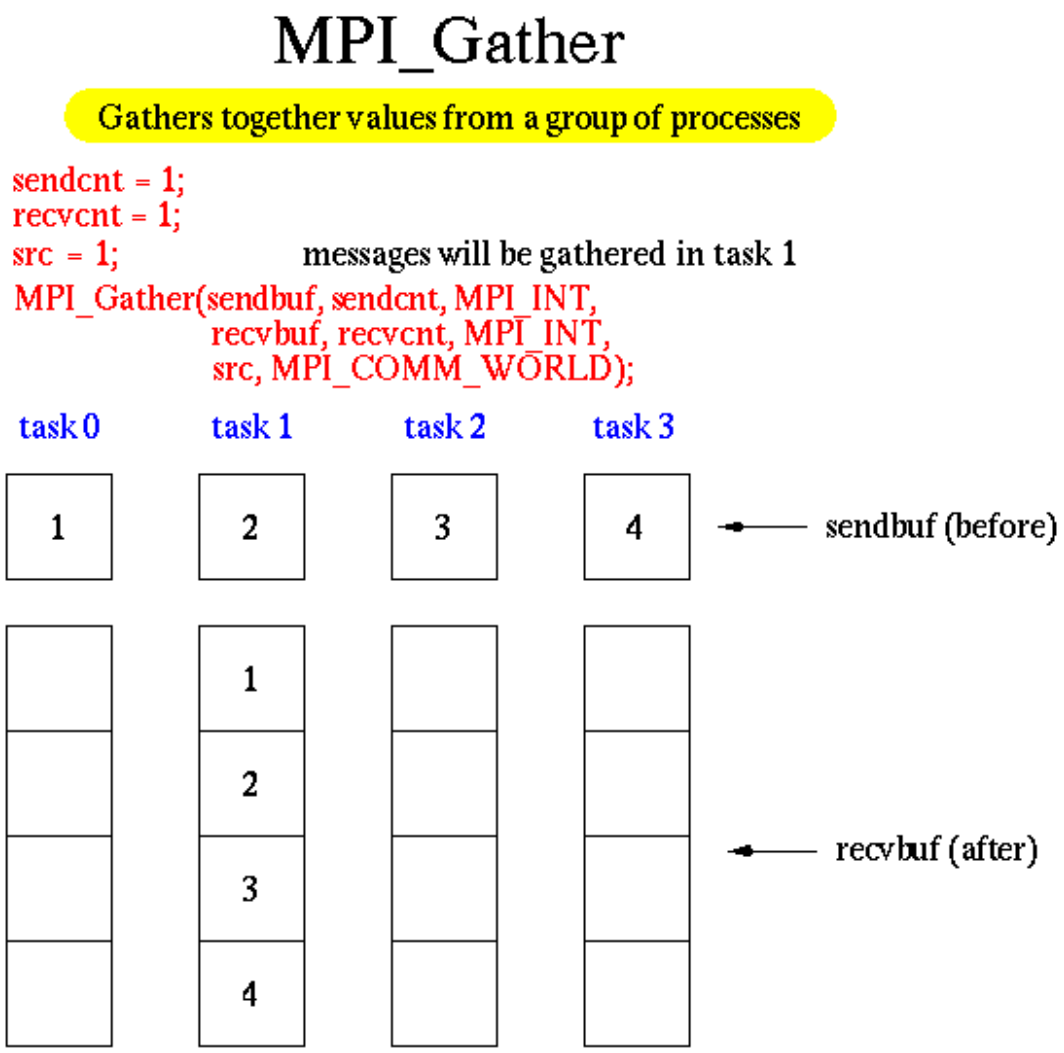
MPI_Scatter (&buf, sendcnt, sendtype,&recvbuf, recvnt,recvtype,root,comm)

&buf: dirección del buffer que reside en el nodo 0 (raiz)
sendcnt: numero de elementos a ser enviados por proceso (elementos del array / numero de nodos)
sendtype: tipo de elementos enviados
&recvbuf: buffer que recibe **recvnt** elementos del tipo **recvtype**
root: nodo raiz
comm: comunicador en el que residen los procesos

MPI_Gather:

Recopila informacion de cada proceso en un solo destino. Reverso de MPI_Scatter

Los elementos estan ordenados de acuerdo al rango del proceso de donde son recibidos



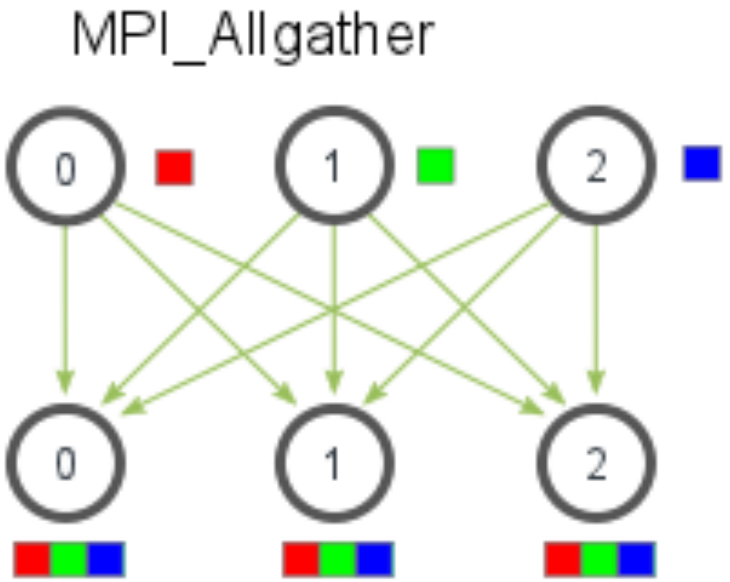
MPI_Gather:

MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf,recvcnt,recvtype,root,comm)

Nota:
recvcnt es el número de elementos recibidos por proceso, NO la suma de elementos de todos los procesos

MPI_Allgather:

Recopila informacion de cada proceso en todos los demas procesos. Actua como un MPI_Gather seguido de un MPI_Bcast



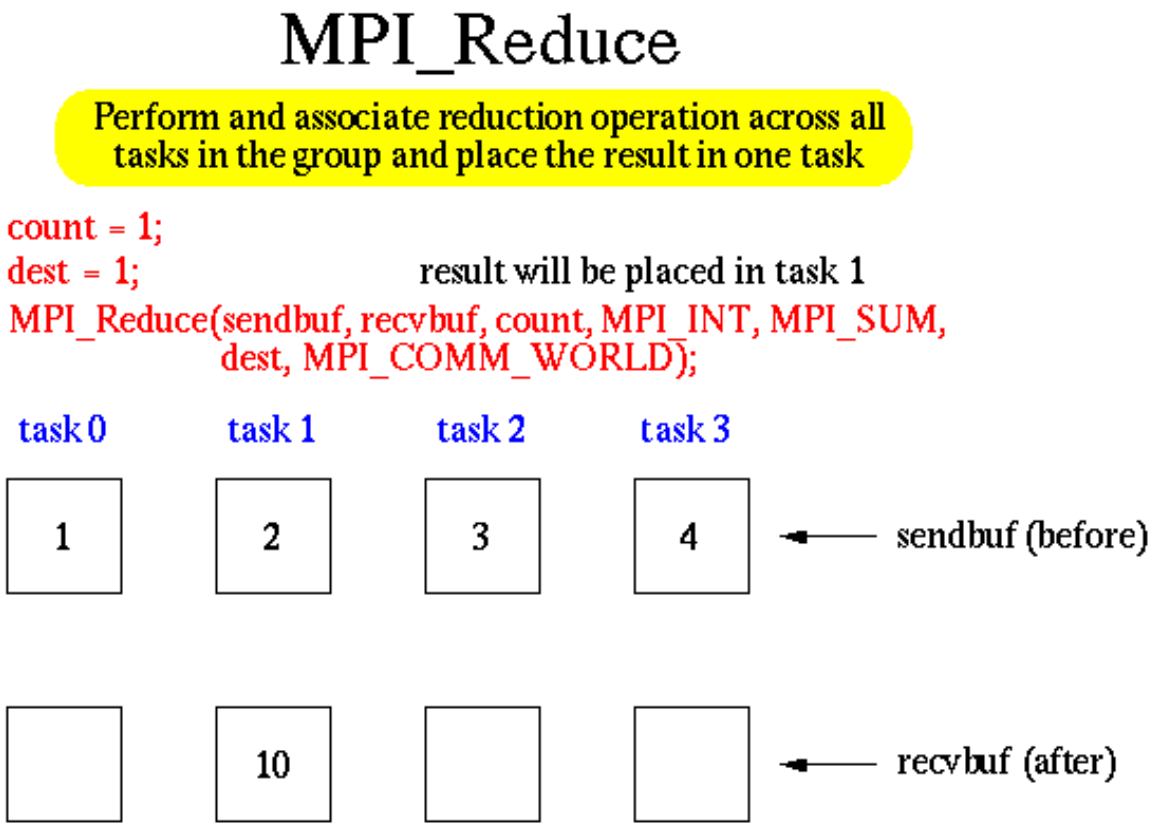
Los elementos son recopilados en el orden de rango de donde provienen

No contiene un proceso principal (root)

MPI_Allgather (&sendbuf, sendcnt, sendtype, &recvbuf,recvcnt,recvtype,comm)

Operaciones de reducción

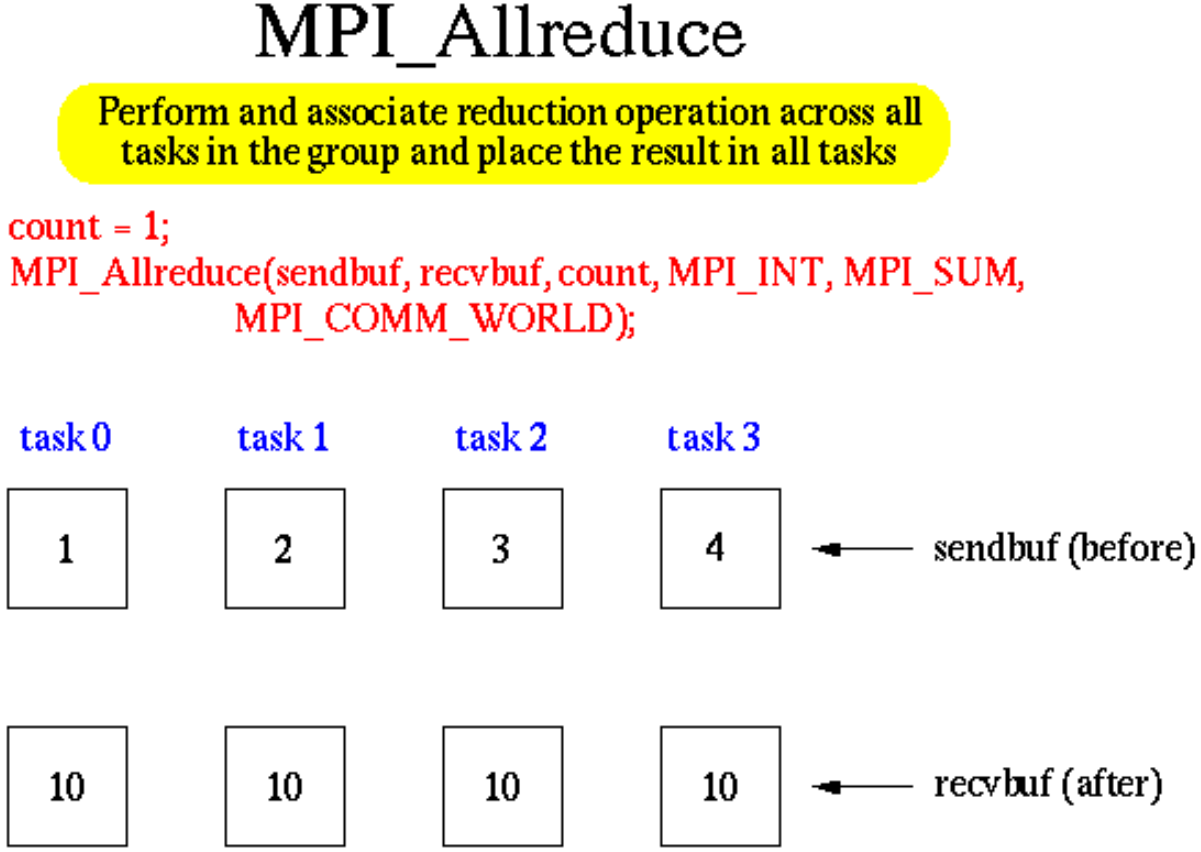
MPI_Reduce:
Recopila los datos del grupo y los pone en un task



MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)

MPI_Allreduce:

Recopila los datos delgrupo
y los pone en todos
los task del grupo



MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)

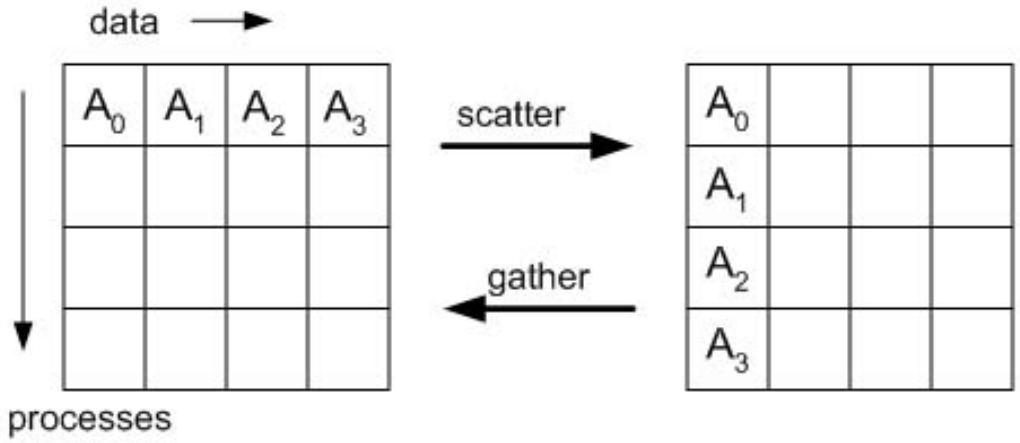
Operaciones de reducción

Operadores pre-definidos

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum & location
MPI_MINLOC	Minimum & location

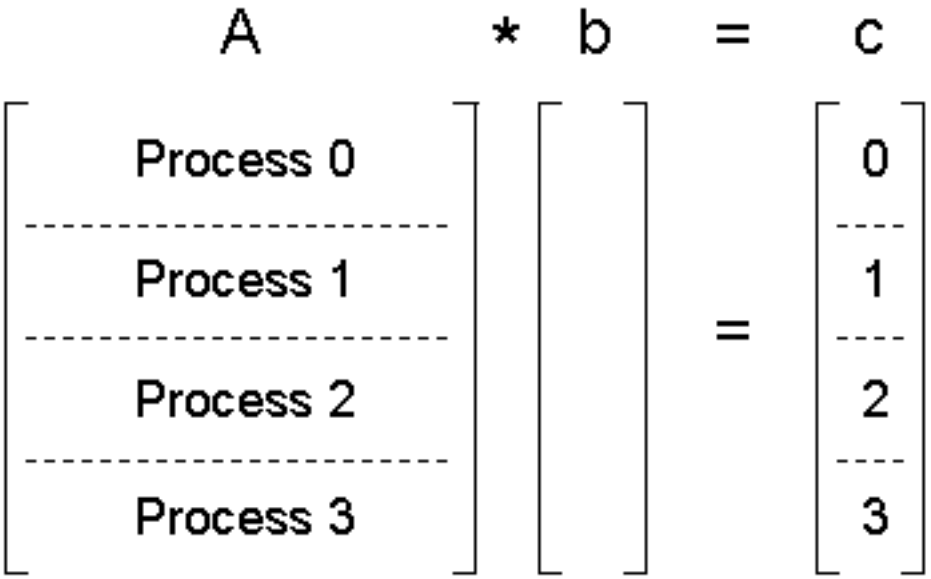
Ejemplo:

Representación de matriz



Multiplicacion Matriz-Vector

- Matriz se distribuye en filas
- producto escalar se realiza en cada proceso
- **MPI_Gather** se usa para recolectar los productos



Ejemplo:

MPI_Gather recolecta resultados de cada proceso y las une en el vector resultante

```
double a[25,100],b[100],cpart[25],ctotal[100];
int root;
root=0;
for(i=0;i<25;i++)
{
  cpart[i]=0;
  for(k=0;k<100;k++)
  {
    cpart[i]=cpart[i]+a[i,k]*b[k];
  } }
MPI_Gather(cpart,25,MPI_REAL,ctotal,25, MPI_REAL, root,
MPI_COMM_WORLD);
```

Ejercicio. Paralelizar la multiplicación de dos matrices

```
for(i=0;i<NA;i++)
  for(j=0;j<NB;j++)
    for(k=0;k<NC;k++)
      C[i][j]+= A[i][k] * B[k][j];
```

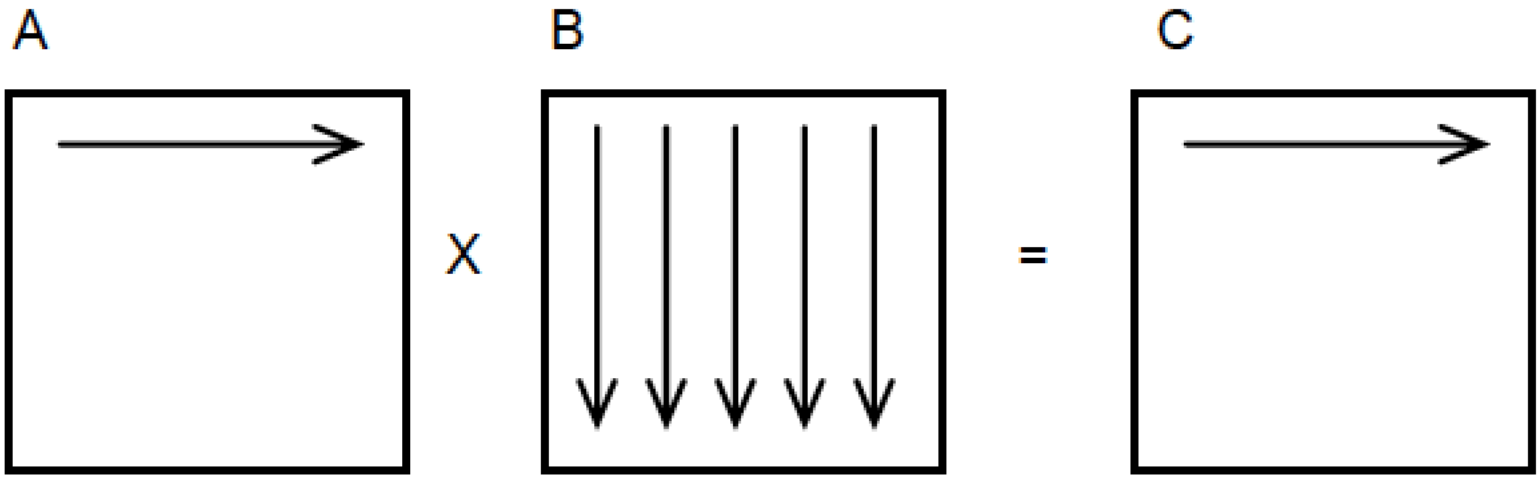
$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

$$\begin{aligned} \text{Task 1: } C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ \text{Task 2: } C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ \text{Task 3: } C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ \text{Task 4: } C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

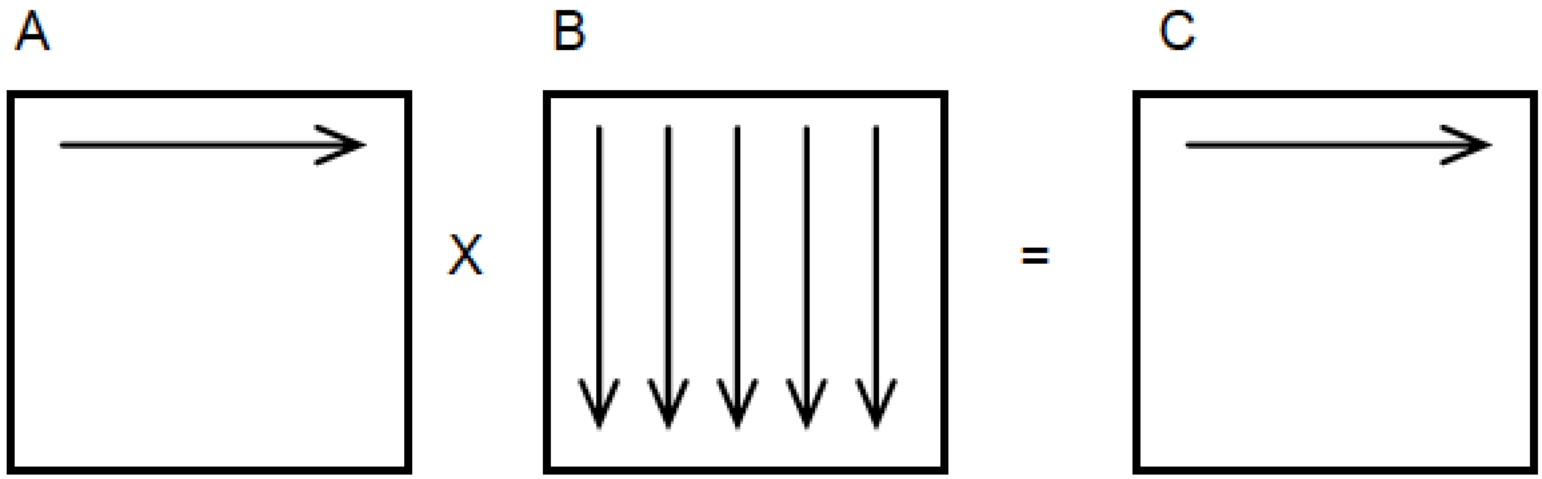
(b)

Ejercicio. Paralelizar la multiplicación de dos matrices



Algoritmo ejecuta lineas de la matriz C en forma secuencial
Cada iteración, una linea de A y todas las columnas de B son procesadas
Complejidad del problema: O(ijk), donde las matrices son i x j

Ejercicio. Paralelizar la multiplicación de dos matrices



La operación básica es calcular un elemento de C

$$c_{ij} = \left(a_i, b_j^T\right),\ a_i = \left(a_{i0,} a_{i1}, \ldots, a_{in-1}\right),\ b_j^T = \left(b_{0j}, b_{1j}, \ldots, b_{n-1j}\right)^T$$

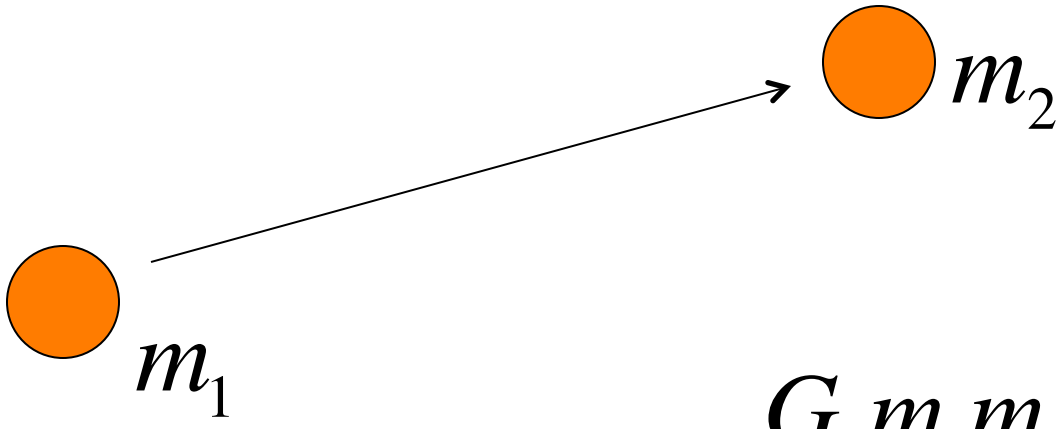
En general, el número de procesos es $p < n^2$ d, por lo que sera necesario realizar sub-operaciones que albergen una fracción de la matriz

Ejercicio. Paralelizar la multiplicación de dos matrices

```
if (rank == 0) {
      // initialize matrices
      // Send matrix data to the worker tasks
      for (dest=1; dest<=np-1; dest++)
      { .....
            rows = .....
MPI_Send(&a[offset][0], rows*NA, MPI_DOUBLE, dest, mtype,MPI_COMM_WORLD);
      }
       for (i=1; i<np; i++)
      { ......
MPI_Recv(&c[offset][0], rows*NB, MPI_DOUBLE, i, mtype,MPI_COMM_WORLD, &status);
      }
}
 if (rank > 0) {
 MPI_Recv(&a, rows*NA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD,&status);
      // calculate sub-matrix
MPI_Send(&c, rows*NB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
....
}
```

Ejemplo: 2-body gravitational problem:

can be solved analytically
(Keplerian orbit, no perturbations)

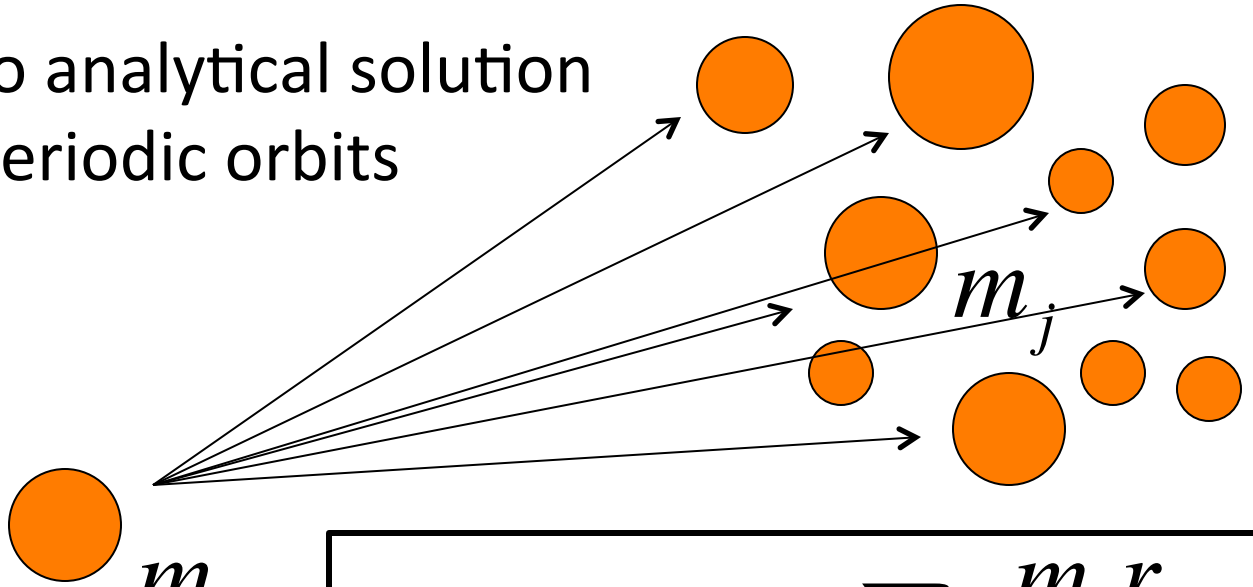


$$F = -\frac{G\, m_1 m_2}{r^2}$$

N-body problem:

from N=3 no analytical solution

No stable/periodic orbits



$$F_i = -Gm_i \sum_{\substack{1 \le j \le N \\ j \neq i}} \frac{m_j r_{ij}}{\left\| r_{ij} \right\|^3}$$

Number of force calculations:

N(N-1)/2

Simple N-body code (C++) for direct summation of forces $O(N^2)$: n-body.cpp

A galaxy is an Nbody class

```
class Nbody
{
public:
float pos[3][N];
float vel[3][N];
Float m[N]
}
```

```
int main (int arg, char
**argv)
{
…
// define class galaxy
Nbody galaxy;
…
// initialize properties
galaxy.init();
…
// integrate forces
galaxy.integr();
return 0;
}
```

```
void integr ()
{
…
// measure CPUtime
start=clock();

force(n, pos, vel, m, dt)

// measure CPUtime
end = clock();
cpuTime= difftime(end,start)/
(CLOCKS_PER_SEC)
…
}
```

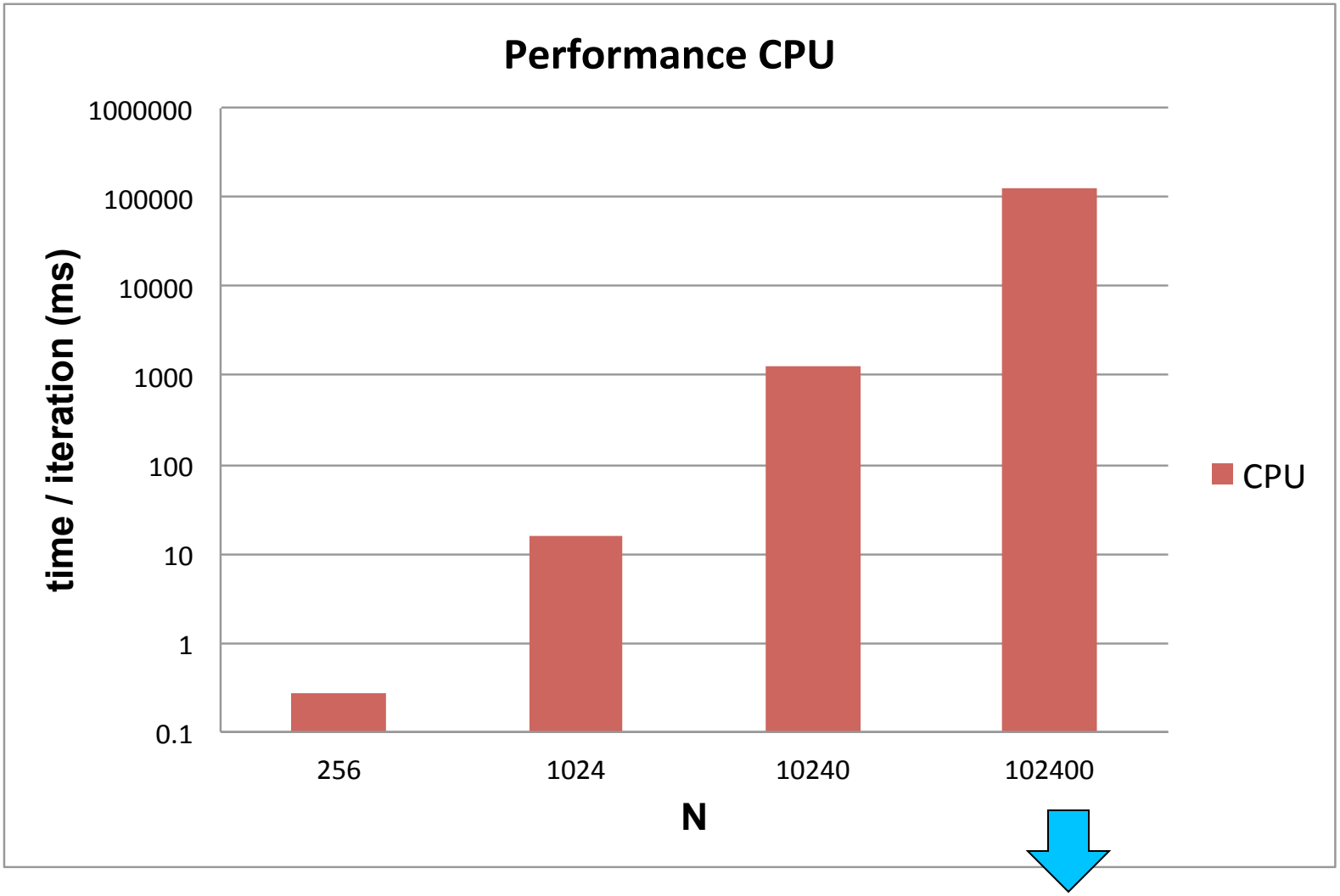
N-body naive code (C++):

```
void force(int n, float pos[][..],
….,float vel[][..], float m[..], float
dt)
{
 // sume over i
  for (int i=0; i<n; i++)
  {
    float my_r_x = r_x[i];
 // sume over j
    for (int j=0; j<n; j++)
    {
        if(j!=i)   // avoid i=j
        {
// compute accelerations
    float d = r_x[j]-my_r_x; // 1 flop

    a_x += G*m[j]/(d*d);  // 4 flops
        …
        }
    }
```

```
//update velocities
v_x[i] += a_x*dt; // 2 flops
// update positions
r_x[i] += v_x[i]*dt; // 2 flops
  }
}
```

N-body simulations on GPU clusters



Our example: N=100K needs ~100 sec./iteration

→ 10^7 iter. gives 10^9 sec (~30 yrs !)

Ejercicio. Paralelizar el problema de N cuerpos

1. Resolver el problema de N cuerpos utilizando MPI para paralelizar el cálculo de fuerzas en varios núcleos
2. Medir los tiempos de cómputo para np=2,4,8,16, fijando N=10000
3. Medir los tiempos de cómputo para N=100,1000,10000, fijando np=4
4. Calcular los FLOPS del algoritmo y plotear FLOPS vs. Np
5. (Alternativo) incluir en un loop la variable de tiempo (delta_t) y calcular la nueva posición y velocidad de los cuerpos