COMPUTER ARCHITECTURE
AND
SYSTEMS GROUP

MPI: Message Passing Interface

Arquitectura de Computadores II
Ingeniería en Informática (4º Curso)
Universidad Carlos III de Madrid

Definición

- MPI es una interfaz de paso de mensaje que representa un esfuerzo prometedor de mejorar la disponibilidad de un software altamente eficiente y portable para satisfacer las necesidades actuales en la computación de alto rendimiento a través de la definición de un estándar de paso de mensajes universal.

William D. Gropp et al.

¿Que es MPI?

- Es un estándar de ley.
- Estándar para una biblioteca de paso de mensajes.
- Implementaciones de MPI:
  - www-unix.mcs.anl.gov/mpi/mpich/
  - www.lam-mpi.org/
- Otras direcciones:
  - www-unix.mcs.anl.gov/mpi/
  - www.mpi-forum.org/

Introducción: Enfoque práctico

- Imaginemos que queremos ejecutar el siguiente programa en 4 máquinas distintas de forma paralela:

```
int main(){
printf("Hola mundo\n");
}
```

```
gcc –Wall hola.c –o hola
./hola
```

- Una opción, es ir a las 4 máquinas, y ejecutar el programa en ellas (**POCO EFICIENTE**).

Introducción: Enfoque práctico

- Otra opción es utilizar MPI, y desde una maquina, ejecutar el siguiente código (**MÁS EFICIENTE**)

```
Int main() {
   MPI_Init(&argc,&argv);
   printf("Hola mundo\n");
   MPI_Finalice();
 }
```

```
mpicc –o hola hola.c
mpirun –n 4 hola
```

Generalidades

- MPI 1.2 tiene 129 funciones
- Las funciones principales de MPI son:
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv

Proceso

- La *unidad básica* en MPI son los **procesos**.
- Tienen espacios de memoria independientes.
- Intercambio de información por paso de mensajes.
- Introduce en concepto de comunicadores (grupo de procesos más contexto).
- Cada proceso se le asigna un identificador interno propio de MPI (rank).

Mi primer programa con MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]) {
    int myrank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,
   &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hola soy el proceso %d de %d\n",
   myrank, size);
    MPI_Finalize();
exit(0);
```

Comunicadores

- MPI agrupa los procesos implicados en una ejecución paralela en comunicadores.
- Un comunicador agrupa a procesos que pueden intercambiarse mensajes.
- El comunicador MPI_COMM_WORLD está creado por defecto y engloba a todos los procesos.

MPI_Init, MPI_Finalize

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

- Primera llamada de cada uno de los procesos MPI
- Establece entorno
- Un proceso solo puede hacer una llamada MPI_INIT

int MPI_Finalize(void)

- Termina la ejecución en MPI
- Libera los recursos utilizados por MPI

Identificación de procesos

- **MPI_Comm_rank (comm, &pid);**
  - Devuelve en ***pid*** el identificador del proceso dentro del comunicador comm especificado
  - (ej.: MPI_COMM_WORLD).
- **MPI_Comm_size (comm, &npr);**
  - Devuelve en ***npr*** el número de procesos del comunicador especificado.

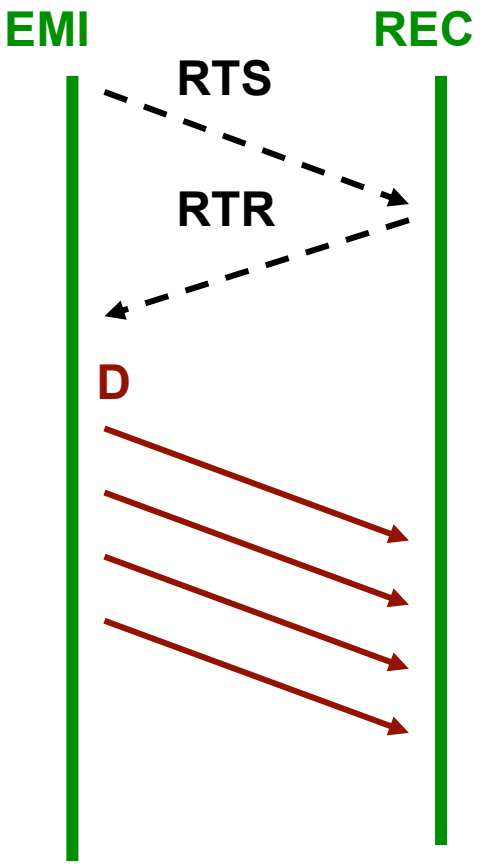
Tipos de mensajes (I)

- Punto a punto: **1 emisor -1 receptor**
  - Síncronos: El emisor se espera a que la comunicación se produzca (o que el buffer del receptor este disponible). **BLOQUEANTE**
    - Send, Recv, SendRecv, Bsend, Ssend, …
  - Asíncronos: El emisor NO se espera a que la comunicación se produzca y se comprueba más tarde si se ha producido. **NO BLOQUEANTES**
    - Isend, Irecv, …

Tipo de mensajes (II)

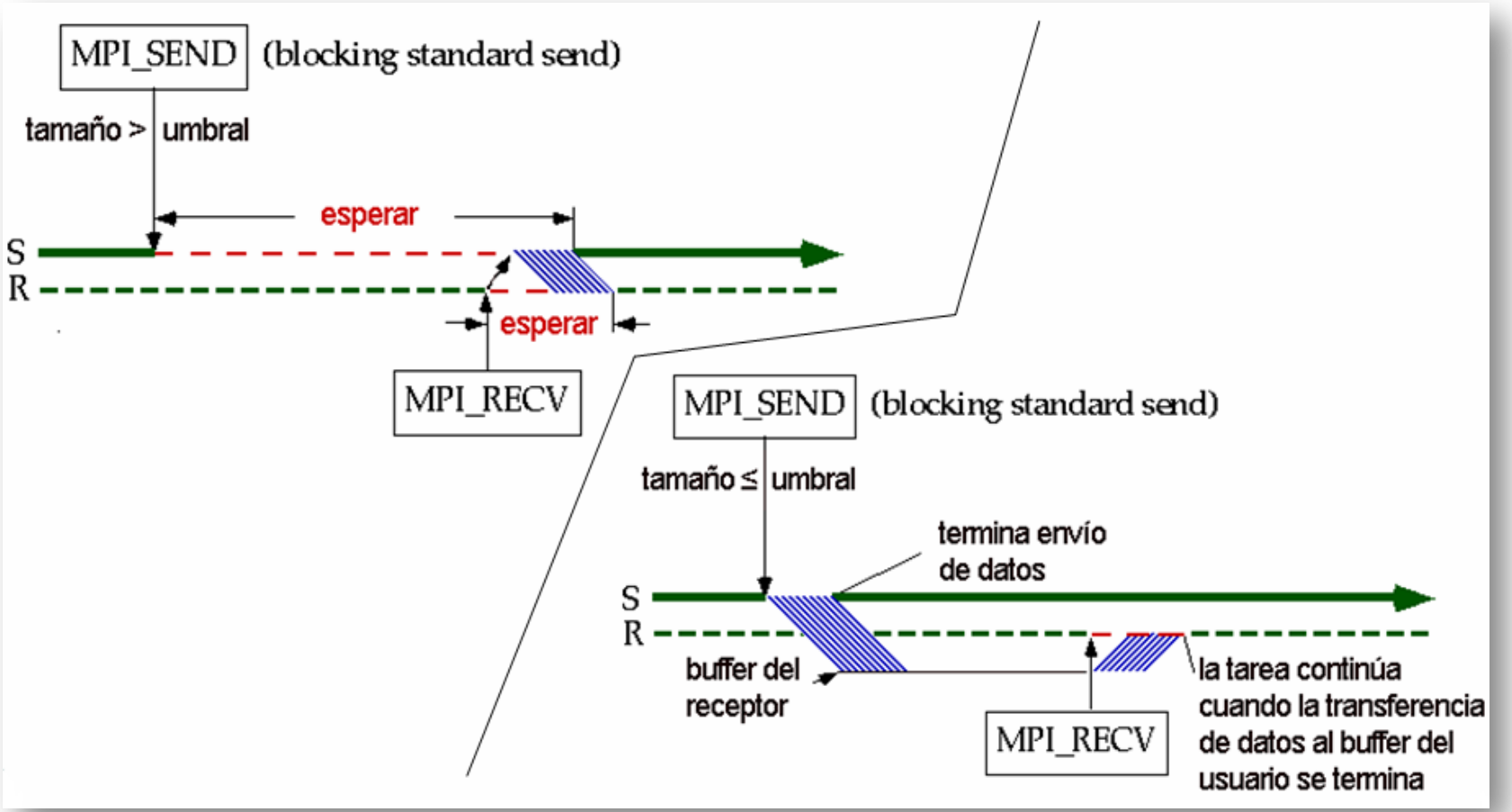
- Multipunto: **1 o N emisores- 1 o N receptores**
  - Bcast, Reduce, Barrier, Gather, Scatter, AllReduce, AllAllgather, …

Ejemplo de comunicación



- Petición de transmisión
- Aceptación de transmisión
- Envío de datos (de buffer de usuario a buffer de usuario

MPI_Send, MPI_Recv (I)



MPI_Send, MPI_Recv (II)

MPI_Send(buf, count, datatype, dest, tag, comm)

MPI_Recv(buf, count, datatype, source, tag, comm)

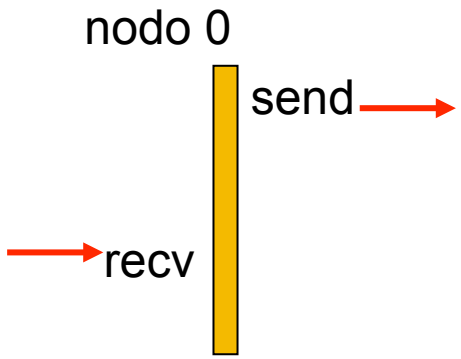
- buf**:** Dirección donde comienza el mensaje.
- count**:** número de elementos del mensaje.
- datatype**:** tipo del mensaje.
- dest/source**:** posición relativa del proceso fuente/ destino dentro del comunicador.
  MPI_ANY_SOURCE: permite recibir mensaje de cualquier fuente

Ejemplo de Send y Recv (I)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
                int numtasks, rank, dest, source, rc, tag=1;
                char inmsg, outmsg='x'; MPI_Status Stat;
                MPI_Init(&argc,&argv);
                MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
                MPI_Comm_rank(MPI_COMM_WORLD, &rank);

                if (rank == 0) {
                    outmnsg="r";
                    dest = 1;
                    printf("Soy %d envio a %d\n",rank,dest);
                    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
                }
                 else
                {
                    source=0;
                    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,&Stat);
                    printf("Soy %d recibo de %d la siguiente letra %c\n",rank,source,inmsg);
                 }
MPI_Finalize();
}
```

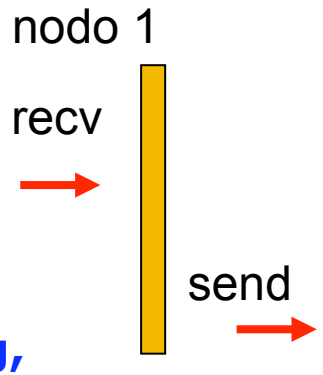
Ejemplo de Send y Recv (II)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
    int numtasks, rank, dest, source, rc, tag=1;
    char inmsg, outmsg='x'; MPI_Status Stat;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        dest = 1;
        source = numtasks-1;
        printf("Soy %d envio a %d\n",rank,dest);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
    MPI_COMM_WORLD);
        printf("Soy %d espero a %d\n",rank,source);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
    MPI_COMM_WORLD,&Stat);
          printf("Soy %d recibo de %d\n",rank,source);
    }
```

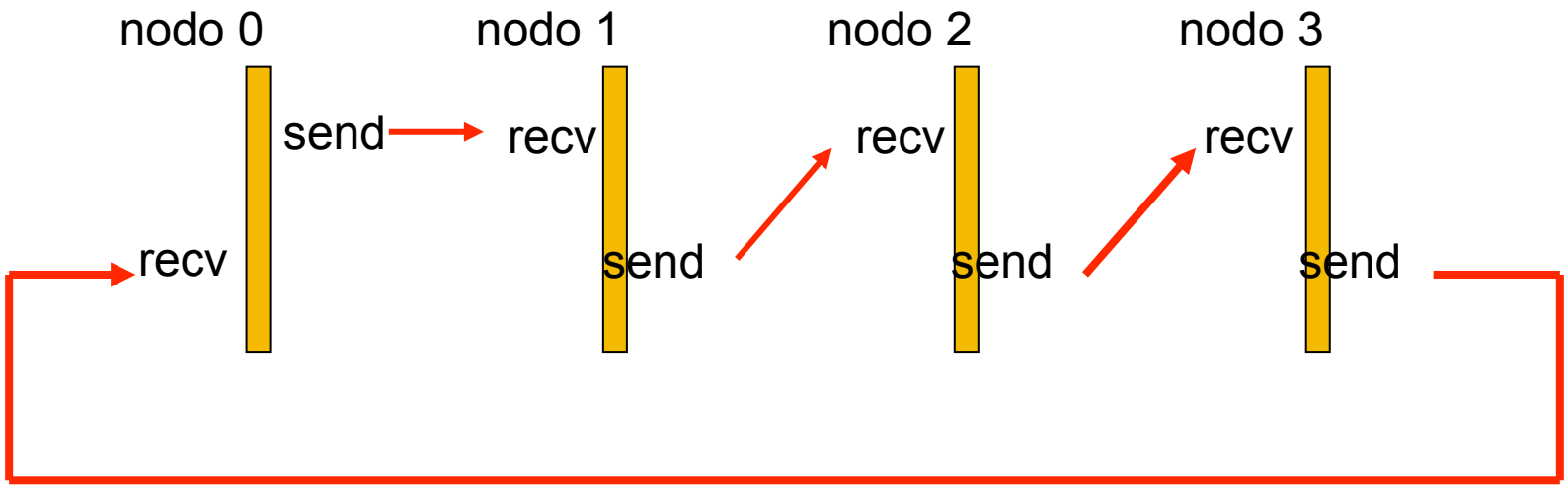


Ejemplo de Send y Recv (III)

```
else {
    dest = rank+1;
    if(dest == numtasks){
            dest = 0;
    }
    source = rank - 1;
    printf("Soy %d espero a %d\n",rank,source);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
    MPI_COMM_WORLD, &Stat);
    printf("Soy %d recibo de %d\n",rank,source);
    printf("Soy %d envio a %d\n",rank,dest);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
    MPI_COMM_WORLD);
    }

MPI_Finalize();
}
```

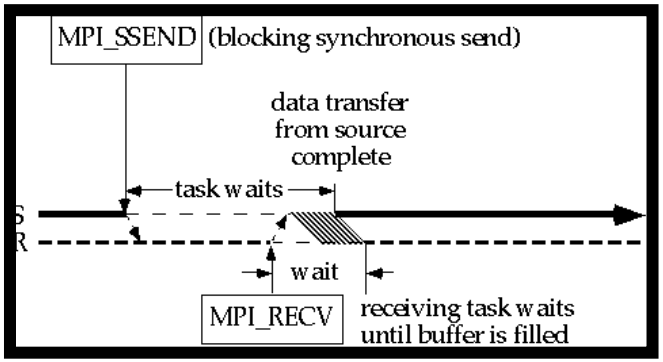


Ejemplo de Send y Recv (IIII)

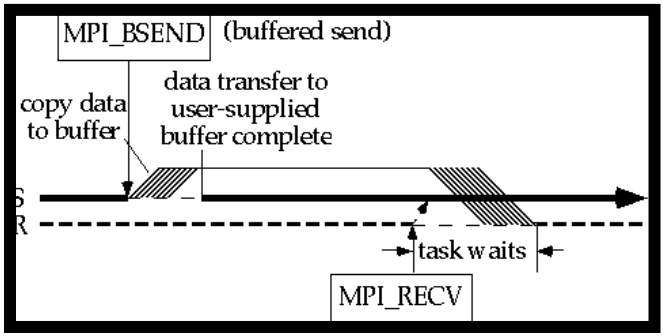


MPI_Send

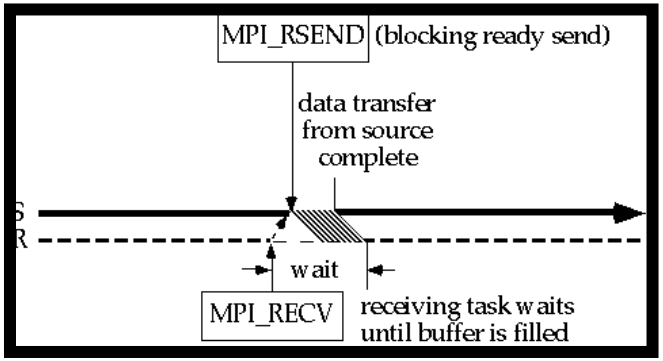
Síncrono:MPI_Ssend(.....)
La operación se da por terminada sólo cuando el mensaje ha sido recibido en destino



Buffered: MPI_Bsend(.....)
Cuando se hace un envío con buffer se guarda inmediatamente, en un buffer al efecto en el emisor, una copia del mensaje. La operación se da por completa en cuanto se ha efectuado esta copia. Si no hay espacio en el buffer, el envío fracasa.

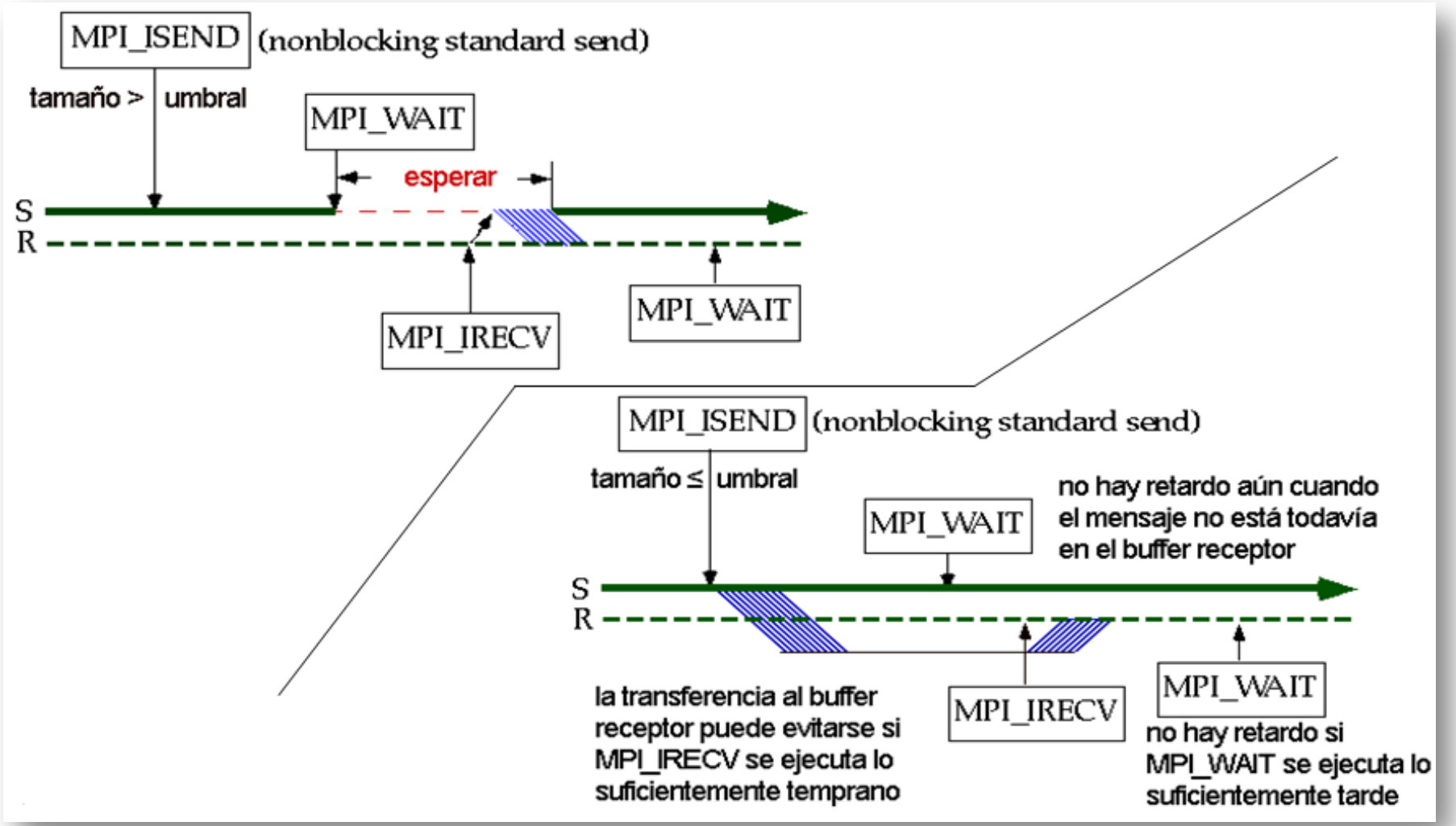


Ready: MPI_Rsend(.....)
Sólo se puede hacer si antes el otro extremo está preparado para una recepción inmediata. No hay copias adicionales del mensaje (como en el caso del modo con buffer), y tampoco podemos confiar en bloquearnos hasta que el receptor esté preparado.



.

MPI_ISend, MPI_Irecv (I)



MPI_ISend, MPI_Irecv (II)

MPI_Isend(buf, count, datatype, dest, tag, comm, request)

MPI_Irecv(buf, count, datatype, source, tag, comm, request)

- request**:** Se usa para saber si la operación ha acabado.
- MPI_Wait()**:** vuelve si la operación se ha completado, espera hasta que se complete.
- MPI_Test()**:** devuelve una bandera que indica si la operación se ha completado.

Llamadas colectivas (I)

MPI_Barrier( )

MPI_Bcast( )

MPI_Gather( )

MPI_Scatter( )

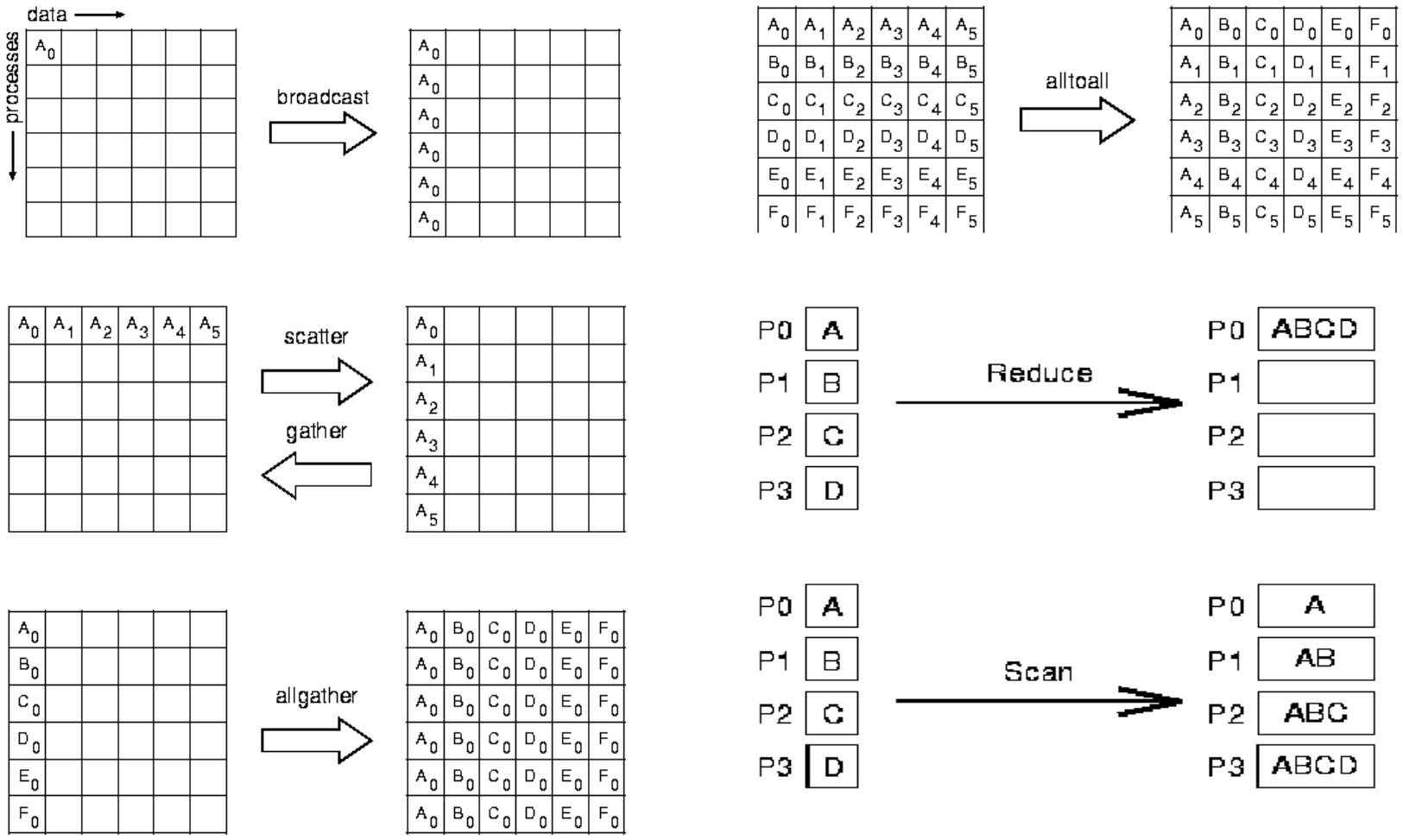
MPI_Alltoall( )

MPI_Reduce( )

MPI_Reduce_scatter( )

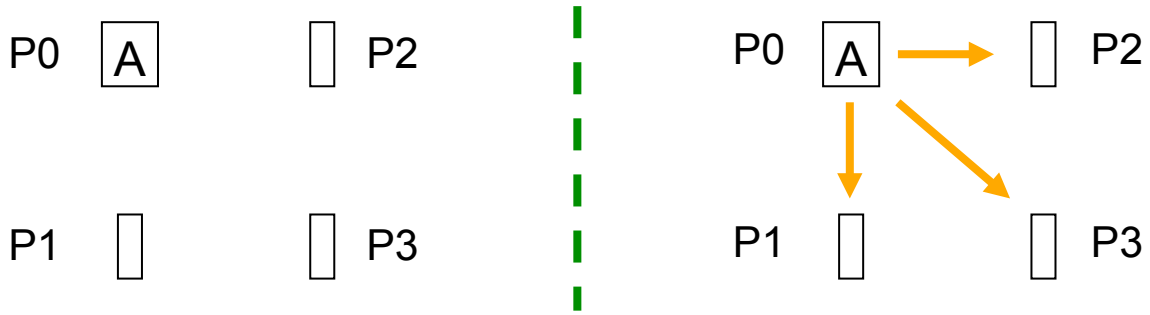
MPI_Scan( )

Llamadas colectivas (II)



Broadcast

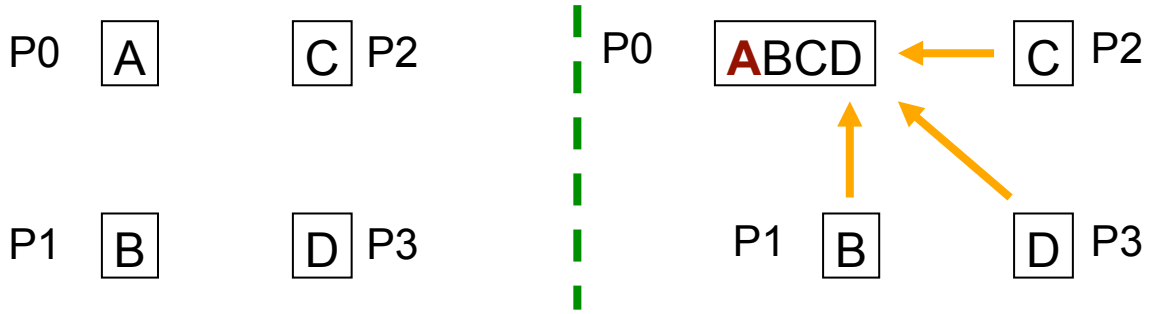
- BROADCAST: envío de datos desde un proceso (root) a todos los demás.



- MPI_Bcast (&message, count, datatype, root, comm);

Gather

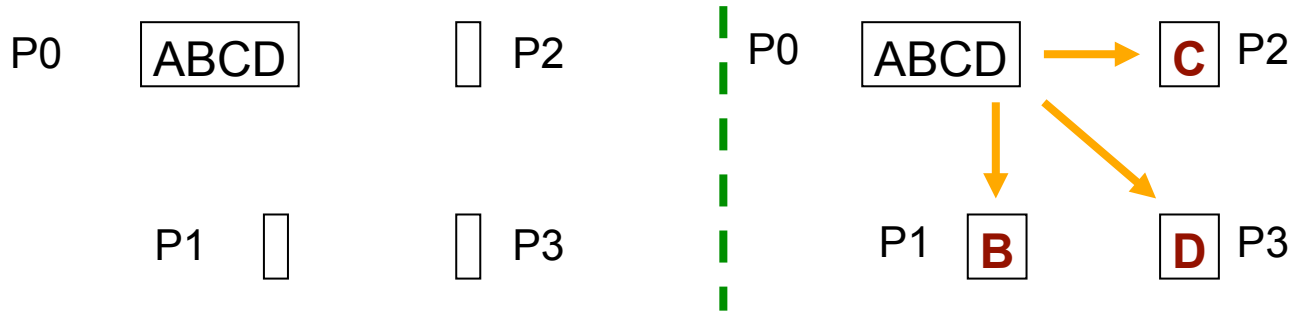
- GATHER: recolección de datos de todos los procesos en uno de ellos (orden estricto pid).



MPI_Gather (&send_data, send_count, send_type,&recv_data, recv_count, recv_type,root, comm);

Scatter

- SCATTER: distribución de datos de un proceso entre todos los demás.



- MPI_Scatter (&send_data, end_count, send_type,&recv_data, recv_count, recv_type,root, comm);

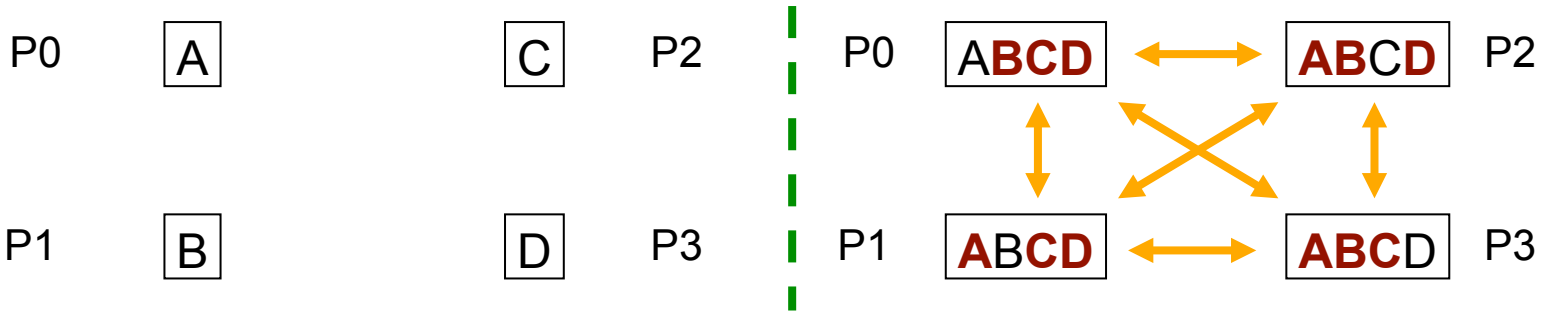
Scatterv

- MPI_Scatterv() se usa si los datos a enviar:
  - no están almacenados de forma consecutiva en memoria, o
  - los bloques a enviar a cada proceso no son todos del mismo tamaño

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

Allgather

- ALLGATHER: gather de todos a todos



- MPI_Allgather(&send_data,send_count, send_type,&recv_buf, recv_count, recv_type,comm);

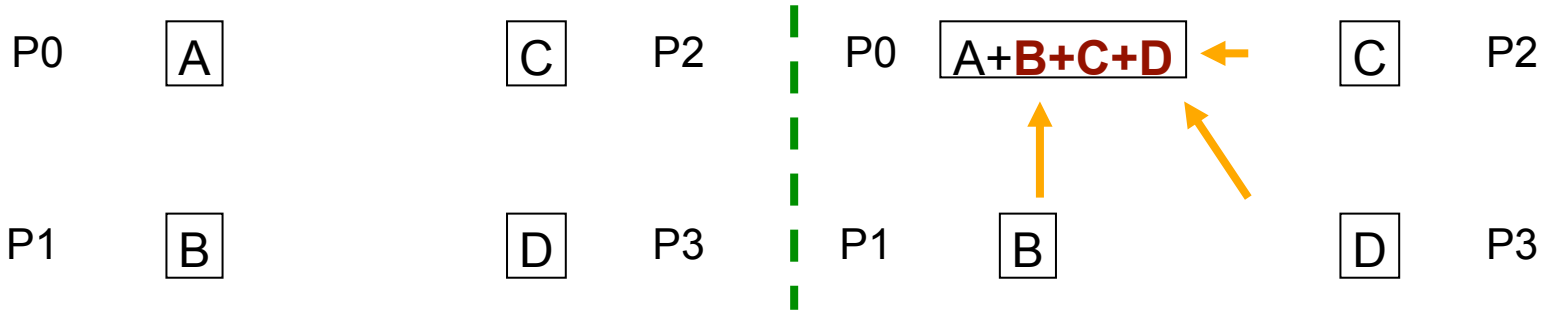
Barrier

- BARRIER: sincronización global entre los procesadores del comunicador.

- MPI_Barrier (comm);
  - La función se bloquea hasta que todos los procesos del comunicador la ejecutan.

Reduce

- REDUCE: una operación de reducción con los datos de cada procesador, dejando el resultado en uno de ellos (root)



MPI_Reduce (&operand, &result, count, datatype,operator, root, comm);

Operadores

MPI Operator	Operation
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	max value and location
MPI_MINLOC	min value and location

Ejemplo MPI_Reduce

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

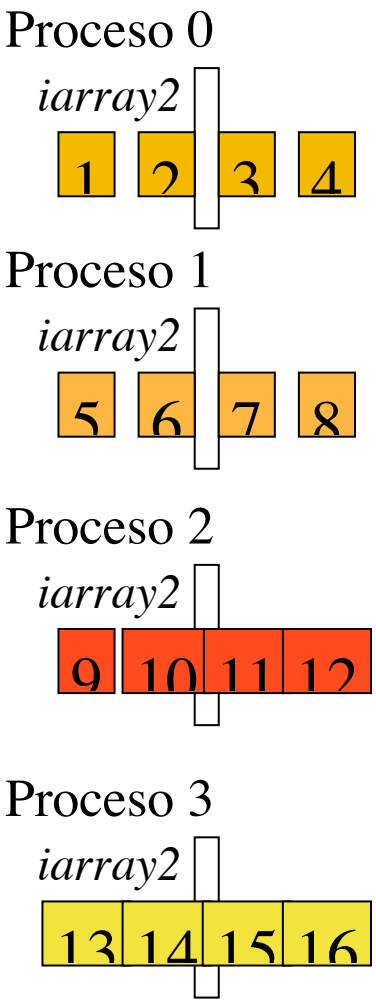
```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char* argv[])
{
  int my_rank;
  int num_procs;
  int num, prod;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

  num = my_rank+1;
  MPI_Reduce(&num,&prod,1,MPI_INT,MPI_PROD,
    0,MPI_COMM_WORLD);
  if (my_rank==0)
        printf("Hola, soy el proceso %d y %d nodos calculamos el producto:
    %d\n",                          my_rank   ,num_procs,   prod);
  MPI_Finalize();
}
```

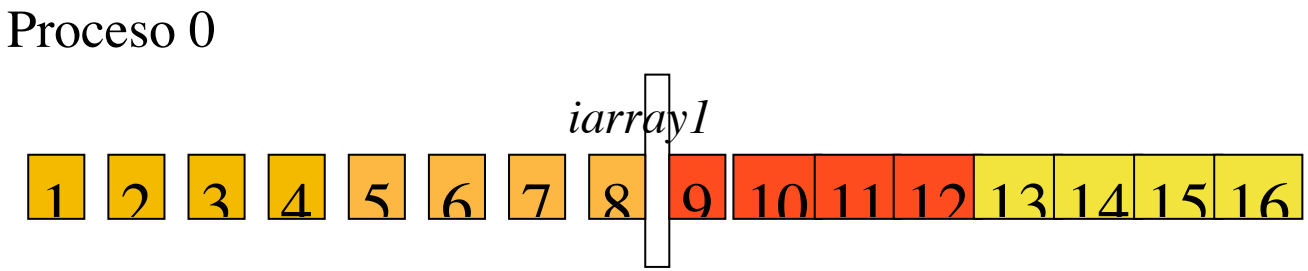
Ejemplo Bcast

```
#include <stdio.h>
 #include <string.h>
 #include "mpi.h"
 main(int argc, char* argv[])
{
     int my_rank;
     int n;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank == 0) {
          printf("Hola, ingresa un número: ");
         scanf("%d",&n);
   }
   MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
   printf("Hola, soy el proceso    %d y el número es el %d
       \n",my_rank,n);
   MPI_Finalize();
}
```

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

Ejemplo uso de MPI_Gather






```
call MPI_GATHER( iarray2, 4, MPI_INTEGER,
                 iarray1,  4, MPI_INTEGER,
                 0, MPI_COMM_WORLD)
```

Ejemplo uso de MPI_GATHER

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

main(int argc, char* argv[])
{     int i,j,k,l;
      int iarray1[16];
      int iarray2[4];
      int myid, numprocs, ierr
      int status(MPI_STATUS_SIZE);

      MPI_INIT( &argc,&argv );
      MPI_COMM_RANK( MPI_COMM_WORLD, myid);
      MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs);

       printf("Soy el nodo %d de %d procesos\n", myid,numproc);
       for (j=0;j<16;i++)
          iarray1[j]=0;
         for(j=0;j<4;j++)
       iarray2[j]=(myid*4)+j;

       MPI_GATHER(iarray2,4,MPI_INTEGER, iarray1,4,MPI_INTEGER,0,MPI_COMM_WORLD);
      if (myid==0)
          for(i=0;i<16;i++)
              printf("%d\n",iarray1[i]);
        MPI_FINALIZE();
      }
```

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

Tipos de Datos en MPI

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_SHOT	**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	

Crear nuevos tipos de datos

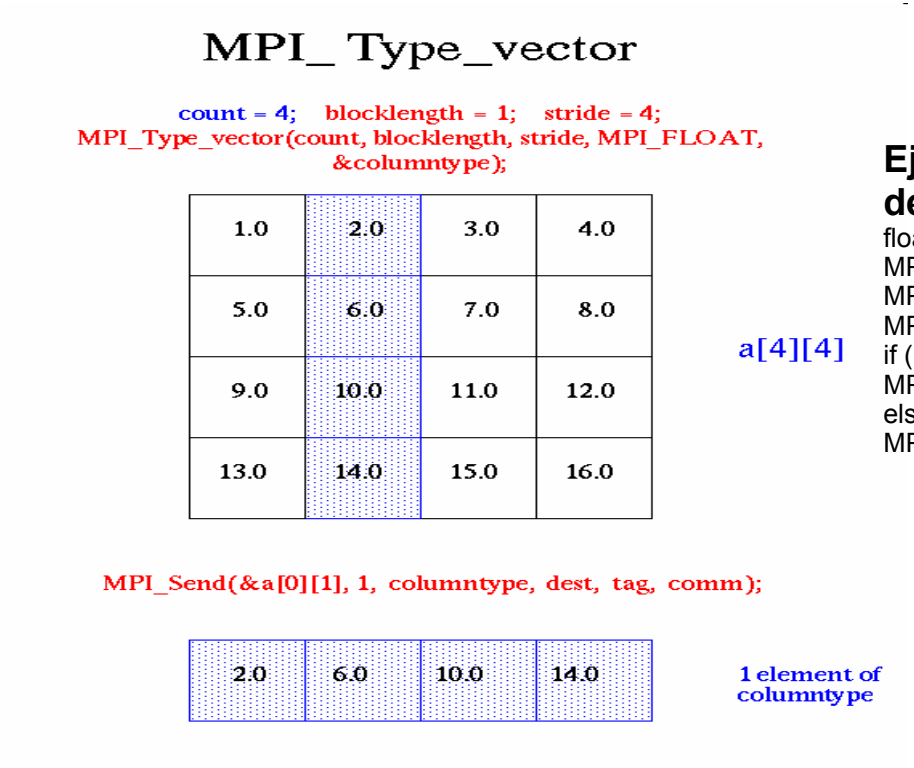
- MPI, nos permite crear nuevos tipos de datos (DataType):
  - n elementos de un vector con stride s
    - MPI_Type_vector → Stride s fijo
    - MPI_Type_contiguous→ Strides s==1
    - MPI_Type_indexed → Stride s variable
  - generales (struct)
  - Empaquetamiento
- MPI_Type_commit

  Hay que llamar a esta rutina antes de poder usar el nuevo tipo de dados que hemos creado.

Tipos de datos derivados

- **MPI_Type_vector**

  Elementos espaciados a intervalos regulares.



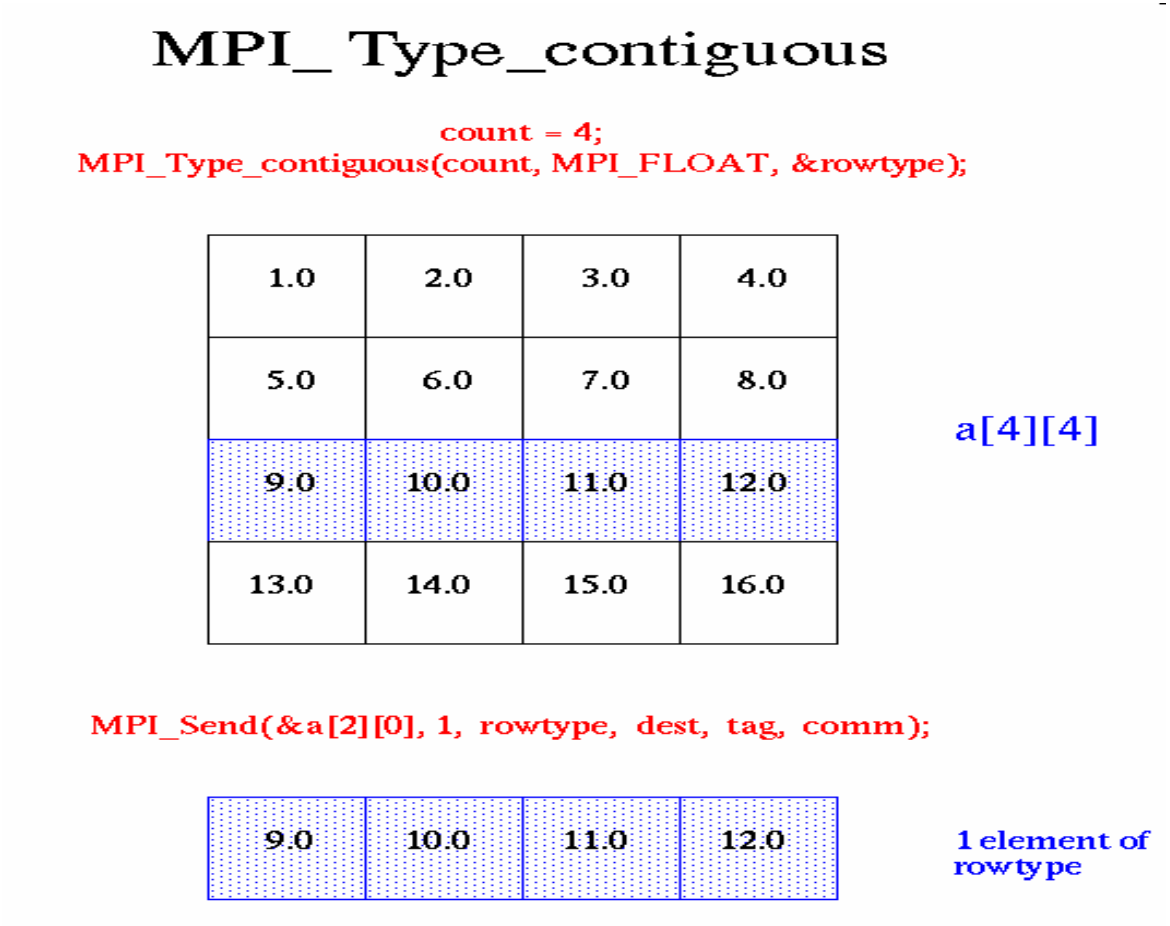
Ejemplo: enviar la columna 2 de la matriz A, de P0 a P1:

```
float a[4][4];
MPI_Datatype columtype;
MPI_Type_vector(4, 1, 4, MPI_FLOAT, columtype);
MPI_Type_commit(&columtype);
if (pid==0)
MPI_Send(&a[0][1], 1, columtype, 1, 0, MPI_COMM_WORLD);
else if (pid==1)
MPI_Recv(&a[0][1], 1, columnatype 0, 0, MPI_COMM_WORLD, &info);
```

Tipos de datos derivados

- **MPI_Type_contiguous**

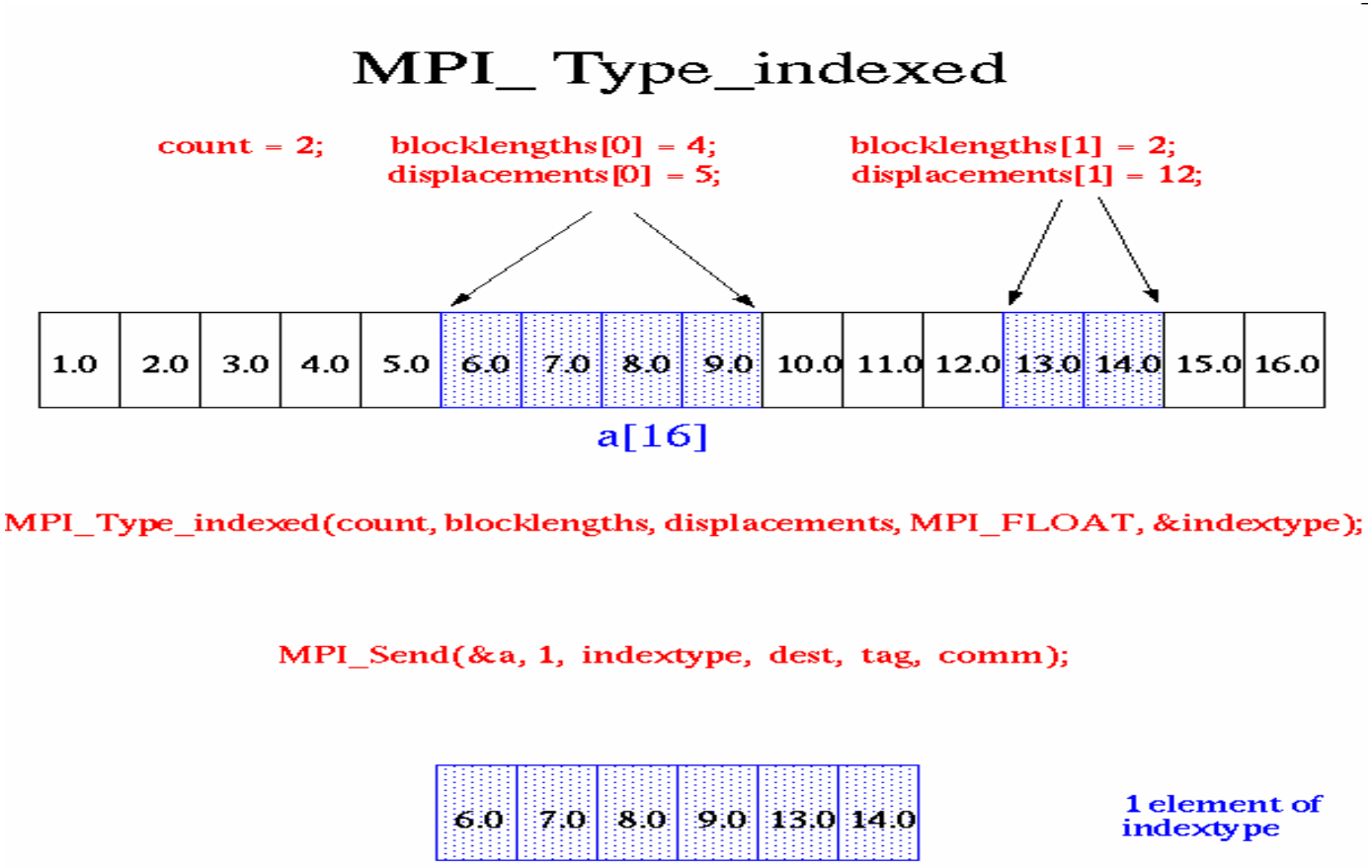
  Colección de elementos de un tipo básico, todos del mismo tamaño y almacenados consecutivamente en memoria.



Tipos de datos derivados

- **MPI_Type_indexed**

  Elementos son entradas arbitrarias de un vector.



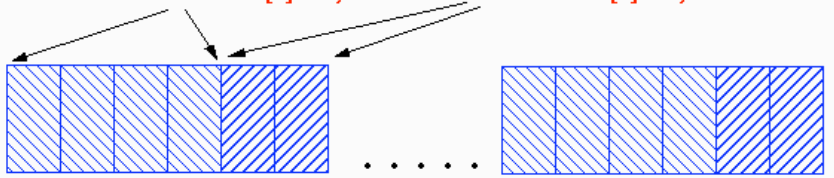
Tipos de datos derivados heterogéneos

El nuevo tipo de dato se forma de acuerdo a una agrupación de tipos de datos

MPI_TYPE_STRUCT(
count,
blockcounts(),
offsets(),
oldtypes,
newtype,
ierr)

MPI_ Type_struct

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2; oldtypes[0] = MPI_FLOAT;   oldtypes[1] = MPI_INT
           offsets[0] = 0;            offsets[1] = 4 * extent;
           blockcounts[0] = 4;        blockcounts[1] = 2;
```



particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

Tipos de datos derivados

- **MPI_Type_Struct** : Nos permite crear un Datatype de con bloques de distinto tipos de datos. Podemos generar un tipo que agrupe datos de tipo float, char y enteros en un mensaje y efectuar un único envío.

```
int MPI_Type_struct(
            int count,
            int blocklens[],
            MPI_Aint indices[],
            MPI_Datatype old_types[],
            MPI_Datatype *newtype )

            blen[0] = 1; indices[0] = 0;                oldtypes[0] = MPI_INT;
            blen[1] = 1; indices[1] = &foo.b - &foo;    oldtypes[1] = MPI_CHAR;
            blen[2] = 1; indices[2] = sizeof(foo);      oldtypes[2] = MPI_UB;
            MPI_Type_struct( 3, blen, indices, oldtypes, &newtype );
```

Empaquetar y desempaquetar

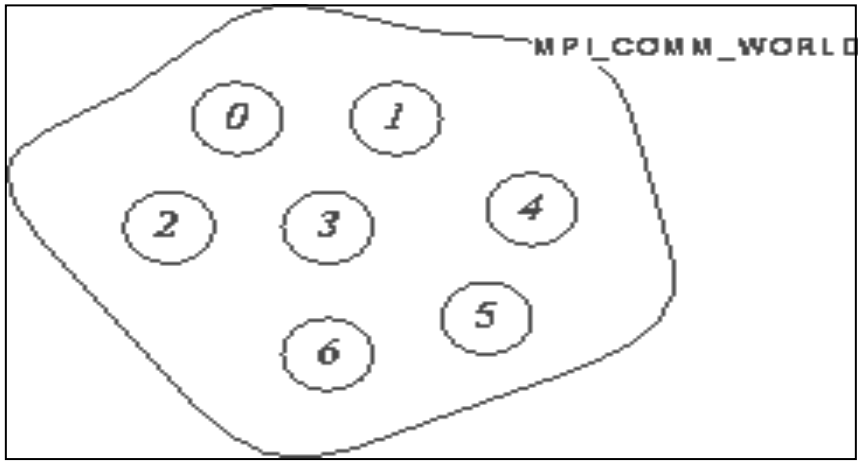
- **MPI_Pack**
  Empaqueta datos en un *buffer* continuo en memoria para ser enviados como un mensaje único.
- **MPI_Unpack**
  Desempaqueta los datos recibidos en un mensaje.

Cuando utilizar cada tipo

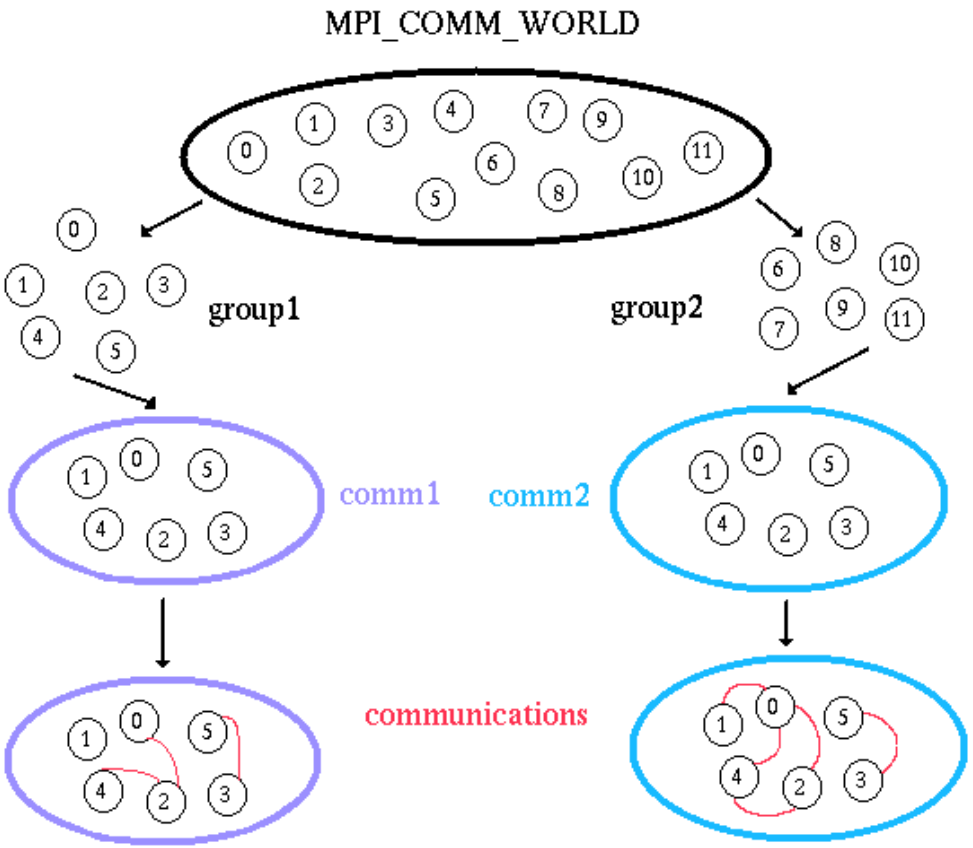
- Empaquetar y desempaquetar se suele utilizar cuando se quieren mandar pocos datos, del mismo tipo y de una sola vez.

- Si los datos a enviar son muchos y no consecutivos, y se envían repetidamente, lo mejor sería definir un tipo específico:
  - datos homogéneos: contiguous, vector, indexed
  - datos heterogéneos: struct

Comunicadores (I)



Comunicadores(I)



- Extraer un *handle* de MPI_COMM_WORLD usando MPI_Comm_group
- Formar nuevos grupo usando MPI_Group_{include, exclude,…}
- Crear un nuevo comunicador para el nuevo grupo con MPI_Comm_create
- Realizar las comunicaciones con las rutinas de paso de mensajes
- Al terminar libere el comunicador con MPI_Comm_free y MPI_Group_free

Comunicadores (II)

- Creación

  MPI_Comm_create(↓Old_Comm, ↓Group_Proc, ↑New_Comm)

  MPI_Comm_dup(↓Comm, ↑New_Comm)

  MPI_Comm_split(↓Old_Comm, ↓split_key, ↑New_Comm)

- Eliminación
  - **MPI_Comm_free(↓↑Comm)**

- Información
  - **MPI_Comm_rank(↓Comm, ↑Rank)**
  - **MPI_Comm_compare(↓Comm1, ↓Comm2, ↑Result)**

Ejemplo usando comunicadores

```
#include "mpi.h";
 MPI_Comm  comm_world, comm_worker;
 MPI_Group  group_world, group_worker;
 int ierr;
 comm_world = MPI_COMM_WORLD;
 MPI_Comm_group(comm_world, &group_world);
 MPI_Group_excl(group_world, 1, 0, &group_worker);
 MPI_Comm_create(comm_world,group_worker,&comm_worker);
 /* ACCIONES */
 MPI_Comm_free(&comm_worker);
```

Grupos (I)

- Creación

MPI_Comm_incl(↓Group, ↓NP, ↓Who[], ↑New_Group)

MPI_Comm_excl(↓Group, ↓NP, ↓Who[], ↑New_Group

MPI_Group_range_incl(↓Group, ↓NP, ↓Rango[][3], ↑New_Group)

MPI_Group_range_excl(↓Group, ↓NP, ↓Rango[][3], ↑New_Group)

MPI_Group_difference(↓Group1, ↓Group2, ↑New_Group)

MPI_Group_intersection(↓Group1, ↓Group2, ↑New_Group)

MPI_Group_union(↓Group1, ↓Group2, ↑New_Group)

Grupos (II)

- Eliminación

  MPI_Comm_free(↓Group)

- Información

  MPI_Comm_group(↓Comm, ↑Group)

  MPI_Group_rank(↓Group, ↑Rank)

  MPI_Group_compare(↓Group1, ↓Group2, ↑Result)

  MPI_Group_translate_ranks(

  ↓Group1, ↓ NP, ↓ Ranks1[], ↓ Group2, ↑Ranks2)

Tiempo

- MPI Wtime()

```
double start, finish, time;
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
…
…
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
time = finish - start;
```

Referencia de MPICH2

- La herramienta ***gcc*** invoca al compilador de C nativo (por ejemplo cc) y provee el path de los ficheros de cabecera, librerías y enlaces implícitos de MPICH2. La librería MPI se enlaza explícitamente:
  - **% gcc –o progama programa.c -lmpich**
- Iniciamos una aplicación MPI invocando al comando ***mpiexec***.
  - Este comando provee la información sobre los nodos e identificadores de proceso necesarios para que los procesos se localicen entre si.

Referencia de MPICH2

- La sintaxis para el comando ***mpiexec*** es la siguiente

  % mpiexec.mpich2 –n<#> programa

  - ***-n <#>*** indica el número de procesos que se crearán.
  - ***Programa*** es el fichero ejecutable (incluido el path para llegar hasta el)

Referencia de MPICH2

- La herramienta ***mpdboot*** inicializa MPICH2 en el cluster especificado.
  - **% mpdboot –n <num máquinas> -f <boot schema>**
  - La opción –*v* permite imprimir mensajes antes de cada paso dado.
- Para comprobar que MPICH2 está corriendo normalmente en el cluster:
  - **% mpdtrace**

Referencia de MPICH2

- Podemos monitorizar cierta información de los procesos MPI, en tiempo de ejecución, mediante los comandos:
  - **mpdlistjobs** visualiza información sobre los procesos MPI, tal como los ranks (identificadores) de la fuente y el destino, el comunicador, y la función que se esta ejecutando

Referencia de MPICH2

- Mediante el comando ***mpdkilljob***, podemos eliminar todos los procesos y mensajes sin necesidad de abortar MPI.
  - **% mpdkilljob <job id>**
- Por último, la orden ***mpdallexit*** elimina todos los procesos y demonios de MPICH2 en las máquinas que estamos utilizando.
  - **% mpdallexit**

Seguridad

- Para ejecutar las prácticas hacer antes:
  - *cd ~/.ssh*
  - *ssh-keygen –t dsa*
  - *ssh-keygen –t rsa*
  - *cat *.pub > authorized_keys*

Compilando y ejecutando

Compilando

- *mpicc –o hola hola.c*

Ejecutando

- Fichero con máquinas: *machines*
- *mpdboot -n 2 -f machines*
- *mpiexec –n 2 hola*

Enlaces

- **http://mpi.deino.net/mpi_functions/**
- **http://www.mpi-forum.org**
- **http://www.mcs.anl.gov/mpi/mpich/**
- http://www.lam-mpi.org/
- http://www.arcos.inf.uc3m.es/~ii_ac2/03-04/ejemplos_mpi.zip
- http://www.mhpcc.edu/training/workshop/mpi/MAIN.html
- http://www.urjc.es/cat/hidra/manuales/MPI/tutorial_MPI.pdf
- http://www.cnb.uam.es/~carazo/practica_mpi.html
- http://numerix.us.es/pers/denk/_parallel/
- http://www.ldc.usb.ve/~ibanez/docencia/MPI/
- http://www.tc.cornell.edu/services/edu/topics/parallelvw.asp
- http://web.tiscali.it/Moncada/documenti/mpi.ppt
- http://www-copa.dsic.upv.es/docencia/iblanque/lcp/
- http://www.arcos.inf.uc3m.es/~mimpi/
- http://www.df.uba.ar/users/marcelo/
- http://www.dirinfo.unsl.edu.ar/~prgparal/Teorias/
- http://www.cecalc.ula.ve/documentacion/tutoriales/mpi/mpi.ps