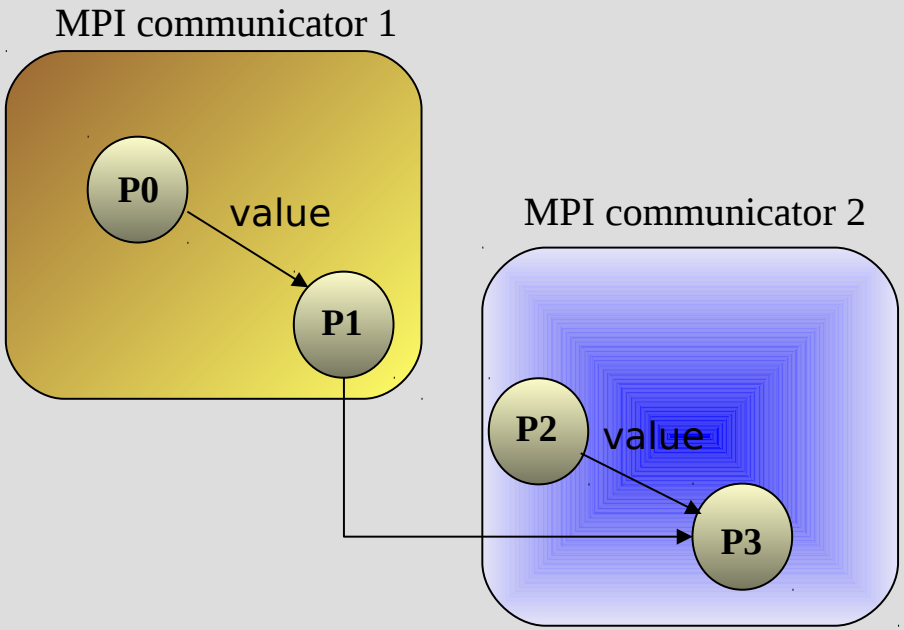
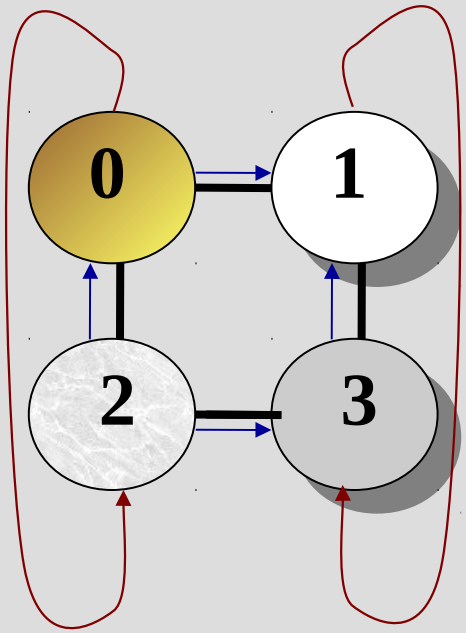
Introducción a la Interfaz de paso de mensajes (MPI)





José Miguel Mantas Ruiz

Depto. de Lenguajes y Sistemas Informáticos
Universidad de Granada

Contenido

1. Message Passing Interface (MPI)
2. Funciones MPI básicas
3. Envío y recepción simultánea
4. Operaciones de comunicación colectiva
5. Comunicación No bloqueante
6. Comunicadores
7. Topologías Cartesianas
8. Tipos de datos derivados y Empaquetado
9. Referencias útiles

1. Message Passing Interface (MPI)

- Funciones para paso de mensajes y operaciones. complementarias (>120 rutinas con numerosos parámetros y variantes).
- "Interfaz" estándar para C y Fortran con diversas implementaciones.
- Objetivo diseño: paso de mensajes portable y fácil de usar.
- Basta con un pequeño subconjunto de MPI.
- Modelo de Programación en MPI:
  - Computación ⇒ Número fijado de procesos se comunican mediante llamadas a funciones de envío y recepción de mensajes.
  - Modelo básico: SPMD
  - Extensión: cada proceso puede ejecutar diferentes programas → MPMD
- Creación e inicialización de procesos
  - No definido en el estándar: depende implementación.
  - En mpich: mpirun prog1 –machinefile maquinas –np 4
    - Comienza 4 copias del ejecutable "prog1".
    - La asignación de procesos a máquinas se define en el archivo "maquinas".

1. Message Passing Interface (MPI)

1.1. Asuntos de implementación

#include "mpi.h“

Define ctes, tipos de datos y los prototipos de las funciones MPI

Funciones devuelven código de error

MPI_SUCCESS ⇒ Ejecución correcta

MPI_Status: estructura con 2 campos:

status.MPI_SOURCE: proceso fuente

status.MPI_TAG: etiqueta del mensaje.

Tipos de datos MPI

MPI_CHAR, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, etc.

Comunicador= Grupos de procesos + contexto

2. Funciones MPI básicas

MPI_INIT :	Inicializa entorno de ejecución MPI.
MPI_FINALIZE :	Finaliza entorno de ejecución MPI.
MPI_COMM_SIZE :	Determina nº procesos del comunicador.
MPI_COMM_RANK:	Determina id. proceso en el comunicador.
MPI_SEND :	Envío básico mensaje.
MPI_RECV :	Recepción básica mensaje.

2.1. Iniciando y finalizando MPI

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

- Llamado antes de cualquier otra función MPI
- Llamar más de una vez durante ejecución ⇒ Error
- Argumentos argc, argv: Argumentos de la línea de orden del progr.

int MPI_Finalize ( )

- Llamado al fin de la computación: Tareas de limpieza para finalizar entorno de ejecución

2. Funciones MPI básicas

2.2. Introducción a los Comunicadores

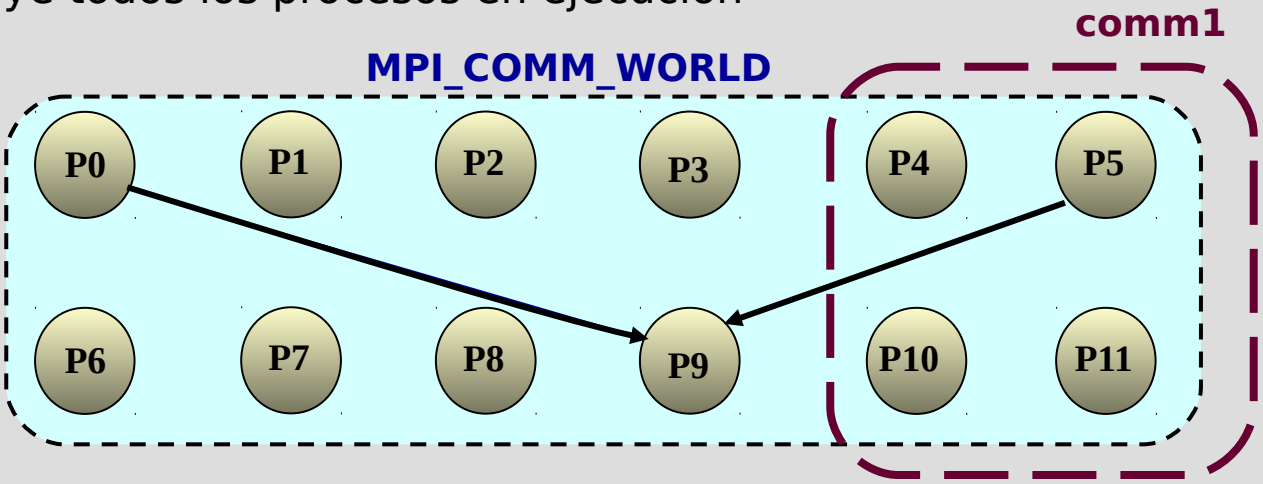
Comunicador = variable de tipo MPI_Comm = Grupo_procs + Contexto

- **Grupo de procesos**: Subconjunto de procesos
- **Contexto de comunicación**: Ámbito de paso de mensajes en el que se comunican dichos procesos. Un mensaje enviado en un contexto sólo se conoce en ese contexto ⇒ elemento del "sobre" del mensaje.

Argumento en funciones de transferencia.

MPI_COMM_WORLD : Comunicador MPI por defecto

Incluye todos los procesos en ejecución

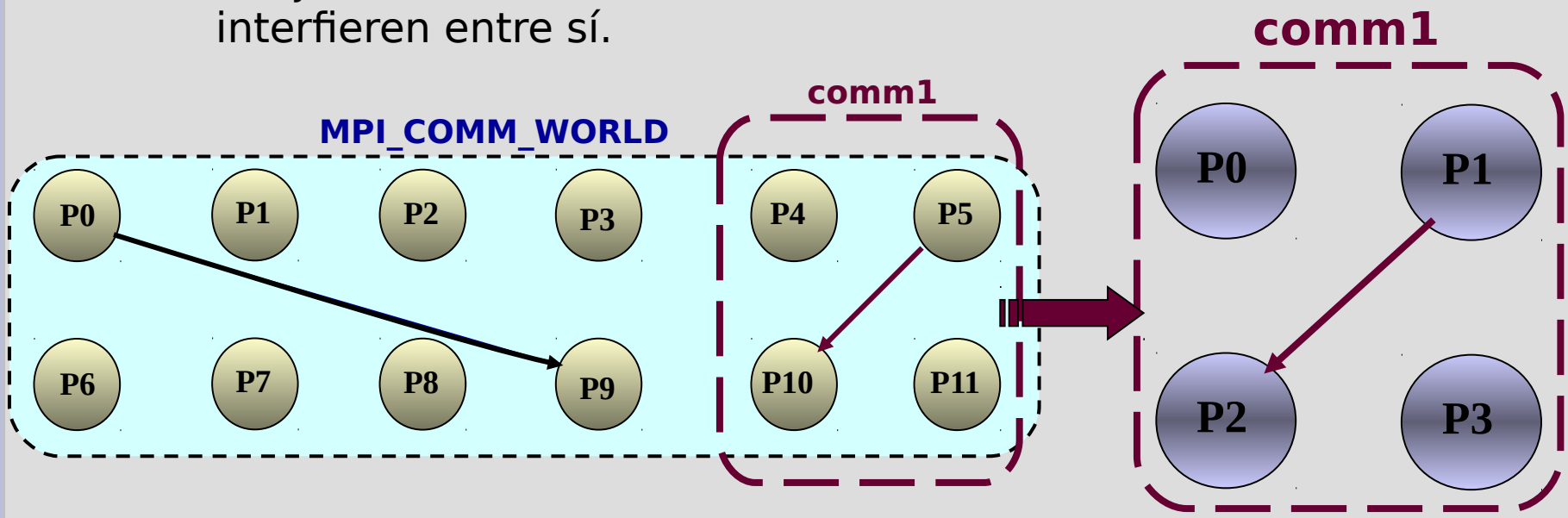


2. Funciones MPI básicas

2.2. Introducción a los Comunicadores (cont.)

Identificación unívoca de procesos participantes en comunicador

- Un proceso puede pertenecer a diferentes comunicadores
- Cada proceso tiene un identificador: desde 0 a size_comm-1
- Mensajes destinados a diferentes contextos de comunicación no interfieren entre sí.



2. Funciones MPI básicas

2.3. Obteniendo Información

int MPI_Comm_size (MPI_Comm comm, int *size)

size= nº de procs que pertenecen al comunicador comm.

Ej.: MPI_Comm_size (MPI_COMM_WORLD, &size) ⇒ size = nº total de procs.

int MPI_Comm_rank (MPI_Comm comm, int *rank)

rank= Identificador del proceso llamador en comm.

```
#include <stdio.h>
#include "mpi.h"
int main(int  argc, char **argv ) {
int rank, size;
MPI_Init( &argc, &argv );
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
printf( "Hello world from process %d of %d\n", rank, size );
MPI_Finalize( ); return 0;}
```

```
% mpicc -o helloworld helloworld.c
% mpirun -np 4 helloworld

Hello world from process 0 of 4
Hello world from process 3 of 4
Hello world from process 1 of 4
Hello world from process 2 of 4
```

2. Funciones MPI básicas

2.4. Envío y recepción de mensajes

int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Envía datos almacenados en buffer (count elem. de tipo datatype) apuntado por buf al proc. dest con la etiqueta tag (entero >0) dentro del comunicador comm.

Existen implementaciones bloqueantes y no bloqueantes

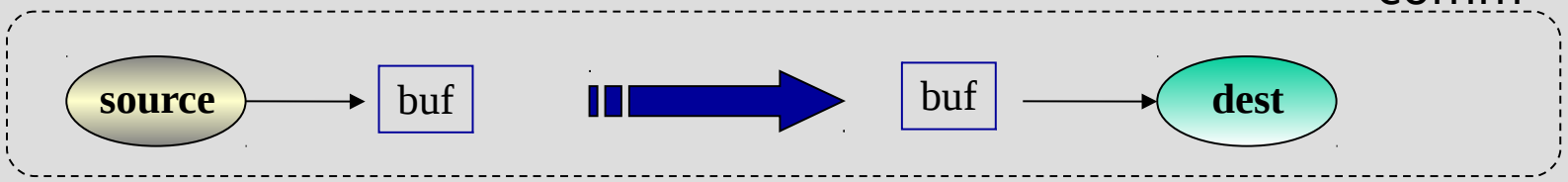
int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Recibe mensaje de proceso source dentro del comunicador comm.

Semántica bloqueante

Sólo se recibe un mensaje enviado desde source con etiqueta tag pero existen argumentos comodín: MPI_ANY_SOURCE, MPI_ANY_TAG.

Mensaje es almacenado en posiciones continuas desde buf.



2. Funciones MPI básicas

2.4. Envío y recepción de mensajes (cont.)

Argumentos count y datatype: especifican la longitud del buffer.

Objeto status → Estructura con campos MPI_SOURCE y MPI_TAG.

Permite obtener información sobre el mensaje recibido

Obtener tamaño del mensaje recibido: Función MPI_Get_count

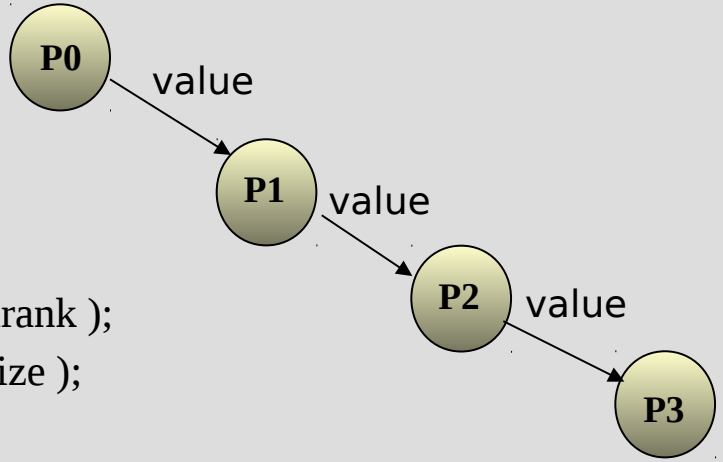
int MPI_Get_count(MPI_Status *status, MPI_Datatype dtype, int *count)

Ejemplo: Programa para dos procesos

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (rank == 0) { value=100;
        MPI_Send (&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );
else    MPI_Recv ( &value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
MPI_Finalize( );
```

2. Funciones MPI básicas

2.4. Envío y recepción de mensajes. Ejemplo



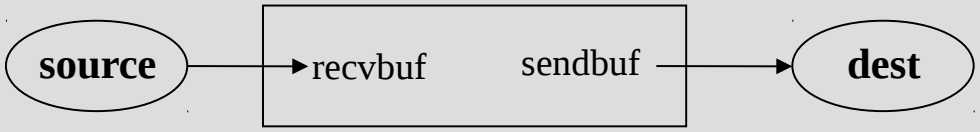

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv); {
int rank, value, size; MPI_Status status;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
do {
 if (rank == 0) { scanf( "%d", &value );
          MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD ); }
 else { MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status );
        if (rank < size - 1) MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD )}
 printf( "Process %d got %d\n", rank, value ); }

while (value >= 0); MPI_Finalize( ); return 0; }
```

3. Envío y recepción simultánea

int MPI_Sendrecv(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)

- Envía un mensaje a dest y simultáneamente los recibe de source
- Muy útil para patrones de comunicación circulares
- Combina argumentos de MPI_Send y MPI_Recv.
  - Los buffers deben ser disjuntos, y se permite que source=dest



int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

- Igual que MPI_Sendrecv pero con un único buffer.
- Los datos recibidos reemplazan a los enviados.
- Tanto send como receive tratan con datos del mismo tipo.

4. Operaciones de comunicación colectiva

Tienen un comunicador como argumento: define el grupo afectado.

Participan todos los procesos del comunicador:

La deben llamar todos con los mismos parámetros.

MPI_Barrier: Sincroniza todos los procesos.

MPI_Broadcast: Envía un dato de un proc. al resto.

MPI_Gather: Recolecta datos de todos los procesos a uno.

MPI_Scatter: Reparte datos de un proceso a todos.

MPI_Reduce: Realiza operaciones simples sobre datos distribuidos.

MPI_Scan: Operación de reducción de prefijo.

4.1. Sincronización de barrera

int MPI_Barrier (MPI_Comm comm)

Bloquea todos los procesos de comm hasta que todos la invoquen.

4. Operaciones de comunicación colectiva

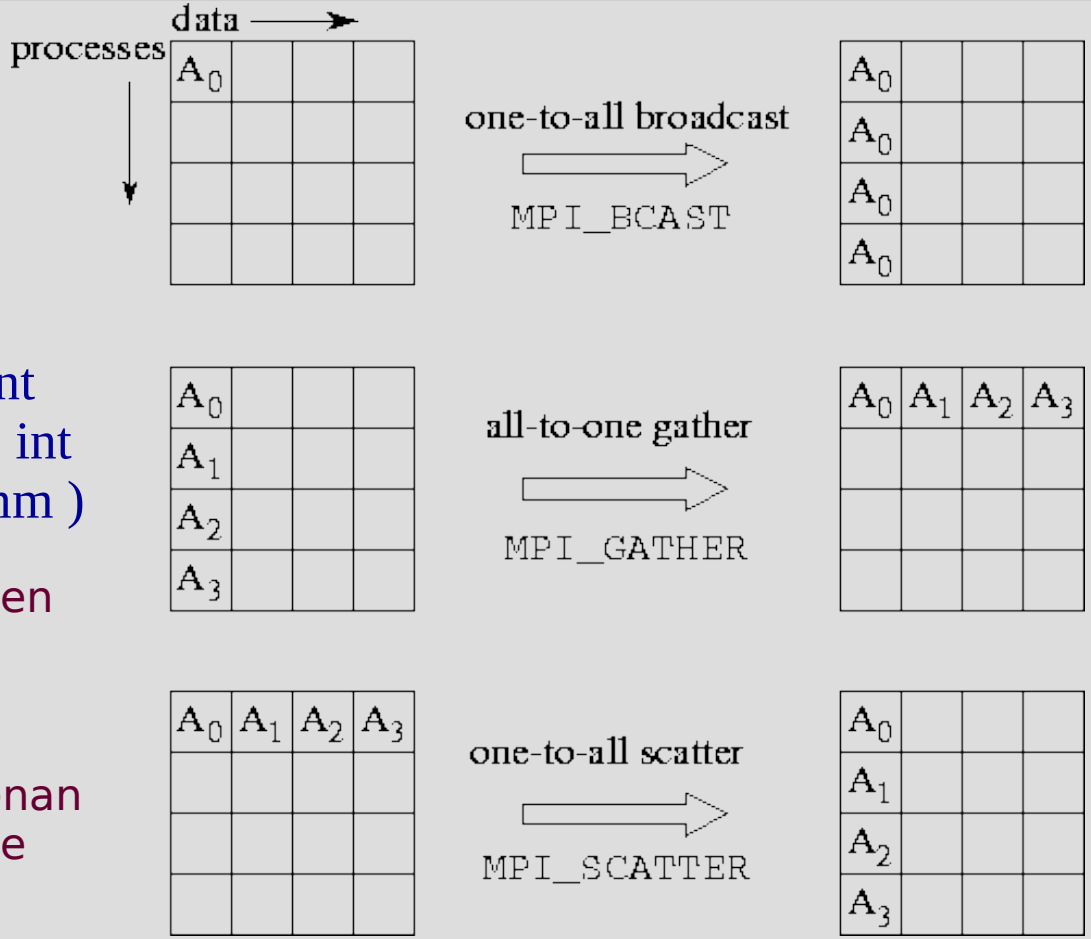
4.2. Movimiento de datos

Todos los procesos interactúan con un proceso distinguido (root) para difundir, recolectar o repartir datos

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

Envía datos almacenados en buffer (count elem. de tipo datatype) en root al resto.

Datos recibidos se almacenan en buffer, count y datatype deben coincidir en todos.



4. Operaciones de comunicación colectiva

4.2. Movimiento de datos (cont.)

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

- Cada proceso envía datos almacenados en el array sendbuf a root.
- Proc root recibe size(comm) buffers dispuestos consecutivamente en recvbuf.
- Argumentos sendtype y sencount→ mismos valores en todos los procs.
- recvcount=nº items recibidos de cada proc ⇒ recvcount=sendcount, si sendtype coincide con recvtype.
- Información sobre buffer de recepción sólo tiene sentido en root.

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

- source envía una parte diferente de sendbuf al recvbuf de cada proc.
- Todos con mismos valores de sendcount, sendtype, recvbuf, recvcount, recvtype, root y comm.
- sendcount=nº items enviados por cada proc

4. Operaciones de comunicación colectiva

4.3. Operaciones de Reducción Global

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

Combina los elementos almacenados en sendbuf de cada proc, usando operación op. Devuelve resultado en recvbuf de root.

Args sendbuf y recvbuf deben tener count items de tipo datatype.

Todos deben proporcionar array recvbuf

Si count>1 ⇒ la operación se aplica elemento a elemento.

Args count, datatype, op, root, comm deben ser idénticos en todos.

Operaciones predefinidas

MPI_MAX (máximo), MPI_MIN (mínimo), MPI_SUM (suma), MPI_PROD (producto), MPI_LAND (AND lógico), MPI_BAND (AND bit a bit), MPI_LOR (OR lógico), PI_BOR (OR bit a bit), MPI_LXOR (XOR lógico), MPI_BXOR (XOR bit a bit).

Es posible definir otras operaciones: MPI_OP_CREATE .

4. Operaciones de comunicación colectiva

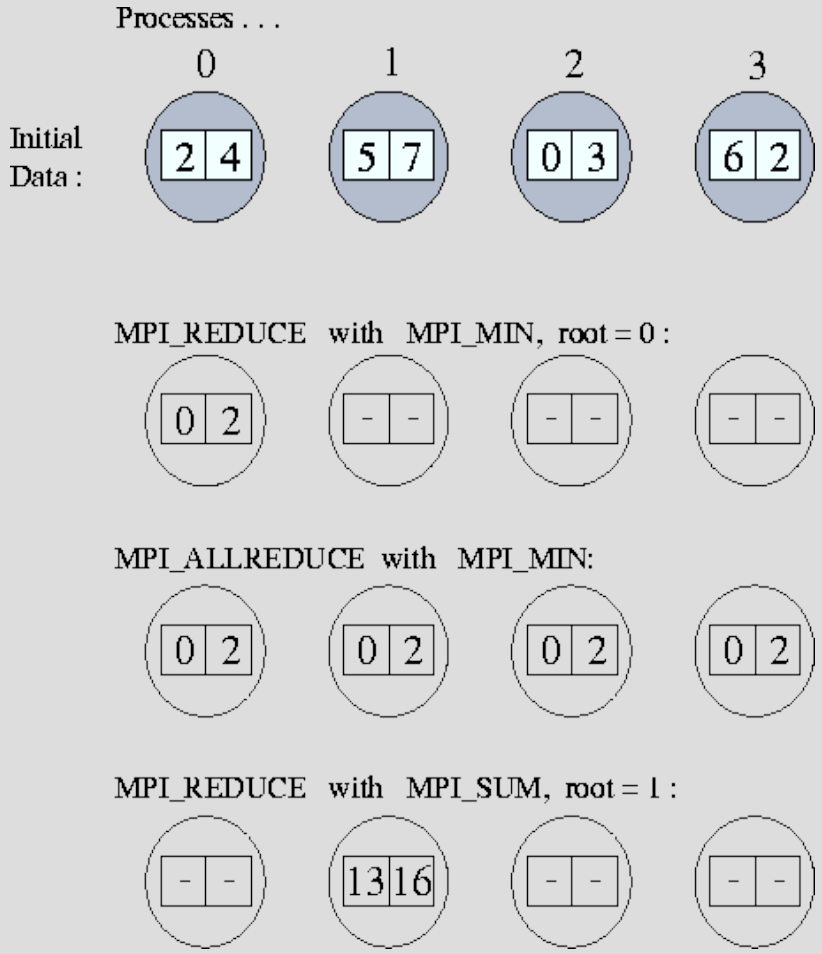
4.3. Operaciones de Reducción Global (cont.)

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

- El resultado es replicado ⇒
  - Argumento root no es necesario

int MPI_Scan (….)

- Argumentos de MPI_Allreduce.
- Calcula reducciones parciales en cada proc.
- Proc i calcula la reducción aplicada a los procesos 0, ...,i.



4. Operaciones de comunicación colectiva

4.4. Ejemplos

Difunde 20 reales de proceso 0 al resto en comunicador comm

```
double vector[20];
MPI_Bcast ( vector, 20, MPI_DOUBLE, 0, comm);
```

Recoge 20 reales de cada proceso en el proceso 0

```
MPI_Comm comm= MPI_COMM_WORLD;
int gsize; double invec[20], *outvec;
MPI_Comm_size( comm, &gsize);
outvec = (double *)malloc(gsize*20*sizeof(double));
MPI_Gather (invec, 20, MPI_DOUBLE, outvec, 20, MPI_DOUBLE, 0, comm);
```

Inversa anterior: Reparte 20 reales desde proc 0

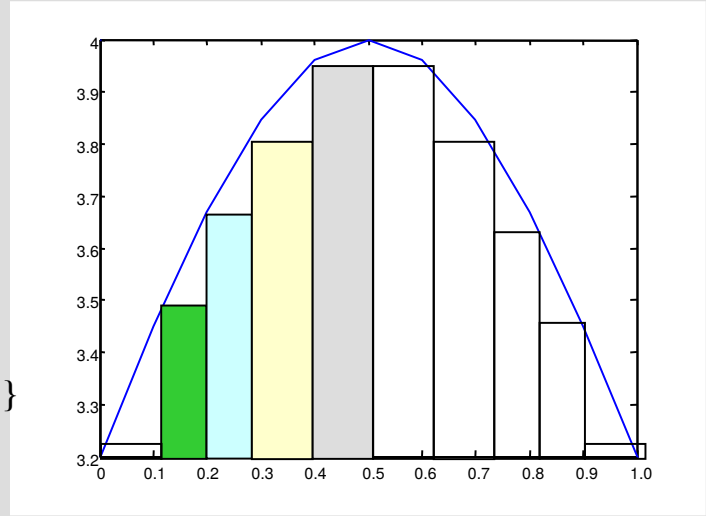
```
MPI_Scatter( outvec, 20, MPI_DOUBLE, invec, 20, MPI_DOUBLE, 0, comm);
```

4. Operaciones de comunicación colectiva

4.4. Ejemplos. Cálculo de π

$$\pi = \int_0^1 \frac{4}{1+x^2}$$

```
#include "mpi.h"
#include <math.h>
int main(int argc, char **argv) {
int n, myid, numprocs, i;   double mypi, pi, h, sum, x;
MPI_Init (&argc,&argv);
MPI_Comm_size (MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank (MPI_COMM_WORLD,&myid);
if (myid == 0) { printf("Number of intervals: "); scanf("%d",&n); }
MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
h = 1.0 / (double) n; sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs)
                    { x = h * ((double)i - 0.5); sum += 4.0 / (1.0 + x*x); }
mypi = h * sum;
MPI_Reduce (&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
                                    0, MPI_COMM_WORLD);
if (myid == 0) printf("pi is approximately %.16f, \n", pi); MPI_Finalize(); return 0; }
```



5. Comunicación No bloqueante

- Las operaciones de comunicación vistas son “bloqueantes”
  - Incluso MPI_Send puede ser bloqueante si la implementación no soporta suficiente buferización.
- Se necesitan operaciones de comunicación no bloqueantes
  - Solapar comunicación con computación:
    - MPI_Isend: Inicia envío pero retorna antes de copiar en buffer.
    - MPI_Irecv: Inicia recepción pero retorna antes de recibir.
    - MPI_Test: Chequea si la operación no bloqueante ha finalizado.
    - MPI_Wait: Bloquea hasta que acabe la operación no bloqueante.
  - Comunicación Asíncrona, .
    - MPI_Iprobe: Chequeo no bloqueante para un mensaje.
    - MPI_Probe: Chequeo bloqueante para un mensaje.

5. Comunicación No bloqueante

5.1. Solapando Comunicación con Computación

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

- Argumento request: Identifica operación cuyo estado se pretende consultar o se espera que finalice.
- No incluyen argumento status: Se obtiene con otras 2 funciones

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

- flag > 0 ⇒ operación identificada ha finalizado, libera request , inicializa status
- Versión para varias count operaciones : int MPI_Testall (int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses)

5. Comunicación No bloqueante

5.1. Solapando Comunicación con Computación (cont.)

int MPI_Wait(MPI_Request *request, MPI_Status *status)

- Bloquea hasta que operación finaliza
- Versión para varias count operaciones : int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])

Posible conectar operaciones no bloqueantes con contrapartes bloqueantes

- int MPI_Request_free(MPI_Request *request) ⇒ Liberación explícita request

5. Comunicación No bloqueante

5.2. Comunicación Asíncrona. Sondeo de mensajes.

Accceso no estructurado a recurso compartido

Comprobar existencia de mensajes sin recibirlos

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

No bloquea si no hay mensajes,. Si hay mensaje, se recibe con MPI_Recv

flag > 0 ⇒ ∃ mensaje pendiente que encaja con (source, tag, comm).

Más información mediante status.

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

Retorna sólo cuando hay mensaje que encaje con los argumentos.

Esperar la llegada mensaje sin conocer procedencia, etiqueta o tamaño.

5. Comunicación No bloqueante

5.2. Comunicación Asíncrona. Ejemplo.

Recepción de fuente desconocida

```
int count, *buf, source;
MPI_Probe(MPI_ANY_SOURCE, 0, comm, &status);
MPI_Get_count(status, MPI_INT, &count);
buf=malloc(count*sizeof(int));
source= status.MPI_SOURCE;
MPI_Recv(buf, count, MPI_INT, source, 0, comm, &status);
```

6. Comunicadores

int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *comm_out)

- Operación colectiva que divide grupo asociado a comm en subgrupos, cada uno asociado a un comunicador diferente comm_out.
- Cada subgrupo contiene los procesos con el mismo valor de color.
- color=MPI_UNDEFINED ⇒proceso no pertenece a nuevo grupo
- La ordenación en el nuevo grupo se hace en base a key.

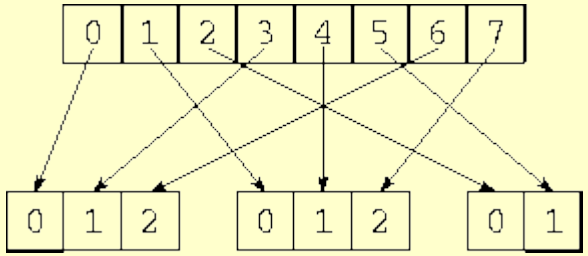
int MPI_Comm_free(MPI_Comm *comm)

- Libera el comunicador comm

6. Comunicadores

6.1. Ejemplos de creación de comunicadores

```
MPI_Comm comm, comm1,comm2,comm3;
int myid, color;
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split (comm, color, myid, &comm1);
```




```
if (myid < 8) color = 1; else color = MPI_UNDEFINED;
MPI_Comm_split (comm, color, myid, &comm2);
```

comm2= Nuevo Comunicador con 8 primeros procs de comm.

```
MPI_Comm_split (comm, 0, myid, &comm3);
```

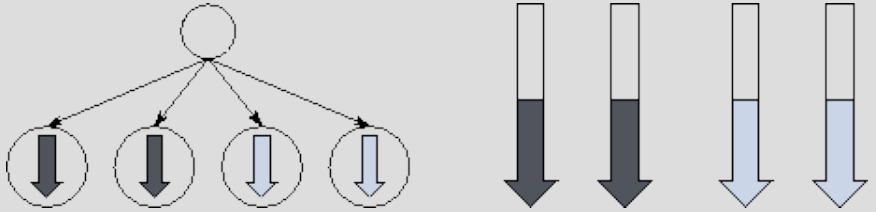
comm3= Nuevo Comunicador con el mismo grupo de procs que comm.

6. Comunicadores

6.2. Objetivos de los Comunicadores. Modularidad

Restringir determinadas operaciones comunicación a ciertos grupos → Implementación modelo MPMD

Programación por grupos



Separar paso de mensajes en grupos de procesos solapados.

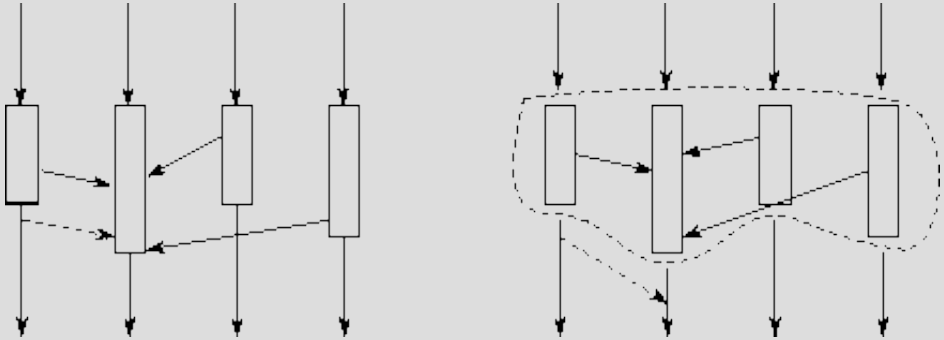
Etiquetas no bastan. Fallan para desarrollo modular.

Aplicación que usa módulo de biblioteca

Asegurar que el módulo usa diferentes etiquetas

Solución: Nuevo comunicador

- Argumento del módulo
- Desacopla paso de mensajes



7. Topologías Cartesianas

Numeración lineal (0,...,P-1) → No apropiada para algunas aplicaciones.

Ejemplo: Algoritmos de multiplicación de matrices → Malla 2D

Asignar proceso MPI a un proceso en una topología n-dimensional

7.1. Creación

int MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

Crea comunicador cartesiano comm_cart con ndims dimensiones a partir de comm.

dims[i] = número de procesos en dimensión i-ésima.

periods[i] > 0 ⇒ dimensión i periódica, periods[i]=0 ⇒ No periódica

reorder=0 ⇒ Se sigue la numeración de comm.

reorder>0 ⇒ Se permite que MPI reordene la numeración según su conveniencia, por ejemplo, para hacer coincidir la topología con la topología física de la red.

7. Topologías Cartesianas

7.2. Identificación de procesos.

int MPI_Cart_coords (MPI_Comm comm_cart, int rank, int maxdims, int *coords)

Devuelve coordenadas del proceso rank dentro del comunicador cartesiano comm_cart con maxdims dimensiones

coords= array, coordenadas del proc. en cada dimensión.

int MPI_Cart_rank (MPI_Comm comm_cart, int *coords, int *rank)

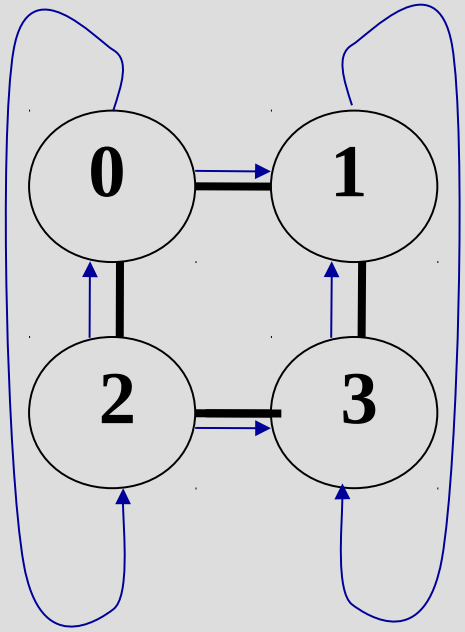
Devuelve el identificador (rank) del proceso que tiene las coordenadas coords en el comunicador cartesiano comm_cart.

int MPI_Cart_shift (MPI_Comm comm_cart, int direction, int displ, int *source, int *dest)

Devuelve el identificador de los procesos fuente y destino (source, dest) para una operación de desplazamiento de displ pasos (displ>0 → arriba, < 0 → abajo) en la dimensión direction dentro de comm_cart.

7. Topologías Cartesianas

7.3. Ejemplo.




```
MPI_Comm comm_2d;
int myrank, size, P, Q, p, q, reorder; MPI_Status status;
int dims[2], local[2], period[2];
int remain_dims[2],izqda, dcha, abajo, arriba;
double buf;
……..
::::::::::::::::::::::::::::::
MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
dims[0] = dims[1] = 2; reorder = 0; period[0] =1;  period[1] = 0;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, period,       reorder, &comm_2d);
MPI_Cart_coords(comm_2d, myrank, 2, local);
MPI_Cart_shift(comm_2d, 0, 1, &arriba, &abajo);
MPI_Sendrecv_replace( &buf, 1, MPI_DOUBLE, arriba, 0, abajo, 0, comm_2d, &status)
```

8. Tipos de datos derivados y empaquetado

int MPI_Type_vector(int count, int blocklen, int stride,
MPI_Datatype old_type, MPI_Datatype *newtype)

Crea tipo newtype = count bloques (cada uno con blocklen unidades de tipo old_type) separados por stride unidades.

int MPI_Type_commit (MPI_Datatype *datatype)

Prepara el nuevo tipo datatype para su uso en una función de comunicación.

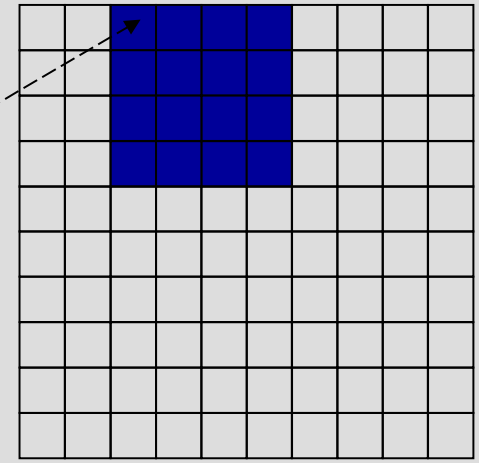
int MPI_Type_free (MPI_Datatype *datatype)

Libera el tipo de datos datatype

8. Tipos de datos derivados y empaquetado

8.1. Ejemplo de Tipos de datos derivados

Enviar un bloque 4x4, comenzando por A[0][2] de una matriz A de proceso 0 al 1



```
float A[10][10];
MPI_Datatype bloque;
……..
MPI_Type_vector(4,4,10,MPI_FLOAT,&bloque);
MPI_Type_commit(&bloque);
If (myrank==0)
    MPI_Send(&(A[0][2]),1,bloque,1,0, MPI_COMM_WORLD);
else
    MPI_Recv(&(A[0][2]),1,bloque, 0, 0, MPI_COMM_WORLD,
                                        &status);
```

8. Tipos de datos derivados y empaquetado

8.3. Empaquetando datos

int MPI_Pack (void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outcount, int *position, MPI_Comm comm)

- Empaqueta mensaje (inbuf , incount, datatype) en área contigua con outcount bytes, desde outbuf.
- Entrada position = 1ª posición buffer salida a usar para empaquetar.
- position se incrementa en el tamaño del mensaje empaquetado
- Salida de position = 1ª posición en buffer salida después empaquetar.
- comm= comunicador para enviar mensaje empaquetado.
- Mensaje empaquetado se envía como tipo MPI_PACKED

Función de Temporización (para medir tiempos)

double MPI_Wtime() → Tiempo de retardo (en segs.) desde un tiempo arbitrario en el pasado.

8. Tipos de datos derivados y empaquetado

8.3.1. Desempaquetando datos

int MPI_Unpack (void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

Desempaqueta mensaje en (outbuf, outcount, datatype) desde el área contigua con insize bytes que comienza en inbuf.

Entrada position= 1ª posición en buffer de entrada ocupada por el mensaje empaquetado.

position se incrementa en el tamaño del mensaje empaquetado

Salida de position= 1ª posición en buffer entrada después de empaquetar posiciones ocupadas por el mensaje desempaquetado.

comm=comunicador para recibir mensaje empaquetado

8. Tipos de datos derivados y empaquetado

8.3.2. Empaquetando datos. Ejemplo

```
int position, i, j, a[2]; char buff[1000]; MPI_Comm comm= MPI_COMM_WORLD;
....
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
    { position = 0;MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, comm);
      MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position , comm);
      MPI_Send( buff, position, MPI_PACKED, 1, 0 , comm); }
else
     MPI_Recv( a, 2, MPI_INT, 0, 0 , comm);
     /* ALTERNATIVA A LO ANTERIOR */
                { MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, &status);
                  position = 0; MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, comm);
                  MPI_Unpack(buff, 1000, &position, &j, 1, MPI_INT, comm) }
```

9. Referencias útiles

Referencias bibliográficas

Open MPI: Open Source High Performance Computing

http://www.open-mpi.org/

MPI Forum

`http://www.mpi-forum.org`

Parallel Programming with MPI

Peter S. Pacheco, Morgan Kaufman Publishers, Inc

Using MPI: Portable Parallel Programing with Message Passing Interface

William Gropp, E. Lusk and A. Skjellum, MIT Press