MPI

Grado en Ingeniería Informática

Programación Paralela con MPI

Departamento de Ingeniería Informática
Universidad de Cádiz
Pablo García Sánchez
Apuntes basados en el trabajo de Guadalupe Ortiz y de J. M. Mantas

Curso 2016 – 2017

Indice

Sección 1 | Introducción

¿Qué es MPI?

- Interfaz de Paso de Mensajes
- Tiene varias implementaciones: MPICH, MPICH2, OpenMPI, HP-MPI, LAM...
- Modelo SPMD (Single Program Multiple Data), extensible a MPMD

Instalación, compilación y ejecución

OJO

La compilación y ejecución dependen de la implementación escogida!

Instalación mpich2 en Ubuntu

```
sudo apt-get install mpich2
```

Compilación

```
mpicc programa.c -o programa
```

Ejecución

```
mpirun -n 4 ./programa
```

- n: número de procesos
- El orden de argumentos afecta
- No olvidar poner el ./

Sección 2 | Funciones MPI Básicas

Iniciar y finalizar

Inicializar

- `int MPI_Init (int *argc, char ***argv)`
- Antes de cualquier otra función de MPI (y sólo una vez)

Finalizar

- `int MPI_Finalize()`
- Al final de la computación para limpiar y cerrar.

Algunas consideraciones

Tipos básicos

- MPI_CHAR, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE

Comunicadores (I)

Grupo de procesadores y su contexto

- Son de tipo `MPI_Comm`
- Subconjunto de procesos. Un proceso puede pertenecer a varios comunicadores.
- Contexto: los mensajes solo se conocen de forma privada en esos procesos.
- Se usan en funciones de transferencia.
- Identificador de proceso: desde 0 hasta `size_comm`-1
- Comunicador por defecto: `MPI_COMM_WORLD`

Comunicadores (II)

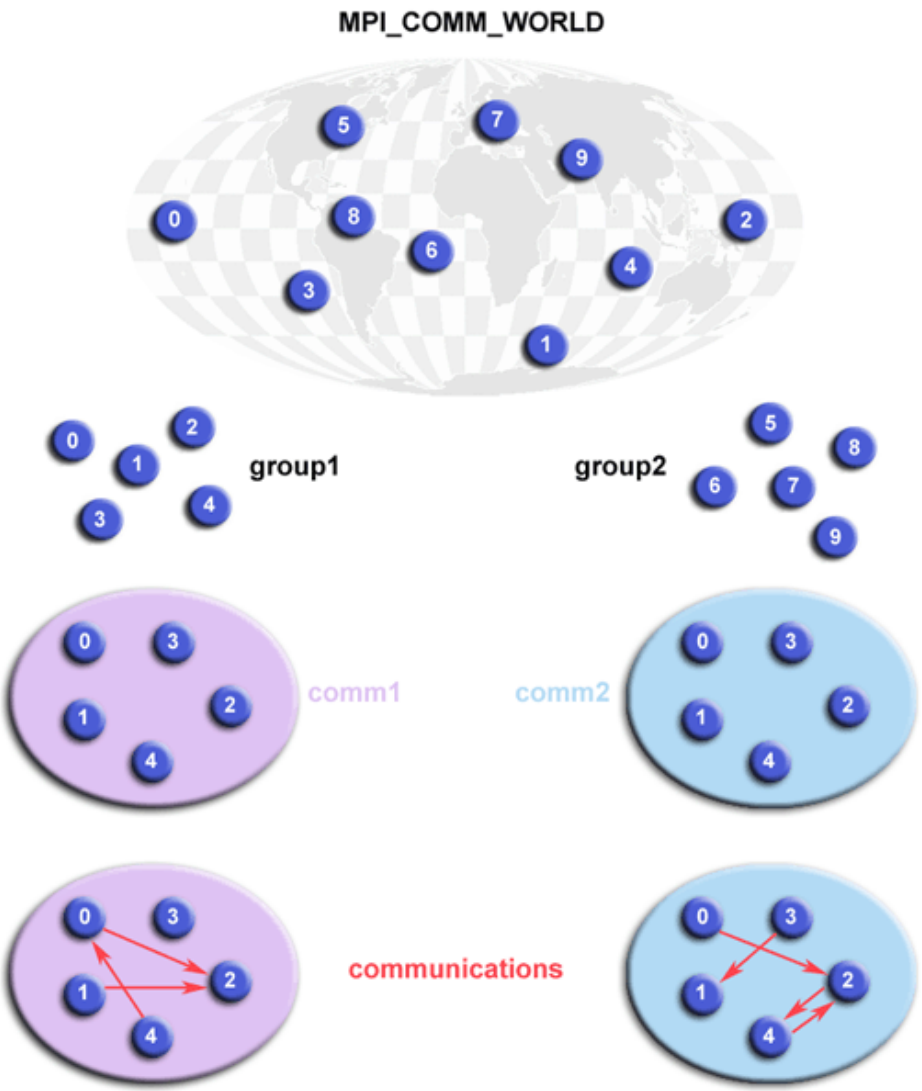


Figura: Extraída de `https://computing.llnl.gov/tutorials/mpi/`.

Comunicadores (III)

```
// Obteniendo el rango y el tamano
//del comunicador MPI\_COMM\_WORLD
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

Sección 3 | Envío y recepción de mensajes

Envío y recepción de mensajes (I)

Funciones

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

- `buf`: buffer donde están los datos a enviar/recibir
- `count`: número de datos a enviar/recibir como máximo
- `datatype`: tipo de dato a enviar
- `dest`: proceso destino
- `source`: proceso fuente
- `tag`: etiqueta
- `comm`: comunicador

Consideraciones

MPI_Send

- MPI_Send se bloquea hasta que el otro reciba.

MPI_Recv

- MPI_Recv se bloquea hasta que reciba.
- Comodines: MPI_ANY_SOURCE, MPI_ANY_TAG
- `count` mayor o igual que el emisor.

MPI_Status

- Campos MPI_SOURCE y MPI_TAG
- Tamaño del mensaje recibido `MPI_Get_count( MPI_Status *status, MPI_Datatype dtype, int *count )`

Ejemplo de Send/Recv

```
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( );
```

Ejemplo de uso de MPI_Status

```
const int MAX_NUMS = 100;
int num_buffer[MAX_NUMS];
int cantidad_nums;
if (world_rank == 0) {
    srand(time(NULL));
    cantidad_nums = (rand() / (float)RAND_MAX) * MAX_NUMS;
    MPI_Send(num_buffer, cantidad_nums, MPI_INT, 1, 0, MPI_COMM_WORLD
    printf("Proceso 0 envia %d numeros al 1\n", cantidad_nums);
} else if (world_rank == 1) {
    MPI_Status status;
    // Recibimos COMO MAXIMO MAX_NUMS del proceso 0
    MPI_Recv(num_buffer, MAX_NUMS, MPI_INT, 0, 0, MPI_COMM_WORLD,
             &status);

    // Despues de recibir el mensaje, miramos en status para
    // ver cuantos numeros han sido recibidos en realidad
    MPI_Get_count(&status, MPI_INT, &cantidad_nums);

    // Imprimimos solo los numeros recibidos, e info extra
    printf("Proceso 1 recibio %d numeros del 0. Fuente = %d, "
           "tag = %d\n",
           cantidad_nums, status.MPI_SOURCE, status.MPI_TAG);
}
```

Sección 4 | Comunicación Colectiva

Barrera

- `int MPI_Barrier ( MPI_Comm comm )`
- Bloquea el proceso hasta que TODOS los procesos del comunicador la invoquen.

Ejemplos de comunicación colectiva

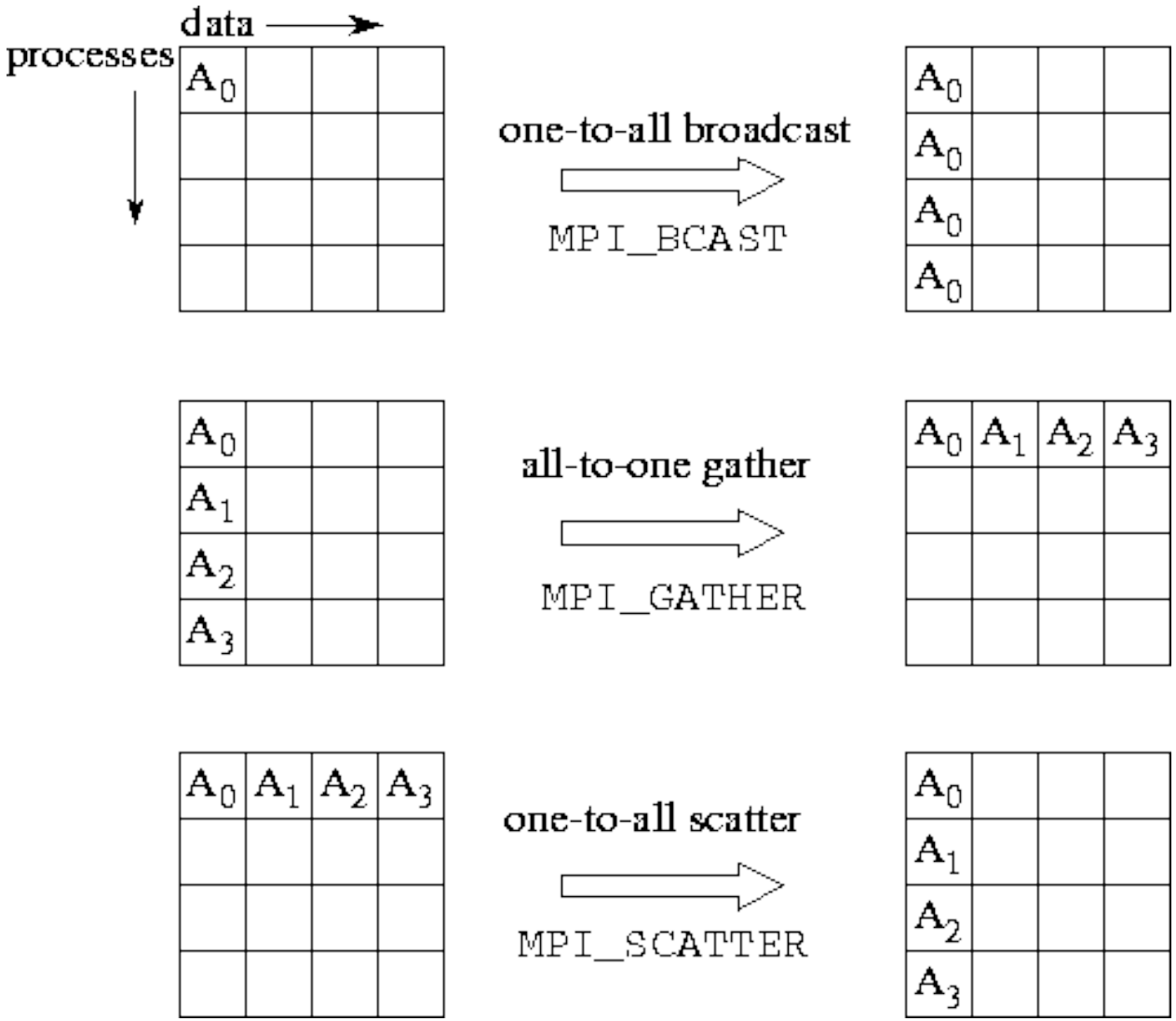


Figura: Extraída de `https://computing.llnl.gov/tutorials/mpi/`.

Broadcast

MPI_Bcast

- `int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
- El proceso `root` envía lo que haya en `buffer` a TODOS los procesos del comunicador.
- El resto de procesos lo guardan en `buffer`.
- Importante si están en distintas secciones: `datatype` y `count` debe ser igual en todos.

Gather

Gather (Los datos se almacenan en el proceso indicado `root` por orden de proceso)

- `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 el contenido de su buffer al proceso `root`. Sólo root los almacena, ordenando por número de proceso.
- De hecho, los que no son el root podrían poner recvbuf a NULL si se quiere.
- `recvcount` es el número de elementos recibidos por proceso, no el total.
- `sendtype` y `sendcount` deben ser identicos en todos los procesos.
- Segmentos de igual tamaño, normalmente N/P. ¿Qué pasa si no es divisible?

Scatter

Scatter (cada segmento de `sendbuf` se envía a un proceso)

- `int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
- `sendbuf` es dividido en segmentos de tamaño `sendcount`. El iésimo se envía al iésimo proceso del grupo, que lo recibe en `recvbuf`
- `recvcount`
- `sendcount`, `sendtype`, `recvbuf`, `recvcount`, `recvtype`, `root` y `comm` iguales en TODOS.

Ejemplo de Scatter/Gather

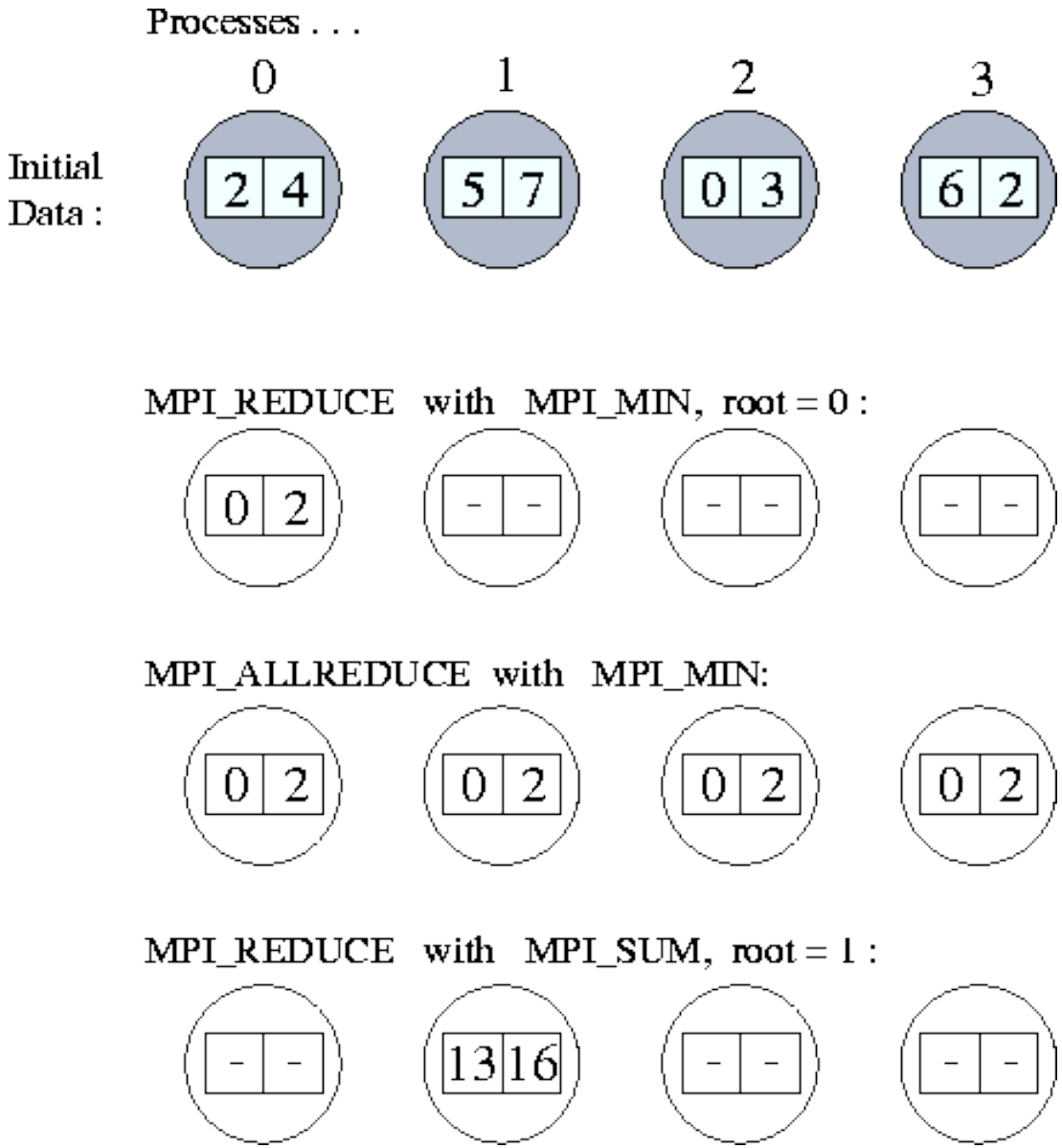
```
if (world_rank == 0) {
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
           MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
```

Reducción

Reduce

- `int MPI_Reduce (void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
- Combina los elementos que hay en `sendbuf` en el recvbuf de `root` usando la operación `op`.
- `recvbuf` no puede ser null
- `count`, `datatype`, `op`, `root`, `comm` deben identicos en todos los procesos
- `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, `MPI_LAND`, `MPI_BAND`, `MPI_LOR`, `MPI_BOR`, `MPI_LXOR`, `MPI_BXOR`.
- Función `MPI_Allreduce`. Igual pero sin argumento root, ya que comparte la reducción en todos los procesos.

Reducción (ejemplos)



Otras consideraciones

- Calcular tamaño del mensaje antes de recibirlo: `MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status)`
- Crear nuevos comunicadores a partir de otros: `MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm* newcomm)`
- Comunicación no bloqueante:
  - `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)`

Referencias

- Almeida, F., Giménez, D., Mantas, J.M., Vidal, A.M. Introducción a la Programación Paralela. Editorial Paraninfo, 2008.
- Apuntes de Guadalupe Ortiz
- `http://mpitutorial.com/`