SEMINARIO-3-CPA.pdf

HuGoCG

Computación paralela

3º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

SEMINARIO 3 PROGRAMACIÓN CON MPI

Conceptos básicos

MPI se basa en funciones de biblioteca. Para su uso se requiere una inicialización.

Ejemplo

```
#include <mpi.h>
int main(int argc, char* argv[]) {
  int k;       /* rango del proceso  */
  int p;       /* número de procesos */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &k);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  printf("Soy el proceso %d de %d\n", k, p);
  MPI_Finalize();
  return 0;
}
```

- Es obligatorio llamar a `MPI_Init` y `MPI_Finalize`
- Una vez inicializado, se pueden realizar diferentes operaciones

Modelo de programación – Comunicadores

Un comunicador es una abstracción que engloba los siguientes conceptos:

- Grupo: conjunto de procesos.
- Contexto: para evitar interferencias entre mensajes distintos.

Un comunicador agrupa a p procesos

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Cada proceso tiene un identificador (rango), un número entre 0 y $p-1$

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Modelo de ejecución

Sigue el esquema de creación simultánea de procesos al lanzar la aplicación.

La ejecución de una aplicación suele hacerse con

```
mpiexec -n p programa [argumentos]
```

Al ejecutar la aplicación se lanzan p copias del mismo ejecutable y se crea el comunicador MPI_COMM_WORLD que engloba a todos los procesos.

Comunicación Punto a Punto

El mensaje

Deben ser enviados explícitamente por el emisor y recibidos explícitamente por el receptor.

Envío estándar:

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

Recepción estándar:

```
MPI_Recv(buf, count, datatype, src, tag, comm, stat)
```

El contenido del mensaje viene definido por los 3 primeros argumentos:

- Un buffer de memoria donde está almacenada la información.
- El número de elementos que componen el mensaje.
- El tipo de datos de los elementos.

El sobre

Para efectuar la comunicación es necesario indicar el destino (dest) y el origen (src).

- La comunicación está permitida solo dentro del mismo comunicador, comm.
- El origen y el destino se indican mediante identificadores de procesos.
- En la recepción se permite utilizar src=MPI_ANY_SOURCE.

Se puede utilizar un número entero (tag) para distinguir mensajes de distinto tipo:

En la recepción se permite usar tag=MPI_ANY_TAG.

En la recepción, el estado (stat) contiene información:

- Proceso emisor (stat.MPI_SOURCE), etiqueta (stat.MPI_TAG).
- Longitud del mensaje.

Modos de envío Punto a Punto

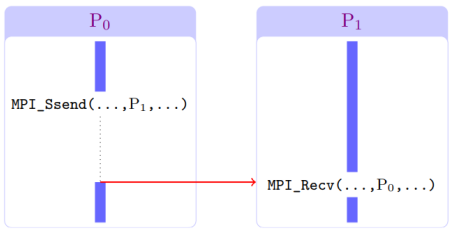
- Envío síncrono.
- Envío con memoria intermedia (buffer).
- Envío estándar.

El estándar es el más utilizado. El resto pueden ser útiles para obtener mejores prestaciones o mayor robustez. Para cada modo existen primitivas bloqueantes y no bloqueantes.

Modo de Envío Síncrono

```
MPI_Ssend(buf, count, datatype, dest, tag, comm)
```

Implementa el modelo de envío con "rendezvous": el emisor se bloquea hasta que el receptor desea recibir el mensaje

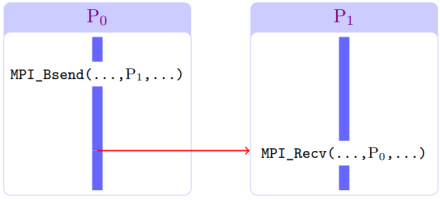


- Ineficiente: el emisor queda bloqueado sin hacer nada útil

Modo de Envío con Buffer

```
MPI_Bsend(buf, count, datatype, dest, tag, comm)
```

El mensaje se copia a una memoria intermedia y el proceso emisor continúa su ejecución



- Inconvenientes: copia adicional y posibilidad de fallo
- Se puede proporcionar un buffer (`MPI_Buffer_attach`)

msi

Tu ordenador lo único que necesita programar es su jubilación.

El Stealth 15M es uno de los portátiles gaming más finos y ligeros. Siempre menos es más. Ve a donde quieras llevando siempre el máximo rendimiento.

Modo de envío Estándar

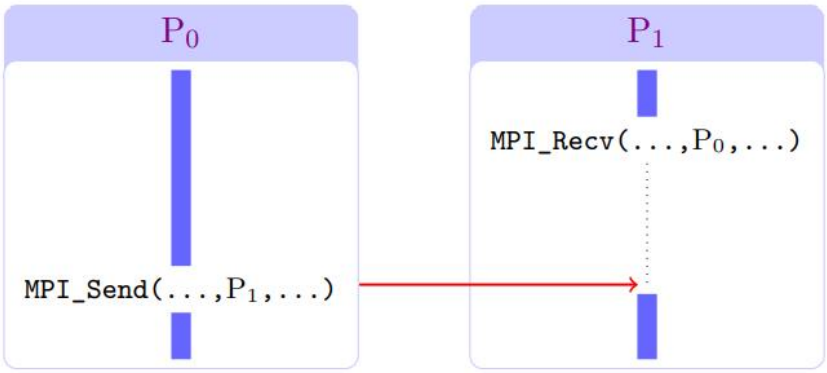
```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

Garantiza el funcionamiento en todo tipo de sistemas ya que evita problemas de almacenamiento. Los mensajes cortos se envían generalmente con MPI_Bsend, mientras que los largos son enviados con MPI_Ssend.

Recepción Estándar

```
MPI_Recv(buf, count, datatype, src, tag, comm, stat)
```

Implementa el modelo de recepción con "rendezvous": el receptor se bloquea hasta que el mensaje llega



- Ineficiente: el proceso receptor queda bloqueado sin hacer nada útil

Primitivas de envío no bloqueantes

```
MPI_Isend(buf, count, datatype, dest, tag, comm, req)
```

Se inicia el envío, pero el emisor no se bloquea. Tiene un argumento adicional (req). Para reutilizar el buffer es necesario asegurarse de que el envío se ha completado.

Ejemplo

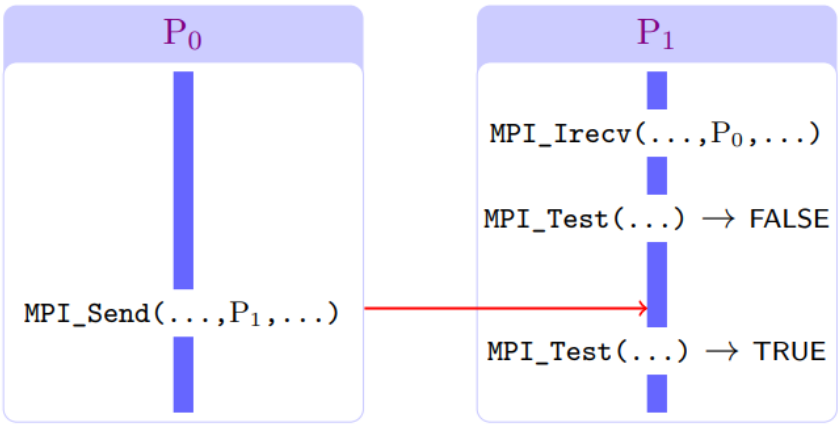
```
MPI_Isend(A, n, MPI_DOUBLE, dest, tag, comm, &req);
...
/* Comprobar que el envío ha terminado,
   con MPI_Test o MPI_Wait */
A[10] = 2.6;
```

Solapamiento de comunicación y cálculo sin copia extra. El inconveniente es que la programación es más difícil.

Recepción no Bloqueante

```
MPI_Irecv(buf, count, type, src, tag, comm, req)
```

Se inicia la recepción, pero el receptor no se bloquea. Se sustituye el argumento stat por req. Es necesario comprobar después si el mensaje ha llegado.



La ventaja es que se solapa la comunicación y el salto, pero la programación es más difícil.

Operaciones combinadas

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
        recvcount, recvtype, source, recvtag, comm, status)
```

Realiza una operación de envío y recepción al mismo tiempo (no necesariamente con el mismo proceso).

```
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source,
                     recvtag, comm, status)
```

Realiza una operación de envío y recepción al mismo tiempo sobre la misma variable.

Diapos 22 a 27 ejemplos programación con MPI

Comunicación colectiva

Las operaciones de comunicación colectiva involucran a todos los procesos de un grupo (comunicados) – todos ellos deben ejecutar la operación.

Operaciones disponibles:

- Sincronización (Barrier)
- Difusión (Bcast)
- Reparto (Scatter)
- Recogida (Gather)
- Multi-recogida (Allgather)
- Todos a todos (Alltoall)
- Reducción (Reduce)
- Prefijación (Scan)

Estás operaciones suelen tener como argumento un proceso (root) que realiza un papel especial.

Prefijo “All”: todos los procesos reciben el resultado.

Sufijo “v”: la cantidad de datos en cada proceso es distinta.

Sincronización

`MPI_Barrier(comm)`

Operación de pura sincronización. Todos los procesos de comm se detienen hasta que todos ellos han invocado esta operación.

Ejemplo – medición de tiempos

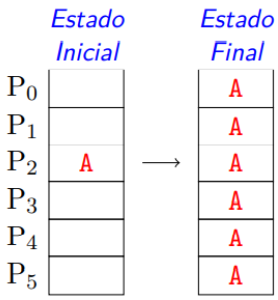
```
MPI_Barrier(comm);
t1 = MPI_Wtime();
/*
    ...
*/
MPI_Barrier(comm);
t2 = MPI_Wtime();

if (!rank) printf("Tiempo transcurrido: %f s.\n", t2-t1);
```

Difusión

`MPI_Bcast(buffer, count, datatype, root, comm)`

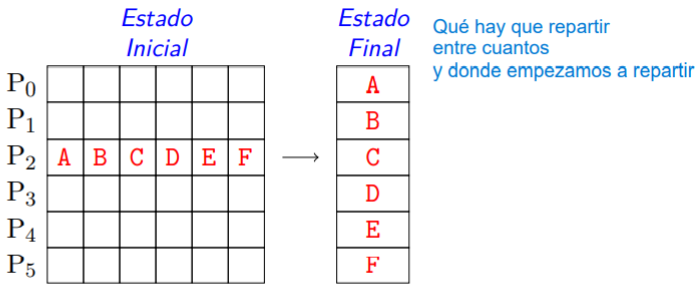
El proceso root difunde al resto de procesos el mensaje definido por los 3 primeros argumentos.



Reparto

`MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`

El proceso root distribuye una serie de fragmentos consecutivos del buffer al resto de procesos (incluyéndose a él mismo).



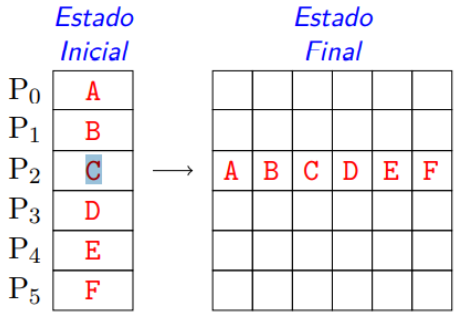
Versión asimétrica: `MPI_Scatterv`

Ejemplo de reparto diapo 33

Recogida

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
recvcount, recvtype, root, comm)
```

Es la operación inversa de MPI_Scatter: cada proceso envía un mensaje a root, el cual lo almacena de forma ordenada de acuerdo al índice del proceso en el buffer de recepción.

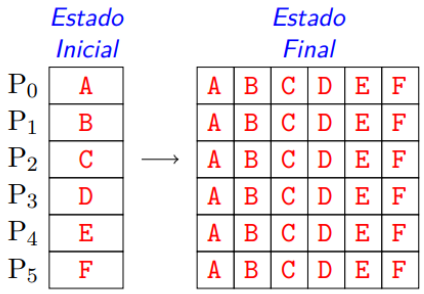


Versión asimétrica: `MPI_Gatherv`

Multi-Recogida

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,
recvcount, recvtype, comm)
```

Similar a MPI_Gather, pero todos los procesos obtienen el resultado.

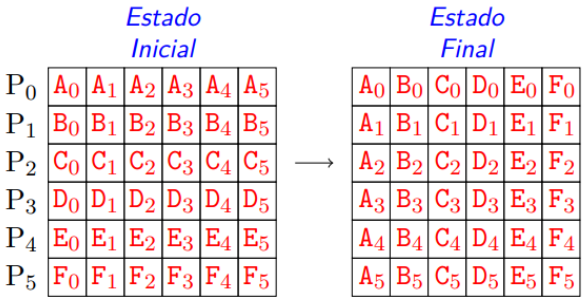


Versión asimétrica: `MPI_Allgatherv`

Todos a Todos

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf,
recvcount, recvtype, comm)
```

Es una extensión de MPI_Allgather, cada proceso envía unos datos distintos y recibe del resto.



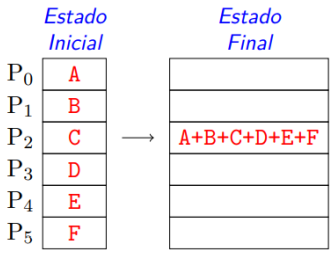
Versión asimétrica: `MPI_Alltoallv`

Reducción

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

Similar a MPI_Gather, pero en lugar de concatenación, se realiza una operación aritmética o lógica (suma, max, and, ..., o definida por el usuario).

El resultado final se devuelve en el proceso `root`



Multi-Reducción

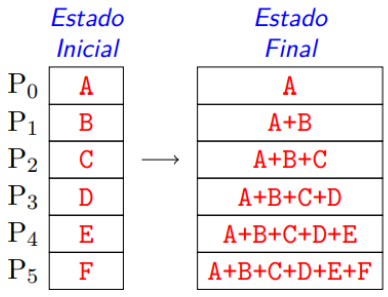
```
MPI_Allreduce(sendbuf, recvbuf, count, type, op, comm)
```

Extension de MPI_Reduce en la que todos reciben el resultado.

Prefijación

```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)
```

Extension de las operaciones de reducción en la que cada proceso recibe el resultado del procesamiento de los elementos de los procesos desde el 0 hasta él mismo.



Otras funcionalidades

Tipos de datos básicos

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

Datos múltiples

Se permite el envío y recepción de múltiples datos:

- El emisor indica el número de datos a enviar en el argumento count.
- El mensaje lo componen los count elementos contiguos en memoria.
- En el receptor, el argumento count indica el tamaño del buffer.

Para saber el tamaño del mensaje:

```
MPI_Get_count(MPI_Status *status, MPI_Datatype
              datatype, int *count)
```

Este Sistema no sirve para componer un mensaje con varios datos de distinto tipo ni enviar datos del mismo tipo pero que no estén contiguos en memoria.

Tipos de datos derivados

MPI nos permite definir tipos nuevos a partir de otros tipos. El funcionamiento se basa en las siguientes fases:

1. El programador define el nuevo tipo, indicando:
    a. Los tipos de los diferentes elementos que lo componen.
    b. El número de elementos de cada tipo.
    c. Los desplazamientos relativos de cada elemento.
2. Se registra como un nuevo tipo de datos MPI (commit).
3. A partir de entonces, se puede usar para crear mensajes como su fuera un tipo de datos básico.
4. Cuando no se va a usar más, el tipo se destruye, (free).

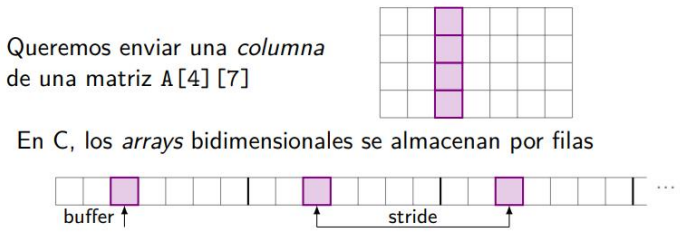
Simplifica la programación cuando se repite muchas veces y no hay copia intermedia.

Tipos de datos derivados regulares

```
MPI_Type_vector(count, length, stride, type, newtype)
```

Crea un tipo de datos homogéneo y regular a partir de elementos de un array equiespaciados.

1. De cuántos bloques se compone (count).
2. De qué longitud son los bloques (length).
3. Qué separación hay entre un elemento de un bloque y el mismo elemento del siguiente bloque (stride).
4. De qué tipo son los elementos individuales (type).




```
double A[4][7];
MPI_Datatype columna;
MPI_Type_vector(4, 1, 7, MPI_DOUBLE, &columna);
MPI_Type_commit(&columna);
if (my_rank == 0) {          /* envía la 3ª columna */
  MPI_Send(&A[0][2], 1, columna, 1, 0, comm);
} else {
  MPI_Recv(&A[0][2], 1, columna, 0, 0, comm, &status);
}
```