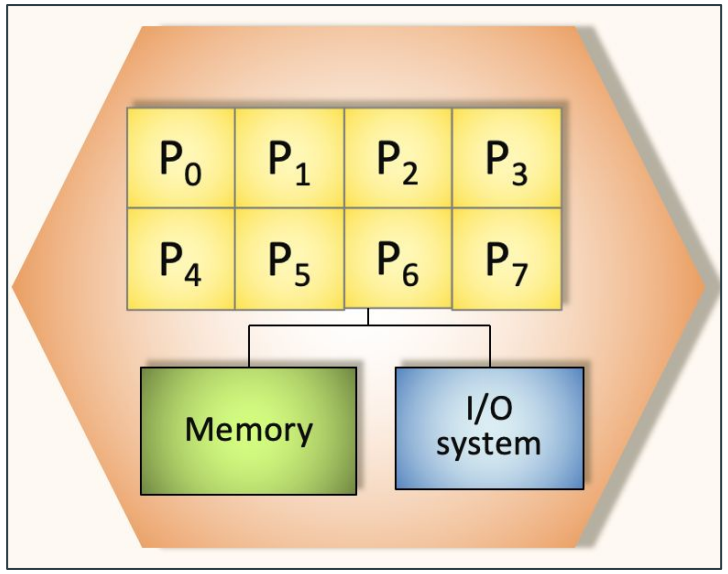
Seminario de MPI

Nicolás Calvo Cruz

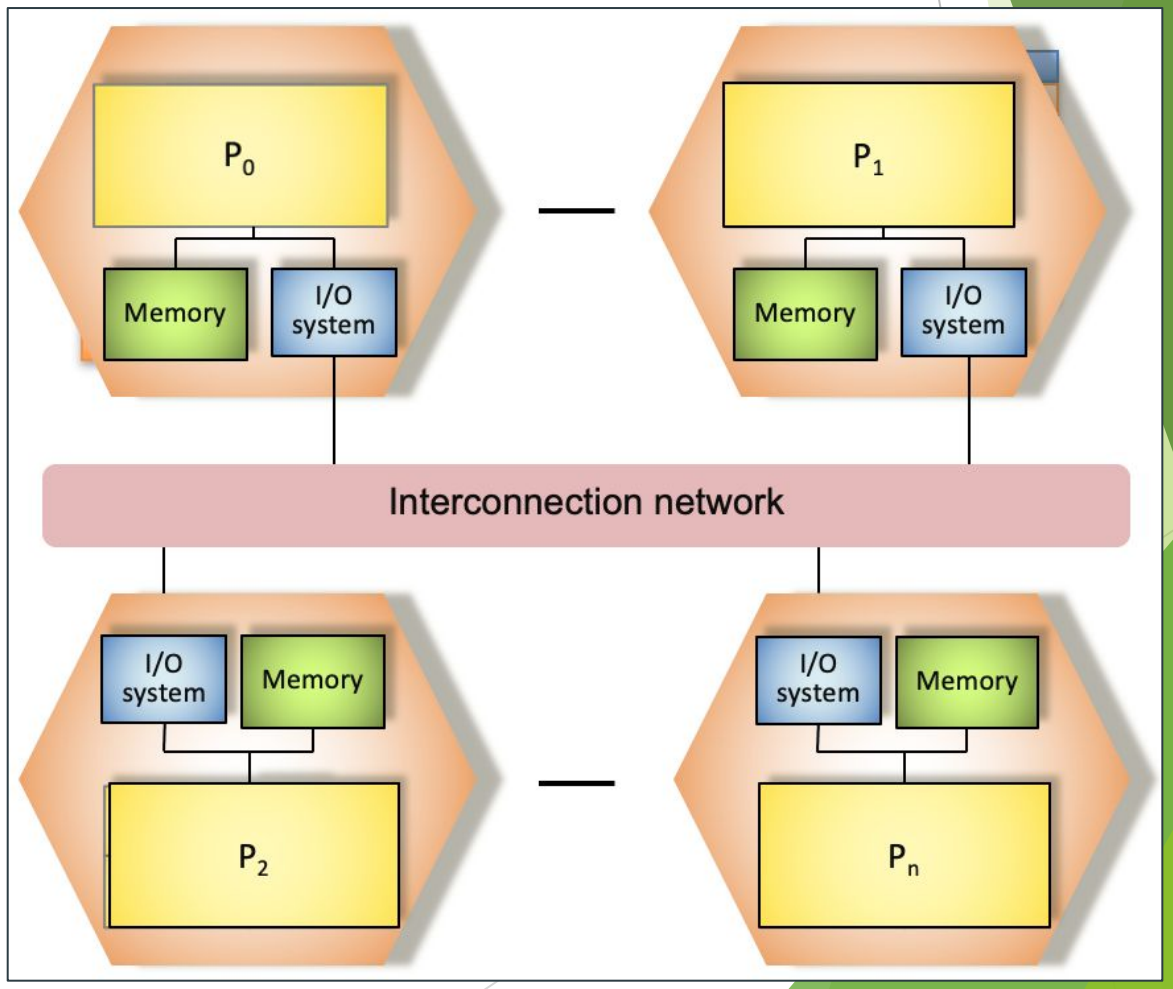
UNIVERSIDAD DE GRANADA

Conceptos básicos (I)

Memoria Compartida



Memoria Distribuida



Conceptos básicos (II)

- **Programa**: Conjunto de **código y datos** que puede ejecutarse en un computador para resolver un problema.

- **Proceso**: **Programa en ejecución** divisible en **tareas**, que se comunican en **memoria compartida** si son internas, y **mediante** paso de **mensajes** si son de otros procesos.

- Podemos crear **múltiples procesos a partir de un mismo programa**, que es la idea clave de programar con MPI.

- **Controlaremos el flujo de ejecución mediante el ID de proceso y la comunicación** (envío y recepción de mensajes).

¿Qué es MPI? (I)

- Lanzado en 1994, MPI es un estándar que define una **API** para la programación en **memoria distribuida** por **paso de mensajes**. De hecho, MPI es "*Message Passing Interface*". Su versión **4.0** es de **2021**.

- Hay diversos proveedores que proporcionan implementaciones de ese estándar, destacando **OpenMPI** y **MPICH**, gratuitas y de código abierto.

- En general, se busca ofrecer **alto rendimiento** y **portabilidad**, siendo MPI una herramienta muy extendida (**estándar de facto**) en la comunidad HPC (p. ej., muchos supercomputadores del TOP500 apuestan por OpenMPI).

- Proporciona una **librería de funciones**, no un lenguaje, por lo que podremos programar con MPI desde **C**, C++, Fortran…

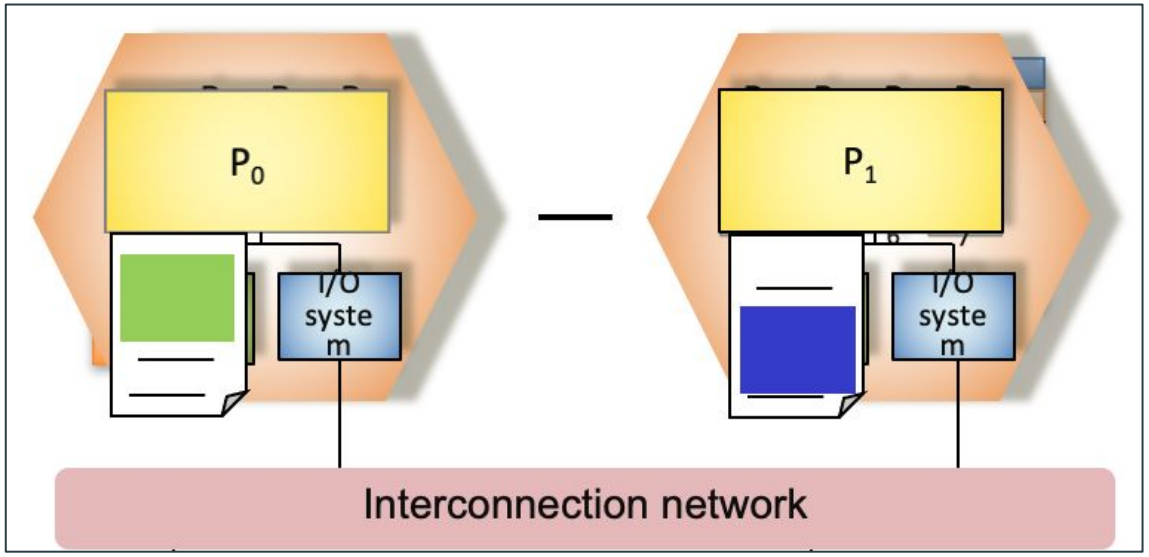
¿Qué es MPI? (II)

- Sigue un modelo de **paralelismo explícito**: definido y controlado totalmente por el programador.
- Los programas se diseñan según un modelo **SPMD**: S*ingle-Program, Multiple Data*.
- El único mecanismo de comunicación entre los procesos es el **paso de mensajes** (independientemente de la arquitectura y cómo se implemente).
- Define **comunicaciones asíncronas y síncronas**.

¿Qué ocurre al ejecutar con MPI?

- A partir de nuestro programa, se crean tantos **procesos independientes** como se requieran (pueden estar en la **misma máquina o en distintas**).

- Cada uno **ejecuta distintas zonas del programa según** su identificación (**rango**) y/o de la **información** que reciba.




```
ID = Quien_soy_yo

Si soy ID = 1 entonces
  envio Datos a 2 y espero confirmacion

Si soy ID = 2 entonces
  recibo Datos de 1 y envio confirmacion
```

Instalación y uso

- **Instalación** (OpenMPI): *sudo apt install openmpi-bin libopenmpi-dev*

(+ info en: https://lsi2.ugr.es/jmantas/ppr/ayuda/instalacion.php?ins=openmpi)

Podemos comprobarlo con el comando: *mpicc --version*

- ¿Cómo **compilamos** programas MPI?

Con mpicc en vec de gcc (*wrapper* de MPI sobre gcc): *mpicc code.c -o prog.exe*

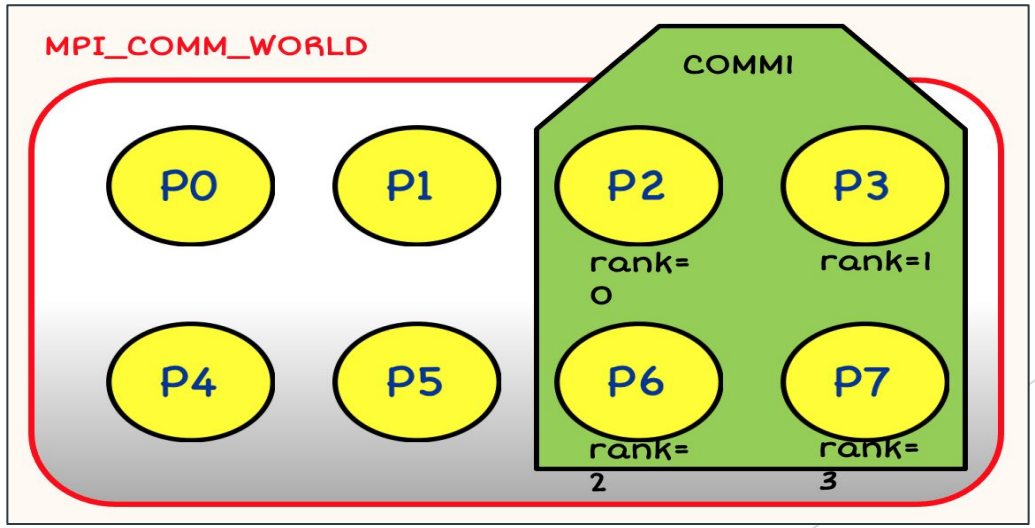
- ¿Cómo **ejecutamos** nuestro programa? *mpiexec -n <#procesos> ejecutable*

Por ejemplo: mpiexec –n 4 prog.exe

Nota: También podemos ejecutar con *mpirun* con esta misma sintaxis, pero preferimos *mpiexec* (https://stackoverflow.com/questions/25287981/mpiexec-vs-mpirun)
Nota: Según la versión de MPI podremos tener problemas para lanzar más procesos que núcleos: (https://stackoverflow.com/questions/35704637/mpirun-not-enough-slots-available)

Programación MPI. Generalidades (I)

- Los procesos MPI se agrupan en **grupos de comunicación** (***communicator***) que definen el **alcance del intercambio de mensajes**.
- Hay un **grupo de comunicación predeterminado**, MPI_COMM_WORLD, con todos los procesos al inicio.
- En un grupo de comunicación, cada proceso se identifica por su rango, un número (de 0 a Núm. de Procesos -1).
- Un proceso **puede estar en varios** grupos de comunicación.



Programación MPI. Generalidades (II)

- Para usar funciones y constantes MPI, hay que cargar la cabecera “mpi.h”:

#include <mpi.h>

- En general, las funciones y constantes de MPI empiezan con el prefijo MPI_:
  - En el caso de funciones, les sigue una letra mayúscula y el resto minúsculas. P.ej.:

    MPI_Init

  - En el caso de constantes, son todas mayúsculas. P.e.j:

    MPI_INT

- Las **funciones que no son MPI se ejecutan con normalidad** y con acceso al **espacio de memoria de su proceso**. P.ej: *printf*

Programación MPI. Generalidades (III)

- Los mensajes MPI son secuencias de ***count*** elementos del tipo ***datatype***.

  MPI define los **siguientes tipos** básicos para C (y C++, como curiosidad):

C	C++
MPI_CHAR	MPI::CHAR
MPI_SHORT	MPI::SHORT
MPI_INT	MPI::INT
MPI_LONG	MPI::LONG
MPI_FLOAT	MPI::FLOAT
MPI_DOUBLE	MPI::DOUBLE
MPI_LONG_DOUBLE	MPI::LONG_DOUBLE
MPI_BYTE	MPI::BYTE
MPI_PACKED	MPI::PACKED

Programación MPI. Generalidades (IV)

- Lo que se **envía y se recibe** ha de estar **consecutivo en memoria**, así que cuidado con las estructuras de datos, las matrices de punteros…

- Para mayor control, se recomienda usar **memoria dinámica** reservada con *malloc* en un **único bloque**, asegurando así la contigüidad (y no se debe olvidar liberar la memoria reservada con *free*).

- Además de los tipos básicos, se pueden definir estructuras de datos propios para facilitar el envío y recepción de información compleja. Se puede hacer con:
  - MPI_Type_struct(...)
  - MPI_Type_commit(...)
  - MPI_Adress(...)
  - MPI_Type_free(...)

Programación MPI. Generalidades (V)

- Los **parámetros de las funciones MPI** se clasifican en **3 categorías**:
  - IN: La función lee el argumento
  - OUT: La función modifica el argumento
  - IN/OUT: La función lee y modifica el argumento

- Las funciones MPI (casi todas) devuelven un entero como código de error:

  error = MPI_Function(...)

- Si todo ha ido **bien, será MPI_SUCCESS (0)**, y si no, será un **valor indicativo del error**.

Funciones MPI (I)

- Las funciones principales de MPI son (en negrita las que veremos):

- **MPI_Init**
- **MPI_Finalize**
- **MPI_Comm_rank**
- **MPI_Comm_size**
- **MPI_Send**
- **MPI_Recv**
- MPI_Isend
- MPI_Irecv
- MPI_Test
- MPI_Wait
- MPI_Waitall
- MPI_Request_free
- MPI_Iprobe
- MPI_Probe
- MPI_Get_count
- MPI_Sendrecv
- MPI_Sendrecv_replace
- MPI_Barrier
- MPI_Bcast
- MPI_Scatter
- MPI_Gather
- MPI_Reduce
- MPI_Allreduce
- MPI_Scan
- MPI_Comm_split
- MPI_Comm_free
- MPI_Cart_create
- MPI_Cart_coords
- MPI_Cart_rank
- MPI_Cart_shift
- MPI_Type_vector
- MPI_Type_commit
- MPY_Type_free
- **MPI_Pack**
- **MPI_Unpack**
- **MPI_Wtime**
- MPI_Error_string
- MPI_Get_processor_name

Funciones MPI (II)

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

Inicializa el entorno MPI y debe llamarse una única vez en el código. La comunicación no funcionará antes. Se le suele dar la dirección de los argumentos de entrada, aunque cómo se usan no está definido en el estándar, y depende de la implementación. Devuelve la constante **MPI_SUCCESS si fue bien.**

- *MPI_Finalize(void)*

Finaliza la infraestructura de comunicación entre procesos. El estándar no define cómo quedan los procesos después, por lo que se recomienda **minimizar las operaciones posteriores** (y reservarlas a un único proceso).

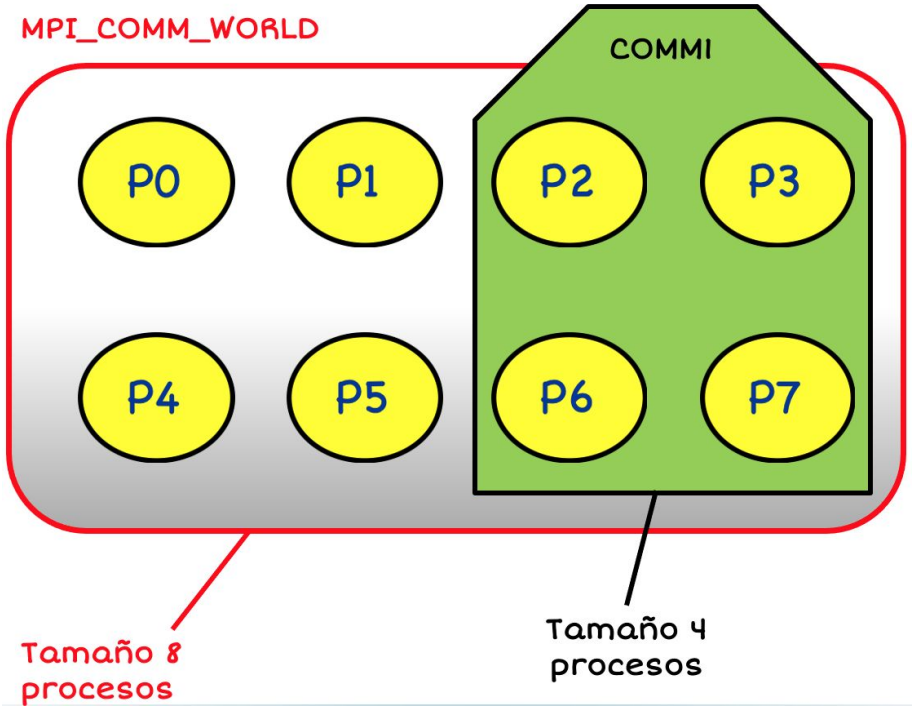
```
int main(int argc, char* argv[]){
    //Declaraciones
    MPI_Init(&argc, &argv); // Entorno paralelo: ON
    //Trabajo paralelo
    MPI_Finalize();         // Entorno paralelo: OFF
    return 0;
}
```

Nota: También existe *MPI_Init_thread*, recomendado para cuando se hibrida MPI con hilos (especialmente si se va a llamar a funciones MPI de forma concurrente).

Funciones MPI (III)

- int MPI_Comm_size(MPI_Comm comm, int* size):

Escribe, **en la dirección que se le pasa como segundo argumento**, el **número de procesos que hay en el grupo de comunicación** del primer argumento.

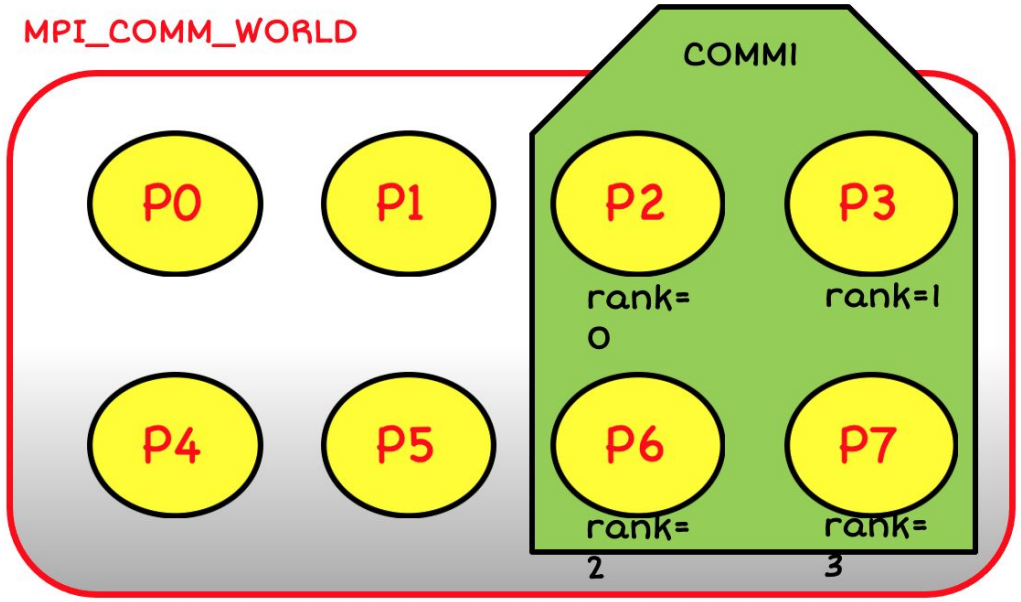


Nota: La salida de la función propiamente dicha será un código de operación MPI (MPI_SUCCESS) u otro. Esto será así en general, así que **lo obviaremos en adelante.**

Funciones MPI (IV)

- int MPI_Comm_rank(MPI_Comm comm, int* rank):

Escribe, **en la dirección que se le pasa como segundo argumento**, el **ID (dentro del grupo de comunicación** dado) del proceso que la ejecuta.



Funciones MPI (V)

- *double MPI_Wtime(**void**):*

Devuelve el tiempo desde el inicio del proceso **en segundos**. Los **tiempos** son **locales** al proceso que llama, por lo que los tiempos pueden variar entre ellos (especialmente si están repartidos en distintas máquinas). Sirve para **medir tiempos de ejecución**:

```
double tIni, tEnd;
tIni = MPI_Wtime();
// TAREAS
tEnd = MPI_Wtime();

printf("El tiempo de ejecución es: %.2lf (s)\n", tEnd-tIni);
```

Ejemplo 1

```
nicolas@lusitania:~/Documentos/Universidad/Docencia/UGR$ mpicc -o ej1 ejemplo1.c

nicolas@lusitania:~/Documentos/Universidad/Docencia/UGR$ mpiexec -n 4 ./ej1
Soy el proceso 0 en el procesador lusitania
MASTER: Hay un total de 4 procesos!
Soy el proceso 1 en el procesador lusitania
Soy el proceso 2 en el procesador lusitania
Soy el proceso 3 en el procesador lusitania
```

```c
#include <stdio.h>
#include <mpi.h>

#define MASTER 0

int main(int argc, char* argv[]){
        int rank, len, numProcs;
        char procName[MPI_MAX_PROCESSOR_NAME];
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Get_processor_name(procName, &len);
        printf("Soy el proceso %d en el procesador %s\n", rank, procName);
        if(rank==MASTER){
                MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
                printf("MASTER: Hay un total de %d procesos!\n", numProcs);
        }
        MPI_Finalize();
        return 0;
}
```

¿Alguna función que te **sorprenda**?

Funciones MPI (VI)

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

Hace un **envío bloqueante** de un proceso de origen a otro de destino. **La ejecución no continua hasta que el buffer de salida pueda ser reutilizado.**

Enlace muy interesante:
https://stackoverflow.com/questions/10017301/mpi-blocking-vs-non-blocking

Los parámetros son:

- buf: Dirección de memoria donde comienzan los datos a enviar ("puntero a cualquier cosa")
- count: Número de elementos a enviar (cuántas posiciones leer a partir de la dirección inicial). Debe ser un número natural.
- datatype: Tipo de dato MPI que se está enviando. Por ejemplo, MPI_INT.

Funciones MPI (VI)

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

Hace un **envío bloqueante** de un proceso de origen a otro de destino. **La ejecución no continua hasta que el buffer de salida pueda ser reutilizado.**

Enlace muy interesante:
https://stackoverflow.com/questions/10017301/mpi-blocking-vs-non-blocking

Los parámetros son:

- dest: ID (rango, *rank*) del proceso de destino dentro del comunicador *comm*.
- tag: Etiqueta del mensaje con la que podemos definir su tipo.
- comm: Comunicador usado para el envío del mensaje.

Funciones MPI (VII)

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

Bloquea al proceso que la ejecuta hasta que reciba un mensaje con las características indicadas.

Los parámetros son:

- buf: Dirección de memoria donde comienzan a guardarse los datos recibidos.
- count: Máximo número de elementos que se espera recibir (tamaño del buffer).
- datatype: Tipo de datos MPI a recibir. Por ejemplo, MPI_INT

Funciones MPI (VII)

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

Bloquea al proceso que la ejecuta hasta que reciba un mensaje con las características indicadas.

Los parámetros son:

- source: ID del proceso del que se espera el envío. Podemos poner la constante **MPI_ANY_SOURCE** para admitir cualquiera.
- tag: Etiqueta del mensaje esperado. Sólo se aceptará un mensaje con la etiqueta esperada, pero podemos poner **MPI_ANY_TAG** para cualquiera.
- comm: Comunicador por el que se recibirá el mensaje (canal).
- status: Estructura MPI_Status con datos del mensaje (como origen, etiqueta, tamaño…)

Ejemplo 2

```
nicolas@lusitania:~/Documentos/Universidad/Docencia/UGR$ mpirun -n 1 ./ejemplo2
Ejecuta este programa con 2 procesos!
nicolas@lusitania:~/Documentos/Universidad/Docencia/UGR$ mpirun -n 2 ./ejemplo2
Soy el proceso 1 y he recibido:
0       1       2       3       4       5       6       7       8       9
```

```c
#include <stdio.h>
#include <mpi.h>

#define BUFF_SIZE 10

int main(int argc, char* argv[]){
        int rank, size;
        int buffer[BUFF_SIZE];
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if(size!=2){
                if(rank==0){
                        printf("Ejecuta este programa con 2 procesos!\n");
                }
        }else{
                if(rank==0){
                        for(int i=0; i<BUFF_SIZE; i++){
                                buffer[i] = i;
                        }
                        MPI_Send(buffer, BUFF_SIZE, MPI_INT, 1, 0, MPI_COMM_WORLD);
                }else{
                        MPI_Recv(buffer, BUFF_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
                        printf("Soy el proceso 1 y he recibido:\n");
                        for(int i=0; i<BUFF_SIZE; i++){
                                printf("%d\t", buffer[i]);
                        }
                        printf("\n");
                }//A veces, si mostramos muchas cosas, tal vez no se vean como esperamos.
        }        //Es porque estan en el buffer de salida => fflush(stdout) nos ayuda
        MPI_Finalize();
        return 0;
}
```

Funciones MPI (VIII)

- int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void* outbuf, int outsize, int* position, MPI_Comm comm):

Empaqueta distintos tipos de datos en memoria de forma contigua para su envío (**de tipo MPI_PACKED**). **La usaremos de forma iterativa en general.**

Estos son sus parámetros:

- inbuf: Dirección de memoria donde empiezan los datos a empaquetar.
- incount: Número de elementos en el buffer anterior.
- datatype: Tipo de datos MPI del buffer anterior.

Funciones MPI (VIII)

- int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void* outbuf, int outsize, int* position, MPI_Comm comm):

Empaqueta distintos tipos de datos en memoria de forma contigua para su envío (**de tipo MPI_PACKED**). **La usaremos de forma iterativa en general.**

Estos son sus parámetros:

- outbuf: Dirección de memoria donde empieza el buffer de destino (a enviar).
- outsize: Tamaño del buffer de salida, en bytes.
- position: Posición actual donde estamos escribiendo en el buffer. Este parámetro se incrementa con cada llamada.
- comm: Comunicador asociado al mensaje empaquetado.

Funciones MPI (IX)

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

Extrae el contenido de un paquete de información variada. Como MPI_Pack, se usará de forma iterativa normalmente.

Recibe los siguientes parámetros:

- inbuf: Dirección del comienzo del paquete en memoria.
- insize: Tamaño del buffer de entrada en bytes.
- position: Posición (número de byte) por el que va la lectura del buffer. Se incrementa con cada llamada.

Funciones MPI (IX)

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

Extrae el contenido de un paquete de información variada. Como MPI_Pack, se usará de forma iterativa normalmente.

Recibe los siguientes parámetros:

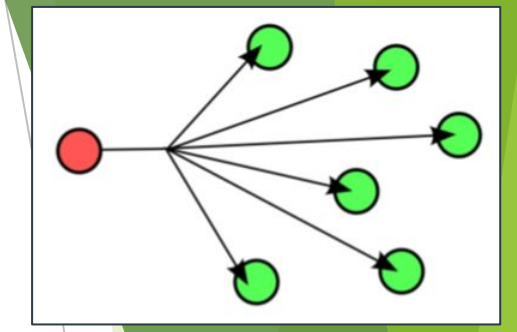
- outbuf: Dirección de inicio del buffer de salida.
- outcount: Número de elementos del tipo de salida.
- datatype: Tipo de dato a extraer.
- comm: Comunicador (canal) del mensaje empaquetado.

Ejemplo 3

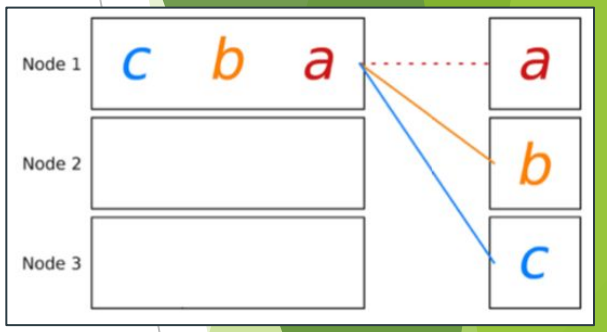
```
nicolas@miriam-pc:~/Escritorio$ mpicc -o ejemplo3 ejemplo3.c
nicolas@miriam-pc:~/Escritorio$ mpirun -n 1 ./ejemplo3
Ejecuta este programa con 2 procesos!
nicolas@miriam-pc:~/Escritorio$ mpirun -n 3 ./ejemplo3
Ejecuta este programa con 2 procesos!
nicolas@miriam-pc:~/Escritorio$ mpirun -n 2 ./ejemplo3
Soy el proceso [1] y he recibido esto:
primerDato=3; segundoDato={4.4, 5.5}:
```

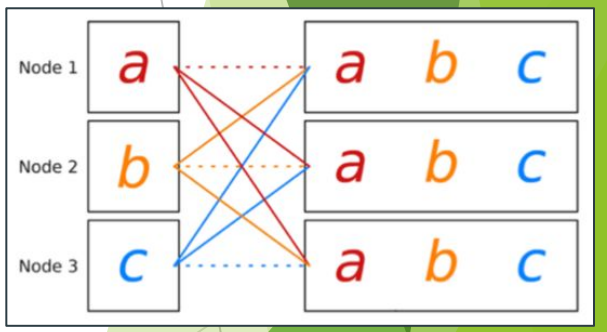
```c
#include <stdio.h>
#include <mpi.h>

const int bufferSize = sizeof(int) + 2*sizeof(double); //Es lo que enviaremos

int main(int argc, char* argv[]){
        int rank, size;
        int position = 0;// Foco sobre el buffer (Ambos van a empezar en 0)
        char buffer[bufferSize]; // char -> byte
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if(size != 2){
                if(rank==0){
                        printf("Ejecuta este programa con 2 procesos!\n");
                }
        }else{
                int primerDato = 0;
                double segundoDato[2];
                if(rank==0){
                        primerDato = 3;
                        segundoDato[0] = 4.4;
                        segundoDato[1] = 5.5;
                        MPI_Pack(&primerDato, 1, MPI_INT, buffer, bufferSize, &position, MPI_COMM_WORLD);
                        MPI_Pack(segundoDato, 2, MPI_DOUBLE, buffer, bufferSize, &position, MPI_COMM_WORLD);
                        MPI_Send(buffer, bufferSize, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
                }else{
                        MPI_Recv(buffer, bufferSize, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
                        MPI_Unpack(buffer, bufferSize, &position, &primerDato, 1, MPI_INT, MPI_COMM_WORLD);
                        MPI_Unpack(buffer, bufferSize, &position, segundoDato, 2, MPI_DOUBLE, MPI_COMM_WORLD);
                        printf("Soy el proceso [%d] y he recibido esto:\n", rank);
                        printf("primerDato=%d; segundoDato={%.1lf, %.1lf}:\n", primerDato, segundoDato[0], segundoDato[1]);
                }
        }

        MPI_Finalize();
        return 0;
}
```

Funciones MPI (X)



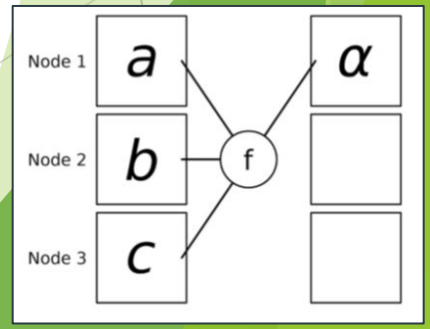
- MPI implementa también un rico conjunto de operaciones de **comunicación 1-N y N-N** (versiones bloqueantes en general, y también no bloqueantes desde MPI v3). Podemos mencionar, por ejemplo:
  - MPI_Bcast(...)
  - MPI_Reduce (...)
  - MPI_Allreduce(...)
  - MPI_Gather(...)
  - MPI_AllGather(...)
  - MPI_Scatter(...)





Además, algunas de ellas tienen su variante "*v*", de forma que se pueda ajustar, para cada proceso, la cantidad de datos que recibe exactamente. Mirad por ejemplo este interesante ejemplo de **MPI_Allgatherv**:

http://stackoverflow.com/questions/15951988/understanding-mpi-allgatherv-in-plain-english



Ejemplo 4

- Vamos a hacer un ejemplo más vistoso:
  - Un programa MPI en el que el proceso 0 crea un vector de enteros definido por parámetro donde v[i] = i, y lo reparte equitativamente entre el resto de procesos que haya (si no hay al menos un proceso además del máster, se cancela).
  - Los procesos que no son el máster no saben lo que van a recibir, así que se enteran para reservar la memoria necesaria (**MPI_Probe & MPI_Get_Count**), y reciben entonces su asignación.
  - Recibida su asignación, los procesos trabajadores suman todo lo recibido y lo devuelven al proceso 0, que muestra el resultado.

Enlaces de interés

- www.open-mpi.org
- https://mpitutorial.com
- https://stackoverflow.com
- N.C. Cruz. Una introducción informal a C

¡Gracias!

¿Alguna pregunta?