Introducción a MPI

Computación de alto rendimiento

Índice de Contenidos

1. Introducción a MPI

1. Introducción a MPI

- **Interface de paso de mensajes (Message Passing Interface, MPI)**
- **Biblioteca de funciones para C, C++, Fortran…**
  - Se necesita incluir en la cabecera
  - Sintaxis:

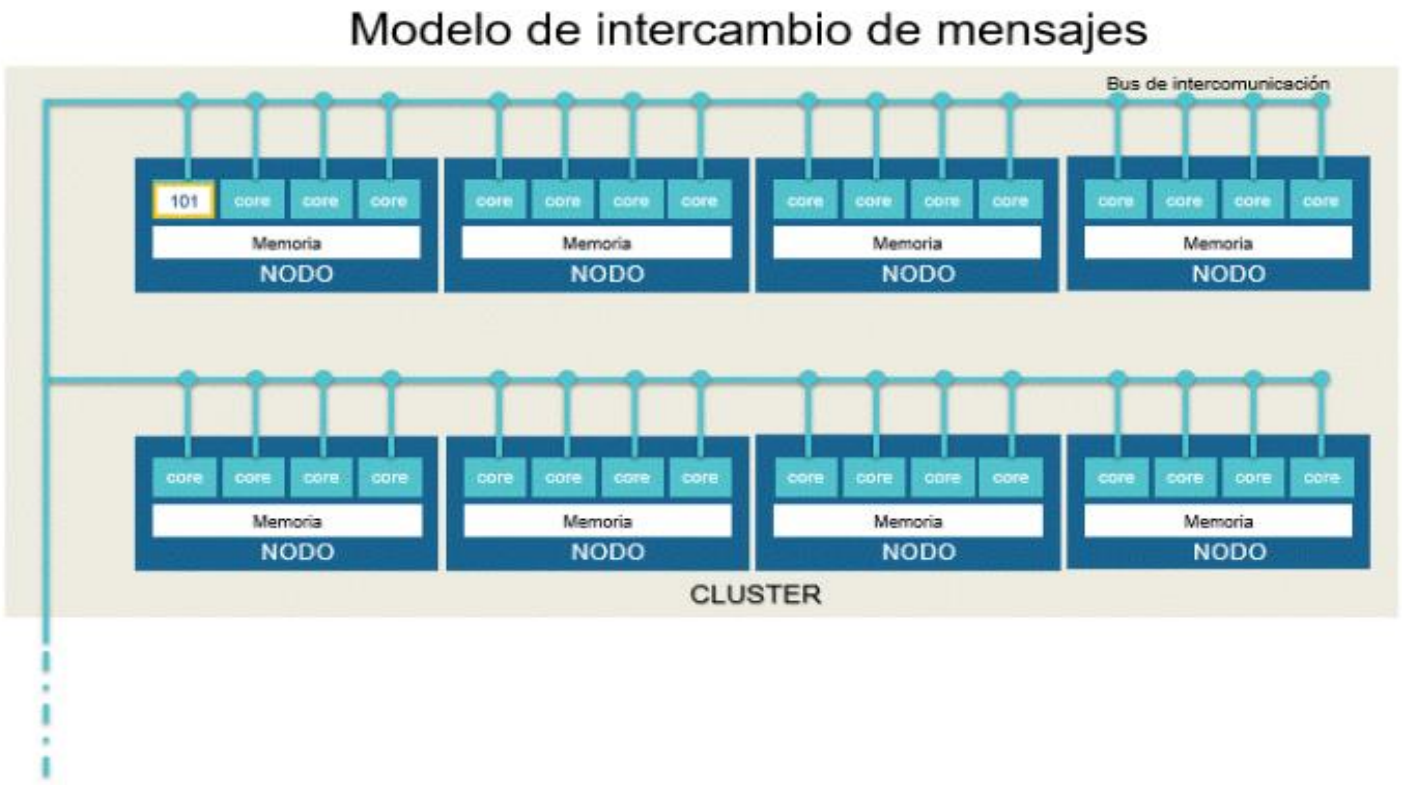
```
#include <mpi.h>
```

- **Desarrollado y mantenido por: MPI Forum**
- **https://www.open-mpi.org/**

1. Introducción a MPI

- **¿Qué es MPI?**
  - Estándar de comunicación para sistemas paralelos
  - Procesos independientes (memoria propia)
  - Comunicación explícita → intercambio de mensajes
  - Pensado para memoria distribuida

- **Cómo distribuye el trabajo**



1. Introducción a MPI

- **Usos Principales**
  - Supercomputadoras
  - Clusters
  - Simulaciones científicas
  - Big Data
  - Modelado climático…

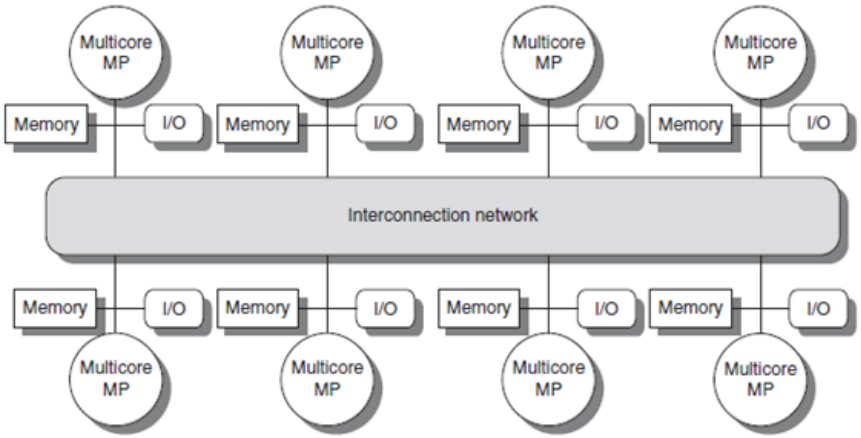
1. Introducción a MPI

- **Características Clave**
  - Portabilidad
  - Escalabilidad
  - Eficiencia
  - Flexibilidad

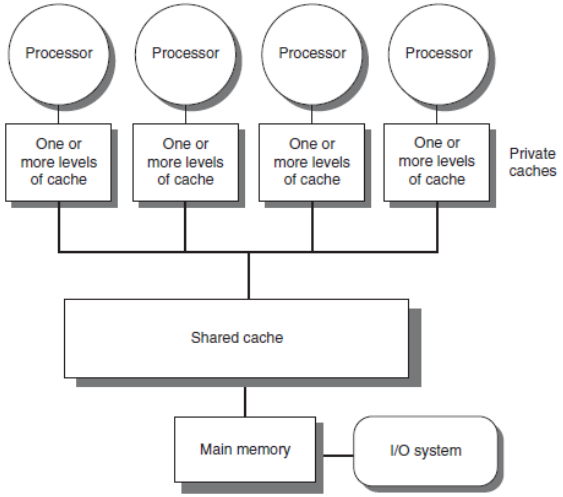
2. Comparación MPI vs OpenMP

2. Comparación MPI vs OpenMP

- **Modelo de Paralelismo**
  - **Dos modelos de arquitectura**
  - **Dos modelos de comunicación**



Clúster

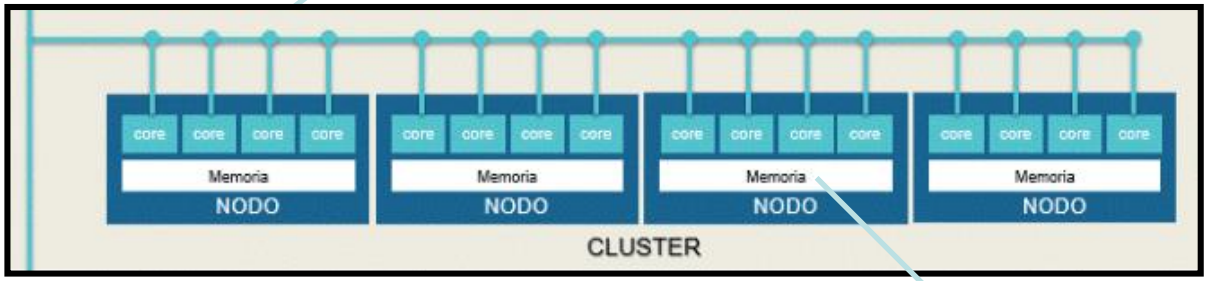


Nodos

2. Comparación MPI vs OpenMP

•Modelo de Paralelismo

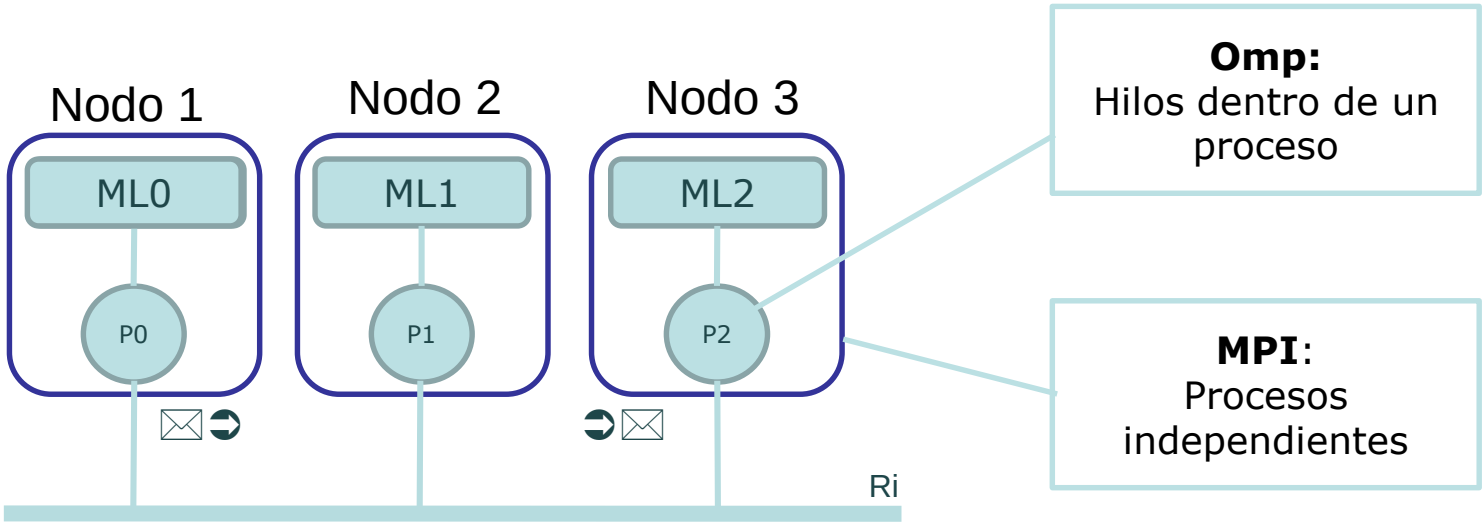
MPI:
- Memoria distribuida
- Procesos independientes
- Cada proceso tiene su propia memoria



OpenMP:
- Memoria compartida
- Hilos en un proceso único
- Todos los hilos acceden a la misma memoria

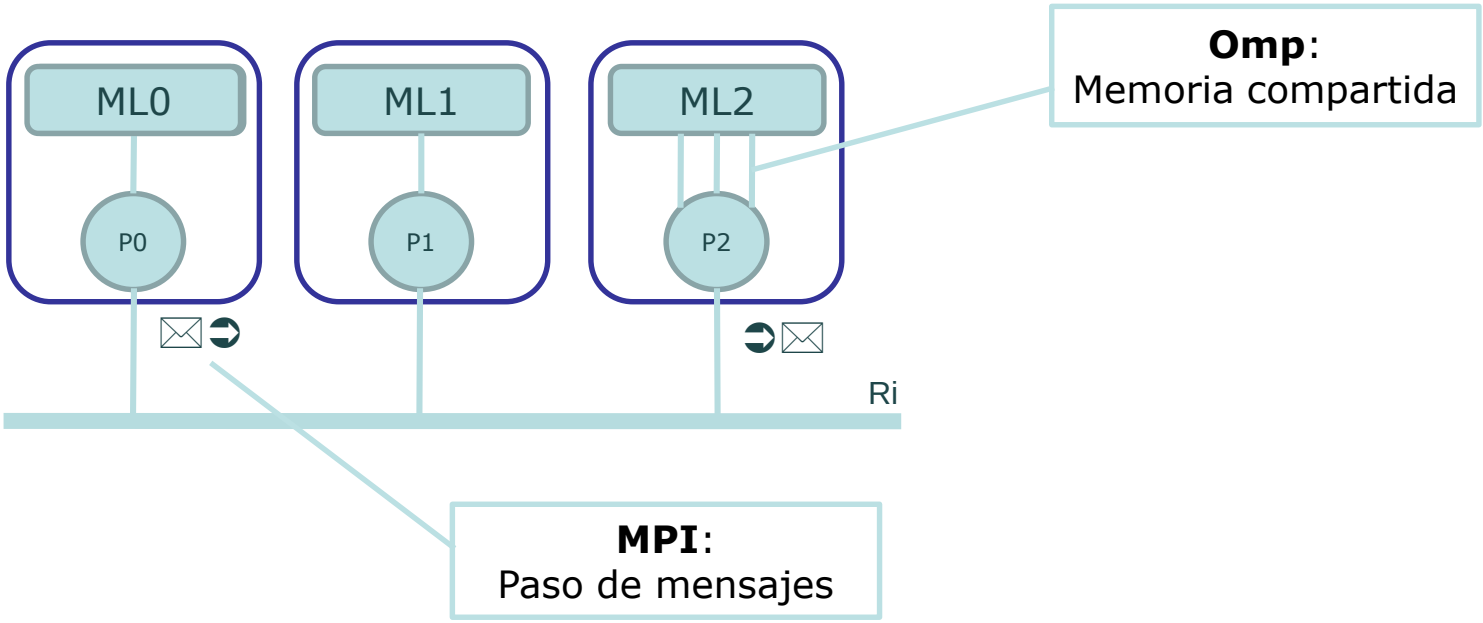
2. Comparación MPI vs OpenMP

- **Unidad de Ejecución**
  - **MPI → Procesos independientes**
  - **OpenMP → Hilos dentro de un proceso**



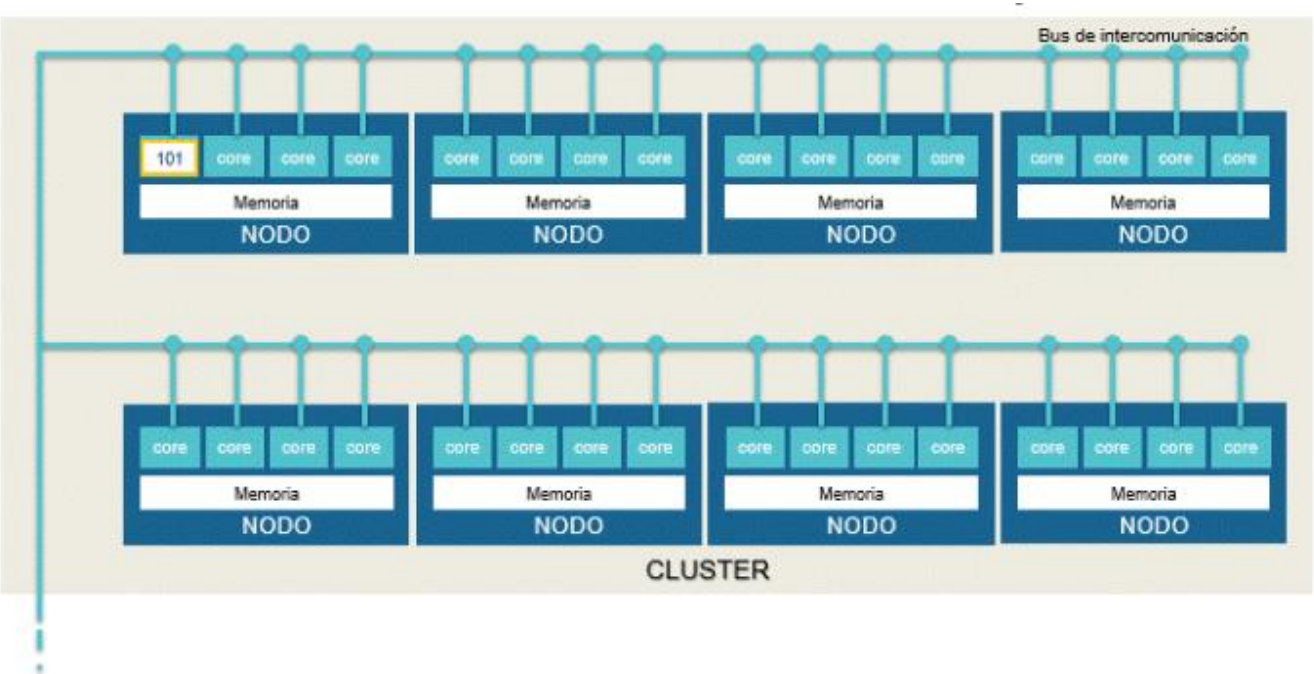
2. Comparación MPI vs OpenMP

- **Comunicación**
  - **MPI → Paso de mensajes**
  - **OpenMP → Variables compartidas**



2. Comparación MPI vs OpenMP

- **Escalabilidad**
  - **MPI → Alta (clusters, supercomputadoras)**
  - **OpenMP → Limitada (máquina única, múltiples núcleos)**



2. Comparación MPI vs OpenMP

- **Complejidad**
  - **OpenPM**
    - Coordinación automática
    - Directivas fáciles (#pragma omp parallel)
    - Baja complejidad:
    - Rápido de implementar
    - Pocos cambios para un programa secuencial
  - **MPI**
    - Cada nodo negocia tiempos y datos
    - Protocolo de envío y recepción manual
    - Alta complejidad:
    - Control de mensajes
    - Sincronización
    - Requiere inicializar y cerrar el entorno MPI manualmente

2. Comparación MPI vs OpenMP

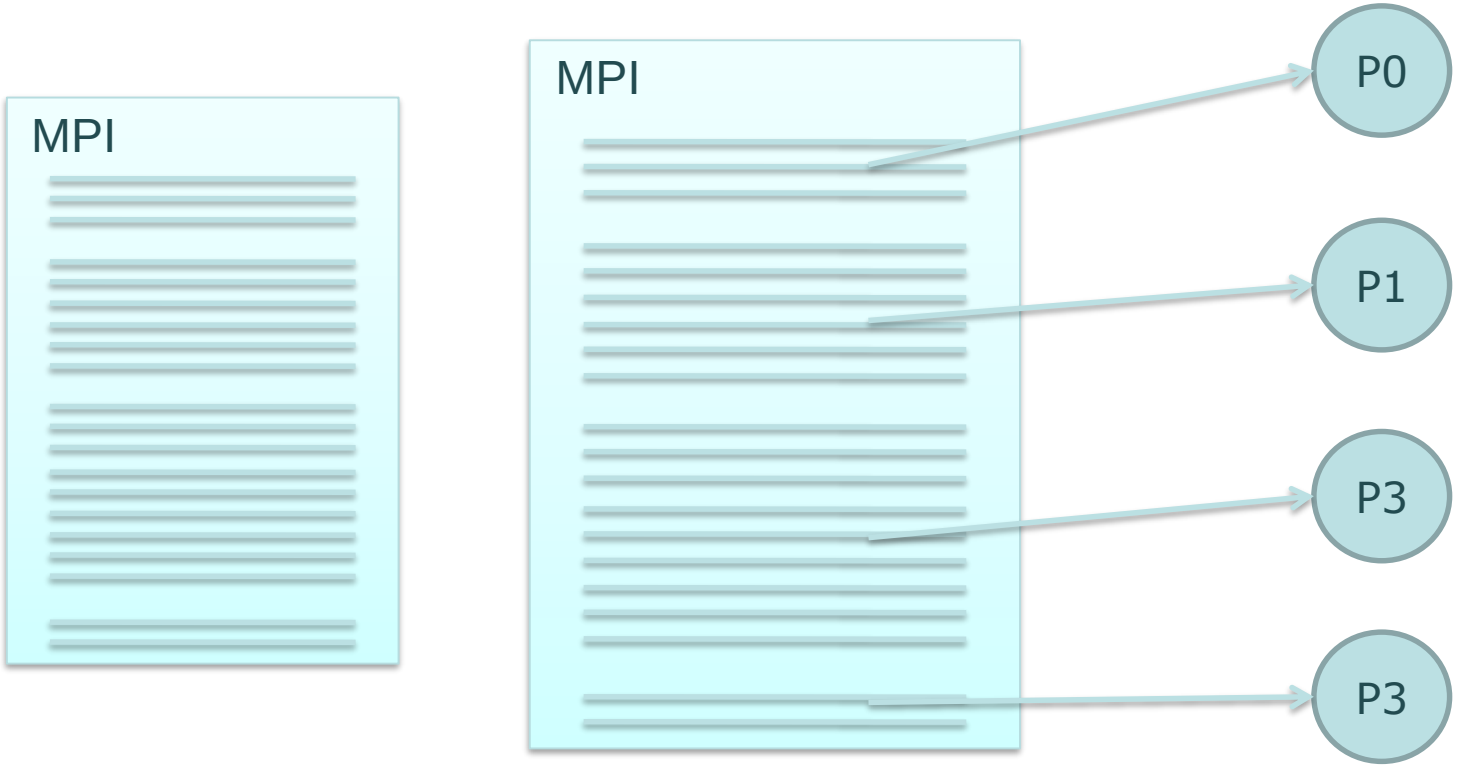
- **Tabla Resumen Comparativa**

Característica	MPI	OpenMP
Memoria	Distribuida	Compartida
Unidad de trabajo	Procesos	Hilos
Comunicación	Paso de mensajes	Memoria compartida
Escalabilidad	Alta (clusters, HPC)	Limitada (máquinas locales)
Complejidad	Alta	Baja
Casos de uso	Supercomputadores, clusters, simulaciones	Multinúcleo, estaciones de trabajo

3. Comunicación entre Procesos MPI

3. Comunicación entre Procesos MPI

- **¿Qué es un Proceso en MIP?**
  - Una unidad independiente de ejecución que forma parte de un programa paralelo.

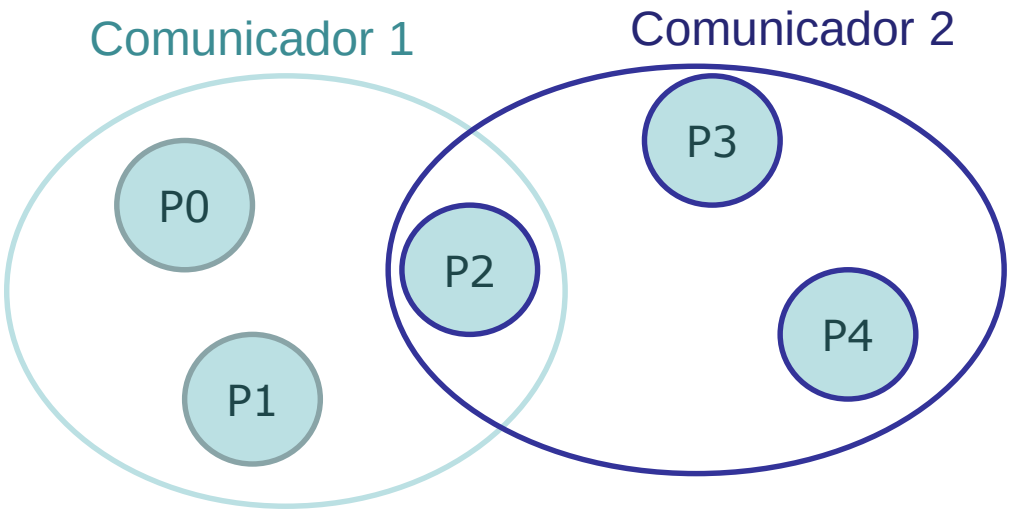


3. Comunicación entre Procesos MPI

- **¿Qué es un Proceso en MIP?**
- **Rank**:
  - Es el identificador único de cada proceso
  - Se una un número que va desde 0 hasta N-1 Procesos
  - Se usa para:
    - Saber qué datos trabajar.
    - Saber a quién enviar mensajes.
- Los Procesos podemos agruparlos dentro de **comunicadores**

3. Comunicación entre Procesos MPI

- **¿Qué es un Comunicador?**



- **Comunicador** = Grupo de procesos + Contexto
- **MPI_COMM_WORLD** (comunicador universal): Incluye todos los procesos
- Comunicador 1 y Comunicador 2: Tienen diferentes contextos

3. Comunicación entre Procesos MPI

•Como se usa un comunicador

- •Siempre está definido el comunicador universal **(MPI_COMM_WORLD),** con todos los Procesos
- •Podemos definir diferentes Comunicadores:
  - •**Ejemplo**: MPI_Comm C1, C2;
- •En toda operación de comunicación **se debe indicar el comunicador usado**, la operación sólo afecta a los procesos de ese comunicador
  - •**MPI_Comm_rank**: Permite conocer el identificador del proceso dentro de un comunicador
  - •**MPI_Comm_size**: Permite conocer el número de procesos en un comunicador
- •**Ejemplo:**

```
int rank, size;
MPI_Comm C1, C2;
MPI_Comm_rank(MPI_COMM_WORLD, &C1);
MPI_Comm_size(MPI_COMM_WORLD, &C2);
```

3. Comunicación entre Procesos MPI

•Comunicación dentro del Comunicador

- Todas las **funciones de comunicación** que veremos en MPI (por ejemplo, Send, Recv, Bcast) **requieren un comunicador**.
- El **Comunicador** define **quién puede hablar con quién**.
- Ejemplo:
  - MPI_Bcast(&**dato**, 1, MPI_INT, **0**, MPI_COMM_WORLD);

- El proceso root (**rank=0**) difunde el **dato** a todos los procesos del comunicador **MPI_COMM_WORLD**.

3. Comunicación entre Procesos MPI

- **Funciones básicas principales (punto a punto):**
  - **MPI_Send()** - Enviar mensaje.
  - **MPI_Recv()** - Recibir mensaje.

send(datos, **P1**) recv(datos, **P0**)

Pi = Proceso
Ri = Red Interconexión

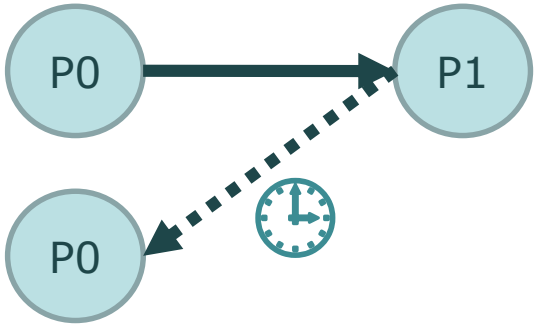
4. Tipos de Comunicación

4. Tipos de Comunicación

- **Comunicación Punto a Punto**
  - **Síncrona** → Espera confirmación de recepción
  - **Asíncrona** → No espera, puede seguir trabajando
  - **Buffered** → Almacenamiento intermedio
- **Comunicación Colectiva**

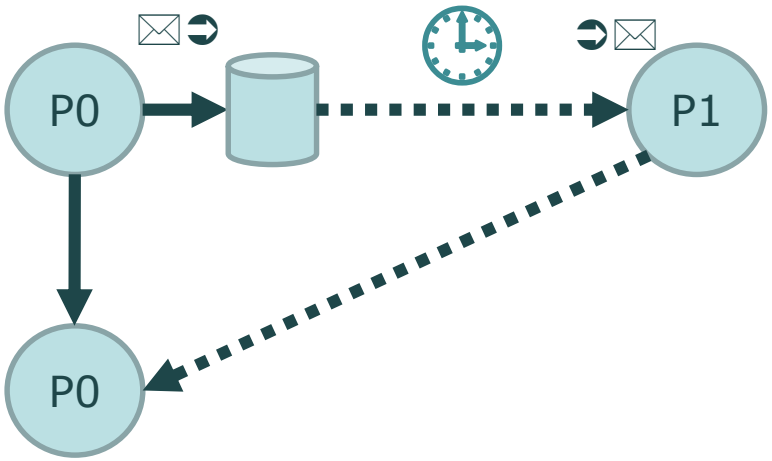
4. Tipos de Comunicación

- **Comunicación Punto a Punto**
- **Síncrona (bloqueante):**
  - Función: **MPI_Ssend()**
  - El emisor espera a que el receptor reciba
  - Asegura sincronización, puede generar esperas



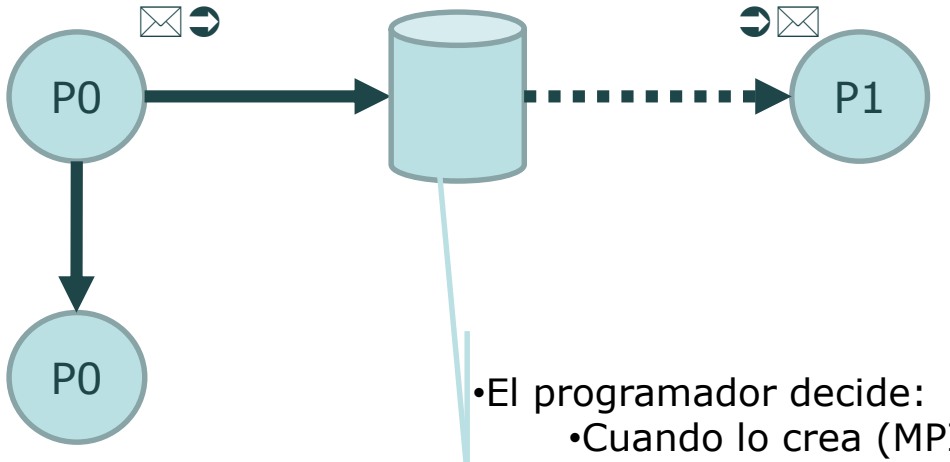
4. Tipos de Comunicación

- **Comunicación Punto a Punto**
- **Asíncrona** (No bloqueante):
  - Función: **MPI_Isend() / MPI_Irecv()**



4. Tipos de Comunicación

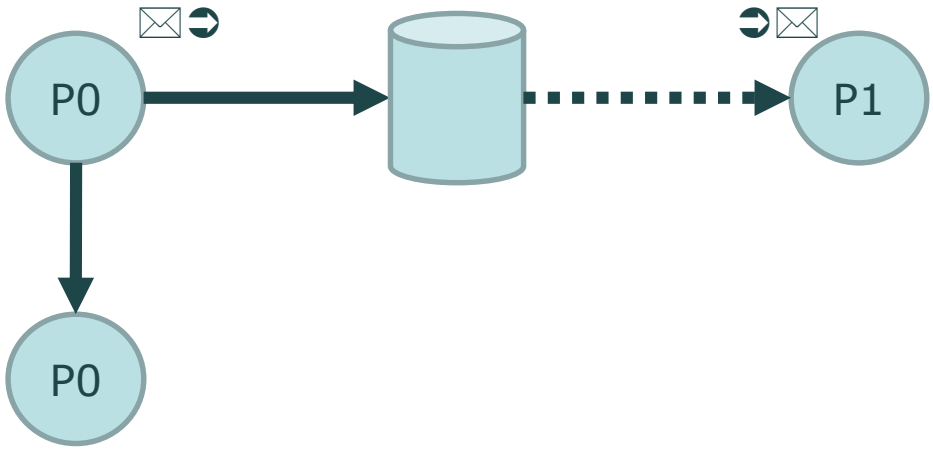
- **Comunicación Punto a Punto**
- **Buffered (No bloqueante):**
  - Función: **MPI_Bsend()**



- El programador decide:
  - Cuando lo crea (MPI_Buffer_attach())
  - Que tamaño tiene (MPI_BSEND_OVERHEAD)
  - Cuando lo libera (MPI_Buffer_detach())

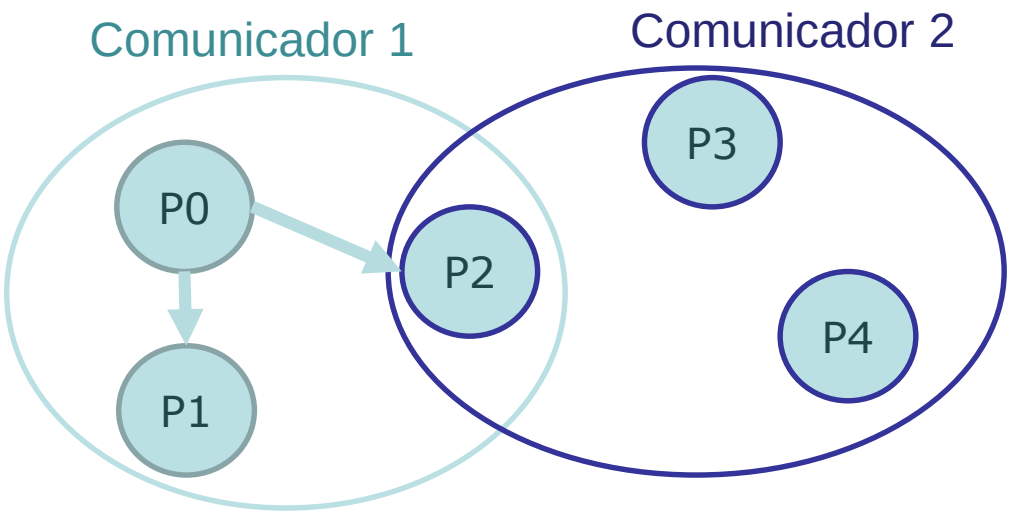
4. Tipos de Comunicación

- **Comunicación Punto a Punto**
- **Buffered (No bloqueante):**
  - **Si el buffer se llena se bloqueará** hasta que haya más espacio (programarlo con atención)
  - Útil para **grandes volúmenes de datos** o comunicaciones programadas



4. Tipos de Comunicación

- **Comunicación Colectiva**
  - Comunicación **entre grupos** de procesos.
  - Todos los procesos participan de forma coordinada.



4. Tipos de Comunicación

- **Comunicación Colectiva**
  - Precisan sincronización implícita.
  - Simplifican el código.
  - Son más eficientes que múltiples comunicaciones punto a punto.

4. Tipos de Comunicación

- **Principales Operaciones de Comunicación Colectiva**

Operación	Descripción
MPI_Bcast()	Difunde datos desde un proceso root a todos
MPI_Scatter()	Divide y reparte datos del root a cada proceso
MPI_Gather()	Recolecta datos desde todos los procesos al root
MPI_Reduce()	Aplica una operación (sumar, max, etc.) y entrega el resultado al root
MPI_Allreduce()	Igual que Reduce pero todos reciben el resultado

Ejemplos de funciones que realizan comunicación colectiva

5. Sincronización

5. Sincronización

- **¿Qué es la Sincronización en MPI?**
  - **Coordinar** procesos paralelos.
  - Asegura que todos los procesos alcancen **un punto común** antes de continuar
  - **Evita condiciones de carrera** en intercambio de datos

5. Sincronización

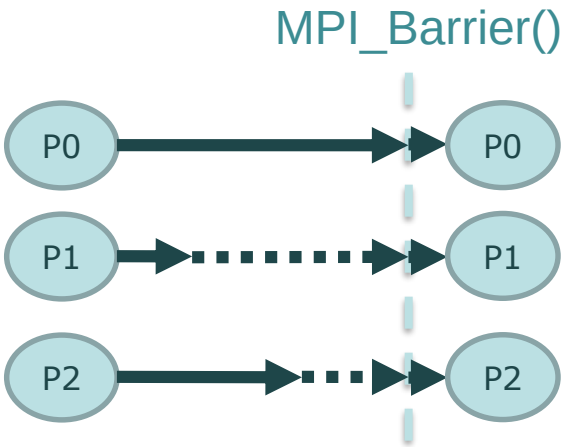
- **Tipos de sincronización en MPI**
  - Sincronización **Global**
    - Todos los procesos esperan en un punto común.

      MPI_Barrier()
  - Sincronización en Comunicación **Punto a Punto**
    - Confirmación del envío/recepción de mensajes.

      MPI_Wait() / MPI_Test()
  - Sincronización en **Comunicación Colectiva**
    - Operaciones como MPI_Bcast(), MPI_Reduce() tienen **sincronización implícita**.
    - Todos los procesos **participan y coordinan** la operación.

5. Sincronización

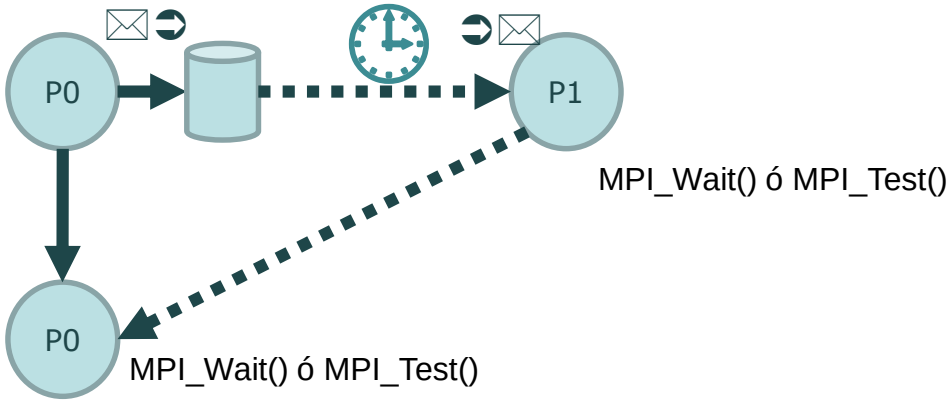
- **MPI_Barrier() Barreras de Sincronización**
  - Todos los Procesos esperan en un punto para ser sincronizados:



  - Coordina fases de trabajo (no transfiere datos).
  - Sincronización **explícita**.

5. Sincronización

- **MPI_Wait() / MPI_Test()**
  - Controlan la finalización local de la comunicación iniciada.
  - Usados después de **MPI_Isend() / MPI_Irecv().**



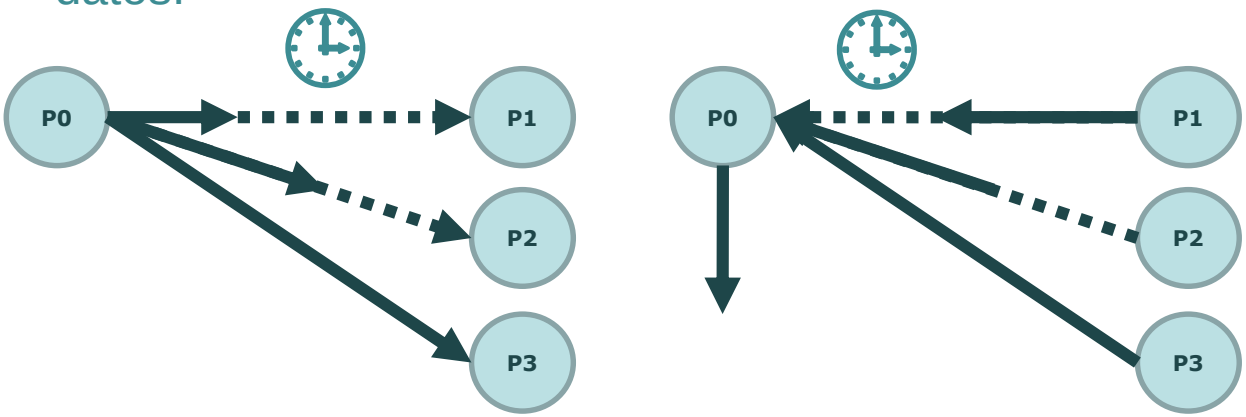
  - Aseguran que una operación **no bloqueante** haya finalizado
  - Controlan **finalización** de la comunicación:
    - MPI_Wait() → bloquea hasta completar la operación.
    - MPI_Test() → verifica sin bloquear.

5. Sincronización

- Sincronización **Implícita** en Operaciones Colectivas
  - **MPI_Bcast()**
  - **MPI_Reduce()**
  - **MPI_Scatter() / MPI_Gather()**
- **Todos los procesos deben llegar** al punto de la llamada para participar.
- La sincronización está **incorporada** en la operación.

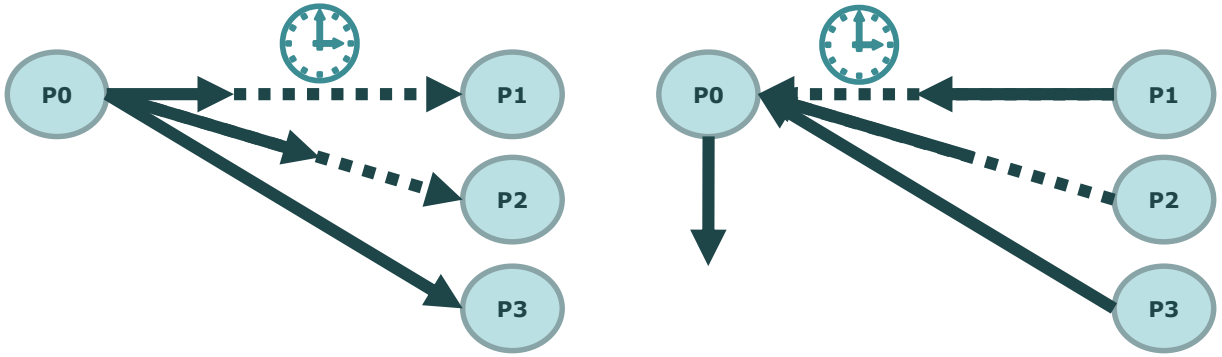
5. Sincronización

- Sincronización **Implícita** en Operaciones Colectivas
  - **MPI_Scatter() / MPI_Gather()**
    - **Scatter reparte** datos desde un proceso a los demás (por ejemplo desde un array).
    - **Gather recoge** datos de todos y los junta en uno solo (por ejemplo en el array).
    - Todos tienen que estar presentes para repartir o juntar datos.



5. Sincronización

- Sincronización **Implícita** en Operaciones Colectivas
  - **MPI_Scatter() / MPI_Gather()**
    - **MPI_Reduce() / MPI_Allreduce()**
    - Todos envían un dato para hacer un **cálculo común** (por ejemplo, sumar números).
    - Nadie puede seguir hasta que todos envíen su dato y el resultado esté listo.
    - Ni siquiera el root continua la ejecución
    - **MPI_Reduce()** : sólo el root recibe los datos / **MPI_Allreduce()** : lo reciben todos



5. Sincronización

- Funciones para sincronización en MPI

- Sincronización Explícita:
- Control directo del programador
- Funciones principales:
  - **MPI_Barrier()** → Sincronización global
  - **MPI_Wait()** → Bloquea hasta completar comunicación no bloqueante
  - **MPI_Test()** → Comprueba el estado de la comunicación sin bloquear.

- Sincronización Implícita:
- Ocurre **automáticamente** en operaciones de comunicación colectiva:
  - **MPI_Bcast()** → Difusión de datos
  - **MPI_Scatter()** → Distribución de datos
  - **MPI_Gather()** → Recolección de datos
  - **MPI_Reduce() / MPI_Allreduce()** → Reducción y distribución de resultados.

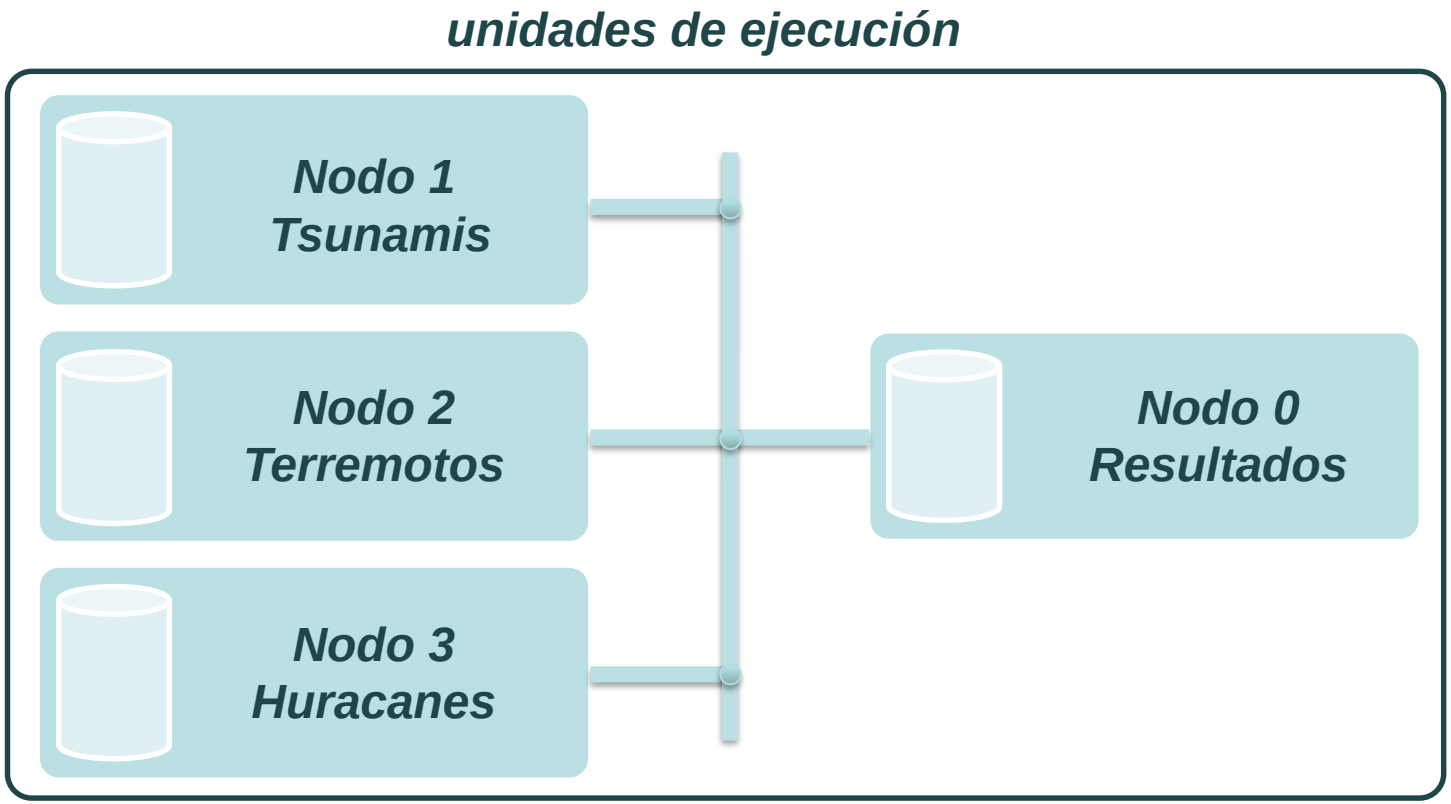
6. Ejemplo conceptual

6. Ejemplo conceptual

- Propuesta Conceptual:
  - Queremos diseñar un **Sistema de Predicción** de Catástrofes Naturales (Tsunamis, Terremotos, Huracanes)
  - La idea es una predicción compleja que combina distintos **modelos de simulación procesados en paralelo**, cada uno en su nodo independiente, y luego unificados.

6. Ejemplo conceptual

- Sistema de predicción de catástrofes naturales

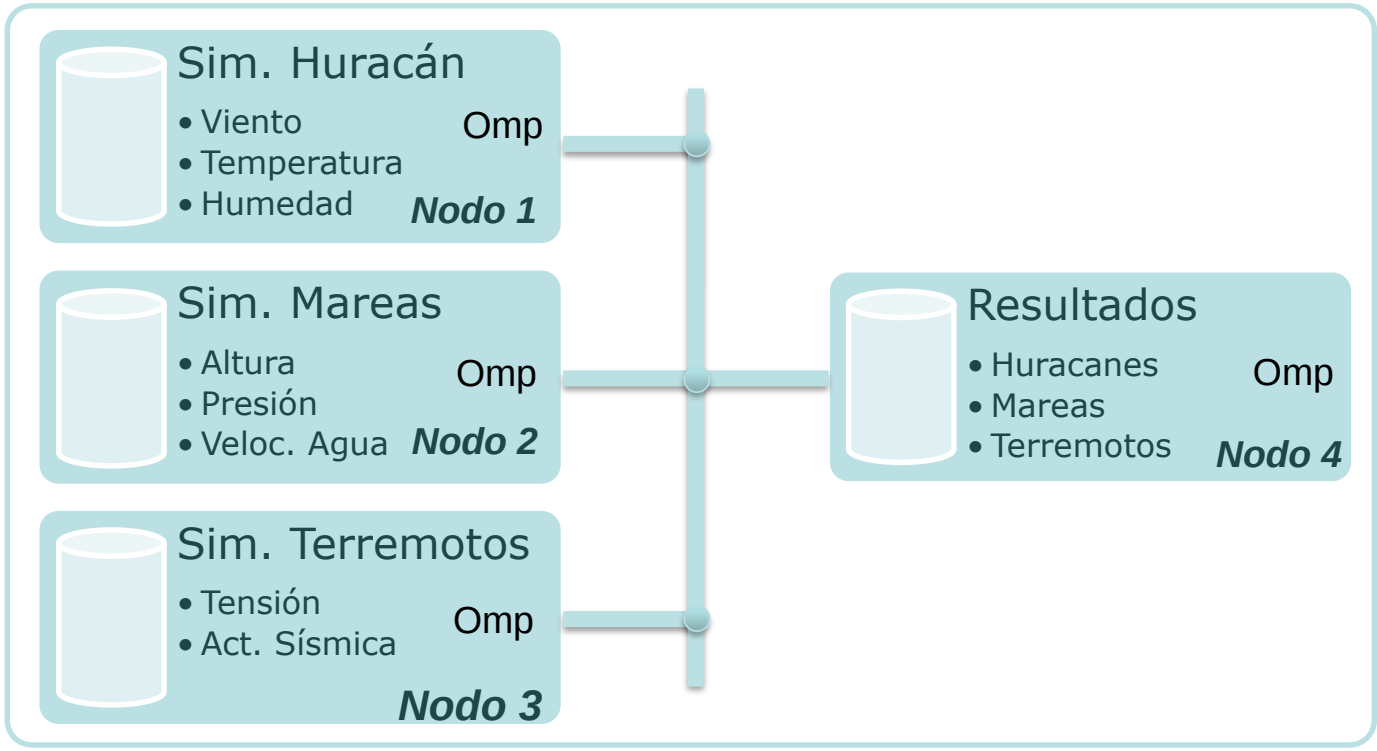


6. Ejemplo conceptual

- **Distribución de Procesos:**
- Cada Nodo un proceso:
  - Ejecuta localmente OpenMP
  - Un **modelo específico** (ej. simulación del comportamiento del océano, análisis de placas tectónicas, predicción de corrientes de aire).
  - Varios **hilos (OpenMP)** procesan **partes de la tarea**, por ejemplo:
    - Hilo 1 → Procesa datos de temperatura.
    - Hilo 2 → Procesa datos de presión.
    - Hilo 3 → Procesa salinidad, etc.
- Todo **compartiendo la misma memoria**: acceden a los mismos datos base.

6. Ejemplo conceptual

- **Distribución de Procesos:**



6. Ejemplo conceptual

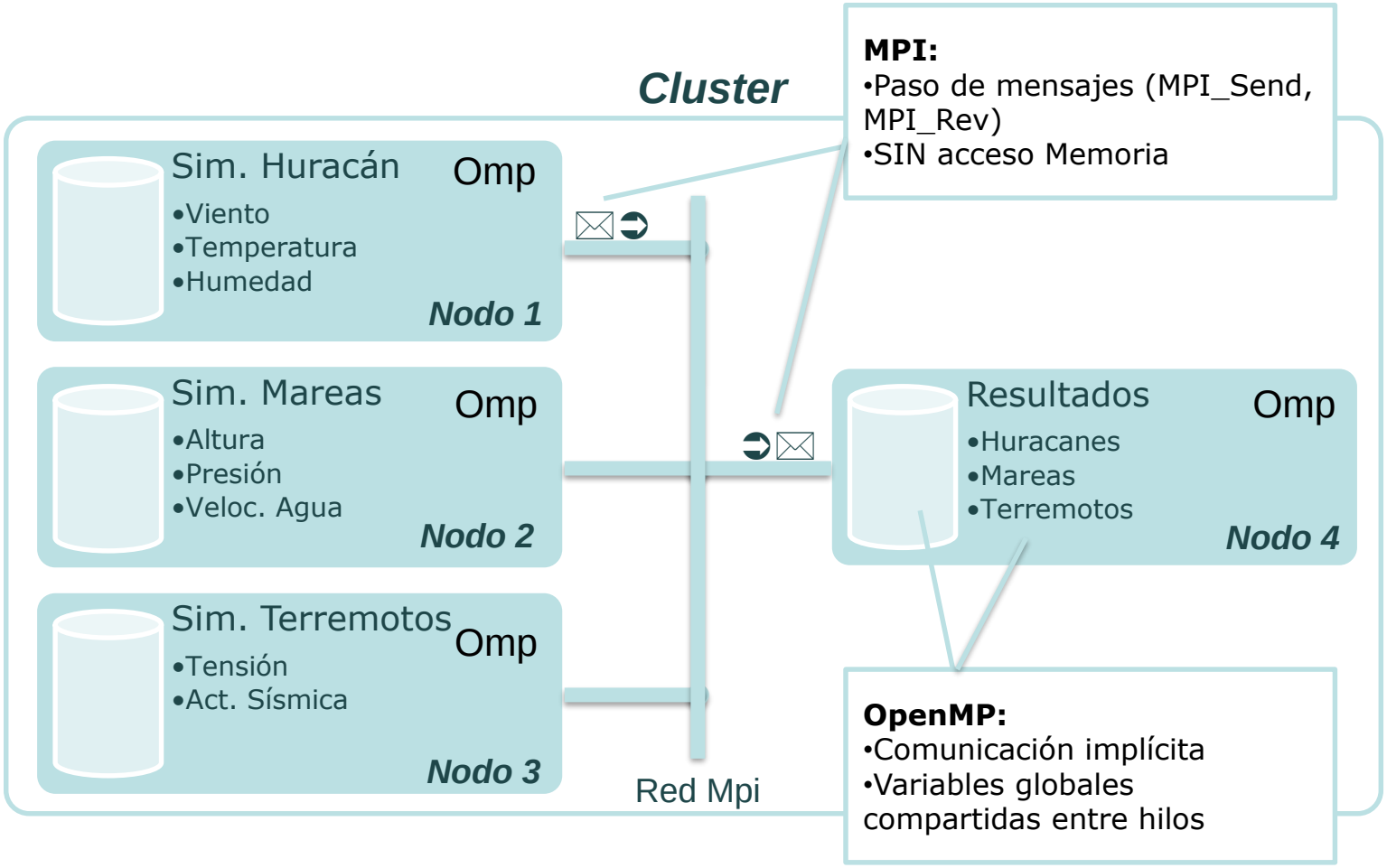
- **Cluster Completo (MPI entre nodos)**
  - Cada nodo trabaja en **un aspecto independiente** del sistema global:
    - Nodo 1 → Simulación de terremotos.
    - Nodo 2 → Simulación de mareas.
    - Nodo 3 → Simulación de huracanes.
  - Los **resultados parciales** de cada nodo se envían mediante **MPI** al nodo maestro o a otros nodos para **integrar la predicción final.**

6. Ejemplo conceptual

- **Cluster Completo (MPI entre nodos)**
  - Cada nodo trabaja en **un aspecto independiente** del sistema global:
    - Nodo 1 → Simulación de terremotos.
    - Nodo 2 → Simulación de mareas.
    - Nodo 3 → Simulación de huracanes.
  - Los **resultados parciales** de cada nodo se envían mediante **MPI** al nodo maestro o a otros nodos para **integrar la predicción final.**

6. Ejemplo conceptual

•Comunicación MPI

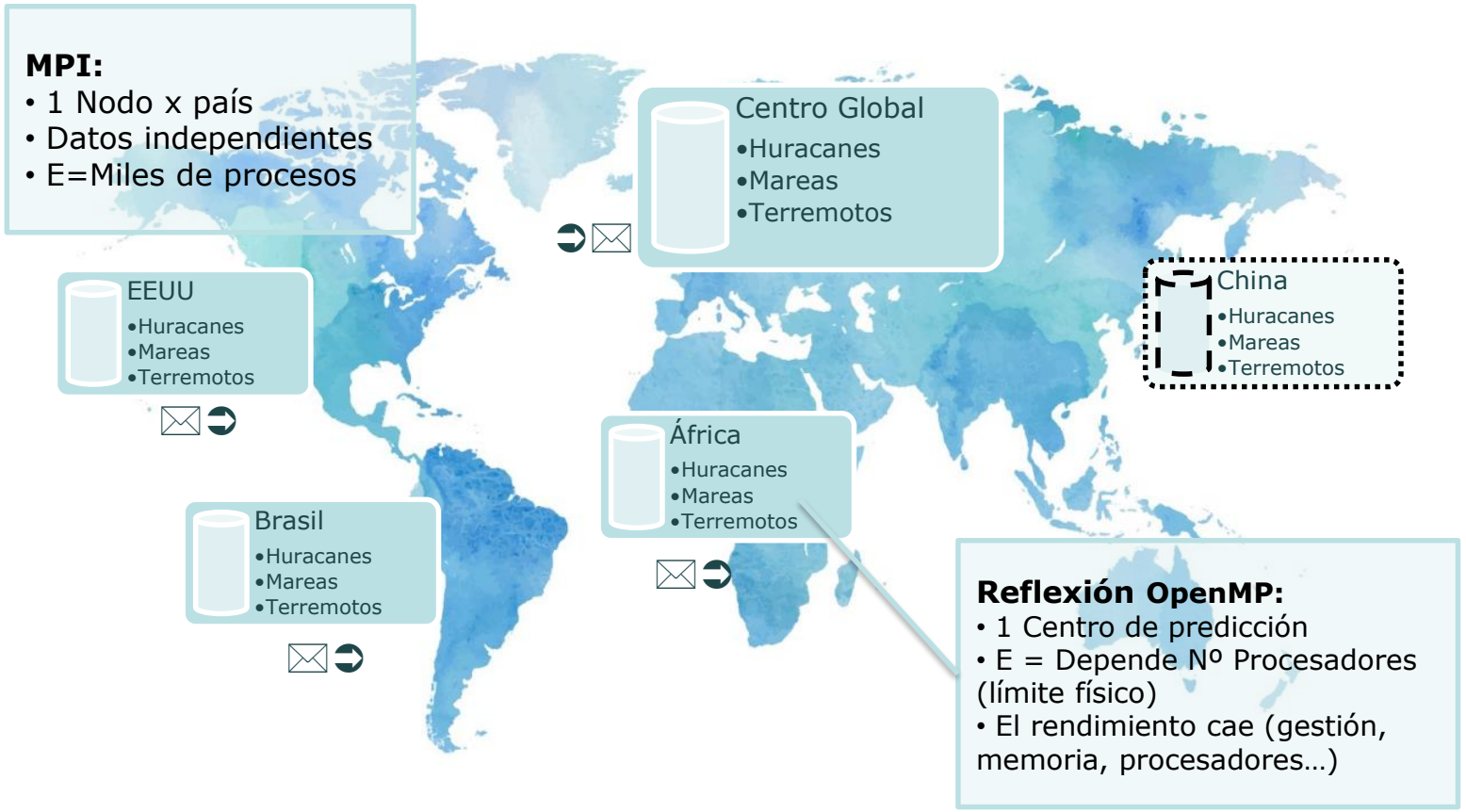


6. Ejemplo conceptual

- **Pregunta sobre Escalabilidad:**
  - ¿Podría escalar a un sistema completo mundial?

6. Ejemplo conceptual

- **Pregunta sobre Escalabilidad:**
  - ¿Podría escalar a un sistema completo mundial?



7. Ejemplos

Ejercicio 0. Estructura básica de un programa MPI

- **Objetivo:**
- Comprender la estructura mínima de un programa en MPI.
- Ver los puntos clave: inicio, comunicación con el entorno, y finalización.

```c
#include <mpi.h>    // Inclusión de la biblioteca MPI
#include <stdio.h>  // Librería estándar para entrada/salida

int main(int argc, char** argv) {

    // Inicializa el entorno MPI
    // Debe ser la primera llamada MPI en el programa
    MPI_Init(&argc, &argv);

    // Aquí iría el código principal del programa MPI

    // Finaliza el entorno MPI
    // Debe ser la última llamada MPI en el programa
    MPI_Finalize();

    return 0;  // Fin del programa
}
```

Argc=Argumento Número
Número entero, que indica **cuántos argumentos se pasaron al programa** desde la línea de comandos.

Argc=Argumento vector
Es un **array de cadenas de caracteres** (char** o char* argv[]), donde cada elemento es uno de esos argumentos.

Ejemplo: si ejecutamos desde la terminal:
mpirun -np 4 ./mi_programa archivo.txt 100

argc será **3** (./mi_programa, archivo.txt, 100)
argv será:
argv[0] → ./mi_programa
argv[1] → archivo.txt
argv[2] → 100

El entorno debe inicializarse y finalizarse claramente con MPI_Init() y MPI_Finalize()

Ejercicio 1: Identificación de Procesos

- **Objetivo:** Aprender cómo cada proceso MPI obtiene su identificador (**rank**) y conoce el número total de procesos (**size**).
- Vamos a añadir a la estructura básica una llamada para identificar los procesos que se han lanzado.
  Cada proceso imprimirá su número de **rank** (ID del proceso) y el número total de procesos que se están ejecutando.

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

int main(int argc, char** argv) {

    // Inicializamos el entorno MPI
    MPI_Init(&argc, &argv);

    int id_proceso;      // ID único del proceso (rank)
    int numero_procesos; // Número total de procesos lanzados

    // Obtenemos el ID del proceso actual (rank)
    MPI_Comm_rank(MPI_COMM_WORLD, &id_proceso);

    // Obtenemos el número total de procesos (size)
    MPI_Comm_size(MPI_COMM_WORLD, &numero_procesos);

    // Imprimimos la información desde cada proceso
    printf("Hola, soy el proceso %d de un total de %d procesos.\n", id_proceso, numero_procesos);

    // Finalizamos el entorno MPI
    MPI_Finalize();

    return 0;
}
```

MPI_Comm_rank()
- Devuelve el identificador (**rank**) del proceso dentro del comunicador global.
- Útil para asignar tareas a procesos distintos.

MPI_Comm_size()
- Devuelve el número total de procesos en el comunicador.
- Permite saber cuántos procesos están participando.

Ejercicio 2: Comunicación Punto a Punto Básica

- **Objetivo:** Aprender cómo dos procesos MPI se comunican directamente utilizando **MPI_Send()** y **MPI_Recv().**
- Este ejercicio muestra la transmisión de un mensaje simple desde un proceso emisor a un proceso receptor, lo que es la base para entender el paso de mensajes en MPI.

```c
#include <mpi.h>    // Librería principal de MPI
#include <stdio.h>  // Librería estándar de entrada/salida

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv); // Inicializamos el entorno MPI
    int id_proceso; // Identificador del proceso actual (rank)
    int numero_procesos; // Número total de procesos lanzados
    MPI_Comm_rank(MPI_COMM_WORLD, &id_proceso); // ID del proceso
    MPI_Comm_size(MPI_COMM_WORLD, &numero_procesos); // Total procesos

    if (numero_procesos < 2) {
        if (id_proceso == 0)
            printf("Este programa necesita al menos 2 procesos.\n");
        MPI_Finalize();
        return 0;
    }

    if (id_proceso == 0) {
        int dato_enviar = 42;
        printf("Proceso %d envía el dato %d al proceso 1.\n", id_proceso, dato_enviar);
        MPI_Send(&dato_enviar, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }

    if (id_proceso == 1) {
        int dato_recibido;
        MPI_Recv(&dato_recibido, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Proceso %d ha recibido el dato %d desde el proceso 0.\n", id_proceso, dato_recibido);
    }

    MPI_Finalize(); // Finalizamos el entorno MPI
    return 0;
}
```

Para que exista una comunicación punto a punto al menos deben comunicar dos procesos

Proceso 0 envía un número entero a **Proceso 1** utilizando **MPI_Send().**

Proceso 1 recibe el número con **MPI_Recv()** y lo muestra en pantalla.

NOTA: La comunicación es bloqueante:
- MPI_Send() espera que MPI_Recv() esté listo en el receptor para que el mensaje se envíe y complete.
- Sincronización implícita entre los dos procesos involucrados.

(Si no lanzas al menos 2 procesos, el envío/recepción no ocurrirá.)

Ejercicio 2: Comunicación Punto a Punto Básica

Notas sobre las funciones usadas.

- **MPI_Send()**
  - Envía un mensaje de un proceso a otro.
  - Sintaxis: MPI_Send(&buffer, count, datatype, destino, tag, comunicador);
- En el ejemplo:
  - **&dato_enviar**: la dirección del dato que se envía.
  - **1**: número de elementos.
  - **MPI_INT:** tipo de dato.
  - **1:** rank del proceso destino.
  - **0:** etiqueta (tag) del mensaje.
  - **MPI_COMM_WORLD:** comunicador.
- **MPI_Recv()**
  - Recibe un mensaje de un proceso.
  - Sintaxis: MPI_Recv(&buffer, count, datatype, origen, tag, comunicador, &status);
- En el ejemplo:
  - **&dato_recibido:** dirección donde se guarda el dato recibido.
  - **1:** número de elementos.
  - **MPI_INT**: tipo de dato.
  - **0:** rank del proceso emisor.
  - **0:** etiqueta (tag) del mensaje.
  - **MPI_COMM_WORLD:** comunicador.
  - **MPI_STATUS_IGNORE:** no nos interesa revisar el estado aquí.

Ejercicio 3: Comunicación en Cadena (Relay de Mensajes)

- **Objetivo:** Aprender cómo varios procesos MPI pueden comunicarse en cadena, donde el mensaje se pasa de uno a otro secuencialmente.
- Este ejercicio permite comprender cómo controlar el flujo de datos en sistemas distribuidos y refuerza el concepto de comunicación punto a punto coordinada.

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int id_proceso, numero_procesos;
    MPI_Comm_rank(MPI_COMM_WORLD, &id_proceso);
    MPI_Comm_size(MPI_COMM_WORLD, &numero_procesos);
    int mensaje;

    if (id_proceso == 0) {
        mensaje = 0; printf("Proceso %d inicia el mensaje con valor %d\n", id_proceso, mensaje);
        MPI_Send(&mensaje, 1, MPI_INT, id_proceso + 1, 0, MPI_COMM_WORLD);
    }

    if (id_proceso > 0) {
        MPI_Recv(&mensaje, 1, MPI_INT, id_proceso - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Proceso %d recibe el mensaje con valor %d\n", id_proceso, mensaje);
        mensaje++;
        if (id_proceso < numero_procesos - 1)
            MPI_Send(&mensaje, 1, MPI_INT, id_proceso + 1, 0, MPI_COMM_WORLD);
        else
            printf("Proceso %d es el último y el valor final es %d\n", id_proceso, mensaje);
    }

    MPI_Finalize(); return 0;
}
```

El proceso 0 **inicia el envío de un mensaje** (un número entero). Y lo envía al siguiente

Recibe el mensaje del proceso anterior (id_proceso - 1).

Cada proceso, al recibir el mensaje, **imprime su valor**, **le suma 1** y lo **envía al siguiente proceso**.

El último proceso finaliza la cadena mostrando el valor final.

Ejercicio 4: Comunicación Colectiva con MPI_Bcast()

- **Objetivo:** Aprender a difundir un valor desde un proceso raíz a todos los demás procesos utilizando la comunicación colectiva MPI_Bcast().
- Este ejercicio permite comprender cómo controlar el flujo de datos en sistemas distribuidos y refuerza el concepto de comunicación punto a punto coordinada.

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int id_proceso, numero_procesos;
    MPI_Comm_rank(MPI_COMM_WORLD, &id_proceso);
    MPI_Comm_size(MPI_COMM_WORLD, &numero_procesos);
    int dato;

    if (id_proceso == 0) {
        dato = 2024; printf("Proceso %d va a difundir el dato %d\n", id_proceso, dato);
    }

    MPI_Bcast(&dato, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Proceso %d ha recibido el dato %d\n", id_proceso, dato);

    MPI_Finalize(); return 0;
}
```

Proceso raíz (por ejemplo, proceso 0) prepara un dato.

Se usa **MPI_Bcast()** para enviar el dato simultáneamente a todos los procesos del comunicador

- En MPI_Bcast(), todos los procesos participan, **incluido el proceso root** (proceso 0 en este caso).
- El dato que difunde **ya lo tiene cargado previamente**, así que **no lo "recibe" desde otro sitio**, pero debe **participar** en la operación colectiva para completar el broadcast.
- Después de MPI_Bcast(), su valor permanece igual que antes de la llamada, pero puede ejecutar el mismo código que el resto para imprimir el valor recibido

Ejercicio 5: Reducción de Valores con MPI_Reduce()

- **Objetivo:** Aprender a combinar valores de todos los procesos y calcular una operación global (por ejemplo, la suma de todos los valores locales) utilizando **MPI_Reduce().** (Puedes cambiar MPI_SUM por otras operaciones como MPI_MAX, MPI_MIN, MPI_PROD, etc.)

Sintaxis: MPI_Reduce(&valor_local, &resultado_global, count, datatype, operacion, root, comunicador);
- En el ejemplo:
  - **&valor_local:** dirección del valor local que aporta cada proceso.
  - **&resultado_global:** dirección donde el root recibirá el resultado.
  - **1:** número de elementos.
  - **MPI_INT:** tipo de dato.
  - **MPI_SUM:** operación de reducción (en este caso, suma).
  - **0:** proceso raíz que recibe el resultado.
  - **MPI_COMM_WORLD:** comunicador.

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int id_proceso, numero_procesos;
    MPI_Comm_rank(MPI_COMM_WORLD, &id_proceso);
    MPI_Comm_size(MPI_COMM_WORLD, &numero_procesos);
    int valor_local = id_proceso + 1;
    int resultado_global;

    printf("Proceso %d tiene el valor local %d\n", id_proceso, valor_local);

    MPI_Reduce(&valor_local, &resultado_global, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (id_proceso == 0)
        printf("Proceso %d muestra el resultado de la suma global = %d\n", id_proceso, resultado_global);

    MPI_Finalize(); return 0;
}
```

Cada proceso genera un valor local (por ejemplo, su rank o cualquier valor entero).

Se usa MPI_Reduce() para combinar todos esos valores en un solo resultado.

El resultado de la operación se envía al proceso raíz, que lo muestra en pantalla. (Los demás procesos no reciben el resultado, aunque sí participan en la operación)