# PRÁCTICA 6: PROCESOS DE ENTRADA/SALIDA

Arquitecturas paralelas, Ingeniería informática, UBU.

Sergio Buil Laliena, Sandro Martín Rubio, Rocío Esteban Valverde

11 noviembre 2023

## Contenido

## Objetivos

Practicar los métodos de envío disponibles en MPI.

## Introducción

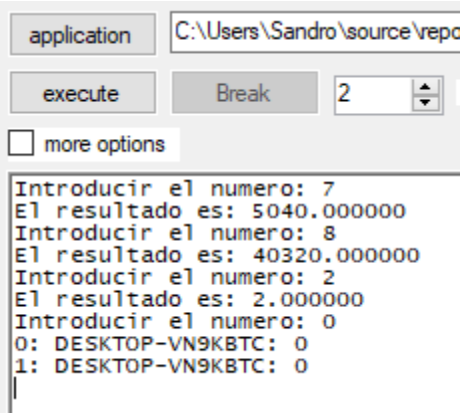
El código proporcionado es un ejemplo de programación paralela utilizando la biblioteca MPI (Message Passing Interface). MPI es ampliamente utilizado en la programación paralela para crear aplicaciones que se ejecutan en sistemas distribuidos o en múltiples procesadores en un clúster. El objetivo de este código es calcular el factorial del número que el usuario introduce por teclado, los procesos que se lancen se encargarán de manera paralela de hacerlo.

## Código

```
/**
* Nombre:PracticaARPA6
* Descripción: Los procesos lanzados calculan el factorial del número que el usuario
escriba por pantalla.
       El resultado se muestra por pantalla y a no ser que el usuario escriba un 0 se
siguen pidiendo números para calcular el factorial.
* Autores: Sergio Buil Laliena, Sandro Martín Rubio, Rocío Esteban Valverde
* Fecha: 11-Nov-2023
*
*/
#include <mpi.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    int mirango, i;
    MPI_Status estado;
    int dato;
    float factorial1, factorial2;
    MPI_Request request1, request2;
    int testigo1, testigo2;

    // Inicialización de MPI y obtención del rango del proceso
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &mirango);

    do {
        // Proceso principal (mirango == 0)
        if (mirango == 0) {
            int numero;
            // Solicitar al usuario ingresar un número
            printf("Introducir el numero: ");
            fflush(stdout);
            scanf("%d", &numero);
            dato = numero;

            // Enviar el número al proceso 1
            MPI_Isend(&dato, 1, MPI_INT, 1, 10, MPI_COMM_WORLD, &request1);
            MPI_Wait(&request1, &estado);
```

```
            // Si el número es 0, abortar todos los procesos
            if (dato == 0) {
                MPI_Abort(MPI_COMM_WORLD, 0);
            }

            // Recibir el resultado del proceso 1 (factorial)
            MPI_Irecv(&factorial2, 1, MPI_FLOAT, 1, 10, MPI_COMM_WORLD, &request2);
            MPI_Test(&request2, &testigo2, &estado);
            while (!testigo2) {
                MPI_Test(&request2, &testigo2, &estado);
            }

            // Mostrar el resultado del cálculo del factorial
            printf("El resultado es: %f\n", factorial2);
        }
        // Proceso secundario (mirango == 1)
        else if (mirango == 1) {
            // Recibir el número del proceso 0
            MPI_Recv(&dato, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &estado);
            factorial1 = 1;

            // Calcular el factorial del número recibido
            for (i = 1; i <= dato; i++) {
                factorial1 = factorial1 * i;
            }

            // Enviar el resultado (factorial) al proceso 0
            MPI_Isend(&factorial1, 1, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, &request2);
            MPI_Wait(&request2, &estado);
        }

    } while (1); // Bucle principal que permite al usuario ingresar múltiples
números

    // Finalización de MPI
    MPI_Finalize();
    return 0;
}
```

## Resultados



# Descripción del Código

## 1. Inicialización de MPI y variables

Se incluyen las bibliotecas necesarias para programar con MPI y para entrada/salida estándar.

## 2. **Declaración de variables:**

Se declaran variables para almacenar el rango del proceso, el estado de la comunicación, el número ingresado por el usuario, los factoriales calculados, y objetos para las solicitudes y testigos de MPI.

## 3. **Inicialización de MPI y obtención del rango:**

Se inicia MPI y se obtiene el rango del proceso actual.

## 4. **Manejo de comunicación y cálculo del factorial**

El programa entra en un bucle que permite al usuario ingresar múltiples números. La condición while(1) indica un bucle infinito.

## 5. Proceso de entrada

En este bloque de código, el proceso con rango 0 (proceso principal) solicita al usuario ingresar un número, lo envía al proceso 1, recibe el resultado del proceso 1 y muestra el resultado del cálculo del factorial.

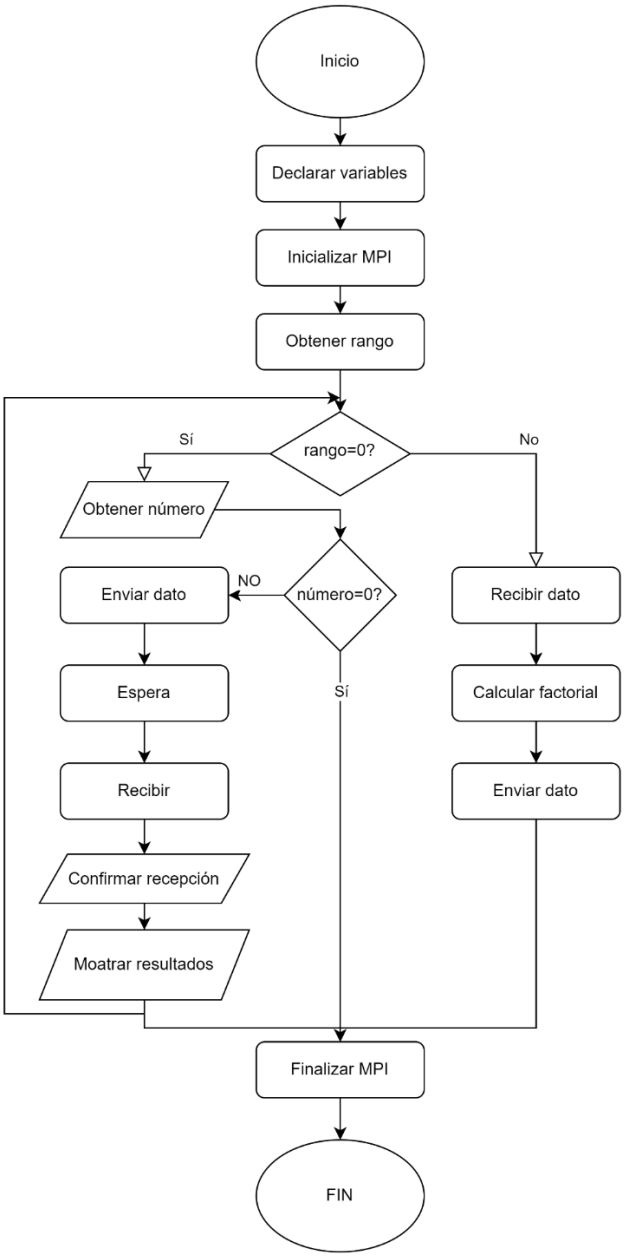
## 6. Proceso de cálculo

En este bloque de código, el proceso con rango 1 (proceso secundario) recibe un número del proceso 0, calcula el factorial de ese número y envía el resultado al proceso 0.

## 7. Finalización de MPI

Se finaliza MPI al salir del bucle y se devuelve 0 para indicar una terminación exitosa del programa.

# Diagrama de flujo



# Cuestiones

## ¿De qué manera se puede aprovechar la potencia de las instrucciones vistas en esta práctica para evitar que los procesos trabajen en ocasiones con datos obsoletos?

MPI_Isend y MPI_Irecv son instrucciones no bloqueantes por lo que los procesos pueden seguir ejecutando cosas no relacionadas con los datos compartidos hasta que MPI_Test y MPI_Wait les indiquen que las comunicaciones se han completado y pueden acceder a los datos más actualizados…

## ¿Se podría producir una situación de abrazo mortal por estar todos los procesos en curso bloqueados en espera de que se complete una petición?

El abrazo mortal en MPI ocurre cuando procesos se bloquean mutuamente, esperando acciones que el otro proceso no puede realizar. Puede suceder con envíos síncronos, no bloqueantes o combinaciones de ambos.

## Realizar una reflexión sobre el concepto de abrazo mortal indicando cómo afecta a los diferentes modos de envío que se conocen.

En MPI, el riesgo de deadlock no depende solo del modo de envío, sino de cómo se estructura el programa. Las combinaciones de envíos y recepciones deben diseñarse cuidadosamente para evitar dependencias circulares que conduzcan a deadlocks.

# Conclusión

El código es un ejemplo de programación paralela utilizando MPI para realizar el cálculo del factorial de los números que se introducen por teclado. Esto permite la ejecución eficiente del cálculo del factorial de un número en un entorno paralelo. El uso de funciones de MPI para mandar mensajes de manera no bloqueante asegura que los procesos se sincronicen adecuadamente y evita problemas de bloqueo entre procesos, de manera que no se pueden producir problemas como deadlocks.