UNIVERSIDAD NEBRIJA

Grado en Ingeniería Informática

Arquitectura de Computadores

Práctica 2

Contenido

Práctica 2: Fundamentos básicos de paralelización: Modelo de comunicación punto a punto

Introducción

MPI define dos tipos de comunicación:

- **Punto a punto (**Fig 1**)**: involucra a 2 procesos (emisor y receptor). Esta comunicación puede ser:
  - Bloqueante: El proceso queda bloqueado hasta que la operación finalice (MPI_Send/MPI_Recv).
  - No bloqueante: el proceso no espera a la finalización de la operación de comunicación y puede continuar solapando cómputo y comunicación. Debe comprobar más adelante si ha terminado la operación (MPI_Isend/MPI_Irecv)
- **Colectiva (**Fig 2**)**: Involucra a todos los procesos de un comunicador.

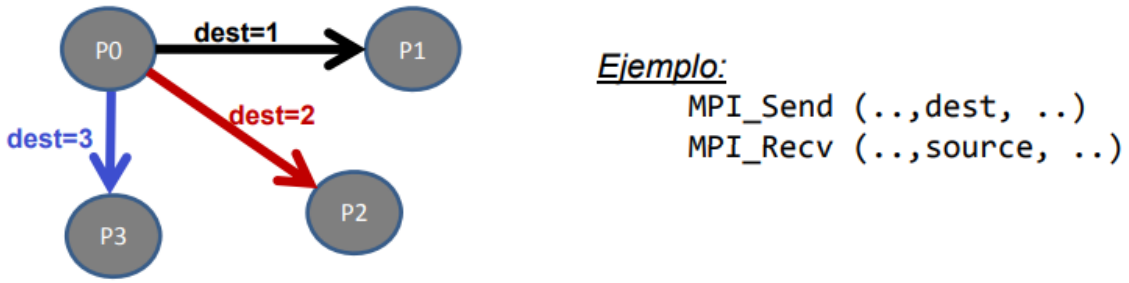


Fig 1 Comunicación punto a punto

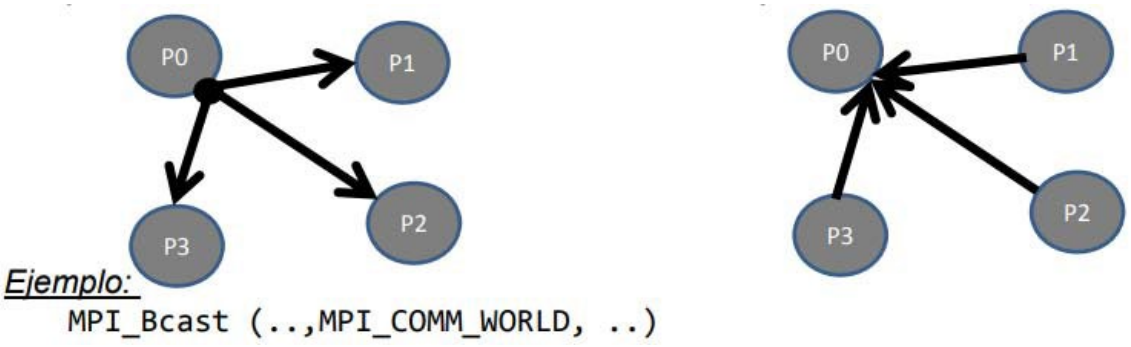


Fig 2 Comunicación colectiva

Funciones send y receive bloqueantes

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

IN	buf	initial address of send buffer
IN	count	number of entries to send
IN	datatype	datatype of each entry
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator

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

OUT	buf	initial address of receive buffer
IN	count	max # of entries to receive
IN	datatype	datatype of each entry
IN	source	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	status	return status (inf. acerca del mensaje recibido)

Ejemplo de comunicación punto a punto

```
#include <stdio.h>
#include "mpi.h"
 int main(int argc, char **argv) {
  int rank, count;
  char msg[20];
  MPI_Status status;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...

 if (rank==0) {
    printf ("I am master. Sending the message.\n\n");
    strcpy(msg,"Hello World!");
    MPI_Send(msg, 13, MPI_CHAR, 1, 100, MPI_COMM_WORLD);
  }
 else {
    printf ("I am the slave %d. Receiving the message.\n", rank);
    MPI_Recv(msg, 13, MPI_CHAR, 0, 100, MPI_COMM_WORLD, &status);
    printf ("The message is: %s\n", msg);

    MPI_Get_count(&status, MPI_CHAR, &count)
    printf("Recibidos %d caracteres de %d con tag= %d\n",
              count, status.MPI_SOURCE, status.MPI_TAG);
 }
  MPI_Finalize();
}
```

Objetivos

En esta práctica se aprenderá el concepto de comunicación punto a punto entre procesos de MPI.

Los objetivos fijados son los siguientes:

- Compilación de programas MPI.
- Ejecución de programas en varios procesos de forma paralela.
- Estructura de un programa MPI.
  - Iniciar y finalizar el entorno MPI con MPI_Init y MPI_Finalize.
  - Identificador (rango) del proceso con MPI_Comm_rank.
  - Consultar el número de procesos lanzados con MPI_Comm_size.
- Primitivas send y recieve.

Compilación y ejecución de programas MPI

Compilación: `mpicc codigo_fuente.c -o ejecutable`

Ejecución: `mpirun –np <number> ejecutable`

Entregables

- Memoria
- Código fuente de los siguientes ejercicios:

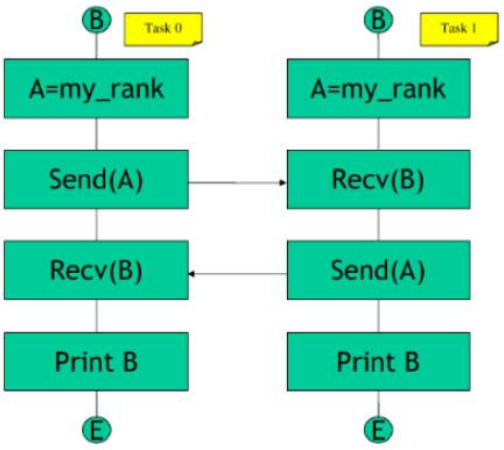
Ejercicio 1

Implementar un código donde utilizando comunicación punto a punto dos procesos rebotan continuamente los mensajes entre sí, hasta que deciden detenerse una vez alcanzado límite autoimpuesto.

Ejercicio 2

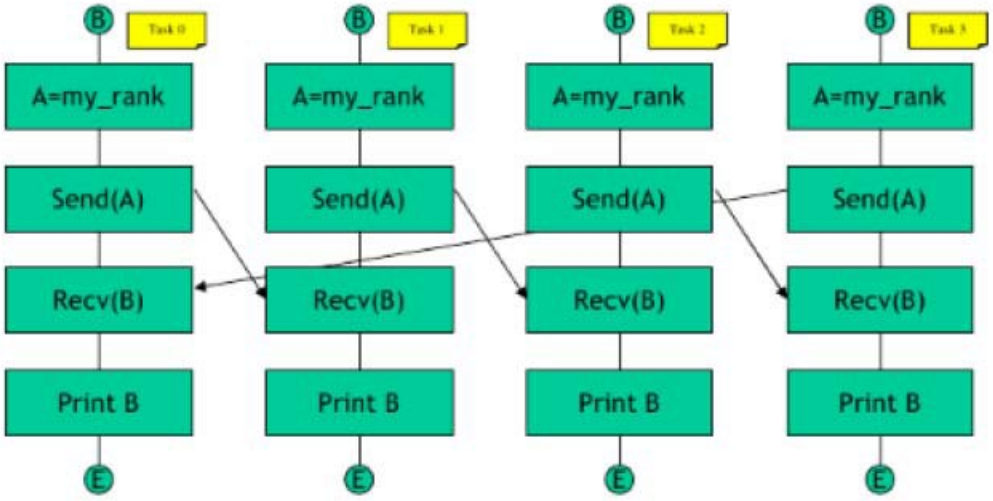
Implementar un código, que comunica dos procesos usando comunicación punto a punto. Cada proceso debe enviar un array de datos al otro. Para ello, cada proceso declara dos arrays de floats, A y B, de tamaño fijo (10000). Todas las posiciones del array A se inicializan con el rango respectivos de cada proceso. Los arrays A y B serán los buffers para las operaciones de SEND y RECEIVE, respectivamente.

Se debe implementar un contador que se incremente cada vez que el primer proceso envía el mensaje. El programa termina cuando el contador llegue a un límite autoimpuesto. El programa debe guiarse por el siguiente esquema:



Ejercicio 3

Implementar un código que usando comunicación punto a punto, lleva a cabo una operación de send/recieve circular, como muestra el siguiente esquema:



Al igual que en el ejercicio anterior, cada proceso genera sus dos buffers A y B con un tamaño de 1000 posiciones inicializadas con sus respectivos rangos en A. Los arrays de tipo B se usan para recibir los mensajes que lleguen del nodo origen. Como se observa, cada proceso envía al que tiene a su derecha y recibe solo del que tiene a su izquierda. El programa termina cuando todos los procesos hayan enviado y recibido un mensaje. Asegurarse de que el programa funcione para un número de procesos arbitrario.

Entrega

Es imprescindible subir los entregables a la actividad del **campus** y enviármelos por **correo** (lrodriguezso@nebrija.es) antes del **25 de noviembre a las 7:59 am**.