Arquitectura de Computadores

Práctica 1

Jose Francisco Romero Rodríguez

Sara Marcos Cornejo

UNIVERSIDAD NEBRIJA

Indice:

Introducción:

En esta práctica se aprenderán los aspectos generales de la programación en MPI así como el entorno sobre el que se desarrollarán las sesiones de prácticas.

MPI es una especificación para programación de paso de mensajes, que proporciona una librería de funciones para C, C++ o Fortran que son empleadas en los programas para comunicar datos entre procesos.

El usuario escribirá su aplicación como un proceso secuencial del que se lanzarán varias instancias que cooperan entre sí. Los procesos invocan diferentes funciones MPI que permiten:

- iniciar, gestionar y finalizar procesos MPI
- comunicar datos entre dos procesos
- realizar operaciones de comunicación entre grupos de procesos
- crear tipos arbitrarios de datos

Algunas de sus características son:

- Interfaz genérica que permite una implementación optimizada en cualquier sistema paralelo.
- Define varias formas de comunicación lo que permite programar de manera natural cualquier algoritmo en paralelo.
- Está pensado para crear bibliotecas paralelas.

Objetivos:

- Preparación del entorno de trabajo: instalación MPI.
- 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.

Ejercicio 1:

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

int main(int argc, char** argv){
        int nproc;
        int myrank;
        double start, finish, tiempo;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD,&nproc);
        MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    MPI_Barrier(MPI_COMM_WORLD);
    start = MPI_Wtime();

    printf("Hola mundo, soy el proceso %d de un total de %d\n",myrank,nproc);

    MPI_Barrier(MPI_COMM_WORLD);
    finish = MPI_Wtime();
    tiempo = finish-start;
    printf("Tiempo de ejecucion proceso %d en segundos:%f\n", myrank, tiempo);
        MPI_Finalize();
}
```

En el ejercicio 1 se realiza un programa que imprima por pantalla Hola Mundo, seguido de su número de proceso en función del rango total que es introducido al ejecutar el programa.

Para esto, se usan las funciones:

- **MPI_Init:** Para inicializar la aplicación
- **MPI_Comm_rank (comm, &pid)**: Devuelve en pid el id del proceso actual
- **MPI_Comm_size (comm, &npr):** Devuelve en npr el número total de procesos del comunicador
- A continuación, se utiliza la función **MPI_Wtime**, que devuelve un tiempo transcurrido en el procesador de llamadas. Se ejecuta después de la función **MPI_Barrier(MPI_Comm comm)** para lograr una sincronización global entre todos los procesos del comunicador.
- Se ejecuta la instrucción de print, llamando después a la funcion PI_Barrier(MPI_Comm comm) de nuevo para que estén sincronizados correctamente. Restando el tiempo de fin al de inicio se puede calcular el tiempo de ejecución de cada proceso.
- Por último, mediante **MPI_Finalize(void)**, finaliza la ejecución.

Resultados:

Para 1 proceso:

```
sara@sara-vivobook:~/Downloads$ mpicc ej1prueba.c -o ejecutable_ej1
sara@sara-vivobook:~/Downloads$ mpirun -np 1 ./ejecutable_ej1
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
Hola mundo, soy el proceso 0 de un total de 1
Tiempo de ejecucion proceso 0 en segundos:0.000005
sara@sara-vivobook:~/Downloads$ S
```

Para 20 procesos:

```
Hola mundo, soy el proceso 0 de un total de 20
Hola mundo, soy el proceso 8 de un total de 20
Hola mundo, soy el proceso 9 de un total de 20
Hola mundo, soy el proceso 1 de un total de 20
Hola mundo, soy el proceso 16 de un total de 20
Hola mundo, soy el proceso 17 de un total de 20
Hola mundo, soy el proceso 2 de un total de 20
Hola mundo, soy el proceso 3 de un total de 20
Hola mundo, soy el proceso 4 de un total de 20
Hola mundo, soy el proceso 6 de un total de 20
Hola mundo, soy el proceso 12 de un total de 20
Hola mundo, soy el proceso 18 de un total de 20
Hola mundo, soy el proceso 19 de un total de 20
Hola mundo, soy el proceso 14 de un total de 20
Hola mundo, soy el proceso 15 de un total de 20
Hola mundo, soy el proceso 10 de un total de 20
Hola mundo, soy el proceso 11 de un total de 20
Hola mundo, soy el proceso 7 de un total de 20
Hola mundo, soy el proceso 13 de un total de 20
Hola mundo, soy el proceso 5 de un total de 20
Tiempo de ejecucion proceso 0 en segundos:0.075503
Tiempo de ejecucion proceso 16 en segundos:0.068007
Tiempo de ejecucion proceso 18 en segundos:0.063983
Tiempo de ejecucion proceso 19 en segundos:0.063983
Tiempo de ejecucion proceso 1 en segundos:0.080840
Tiempo de ejecucion proceso 8 en segundos:0.088023
Tiempo de ejecucion proceso 10 en segundos:0.050715
Tiempo de ejecucion proceso 17 en segundos:0.085047
Tiempo de ejecucion proceso 12 en segundos:0.076012
Tiempo de ejecucion proceso 11 en segundos:0.053098
Tiempo de ejecucion proceso 9 en segundos:0.092226
Tiempo de ejecucion proceso 4 en segundos:0.088032
Tiempo de ejecucion proceso 5 en segundos:0.056061
Tiempo de ejecucion proceso 6 en segundos:0.088670
Tiempo de ejecucion proceso 2 en segundos:0.100006
Tiempo de ejecucion proceso 13 en segundos:0.067965
Tiempo de ejecucion proceso 3 en segundos:0.104408
Tiempo de ejecucion proceso 14 en segundos:0.096025
Tiempo de ejecucion proceso 15 en segundos:0.103703
Tiempo de ejecucion proceso 7 en segundos:0.092022
sara@sara-vivobook:~/Downloads$ S
```

Para 50 procesos:

```
Hola mundo, soy el proceso 47 de un total de 50
Hola mundo, soy el proceso 6 de un total de 50
Hola mundo, soy el proceso 7 de un total de 50
Hola mundo, soy el proceso 15 de un total de 50
Tiempo de ejecucion proceso 0 en segundos:0.196006
Tiempo de ejecucion proceso 32 en segundos:0.207991
Tiempo de ejecucion proceso 48 en segundos:0.179974
Tiempo de ejecucion proceso 1 en segundos:0.197799
Tiempo de ejecucion proceso 4 en segundos:0.136867
Tiempo de ejecucion proceso 6 en segundos:0.088011
Tiempo de ejecucion proceso 36 en segundos:0.212006
Tiempo de ejecucion proceso 34 en segundos:0.193255
Tiempo de ejecucion proceso 5 en segundos:0.139999
Tiempo de ejecucion proceso 33 en segundos:0.183998
Tiempo de ejecucion proceso 49 en segundos:0.183924
Tiempo de ejecucion proceso 16 en segundos:0.202194
Tiempo de ejecucion proceso 8 en segundos:0.223989
Tiempo de ejecucion proceso 35 en segundos:0.200003
Tiempo de ejecucion proceso 38 en segundos:0.184020
Tiempo de ejecucion proceso 7 en segundos:0.104168
Tiempo de ejecucion proceso 24 en segundos:0.195977
Tiempo de ejecucion proceso 26 en segundos:0.192002
Tiempo de ejecucion proceso 28 en segundos:0.192004
Tiempo de ejecucion proceso 37 en segundos:0.228003
Tiempo de ejecucion proceso 20 en segundos:0.188018
Tiempo de ejecucion proceso 12 en segundos:0.135994
Tiempo de ejecucion proceso 29 en segundos:0.171989
Tiempo de ejecucion proceso 13 en segundos:0.127988
Tiempo de ejecucion proceso 40 en segundos:0.211961
Tiempo de ejecucion proceso 44 en segundos:0.204098
Tiempo de ejecucion proceso 45 en segundos:0.184109
Tiempo de ejecucion proceso 18 en segundos:0.188166
Tiempo de ejecucion proceso 10 en segundos:0.217293
Tiempo de ejecucion proceso 39 en segundos:0.171982
Tiempo de ejecucion proceso 46 en segundos:0.148002
Tiempo de ejecucion proceso 22 en segundos:0.200006
Tiempo de ejecucion proceso 2 en segundos:0.243982
Tiempo de ejecucion proceso 30 en segundos:0.179984
```

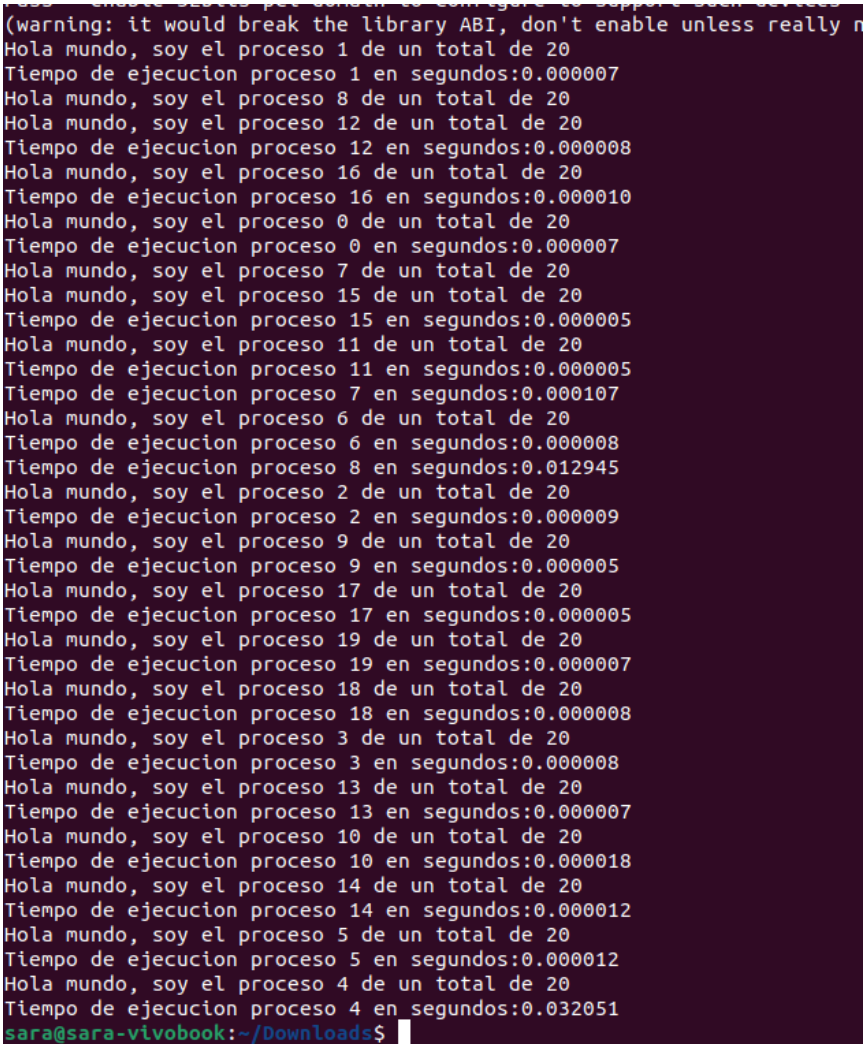
Como se puede observar, según se van añadiendo más procesos, el tiempo de ejecución de cada proceso aumenta, siendo el tiempo mínimo de ejecución/proceso:

- Para 1 proceso: 0.000005s
- Para 20 procesos: 0.050715s
- Para 50 procesos: 0.136867s

1. ¿Los procesos imprimen el hola mundo y el tiempo en orden? ¿A qué se debe esto?

Primero imprime el hola mundo para todos los procesos, y luego el tiempo de ejecución, pero no siguen un orden en cuanto a tiempo en segundos. Esto se debe al uso de barreras y a la programación paralela, el orden de ejecución puede variar como se aprecia arriba.

2. ¿Qué pasa con el tiempo de ejecución si eliminamos las barreras (MPI_Barrier())? ¿Y con el orden de ejecución? ¿Porqué?



Al eliminar las barreras, el tiempo de ejecución de cada proceso disminuye, y el orden de ejecución no están sincronizados. A través de MPI_Barrier() se bloquea al proceso hasta que todos los procesos pertenecientes al comunicador especificado lo ejecuten, es decir, todos los procesos de un comunicador dado esperarán dentro del MPI_Barrier() hasta que todos estén dentro, y entonces, dejarán la operación.

Debido a que tiene que esperar a que terminen los procesos, el tiempo de ejecución con las barreras será mayor que sin ellas.

Por otra parte, al usar barreras todos los procesos esperarían hasta que se haya ejecutado el print de hola mundo para todos los procesos, pero aquí no necesitan esperar, y el print del tiempo de ejecución se ejecuta después del print de hola mundo de cada proceso.

Ejercicio 2:

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

int main(int argc, char *argv[])
{
    int rank,size,num;
    MPI_Status estado;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);


    if(rank == 0){
    printf("Introduce el numero que desea transmitir: ");
    scanf("%d", &num);

        printf("Proceso %d envia %d al proceso %d\n", rank, num, rank+1);
        MPI_Send(&num,1 ,MPI_INT,rank+1,0,MPI_COMM_WORLD);

    }else{
        MPI_Recv(&num,1 ,MPI_INT,rank-1,0,MPI_COMM_WORLD,&estado);

        printf("Soy el proceso %d y he recibido %d\n",rank, num);
        if(rank != size-1){
        printf("Proceso %d envia %d al proceso %d\n", rank, num, rank+1);
            MPI_Send(&num, 1 ,MPI_INT ,rank+1 , 0 ,MPI_COMM_WORLD);
            }

    }
    MPI_Finalize();
    return 0;
}
```

En el ejercicio 2 se pide transmitir un dato introducido por el usuario de un proceso al siguiente, de tal forma que el proceso i recibe de i-1 y transmite el dato a i+1, hasta que el dato alcanza el último nodo.

Se hace uso de las funciones MPI_Init, MPI_Comm_rank (comm, &pid) y MPI_Comm_size (comm, &npr) al igual que en el ejercicio 1.

A continuación se establece la lógica para ir mandando y recibiendo el número a los correspondientes procesos:

Si (rank == 0) (si es el master o proceso 0): le pide que introduzca el número que se va a transmitir. Se ejecuta dentro de este if la función **MPI_Send**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) para el envío de un mensaje contenido en buf de count elementos de tipo datatype al proceso dest del comunicador comm y con etiqueta tag. En este caso, los parámetros serán:

MPI_Send(&num //referencia al dato que se va a enviar
,1 // tamaño del dato a enviar
,MPI_INT // Tipo de dato que se envía
,rank+1 // pid del proceso destino
,0 //tag
,MPI_COMM_WORLD); //Comunicador por el que se manda

Dentro del else (si no es el proceso 0) se ejecuta la función **MPI_Recv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status), que permite la recepción de un mensaje a almacenar en buf de count elementos de tipo datatype, procedente del proceso source del comunicador comm y con etiqueta tag. status es una estructura interna donde se almacena información al finalizar la recepción. En este caso, los parámetros serían los siguientes:

```
MPI_Recv(&num// Referencia a donde se almacenará lo recibido
        ,1 // tamaño del vector a recibir
        ,MPI_INT // Tipo de dato que recibe
        ,rank-1 // pid del proceso origen de la que se recibe
        ,0 // tag
        ,MPI_COMM_WORLD // Comunicador por el que se recibe
        ,&estado); // estructura informativa del estado
```

A continuación, se establece que realice de nuevo la función MPI_Send() si no es el último proceso, ya que este sólo recibe el dato.

Resultados:

La salida por consola sería la siguiente:

```
sara@sara-vivobook:~/Downloads$ mpicc ej2prueba.c -o ejecutable_ej2
sara@sara-vivobook:~/Downloads$ mpirun -np 6 ./ejecutable_ej2
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
Introduce el numero que desea transmitir: 8
Proceso 0 envia 8 al proceso 1
Soy el proceso 1 y he recibido 8
Proceso 1 envia 8 al proceso 2
Soy el proceso 2 y he recibido 8
Proceso 2 envia 8 al proceso 3
Soy el proceso 3 y he recibido 8
Proceso 3 envia 8 al proceso 4
Soy el proceso 4 y he recibido 8
Proceso 4 envia 8 al proceso 5
Soy el proceso 5 y he recibido 8
sara@sara-vivobook:~/Downloads$
```

Como se puede observar, los procesos reciben y envían el dato como descrito previamente.

Ejercicio 3:

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

int main(int argc, char *argv[])
{
  MPI_Status status;
  int num, rank, size, tag, next, from;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);


  next = (rank + 1) % size;
  from = (rank + size - 1) % size;


  if (rank == 0) {
    printf("Introduce el numero de vueltas al anillo: ");
    scanf("%d", &num);

    printf("Proceso %d envia %d al proceso %d\n", rank, num, next);
    MPI_Send(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD);
  }

  do{

    MPI_Recv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
    printf("Proceso %d ha recibido %d\n", rank, num);

    if (rank == 0) {
      --num;
    }

    printf("Proceso %d envia %d al proceso %d\n", rank, num, next);
    MPI_Send(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD);
  } while (num > 1);

  if (rank == 0)
    MPI_Recv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &status);

  MPI_Finalize();
  return 0;
}
```

En el ejercicio 3 se pide modificar la implementación del ejercicio 2 para que el dato introducido por el usuario dé tantas vuelvas en anillo como se indique.

Las modificaciones realizadas han sido: quitar el else y hacer un do while para números mayores a 1 y añadir un if para rank == 0.

Dentro del do se ejecuta la función **MPI_Recv** para almacenar lo que se le envió anteriormente, se imprime que el proceso X ha recibido Y. A continuación se implementa un if rank == 0 que hace que se reste una unidad al número de vueltas al anillo. Al hacer esto, el dato a transmitir también será 1 unidad menos en cada iteración. Se imprime que el proceso X envía Y al proceso en cuestión y se usa el **MPI_Send** para enviar el mensaje.

En el siguiente if rank == 0, se usa **MPI_Recv** para recibir el mensaje.

Resultados:

La salida por consola sería la siguiente:

```
Introduce el numero de vueltas al anillo: 5
Proceso 0 envia 5 al proceso 1
Proceso 1 ha recibido 5
Proceso 1 envia 5 al proceso 2
Proceso 2 ha recibido 5
Proceso 2 envia 5 al proceso 3
Proceso 3 ha recibido 5
Proceso 3 envia 5 al proceso 0
Proceso 0 ha recibido 5
Proceso 0 envia 4 al proceso 1
Proceso 1 ha recibido 4
Proceso 1 envia 4 al proceso 2
Proceso 2 ha recibido 4
Proceso 2 envia 4 al proceso 3
Proceso 3 ha recibido 4
Proceso 3 envia 4 al proceso 0
Proceso 0 ha recibido 4
Proceso 0 envia 3 al proceso 1
Proceso 1 ha recibido 3
Proceso 1 envia 3 al proceso 2
Proceso 2 ha recibido 3
Proceso 2 envia 3 al proceso 3
Proceso 3 ha recibido 3
Proceso 3 envia 3 al proceso 0
Proceso 0 ha recibido 3
Proceso 0 envia 2 al proceso 1
Proceso 1 ha recibido 2
Proceso 1 envia 2 al proceso 2
Proceso 2 ha recibido 2
Proceso 2 envia 2 al proceso 3
Proceso 3 ha recibido 2
Proceso 3 envia 2 al proceso 0
Proceso 0 ha recibido 2
Proceso 0 envia 1 al proceso 1
Proceso 1 ha recibido 1
Proceso 1 envia 1 al proceso 2
Proceso 2 ha recibido 1
Proceso 2 envia 1 al proceso 3
Proceso 3 ha recibido 1
Proceso 3 envia 1 al proceso 0
sara@sara-vivobook:~/Downloads$
```

1. ¿Qué desventaja se aprecia en este tipo de comunicaciones punto a punto a medida que aumentan el número de procesos requeridos? Razonar la respuesta.

A medida que se va aumentando el número de procesos se va ralentizando el funcionamiento del programa al tener que ejecutar más procesos dar vueltas al anillo.

Cuando son pocos procesos no se nota tanto el tiempo de carga y ejecución del programa, pero al usar por ejemplo 50 procesos se nota el tiempo de espera para que carguen los 50 procesos, (pasen del rank 0 al rank 1 y sucesivamente) y salga por pantalla el print.

Adicionalmente, al utilizar MPI_Send y MPI_Recv el proceso de envío esperará hasta que el mensaje completo se haya enviado correctamente y el proceso de recepción se bloqueará mientras espera recibir correctamente el mensaje completo.

2. ¿Cómo podría mejorar el sistema y su implementación? Razonar la respuesta.

Podríamos mejorar el rendimiento utilizando la comunicación sin bloqueo, que incluye envío y recepción de llamadas sin bloqueo que serían **MPI_Isend** y **MPI_Irecv.**

Respectivamente estas funciones inician el envío y la recepción, pero no la completan.

La llamada de envío volverá antes de que el mensaje se copie en el búfer de envío, para que se complete necesitará una llamada completa de envío separada, para verificar que los datos se han copiado fuera del búfer de envío.

La llamada de recepción regresará antes de que se almacene un mensaje en el búfer de recepción, se necesita una llamada MPI_Wait para completar la operación de recepción.

Conclusiones:

En esta práctica se ha podido iniciar la programación en MPI y profundizar acerca del comportamiento de los procesos, algunas de sus funciones tales como MPI_Init(): (inicialización), MPI_Comm_rank (comm, &pid): (conocer el pid),MPI_Comm_size (comm, &npr):(nº total procesos), MPI_Barrier(MPI_Comm comm):(sincronización), MPI_Send():(envío de datos), MPI_Recv():(recepción de datos), MPI_Finalize(void): (finalizar ejecución).

Bibliografía:

http://www.datsi.fi.upm.es/~dopico/MPI/MPI.pdf