Arquitectura de Sistemas Paralelos

# Práctica 1

**Programación MPI**
*Modos de Comunicación*

Michael Alexander Fajardo Malacatus
Gloria del Valle Cano
michael.fajardo@estudiante.uam.es & gloria.valle@estudiante.uam.es

Grado en Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
30-03-2020

# Índice general

# Índice de figuras

# Introducción

En esta primera práctica realizaremos los primeros programas con MPI implementando versiones paralelas de varios algoritmos, utilizando una arquitectura basada en el paradigma maestro/esclavo con el fin de intercambiar datos entre un proceso maestro y varios esclavos. Comprobaremos, además, el funcionamiento de las comunicaciones en MPI, para comprender los diferentes modos de comunicación y la utilización de llamadas bloqueantes y no bloqueantes. Además, estudiaremos el tiempo de intercambio de mensajes sabiendo que éste depende de la velocidad de las máquinas nativas, la transmisión de la red y el tamaño de los mensajes.

# Semana 1: Ejercicios básicos en MPI

## 1.1. Suma de N números enteros

En este ejercicio analizamos el código base proporcionado y solucionamos el problema planteado. En el reparto cambiamos a `tam = ceil(N/(nump*TAM))`, siendo este de 4 como se nos pedía en el enunciado. Además comentamos `suma_mpi.c` para familiarizarnos con MPI y realizar el resto de los ejercicios.

## 1.2. Estimar el valor de Pi con un método Monte Carlo

El método de Monte Carlo nos permite calcular una aproximación al resultado que buscamos por medio de la estadística y la probabilidad. El código base, `monte_carlo.c`, se encarga de construir un cuadrado, encontrar el círculo dentro del mismo y generar puntos al azar dentro del cuadrado. Aplicando Monte Carlo, encontramos la siguiente relación:

$$\frac{A_{circulo}}{A_{cuadrado}} = \frac{N_{puntos}}{N_{total}}$$

Siendo, por aclarar, $N_{puntos}$ el número de puntos dentro del círculo, y $N_{total}$ el número de puntos totales. Es decir, el cociente nos da un estimado de $\frac{\pi}{4}$, por lo que solo hay que multiplicar por 4 para conseguir $\pi$. Esto también nos dice que es relevante que los puntos estén uniformemente distribuidos y que teniendo un número finito de puntos de donde escoger, dependiendo del ordenador, la aproximación de $\pi$ será mejor a medida que aumenta el número de puntos. Nosotros hemos conseguido buenos resultados con $10^5$ puntos.
En la versión de MPI que hemos realizado, `monte_carlo_parallel.c`, por cada esclavo se generan los puntos aleatorios y cada uno genera un conteo ($k$), que servirá para sumarlo con `MPI_SUM` en la función `MPI_Reduce`. Finalmente, el master se encarga de estimar $\pi$.

## 1.3. Multiplicación de Matrices (NxN)

Se ha realizado la multiplicación de matrices $NxN$ (siendo $N$ el tamaño de la matriz que introduce el usuario), en el fichero `multiplicacion_nxn.c`. La comunicación que se ha elegido es la de punto-punto.

El algoritmo realizado para el cálculo del producto de matrices mediante la utilización de MPI es el siguiente:

1. **Maestro**
   (a) Obtener el **reparto** del número de filas con las que debe trabajar cada procesador; diviendo el tamaño de la matriz entre el número de procesadores.
   (b) Mediante un **bucle** for se hace el **reparto** del número de **filas** y la **matriz** con la que se va multiplicar la primera, además de su tamaño. Estos tres parámetros se envían mediante tres funciones `MPI_Send`.

(c) En caso de haber repartido todas las filas de la primera matriz a todos los esclavos, se **recogerá** los resultados enviados por los esclavos, según el orden de envío; mediante nuevamente un bucle *for* y con la instrucción `MPI_Recv`. En caso contrario se realizará el cálculo de las filas restantes; una vez terminado, se procede a la recogida como se ha mencionado anteriormente.

(d) Los resultados parciales recibidos, se guardarán en una matriz que contendrá el resultado de la multiplicación de las dos matrices.

(e) Se imprimirá la matriz resultante de la multiplicación.

2. **Esclavo**

(a) Recoger el **tamaño** de la matriz, para obtener el reparto que corresponde a cada esclavo.

(b) **Recibir** tanto las **filas** con las que trabajar como la **matriz** con la que se multiplicará cada fila. Todos estos parámatros se recibirán con la función `MPI_Recv`.

(c) Realizar el cáculo de la multiplicación mediante producto escalar.

(d) Guardar en matriz los resultados obtenidos.

(e) Enviar al maestro matriz a través de función `MPI_Send`.

Cabe mencionar que la matriz con que se menciona no es más que un array con un indexación.

# Semana 2: Comunicaciones punto a punto y comunicaciones colectivas

## 2.1. Tiempos de comunicación de los diferentes modos

1. *Estudie el código del programa identificando donde se toman las medidas de tiempos. Compílelo y ejecútelo en una máquina Multiprocesador con 2 equipos conectados por red.*

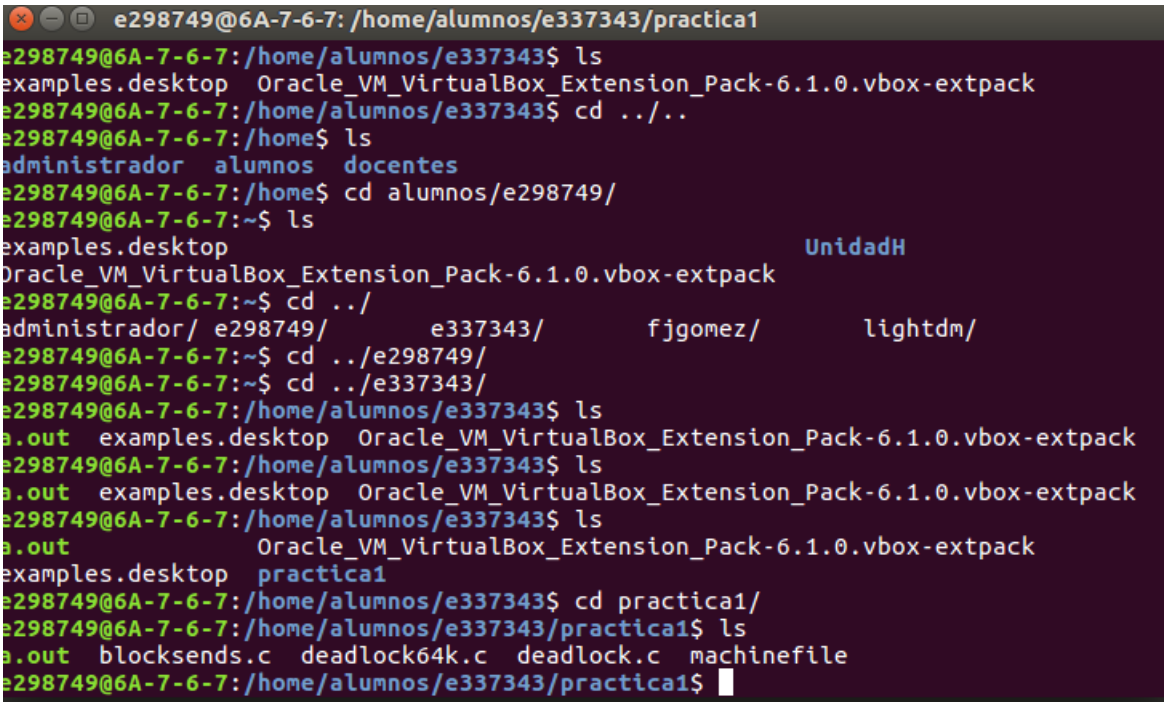


Figura 1: Conexión en el laboratorio mediante dos máquinas

```
e337343@6A-8-6-8: ~/practica1
e337343@6A-8-6-8:~/practica1$ mpirun -machinefile machinefile -np 8 ./a.out 4
Message size =     4 floats
Task 2 initialized
Message size =     4 floats
Task 1 initialized
Message size =     4 floats
Task 3 initialized
Message size =     4 floats
Task 0 initialized
Ready to send messages
Elapsed time for synchronous send =       35 uSec
Elapsed time for ready send =        9 uSec
Elapsed time for buffer allocation =       10 uSec
Elapsed time for buffered send =        5 uSec
Elapsed time for standard send =        2 uSec
Message size =     4 floats
Task 5 initialized
Message size =     4 floats
Task 6 initialized
Message size =     4 floats
Task 4 initialized
Message size =     4 floats
Task 7 initialized
e337343@6A-8-6-8:~/practica1$
```

Figura 2: Ejecución de blocksends

2. *Represente en un diagrama los tiempos involucrados en la comunicación para los distintos modos.*

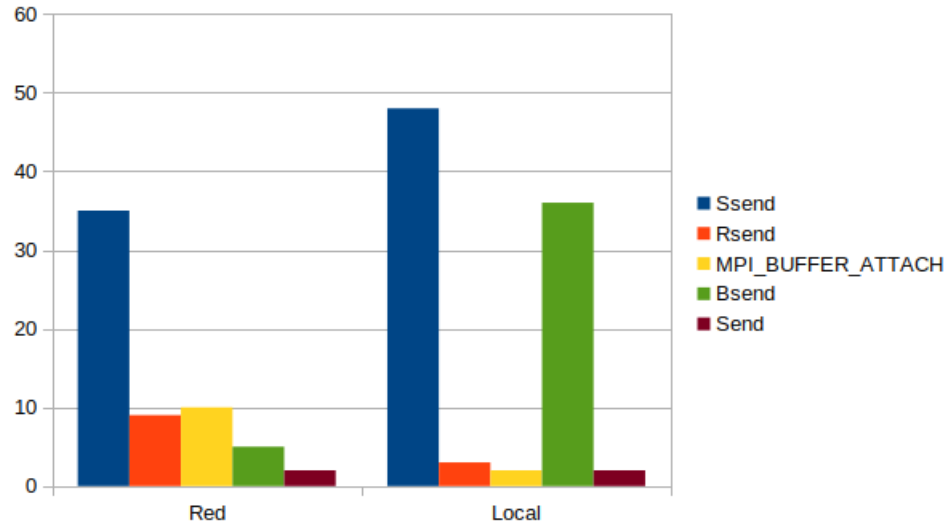


Figura 3: Diagrama de comparativa de tiempos

3. *Compare el tiempo empleado por el blocking send en los cuatro modos para distintos tamaños de mensaje.*

Para un tamaño de 4:

```
[asp06@labomat36 p1]$ mpirun -np 4 ./blocksends 4
 Message size =     4 floats
 Task 2 initialized
 Message size =     4 floats
 Task 3 initialized
 Message size =     4 floats
 Task 0 initialized
 Message size =     4 floats
 Task 1 initialized
 Ready to send messages
 Elapsed time for synchronous send =      314 uSec
 Elapsed time for ready send =       43 uSec
 Elapsed time for buffer allocation =       22 uSec
 Elapsed time for buffered send =       69 uSec
 Elapsed time for standard send =       26 uSec
```

Para un tamaño de 32:

```
 [asp06@labomat36 p1]$ mpirun -np 4 ./blocksends 32
 Message size =    32 floats
 Task 0 initialized
 Message size =    32 floats
 Task 1 initialized
 Message size =    32 floats
 Task 2 initialized
 Message size =    32 floats
 Task 3 initialized
 Ready to send messages
 Elapsed time for synchronous send =      283 uSec
 Elapsed time for ready send =       36 uSec
 Elapsed time for buffer allocation =       16 uSec
 Elapsed time for buffered send =       30 uSec
 Elapsed time for standard send =       24 uSec
```

Para un tamaño de 329978:

```
[asp06@labomat36 p1]$ mpirun -np 4 ./blocksends 329978
 Message size = 329978 floats
 Task 2 initialized
 Message size = 329978 floats
 Task 3 initialized
 Message size = 329978 floats
 Task 0 initialized
```

```
Message size = 329978 floats
Task 1 initialized
Ready to send messages
Elapsed time for synchronous send =     1973 uSec
Elapsed time for ready send =     1745 uSec
Elapsed time for buffer allocation =       31 uSec
Elapsed time for buffered send =     1698 uSec
Elapsed time for standard send =     2130 uSec
```

En general podemos ver que el modo síncrono es el que más tarda y el modo listo el que menos en cualquier caso. La razón de esto es que el modo síncrono espera a Recv y empieza tanto si la máquina está activa o no, mientras que el listo indica que el buffer de envío se puede reutilizar si la máquina está activa. El modo buffered funciona a nivel local, mientras que el standard no.

4. *Compare los resultados anteriores con la ejecución en un sólo equipo. Explique los resultados.*
   En este caso no se mandan mensajes por red, sino que simplemente se comunican entre procesos por lo cual el medio que utilizan para los mensajes no es la red sino la memoria.

## 2.2. Bloqueo en las comunicaciones

1. *Compile los programas deadlock y deadlock64, ejecútelos con dos procesos. Uno de los programas escribirá unas líneas y luego se detendrá. Aborte el programa. ¿Explique por qué el programa entra en interbloqueo? ¿De qué parámetro depende? Estime el valor del parámetro para su sistema que delimita ambos comportamientos.*
   El programa entra en interbloqueo porque no se estaba haciendo ningún tipo de diferenciación entre el maestro y el esclavo. Con lo que ambos maestro y esclavo entraban a esa sección de código, con lo cual se entraba en una condición de carrera, lo que a su vez implica que no de dependía del orden de las funciones `MPI_Send` y `MPI_Recv`.

2. *Corrija la estructura del programa para evitar el interbloqueo sin cambiar ninguna función de comunicaciones.*
   Esto se logra diferenciando el código del maestro con el del esclavo.

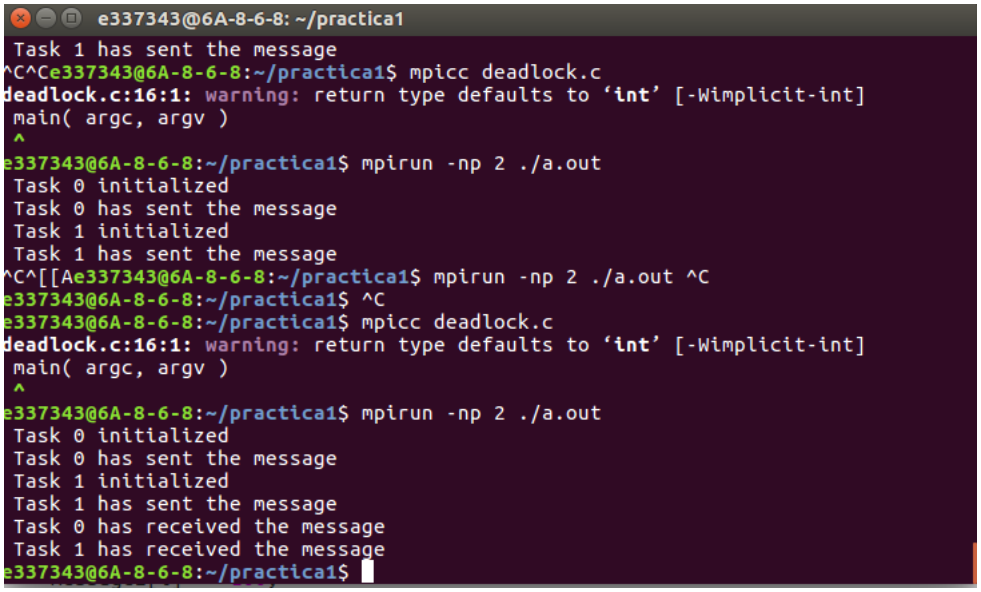


Figura 4: Ejecución de deadlock

3. *Corrija el programa sin cambiar la estructura, utilizando funciones de comunicaciones que garanticen el funcionamiento correcto*
   Con las funciones que vienen dadas en el código y diferenciando cada parte del código además de colocando adecuadamente los tags de cada mensaje. El programa funciona correctamente.

4. *Cambie el programa que corresponda y sin cambiar la estructura realice las modificaciones oportunas para que ambos programas tengan un comportamiento bloqueante.*
   Para que send y recv se ha de poner un Ssend en lugar de un Send, ya que Ssend es síncrono esperará hasta que el esclavo reciba el mensaje.

5. *Cambie el programa y utilice la función* `MPI_Send` *recv para proponer una estructura de comunicaciones que evite los bloqueos.*
   Para esto valdría con las funciones `MPI_ISend` e `MPI_IRecv` ya que ninguna de ellas es bloqueante. Además habría que añadir `MPI_wait`.

## 2.3. Medida de la latencia de la comunicación

Para este ejercicio se ha medido el tiempo con `medida_latencia.c` y se ha desarrollado un script `graficas.sh` con el fin de representar el tiempo transcurrido con cada tamaño de mensaje.
Para este ejercicio, se han seguido las instrucciones del enunciado y se ha medido el tiempo de latencia total en diferentes tamaños de mensaje. Representamos los datos en la siguiente gráfica.

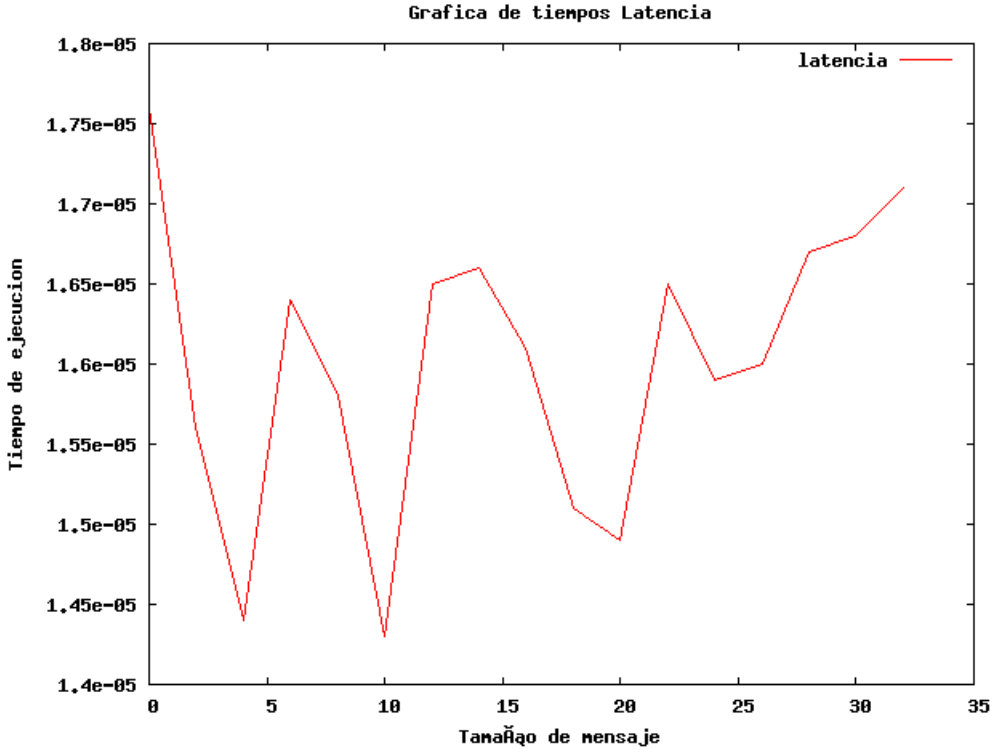


Figura 5: Representación del tiempo de ejecución frente al tamaño de mensaje

Lo que es destacable es que en tiempo de ejecución es mucho mayor cuanto mayor es el tamaño del mensaje.