UNIVERSIDAD DE BURGOS

Grado en Ingeniería Informática

Arquitecturas Paralelas

Curso 2024-2025

Quinta Práctica Entregable

PROCESOS DE ENTRADA/SALIDA

AUTORES

Diego Urbaneja

Hugo Gómez

Nicolás Villanueva Ortega

ÍNDICE

Introducción

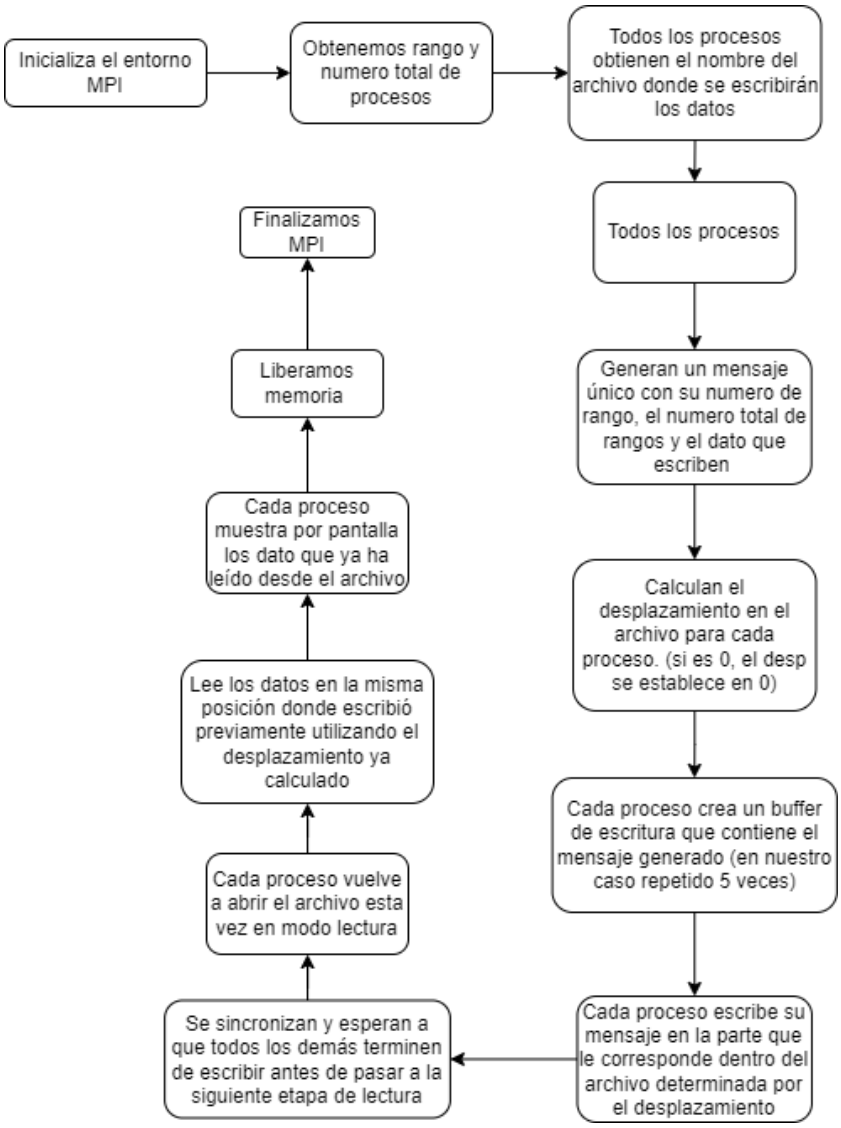
Durante esta práctica estudiaremos la implementación de procesos de entrada/salida en paralelo utilizando MPI. El objetivo es introducirse en las técnicas de entrada/salida en paralelo, contrastando con la tradicional entrada/salida en serie.

Entrada/Salida serie: En este modo, solo un proceso (generalmente el proceso 0) realiza todas las operaciones de entrada y salida, generando un cuello de botella en la comunicación.

Entrada/Salida en paralelo: Todos los procesos pueden acceder a un archivo común, repartiendo las operaciones de lectura y escritura, y cada proceso tiene su propia "vista" del archivo, evitando la interferencia en los datos de otros procesos.

Recomendada la visualización del documento con un 140% - 150% de zoom.

Flujograma



Siguiendo el flujograma, al realizar el programa inicializamos el entorno MPI, seguido, obtenemos el rango y el número total de procesos como anteriores prácticas. Dado a que vamos a escribir los datos en un fichero '.txt', tenemos que especificar la ruta y el nombre del archivo a cada proceso (en este caso ya no tenemos proceso maestro y esclavos al igual que en la anterior practica).

Todos los procesos generan mensajes únicos con su número identificador de rango, el número de rangos totales y el dato que escribe cada uno. Antes de almacenar ese mensaje escrito en el buffer, calculamos el desplazamiento en el archivo para cada proceso (si el rango es 0 el desplazamiento se establece en 0).

Después de calcular el desplazamiento cada proceso crea un buffer de escritura que contiene el mensaje generado un número de veces preestablecidas, en nuestro caso 5 veces. Después cada proceso escribe su mensaje proveniente del buffer en la parte que le corresponde dentro del archivo determinado por el desplazamiento, estos se sincronizan y esperan a que todos los procesos terminen de escribir antes de pasar a la siguiente etapa.

Una vez escrito en el fichero todos los datos de todos los procesos, cada proceso vuelve a abrir el archivo, en este caso en modo lectura. Cada proceso lee los datos en la misma posición donde escribió previamente utilizando el desplazamiento ya calculado y muestra por pantalla los datos que ya ha leído desde el archivo. Para finalizar, liberamos memoria y paramos la ejecución de MPI.

Código

Para explicar el código de manera más ordenada vamos a dividir el código en diferentes bloques:

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>     // Para usar strrchr
#include <windows.h>    // Para GetModuleFileName

// Función para obtener el directorio de un path
char* my_dirname(char* path) {
    char* last_slash = strrchr(path, '\\');
    if (last_slash) {
        *last_slash = '\0'; // Termina la cadena en el último '\\'
    }
    return path;
}
```

En esta primera parte se pueden observar las diferentes librerías incluidas en nuestro código, así como una función para obtener el directorio del ejecutable. Esto se debe a que al principio cuando se ejecutaba el programa, no sabíamos donde se estaba generado el archivo entonces para asegurarnos de que se generaba en el mismo directorio que el ejecutable para ello obtenemos la ruta del ejecutable.

Dentro del main():

```c
// Función main
int main(int argc, char* argv[])
{
    // Variables para indentificar los procesos
    int mirango, size;
    int longitud;
    char nombre[32];

    // N?mero de veces que cada proceso escribe su mensaje
    int N = 5;
```

Declaración de las variables como el número de procesos, definir un char...

```c
// Inicio del entorno MPI y obtenci?n de informaci?n de procesos
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &mirango);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Get_processor_name(nombre, &longitud);
```

Inicialización de las variables del entorno MPI

```c
// Nombre del fichero
char filename[] = "practica_05.txt";

// Buffer para almacenar la ruta del ejecutable
char exePath[1024];
// Obtener la ruta completa del ejecutable
GetModuleFileName(NULL, exePath, sizeof(exePath));

// Obtener el directorio donde se encuentra el ejecutable
char* exeDir = my_dirname(exePath);  // Extraer el directorio

// Crear una variable que contenga la ruta completa (directorio + nombre de archivo)
char fullpath[1024];
// Concatenar el path con el nombre del archivo
snprintf(fullpath, sizeof(fullpath), "%s\\%s", exeDir, filename);
```

Variables referentes a la configuración del archivo como el nombre, el directorio del ejecutable,

Y crear la ruta donde se va a aguardar el archivo para tenerlo localizado.

```c
// Generar el mensaje
char message[100];
int n = mirango;
int m = n + 1;
snprintf(message, sizeof(message), "Soy el proceso %d de %d y escribo el dato %d\n", n, size, m);
int msglen = strlen(message);

// Calcular el tama?o de los datos a escribir
long long data_size = N * msglen;
```

Esta parte del código hace referencia el mensaje a escribir, definiendo la variable del mensaje, el número del proceso, el texto a escribir, y una vez que tenemos el mensaje calculamos el tamaño de este.

```c
// Calcular el desplazamiento (offset) usando MPI_Exscan
MPI_Offset offset = 0;
MPI_Exscan(&data_size, &offset, 1, MPI_LONG_LONG_INT, MPI_SUM, MPI_COMM_WORLD);
if (mirango == 0)
    offset = 0;
```

Calculamos el desplazamiento para asegurar que cada proceso escriba sus datos en una posición única del archivo, evitando sobre escrituras y asegurando la coherencia de los datos.

```
// Preparar el buffer para escribir
char* write_buf = (char*)malloc(data_size);
for (int i = 0; i < N; i++)
{
    memcpy(write_buf + i * msglen, message, msglen);
}
```

Preparamos el buffer de escritura llenándolo con copias del mensaje para poder realizar la escritura concurrente.

```
// Abrir el fichero para escritura
MPI_File_open(MPI_COMM_WORLD, fullpath, MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);

// Escribir los datos en el fichero en la posici?n correspondiente
MPI_File_write_at(fh, offset, write_buf, data_size, MPI_CHAR, &status);

// Cerrar el fichero despu?s de escribir
MPI_File_close(&fh);
```

Realizamos la escritura concurrente en el archivo para ello abrimos el archivo, escribimos el buffer en el archivo en la posición especificada por el offset, y finalmente cerramos el archivo.

Cada proceso abre el archivo común y escribe su bloque de datos (write_buf) en una posición única (offset), garantizando que no haya solapamiento entre las escrituras de diferentes procesos.

```
// Sincronizar antes de leer
MPI_Barrier(MPI_COMM_WORLD);

// Abrir el fichero para lectura
MPI_File_open(MPI_COMM_WORLD, fullpath, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

// Preparar el buffer para leer
char* read_buf = (char*)malloc(data_size);

// Leer los datos desde el fichero en la posici?n correspondiente
MPI_File_read_at(fh, offset, read_buf, data_size, MPI_CHAR, &status);

// Cerrar el fichero despu?s de leer
MPI_File_close(&fh);
```

Sincronizamos los procesos para poder leer el contenido. Abrimos el fichero en modo lectura, almacenamos los datos leídos en el archivo en un buffer para leerlos posteriormente y cerramos el archivo.

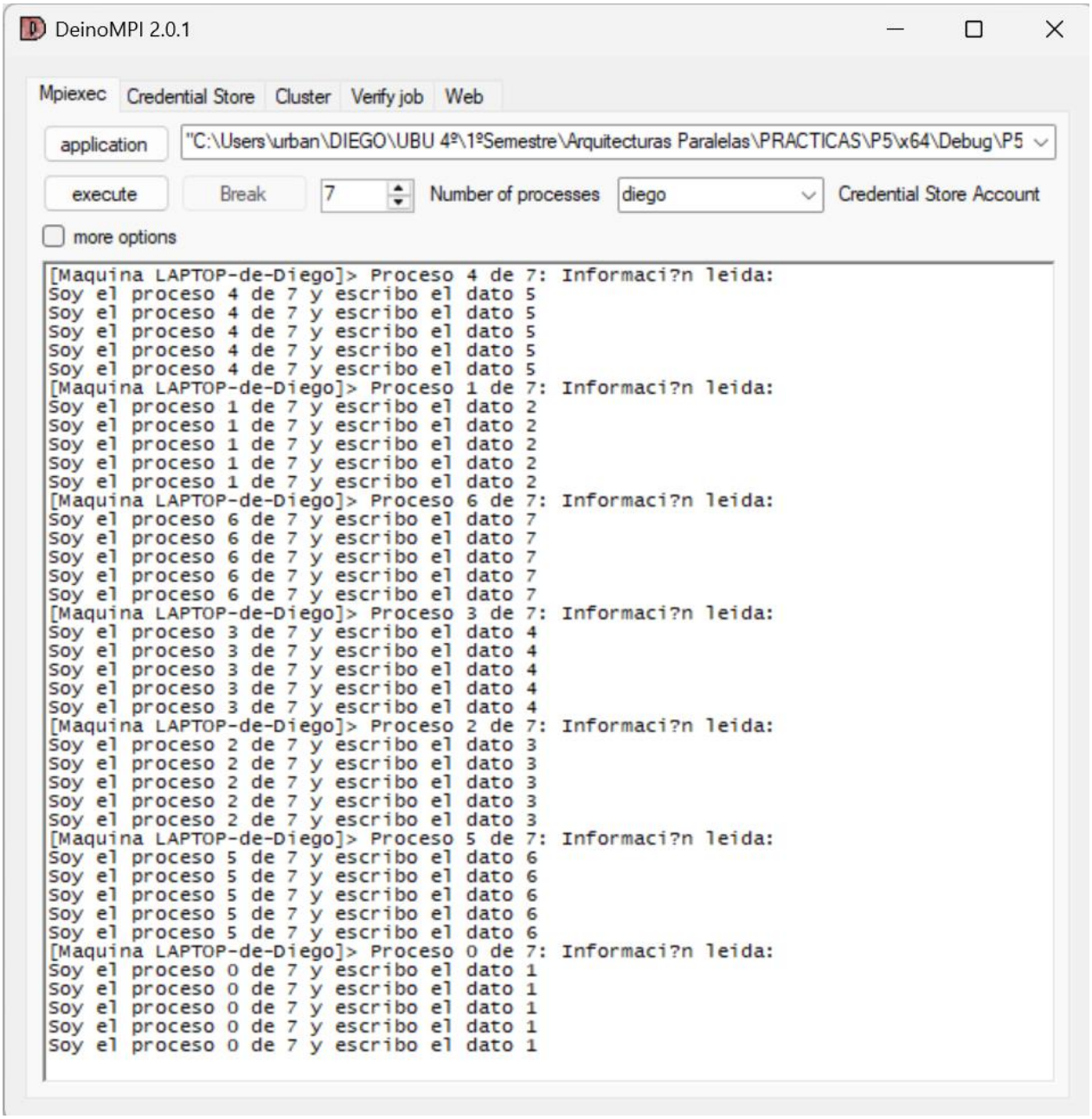
```
// Mostrar los datos le?dos por cada proceso
printf("[Maquina %s]> Proceso %d de %d: Informaci?n leida:\n%.*s", nombre, mirango, size, (int)data_size, read_buf);
fflush(stdout);

// Liberar memoria
free(write_buf);
free(read_buf);

// Finalizar entorno MPI
MPI_Finalize();
return 0;
```

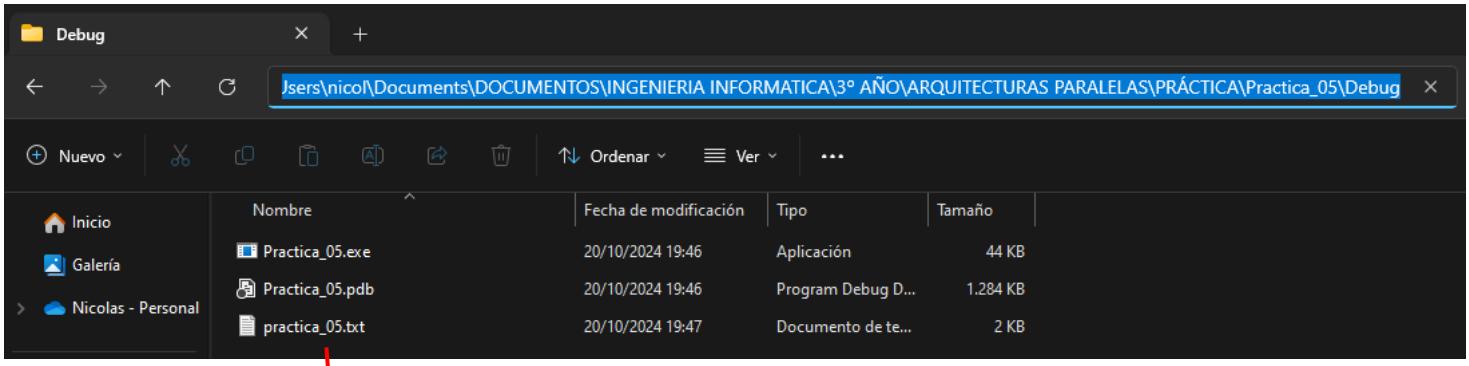
Finalmente mostramos los datos leídos y liberamos memoria y finalizamos el entorno MPI.

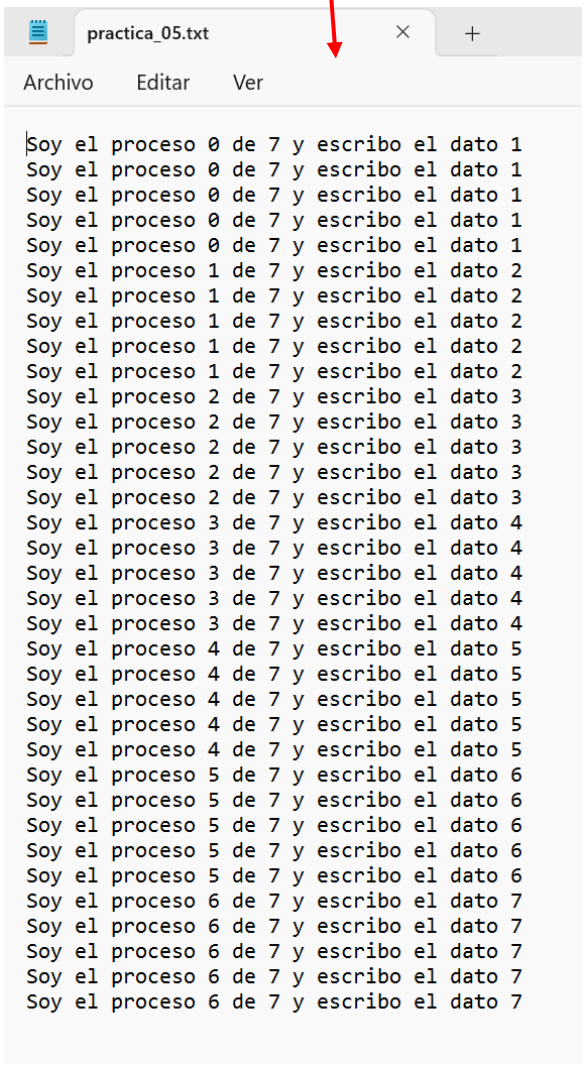
Salida por Pantalla



Como se puede apreciar cada uno de los procesos ha escrito en el archivo .txt de salida su mensaje correspondiente 5 veces (constante indicada en el código). Y esto lo sabemos porque la salida por pantalla es la lectura de dicho archivo por cada uno de los procesos que han generado la información.

Archivo Generado





Como se puede apreciar el archivo se crea en la carpeta donde se encuentra el ejecutable del programa y contiene la información generada por cada proceso correctamente.

Como ya se ha explicado anteriormente no hace falta definir una ruta distinta por cada equipo en el que se ejecute el programa, sino que este obtendrá la ruta del ejecutable automáticamente y generará el archivo de salida .txt en el mismo directorio.

De esta manera podemos tenerlo localizado y fácilmente accesible en todo momento.

Cuestiones

¿Se puede pensar en la entrada salida paralela como forma de que un proceso reparta datos a otros alternativamente a las funciones de reparto conocidas?

Sí, la entrada/salida paralela distribuye las operaciones de acceso a disco entre los procesos, similar a cómo las funciones de reparto distribuyen datos en memoria. Sin embargo, **su propósito no es estrictamente el mismo**, ya que las **funciones de reparto se usan para distribuir datos desde un proceso maestro**, mientras que **en la entrada y salida** en paralelo mejora el rendimiento al hacer que **todos los procesos puedan acceder directamente al archivo.**

¿Qué inconvenientes plantea esto?

Puede ser más **compleja de implementar**, requiere **coordinación precisa entre procesos** para evitar colisiones en el acceso a los archivos, y **puede no mejorar el rendimiento en sistemas de archivos no optimizados** debido al tiempo de espera para la sincronización de los procesos.

¿Puede aportar alguna ventaja?

Reduce el cuello de botella en la entrada/salida al permitir que varios procesos accedan al archivo de forma independiente.

Mejora la escalabilidad al distribuir las operaciones entre procesos y **aprovecha mejores arquitecturas de almacenamiento distribuidas** para incrementar el rendimiento.

Bibliografía

Hemos obtenido la información para la realización de la práctica del guion proporcionado por el profesor.

Para la resolución de dudas con el código hemos empleado chat GPT o1-mini.