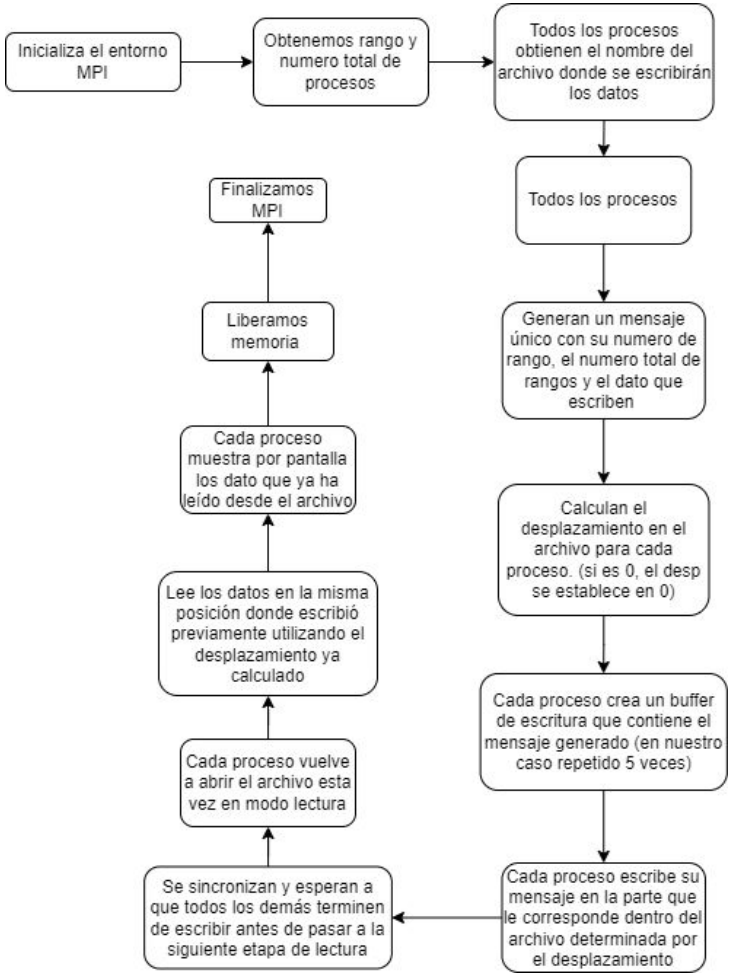
Práctica 5: Procesos de Entrada/Salida

DIEGO URBANEJA
HUGO GÓMEZ
NICOLÁS VILLANUEVA

ÍNDICE

FLUJOGRAMA


Inicializa el entorno MPI
Obtenemos rango y numero total de procesos
Todos los procesos obtienen el nombre del archivo donde se escribirán los datos
Todos los procesos
Generan un mensaje único con su numero de rango, el numero total de rangos y el dato que escriben
Calculan el desplazamiento en el archivo para cada proceso. (si es 0, el desp se establece en 0)
Cada proceso crea un buffer de escritura que contiene el mensaje generado (en nuestro caso repetido 5 veces)
Cada proceso escribe su mensaje en la parte que le corresponde dentro del archivo determinada por el desplazamiento
Se sincronizan y esperan a que todos los demás terminen de escribir antes de pasar a la siguiente etapa de lectura
Cada proceso vuelve a abrir el archivo esta vez en modo lectura
Lee los datos en la misma posición donde escribió previamente utilizando el desplazamiento ya calculado
Cada proceso muestra por pantalla los dato que ya ha leído desde el archivo
Liberamos memoria
Finalizamos MPI

CÓDIGO DEL PROGRAMA

```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;
}

// 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;

    // 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);
```

```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);

// Variable MPI que guardar? una referencia al archivo utilizado
MPI_File fh;
MPI_Status status;

// 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);
```

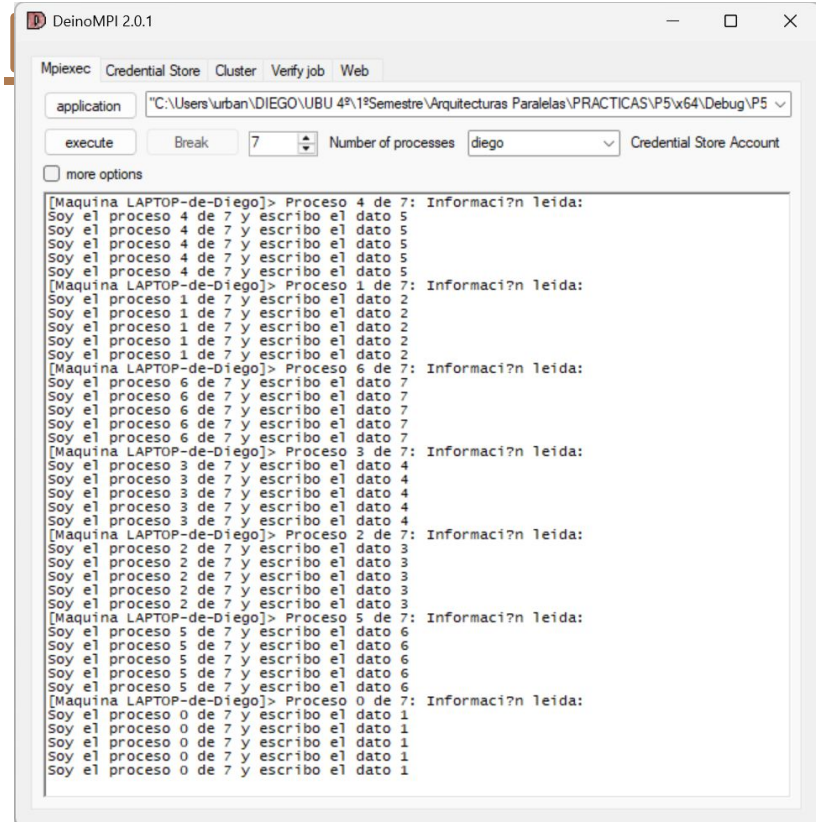
CÓDIGO DEL PROGRAMA

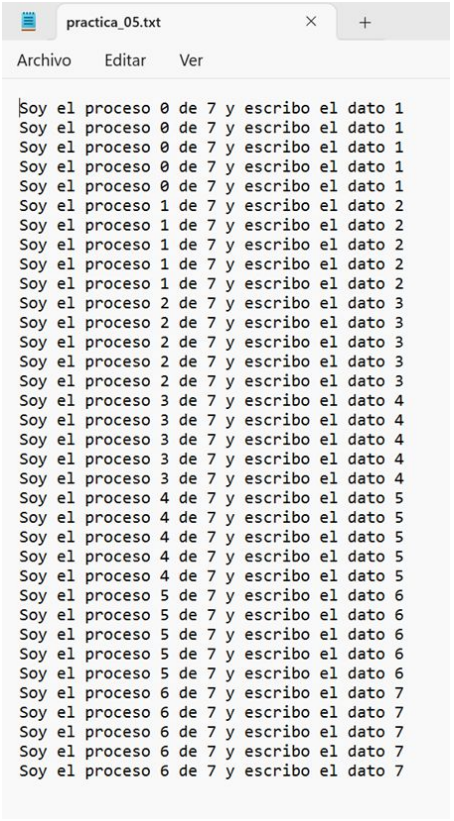
```
// Calcular el tama?o de los datos a escribir
long long data_size = N * msglen;

// 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;

// 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);
}

// 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);
```

CÓDIGO DEL PROGRAMA

```
// 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);

// 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;
```

EJECUCIÓN Y SALIDA POR





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.