PRÁCTICA 5: PROCESOS DE ENTRADA/SALIDA

Arquitecturas paralelas, Ingeniería informática, UBU.

Sergio Buil Laliena, Sandro Martín Rubio, Rocío Esteban Valverde

5 noviembre 2023

Contenido

Objetivos

Practicar los métodos de entrada/salida en paralelo

Introducción

El código proporcionado es un ejemplo de programación paralela utilizando la biblioteca MPI (Message Passing Interface). MPI es ampliamente utilizado en la programación paralela para crear aplicaciones que se ejecutan en sistemas distribuidos o en múltiples procesadores en un clúster. El objetivo de este código es escribir datos en un archivo y luego leerlos desde múltiples procesos que se ejecutan de manera paralela.

Código

```
/**
* Nombre:PracticaARPA5
* Descripción: Los procesos lanzados escriben en un fichero su rango un cierto número de veces
               y posteriormente cada proceso leerá del fichero los datos que introdujo y los mostrará.
almacenarlo para poder visualizarlo como ASCII. También se puede guardar el rango como carácter.
* Autores: Sergio Buil Laliena, Sandro Martín Rubio, Rocío Esteban Valverde
* Fecha: 5-Nov-2023
*
*/

#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define N 20 // Número de veces que cada proceso escribe en el archivo

int main(int argc, char* argv[]){
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_File file;
    MPI_Status estado;

    char filename[] = "output.txt";
    char datarep[] = "native";
    int escribir[N], leer[N], i;

    // Abrir el archivo en modo de escritura
    MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &file);
    MPI_File_set_view(file, N * rank * sizeof(int), MPI_INT, MPI_INT, datarep, MPI_INFO_NULL);

    // Escribir en el archivo el rango del proceso N veces
    for (i = 0; i < N; i++) {
        escribir[i] = 48 + rank;// Convertir el rango a ASCII
    }
    MPI_File_write_at(file, 0, escribir, N, MPI_INT, &estado);

    // Cerrar el archivo
    MPI_File_close(&file);

    // Abrir el archivo en modo de lectura
    MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &file);
    // Leer del archivo
```

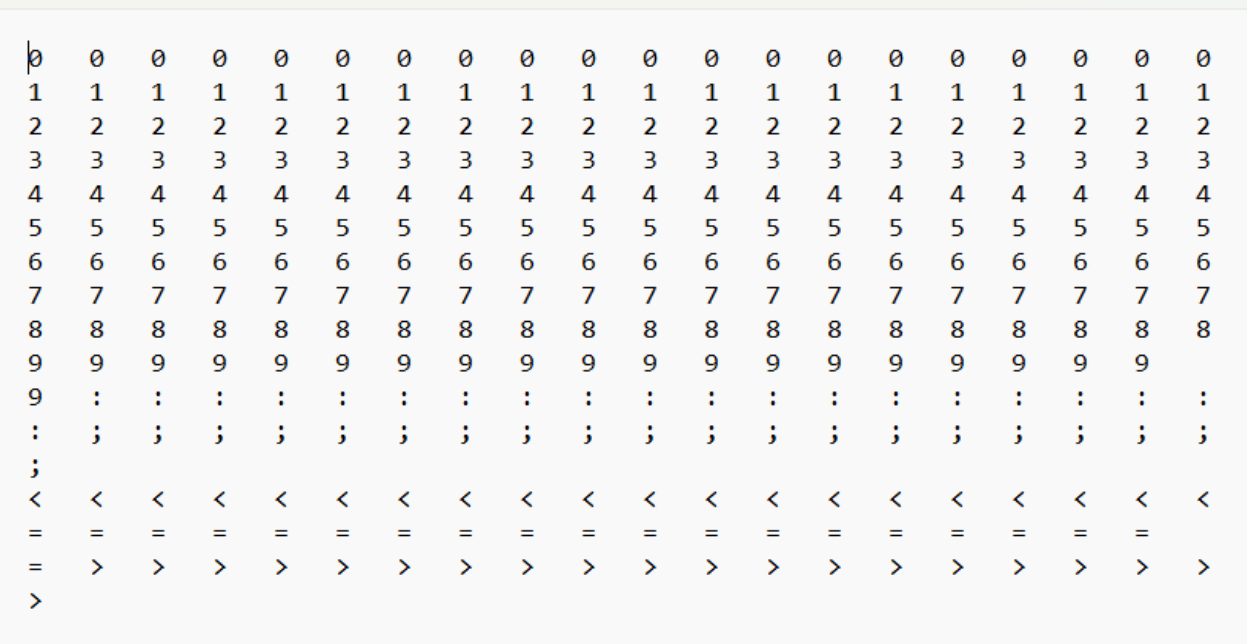
```
    MPI_File_set_view(file, N * rank * sizeof(int), MPI_INT, MPI_INT, datarep,
MPI_INFO_NULL);
    MPI_File_read_at(file, 0, leer, N, MPI_INT, &estado);

    // Cerrar el archivo
    MPI_File_close(&file);

    // Mostrar los datos
    for (i = 0; i < N; i++) {
        leer[i] = leer[i] - 48;
    }
    printf("Proceso %d lee:\n ", rank);

    for (i = 0; i < N; i++) {
        printf("%d ", leer[i]);
    }
    printf("\n");

    MPI_Finalize();
    return 0;
}
```

Resultados

La salida por pantalla al ejecutar la aplicación en DeinoMPI:

```
Proceso 4
 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
Proceso 3
 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
Proceso 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Proceso 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Proceso 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
Proceso 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
Proceso 10
 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
Proceso 9
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
Proceso 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Proceso 11
 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
Proceso 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
Proceso 6
 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
0: DESKTOP-VN9KBTC: 0
1: DESKTOP-VN9KBTC: 0
2: DESKTOP-VN9KBTC: 0
3: DESKTOP-VN9KBTC: 0
4: DESKTOP-VN9KBTC: 0
5: DESKTOP-VN9KBTC: 0
6: DESKTOP-VN9KBTC: 0
7: DESKTOP-VN9KBTC: 0
8: DESKTOP-VN9KBTC: 0
9: DESKTOP-VN9KBTC: 0
10: DESKTOP-VN9KBTC: 0
11: DESKTOP-VN9KBTC: 0
```

El fichero con los números escritos en orden:



La cantidad de números que se escriben son el número de procesos que hay lanzados. Si se lanzan menos procesos habrá menos números en el fichero. Además, van en orden del rango.

Como se convierten los caracteres a ASCII, a partir del proceso 10, en vez de escribir el número se escribe el siguiente carácter de la tabla ASCII.

Descripción del Código

1. Inicialización de MPI y variables

Se inicia MPI con `MPI_Init`, lo que permite la creación y gestión de comunicadores entre los procesos. Además, se crean variables para la escritura, la lectura, para el rango de los procesos y una constante para el número de datos con valor de 20, estas nos servirán para imprimir los resultados.

2. Apertura del Archivo en Modo Escritura

El código abre un archivo llamado "output.txt" con permiso de escritura (`MPI_MODE_CREATE | MPI_MODE_RDWR`) utilizando `MPI_File_open`. Esto asegura que el archivo exista o se cree si no existe. Se utiliza `MPI_INFO_NULL` para los atributos del archivo.

3. Escritura en el Archivo

El programa entra en un bucle para escribir datos en el archivo. Cada proceso escribe su rango en el archivo `N` veces. Los datos se convierten en caracteres ASCII y se escriben en el archivo utilizando `MPI_File_write_at`.

4. Cierre del Archivo

Una vez que se han escrito los datos, el archivo se cierra utilizando `MPI_File_close`.

5. Sincronización de Procesos

Se utiliza `MPI_Barrier` en el comunicador global `MPI_COMM_WORLD` para asegurarse de que todos los procesos hayan completado la escritura antes de continuar. Esto evita que los procesos de lectura intenten leer el archivo antes de que todos los datos estén disponibles.

6. Apertura del Archivo en Modo Lectura

Luego, el código vuelve a abrir el archivo "output.txt", pero esta vez en modo lectura (`MPI_MODE_RDONLY`) y configura una vista con `MPI_File_set_view` para la lectura de datos.

7. Lectura de Datos

El código entra en un bucle para leer los datos del archivo y mostrarlos. Cada proceso lee un dato de dos caracteres (el número y un carácter de salto de línea) utilizando `MPI_File_read_at` y luego imprime el dato leído en la pantalla.

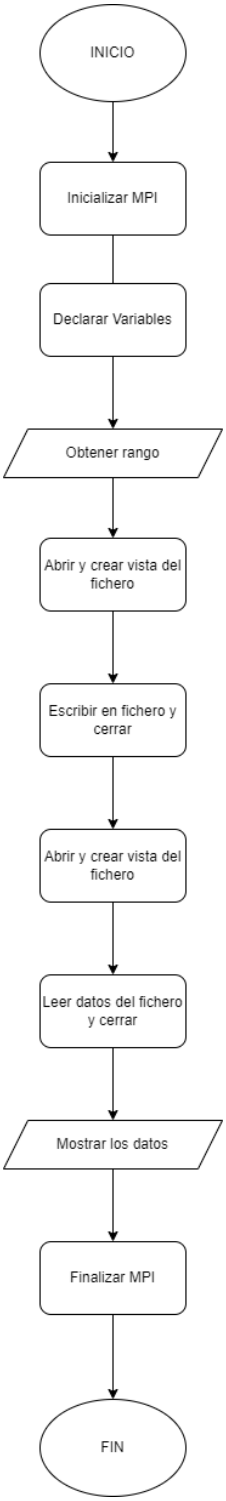
8. Cierre del Archivo

Finalmente, el archivo se cierra nuevamente con `MPI_File_close`.

9. Finalización de MPI

El programa finaliza el entorno MPI con `MPI_Finalize`.

Diagrama de flujo



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 porque se usa una estructura esclavo-maestro entre procesos.

Esto puede ser beneficioso en situaciones en las que varios procesos necesitan acceder o compartir datos de manera simultánea o coordinada ya que el proceso maestro se responsabiliza de la coordinación y control general del programa además de la sincronización entre los procesos esclavo que son los que se encargan de realizar las operaciones de E/S y de comunicar los resultados al maestro.

• ¿Qué inconvenientes plantea esto?

La escalabilidad es limitada ya que un gran número de procesos de E/S es complejo de coordinar. La sincronización excesiva puede ralentizar el acceso a los datos compartidos. Complejidad en la programación por el manejo de la sincronización de procesos que acceden a un mismo archivo.

• ¿Puede aportar alguna ventaja?

Sí, el rendimiento al acceder y manipular los datos de un fichero en paralelo, la flexibilidad en el acceso a datos compartidos, la efectividad en el tiempo de ejecución y menor impacto en el rendimiento de una aplicación al evitar bloqueos o esperas en operaciones de E/S.

Conclusión

El código es un ejemplo de programación paralela utilizando MPI para realizar operaciones de escritura y lectura en un archivo desde múltiples procesos. Esto permite la ejecución eficiente de tareas de lectura/escritura en un entorno paralelo. El uso de funciones de MPI para abrir, escribir y leer archivos asegura que los procesos se sincronicen adecuadamente y evita problemas de concurrencia en la manipulación de archivos.