

PRÁCTICA 7: TIPOS DE DATOS DERIVADOS

Arquitecturas paralelas, Ingeniería informática, UBU.

Sergio Buil Laliena, Sandro Martín Rubio, Rocío Esteban Valverde

27 noviembre 2023

Índice

Objetivos.....	2
Introducción	2
Código.....	2
Resultados.....	5
Descripción del Código.....	5
Diagrama de flujo	7
Cuestiones.....	8
Conclusión.....	8

Objetivos

Mejorar la capacidad de intercambio de datos con el empleo de tipos derivados.

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 crear una matriz con números aleatorios y enviar la matriz triangular superior e inferior de esta a dos procesos que al recibirla la imprimirán por pantalla.

Código

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 4 // Tamaño de la matriz

// Función para inicializar la matriz con números enteros aleatorios
void inicializar_matriz(int matriz[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            matriz[i][j] = rand() % 10; // Números aleatorios entre 0 y 99
        }
    }
}

// Función para imprimir la matriz
void imprimir_matriz(int matriz[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }
}

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

    srand(time(NULL) + mirango); // Semilla para la generación de números aleatorios

    int matriz[N][N];

    // Crear los tipos de datos derivados para las matrices triangulares
```

```

MPI_Datatype tipo_triang_sup, tipo_triang_inf;

// Tipo para la matriz triangular superior
int longitudes[N];
MPI_Aint desplazamientos[N];
for (int i = 0; i < N; i++) {
    longitudes[i] = N - i;
    desplazamientos[i] = (i * N * sizeof(int) + i * sizeof(int));
}
MPI_Type_create_hindexed(N, longitudes, desplazamientos, MPI_INT, &tipo_triang_sup);
MPI_Type_commit(&tipo_triang_sup);

// Tipo para la matriz triangular inferior
for (int i = 0; i < N; i++) {
    longitudes[i] = i + 1;
    desplazamientos[i] = i * N * sizeof(int);
}
MPI_Type_create_hindexed(N, longitudes, desplazamientos, MPI_INT, &tipo_triang_inf);
MPI_Type_commit(&tipo_triang_inf);

// Proceso principal inicializa la matriz y la envía
if (mirango == 0) {
    inicializar_matriz(matriz);
    printf("Proceso %d tiene la matriz completa:\n", mirango);
    imprimir_matriz(matriz);
    fflush(stdout);

    // Enviar la matriz triangular superior al proceso 1
    if (tamano > 1) {
        MPI_Send(matriz, 1, tipo_triang_sup, 1, 0, MPI_COMM_WORLD);
    }

    // Enviar la matriz triangular inferior al proceso 2
    if (tamano > 2) {
        MPI_Send(matriz, 1, tipo_triang_inf, 2, 0, MPI_COMM_WORLD);
    }
}

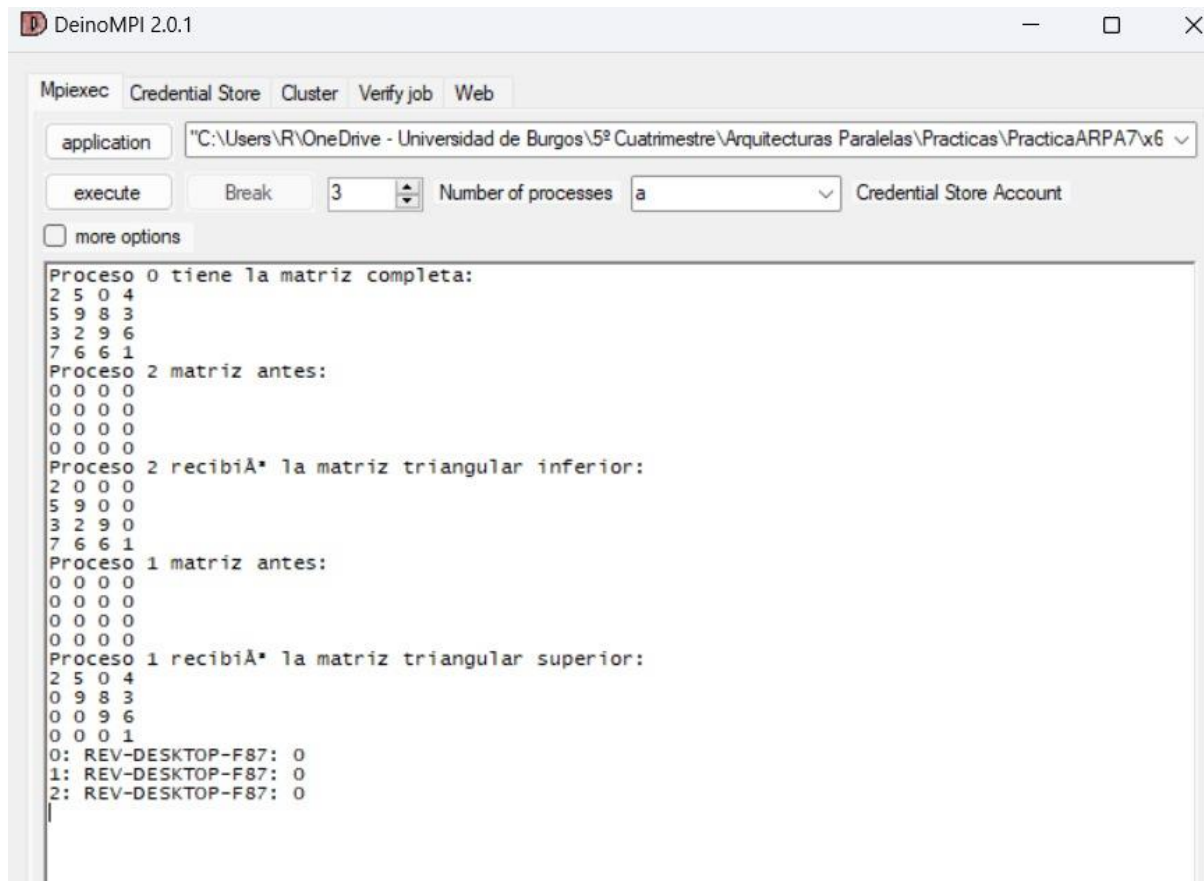
// Procesos receptores reciben su matriz correspondiente y la imprimen
if (mirango == 1) {
    int matriz_recibida_sup[N][N] = {};
    printf("Proceso %d matriz antes:\n", mirango);
    imprimir_matriz(matriz_recibida_sup);
    fflush(stdout);
    MPI_Recv(matriz_recibida_sup, 1, tipo_triang_sup, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    printf("Proceso %d recibió la matriz triangular superior:\n", mirango);
    imprimir_matriz(matriz_recibida_sup);
}
else if (mirango == 2) {
    int matriz_recibida_inf[N][N] = {};
    printf("Proceso %d matriz antes:\n", mirango);
    imprimir_matriz(matriz_recibida_inf);
    fflush(stdout);
}

```

```
        MPI_Recv(matriz_recibida_inf, 1, tipo_triang_inf, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Proceso %d recibió la matriz triangular inferior:\n", mirango);
        imprimir_matriz(matriz_recibida_inf);
    }

    // Liberar los tipos de datos derivados y finalizar MPI
    MPI_Type_free(&tipo_triang_sup);
    MPI_Type_free(&tipo_triang_inf);
    MPI_Finalize();
    return 0;
}
```

Resultados



```
Proceso 0 tiene la matriz completa:
2 5 0 4
5 9 8 3
3 2 9 6
7 6 6 1
Proceso 2 matriz antes:
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
Proceso 2 recibirá la matriz triangular inferior:
2 0 0 0
5 9 0 0
3 2 9 0
7 6 6 1
Proceso 1 matriz antes:
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
Proceso 1 recibirá la matriz triangular superior:
2 5 0 4
0 9 8 3
0 0 9 6
0 0 0 1
0: REV-DESKTOP-F87: 0
1: REV-DESKTOP-F87: 0
2: REV-DESKTOP-F87: 0
```

Descripción del Código

1. Inicialización de MPI

Se incluyen las bibliotecas necesarias para programar con MPI.

2. Declaración de variables:

Se declaran variables para almacenar el rango del proceso actual y el número total de procesos. Además se declara una variable para definir el tamaño de la matriz.

3. Funciones sobre la inicialización e impresión de la matriz

Se crean dos funciones, una sirve para inicializar la matriz con los números aleatorios y la otra para mostrar el estado de la matriz por pantalla. Para que los números aleatorios no sean siempre los mismos se cambia la semilla según el rango del proceso.

4. Inicialización de la matriz y tipos de datos MPI:

Se declara la matriz y los tipos de datos MPI para representar las matrices triangulares superior e inferior.

5. Creación de tipos de datos MPI para matrices triangulares:

Se crean dos tipos de datos MPI usando "*MPI_Type_create_hindexed*" para representar las matrices triangulares superior e inferior.

6. Proceso principal

En este bloque de código, el proceso con rango 0 inicializa y muestra por pantalla la matriz, envía la matriz triangular superior al proceso 1 y envía la matriz triangular inferior al proceso 2.

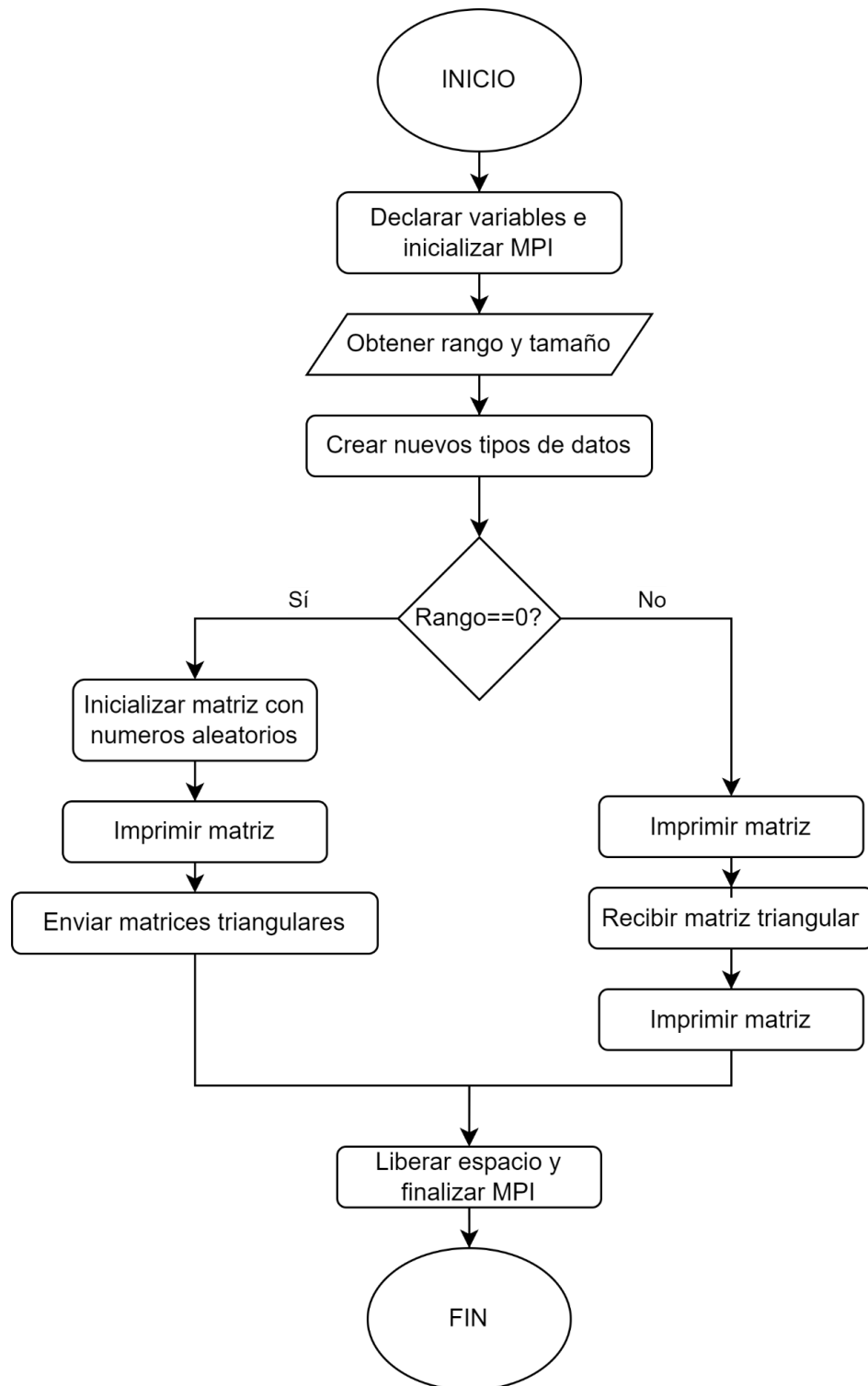
7. Procesos secundarios

En estos bloques de código, tanto el proceso 1 como el 2 imprimen la matriz que tienen antes de recibir nada del proceso 0, reciben la matriz triangular, sea inferior o superior, y la imprimen.

8. Finalización de MPI

Se liberan los tipos de datos MPI, se finaliza MPI al salir del bucle y se devuelve 0 para indicar una terminación exitosa del programa.

Diagrama de flujo



Cuestiones

En la práctica realizada, ¿qué problemas aparecen si se realiza reserva dinámica de memoria para crear espacio para las matrices? ¿Cómo afecta esto a la definición de nuevos tipos de datos?

La reserva dinámica de memoria para matrices puede romper la contigüidad de la memoria, lo que complica la creación de tipos de datos derivados en MPI. Esto se debe a que cada fila puede terminar en una ubicación de memoria separada, y para comunicar secciones de estas matrices, se tendrían que especificar desplazamientos individuales.

Plantear otras situaciones en las que sea de utilidad la definición de tipos de datos derivados.

Otras situaciones en las que serían útiles los tipos de datos derivados

- Simulaciones de Partículas: Enviar conjuntos de propiedades de partículas en simulaciones físicas.
- Álgebra Lineal: Facilitar la comunicación de columnas en operaciones matriciales.
- Estructuras de Datos Complejas: Transferir objetos complejos en bases de datos y sistemas de archivos distribuidos.
- Simulación de Redes: Comunicar información detallada de sub-redes.
- E/S en Paralelo con MPI-IO: Describir y acceder a patrones complejos de datos en archivos.

Conclusión

Este código es un ejemplo del uso de programación paralela con MPI, destacando la importancia de tipos derivados para facilitar el intercambio de datos entre procesos. La estrategia paralela mejora la eficiencia y escalabilidad del código, permitiendo resolver problemas más complejos en entornos distribuidos. La implementación de tipos derivados no solo simplifica la gestión de datos complejos, sino que también mejora la legibilidad y mantenibilidad del código, contribuyendo a un desarrollo eficiente y adaptable en diferentes arquitecturas informáticas. En resumen, este código representa un ejemplo valioso de cómo combinar programación paralela y tipos derivados para optimizar el rendimiento y la mantenibilidad de aplicaciones.