UA

Universidad

de Alicante

ESCUELA POLITÉCNICA SUPERIOR

Introducción a MPI 3

Curso académico 2024 / 2025

Semestre 2

Grado en Ingeniería en Inteligencia Artificial

Grupo 1

Profesora: Ricardo Moreno Rodríguez

Alumno: Santiago Álvarez Geanta

Universitat d'Alacant
Universidad de Alicante

Abril 2025

ÍNDICE

Introducción

Esta práctica tiene como objetivo aplicar de forma integrada los conocimientos adquiridos en torno a la programación con MPI (Message Passing Interface) en un entorno de computación distribuida. El planteamiento consiste en simular una red de sensores distribuidos que recogen datos (como temperaturas o presiones) para calcular medias locales y detectar posibles valores críticos que activen una alerta global.

A través de esta simulación, se hace uso de las principales funciones de MPI: distribución y recolección de datos (MPI_Scatter, MPI_Gather), sincronización (MPI_Barrier), medición de tiempos (MPI_Wtime) y gestión básica de errores. El enfoque busca no solo evaluar la correcta implementación de la API de MPI, sino también el análisis del paralelismo y las estrategias de comunicación entre procesos.

Desarrollo

Explicación general

El sistema distribuido sigue una arquitectura maestro-trabajadores, donde el proceso 0 actúa como coordinador que genera los datos y centraliza los resultados. Cada proceso trabajador recibe una parte de los datos, calcula su media local y la envía de vuelta al coordinador. Si alguna media local supera un umbral crítico, se genera una alerta.

En primer lugar, se inicializa el entorno de MPI con **MPI_Init**. Cada proceso obtiene su identificador (rank) y el número total de procesos (size). Posteriormente, el proceso 0 calcula el número total de elementos y verifica que sea múltiplo del número de procesos. Luego, genera números aleatorios entre 0 y 100. A continuación, mediante MPI_Scatter, el proceso 0 distribuye equitativamente los datos generados entre todos los procesos, de modo que cada uno recibe un subconjunto de datos (5 enteros). Cada proceso calcula la media de los valores que recibió y, utilizando **MPI_Gather**, envía su media local al proceso 0, que las recopila todas. Este último analiza las medias locales y, si alguna supera el umbral definido (por ejemplo, 50), emite una alerta e identifica al proceso correspondiente. Para garantizar la coherencia en el análisis y la medición del tiempo, se utilizan barreras de sincronización (MPI_Barrier) antes y después de la recolección. Asimismo, se emplea **MPI_Wtime** para medir con precisión el tiempo total de ejecución desde el inicio de la distribución de datos hasta la finalización del análisis.

Código del ejercicio

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

#define VALORES_POR_PROCESO 5
#define UMBRAL 50

double calcularMedia(int *datos, int n) {
    int suma = 0;
    for (int i = 0; i < n; i++) {
        suma += datos[i];
    }
    return (double)suma / n;
}

int main(int argc, char *argv[]) {
    int rank, size;
    int *datos = NULL;
    int totalElementos;
    int *localDatos;
    double localMedia;
    double *mediasLocales = NULL;
    double t_inicial, t_final;

    // Inicialización del entorno MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    totalElementos = VALORES_POR_PROCESO * size;

    // Verificación de que el número de elementos sea múltiplo del número de procesos
    if(totalElementos % size != 0) {
        if(rank == 0) {
            fprintf(stderr, "El número total de elementos (%d) debe ser múltiplo de la cantidad de procesos (%d).\n", totalElementos, size);
        }
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }

    // Reserva de memoria para los datos locales en cada proceso
    localDatos = (int *)malloc(VALORES_POR_PROCESO * sizeof(int));
    if(localDatos == NULL) {
        fprintf(stderr, "Error al asignar memoria en el proceso %d.\n", rank);
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }

    // El proceso 0 genera los datos aleatorios
    if(rank == 0) {
        datos = (int *)malloc(totalElementos * sizeof(int));
        if(datos == NULL) {
            fprintf(stderr, "Error al asignar memoria en el proceso 0.\n");
            MPI_Finalize();
            exit(EXIT_FAILURE);
        }
        srand(time(NULL));
        printf("Datos generados:\n");
        for (int i = 0; i < totalElementos; i++) {
            datos[i] = rand() % 101; // Números aleatorios entre 0 y 100
            printf("%d ", datos[i]);
        }
        printf("\n");
    }

    // Sincronización antes de iniciar el conteo de tiempo
    MPI_Barrier(MPI_COMM_WORLD);
    t_inicial = MPI_Wtime();

    // Distribución de los datos a todos los procesos
    MPI_Scatter(datos, VALORES_POR_PROCESO, MPI_INT, localDatos, VALORES_POR_PROCESO, MPI_INT, 0, MPI_COMM_WORLD);

    // Cada proceso calcula la media local de sus datos
    localMedia = calcularMedia(localDatos, VALORES_POR_PROCESO);

    // El proceso 0 reserva memoria para las medias locales recibidas
    if(rank == 0) {
        mediasLocales = (double *)malloc(size * sizeof(double));
        if(mediasLocales == NULL) {
            fprintf(stderr, "Error al asignar memoria para las medias en el proceso 0.\n");
            MPI_Finalize();
            exit(EXIT_FAILURE);
        }
    }

    // Se recogen las medias locales en el proceso 0
    MPI_Gather(&localMedia, 1, MPI_DOUBLE, mediasLocales, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Sincronización previa al análisis final
    MPI_Barrier(MPI_COMM_WORLD);

    // El proceso 0 analiza las medias y verifica el umbral crítico
    if(rank == 0) {
        printf("\nMedias locales por proceso:\n");
        for(int i = 0; i < size; i++) {
            printf("Proceso %d: %f\n", i, mediasLocales[i]);
        }

        // Verifica si alguna media supera el umbral
        int alerta = 0;
        printf("\nProcesos que superan el umbral de %d:\n", UMBRAL);
        for(int i = 0; i < size; i++) {
            if(mediasLocales[i] > UMBRAL) {
                printf("¡Alerta! Proceso %d con media %f\n", i, mediasLocales[i]);
                alerta = 1;
            }
        }
        if(!alerta) {
            printf("Ningún proceso supera el umbral crítico.\n");
        }
        t_final = MPI_Wtime();
        printf("\nTiempo total de ejecución: %f segundos\n", t_final - t_inicial);
    }

    // Liberación de memoria
```

Conclusión

Esta práctica permite consolidar de forma práctica los conocimientos sobre programación paralela con MPI, aplicando diversas formas de comunicación colectiva (Scatter, Gather), sincronización (Barrier) y análisis de datos distribuidos.

La simulación ilustra un caso realista: un sistema distribuido de sensores donde es necesario realizar análisis locales y una agregación centralizada para detectar eventos anómalos. El uso de **MPI_Wtime** añade una dimensión de evaluación del rendimiento, fundamental en aplicaciones de cómputo de alto rendimiento.

Además, hemos observado aspectos típicos de diseño paralelo, como el desequilibrio de carga entre procesos (el proceso 0 tiene más trabajo) y la importancia de validar correctamente la configuración de datos para evitar errores de distribución.

Esta práctica cierra adecuadamente la introducción a MPI, integrando todos los conceptos fundamentales vistos en clase en un proyecto práctico, funcional y fácilmente escalable