Práctica 9.3

Introducción a MPI (Práctica Final)

Jordi Blasco Lozano

Computación de alto rendimiento

Grado en Inteligencia Artificial

## Índice:

[Índice: 2](#_Toc196042572)

[1. Introducción 2](#_Toc196042573)

[2. Desarrollo 3](#_Toc196042574)

[Código 3](#_Toc196042575)

[Ejecución 5](#_Toc196042576)

[3. Preguntas de reflexión 6](#_Toc196042577)

## Introducción

En esta práctica final de Introducción a MPI se aborda la simulación de una red distribuida de sensores que recogen valores numéricos y colaboran para calcular medias locales y detectar posibles condiciones críticas, mediante el uso de las siguientes primitivas de MPI:

* **Inicialización y finalización**: MPI\_Init, MPI\_Finalize
* **Identificación de procesos**: MPI\_Comm\_rank, MPI\_Comm\_size
* **Comunicación colectiva**:
* Distribución de datos con MPI\_Scatter
* Recolección de resultados con MPI\_Gather
* Sincronización con MPI\_Barrier
* **Medición de rendimiento**: MPI\_Wtime

## Desarrollo

El siguiente programa que simula una red de sensores distribuidos usando MPI funciona de la siguiente forma:

* El proceso 0 genera un array de valores enteros y lo reparte equitativamente a todos los procesos mediante MPI\_Scatter.
* Cada proceso calcula la media de sus valores locales y, tras sincronizarse con MPI\_Barrier, envía ese resultado al proceso 0 usando MPI\_Gather.
* El proceso 0 recopila todas las medias, identifica cuáles superan un umbral crítico y emite las alertas correspondientes.

## Código

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM\_VALUES 5    // Número de valores por proceso

#define THRESHOLD 50    // Umbral crítico para la alerta

int main(int argc, char \*argv[]) {

    int rank, size;

    int \*data = NULL;

    double start\_time, end\_time;

    double local\_sum = 0.0, local\_avg;

    double \*local\_avgs = NULL;

    int i;

    // Inicialización de MPI

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

    // Cálculo del número total de elementos

    int total\_elements = NUM\_VALUES \* size;

    // Comprobación de robustez: total\_elements debe ser múltiplo de size

    if (total\_elements % size != 0) {

        if (rank == 0) {

            fprintf(stderr, "Error: total\_elements (%d) no es múltiplo de número de procesos

(%d)\n", total\_elements, size);

        }

        MPI\_Finalize();

        return EXIT\_FAILURE;

    }

    // El proceso 0 asigna y genera los datos

    if (rank == 0) {

        data = malloc(total\_elements \* sizeof(int));

        // Semilla basada en tiempo MPI para evitar <time.h>

        srand((unsigned)(MPI\_Wtime() \* 1000));

        for (i = 0; i < total\_elements; i++) {

            data[i] = rand() % 100;  // Valores aleatorios entre 0 y 99

        }

    }

// Cada proceso reserva espacio para sus datos locales

    int local\_data[NUM\_VALUES];

    // Sincronización antes de iniciar la medición de tiempo

    MPI\_Barrier(MPI\_COMM\_WORLD);

    start\_time = MPI\_Wtime();

    // Distribución de datos a todos los procesos

    MPI\_Scatter(data, NUM\_VALUES, MPI\_INT, local\_data, NUM\_VALUES, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Cálculo de la suma y media local

    for (i = 0; i < NUM\_VALUES; i++) {

        local\_sum += local\_data[i];

    }

    local\_avg = local\_sum / NUM\_VALUES;

    // El proceso 0 reserva espacio para recopilar medias

    if (rank == 0) {

        local\_avgs = malloc(size \* sizeof(double));

    }

    // Recolección de las medias locales en el proceso 0

    MPI\_Gather(&local\_avg, 1, MPI\_DOUBLE, local\_avgs, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    end\_time = MPI\_Wtime();

    // Proceso 0 muestra resultados y alertas

    if (rank == 0) {

        printf("Tiempo de ejecución: %f segundos\n", end\_time - start\_time);

        printf("Medias locales calculadas:\n");

        for (i = 0; i < size; i++) {

            printf("Proceso %d: %.2f\n", i, local\_avgs[i]);

        }

        printf("\nAlertas (media > %d):\n", THRESHOLD);

        for (i = 0; i < size; i++) {

            if (local\_avgs[i] > THRESHOLD) {

                printf("Proceso %d con media %.2f\n", i, local\_avgs[i]);

            }

        }

    }

    // Liberación de memoria y finalización de MPI

    if (data != NULL) free(data);

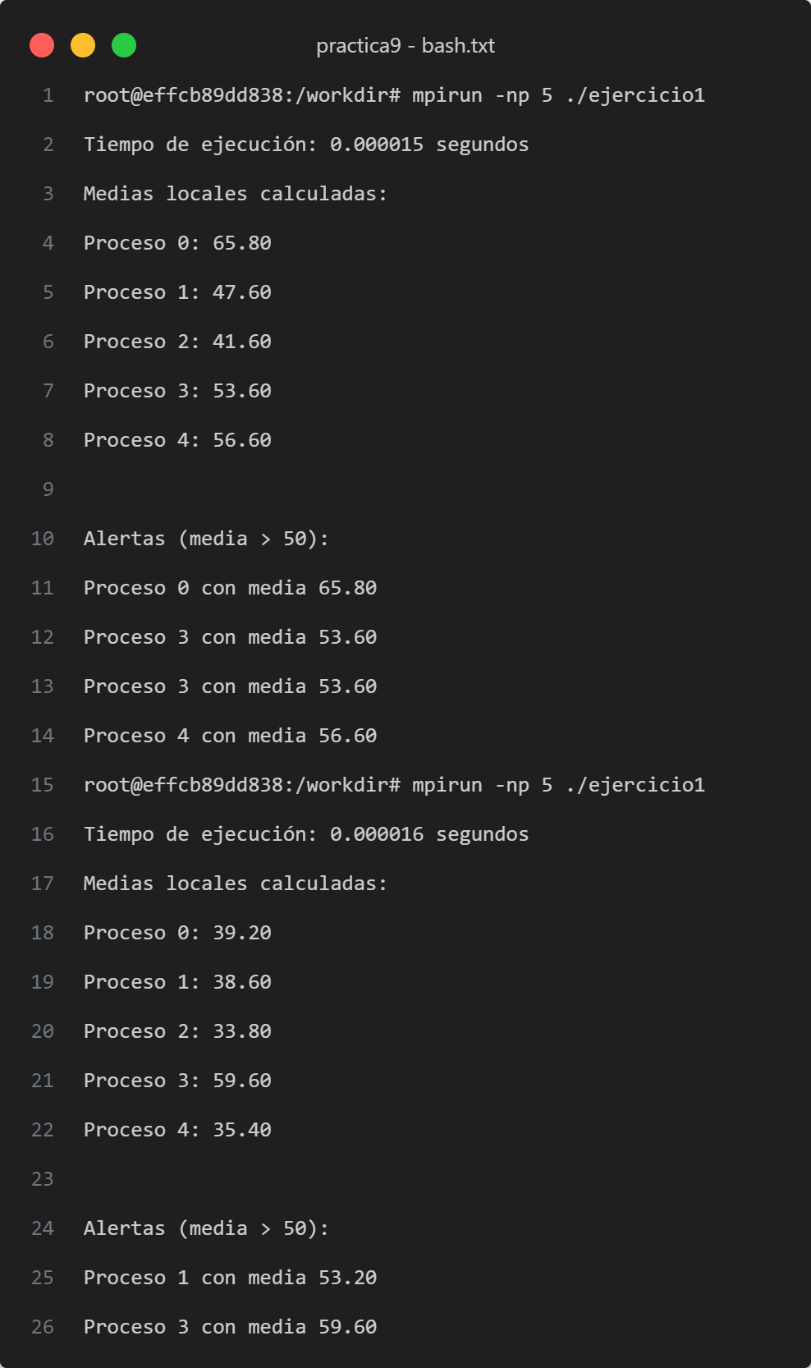
    if (local\_avgs != NULL) free(local\_avgs);

    MPI\_Finalize();

    return 0;

}

## Ejecución

Como podemos observar en la ejecución del programa, al ejecutar el código con 5 procesos:

* El tiempo de ejecución es prácticamente despreciable 0.000015s, lo que demuestra el bajo coste de la comunicación para un tamaño de datos tan pequeño.
* Las medias locales varían en cada ejecución, reflejando la naturaleza aleatoria de los datos generados por el proceso raíz.
* El mecanismo de alerta identifica correctamente todos los procesos cuyas medias superan el umbral crítico de 50.

En conjunto, estos resultados confirman que la distribución de trabajo y la recolección de resultados mediante las directivas MPI funcionan de forma eficiente y fiable. Para volúmenes de datos mayores o escenarios reales, este patrón se mantendría, y la ventaja paralela se haría aún más evidente frente al procesamiento secuencial.

## Preguntas de reflexión

**¿Qué tipo de comunicación has usado en cada parte del programa?**

**Comunicación colectiva**

MPI\_Scatter: Se utiliza para distribuir el array generado por el proceso 0 a todos los procesos. Es una comunicación colectiva en la que cada proceso recibe un bloque de datos.

* MPI\_Gather: Se emplea para recolectar las medias locales calculadas por cada proceso en el proceso 0.
* MPI\_Barrier: Se usa para sincronizar a todos los procesos antes de realizar el análisis de los resultados.

**Comunicación punto a punto**

En este ejemplo no se hace uso explícito de comunicaciones punto a punto como MPI\_Send y MPI\_Recv, ya que se ha optado por emplear las comunicaciones colectivas que abstraen estas operaciones en un contexto distribuido.

**¿Hay algún tipo de desequilibrio en el trabajo de los procesos?**

En este diseño, la carga de trabajo se distribuye de forma casi equitativa:

* **Procesos secundarios**: Cada uno se encarga de calcular la media de sus propios datos, lo cual es una operación de coste constante para cada proceso
* **Proceso 0**: Además de repartir y recopilar los datos, realiza la generación del array inicial y el análisis final (impresión y detección de alertas). Esto introduce un pequeño desequilibrio, ya que el proceso 0 realiza tareas adicionales.

**¿Dónde podrías aplicar paralelismo adicional?**

Existen algunas áreas donde se puede incrementar el paralelismo:

* **Generación de datos**: Si la cantidad de datos a generar fuera muy grande, se podría distribuir la generación de los datos entre varios procesos en lugar de hacerlo solo en el proceso 0.
* **Cálculo de la media**: En el caso de conjuntos de datos mayores por proceso, se podría aplicar un paralelismo dentro de cada proceso (por ejemplo, usando OpenMP) para dividir el cálculo de la suma local.
* **Análisis de alertas**: En lugar de centralizar la verificación de las alertas en el proceso 0, cada proceso podría determinar localmente si sus datos exceden el umbral y luego comunicarse mediante una operación colectiva como MPI\_Reduce o MPI\_Allreduce para consolidar la información.

**¿Qué otras estrategias se te ocurren para lanzar alertas o combinar resultados?**

Algunas estrategias adicionales podrían incluir:

* **Uso de MPI\_Reduce**: Se podría utilizar MPI\_Reduce para obtener, por ejemplo, el máximo promedio de todos los procesos y determinar si se supera el umbral. Esto evitaría enviar todos los promedios al proceso 0 si solo interesa conocer el máximo.
* **Comunicación asíncrona**: Emplear comunicaciones no bloqueantes (MPI\_Isend, MPI\_Irecv) para que los procesos puedan continuar con otros cálculos mientras se transmite la información, lo cual puede mejorar la eficiencia.
* **Alertas locales con broadcast**: Cada proceso podría evaluar su propia condición de alerta y, en caso de detectarla, enviar una señal o mensaje a todos los demás mediante MPI\_Bcast para una reacción inmediata ante valores críticos.
* **Estrategias híbridas**: Combinando paralelismo a nivel de procesos (MPI) con paralelismo a nivel de hilos (OpenMP o similar) para optimizar tanto la generación, procesamiento y análisis de grandes volúmenes de datos.