Práctica 9.3
Introducción a MPI (Práctica Final)

Jordi Blasco Lozano
Computación de alto rendimiento
Grado en Inteligencia Artificial

Índice:

1. 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

2. 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

```c
#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.

practica9 - bash.txt

```
root@effcb89dd838:/workdir# mpirun -np 5 ./ejercicio1
Tiempo de ejecución: 0.000015 segundos
Medias locales calculadas:
Proceso 0: 65.80
Proceso 1: 47.60
Proceso 2: 41.60
Proceso 3: 53.60
Proceso 4: 56.60

Alertas (media > 50):
Proceso 0 con media 65.80
Proceso 3 con media 53.60
Proceso 3 con media 53.60
Proceso 4 con media 56.60
root@effcb89dd838:/workdir# mpirun -np 5 ./ejercicio1
Tiempo de ejecución: 0.000016 segundos
Medias locales calculadas:
Proceso 0: 39.20
Proceso 1: 38.60
Proceso 2: 33.80
Proceso 3: 59.60
Proceso 4: 35.40

Alertas (media > 50):
Proceso 1 con media 53.20
Proceso 3 con media 59.60
```

3. 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.