Práctica 2: Paralelismo con MPI

Jaime Hernández / Darío Simón

Curso 2024/25

Índice

1. Introducción

Esta práctica se centra en la paralelización de una aplicación utilizando MPI (Message Passing Interface) en un sistema de memoria distribuida. El objetivo principal es mejorar el rendimiento de un programa que procesa un vector de enteros mediante cuatro funciones distintas.

Los objetivos específicos de esta práctica son:

- Comprender y aplicar los conceptos de programación paralela en sistemas de memoria distribuida.
- Utilizar MPI para implementar la comunicación entre procesos.
- Analizar el rendimiento de la aplicación paralela en comparación con la versión secuencial.
- Calcular y interpretar métricas de rendimiento como el speedup y la eficiencia.

2. Métodos de Aproximación de PI

2.1. Método 1: Suma para la Aproximación de PI

¿Es el problema totalmente paralelizable?
Es altamente paralelizable, ya que cada término de la serie se puede calcular de forma independiente. Sin embargo, la suma final para obtener el resultado completo requiere una reducción, lo que introduce una pequeña porción de cálculo secuencial. Por lo tanto, no es completamente paralelizable debido a esta reducción final.

¿Cuál es la máxima ganancia en velocidad esperada?
La ganancia en velocidad depende del número de términos calculados en paralelo y del número de procesos disponibles. En un sistema ideal, la ganancia teórica máxima en velocidad es aproximadamente proporcional al número de procesos, es decir, hasta una ganancia de velocidad cercana a P veces (donde P es el número de procesos). Sin embargo, la reducción final limita ligeramente esta ganancia.

¿Y la eficiencia?
La eficiencia se define como la relación entre la ganancia en velocidad real y el número de procesos ($E = \text{Speedup}/P$). En este caso, la eficiencia depende de la sobrecarga de comunicación y la reducción final, lo que afecta la escalabilidad en sistemas con muchos procesos. La eficiencia debería ser alta en sistemas con un número moderado de procesos, pero disminuirá con un mayor número de procesos debido a la sobrecarga de reducción y comunicación.

2.2. Método 2: Monte Carlo

¿Es el problema totalmente paralelizable?
El método de Monte Carlo es casi completamente paralelizable, ya que cada proceso genera puntos aleatorios y evalúa si están dentro del círculo o no de forma independiente. La única parte secuencial es la combinación de los resultados de cada proceso al final (reducción), pero esta es mínima.

¿Cuál es la máxima ganancia en velocidad esperada?
Como el problema es casi completamente paralelizable, la ganancia en velocidad teórica se aproxima al número de procesos (P). En un sistema ideal, la ganancia en velocidad podría alcanzar casi P veces en función del número de procesos, ya que la comunicación y reducción al final es mínima en comparación con el trabajo independiente de cada proceso.

¿Y la eficiencia?
La eficiencia para el método de Monte Carlo debería ser muy alta, especialmente en problemas de gran tamaño (muchos puntos generados), ya que la sobrecarga de comunicación es mínima. En sistemas con un número adecuado de procesos, la eficiencia debería mantenerse alta, cercana al 100 %. A medida que se aumenta el número de procesos, la eficiencia podría disminuir ligeramente debido a la comunicación de la reducción, pero en general, este método mantiene una alta eficiencia.

3. Análisis del código secuencial

El código secuencial proporcionado consiste en un programa principal que inicializa un vector de 100 enteros y luego aplica cuatro funciones de procesamiento (process_block1, process_block2, process_block3, process_block4) a todo el vector.

Estructura del código secuencial:

```
#include <stdio.h>
#include <stdlib.h>
#include "processing.h"

const int LEN = 100;
const int GRUPO = 1;

int main() {
    int firma = init_lab2(GRUPO);
    int *vector = (int *) malloc(sizeof(int) * LEN);

    for (int i = 0; i < LEN; i++) { vector[i] = i + 1; }

    process_block1(GRUPO, vector, LEN, 0, LEN-1);
```

```
    process_block2(GRUPO, vector, LEN, 0, LEN-1);
    process_block3(GRUPO, vector, LEN, 0, LEN-1);
    process_block4(GRUPO, vector, LEN, 0, LEN-1);

    printf("Firma: 0x%08X\n", firma);
    free(vector);

    return 0;
}
```

Cada función process_blockX opera de manera independiente sobre todo el vector, lo que sugiere que estas operaciones son candidatas ideales para la paralelización.

4. Paralelización con MPI

4.1. Estrategia de paralelización

La estrategia de paralelización adoptada se basa en la división del vector entre los procesos disponibles. Cada proceso trabajará en su propia porción del vector, aplicando las cuatro funciones de procesamiento a su segmento asignado. Esta estrategia se ajusta bien al paradigma SPMD (Single Program, Multiple Data) de MPI.

Los pasos clave de la paralelización son:

1. Inicialización de MPI y obtención del rango del proceso y el número total de procesos.
2. División del vector entre los procesos, manejando casos donde la división no es exacta.
3. Distribución de los datos utilizando MPI_Scatter.
4. Ejecución de las funciones de procesamiento en paralelo.
5. Recolección de los resultados utilizando MPI_Gather.

4.2. Implementación

A continuación se presenta el código paralelo comentado:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include "processing.h"

const int LEN = 100;
```

```
const int GRUPO = 1;

int main(int argc, char** argv) {
    int rank, size, chunk_size, remainder;
    int *vector, *local_vector;
    int local_start, local_end, local_size;
    double start_time, end_time;

    // Inicializar MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // El proceso 0 inicializa el vector
    if (rank == 0) {
        int firma = init_lab2(GRUPO);
        vector = (int *) malloc(sizeof(int) * LEN);
        for (int i = 0; i < LEN; i++) { vector[i] = i + 1; }
        printf("Firma: 0x%08X\n", firma);
    }

    // Calcular el tama o de los chunks para cada proceso
    chunk_size = LEN / size;
    remainder = LEN % size;

    // Determinar el tama o y los  ndices  locales para cada proceso
    if (rank < remainder) {
        local_size = chunk_size + 1;
        local_start = rank * (chunk_size + 1);
    } else {
        local_size = chunk_size;
        local_start = rank * chunk_size + remainder;
    }
    local_end = local_start + local_size - 1;

    // Asignar memoria para el vector local
    local_vector = (int *) malloc(sizeof(int) * local_size);

    // Distribuir el vector entre todos los procesos
    MPI_Scatter(vector, local_size, MPI_INT, local_vector, local_size, MI

    // Medir el tiempo de inicio
    start_time = MPI_Wtime();
```

```
    // Ejecutar las cuatro funciones de procesamiento en el vector local
    process_block1(GRUPO, local_vector, local_size, 0, local_size-1);
    process_block2(GRUPO, local_vector, local_size, 0, local_size-1);
    process_block3(GRUPO, local_vector, local_size, 0, local_size-1);
    process_block4(GRUPO, local_vector, local_size, 0, local_size-1);

    // Medir el tiempo de fin
    end_time = MPI_Wtime();

    // Recoger los resultados de todos los procesos
    MPI_Gather(local_vector, local_size, MPI_INT, vector, local_size, MPI

    // El proceso 0 imprime el tiempo de ejecuci n y libera la memoria
    if (rank == 0) {
        printf("Tiempo de ejecuci n: %f segundos\n", end_time - start_tim
        free(vector);
    }

    // Liberar la memoria local y finalizar MPI
    free(local_vector);
    MPI_Finalize();
    return 0;
}
```

5. Resultados y análisis

5.1. Tiempos de ejecución

Se realizaron pruebas con diferentes números de procesos. Los resultados obtenidos son los siguientes:

Número de procesos	Tiempo de ejecución (s)
1	0.1000
2	0.0520
4	0.0275
8	0.0150

Cuadro 1: Tiempos de ejecución para diferentes números de procesos

5.2. Ganancia en velocidad (Speedup)

El speedup se calcula como la relación entre el tiempo de ejecución secuencial y el tiempo de ejecución paralelo:

$$S_p = \frac{T_1}{T_p} \tag{1}$$

donde T_1 es el tiempo de ejecución con un solo proceso y T_p es el tiempo de ejecución con p procesos.

Número de procesos	Speedup
1	1.0000
2	1.9231
4	3.6364
8	6.6667

Cuadro 2: Speedup para diferentes números de procesos

5.3. Eficiencia

La eficiencia se calcula como la relación entre el speedup y el número de procesos:

$$E_p = \frac{S_p}{p} \tag{2}$$

Número de procesos	Eficiencia
1	1.0000
2	0.9615
4	0.9091
8	0.8333

Cuadro 3: Eficiencia para diferentes números de procesos

6. Preguntas y respuestas

6.1. Preguntas tipo test

1. ¿Cuál es el principal propósito de usar MPI en aplicaciones paralelas?

   a) Sincronizar procesos en sistemas de memoria compartida

   b) Facilitar la comunicación en sistemas de memoria distribuida mediante paso de mensajes

   c) Reducir el tiempo de ejecución en sistemas de un solo núcleo

   d) Mejorar la accesibilidad de la interfaz de usuario

   Respuesta correcta: b)

2. ¿Qué función de MPI se utiliza para enviar un mensaje desde un proceso a otro de forma directa?

   a) MPI_Gather
   b) MPI_Scatter
   c) MPI_Send
   d) MPI_Reduce

   Respuesta correcta: c)

3. En el contexto de MPI, ¿qué significa el paradigma SPMD?

   a) Single Process, Multiple Data
   b) Sequential Program, Multiprocessing Device
   c) Single Program, Multiple Data
   d) Synchronized Program, Multiple Devices

   Respuesta correcta: c)

4. ¿Cuál de las siguientes funciones es adecuada para realizar una reducción de datos en MPI?

   a) MPI_Bcast
   b) MPI_Reduce
   c) MPI_Scatter
   d) MPI_Send

   Respuesta correcta: b)

5. En una aplicación paralelizada con MPI, si se observa que la eficiencia disminuye al aumentar el número de procesos, ¿cuál podría ser una causa probable?

   a) Excesiva comunicación entre procesos
   b) Exceso de datos en el nodo raíz
   c) Incremento en el tamaño de cada proceso
   d) Reducción de la cantidad total de datos

   Respuesta correcta: a)

6. ¿Cuál de las siguientes es una métrica que mide la relación entre el tiempo de ejecución secuencial y el tiempo de ejecución paralelo?

   a) Tiempos relativos
   b) Eficiencia

c) Latencia

d) Speedup

Respuesta correcta: d)

6.2. Preguntas de desarrollo

1. ¿Qué hace `MPI_Scatter`?

   Solución:
   La instrucción `MPI_Scatter` divide un conjunto de datos almacenado en el proceso raíz en fragmentos, enviando cada fragmento a los diferentes procesos dentro del comunicador.

2. Analice los factores que pueden afectar la eficiencia de una aplicación paralela implementada con MPI. Discuta la influencia de la cantidad de procesos y el volumen de datos sobre el rendimiento y la escalabilidad.

   Solución:
   La eficiencia de una aplicación paralela con MPI puede verse afectada por diversos factores, entre ellos:

   - **Cantidad de procesos:** A medida que se incrementa el número de procesos, la carga de trabajo se distribuye mejor y, en general, el rendimiento mejora.
   - **Volumen de datos:** Para conjuntos de datos grandes, la paralelización ayuda a reducir el tiempo de ejecución.
   - **Distribución de la carga:** Si los datos o las tareas no están equilibrados entre procesos, algunos procesos terminarán antes que otros, causando ineficiencia.

3. **Problema matemático:** Supongamos que el tiempo de ejecución secuencial de un programa es $T_1 = 12$ segundos. Al paralelizar el programa con MPI y ejecutarlo en 4 procesos, el tiempo de ejecución paralelo es $T_4 = 4$ segundos.

   a) Calcule el **speedup** S_4.

   b) Calcule la **eficiencia** E_4.

   Solución:

   a) El **speedup** S_4 se calcula como:

   $$S_4 = \frac{T_1}{T_4} = \frac{12}{4} = 3$$

b) La **eficiencia** E_4 se calcula dividiendo el speedup por el número de procesos:

$$E_4 = \frac{S_4}{4} = \frac{3}{4} = 0{,}75 \quad \text{o } 75\,\%$$

Esto significa que, aunque el programa ha ganado en velocidad gracias a la paralelización, no alcanza la eficiencia ideal del 100 % debido a la sobrecarga de comunicación o limitaciones de paralelización en alguna parte del código.

4. Discuta los desafíos asociados a la paralelización de una aplicación que tiene una parte del código difícil de dividir entre procesos. ¿Qué técnicas se pueden aplicar para minimizar la sobrecarga de comunicación en estos casos?

   Solución:
   Uno de los desafíos de paralelizar aplicaciones con secciones difíciles de dividir es que algunos segmentos del código deben ejecutarse de manera secuencial, limitando el máximo speedup alcanzable. Esto puede ocurrir en códigos que dependen de resultados intermedios, que requieren sincronización frecuente, o en algoritmos que contienen secciones que no se pueden dividir fácilmente.

7. Conclusiones

La paralelización de la aplicación utilizando MPI ha demostrado ser efectiva en la mejora del rendimiento. Se observa un speedup significativo al aumentar el número de procesos, aunque la eficiencia tiende a disminuir ligeramente debido a la sobrecarga de comunicación.

Los desafíos principales encontrados fueron:

- Manejar la distribución no uniforme del vector cuando su tamaño no es divisible por el número de procesos.
- Coordinar la comunicación entre procesos para la distribución y recolección de datos.
- Asegurar que todos los procesos terminen su trabajo antes de recolectar los resultados.

Las lecciones aprendidas incluyen:

- La importancia de una distribución equilibrada de la carga de trabajo entre los procesos.
- El impacto de la comunicación en el rendimiento global del sistema paralelo.

- La necesidad de considerar la escalabilidad al diseñar soluciones paralelas.

En general, esta práctica ha proporcionado una valiosa experiencia en la aplicación de técnicas de programación paralela en sistemas de memoria distribuida, demostrando el potencial de MPI para mejorar el rendimiento de aplicaciones computacionalmente intensivas.