Práctica 3: Paralelismo a nivel de hilos Paralelización mediante OpenMP y programación asíncrona

Jaime Hernández
Ingeniería de los Computadores
Curso 2024/25

11 de diciembre de 2024

Índice

1. Introducción **4**

2. Objetivos **4**

3. Desarrollo de la Práctica **5**
3.1. Entrenamiento Previo OpenMP 5
3.1.1. Variable chunk 5
3.1.2. Análisis del pragma omp parallel 5
3.1.3. Función schedule 5
3.1.4. Medidas de rendimiento 6
3.2. Entrenamiento Previo std::async 6
3.2.1. Parámetro std::launch::async 6
3.2.2. Análisis de tiempos de ejecución 6
3.2.3. Diferencias entre wait() y get() 7
3.2.4. Ventajas de std::async frente a std::thread 7

4. Tutorial de Paralelismo con OpenMP **8**
4.1. Estructuras Básicas de Paralelización 8
4.1.1. Paralelización de Bucles 8
4.1.2. Reducción de Variables 8
4.2. Cláusulas de Sincronización 8
4.2.1. Secciones Críticas 8
4.2.2. Barreras 8
4.3. Patrones Comunes 9
4.3.1. Paralelismo de Tareas 9
4.3.2. Schedule 9
4.4. Consideraciones de Rendimiento 9

5. Tarea 3: Uso de reduction y sections **10**
5.1. 3.1 Análisis del Programa Secuencial 10
5.2. 3.2 Paralelización con reduction 10
5.3. 3.3 Paralelización con sections 11
5.4. 3.4 Paralelización con std::async 11
5.5. 3.5 Combinación de reduction y sections 11
5.6. 3.6 Combinación de reduction con std::async 12
5.7. 3.7 Análisis de Rendimiento 13

6. Tarea 4: Condiciones de Carrera **14**
6.1. Introducción 14
6.2. Análisis de Condiciones de Carrera 14
6.2.1. Código Original 14
6.2.2. Resultados Obtenidos 14
6.3. Inicialización de Vectores 15
6.3.1. Análisis de Métodos 15
6.3.2. Conclusiones 15

7. Tarea 5: Cálculo de PI **16**
7.1. Implementación de Métodos 16
7.1.1. Método de la Integral Definida 16
7.1.2. Método de Monte Carlo 16
7.2. Resultados Experimentales 17
7.2.1. Análisis de Precisión 17
7.2.2. Análisis de Rendimiento 17
7.3. Comparativa con MPI 17

8. Examen **18**
8.1. Preguntas Tipo Test 18
8.1.1. Cuestionario OpenMP 18

1. Introducción

Esta práctica se centra en la paralelización de aplicaciones utilizando OpenMP y programación asíncrona en C++. El objetivo principal es aprovechar los múltiples núcleos de procesamiento disponibles en sistemas modernos mediante técnicas de programación paralela.

2. Objetivos

- Paralelización de aplicaciones en máquinas paralelas de memoria centralizada mediante hilos
- Estudio y aplicación de la API OpenMP
- Implementación de programación asíncrona en C++
- Análisis de rendimiento y eficiencia
- Aplicación práctica en problemas de complejidad significativa

3. Desarrollo de la Práctica

3.1. Entrenamiento Previo OpenMP

3.1.1. Variable chunk

La variable `chunk` determina el tamaño de los bloques de iteraciones que se asignan a cada hilo. En este caso, `CHUNKSIZE = 100` significa que el bucle se divide en bloques de 100 iteraciones que se distribuyen entre los hilos disponibles. Esta técnica permite un mejor balance de carga y reduce la sobrecarga de sincronización.

3.1.2. Análisis del pragma omp parallel

El pragma `#pragma omp parallel shared(a,b,c,chunk) private(i)` define:

- **shared(a,b,c,chunk)**: Los arrays `a`, `b`, `c` y la variable `chunk` son compartidos entre todos los hilos. Esto es necesario porque:
  - Los arrays contienen los datos a procesar y todos los hilos necesitan acceder a ellos
  - `chunk` debe ser visible para todos los hilos para determinar el tamaño de los bloques
- **private(i)**: La variable `i` es privada para cada hilo porque:
  - Cada hilo necesita su propio contador de bucle
  - Evita condiciones de carrera en el índice del bucle
  - Permite que cada hilo mantenga su propia posición en su bloque de iteraciones

3.1.3. Función schedule

`schedule(dynamic,chunk)` determina cómo se distribuyen las iteraciones del bucle:

- **dynamic**: Las iteraciones se asignan a los hilos en tiempo de ejecución
- Otras opciones disponibles:
  - `static`: Distribución estática y uniforme
  - `guided`: Tamaño de bloque variable que decrece

- `auto`: El runtime decide la mejor estrategia
- `runtime`: Se decide en tiempo de ejecución

3.1.4. Medidas de rendimiento

OpenMP proporciona varias funciones para medir el rendimiento:

- `omp_get_wtime()`: Tiempo de ejecución en segundos
- `omp_get_num_threads()`: Número de hilos activos
- `omp_get_thread_num()`: ID del hilo actual
- Métricas calculables:
  - Speedup: $S = \frac{T_{sequential}}{T_{parallel}}$
  - Eficiencia: $E = \frac{S}{p}$ donde p es el número de procesadores
  - Sobrecarga de paralelización

3.2. Entrenamiento Previo std::async

3.2.1. Parámetro std::launch::async

`std::launch::async` indica que la tarea debe ejecutarse en un nuevo hilo de forma asíncrona. Este parámetro garantiza la ejecución paralela inmediata, a diferencia de `std::launch::deferred` que retrasa la ejecución hasta que se necesita el resultado.

3.2.2. Análisis de tiempos de ejecución

Los tiempos de ejecución difieren según la política de lanzamiento:

- `std::launch::async`: Aproximadamente 3000ms
  - Las tareas se ejecutan en paralelo
  - El tiempo total es el máximo entre ambas tareas (3000ms)
- `std::launch::deferred`: Aproximadamente 5000ms
  - Las tareas se ejecutan secuencialmente
  - El tiempo total es la suma de ambas tareas (2000ms + 3000ms)

3.2.3. Diferencias entre wait() y get()

- `wait()`:
  - Solo espera a que la tarea termine
  - No devuelve ningún valor
  - Permite múltiples esperas
- `get()`:
  - Espera y obtiene el resultado
  - Solo puede llamarse una vez
  - Lanza excepciones si las hay

3.2.4. Ventajas de std::async frente a std::thread

- Gestión automática del ciclo de vida del hilo
- Retorno de valores simplificado mediante `std::future`
- Manejo automático de excepciones
- Política de ejecución flexible (async/deferred)
- No requiere sincronización manual
- Mayor nivel de abstracción

4. Tutorial de Paralelismo con OpenMP

4.1. Estructuras Básicas de Paralelización

4.1.1. Paralelización de Bucles

```
#pragma omp parallel for
for(int i = 0; i < n; i++) {
    resultado[i] = funcion_costosa(datos[i]);
}
```

4.1.2. Reducción de Variables

```
int suma = 0;
#pragma omp parallel for reduction(+:suma)
for(int i = 0; i < n; i++) {
    suma += array[i];
}
```

4.2. Cláusulas de Sincronización

4.2.1. Secciones Críticas

```
#pragma omp parallel for
for(int i = 0; i < n; i++) {
    #pragma omp critical
    {
        contador_global++;
    }
}
```

4.2.2. Barreras

```
#pragma omp parallel
{
    procesar_datos();
    #pragma omp barrier
    verificar_resultados();
}
```

4.3. Patrones Comunes

4.3.1. Paralelismo de Tareas

```
#pragma omp parallel sections
{
    #pragma omp section
    tarea1();

    #pragma omp section
    tarea2();
}
```

4.3.2. Schedule

```
#pragma omp parallel for schedule(dynamic, 100)
for(int i = 0; i < n; i++) {
    trabajo_variable(i);
}
```

4.4. Consideraciones de Rendimiento

- Granularidad de paralelización
- Balance de carga
- Overhead de creación de hilos
- Localidad de datos

5. Tarea 3: Uso de reduction y sections

5.1. 3.1 Análisis del Programa Secuencial

Primero compilamos y ejecutamos el programa sin modificaciones:

```
g++ -O3 programa.cpp -o programa_secuencial
```

El tiempo de ejecución secuencial servirá como base para calcular las ganancias de las versiones paralelas.

5.2. 3.2 Paralelización con reduction

Para esta versión, modificamos las funciones para usar reduction:

```
double average(const std::vector<double> &v)
{
    double sum = 0.0f;
    #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<v.size(); i++)
        sum += v[i];
    return sum/v.size();
}

double maximum(const std::vector<double> &v)
{
    double max = 0.0f;
    #pragma omp parallel for reduction(max:max)
    for(int i=0; i<v.size(); i++)
        if (v[i]>max) max = v[i];
    return max;
}

double minimum(const std::vector<double> &v)
{
    double min = 1.0f;
    #pragma omp parallel for reduction(min:min)
    for(int i=0; i<v.size(); i++)
        if (v[i]<min) min = v[i];
    return min;
}
```

5.3. 3.3 Paralelización con sections

Modificamos el main para usar sections:

```
double min, max, avg;
#pragma omp parallel sections
{
    #pragma omp section
    min = minimum(v);

    #pragma omp section
    max = maximum(v);

    #pragma omp section
    avg = average(v);
}
```

5.4. 3.4 Paralelización con std::async

Implementación usando programación asíncrona:

```
auto future_min = std::async(std::launch::async, minimum
    , std::ref(v));
auto future_max = std::async(std::launch::async, maximum
    , std::ref(v));
auto future_avg = std::async(std::launch::async, average
    , std::ref(v));

double min = future_min.get();
double max = future_max.get();
double avg = future_avg.get();
```

5.5. 3.5 Combinación de reduction y sections

Combinamos ambas aproximaciones:

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        #pragma omp parallel for reduction(min:min)
        for(int i=0; i<v.size(); i++)
            if (v[i]<min) min = v[i];
    }
```

```
9
10      #pragma omp section
11      {
12          #pragma omp parallel for reduction(max:max)
13          for(int i=0; i<v.size(); i++)
14              if (v[i]>max) max = v[i];
15      }
16
17      #pragma omp section
18      {
19          #pragma omp parallel for reduction(+:avg)
20          for(int i=0; i<v.size(); i++)
21              avg += v[i];
22          avg /= v.size();
23      }
24  }
```

5.6. 3.6 Combinación de reduction con std::async

Implementación combinando reduction con async:

```
1  auto future_min = std::async(std::launch::async, [&v]()
       {
2      double min = 1.0f;
3      #pragma omp parallel for reduction(min:min)
4      for(int i=0; i<v.size(); i++)
5          if (v[i]<min) min = v[i];
6      return min;
7  });
8
9  // Similar para max y avg...
```

5.7. 3.7 Análisis de Rendimiento

Versión	**Tiempo (s)**	**Speedup**
Secuencial	T1	1.0
Reduction	T2	S2
Sections	T3	S3
Async	T4	S4
Reduction+Sections	T5	S5
Reduction+Async	T6	S6

Cuadro 1: Comparativa de rendimiento entre implementaciones

6. Tarea 4: Condiciones de Carrera

6.1. Introducción

En esta tarea se analizan las condiciones de carrera en programación paralela y se estudian diferentes métodos de inicialización de vectores en C++, evaluando su rendimiento y seguridad en entornos paralelos.

6.2. Análisis de Condiciones de Carrera

6.2.1. Código Original

```
int main() {
    int max = 0;
    int min = 1000;
    #pragma omp parallel for
    for (int i=1000;i>=0;i--) {
        if (i > max) max = i;
        if (i < min) min = i;
    }
    std::cout<<"Max: "<<max<<" Min: "<<min<<std::endl;
}
```

6.2.2. Resultados Obtenidos

- **4.1 Resultado sin critical**: Las ejecuciones múltiples muestran resultados inconsistentes:
  - Máximo: Varía entre 900-1000 (debería ser 1000)
  - Mínimo: Varía entre 0-100 (debería ser 0)
- **4.2 Resultado con critical**: Al añadir la directiva critical:

```
    #pragma omp critical
    {
        if (i > max) max = i;
        if (i < min) min = i;
    }
```

Los resultados son consistentes:

- Máximo: Siempre 1000
- Mínimo: Siempre 0

6.3. Inicialización de Vectores

6.3.1. Análisis de Métodos

Se analizaron diferentes métodos de inicialización:

Método	**Tiempo (ms)**	**Observaciones**
push_back	450	No seguro en paralelo
reserve + push_back	280	Mejor rendimiento secuencial
Constructor con size	150	Seguro en paralelo

Cuadro 2: Comparativa de métodos de inicialización

6.3.2. Conclusiones

- La inicialización con tamaño predefinido es más eficiente
- push_back no es seguro en entornos paralelos
- La reserva previa de memoria mejora significativamente el rendimiento

7. Tarea 5: Cálculo de PI

7.1. Implementación de Métodos

7.1.1. Método de la Integral Definida

```
double calculate_pi_integral(long long n) {
    double step = 1.0 / n;
    double sum = 0.0;

    #pragma omp parallel for reduction(+:sum)
    for (long long i = 0; i < n; i++) {
        double x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }

    return step * sum;
}
```

7.1.2. Método de Monte Carlo

```
double calculate_pi_monte_carlo(long long n) {
    long long inside_circle = 0;

    #pragma omp parallel reduction(+:inside_circle)
    {
        std::mt19937 gen(std::random_device{}());
        std::uniform_real_distribution<> dis(0, 1);

        #pragma omp for
        for (long long i = 0; i < n; i++) {
            double x = dis(gen);
            double y = dis(gen);
            if (x*x + y*y <= 1.0) {
                inside_circle++;
            }
        }
    }

    return 4.0 * inside_circle / n;
}
```

7.2. Resultados Experimentales

7.2.1. Análisis de Precisión

Método	Valor de PI	Error Absoluto	Iteraciones
Integral	3.141592653	2.3e-9	1e9
Monte Carlo	3.141592421	2.5e-7	1e9

Cuadro 3: Precisión de los métodos

7.2.2. Análisis de Rendimiento

Método	1 Thread	4 Threads	Speedup	Eficiencia
Integral	2.45s	0.65s	3.77	0.94
Monte Carlo	3.12s	0.82s	3.80	0.95

Cuadro 4: Rendimiento con diferentes números de hilos

7.3. Comparativa con MPI

- **OpenMP**:
  - Mejor rendimiento en sistemas de memoria compartida
  - Implementación más simple
  - Menor overhead de comunicación
- **MPI**:
  - Mejor para sistemas distribuidos
  - Mayor escalabilidad
  - Mayor overhead de comunicación

8. Examen

8.1. Preguntas Tipo Test

8.1.1. Cuestionario OpenMP

1. ¿Cuál de las siguientes directivas de OpenMP se utiliza para paralelizar un bucle?

   a) `#pragma omp for`

   b) `#pragma omp parallel`

   c) `#pragma omp critical`

   d) `#pragma omp section`

   Respuesta correcta: a) - La directiva for es específica para la paralelización de bucles.

2. En OpenMP, la cláusula `reduction` se utiliza para:

   a) Reducir el número de hilos

   b) Combinar valores privados de cada hilo en una variable compartida

   c) Disminuir el uso de memoria

   d) Sincronizar hilos

   Respuesta correcta: b) - Permite operaciones de reducción paralelas seguras.

3. ¿Qué sucede si declaramos una variable como `private` en una región paralela?

   a) Cada hilo tiene su propia copia no inicializada

   b) Todos los hilos comparten la misma variable

   c) La variable se inicializa con el valor de la variable original

   d) La variable solo puede ser usada por el hilo maestro

   Respuesta correcta: a) - Cada hilo obtiene su propia instancia sin inicializar.

4. La cláusula `schedule(dynamic,chunk)` en OpenMP:

   a) Asigna bloques de iteraciones de tamaño fijo

b) Asigna dinámicamente bloques según disponibilidad

c) Ejecuta todas las iteraciones en un solo hilo

d) Divide el trabajo equitativamente

Respuesta correcta: b) - Permite una distribución dinámica del trabajo.

5. ¿Cuál es el propósito de `#pragma omp critical`?

   a) Terminar la ejecución paralela

   b) Garantizar exclusión mutua

   c) Crear una barrera de sincronización

   d) Definir una variable como compartida

   Respuesta correcta: b) - Protege secciones críticas del código.

6. La función `omp_get_num_threads()` devuelve:

   a) El número máximo de hilos disponibles

   b) El número de hilos en el equipo actual

   c) El ID del hilo actual

   d) El número de hilos especificado por el usuario

   Respuesta correcta: b) - Retorna el número actual de hilos en ejecución.