UA

Universidad
de Alicante

ESCUELA POLITÉCNICA SUPERIOR

Análisis de Rendimiento en OpenMP

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

Marzo 2025

ÍNDICE

Introducción

En esta práctica se analizará el rendimiento de programas paralelos usando OpenMP. Se medirá el tiempo de ejecución con distintos números de hilos y se calculará el Speed-up, la eficiencia y la aplicación de la Ley de Amdahl.

Ejercicio 1: Medición de Speed-up y Eficiencia

A continuación mediremos el tiempo de ejecución de una operación paralelizable con distintos números de hilos, y, posteriormente calcularemos el Speed-up y la eficiencia. Recordemos que el Speed-up es el cociente del tiempo que tarda el programa ejecutándose secuencialmente, es decir, cuando trabajamos con un hilo; entre el tiempo que tarda ejecutándose con **N** hilos. Respecto a la Eficiencia, esta será el cociente entre el valor del Speed-up entre el número **N** de hilos utilizados.

Código en OpenMP

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {

    long long N  = 10000000000; // 10 millones
    double suma = 0.0;
    double start, end;

    for (int num_hilos = 1; num_hilos <= 8; num_hilos *= 2) {
        suma = 0.0;
        start = omp_get_wtime();

        #pragma omp parallel for num_threads(num_hilos) reduction(+:suma)
        for (long long i = 0; i < N; i++) {
            suma += (i * 1.5) / (i + 1.0);
        }

        end = omp_get_wtime();

        printf("Resultado final (%i hilos): %2f\n", num_hilos, suma);
        printf("Tiempo: %f segundos\n", end - start);
    }

    return 0;
}
```

Resultados obtenidos

Número hilos	Tiempo (s)	Speed Up	Eficiencia
1	34.48	1.00	1.00
2	14.45	2.39	1.20
4	7.43	4.64	1.16
8	4.35	7.93	0.99

Los resultados muestran que el Speed-up aumenta a medida que se incrementa el número de hilos, lo que indica que la paralelización mejora el rendimiento del programa. Sin embargo, la eficiencia disminuye levemente a medida que se utilizan más hilos, lo que sugiere la presencia de sobrecarga en la gestión de hilos y sincronización.

En particular, la eficiencia **supera el 100%** en el caso de 2 y 4 hilos, lo cual podría atribuirse a una mejor distribución de la carga de trabajo o al uso efectivo del caché. Sin embargo, al emplear 8 hilos, la eficiencia se reduce a 0.99, lo que indica que los beneficios de la paralelización comienzan a disminuir debido a la sobrecarga.

Ejercicio 2 Comparación de Sincronización

En el siguiente ejercicio analizaremos el impacto de diferentes métodos de sincronización en OpenMP sobre el rendimiento de una operación de suma paralelizada. Se compararán tres enfoques: **critical**, **atomic** y **reduction**, midiendo los tiempos de ejecución, Speed-up y eficiencia para distintos números de hilos. El objetivo es determinar cuál de estas técnicas de sincronización es más eficiente en términos de rendimiento y escalabilidad.

Código en OpenMP

```
int main() {
    long long N = 1000000000; // 1.000 millones
    double suma;
    double start, end;

    for (int hilos = 1; hilos <= 8; hilos *= 2) {
        printf("\nEjecutando con %d hilos:\n", hilos);

        for (int i = 0; i < 3; i++) {
            suma = 0.0;
            start = omp_get_wtime();

            // Configuramos el número de hilos
            omp_set_num_threads(hilos);

            #pragma omp parallel
            {
                if (i == 0) {
                    // Critical
                    #pragma omp for
                    for (long long i = 1; i <= N; i++) {
                        double valor = (i * 1.5) / (i + 1.0);
                        #pragma omp critical
                        {
                            suma += valor;
                        }
                    }
                    end = omp_get_wtime();
                    printf("Método Critical: -> Resultado: %.2f, Tiempo: %f segundos\n", suma, end - start);
                }
                else if (i == 1) {
                    // Atomic
                    #pragma omp for
                    for (long long i = 1; i <= N; i++) {
                        double valor = (i * 1.5) / (i + 1.0);
                        #pragma omp atomic
                        suma += valor;
                    }
                    end = omp_get_wtime();
                    printf("Método Atomic: -> Resultado: %.2f, Tiempo: %f segundos\n", suma, end - start);
                }
                else {
                    // Reduction
                    #pragma omp for reduction(+:suma)
                    for (long long i = 1; i <= N; i++) {
                        suma += (i * 1.5) / (i + 1.0);
                    }
                    end = omp_get_wtime();
                    printf("Método Reduction: -> Resultado: %.2f, Tiempo: %f segundos\n", suma, end - start);
                }
            }
        }
    }

    return 0;
}
```

Resultados obtenidos (critical version)

Número hilos	Tiempo (s)	Speed Up	Eficiencia
1	7.71	1	1
2	49.52	7.71/49.52 = 0.16	0.16/2 = 0.08
4	99.07	7.71/99.07 = 0.078	0.078/4 = 0.0195
8	147.00	7.71 / 147.00 = 0.0525	0.0525 / 8 = 0.0066

Resultados obtenidos (atomic version)

Número hilos	Tiempo (s)	Speed Up	Eficiencia
1	6.50	1	1
2	27.56	6.50/27.56 = 0.236	0.236/2 = 0.118
4	53.26	6.50/53.26 = 0.122	0.122/4 = 0.0305
8	68.21	6.50/68.21 = 0.0953	0.0953/8 = 0.0119

Resultados obtenidos (reduction version)

Número hilos	Tiempo (s)	Speed Up	Eficiencia
1	2.80	1	1
2	1.47	2.80 / 1.47 = 1.90	1.90 / 2 = 0.95
4	0.75	2.80 / 0.75 = 3.73	3.73 / 4 = 0.9325
8	0.43	2.80 / 0.43 = 6.51	6.51 / 8 = 0.81375

Análisis y Conclusiones

Los resultados obtenidos muestran diferencias significativas entre los tres enfoques:

- La versión **Critical** muestra una drástica pérdida de rendimiento a medida que aumenta el número de hilos. La contención generada por la sección crítica impacta severamente la escalabilidad, volviéndola ineficiente ya que estamos tratando una simple operación utilizando Critical lo que provoca un cuello de botella.
- La versión **Atomic** mejora en comparación con Critical ya que simplemente bloquea la instrucción específica, no toda la región de código. Pero sigue mostrando una pérdida de rendimiento conforme se incrementan los hilos. Aunque reduce la contención, la sincronización sigue afectando el tiempo de ejecución.
- La versión **Reduction** es claramente la más eficiente. Su tiempo de ejecución disminuye considerablemente con más hilos, logrando una escalabilidad mucho mejor. Esto se debe a que la operación de reducción minimiza la sobrecarga de sincronización al utilizar una memoria local

En conclusión, **la técnica de reducción es la opción más eficiente para la suma paralela en OpenMP**, ya que minimiza la contención y mejora la escalabilidad del programa.

Ejercicio 3 Balanceo de carga y Schedule

En OpenMP, la directiva **#pragma omp for schedule(tipo)** permite definir cómo se distribuyen las iteraciones de un bucle entre los hilos disponibles. En este ejercicio compararemos tres estrategias de planificación: ***Static, Dynamic y Guided***. Cada una tiene un impacto diferente en el balanceo de carga y el rendimiento.

Código en OpenMP

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {

    long long N  = 100000000; // 100 millones
    double start, end;

    for (int num_hilos = 4; num_hilos <= 8; num_hilos *= 2) {
        printf("\nEjecutando con %d hilos:\n", num_hilos);

        for (int k = 0; k < 3; k++) {
            double resultado_final = 0.0;

            if (k==0) {
                start = omp_get_wtime();
                #pragma omp parallel for num_threads(num_hilos) reduction(+:resultado_final) schedule(static)
                for (long long i = 0; i < 16; i++) {
                    double local_result = 0.0;
                    long long carga = N * (i % 4 + 1);
                    for (long long j = 0; j < carga; j++) {
                        local_result += (j * 0.5) / (j + 1.0);
                    }
                    resultado_final += local_result;
                }

                end = omp_get_wtime();
                if (omp_get_thread_num() == 0) {
                    printf("Resultado final (%i hilos): %2f\n", num_hilos, resultado_final);
                    printf("Tiempo: %f segundos\n", end - start);
                }
            } else if (k==1) {

                start = omp_get_wtime();
                #pragma omp parallel for num_threads(num_hilos) reduction(+:resultado_final) schedule(dynamic, 1)
                for (long long i = 0; i < 16; i++) {
                    double local_result = 0.0;
                    long long carga = N * (i % 4 + 1);
                    for (long long j = 0; j < carga; j++) {
                        local_result += (j * 0.5) / (j + 1.0);
                    }
                    resultado_final += local_result;
                }

                end = omp_get_wtime();
                if (omp_get_thread_num() == 0) {
                    printf("Resultado final (%i hilos): %2f\n", num_hilos, resultado_final);
                    printf("Tiempo: %f segundos\n", end - start);
                }
            } else if (k==2) {

                start = omp_get_wtime();
                #pragma omp parallel for num_threads(num_hilos) reduction(+:resultado_final) schedule(guided)
                for (long long i = 0; i < 16; i++) {
                    double local_result = 0.0;
                    long long carga = N * (i % 4 + 1);
                    for (long long j = 0; j < carga; j++) {
                        local_result += (j * 0.5) / (j + 1.0);
                    }
                    resultado_final += local_result;
                }

                end = omp_get_wtime();
                if (omp_get_thread_num() == 0) {
                    printf("Resultado final (%i hilos): %2f\n", num_hilos, resultado_final);
                    printf("Tiempo: %f segundos\n", end - start);
                }
            }
```

Análisis y Conclusiones

Tipo de Schedule	4 Hilos (s)	8 Hilos (s)
Schedule Static	8.433	2.675
Schedule Dynamic	4.212	2.456
Schedule Guided	3.735	2.414

Static divide las iteraciones equitativamente entre los hilos sin redistribuirlas en tiempo de ejecución. Esto funciona bien cuando sabemos que **todas las iteraciones tardan lo mismo**, pero en este caso, donde algunas requieren más trabajo, genera un mal balance. Como resultado, Static es la peor opción, con 8.433 segundos en 4 hilos y 2.675 segundos en 8 hilos, reflejando una baja eficiencia al no aprovechar correctamente los recursos disponibles.

Dynamic mejora el balanceo de carga asignando iteraciones dinámicamente **conforme los hilos terminan** su trabajo. Esto **evita que algunos hilos queden inactivos**, aunque introduce un overhead administrativo. En este experimento, Dynamic reduce significativamente el tiempo en 4 hilos (4.212 s, casi la mitad de Static), pero con 8 hilos el beneficio es menor (2.456 s), indicando que la sobrecarga administrativa afecta el rendimiento en mayores cantidades de hilos.

Guided distribuye inicialmente grandes **bloques de iteraciones**, **reduciendo su tamaño** progresivamente. Esto mejora la eficiencia respecto a Dynamic al disminuir el overhead mientras mantiene un buen balance de carga. En este caso, Guided obtiene los mejores tiempos (3.735 s con 4 hilos y 2.414 s con 8 hilos), mostrando que es la opción más eficiente, combinando un buen rendimiento con una menor sobrecarga en la asignación de tareas.

Ejercicio 4: Análisis de Rendimiento y Ley de Amdahl

1. Supón que el 85% del código es paralelizable (p = 0.85).

2. Calcula el Speed-up máximo teórico (hilos infinitos):

$$S_{max} = 1 / (1 - p)$$

$$S_{max} = 1 / (1 - 0.85) = 6.667$$

3. Calcula el Speed-up teórico para 4 y 8 hilos:

$$S = 1 /((1 - p) + p/N)$$

$$S_4 = 1 /((1 - 0.85) + 0.85/4) = 2,759$$

$$S_8 = 1 /((1 - 0.85) + 0.85/8) = 3,902$$

4. Compara con los Speed-up reales del Ejercicio 1.

$$S_{4(E1)} = 4.64. \; ; \; S_{8(E1)} = 7.93$$

5. Reflexiona sobre la relación entre teoría y práctica.

La Ley de Amdahl asume que la parte paralelizable del código es perfectamente balanceada y que no hay efectos adicionales que mejoren el rendimiento. Sin embargo, en la práctica, **otros factores** como la eficiencia del sistema o la optimización del hardware pueden alterar los resultados.

En este caso, la ejecución real obtuvo **mayor Speed-up que el teórico**, lo que indica que el código paralelo es altamente eficiente y puede estar beneficiándose de efectos como **mejor utilización de la caché, menor latencia de acceso a memoria o una distribución más favorable de la carga de trabajo**.

Por tanto, aunque la Ley de Amdahl es una buena referencia, no siempre refleja el rendimiento real en entornos optimizados.

Ejercicio 5 (Opcional): Escalabilidad Débil

En este ejercicio, evaluaremos la escalabilidad débil del sistema al aumentar tanto el número de hilos como la carga de trabajo de manera proporcional. El objetivo será analizar si el tiempo de ejecución se mantiene **constante conforme crece la cantidad de datos** procesados.

El código que se nos pide implementar incrementa la carga de trabajo en función del número de hilos, asegurando que cada hilo procese 10,000 millones de elementos.

Código en OpenMP

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {

    long long N  = 10000000000; // 10 000 millones
    double start, end;
    double suma = 0.0;

    for (int num_hilos = 1; num_hilos <= 8; num_hilos *= 2) {
        printf("\nEjecutando con %d hilos:\n", num_hilos);
        suma = 0.0;
        start = omp_get_wtime();

        #pragma omp parallel for num_threads(num_hilos) reduction(+:suma)
        for (long long i = 0; i < N*num_hilos; i++) {
            suma += 1;
        }

        end = omp_get_wtime();

        printf("Resultado final (%i hilos): %2f\n", num_hilos, suma);
        printf("Tiempo: %f segundos\n", end - start);
    }
}
```

Análisis y Conclusiones

Número de Hilos	Tiempo (s)
1	34.95
2	29.05
4	29.55
8	32.35

El concepto de **escalabilidad débil** establece que si el sistema es eficiente en la paralelización, el tiempo de ejecución debería permanecer constante al incrementar el número de hilos y la carga proporcionalmente.

En los resultados obtenidos:

- De 1 a 2 hilos, el tiempo disminuye significativamente, lo que sugiere que la paralelización inicial **mejora** el rendimiento.
- De 2 a 4 hilos, el tiempo se mantiene **casi igual** (29.05s vs. 29.55s), lo que indica que la sobrecarga de gestión de hilos comienza a afectar el rendimiento.
- De 4 a 8 hilos, el tiempo aumenta (de 29.55s a 32.35s), lo que sugiere que el beneficio de paralelización se ve reducido debido a la sobrecarga de sincronización o limitaciones del hardware.

En conclusión, los resultados muestran que el sistema **escala bien hasta 4 hilos**, pero a partir de este punto la eficiencia comienza a disminuir. Aunque el tiempo se mantiene relativamente constante hasta 4 hilos, a partir de 8 hilos se observa una degradación del rendimiento, lo que sugiere que la escalabilidad débil no se mantiene completamente y que el sistema enfrenta cuellos de botella en la gestión de recursos.

Santiago Álvarez Geanta Grupo 1

UA Universidad de Alicante

Universidad de Alicante