Universidad del Valle de Guatemala

Facultad de Ingeniería
CC3069 Computación Paralela y Distribuida

Laboratorio 2

Transformación de Quicksort secuencial a paralelo utilizando OpenMP

Por: Samuel A. Chamalé
Email: cha21881@uvg.edu.gt

Fecha de entrega: 29/08/2024

Índice

1. Repositorio **1**

2. Modificaciones Realizadas **1**

2.1. Versión Paralela 1

2.1.1. Razonamiento 1

2.1.2. Código Implementado 1

2.1.3. Resultados 2

2.2. Versión Optimizada Paralela 2

2.2.1. Razonamiento 2

2.2.2. Código Implementado 3

2.2.3. Resultados 3

3. Análisis de Resultados **4**

3.1. Comparación de Speedup 4

3.1.1. Razonamiento 4

3.1.2. Resultados 4

3.1.3. Tabla de Resultados 5

1. Repositorio

https://github.com/chamale-rac/seq2par2

2. Modificaciones Realizadas

Este informe detalla las modificaciones realizadas al programa secuencial para implementar versiones paralelas y optimizadas del algoritmo Quicksort. A continuación, se presentan las modificaciones, su justificación y los resultados obtenidos.

2.1. Versión Paralela

> **Modificación.** Se implementó una versión paralela del algoritmo Quicksort utilizando OpenMP para mejorar el rendimiento en sistemas multi-core.

2.1.1. Razonamiento

El objetivo de paralelizar Quicksort es reducir el tiempo de ejecución dividiendo la tarea de ordenar el array entre múltiples hilos. Esto es especialmente beneficioso para arrays grandes, donde la división y conquista del algoritmo puede ser distribuida eficientemente.

2.1.2. Código Implementado

```cpp
// Implementación de Quicksort Paralelo
void parallelQuickSort(std::vector<int>& arr, int low, int high, int depth) {
    if (high - low < SEQUENTIAL_THRESHOLD || depth > 3) {
        sequentialQuickSort(arr, low, high);
        return;
    }

    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
```

```
    int pi = i + 1;

    #pragma omp task shared(arr) if(depth <= 3)
    parallelQuickSort(arr, low, pi - 1, depth + 1);

    #pragma omp task shared(arr) if(depth <= 3)
    parallelQuickSort(arr, pi + 1, high, depth + 1);

    #pragma omp taskwait
}
```

2.1.3. Resultados

La versión paralela demostró una mejora significativa en comparación con la versión secuencial, especialmente para tamaños de entrada grandes.

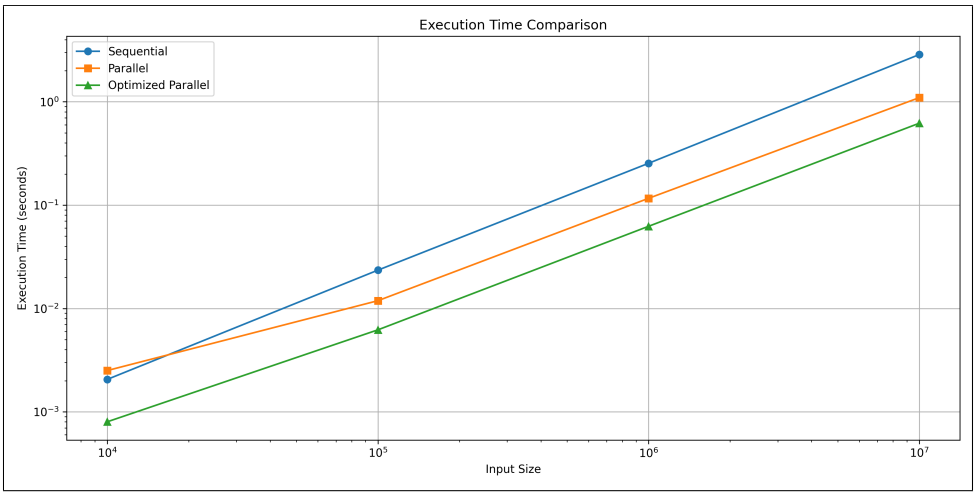

Figura 1: Comparación de Tiempos de Ejecución

2.2. Versión Optimizada Paralela

Modificación. Se implementó una versión optimizada de Quicksort paralela con mejor sincronización y un manejo más eficiente de los recursos del sistema.

2.2.1. Razonamiento

La optimización incluye la reducción de la sobrecarga de sincronización y un manejo más eficiente del trabajo distribuido entre los hilos. Esto se logra ajustando el número de tareas activas según la profundidad de la recursión.

2.2.2. Código Implementado

```
// Implementación de Quicksort Paralelo Optimizado
void optimizedParallelQuickSort(std::vector<int>& arr, int left, int right) {
    if (right - left <= SMALL_ARRAY_THRESHOLD) {
        sequentialQuickSort(arr, left, right);
    } else if (left < right) {
        int pivot = arr[right];
        int i = left - 1;

        for (int j = left; j < right; j++) {
            if (arr[j] <= pivot) {
                i++;
                std::swap(arr[i], arr[j]);
            }
        }
        std::swap(arr[i + 1], arr[right]);

        int partition = i + 1;

        #pragma omp task shared(arr)
        optimizedParallelQuickSort(arr, left, partition - 1);

        #pragma omp task shared(arr)
        optimizedParallelQuickSort(arr, partition + 1, right);

        #pragma omp taskwait
    }
}
```

2.2.3. Resultados

La versión optimizada logró una mayor eficiencia y mejoró el tiempo de ejecución en comparación con la versión paralela estándar.

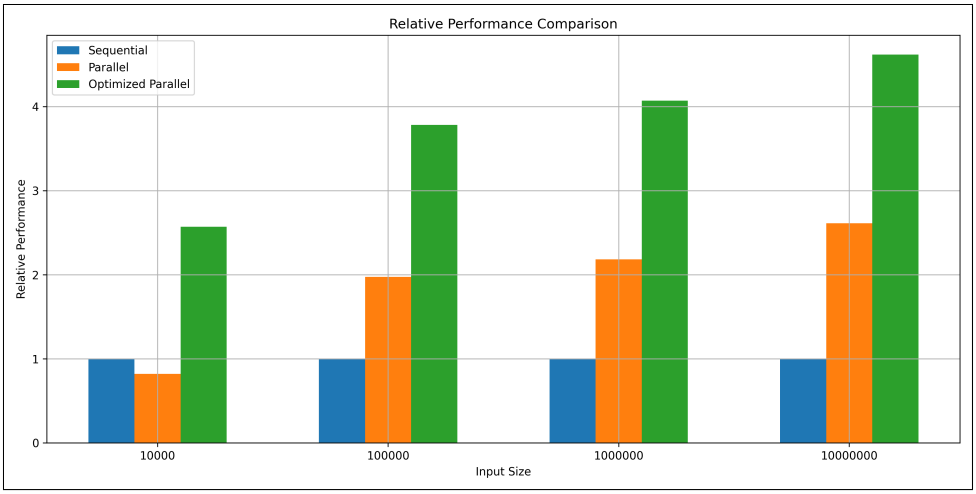

Figura 2: Comparación de Rendimiento Relativo

3. Análisis de Resultados

3.1. Comparación de Speedup

> **Modificación.** Se midió el speedup de las versiones paralelas y optimizadas comparado con la versión secuencial.

3.1.1. Razonamiento

El speedup permite evaluar la eficiencia del paralelismo en diferentes tamaños de entrada y configura la escalabilidad del algoritmo.

3.1.2. Resultados

La versión optimizada presentó un speedup superior al de la versión paralela básica, especialmente con tamaños de entrada mayores.

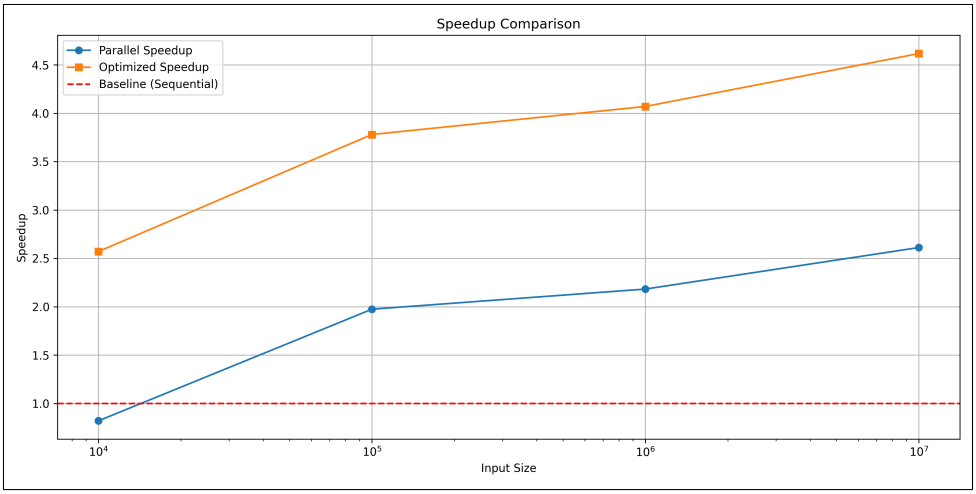

Figura 3: Comparación de Speedup

3.1.3. Tabla de Resultados

Tamaño de Entrada	Tiempo Secuencial (s)	Tiempo Paralelo (s)	Tiempo Optimizado (s)	Speedup Paralelo	Speedup Optimizado
10,000	0.0021	0.0025	0.0008	0.82	2.57
100,000	0.0235	0.0119	0.0062	1.97	3.78
1,000,000	0.2540	0.1164	0.0624	2.18	4.07
10,000,000	2.8642	1.0965	0.6203	2.61	4.62

Cuadro 1: Resultados de Tiempos de Ejecución y Speedup

Referencias

[Ope] *Using OpenMP with C — Research Computing, University of Colorado Boulder documentation*. `https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html`. Accessed: 2024-08-11. n.d.

Índice de figuras

1. Comparación de Tiempos de Ejecución 2
2. Comparación de Rendimiento Relativo 4
3. Comparación de Speedup 5