UNIVERSIDAD DE CARABOBO

FACULTAD EXPERIMENTAL DE CIENCIAS Y TECNOLOGÍA

DEPARTAMENTO DE COMPUTACIÓN

PROYECTO

Análisis de Accesos a Memoria en Algoritmos de Ordenamiento

Integrantes:

Rolando Suarez C.I: 30.445.947
Jesus Muñoz C.I: 27.188.137

Fecha: Valencia, 16 de octubre del 2025

1. Introducción

1.1. Contexto y Motivación

En computación de alto rendimiento, la optimización de algoritmos debe considerar la jerarquía de memoria. La memoria caché, siendo la más rápida y cercana al procesador, es crucial para el rendimiento. Este estudio analiza cómo diferentes algoritmos de ordenamiento interactúan con la memoria caché y cómo pueden optimizarse.

1.2. Objetivos

El proyecto consiste en implementar un simulador de memoria caché configurable, con el objetivo de analizar el comportamiento de acceso a memoria de distintos algoritmos de ordenamiento. A partir de esta evaluación, se desarrollarán versiones optimizadas que aprovechen los principios de localidad, permitiendo así comparar el rendimiento entre algoritmos estándar y sus variantes optimizadas. Finalmente, se evaluará el impacto de diferentes configuraciones de caché sobre el desempeño general, proporcionando una visión integral del comportamiento de la memoria en escenarios computacionales diversos.

2. Marco Teórico

2.1. Memoria Caché

2.1.1. Conceptos Fundamentales

La memoria caché es una memoria de pequeño tamaño y alta velocidad que almacena copias de datos frecuentemente accedidos. Su funcionamiento se basa en:

- **Localidad Temporal**: Datos accedidos recientemente tienden a ser accedidos nuevamente
- **Localidad Espacial**: Datos cercanos en memoria tienden a ser accedidos secuencialmente

2.1.2. Parámetros de Configuración

- **Tamaño de Línea**: Bytes transferidos en cada operación
- **Tamaño Total**: Capacidad total de almacenamiento
- **Política de Reemplazo**: Algoritmo para decidir qué línea eliminar

2.1.3. Métricas de Rendimiento

$$\text{Hit Rate} = \frac{\text{Número de hits}}{\text{Total de accesos}} \times 100\,\%$$

$$\text{Miss Rate} = 1 - \text{Hit Rate}$$

2.2. Algoritmo QuickSort

QuickSort es un eficiente algoritmo de ordenamiento que se basa en el paradigma de divide y vencerás. Su funcionamiento comienza con la selección de un elemento del arreglo como pivote, el cual sirve de referencia para reorganizar los demás elementos: aquellos menores al pivote se ubican a su izquierda y los mayores a su derecha. Esta operación de partición divide el problema en dos subarreglos más pequeños, que luego son ordenados de forma recursiva aplicando el mismo procedimiento. Gracias a esta estrategia, QuickSort logra un rendimiento notable en la mayoría de los casos, aprovechando la eficiencia de la recursión y la simplicidad de la partición para ordenar grandes volúmenes de datos.

Complejidad temporal promedio: $O(n \log n)$, peor caso: $O(n^2)$.

2.3. Algoritmo MergeSort

MergeSort garantiza complejidad $O(n \log n)$ en todos los casos:

MergeSort es un algoritmo de ordenamiento que también se basa en el paradigma de divide y vencerás, y se caracteriza por su estabilidad y eficiencia en grandes volúmenes de datos. Su funcionamiento inicia dividiendo el arreglo original en mitades de forma recursiva hasta obtener subarreglos de un solo elemento, que por definición están ordenados. Luego, se procede a la fase de mezcla, en la cual se combinan ordenadamente los subarreglos, comparando elementos y construyendo arreglos más grandes hasta reconstruir el arreglo original completamente ordenado.

2.4. Optimización Basada en Caché

2.4.1. Principio de Blocking

La técnica de blocking divide los datos en bloques que caben en la caché, procesando cada bloque por separado para mejorar la localidad temporal.

2.4.2. Umbral para Insertion Sort

Para subproblemas pequeños, es más eficiente usar Insertion Sort que tiene mejor localidad espacial.

3. Metodología Experimental

3.1. Implementación del Simulador de Caché

Se implementó la clase CacheAnalyzer que simula una memoria caché:

```
class CacheAnalyzer {
  private:
  long long memory_access_count;
  long long cache_misses;
  int cache_line_size;
  int cache_size;
  map<size_t, bool> cache_lines;

  public:
  void memoryAccess(const void* address) {
    memory_access_count++;
```

```
        size_t line_address =
        (reinterpret_cast<size_t>(address) /
        cache_line_size) * cache_line_size;

        if (!cache_lines[line_address]) {
          cache_misses++;
          cache_lines[line_address] = true;

          if (cache_lines.size() >
          static_cast<size_t>(cache_size /
          cache_line_size)) {
            cache_lines.erase(cache_lines.begin());
          }
        }
      }
    };

```

Listing 1: Implementación del Simulador de Caché

3.2. Algoritmos Implementados

3.2.1. QuickSort Estándar

Implementación recursiva clásica con pivote central.

3.2.2. Blocking QuickSort

Versión optimizada que utiliza umbral basado en capacidad de caché:

```
    template<typename Element>
    void blockingQuickSort(vector<Element>& arr, int left,
    int right, CacheAnalyzer& analyzer) {
      if (left >= right) return;

      int cacheCapacity = (analyzer.getCacheSize() /
      analyzer.getCacheLineSize());

      if (right - left < cacheCapacity) {
        insertionSort(arr, left, right, analyzer);
        return;
      }
      // ... resto del algoritmo
    }

```

Listing 2: Blocking QuickSort Optimizado

3.3. Configuración Experimental

3.3.1. Datasets de Prueba

- **Datos Aleatorios**: Distribución uniforme
- **Datos Ordenados**: Secuencia ascendente
- **Datos Inversamente Ordenados**: Secuencia descendente

3.3.2. Tamaños de Dataset

- 1,000 elementos
- 10,000 elementos
- 50,000 elementos

4. Resultados y Análisis

4.1. Resultados por Tamaño de Dataset y Tipo de Datos

Cuadro 1: Resultados para 1,000 elementos (Datos Aleatorios)

Algoritmo	Accesos	Fallos	Hit Rate	Tiempo (μs)
QuickSort	14,565	63	99.57 %	596
Blocking QuickSort	14,144	63	99.55 %	549
MergeSort	28,656	63	99.78 %	854

Cuadro 2: Resultados para 1,000 elementos (Datos Ordenados)

Algoritmo	Accesos	Fallos	Hit Rate	Tiempo (μs)
QuickSort	9,520	63	99.34 %	347
Blocking QuickSort	7,815	63	99.19 %	227
MergeSort	24,996	63	99.75 %	676

Cuadro 3: Resultados para 1,000 elementos (Datos Inversamente Ordenados)

Algoritmo	Accesos	Fallos	Hit Rate	Tiempo (μs)
QuickSort	9,527	63	99.34 %	303
Blocking QuickSort	7,821	63	99.19 %	233
MergeSort	24,884	63	99.75 %	679

Cuadro 4: Resultados para 10,000 elementos (Datos Aleatorios)

Algoritmo	Accesos	Fallos	Hit Rate	Tiempo (μs)
QuickSort	183,770	149,967	18.39 %	16,882
Blocking QuickSort	179,542	146,481	18.41 %	15,451
MergeSort	387,646	47,788	87.67 %	14,987

Cuadro 5: Resultados para 10,000 elementos (Datos Ordenados)

Algoritmo	Accesos	Fallos	Hit Rate	Tiempo (μs)
QuickSort	131,343	99,982	23.88 %	9,715
Blocking QuickSort	116,939	88,476	24.34 %	8,630
MergeSort	336,240	42,042	87.50 %	12,039

Cuadro 6: Resultados para 10,000 elementos (Datos Inversamente Ordenados)

Algoritmo	Accesos	Fallos	Hit Rate	Tiempo (μs)
QuickSort	131,357	102,801	21.74 %	9,849
Blocking QuickSort	116,949	91,286	21.94 %	8,791
MergeSort	331,840	47,042	85.82 %	12,345

Cuadro 7: Resultados para 50,000 elementos (Datos Aleatorios)

Algoritmo	Accesos	Fallos	Hit Rate	Tiempo (μs)
QuickSort	1,124,800	1,085,448	3.50 %	102,169
Blocking QuickSort	1,103,577	1,064,850	3.51 %	99,392
MergeSort	2,286,959	563,375	75.37 %	100,377

Cuadro 8: Resultados para 50,000 elementos (Datos Ordenados)

Algoritmo	Accesos	Fallos	Hit Rate	Tiempo (μs)
QuickSort	782,782	727,532	7.06 %	65,622
Blocking QuickSort	687,725	636,297	7.48 %	57,300
MergeSort	1,970,880	476,565	75.82 %	81,077

Cuadro 9: Resultados para 50,000 elementos (Datos Inversamente Ordenados)

Algoritmo	Accesos	Fallos	Hit Rate	Tiempo (μs)
QuickSort	782,795	749,100	4.30 %	67,268
Blocking QuickSort	687,737	657,859	4.34 %	57,952
MergeSort	1,951,440	501,557	74.30 %	82,880

Tamaños de línea mayores mejoran el hit rate debido a mejor aprovechamiento de la localidad espacial.

5. Discusión

5.1. Efectividad de las Optimizaciones

Los resultados obtenidos permiten evaluar el impacto de la optimización aplicada al algoritmo QuickSort mediante la técnica de *blocking*, la cual busca mejorar el comportamiento de caché aprovechando los principios de localidad espacial y temporal. Esta optimización se refleja en mejoras consistentes en el rendimiento, especialmente en términos de tiempo de ejecución y tasa de aciertos de caché.

Para **datasets pequeños** (1,000 elementos), Blocking QuickSort muestra una ligera mejora en tiempo respecto a QuickSort (549μs vs 596μs), manteniendo tasas de aciertos prácticamente equivalentes (99.55 % vs 99.57 %). Aunque MergeSort presenta una tasa de aciertos superior (99.78 %), su tiempo de ejecución es mayor (854μs), lo que sugiere que el costo computacional de la mezcla supera los beneficios de caché en este tamaño.

En **datasets medianos** (10,000 elementos), la ventaja de Blocking QuickSort se hace más evidente. Frente a QuickSort, reduce el tiempo de ejecución en aproximadamente

8.5 % (15,451μs vs 16,882μs) y mejora ligeramente la tasa de aciertos (18.41 % vs 18.39 %). Aunque MergeSort mantiene una tasa de aciertos significativamente superior (87.67 %), su tiempo es comparable (14,987μs), lo que indica que Blocking QuickSort logra eficiencia competitiva con menor complejidad algorítmica.

En **datasets grandes** (50,000 elementos), Blocking QuickSort conserva su ventaja frente a QuickSort, con una reducción de tiempo cercana al 3 % (99,392μs vs 102,169μs) y una leve mejora en tasa de aciertos (3.51 % vs 3.50 %). Sin embargo, MergeSort domina en eficiencia de caché (75.37 %) y tiempo (100,377μs), lo que sugiere que para volúmenes masivos de datos, la estructura de MergeSort se adapta mejor a patrones de acceso secuencial.

Al analizar los **tipos de datos**, Blocking QuickSort mantiene mejoras consistentes en tiempo y tasa de aciertos frente a QuickSort, tanto en datos ordenados como inversamente ordenados. Por ejemplo, en 10,000 elementos ordenados, Blocking QuickSort mejora el tiempo en más de 11 % y la tasa de aciertos en 0.46 %. En datos inversamente ordenados, la mejora es de 10.7 % en tiempo y 0.20 % en tasa de aciertos. Aunque MergeSort sigue siendo superior en tasa de aciertos en todos los casos, su tiempo no siempre compensa esa ventaja, especialmente en datos aleatorios.

5.2. Comportamiento de los Algoritmos

QuickSort estándar muestra patrón de acceso errático que genera muchos fallos de caché. Blocking QuickSort mitiga este problema mediante:

- **Umbral inteligente**: Cambia a Insertion Sort para subproblemas pequeños
- **Localidad mejorada**: Procesa bloques completos
- **Reducción de overhead**: Minimiza costos de recursión

MergeSort tiene acceso más secuencial pero genera más operaciones de memoria total.

5.3. Implicaciones Prácticas

1. Optimizaciones de caché son más efectivas para datasets grandes
2. Tamaño óptimo de línea depende del algoritmo
3. Elección del algoritmo debe considerar comportamiento con jerarquía de memoria

6. Conclusiones

6.1. Conclusiones Principales

1. Las optimizaciones de caché mediante técnicas como *blocking* ofrecen mejoras moderadas en tiempo de ejecución frente a QuickSort, especialmente en datasets medianos y grandes.
2. Blocking QuickSort presenta mejoras consistentes en tiempo respecto a QuickSort, aunque su tasa de aciertos en caché es similar; MergeSort supera ampliamente en eficiencia de caché, pero no siempre en tiempo.

3. El tamaño del dataset influye significativamente en el comportamiento de los algoritmos, siendo MergeSort más eficiente en caché para volúmenes grandes, mientras que Blocking QuickSort mantiene tiempos competitivos.

4. Los datos ordenados e inversamente ordenados favorecen la localidad, mejorando la tasa de aciertos y reduciendo el tiempo de ejecución en todos los algoritmos, especialmente en Blocking QuickSort.

6.2. Trabajo Futuro

- Implementar políticas de reemplazo más sofisticadas
- Estudiar impacto de la asociatividad de caché
- Extender análisis a algoritmos de ordenamiento externo

Referencias

1. Hennessy, J. L., & Patterson, D. A. Computer Architecture: A Quantitative Approach

2. Cormen, T. H., et al. Introduction to Algorithms

3. Knuth, D. E. The Art of Computer Programming