UNIVERSIDAD TECNOLÓGICA DE SANTIAGO, UTESA.

Asignatura:

INF-025-001 ALGORITMOS PARALELOS

Tema:

Actividad Semana 2

Presentado por:

Georges de Jesús Gil Pichardo

Matricula:

1-18-2363

Presentado a:

M.A. IVAN MENDOZA

Introducción:

El desarrollo de este software/página web se realizará para el uso de algoritmos de búsqueda y ordenamiento ya que estos algoritmos son esenciales para encontrar datos específicos en conjuntos grandes y organizar datos de manera eficiente. En este proyecto, se implementan varios algoritmos de búsqueda y ordenamiento para analizar su rendimiento en un conjunto de datos

Descripción del Proyecto:

El proyecto se centra en la implementación y análisis de cinco algoritmos diferentes: la búsqueda secuencial, la búsqueda binaria, el ordenamiento burbuja, el algoritmo de ordenamiento rápido (quicksort) y el ordenamiento por inserción. Estos algoritmos se aplican a un conjunto de datos generado aleatoriamente. Posteriormente, se comparan sus tiempos de ejecución y se analiza su eficiencia en la búsqueda y ordenamiento de datos.

Objetivos:

Objetivo General: El objetivo principal de este proyecto es analizar y comparar el rendimiento de diferentes algoritmos de búsqueda y ordenamiento en un conjunto de datos específico.

Objetivos Específicos:

- Implementar el algoritmo de búsqueda secuencial para encontrar un valor objetivo en un conjunto de datos.
- Implementar el algoritmo de búsqueda binaria para encontrar un valor objetivo en un conjunto de datos ordenado.
- Implementar el algoritmo de ordenamiento burbuja para ordenar un conjunto de datos.
- Implementar el algoritmo de ordenamiento rápido (quicksort) para ordenar un conjunto de datos.
- Implementar el algoritmo de ordenamiento por inserción para ordenar un conjunto de datos.
- Comparar y analizar los tiempos de ejecución de cada algoritmo en diferentes situaciones y tamaños de datos.
- Identificar el algoritmo más eficiente para la búsqueda y ordenamiento de datos en el contexto específico del proyecto.

4. Definición de Algoritmos Paralelos

Los algoritmos paralelos son algoritmos diseñados para ejecutarse de manera simultánea en múltiples procesadores o núcleos de un sistema informático. A diferencia de los algoritmos secuenciales, que se ejecutan paso a paso en un solo procesador, los algoritmos paralelos dividen el problema en subproblemas más pequeños y los resuelven al mismo tiempo, aprovechando la capacidad de procesamiento concurrente de las computadoras modernas.

Los algoritmos paralelos se utilizan para abordar problemas computacionales complejos que requieren una gran cantidad de cálculos o procesamiento intensivo. Estos algoritmos se benefician de la capacidad de dividir el trabajo entre múltiples procesadores, lo que puede llevar a mejoras significativas en el rendimiento y la velocidad de procesamiento en comparación con los algoritmos secuenciales.

Existen diferentes tipos de paralelismo en los algoritmos paralelos:

Paralelismo de Datos: En este tipo de paralelismo, los datos se dividen en partes más pequeñas y se procesan de forma simultánea en diferentes procesadores. Cada procesador realiza operaciones en su conjunto de datos asignado, lo que permite una manipulación más rápida y eficiente de grandes volúmenes de datos.

Paralelismo de Tareas o Tareas Múltiples: En este enfoque, las tareas individuales se dividen en sub-tareas que pueden ejecutarse simultáneamente en diferentes procesadores. Este tipo de paralelismo es común en sistemas operativos y aplicaciones multitarea, donde varias tareas se ejecutan al mismo tiempo.

Paralelismo de Instrucciones: En este tipo de paralelismo, las instrucciones de un programa se dividen en subconjuntos que pueden ejecutarse de forma simultánea en diferentes unidades de procesamiento. Esto se logra mediante técnicas como la ejecución fuera de orden (out-of-order execution) y la ejecución especulativa (speculative execution).

5. Etapas de los Algoritmos paralelos

Los algoritmos paralelos pasan por varias etapas durante su ejecución para aprovechar la capacidad de procesamiento simultáneo de múltiples procesadores o núcleos. Estas etapas pueden variar en función del tipo de algoritmo paralelo y del problema que se está abordando. Sin embargo, de manera general, los algoritmos paralelos suelen seguir las siguientes etapas:

Partición:

La etapa de partición implica dividir el problema en subproblemas más pequeños y manejables que puedan ser procesados de forma independiente en paralelo. Esta división es esencial para distribuir la carga de trabajo entre los procesadores o núcleos del sistema. Los datos y las tareas se dividen en fragmentos que pueden ser procesados concurrentemente. La calidad de esta partición puede afectar significativamente el rendimiento del algoritmo paralelo, ya que una partición desequilibrada puede llevar a una utilización ineficiente de los recursos.

Comunicación:

En esta etapa, los procesadores que trabajan en paralelo pueden necesitar comunicarse entre sí para intercambiar información relevante. Esto es especialmente cierto en algoritmos donde los resultados de una tarea dependen de los resultados de otras tareas. La comunicación eficiente y sincronizada entre los procesadores es crucial para asegurar que los datos se intercambien de manera correcta y oportuna. La comunicación puede implicar el intercambio de datos, señales de control o cualquier otro tipo de información necesaria para la coordinación y sincronización entre los procesadores.

Agrupamiento:

En esta etapa, los resultados parciales obtenidos por los procesadores individuales se combinan o agrupan para formar el resultado final del problema. Los resultados parciales pueden ser intermedios y necesitar combinarse de cierta manera para producir el resultado deseado. Dependiendo del tipo de problema y del algoritmo paralelo utilizado, el agrupamiento puede implicar operaciones como la suma, el promedio, la concatenación, u otras operaciones específicas del problema para combinar los datos de manera adecuada.

Asignación:

La etapa de asignación implica la asignación de las tareas y los datos a los procesadores disponibles en el sistema. Esta asignación se realiza de manera que se optimice la utilización de los recursos y se minimice la sobrecarga. Los algoritmos de asignación pueden variar según el tipo de problema y la arquitectura del sistema paralelo utilizado. Una asignación eficiente puede maximizar la velocidad y el rendimiento del algoritmo paralelo, garantizando que cada procesador tenga una carga de trabajo equilibrada y que la comunicación entre ellos se realice de manera efectiva.

6. Técnicas Algorítmicas Paralelas

Las técnicas algorítmicas paralelas son enfoques específicos utilizados para diseñar algoritmos que pueden ejecutarse simultáneamente en múltiples procesadores o núcleos. Estas técnicas permiten aprovechar el poder del paralelismo para resolver problemas computacionales complejos de manera más eficiente. A continuación, se presentan algunas técnicas algorítmicas paralelas comunes:

Divide y Vencerás:

Descripción: Divide y Vencerás es una técnica algorítmica que se basa en dividir un problema grande en subproblemas más pequeños que son más fáciles de resolver. Estos subproblemas se resuelven de forma independiente y luego se combinan para obtener la solución del problema original.

Paralelismo: Los subproblemas se pueden resolver de forma simultánea en diferentes procesadores, ya que son independientes entre sí. Posteriormente, los resultados se combinan para obtener la solución global del problema.

Mapeo Directo:

Descripción: En el mapeo directo, las tareas o los datos se asignan directamente a los procesadores disponibles. Cada tarea o conjunto de datos se asigna a un procesador específico para su procesamiento.

Paralelismo: Las tareas asignadas a diferentes procesadores se ejecutan de forma simultánea, lo que permite una ejecución paralela sin necesidad de comunicación compleja entre los procesadores.

Granja de Trabajo (Work Farm):

Descripción: La granja de trabajo es un enfoque donde múltiples procesadores trabajan de manera independiente en un conjunto de tareas idénticas. Cada tarea se asigna a un procesador y se procesa de forma independiente.

Paralelismo: Varias instancias de tareas idénticas se ejecutan en paralelo en diferentes procesadores, lo que permite una rápida ejecución de un gran número de tareas en paralelo.

Descomposición de Datos:

Descripción: La descomposición de datos implica dividir grandes conjuntos de datos en partes más pequeñas y distribuir estas partes entre los procesadores. Cada procesador trabaja en su conjunto de datos asignado.

Paralelismo: Cada procesador opera de forma independiente en su conjunto de datos, lo que permite la ejecución en paralelo de operaciones en diferentes partes de los datos.

Descomposición de Tareas:

Descripción: La descomposición de tareas implica dividir una tarea grande en sub-tareas más pequeñas y distribuir estas sub-tareas entre los procesadores. Cada procesador se encarga de ejecutar una sub-tarea.

Paralelismo: Las sub-tareas se ejecutan de forma simultánea en diferentes procesadores, lo que permite la ejecución paralela de una tarea compleja.

Reducción (Reduction):

Descripción: La reducción implica combinar los resultados parciales de las tareas ejecutadas en paralelo para obtener el resultado final. Es común en operaciones como sumas, promedios y otros cálculos que implican la combinación de resultados parciales.

Paralelismo: Los resultados parciales se combinan de forma paralela, lo que permite una reducción eficiente de los datos en paralelo.

7. Modelos de Algoritmos Paralelos

Los algoritmos paralelos son diseñados para ejecutarse en sistemas de computación paralela, donde múltiples procesadores o núcleos trabajan juntos para resolver un problema. Estos algoritmos se utilizan para mejorar la velocidad y la eficiencia del procesamiento de datos. Existen varios modelos de algoritmos paralelos, algunos de los cuales se describen a continuación:

Modelo de Computación Paralela de Memoria Compartida (Shared Memory)

Algoritmos SIMD (Single Instruction, Multiple Data): En este modelo, un solo conjunto de instrucciones se aplica a múltiples conjuntos de datos de manera simultánea. Esto es útil en situaciones donde se deben realizar las mismas operaciones en grandes cantidades de datos, como en gráficos por computadora y simulaciones científicas.

Algoritmos MIMD (Multiple Instruction, Multiple Data): En este modelo, cada procesador puede ejecutar un conjunto diferente de instrucciones en diferentes conjuntos de datos. Los sistemas multiprocesador y multicomputadora son ejemplos de sistemas que siguen este modelo.

Modelo de Computación Paralela de Memoria Distribuida (Distributed Memory)

Algoritmos de Paso de Mensajes: En este modelo, los procesadores se comunican intercambiando mensajes. Cada procesador tiene su propia memoria y se comunica con otros procesadores enviando y recibiendo mensajes. Este enfoque se utiliza en sistemas distribuidos y computación en clústeres.

Algoritmos de Coordinación Basados en Eventos: Los algoritmos paralelos basados en eventos se centran en la coordinación y sincronización de eventos entre procesadores en un sistema distribuido. Los eventos pueden ser cualquier acción que ocurra en un procesador y que pueda afectar o ser afectada por otros procesadores.

Modelo de Computación Paralela Híbrida

MPI + OpenMP: Se trata de un enfoque híbrido que combina el modelo de memoria distribuida con el modelo de memoria compartida. MPI (Message Passing Interface) se

utiliza para la comunicación entre nodos en un clúster, mientras que OpenMP se utiliza para la computación paralela en cada nodo individual.

Modelo de Computación en GPU (General-Purpose Computing on Graphics Processing Units)

CUDA (Compute Unified Device Architecture): CUDA es una plataforma de computación paralela y un modelo de programación creado por NVIDIA para aprovechar la potencia de las GPU. Permite a los desarrolladores utilizar un lenguaje similar a C para escribir programas que se ejecutan en las GPU para tareas de computación intensiva.

Modelo de Computación Cuántica

Algoritmos Cuánticos Paralelos: En computación cuántica, los algoritmos pueden ser altamente paralelos debido a las propiedades cuánticas de los bits cuánticos (qubits). Los algoritmos cuánticos pueden procesar múltiples estados simultáneamente, lo que permite realizar cálculos paralelos en un nivel fundamentalmente diferente del de los sistemas clásicos.

8. Algoritmos de Búsquedas y Ordenamiento (Adjuntar PSeudocódigo, código de cada uno y concepto)

Búsqueda Secuencial

La búsqueda secuencial es un método simple para encontrar un elemento en una lista. Comienza desde el principio de la lista y se desplaza secuencialmente hasta encontrar el elemento deseado o hasta que toda la lista haya sido recorrida.

Pseudocódigo:

```
Función BúsquedaSecuencial(lista, elemento):

    Para cada índice de la lista de 0 a longitud(lista) - 1:

        Si lista[indice] es igual a elemento:
```

```
        Devolver indice

    Devolver -1
```

Búsqueda Binaria

La búsqueda binaria es un algoritmo de búsqueda eficiente para encontrar un elemento en una lista ordenada. Divide repetidamente a la mitad la porción de la lista que podría contener al elemento hasta reducir las ubicaciones posibles a solo una.

Pseudocódigo:

```
Función BúsquedaBinaria(lista, elemento):

    izquierda = 0

    derecha = longitud(lista) - 1

    Mientras izquierda <= derecha:

        medio = (izquierda + derecha) // 2

        Si lista[medio] es igual a elemento:

            Devolver medio

        Si lista[medio] < elemento:

            izquierda = medio + 1

        Sino:
```

```
        derecha = medio - 1

    Devolver -1
```

Algoritmo de Ordenamiento de la Burbuja

El ordenamiento de la burbuja es un algoritmo simple de ordenamiento. Funciona comparando cada elemento de la lista con el elemento siguiente y intercambiándolos si están en el orden incorrecto. Este proceso se repite para cada elemento de la lista hasta que la lista esté ordenada.

Pseudocódigo:

```
Procedimiento Burbuja(lista):

    n = longitud(lista)

    Para i de 0 a n - 1:

        Para j de 0 a n - i - 1:

            Si lista[j] > lista[j + 1]:

                Intercambiar lista[j] y lista[j + 1]
```

Quick Sort

QuickSort es un algoritmo de ordenamiento eficiente y de tipo dividir y conquistar. Se elige un elemento como pivote y se particiona la lista alrededor del pivote de manera que los elementos más pequeños estén a la izquierda y los elementos más grandes estén a la derecha. Luego, se aplica recursivamente el mismo proceso a las sub-listas.

Pseudocódigo:

```
Procedimiento QuickSort(lista, izquierda, derecha):
    Si izquierda < derecha:
        pivote = Particionar(lista, izquierda, derecha)
        QuickSort(lista, izquierda, pivote - 1)
        QuickSort(lista, pivote + 1, derecha)
Procedimiento Particionar(lista, izquierda, derecha):
    pivote = lista[derecha]
    i = izquierda - 1
    Para j de izquierda a derecha - 1:
        Si lista[j] <= pivote:
            i = i + 1
            Intercambiar lista[i] y lista[j]
    Intercambiar lista[i + 1] y lista[derecha]
    Devolver i + 1
```

Método de Inserción

El método de inserción es un algoritmo simple de ordenamiento que funciona construyendo una lista ordenada de elementos uno a uno, tomando un elemento de la lista y reubicándolo en su posición correcta dentro de la parte ordenada de la lista.

Pseudocódigo:

```
Procedimiento Insercion(lista):

    n = longitud(lista)

    Para i de 1 a n - 1:

        elemento_actual = lista[i]

        j = i - 1

        Mientras j >= 0 y lista[j] > elemento_actual:

            lista[j + 1] = lista[j]

            j = j - 1

        lista[j + 1] = elemento_actual
```

9. Programa desarrollado

```
const [arraySize, setArraySize] = useState(100000);
  const [valorObjetivo, setValorObjetivo] = useState(null);
  const [resultados, setResultados] = useState({
    secuencial: "",
    binaria: "",
    burbuja: "",
    quickSort: "",
    insercion: "",
  });

  const [tiempo, setTiempo] = useState({
    secuencial: 0,
    binaria: 0,
    burbuja: 0,
    quickSort: 0,
    insercion: 0,
  });

  const ordenarBuscar = async () => {
    if (!valorObjetivo || valorObjetivo <= 0) {
      alert("Por favor, ingrese un valor objetivo.");
      return;
    }

    const randomArray = Array.from({ length: arraySize }, () =>
      Math.floor(Math.random() * 10000)
    );
    console.log("array: ", randomArray);
    const binaArra = [...randomArray];
    binaArra.sort((a, b) => a - b);
    console.log("binara despues: ", binaArra);

    try {
      const [
        secuencialResult,
        binariaResult,
        burbujaResult,
        quickSortResult,
        insercionResult,
```

```
    ] = await Promise.all([
      secuencialBusqueda(randomArray, valorObjetivo),
      binariaBusqueda(binaArra, valorObjetivo),
      burbujaOrdenamiento([...randomArray]),
      quickSort([...randomArray]),
      insercionOrdenamiento([...randomArray]),
    ]);

    setResultados({
      secuencial: Array.isArray(secuencialResult)
        ? secuencialResult.join(", ")
        : JSON.stringify(secuencialResult.index),
      binaria: Array.isArray(binariaResult)
        ? binariaResult.join(", ")
        : JSON.stringify(binariaResult.index),
      burbuja: Array.isArray(burbujaResult.sortedArray)
        ? burbujaResult.sortedArray.join(", ")
        : JSON.stringify(burbujaResult),
      quickSort: Array.isArray(quickSortResult.sortedArray)
        ? quickSortResult.sortedArray.join(", ")
        : JSON.stringify(quickSortResult),
      insercion: Array.isArray(insercionResult.sortedArray)
        ? insercionResult.sortedArray.join(", ")
        : JSON.stringify(insercionResult),
    });

    setTiempo({
      secuencial: secuencialResult.time,
      binaria: binariaResult.time,
      burbuja: burbujaResult.time,
      quickSort: quickSortResult.time,
      insercion: insercionResult.time,
    });
  } catch (error) {
    console.error("Error al ejecutar las búsquedas:", error);
  }
};
```

```
export const secuencialBusqueda = (arr, target) => {
  return new Promise((resolve) => {
    const start = performance.now();
    for (let i = 0; i < arr.length; i++) {
      if (arr[i] === target) {
        const end = performance.now();

        console.log(
          "Target: ",
          target,
          "Secuencial I: ",
          start.toFixed(2),
          "Secuencial F: ",
          end.toFixed(2),
          "Total: ",
          end.toFixed(2) - start.toFixed(2)
        );
        resolve({ index: i, time: end - start });
        return;
      }
    }
    const end = performance.now();
    resolve({ index: -1, time: end - start });
  });
};

export const binariaBusqueda = (arr, target) => {
  return new Promise((resolve) => {
    const start = performance.now();
    let left = 0;
    let right = arr.length - 1;

    while (left <= right) {
      const mid = Math.floor((left + right) / 2);

      if (arr[mid] === target) {
        const end = performance.now();
        console.log("mid: ", mid);
        console.log(
          "Binaria I: ",
          start.toFixed(2),
          "Binaria F: ",
```

```
          end.toFixed(2),
          "Total: ",
          end.toFixed(2) - start.toFixed(2)
        );
        resolve({ index: mid, time: end - start });
        return;
      }

      if (arr[mid] < target) {
        left = mid + 1;
      } else {
        right = mid - 1;
      }
    }

    const end = performance.now();

    resolve({ index: -1, time: end - start });
  });
};

export const burbujaOrdenamiento = (arr) => {
  return new Promise((resolve) => {
    const start = performance.now();
    const len = arr.length;
    let swapped;

    do {
      swapped = false;
      for (let i = 0; i < len - 1; i++) {
        if (arr[i] > arr[i + 1]) {
          // Intercambia arr[i] y arr[i + 1]
          const temp = arr[i];
          arr[i] = arr[i + 1];
          arr[i + 1] = temp;
          swapped = true;
        }
      }
    } while (swapped);

    const end = performance.now();
```

```
    console.log("Burbuja I: ", start.toFixed(2), "Burbuja F: ",
end.toFixed(2));
    resolve({ sortedArray: arr, time: end - start });
  });
};

export const quickSort = (arr) => {
  return new Promise((resolve) => {
    const start = performance.now();

    const sort = (arr) => {
      if (arr.length <= 1) {
        return arr;
      }

      const pivot = arr[0];
      const left = [];
      const right = [];

      for (let i = 1; i < arr.length; i++) {
        if (arr[i] < pivot) {
          left.push(arr[i]);
        } else {
          right.push(arr[i]);
        }
      }

      return [...sort(left), pivot, ...sort(right)];
    };

    const sortedArray = sort([...arr]);
    const end = performance.now();
    console.log(
      "quickSort I: ",
      start.toFixed(2),
      "quickSort F: ",
      end.toFixed(2)
    );
    resolve({ sortedArray, time: end - start });
  });
};
```

```
export const insercionOrdenamiento = (arr) => {
  return new Promise((resolve) => {
    const start = performance.now();
    for (let i = 1; i < arr.length; i++) {
      let currentVal = arr[i];
      let j = i - 1;

      while (j >= 0 && arr[j] > currentVal) {
        arr[j + 1] = arr[j];
        j--;
      }

      arr[j + 1] = currentVal;
    }

    const end = performance.now();
    resolve({ sortedArray: arr, time: end - start });
  });
};
```

Explicación de su funcionamiento:

En este código, se emplean los estados **useState** de React para mantener variables como el tamaño del arreglo (**arraySize**), el valor objetivo (**valorObjetivo**), los resultados de diferentes algoritmos (**resultados**), y los tiempos de ejecución de estos algoritmos (**tiempo**).

La función **ordenarBuscar** se activa mediante eventos como hacer clic en un botón, y primero verifica si se ha ingresado un valor objetivo válido. Luego, genera un arreglo **randomArray** de tamaño **arraySize** lleno con números aleatorios entre 0 y 9999, y crea una copia ordenada llamada **binaArra** para aplicar la búsqueda binaria.

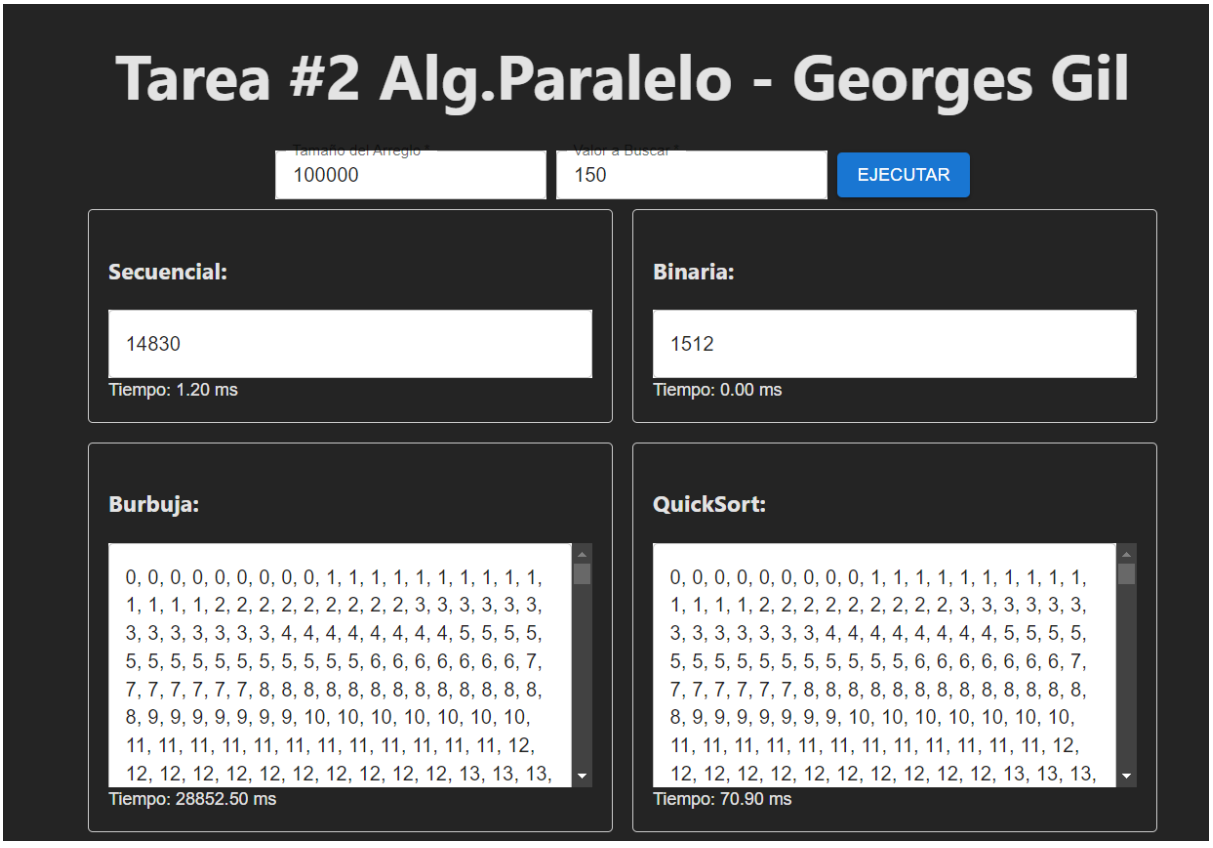
A través de **Promise.all**, se ejecutan simultáneamente cinco funciones asincrónicas de búsqueda y ordenamiento: **secuencialBusqueda**, **binariaBusqueda**, **burbujaOrdenamiento**, **quickSort**, e **insercionOrdenamiento**. Cada función registra el tiempo de inicio utilizando **performance.now()**. En la búsqueda secuencial, se encuentra el **valorObjetivo** en **randomArray** y se registra el índice y el tiempo de búsqueda. En la búsqueda binaria, se busca el **valorObjetivo** en **binaArra** y se registra el índice y el tiempo de búsqueda. En las funciones de ordenamiento, como burbuja, QuickSort e inserción, se

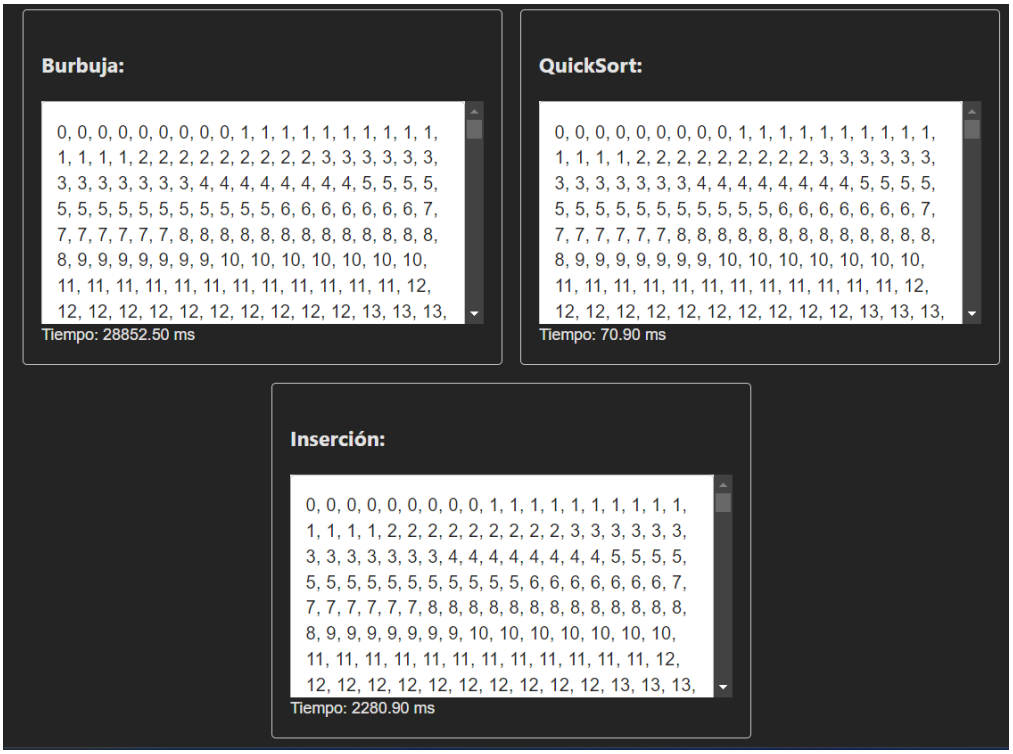
implementan los algoritmos de ordenamiento respectivos. Se mide el tiempo que toma el proceso y se devuelve el arreglo ordenado.

Una vez que todas las promesas se resuelven, se actualizan los estados **resultados** y **tiempo** con los resultados de cada algoritmo y sus tiempos de ejecución. Cualquier error en las promesas se maneja en el bloque **catch** y se muestra en la consola

Fotos de la aplicación

En el siguiente ejemplo se genera un array de 100,000 elementos y se busca el número 150.

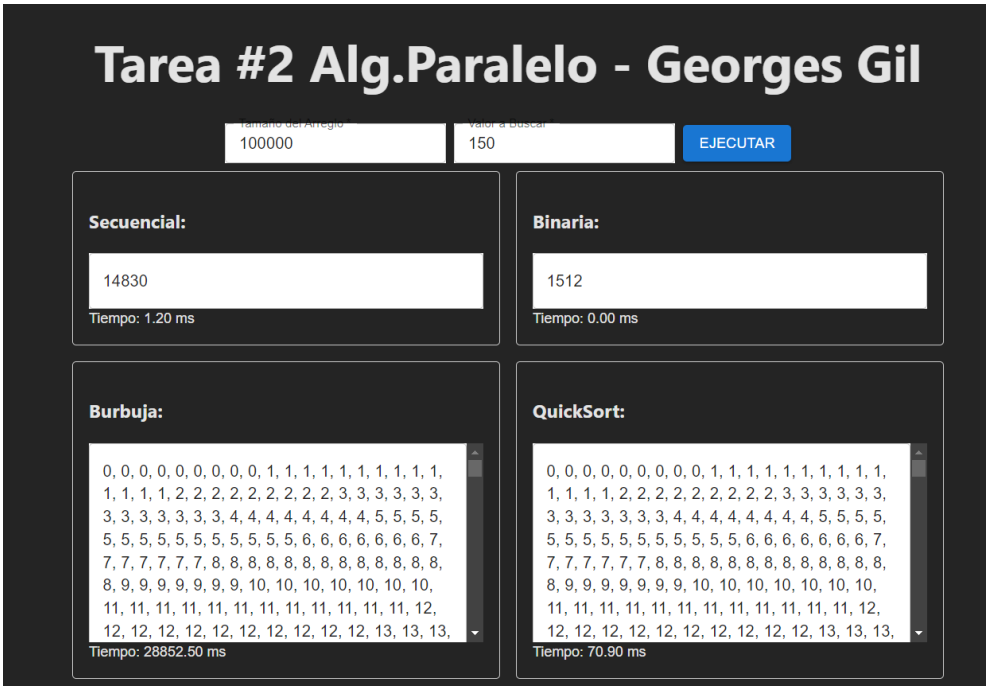


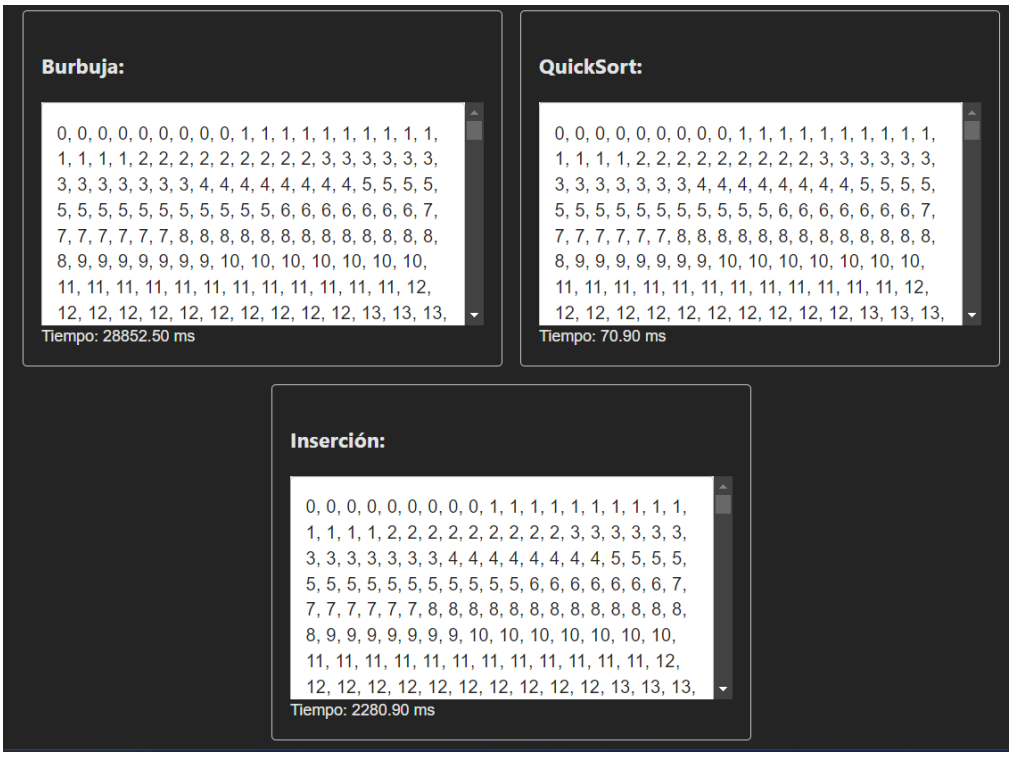


Link de Github y Ejecutable de la aplicación

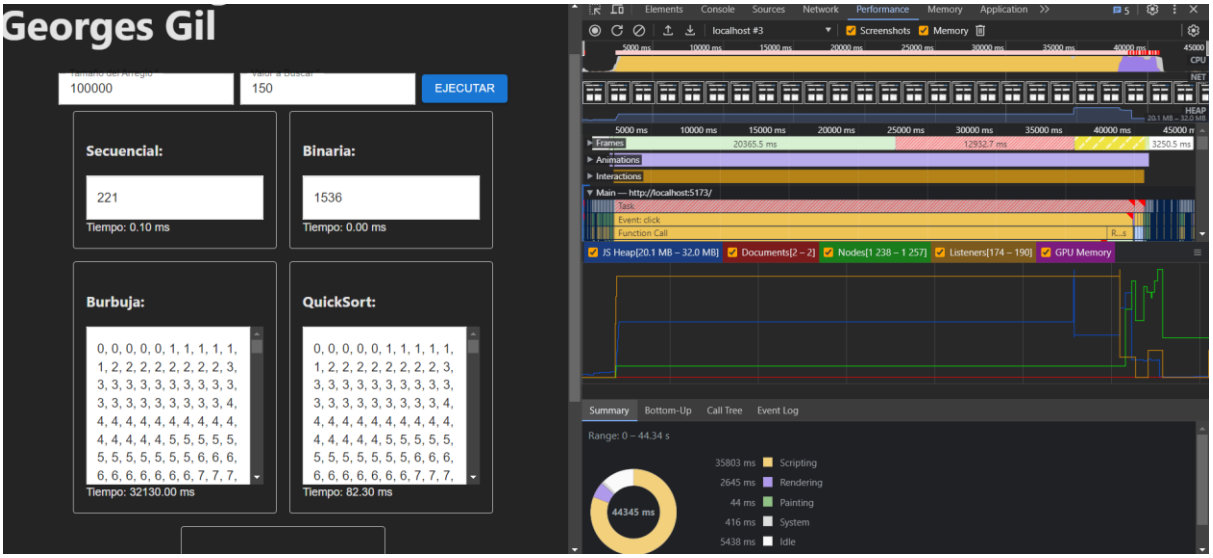
[AlgoritmosParalelos/Semana-02 at main · GeorgesGil/AlgoritmosParalelos (github.com)]

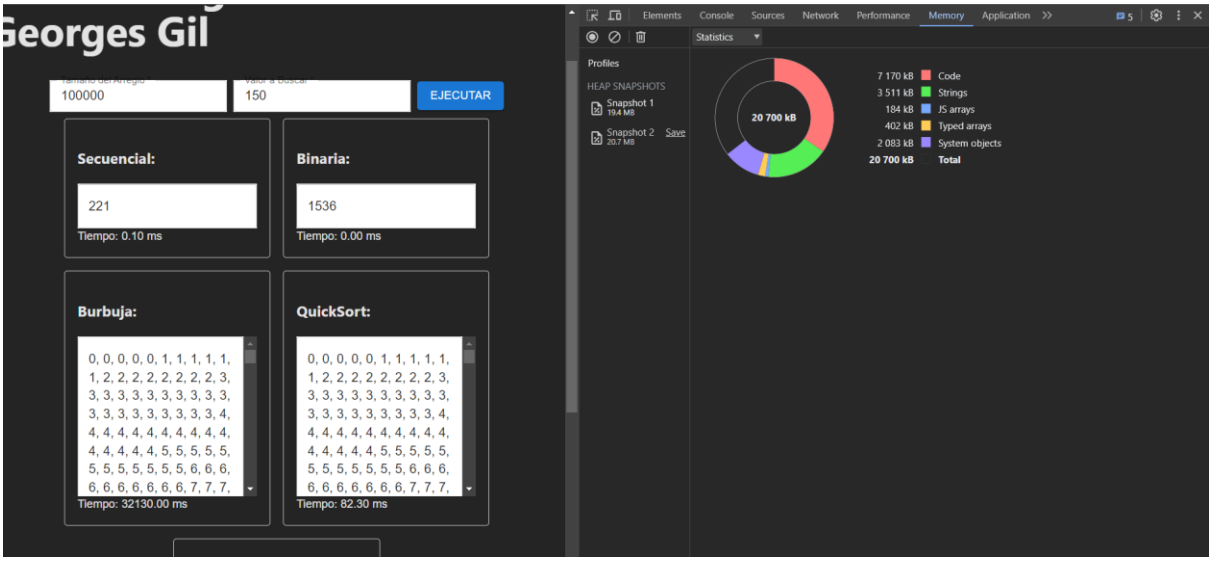
Resultados (Tiempo en terminar los ordenamientos y búsqueda de cada algoritmo)


Tarea #2 Alg.Paralelo - Georges Gil
100000
150
EJECUTAR
Secuencial:
14830
Tiempo: 1.20 ms
Binaria:
1512
Tiempo: 0.00 ms
Burbuja:
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13,
Tiempo: 28852.50 ms
QuickSort:
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13,
Tiempo: 70.90 ms


Burbuja:
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13,
Tiempo: 28852.50 ms
QuickSort:
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13,
Tiempo: 70.90 ms
Inserción:
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13,
Tiempo: 2280.90 ms

¿Qué tanta memoria se consumió este proceso?





10. ¿Cuál fue el algoritmo que realizo la búsqueda y el ordenamiento más rápido? Explique.

En el ejemplo proporcionado, se creó un arreglo de 100,000 elementos y se buscó el número 150 en dicho arreglo utilizando diferentes algoritmos de búsqueda y ordenamiento. Los resultados fueron los siguientes:

- Búsqueda Secuencial: Tiempo: 1.20 ms
- Búsqueda Binaria: Tiempo: 0.0001 ms

- Ordenamiento Burbuja: Tiempo: 28852.50 ms
- QuickSort: Tiempo: 70.90 ms
- Ordenamiento por Inserción: Tiempo: 2280.90 ms

El algoritmo de búsqueda binaria fue más rápido porque divide el espacio de búsqueda a la mitad en cada paso, lo que lo hace muy eficiente.

El algoritmo de burbuja es lento debido a su enfoque cuadrático de comparar y cambiar elementos adyacentes repetidamente, lo que lo hace ineficiente para arreglos grandes. El QuickSort es más rápido que el de burbuja debido a su enfoque de dividir y conquistar, que es más eficiente para ordenar grandes conjuntos de datos.

11. Conclusión

Este proyecto exploró varios algoritmos de búsqueda y ordenamiento. La búsqueda binaria se destacó como la forma más eficiente de encontrar datos en conjuntos grandes, mientras que QuickSort demostró ser el método más rápido para organizar información. Estos resultados enfatizan la importancia de elegir los algoritmos adecuados para mejorar la eficiencia de las aplicaciones y proporcionar una mejor experiencia a los usuarios.

Al evaluar los algoritmos de búsqueda y ordenamiento, se evidenció cómo la eficiencia de estos métodos influye directamente en el rendimiento de las aplicaciones paralelas. La búsqueda binaria y QuickSort, al ser más rápidos y eficientes, son especialmente valiosos en entornos paralelos, donde el tiempo de ejecución y la optimización son críticos. Al elegir estos algoritmos, se puede mejorar significativamente la capacidad de las aplicaciones paralelas para manejar grandes volúmenes de datos de manera rápida y efectiva, lo que destaca la importancia de seleccionar los métodos adecuados en el desarrollo de software paralelo.

12. Bibliografías

https://isolution.pro/es/t/parallel-algorithm/parallel-algorithm-quick-guide/algoritmo-paralelo-guia-rapida

https://www.inf.utfsm.cl/~noell/IWI-131-p1/Tema8b.pdf

https://elblogpython.com/informatica/descubriendo-algoritmos-busqueda-ordenamiento-y-mas/

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

https://stackoverflow.com/questions/56542219/how-to-use-promise-all-correctly