UNIVERSIDAD TECNOLOGICA DE SANTIAGO

UTESA

ALGORITMOS PARALELOS – INF-025-001

Tarea Semana 2.

Presentado Por:

Benjamin Tavarez

1-19-2141

Presentado a:

Ing. Iván Mendoza

04 de febrero, 2024

Santiago de los caballeros,

República Dominicana

Índice de contenido

1. Introducción

En el ámbito de la ciencia de la computación, la optimización de algoritmos de búsqueda y ordenamiento es esencial para mejorar la eficiencia en el procesamiento de datos. Este proyecto se centra en la evaluación comparativa de diversos algoritmos, incluyendo la Búsqueda Secuencial, la Búsqueda Binaria, el Ordenamiento Burbuja, el Quicksort y el Método de Inserción. El propósito fundamental es determinar cuál de estos algoritmos destaca como el más veloz cuando se enfrenta a la tarea de manipular datos en un entorno controlado.

2. Descripción del Proyecto

Este proyecto, involucra la implementación de un conjunto de pruebas en las que cada algoritmo será sometido a una serie de escenarios utilizando un conjunto de 10,000 números enteros generados de manera aleatoria en un intervalo de 1 a 200,000. Estas pruebas se llevarán a cabo en un entorno controlado, garantizando condiciones homogéneas para cada algoritmo y permitiendo una evaluación precisa de sus capacidades individuales.

Previa a la presentación de los resultados, se desarrollará un Marco Teórico que presentara una explicación detallada de cada algoritmo, proporcionando un contexto teórico para la comprensión de los principios fundamentales que rigen cada método.

3. Objetivos

a) Objetivo General

Evaluar y determinar cuál de los algoritmos de búsqueda y ordenamiento, descritos en este documento, exhibe el mejor desempeño en términos de velocidad al manipular un conjunto grande de números, ejecutándolos de manera simultánea.

b) Objetivos Específicos

- Analizar el consumo de recursos asociados a los algoritmos de búsqueda y ordenamiento, tales como Búsqueda Secuencial, Búsqueda Binaria, Ordenamiento Burbuja, Quick Sort y Método de Inserción.
- Medir el tiempo de ejecución individual de cada algoritmo de búsqueda y ordenamiento, incluyendo Búsqueda Secuencial, Búsqueda Binaria, Ordenamiento Burbuja, Quick Sort y Método de Inserción.
- Observar y documentar los resultados obtenidos por cada algoritmo de búsqueda y ordenamiento, detallando su rendimiento y eficacia en la manipulación del conjunto de datos.

4. Definición de Algoritmos Paralelos

A diferencia de los algoritmos convencionales, los algoritmos paralelos tienen la capacidad de realizar múltiples operaciones simultáneamente. Esto implica que pueden ser ejecutados por partes y, de manera concurrente, por diferentes unidades de procesamiento, para luego combinar todas las partes y obtener un resultado preciso. La característica de ser paralelizable o no paralelizable resalta la capacidad de algunos problemas para ser divididos y abordados de manera simultánea, mientras que otros no lo permiten.

La importancia de los algoritmos paralelos radica en su capacidad para abordar eficientemente tareas de computación voluminosas al realizarlas en paralelo, en lugar de seguir una secuencia lineal. Este enfoque mejora significativamente la velocidad y la eficiencia en el manejo de grandes volúmenes de datos y operaciones computacionales.

5. Etapas de los Algoritmos paralelos

a) Partición

Se descomponen los datos y el computo sobre los que opera el algoritmo. Se ignoran aspectos como numero de procesadores que posee la maquina a utilizar y se concentra en explotar oportunidades de paralelismo.

b) Comunicación

Se determina la comunicación necesaria que pueda ocurrir, incluye intercambio de datos, mensajes o resultados entre procesadores. Se definen estructuras y algoritmos de comunicación.

- Primero, se definen los canales que conecten las tareas que requerirán datos.
- Segundo, se especifica la información o mensajes a ser enviados y recibidos.

c) Agrupamiento

Se evalúan los resultados de las etapas anteriores tanto en eficiencia como en costo y se determina si es necesario agrupar o combinar para obtener un resultado total o parcial para seguir utilizándolo en las siguientes etapas del algoritmo, en esencia, pretende reducir la cantidad de datos a enviar. Encontramos: suma, concatenación o fusión de datos.

d) Asignación

Cada tarea se asigna a un procesador, tratando de utilizarlos al máximo y reducir el costo de comunicación. Puede ser estática (antes de la ejecución) o en tiempo de ejecución (mediante algún algoritmo de balanceo de carga).

6. Técnicas Algorítmicas Paralelas

Existen diversas técnicas que permiten aprovechar las capacidades de computación de sistemas con múltiples núcleos, multiprocesadores o clústeres. Entre ellas, destacan:

- **Divide y Conquista Paralela:**

  Consiste en dividir el problema principal en subproblemas independientes que pueden resolverse en paralelo. Luego, se combinan los resultados para obtener la solución del problema original.

- **Paralelismo de Datos:**

  Implica dividir los datos en partes iguales y procesarlos simultáneamente en múltiples núcleos o procesadores. Cada núcleo realiza operaciones distintas con los datos.

- **Paralelismo de Tareas:**

  Consiste en dividir las tareas en subconjuntos independientes, asignando cada subconjunto a un núcleo o procesador para su ejecución simultánea.

- **Comunicación Paralela:**

  Se refiere a modelos de programación que facilitan el desarrollo de algoritmos paralelos, como el MPI (Modelo de Paso de Mensajes), OpenMP, CUDA, entre otros. Estos modelos simplifican la escritura de código paralelo.

- **Balanceo de Carga:**

  Implica distribuir de manera eficiente las tareas entre los núcleos o procesadores en sistemas paralelos para aprovechar al máximo los recursos disponibles. El balanceo de carga es crucial para una ejecución eficiente.

7. Modelos de Algoritmos Paralelos

Modelo de PRC (Parallel Random-Acces Machine)

En este se asume que varios procesadores pueden acceder de forma simultanea a la memoria principal y realizar las operaciones en un tiempo estimado.

Modelo de PRAM (Parallel Ramdom-Access Machine)

Es una extensión del anterior. Aquí muchos procesadores acceden a una memoria compartida utilizando operaciones de lectura y escritura en tiempo constante.

Modelo de BSP (Bulk Synchronous Parallel)

En este dividimos los algoritmos en etapas o superpasos. Por cada etapa los procesadores realizan unos cálculos de forma local y luego se sincronizan al final para intercambiar datos.

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

a) Búsqueda Secuencial

La búsqueda secuencial es un método simple que recorre un conjunto de datos de manera secuencial para encontrar un elemento específico o por el contrario que llegue al final. Se utiliza si el vector no esta en orden o no se puede ordenar previo a la búsqueda.

- **Pseudocódigo**

```
Función BusquedaSecuencial(lista, elemento):
    Para cada índice i desde 0 hasta el tamaño de la lista - 1:
        Si lista[i] es igual al elemento:
            Retornar i
    Retornar -1
```

- **Código**

```
private static int BuscarSecuencial(int[] lista, int elemento)
 {
     for (int i = 0; i < lista.Length; i++)
     {
         if (lista[i] == elemento)
             return i;
     }
     return -1;
 }
```

b) Búsqueda Binaria

La Búsqueda Binaria es una técnica eficiente para localizar elementos en vectores ordenados. Este compara el elemento a buscar con un valor central. Si el valor encontrado es mayor o menor, se itera la búsqueda en el segmento de la izquierda o derecha respectivamente hasta que este no pueda ser dividió o se encuentre el elemento.

- **Pseudocódigo**

```
Función BusquedaBinaria(lista, elemento):
    Inicializar inicio a 0 y fin al tamaño de la lista - 1

    Mientras inicio sea menor o igual a fin:
        Calcular medio como (inicio + fin)
        Si lista[medio] es igual al elemento:
            Retornar medio
        Si lista[medio] es menor que el elemento:
            Establecer inicio como medio + 1
```

Sino:
Establecer fin como medio - 1

Retornar -1

- **Código**

```
private static int BusquedaBinaria(int[] lista, int elemento)
 {
     int inicio = 0;
     int fin = lista.Length - 1;

     while (inicio <= fin)
     {
         int medio = (inicio + fin) / 2;

         if (lista[medio] == elemento)
             return medio;

         if (lista[medio] < elemento)
             inicio = medio + 1;
         else
             fin = medio - 1;
     }

     return -1;
 }
```

c) Algoritmo de Ordenamiento de la Burbuja

Es un algoritmo de ordenamiento sencillo, aunque no es el mas eficiente. Funciona tomando los 2 primeros elementos y, si no están en orden, les intercambiamos lugar, el proceso se va a repetir hasta llegar al final. El proceso se acorta al llegar al final porque se va acortando el proceso debido a que los números ya ordenados no se van evaluando.

- **Pseudocódigo**

Procedimiento Burbuja(lista):

Para i desde 0 hasta el tamaño de la lista - 1:

Para j desde 0 hasta el tamaño de la lista - 1 - i:

Si lista[j] es mayor que lista[j + 1]:

Auxiliar = lista[j]

lista[j] = lista[j+1]

lista[j+1] = Auxiliar

- **Código**

```
private static void OrdenamientoBurbuja(int[] lista)
        {
            for (int i = 0; i < lista.Length; i++)
            {
                for (int j = 0; j < lista.Length - 1 - i; j++)
                {
                    if (lista[j] > lista[j + 1])
                    {
                        int temp = lista[j];
                        lista[j] = lista[j + 1];
                        lista[j + 1] = temp;
                    }
                }
            }
        }
```

d) Quick Sort

Quick Sort es el algoritmo de ordenación más rápido conocido, destacando por realizar menos operaciones gracias al método de partición. Este algoritmo se destaca por su eficiencia, pero la elección adecuada del pivote es una consideración crítica para optimizar su desempeño.

En este algoritmo, se selecciona un elemento de la lista, denominado pivote, y se procede a particionar la lista en tres subconjuntos. La primera sublista contiene todos los elementos menores que el pivote (sin incluirlo), la segunda contiene únicamente el pivote y la tercera incluye todos los elementos mayores. Luego, se aplica recursión a las primeras y terceras sublistas, y los resultados se unen concatenando las tres listas.

- **Pseudocódigo**

```
Procedimiento QuickSort(lista, inicio, fin):
  Si inicio < fin:
    pivote = Particionar(lista, inicio, fin)
    QuickSort(lista, inicio, pivote - 1)
    QuickSort(lista, pivote + 1, fin)

Función Particionar(lista, inicio, fin):
  pivote = lista[fin]
  i = inicio - 1

  Para j desde inicio hasta fin - 1:
    Si lista[j] <= pivote:
      i = i + 1
      Intercambiar(lista[i], lista[j])

  Intercambiar(lista[i + 1], lista[fin])
  Retornar i + 1
```

- **Código**

```
  private static void QuickSort(int[] lista, int inicio, int fin)
        {
            if (inicio < fin)
            {
                int pivote = PartirQuiksort(lista, inicio, fin);
                QuickSort(lista, inicio, pivote - 1);
                QuickSort(lista, pivote + 1, fin);
            }
        }

        // Método auxiliar para realizar la partición del arreglo
        private static int PartirQuiksort(int[] lista, int inicio, int
fin)
        {
            int pivote = lista[fin];
            int i = inicio - 1;

            for (int j = inicio; j < fin; j++)
            {
                if (lista[j] <= pivote)
                {
                    i++;
                    Intercambiar(ref lista[i], ref lista[j]);
                }
            }

            Intercambiar(ref lista[i + 1], ref lista[fin]);
            return i + 1;
        }

        private static void Intercambiar(ref int a, ref int b)
        {
            int temp = a;
            a = b;
            b = temp;
        }
```

e) Método de Inserción

El Método de Inserción facilita el ordenamiento de una lista y se caracteriza por su aplicación sencilla. Consiste en recorrer la lista, seleccionando en cada iteración un valor como clave y comparándolo con los valores restantes hasta insertarlo en su posición correspondiente.

El proceso comienza con una lista desordenada. Se elige el segundo valor como clave y se compara con el valor a su izquierda. Si es menor, se inserta en ese lugar. Luego, se selecciona el siguiente número y se repite el proceso para todos los valores anteriores.

La comparación se extiende hacia la izquierda en cada caso, encontrando un número menor que la clave o llegando al inicio de la lista. El resultado es una lista ordenada, donde cada iteración inserta un nuevo valor en su posición correcta. Este método es eficiente para listas pequeñas o parcialmente ordenadas.

- **Pseudocógido**

```
Procedimiento Insercion(lista):
   Para i desde 1 hasta el tamaño de la lista:
      clave = lista[i]
      j = i - 1

      Mientras j sea mayor o igual a 0 y lista[j] > clave:
         lista[j + 1] = lista[j]
         j = j - 1

      lista[j + 1] = clave
```

- **Código**

```
private static void OrdenamientoPorInsercion(int[] lista)
 {
     for (int i = 1; i < lista.Length; i++)
     {
         int clave = lista[i];
         int j = i - 1;

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

         lista[j + 1] = clave;
     }
}
```

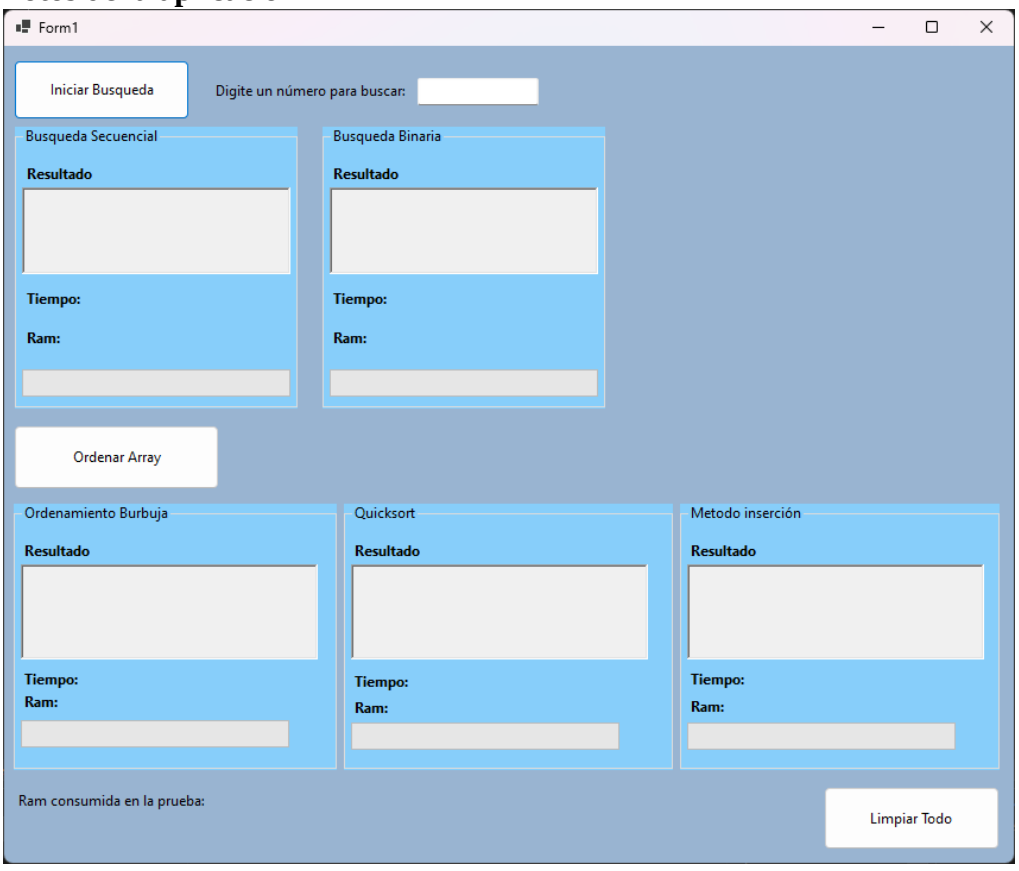
9. Programa desarrollado

a) Explicación de su funcionamiento

Esta aplicación presenta un cuadro de interfaz en el que se espera que el usuario ingrese un número entero. Este número actuará como el valor a buscar por los algoritmos de búsqueda implementados en la aplicación. Luego, al presionar el botón "Iniciar", la aplicación ejecutará una serie de procesos.

- **Generación de Arreglo:** Se crea un arreglo de 10,000 números enteros de manera aleatoria en un rango de 1 a 200,000. Este conjunto de datos servirá como la entrada para los algoritmos de búsqueda y ordenamiento.
- **Ejecución de Algoritmos:** La aplicación ejecuta diversos algoritmos de búsqueda y ordenamiento en paralelo. Los algoritmos involucrados se mencionan en la interfaz y pueden incluir Búsqueda Secuencial, Búsqueda Binaria, Ordenamiento Burbuja, Quick Sort, Método de Inserción, entre otros.
- **Resultados y Métricas:** Después de ejecutar todos los procesos, la aplicación presenta los resultados obtenidos por cada algoritmo. Esto incluye información sobre si el número ingresado se encontró, en qué posición, el consumo de memoria RAM y el tiempo que tomó realizar la operación.

b) Fotos de la aplicación



c) **Link de Github y Ejecutable de la aplicación**
https://github.com/SirBeho/carrera_paralelo

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

- **Búsqueda Secuencial**

  Tiempo: 00:00:00.0003886

  Ram: 0 Bytes

- **Búsqueda Binaria**

  Tiempo: 00:01:00.0717094

  Ram: 0 Bytes

- **Ordenamiento Burbuja**

  Tiempo: 00:01:00.0760864

  Ram: 720896 Bytes

- **Quicksort**

  Tiempo: 00:00:00.0118701

  Ram: 540672 Bytes

- **Método Inserción**

  Tiempo: 00:00:12.3436852

  Ram: 643072 Bytes

e) **¿Qué tanta memoria se consumió este proceso?**
Se consumió alrededor de 33 MB, de la memoria RAM disponible en el computador utilizado para la prueba

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

Búsqueda más rápida:

El algoritmo de búsqueda más rápida depende mucho del contexto, ya que si hablamos de un array ordenado Búsqueda Binaria siempre tendrá mas velocidad y optimización. Sin embargo en casos que el array no este ordenado este no puede ser utilizado dando como ganador el de búsqueda linear.

Si el array está ordenado, la Búsqueda Binaria generalmente será más eficiente en comparación con la Búsqueda Secuencial

Ordenamiento más rápido:

El más rápido fue Quicksort, que tomo tan solo 0.0118701 segundos en ordenar el arreglo pasado como prueba conteniendo 100,000 números, No es de sorprender pues Quicksort es muy conocido por ser eficiente clasificando, por lo que ha sido una elección muy popular para listas grandes. En esta prueba en especifico supero tanto al Método de Inserción como al Ordenamiento Burbuja.

11. Conclusión

Este proyecto subraya la crítica importancia de elegir algoritmos de manera cuidadosa, considerando factores específicos como el tamaño de los datos y el entorno de implementación. Quicksort destaca como una opción eficiente para ordenar, mientras que la velocidad de búsqueda puede variar según el contexto. La evaluación continua de los algoritmos es esencial para optimizar el rendimiento en situaciones específicas. En resumen, la selección informada de algoritmos es clave para alcanzar la eficiencia deseada en diversas tareas.

12. Bibliografías

https://gc.scalahed.com/recursos/files/r161r/w25474w/DisenoDeAlgoritmosParalelos.pdf

https://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro9/bsqueda_secuencial.html

https://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro9/bsqueda_binaria.html

https://juncotic.com/ordenamiento-de-burbuja-algoritmos-de-ordenamiento/

https://www.genbeta.com/desarrollo/implementando-el-algoritmo-quicksort

https://juncotic.com/ordenamiento-por-insercion-algoritmos-de-ordenamiento/