**tRABAJO PRACTICO FINAL**

**COMPARACION Y EFICIENCIA DE ALGORITMOS**

* + Guerrero Manuel Florián
  + 44420377
  + <https://github.com/ManuFGuerrero/MergeSortConcurrente>
  + <https://www.youtube.com/watch?v=AO4DU2rB7XE>
  + Manuguerrero02@gmail.com

**RESUMEN GENERAL:**

Para esta investigación me centre en las diferencias que existen entre los algoritmos secuenciales y aquellos que son concurrentes. Para ello, a través del **algoritmo de ordenamiento** MergeSort, me propuse realizar distintas comparaciones en cuanto a tiempo y **eficiencia algorítmica** y determinar en que situaciones es recomendable utilizar una técnica o la otra. El algoritmo fue extraído del foro Geeksforgeeks, fue subido en el año 2024 y se encontrara su referencia correspondiente en el apartado final. Este algoritmo cuenta con la implementación del método de ordenamiento de forma **secuencial** y **recursiva** y su ejecución con el uso de **hilos**, utilizando la cantidad de hilos que el sistema que ejecute el programa tenga disponibles para utilizar. Este algoritmo cuenta con una **complejidad** de O(n log(n) ) y se encuentra en unos niveles de complejidad que son aceptables, llegando a ser muy buenos si se aplica de forma concurrente

**1. Introduccion**

Para poder explicar el método de ordenamiento MergeSort voy a utilizar algunos términos bastantes sencillos dentro de la programación pero que no está de mas refrescar. Principalmente el termino de array, que refiere a una secuencia de elementos que se identifican con el índice de guardado dentro del mismo, tiene un tamaño definido con antelación y suele tener definido el tipo de dato que van a respetar los elementos que van a pertenecer al mismo. Además, se va a utilizar el término de Hilo, que refiere a una representación de un procesador lógico que se utiliza para ejecutar una porción del programa de forma tal que pueda ser ejecutada o no al mismo tiempo que otras.

Por lo tanto, contamos con un array inicial que contiene números enteros aleatorios, con un tamaño definido que va a ser variable según las distintas pruebas que se van a realizar a lo largo de la investigación.

Tabla

Descripción generada automáticamente con confianza baja

figura 1.1 – Representación de Array.

Como se puede ver este array contiene 5 elementos que se encuentran desordenados. El algoritmo MergeSort se va a encargar de dividir al arreglo en 2 mitades, utilizando una regla de oro muy conocida: “Divide y vencerás”, para poder manejar arreglos cada vez mas pequeños y por lo tanto más sencillos de ordenar. Aquí es donde sucede la primera bifurcación dentro de las 2 formas de ejecución, se explicará la parte secuencial primero y más adelante se explicará como son las distintas diferencias con su parte concurrente.

A partir de la subdivisión en distintos arreglos cada vez mas pequeños, se busca llegar al punto en donde el arreglo no logre poder ser dividido nuevamente, aquí ocurre y termina la primera gran parte de este algoritmo.

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

Figura 1.2 – Representación de subdivisión.

A partir de ahora el algoritmo va a tomar cada uno de los elementos indivisibles y los va a ordenar individualmente cada uno de ellos, luego toma 2 de los elementos atómicos existentes y los va a mezclar, que significa reordenarlos en un nuevo arreglo con sus elementos ya en su posición correcta, formando así paso por paso el array de salida correspondiente.

Por ultimo el algoritmo finaliza mezclando sus últimos 2 elementos, cada uno ya ordenado, en este array de salida que va a devolver para su futuro uso. Diagrama

Descripción generada automáticamente

Figura 1.3 – Representación de la convergencia y ordenamiento.

**2. IMPLEMENTACIÓN CONCURRENTE**

Para su aplicación concurrente se debe tener en cuenta distintos factores dentro del programa, para poder realizarlo de forma tal que cada uno de los hilos se encargue de una fracción del array de entrada, que los datos no se pisen y que cada hilo trabaje con su conjunto de datos, sin entrar en el dominio de su hilo contiguo. Es por eso que se debe conocer con que cantidad de hilos se va a trabajar para realizar el algoritmo. En este proyecto se utilizó el método de la clase Runtime:

Runtime.*getRuntime*().availableProcessors();

Este método permite saber la cantidad de procesadores lógicos disponibles dentro de la maquina que ejecuta el proyecto, sabiendo este numero el algoritmo lo divide y compara con el tamaño del array para hacer una cuenta y conocer con cuanta cantidad de elementos va a trabajar cada uno de los hilos, si el numero de hilos dividido la cantidad de elementos da un numero con resto 0 entonces estamos ante el escenario ideal donde el tamaño es divisible por la cantidad de hilos, en caso contrario se realiza esta división para todos los hilos menos para el ultimo que se encarga de realizar su parte + el resto de elementos.

Como segundo gran elemento a tener en cuenta es la asignación de cada uno de estos limites dentro de los hilos, utilizando un bucle que le da a cada hilo su cantidad de elementos correspondientes:

for (int i = 0; i < length; i += maxlim) {

int beg = i; //Elemento de inicio del hilo

int remain = (length) - i; //Elementos restantes

int end = remain < maxlim ? i + (remain - 1) : i + (maxlim - 1); //Elemento final del hilo

final SortThreads t = new SortThreads(array, beg, end);

// Agregar las referencias de los hilos para unirlos más tarde

threads.add(t);

}

A partir de este punto, ya que el start( ) de los hilos se encuentra dentro de su construcción:

private static class SortThreads extends Thread { //Se utiliza esta clase para llamar a la funcion de ordenamiento e iniciar hilos

SortThreads(Integer[] array, int begin, int end) {

super(() -> {

MergeSort.*mergeSort*(array, begin, end);

});

this.start();

}

Cada uno comienza a trabajar con su porción del arreglo, realizan la parte de dividir su array asignado, de forma recursiva, hasta llegar a un array indivisible. Una vez que cada uno de los hilos termina, se realiza el llamado a la función para hacer el Merge o la mezcla de estos elementos de forma tal que se comiencen a insertar ordenadamente en el array de salida quien al momento de estar completo indica el fin del algoritmo.

**3. COMPARATIVA y desempeño**

Para realizar cada una de las comparativas dentro de mi proyecto y poder comprender las pruebas realizadas con sus conclusiones particulares, se deben tener en cuenta 3 grandes puntos: Mi computadora cuenta con 12 procesadores lógicos para poder crear hilos, las pruebas van a estar contempladas en tiempo de ejecución en Milisegundos o MS para su comparación y los resultados van a ser tomados con números aproximados ya que cada prueba, más teniendo en cuenta las que manejan tamaños muy grandes, arroja resultados distintos pero lo que se quiere conseguir es poder comparar la diferencia entre esos tiempos aproximados.

Si tomamos de ejemplo de primera prueba el array que se muestra en las imágenes de referencia, con un tamaño bastante pequeño, podemos sacar una conclusión bastante rápida y es que el algoritmo secuencial le gana por bastante al algoritmo concurrente, ya que el secuencial no llega a tardar 1 ms y el concurrente tarda aproximadamente 5 o 6 ms.

Es por lo que para estas pruebas se van a tomar valores de tamaño que van a ir aumentando de forma exponencial, comenzando justamente con 10 elementos, y sumándole 2 ceros cada vez que se realice una nueva prueba.

**-Prueba para un tamaño de 10 elementos:**

Para esta prueba se obtuvieron tiempos aproximados de 1ms para el secuencial y 5ms para el concurrente. Esto sucede ya que, al manejar mas hilos de los necesarios incluso para la cantidad de elementos del arreglo, y teniendo en cuenta que se deben manejar estos hilos, llamándolos, esperándolos y uniendo sus resultados, este tiempo que se utiliza es el causante de que el secuencial sea el ganador.

**-Prueba para un tamaño de 1000 elementos:**

Esta prueba vuelca resultados muy similares a la prueba anterior, donde el algoritmo secuencial sigue siendo el ganador de esta carrera, continuando con su eficiencia de 1ms aproximadamente para completar el ordenamiento y su contraparte concurrente alcanza a tardar unos 7ms aproximadamente.

**-Prueba para un tamaño de 100.000 elementos:**

Al momento de realizar esta prueba pude comenzar a ver la verdadera eficiencia del algoritmo concurrente, que tarda unos 13ms aproximadamente para completar su ordenamiento, a comparación del algoritmo secuencial que tarda unos 38ms, justamente cuando tenemos un manejo de datos que comienza a oscilar entre estos números tan grandes es cuando los algoritmos concurrentes suelen ser los más óptimos para realizar las tareas correspondientes ya que muestran una facilidad muy grande para manejar tantos datos, además de aprovechar a través de los hilos el tiempo de forma correcta.

**-Prueba para un tamaño de 10.000.000:**

Para esta ultima prueba quise llevar un poco mas al limite a ambos algoritmos y ver resultados que tengan una brecha mucho mas marcada, conseguí resultados de 3780ms para el algoritmo secuencial (3,7 segundos) y 3143 ms (3,1 segundos) para el concurrente, mas de 600 ms entre uno y otro, consiguiendo así la mayor diferencia hasta el momento y comprobando de forma absoluta que el algoritmo concurrente es el ideal para manejar estos tamaños tan grandes.

Los casos de prueba con tamaños mas grandes que permitan ver resultados significativamente distintos a los ya enseñados, no son posibles debido principalmente a la composición del algoritmo, que este pensado y codificado de forma recursiva hace que la pila del sistema llegue a su limite produciendo el famoso error de “Stack Overflow”, ya que se acumulan demasiadas llamadas recursivas a la funcion MergeSort, es por eso que para realizar las conclusiones finales voy a realizar algunas pruebas más con números intermedios para formar una tabla que de forma gráfica indique fácilmente los resultados en ms y el tamaño que se utilizo para su prueba.

|  |  |  |  |
| --- | --- | --- | --- |
| N° | Cantidad de elementos | Tiempo Secuencial | Tiempo Concurrente |
| 1 | 100 | Entre 0ms y 1ms | Entre 4ms y 6ms |
| 2 | 10.000 | Entre 3ms y 5ms | Entre 14ms y 15ms |
| 3 | 1.000.000 | Entre 260ms y 275ms | Entre 155ms y 170ms |
| 4 | 9.999.999 | 3650ms aprox. | 3100 ms aprox |

Figura 3.1 – Representación de resultados de las pruebas

**4. CONCLUSIÓN**

Para concluir con este documento donde se realiza un análisis sobre la comparación del funcionamiento de un mismo método para ordenar un array de números aplicado en 2 distintas formas de programación, la forma secuencial y la forma concurrente, quisiera dar una idea final sobre como fue poder realizar esta investigación. Principalmente me sirvió a modo personal para poder conocer no solo este método, sino otros más que baraje para utilizar como herramienta de análisis, además de poder conocer algunos foros donde se comparte código, ayudando así a la fácil comunicación de la comunidad de programadores, en inglés y español, además de poder adaptarme a entender un código escrito por un tercero y poder no solo utilizarlo sino también explicárselo a la persona que lea este informe. Por último, este trabajo me llevo a entender de forma empírica y con números cuantificables la importancia de pensar los algoritmos de forma tal que ayuden a poder manejar cantidades de datos que en el entorno del trabajo en el área de sistemas es muy común y cotidiano. Para cerrar esta conclusión y este trabajo también debo reconocer que me ha resultado muy útil para seguir puliendo mis habilidades en cuanto a redactar informes claros y correctos que puedan ser utilizados como documentos oficiales a futuro.

**REFERENCIAS**

# -Shubham\_rana\_77 (2024) Merge Sort using Multi-threading

<https://www.geeksforgeeks.org/merge-sort-using-multi-threading/?ref=gcse>