Proyecto final del curso de Estructuras de Datos Avanzadas (2022-II): Comparando implementación secuencial y paralela del Árbol AVL

Luis Huachaca

Abstract—En este informe compararemos la estructura de Árbol AVL, una implementación paralela y una implementación secuencial, tambien la compararemos con otras implementaciones de la estructura, explicando el proceso, su funcionamiento y oportunidades de mejora.

Index Terms—Árbol, AVL, Estructura, Paralelos, Mutex, Lock.

1 Introducción

La programación en paralelo, es el uso de dos o mas procesadores en combinación para resolver un problema simple, nos ayuda en dividir la carga de trabajo, la mayoría de computadores utilizan este tipo de operación, sin embargo tiene complicaciones, de entendimiento porque es difícil trabajar con ellos, hay ocasiones donde afecta ya que al hacer procesos paralelos, se complican al momento de hacer merge o join, sobre todo en un arbol con rebalanceo automático y datos compartidos, esto puede llevar a demorar más que con una insercion secuencial, tambien tiene costo extra, como la transferencia de datos, sincronización, creación, destrucción, etc.

2 Mi estructura de datos

2.1 Como funciona

La estructura de datos AVL es bastante conocida entre los programadores, podríamos decir que AVL es un Binary Search Tree, que puede balancearse a si mismo, esta es la ventaja ya que se vuelve mas rápido en una búsqueda de peor caso, eso permite que se vuelva una estructura 'justa', ya que cuando esto no ocurre, las búsquedas empeoran en tiempo, AVL impone un límite al árbol que mantiene todas las operaciones en un O(log n) de tiempo.

Hablemos de como funciona, ya dijimos que esta estructura se re balancea y es un BST, donde la idea básica es, la diferencia entre la profundidad del hijo derecho o izquierdo no puede ser mayor que uno, este es el llamado factor de balanceo. Para garantizar esto, debemos incluir en la implementación un algoritmo de rebalanceo al agregar o eliminar un elemento.

2.1.1 Proceso de agregar

La inserción en AVL, considera rotaciones a la izquierda o a la derecha.

- Si hay un desbalance en el hijo izquierdo del subárbol derecho, hacemos una rotación izquierda a derecha
- Si hay un desbalance en el hijo izquiero del subárbol izquierdo, hacemos una rotación a la derecha.
- Si hay un desbalance en el hijo derecho del subárbol derecho, hacemos una rotación a la izquierda.
- Si hay un desbalance en el hijo derecho del subárbol izquierdo, hacemos una rotación de derecha a izquierda.

2.1.2 Rotaciones del árbol AVL

Se revisa el factor de balanceo en cada nodo, si cada nodo satisface este factor, la operación termina, si no, el árbol necesita ser rebalanceado con las operaciones de rotación. Estas rotaciones se clasifican en dos tipos.

- Rotacion LL: cada nodo se mueve una posición a la izquierda.
- Rotacion RR: Cada nodo se mueve una posición a la derecha.
- Rotacion LR: Se hace una rotacion LL y después una rotacion RR.
- Rotacion RL: Se hace una rotación RR y después una rotación LL.

2.2 Implementación Computacional

Para la implementación, se usan las operaciones básicas de la teoría del AVL. Como las rotaciones, adicionalmente se han implementado formas de impresiones, como la impresión PreOrder, impresión PostOrder y una impresión por niveles, para corroborar que ambas implementaciones, la secuencial y paralela generan el mismo árbol. Por último se usa la librería chrono para calcular el tiempo de ejecución, esto servirá en calcular la tabla de tiempos e indicarnos si nuestra estructura cumple el objetivo de mejorar el tiempo

de ejecución. Cabe destacar que el nodo es un struct, ya que debe considerar el nivel que esta, el valor que tiene y el puntero al nodo.

2.2.1 Estrategia Secuencial

La estrategia secuencial, como lo indicado en la implementación computacional, se crean rotaciones a la izquierta y rotaciones a la derecha, la idea principal es tomar puntero del hijo derecho, asimismo toma el puntero al nodo del hijo izquierdo, considerando como parámetro el nodo donde capto el desorden de nivel, hace la rotación con un swap, y le quita el nivel en menos uno a cada nodo. al final returna el nuevo nodo que corresponde en el punto de entrada (donde capto el desbalance).
Las inserciones evaluan donde debería ir el nodo de acuerdo a si es menor o mayor que el nodo dado. Luego verifican el nivel, con sus rotaciones.

2.2.2 Estrategia Paralela

Es importante mencionar que las operaciones de rebalanceo, inserción y eliminación complican la paralelización de la estructura, ya que tienen diferentes entradas y códigos que subejecutan en el proceso principal.
La estrategia que queremos aplicar debe aislar sub ejecuciones por ello el uso de semaforas y mutex es indispensable.
Lo primero que haremos para lograr nuestro objetivo serán abstraer funciones, para poder implementar paralelización en ellas, para eso crearemos un header que considera funciones específicas del avl, como las rotaciones o verificación de balance, sin embargo la mejor estrategia es abstraer avl como un pariente de la clase Binary Search Tree, ya que con esto vamos a poder darle funciones de balanceo y rotación a la clase BST, lo que permitira tener funciones simplificadas como agregar, eliminar o buscar en nuestra clase AVL. Estas simplificaciones seran perfectas al crear threads, ya que vamos a poder aplicar ideas en paralelo sin complicaciones, como agregar de cierto punto a cierto punto, Por ejemplo: tenemos 4 threads, si queremos ejecutarlo en nuestro árbol debemos buscar el nivel donde esten cuatro nodos y darle cada nodo como root a nuestro thread, sin olvidarnos de los nodos previos a este.
Simplificar las funciones no es lo único que nos servirá para lograr la paralelización, aun tenemos otro problema que es el shared memory, sabemos que cada thread debe compartir datos, especialmente en caso de rotaciones, por eso algunas implementaciones consideran el "Relaxed AVL"[6], que es una forma de evitar algunas rotaciones para evitar inseguridad en la memoria compartida. Pero nosotros lo que intentaremos será crear mutex y semáforas de la mejor forma posible, esto nos permitirá tener algún problema de miss, por ello en la abstracción de clase AVL intentamos colocar "mutex-lock(tree.lock)"[2] en la función de agregar, eliminar y la búsqueda, para ello creamos un lock en el inicializador de la clase AVL, asimismo creamos un mutex init al momento de llamar la clase, esto nos permitirá abrir y cerrar la puerta de memoria específicamente en el momento que las llamemos.
Las ventajas de agregar semáforas y mutex en la creación de threads son grandes, sin embargo, hay otras ventajas inesperadas que vienen con estos agregados, reducirán el miss en el momento de manejar datos, quita riesgos de sobreescribir el dato, por último, le dan exclusividad en la ejecución de funciones lo que también hace que no haya mucho cambio en el procesador, esto también mejora la velocidad de la estructura.

Aquí podremos ver los lock implementados para mejor entendimiento.

```
//class AVL
        public:
        pthread_mutex_t lock;
        avlTree(){
            root = NULL;
            nNodes = 0;
            //pthread_mutex_init(&lock, NULL);
        }
        bool avl_contains(int n){
            pthread_mutex_lock(&lock);
            //bool b = searchTree(root, n);
            bool b = searchTree(n);
            pthread_mutex_unlock(&lock);
            return true;
        };
        void avl_add(int n){
            pthread_mutex_lock(&lock);
            //add_avl(root,n);
            add(n);
            pthread_mutex_unlock(&lock);
            //add_avl(root,n);
            return;
        };
        void avl_delete(int n){
            /*if(!avl_contains(n))
                return;
            else{*/
                pthread_mutex_lock(&lock);
                delNode(n);
                pthread_mutex_unlock(&lock);
            //}
        }
```

3 Experimentos

Para hacer la experimentación, vamos a usar 3 códigos, el primero será la implementacion de AVL de geeks for geeks, el segundo sera la implementación AVL secuencial presentada para el proyecto parcial, y el tercero será la implementacion con paralelismo para el proyecto final, incluimos el AVL de geeks for geeks ya que tiene unas rotaciones extra y será interesante como se ve su cálculo de acuerdo al tiempo.

3.1 Medidas de tiempo

Cómo se mencionó en la implementación computacional, usaremos la librería chrono para tener los tiempos, en inserción usaremos insercion de 100, 1000 hasta 10 millones de datos, el salto es agregándole un cero al anterior. Para la eliminación solo contaremos el tiempo de eliminación y no

de inserción,para las búsquedas, vamos a contar el tiempo de eliminación, inserción, mas la búsqueda del número pedido.
La columna x de nuestras tablas serán la cantidad de datos insertados, eliminados o usados. La columna y representa el promedio de tiempos que tuvo, en representación log10 para mayor orden.

3.1.1 Inserción

Figura 1.

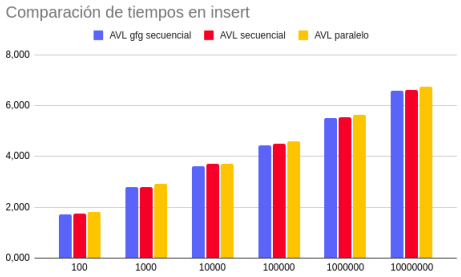


Fig. 1. Comparación de tiempos de inserción

3.1.2 Búsqueda

Para hacer esta consulta se estan sumando los tiempos de insertar, buscar y eliminar (figura 2).

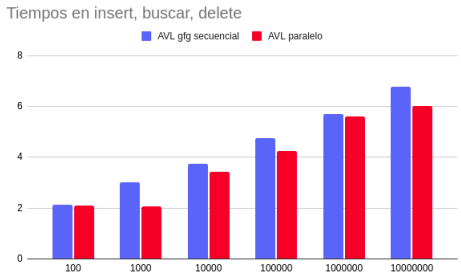


Fig. 2. Comparación de tiempos de búsqueda

3.1.3 Eliminación

Gráfico de comparación en Delete (figura 3).

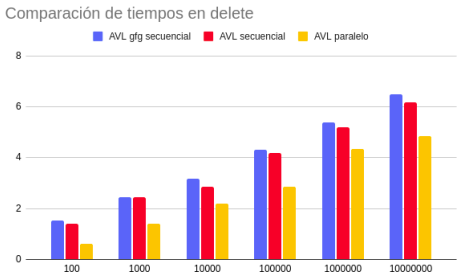


Fig. 3. Comparación de tiempos de eliminación

3.2 Otros aspectos

Como se ha dicho, el usar semáforas y mutex ayuda mucho al tema de evitar escribir en lugares donde no pertenecen, faults, misses, etc. Entonces la implementación paralela tiene una ventaja en ese aspecto.

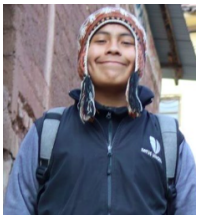
4 Conclusión

Después de analizar los gráficos podemos ver ventajas en cuanto a la paralelización de nuestra estructura, más en el delete que en el insert, también podemos ver mejoras en el proceso en conjunto, cabe resaltar que hacer los tiempos a log10 ayudo mucho al orden, sin embargo no denotó claramente la ventaja de unos a otros, ya que, si analizamos las tablas de tiempos a profundidad, encontraremos que la versián paralela en ciertos puntos es incluso la mitad de tiempo que la versión secuencial, concluimos que es una buena mejora paralelizar, debido a la capacidad computacional que tenemos hoy en día, sin embargo, no es tan segura ya que al trabajar con memoria compartida puede haber riesgos de seguridad.

Finalmente el código y tablas se encuentran en el siguiente repositorio: https://github.com/Luis-Huachaca-HV/EDA/tree/main/EntregaFinal

References

[1] https://www.freecodecamp.org/news/avl-tree-insertion-rotation-and-balance-factor/
[2] https://github.com/piwatake/COP4520TermProject
[3] https://github.com/joshroybal/multithreaded-tree-test/blob/master/src/main.cpp
[4] https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
[5] https://www.geekboots.com/story/parallel-computing-and-its-advantage-and-disadvantage
[6] https://dl.acm.org/doi/pdf/10.1145/28659.28677



Luis Huachaca Estudiante de CCOMP en la universidad Católica San pablo, nació en Sicuani en abril del 2003, participó en el proyecto rutas de la UCSP creando código para el robot Pepper, participó en el CEP(concurso escolar de programación) como entrenador, participó como embajador para el proyecto de acreditación ABET de la carrera CCOMP en UCSP, participó como voluntario en GIPE2022 y ha sido seleccionado para el proyecto GIPE2023 que tendrá un springschool en alemania donde desarrollarán un proyecto de interrelación cultural con alumnos de Namibia, Indonesia, Peru y Alemania en los acelerators labs de UNDP.