RECORRIDOS CONCURRENTES Y SECUENCIALES EN UN ÁRBOL BINARIO

Datos del autor: Leandro Iván Vera
Repositorio con el trabajo: https://github.com/Lean-IV/arbolBinario
Video explicativo (<10min): https://youtu.be/KABlWZGStNM?si=MItkUxBrHESMYF4I
E.mail: lean.ivan.vera@gmail.com

RESUMEN (ABSTRACT)

Este trabajo presenta la implementación de recorridos secuenciales y concurrentes (PreOrder, InOrder y PostOrder) en un árbol binario de búsqueda, previamente estudiado en la materia "Algoritmos y estructuras de datos", utilizando la biblioteca pthread.h para gestionar la concurrencia. Se desarrolla un generador de árboles con datos aleatorios y se evalúa el rendimiento mediante funciones artificiales. La comparación entre ambas versiones se realiza midiendo el tiempo de ejecución con clock() en árboles de diferentes tamaños. Se controla la cantidad de hilos en ejecución de forma adaptativa según los núcleos disponibles. La estructura de datos se encapsula mediante getters y setters para asegurar buenas prácticas de diseño modular. El algoritmo concurrente fue desarrollado por el autor como parte del trabajo práctico final de Programación Concurrente.

Keywords: árbol binario, pthreads, concurrencia, sincronización, recorridos PreOrder, InOrder y PostOrder.

1. INTRODUCCIÓN

En este trabajo se analiza y modifica un algoritmo de recorridos de árboles binarios de búsqueda, implementado de forma secuencial y recursiva. El objetivo es transformarlo en una versión concurrente que aproveche múltiples núcleos de CPU, utilizando la biblioteca pthread.h del lenguaje C.

Los recorridos implementados son PreOrder, InOrder y PostOrder, clásicos en estructuras de datos. Estas funciones recorren el árbol respetando diferentes órdenes de visita:

PreOrder: raíz → subárbol izquierdo → subárbol derecho

InOrder: subárbol izquierdo → raíz → subárbol derecho

PostOrder: subárbol izquierdo → subárbol derecho → raíz

A modo de ejemplo, la función secuencial postOrder() se ve así:

```
void postOrder(ArbolPtr arbol) {
   if (arbol != NULL) {
      postOrder(arbol->izq);
      postOrder(arbol->der);
      printf(" %d ", arbol->dato);
   }
}
```

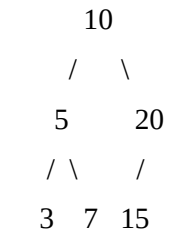
Esta implementación recursiva fue originalmente vista en la asignatura "Algoritmos y Estructuras de Datos", correspondiente al plan de estudios de la carrera Licenciatura en Sistemas de la Universidad Nacional de Lanús. En dicha materia se trabajaron estructuras dinámicas como listas enlazadas, aplicando buenas prácticas de encapsulamiento mediante getters y setters. A partir de esa base, y siguiendo el mismo enfoque de diseño modular aprendido en clase, se adaptó el código del árbol binario aplicando dicho encapsulamiento, con el objetivo de estructurarlo correctamente para futuras mejoras y análisis.

El código secuencial base se encuentra disponible en el siguiente repositorio:

https://github.com/Lean-IV/arbolBinario

(en la carpeta raíz, archivos arbol.c y arbol.h).

A continuación, se muestra un diagrama simple que ilustra el recorrido PostOrder en un árbol binario:



El recorrido PostOrder de este árbol sería: 3 → 7 → 5 → 15 → 20 → 10

Cada llamada recursiva visita primero ambos subárboles antes de procesar el nodo actual.

2. IMPLEMENTACIÓN CONCURRENTE

A partir del algoritmo secuencial de recorrido de árboles binarios explicado en el apartado anterior, se implementaron versiones concurrentes para los tres tipos de recorrido: PreOrder, InOrder y PostOrder. El objetivo fue paralelizar las llamadas recursivas a los subárboles izquierdo y derecho utilizando hilos, de forma que se puedan procesar ramas del árbol en paralelo, siempre que haya recursos disponibles.

La implementación se realizó utilizando la biblioteca estándar de hilos en C: pthread.h. Para evitar una sobrecarga de hilos innecesaria, se controla la cantidad de niveles de concurrencia en función de la cantidad de núcleos del procesador. Dado que este proyecto fue desarrollado y probado en Windows, se utilizó la función GetSystemInfo de la biblioteca <windows.h> para obtener la cantidad de núcleos disponibles:

```
int obtenerCantidadNucleos() {
    SYSTEM_INFO sysinfo;
    GetSystemInfo(&sysinfo);
    return sysinfo.dwNumberOfProcessors;
}
```

Esta información permite limitar el número de hilos creados según la profundidad del árbol, evitando una explosión de hilos innecesaria. La lógica aplicada es: si el nivel de profundidad de la recursión es menor que la cantidad de núcleos, se crean hilos para recorrer los subárboles en paralelo; de lo contrario, el recorrido continúa de forma secuencial.

A continuación, se presenta un fragmento del código que ilustra cómo se crea un hilo para recorrer el subárbol izquierdo en PreOrder:

```
if (obtenerIzq(arbol)) {
    if (data->nivel < data->maxNiveles) {
        tieneIzq = 1;
        pthread_create(&hiloIzq, NULL, preOrderConcurrenteThread, &izqData);
    } else {
        preOrderConcurrenteThread(&izqData);
    }
}
```

Esta lógica se aplica tanto en los recorridos PreOrder, InOrder y PostOrder, usando una estructura auxiliar ThreadData para mantener el árbol actual, el nivel de profundidad y el número máximo de niveles permitidos para concurrencia.

El archivo que contiene la implementación de los recorridos concurrentes se encuentra en:

https://github.com/Lean-IV/arbolBinario

Para el recorrido PostOrder, además, se incluyó una función llamada tareaPesada(int valor) que simula una carga computacional importante en cada nodo, con el fin de hacer más notoria la diferencia de rendimiento entre el algoritmo secuencial y el concurrente:

```
void tareaPesada(int valor) {
    double resultado = 0;
    for (int i = 0; i < 100000; i++) {
        resultado += sin(valor) * cos(valor);
    }
}
```

3. COMPARATIVA Y DESEMPEÑO

Para comparar el rendimiento de los recorridos secuenciales y concurrentes implementados, se realizaron distintas pruebas utilizando árboles binarios de diferentes tamaños, medidos en cantidad de nodos insertados.

La generación de datos del árbol se realizó con números enteros aleatorios mediante la función llenarArbol(). En cada nodo visitado se ejecuta una función tareaPesada(int valor) que simula una carga de procesamiento, con el fin de hacer evidente la diferencia entre procesamiento secuencial y paralelo.

El tiempo de ejecución fue medido con la función clock() de la biblioteca <time.h>, obteniendo el tiempo total consumido por cada recorrido. A continuación se detallan las especificaciones del sistema utilizado y los resultados obtenidos.

Sistema Operativo: Windows 11 Home

Procesador: AMD Ryzen 5 8645HS w/ Radeon 760M Graphics 4.30 GHz

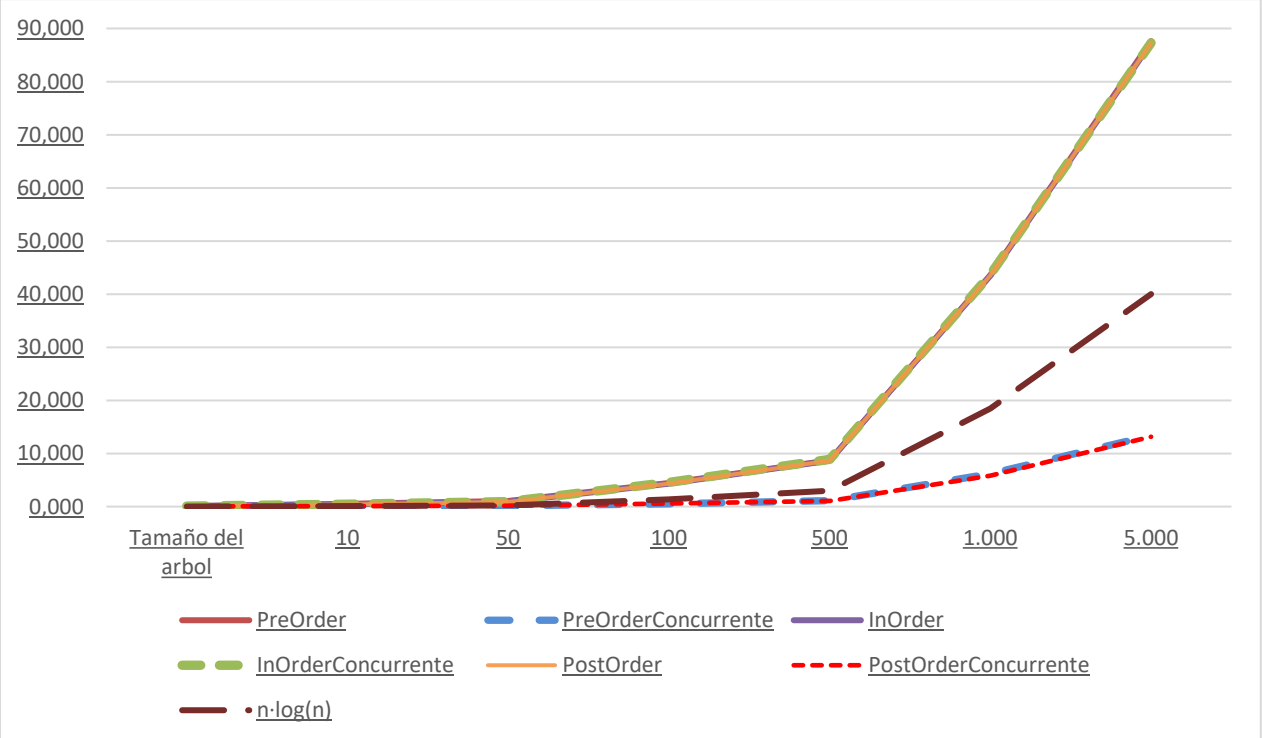
Núcleos físicos: 6

RAM: 8 GB

Compilador: Code::Blocks con GCC (MinGW)

Tabla y grafico comparativo

Tam. del arbol	PreOrder	PreOrder Conc	InOrder	InOrder Conc	PostOrder	PostOrder Conc
10	0.088 s.	0.035 s.	0.091 s.	0.094 s.	0.086 s.	0.040 s.
50	0.527 s.	0.096 s.	0.438 s.	0.461 s.	0.426 s.	0.110 s.
100	1.051 s.	0.134 s.	0.860 s.	0.923 s.	0.871 s.	0.155 s.
500	4.412 s.	0.500 s.	4.341 s.	4.558 s.	4.322 s.	0.587 s.
1.000	8.731 s.	1.092 s.	8.674 s.	8.902 s.	8.660 s.	1.077 s.
5.000	43.593 s.	6.156 s.	43.491 s.	43.734 s.	43.533 s.	5.805 s.
10.000	87.631 s.	13.381 s.	87.251 s.	87.290 s.	87.258 s.	13.147 s.



Como se puede apreciar en la tabla y el gráfico anterior, los recorridos concurrentes presentan una mejora progresiva de rendimiento a medida que aumenta el tamaño del árbol. Esta mejora es evidente

a partir de estructuras con 100 nodos o más, donde el costo de la tarea aplicada se vuelve significativo y, por lo tanto, aprovechable mediante paralelización.

El recorrido PostOrderConcurrente y PreOrderConcurrente son los que logran una diferencia más notoria con respecto a sus versiones secuenciales. Esto se debe a que la función tareaPesada() se ejecuta después de visitar subárboles, permitiendo que estos se recorran en paralelo sin afectar el orden lógico del algoritmo.

En cambio, el recorrido InOrderConcurrente no presenta mejoras e incluso puede ser levemente más lento en algunos casos, debido a que debe esperar que finalice el recorrido completo del subárbol izquierdo antes de continuar con el nodo actual y luego con el subárbol derecho. Esta dependencia secuencial limita el paralelismo efectivo y reduce los beneficios que podrían obtenerse mediante la concurrencia.

4. CONCLUSIÓN

Este trabajo permitió analizar, implementar y comparar distintas versiones de recorridos de árboles binarios —tanto secuenciales como concurrentes— en lenguaje C, utilizando pthread.h para la creación y gestión de hilos. A partir de una base académica abordada en la Universidad Nacional de Lanus, se adaptó el algoritmo original, aplicando buenas prácticas de diseño como encapsulamiento mediante getters y setters, y se extendió el código para evaluar su rendimiento en contextos concurrentes.

El desarrollo del proyecto se realizó de forma progresiva, atravesando las siguientes etapas:

Se partió de un código de recorrido secuencial de árbol binario visto en la universidad.

Se incorporó encapsulamiento mediante funciones getter y setter, según la metodología trabajada en clase.

Se agregó la medición de tiempos de ejecución utilizando la función clock().

Se creó una función llenarArbol() que permite generar árboles con datos aleatorios y tamaño variable.

Se desarrollaron versiones concurrentes de los recorridos preOrder, inOrder y postOrder utilizando pthread.h.

Se incorporó la detección automática de la cantidad de núcleos físicos del procesador, mediante la función GetSystemInfo() de <windows.h>, para limitar la cantidad de hilos creados.

Se observó que sin una carga computacional relevante no se notaban diferencias de rendimiento, por lo que se agregó una función tareaPesada() ejecutada en cada nodo.

Finalmente, se analizó que el recorrido inOrderConcurrente no se beneficiaba de la concurrencia debido a su dependencia secuencial, lo cual fue documentado como observación técnica.

Los resultados obtenidos mostraron que los beneficios de la concurrencia se hacen evidentes cuando el árbol alcanza cierto tamaño y el trabajo por nodo es significativo. En la práctica, tanto el recorrido `PreOrderConcurrente` como el `PostOrderConcurrente` ofrecieron mejoras de rendimiento sustanciales en comparación con sus versiones secuenciales. Si bien teóricamente el recorrido PostOrder permite una mayor paralelización al procesar el nodo después de ambos subárboles, en la implementación realizada ambos recorridos concurrentes mostraron rendimientos similares en términos de tiempos medidos.

Por otro lado, el recorrido `InOrderConcurrente`, por su estructura, no presentó mejoras notables. Esto evidencia que no todos los algoritmos recursivos se benefician de la concurrencia por igual, y que es necesario comprender la dependencia entre pasos para decidir si es posible paralelizar de forma efectiva.

Es importante destacar que, al paralelizar el procesamiento de los subárboles, no se garantiza el orden estricto de recorrido tradicional.

En los algoritmos secuenciales, los recorridos respetan un orden bien definido (por ejemplo,

primero el nodo, luego el subárbol izquierdo, etc.). Sin embargo, en las versiones concurrentes, los subárboles pueden terminar de procesarse en un orden distinto al esperado, dependiendo de la planificación del sistema operativo y la disponibilidad de los núcleos.

Por esta razón, el recorrido concurrente no debe utilizarse cuando el orden es crítico. En cambio, sí es adecuado cuando lo que se busca es rendimiento, especialmente si en cada nodo se realiza una tarea computacional intensiva e independiente, como la simulada con tareaPesada() en este proyecto.

REFERENCIAS

Como crear hilos en C usando Pthread explicadas en:

https://www.youtube.com/watch?v=ROzCbSwrltA

Obtener detalles sobre el procesador: https://localhorse.net/article/como-obtener-informacion-del-sistema-en-c++-usando-las-funciones-de-la-api-de-windows

Encontrar el número de núcleos en C:

https://www.reddit.com/r/C_Programming/comments/6zxnr1/how_to_find_the_number_of_cores_in_c/?tl=es-419

Uso de funciones trigonométricas para tareas pesadas:

https://stackoverflow.com/questions/2479517/is-trigonometry-computationally-expensive