Árvore de Pesquisa Binária Concorrente

Hugo Gonçalves[1], Paulo Shirley[2]

[1] Universidade Aberta de Portugal, Lisboa, Portugal
email@hugogoncalves.pt
[2] Universidade Aberta de Portugal, Lisboa, Portugal
paulo.shirley@uab.pt

Resumo

Este artigo propõe a implementação de uma biblioteca em linguagem C para uma Árvore de Pesquisa Binária (*Binary Search Tree*), onde se oferecem diversos métodos de controlo de concorrência para as operações mais comuns efetuadas nesta estrutura de dados. É apresentada uma análise sobre os ganhos ou perdas de cada método para diversos cenários replicáveis de utilização da árvore.

Palavras-chave: árvore pesquisa binária, BST, estrutura de dados, concorrência, multitarefa, programação paralela.

Title: Concurrent Binary Search Tree

Abstract: This article proposes the implementation of a library in C language for a Binary Search Tree, where several concurrency control methods are offered for the most common operations performed in this data structure. An analysis of the gains or losses of each method in several replicable scenarios is presented.

Keywords: Binary Search Tree, BST, data structure, concurrent, multithreaded, parallel programming.

1. Introdução

Com a evolução dos processadores e o aumento do número de núcleos que estes oferecem [Nguyen, 2023], torna-se, cada vez mais necessário que os algoritmos e estruturas de dados sejam adaptados por forma a utilizar toda a capacidade oferecida por estes. Neste âmbito, desenvolveu-se uma biblioteca em linguagem C com a implementação de uma árvore de pesquisa binária (*Binary Search Tree* [Wikipedia, 2020]) com diversos métodos de controlo de concorrência para as operações mais comuns. A gestão de concorrência é obtida pela utilização de primitivas oferecidas pela biblioteca POSIX *pthread* [pthreads, 2024] e *stdatomic* [Standard library header, 2022]. Existindo diversos mecanismos de controlo de concorrência, pretende-se fazer um estudo sobre o impacto no desempenho

nesta estrutura de dados entre a versão Sequencial (*Singlethread*) e Concorrente (*Multithread*), bem como comparar as diferenças entre os vários mecanismos de controlo de concorrência. Para tal, a biblioteca irá conter diferentes funções para cada tipo de operação a paralelizar, onde cada função, utiliza um mecanismo diferente de controlo de concorrência.

A tabela seguinte, ilustra as operações e os métodos de controlo de concorrência a implementar:

OPERAÇÃO	MECANISMOS DE CONTROLO
INSERÇÃO	Sequencial, RwLock Global, Mutex Local, Lock-Free/CAS
REMOÇÃO	Sequencial, RwLock Global, Mutex Local, Lock-Free/CAS
PROCURA	Sequencial, RwLock Global, Lock-Free/CAS
MIN/MAX	Sequencial, RwLock Global, Lock-Free/CAS

Tabela 1.1. Operações na estrutura de dados e mecanismos de controlo de concorrência. Legenda: RwLock – Trinco de leitura e escrita; Lock Free – Sem trincos; CAS – Operação de Comparar e Atribuir (*Compare and Set*).

Mecanismo Sequencial

O mecanismo Sequencial representa as operações sequenciais convencionais sobre a árvore binária, perante as quais se pretende obter um ganho no desempenho. As mesmas estarão presentes na biblioteca como ponto de comparação para a análise a efetuar.

Mecanismo com RwLock Global

Com este mecanismo, pretende-se utilizar um único RwLock (*Coarse-grained Lock)* [Djakaria, K, 2005] para gerir a concorrência na árvore binária. Este será o primeiro nível de otimização do desempenho das operações concorrentes nesta estrutura de dados.

Mecanismo com Mutex Local

Este mecanismo utiliza um *Mutex* local (*Fine-grained Lock*) [CMU, 2018] ao nó a modificar, substituindo o RwLock global. Este será o segundo nível de otimização, pretendendo obter ganhos adicionais de desempenho executando mais operações paralelas de escrita em nós distintos.

Mecanismo Lock-Free/CAS

Lock-free ou CAS, [CMU, 2018] [Pimpale, 2015] [Agrawal, 2024] será o último nível de otimização onde se espera o maior ganho de desempenho, através da utilização de operações atómicas, que substituem os mecanismos de bloqueio (trincos). Utilizou-se o esquema de *Hazard Pointers* [Michael, 2004] para evitar acessos a memória inválida.

Através da análise dos ganhos de desempenho, utilizando os diversos mecanismos de gestão da concorrência nas operações da árvore binária, pretende-se identificar quais são os mais apropriados – leia-se, com maiores ganhos – em cenários distintos de utilização desta estrutura de dados.

Numa fase inicial, o primeiro objetivo será desenhar e implementar os diversos mecanismos almejando uma utilização simples dos mesmos. Posteriormente, serão identificados casos comuns de utilização de árvores binárias e o desenho de baterias de testes, que simulem estes cenários por fim a obter dados concretos do seu desempenho.

Por último, pretende-se analisar os dados obtidos e retirar conclusões, acerca dos mecanismos a utilizar consoante o tipo de utilização da estrutura de dados. Preveem-se três níveis de otimização do desempenho, sendo esperado que cada um incremente o desempenho da árvore binária, ainda que marginalmente para alguns casos

A estrutura do artigo encontra-se dividida da seguinte forma. Na secção 2 abordam-se os algoritmos utilizados para manipular a árvore sem e com concorrência, assim como as estratégias de teste das diversas versões. Na secção 3 descreve-se a interface de linha de comandos do software desenvolvido. Na secção 4 apresentam-se as métricas utilizadas para comparar os algoritmos e os resultados obtidos. Por último, na secção 5 apresentam-se as conclusões.

2. Algoritmos e Estratégias de Teste

As versões implementadas compreendem uma árvore binária clássica sem balanceamento automático que armazena inteiros de 64 bits. As versões desenvolvidas estão disponíveis num repositório de acesso livre [Gonçalves, 2024], sendo que os ficheiros bst_st contêm a versão Sequencial (S*ingle Thread*), os ficheiros bst_mt_cgl contêm a versão concorrente de bloqueio grosso (*Coarse-Grained Lock*), os ficheiros bst_mt_fgl contêm a versão concorrente de bloqueio fino (*Fine-Grained Lock*) e os ficheiros bst_at contêm a versão sem bloqueios (*Lock-free*) com operações atómicas.

2.1 Versão Sequencial

Esta versão da biblioteca implementa as operações mencionadas na secção 1, sem controlo de concorrência, ou seja, devem ser sempre utilizadas sequencialmente para prevenir comportamentos inesperados.

Esta é a versão base, sobre a qual serão recolhidas métricas com várias Estratégias de Teste, para posteriormente aferir as melhorias relativamente às versões seguintes. Entre todas as operações implementadas, detalha-se a seguir o algoritmo utilizado para inserção de novos valores. As restantes operações utilizam versões modificadas do mesmo.

```
Inicio;
Se raiz vazia:
        Adicionar elemento na raiz;
        Terminar;
Se raiz não vazia:
        Ponteiro p = raiz;
        Enquanto p não vazio:
                Se p.valor == valor:
                        Valor já inserido;
                        Terminar;
                Se p.valor < valor:
                        Se p.esquerda vazio:
                                Adicionar elemento em p.esquerda;
                                Terminar;
                        p = p.esquerda;
                Se p.valor > valor:
                        Se p. direita vazio:
                                Adicionar elemento em p.esquerda;
                                Terminar;
                        p = p.direita;
```

Pseudocódigo 2.1. Inserção Sequencial.

2.2 Versão com RwLock Global

A versão concorrente de bloqueio grosso (*Coarse-grained Lock*) [Djakaria, 2005], utiliza um único trinco RwLock global para proteger o acesso à árvore binária na sua totalidade, tanto aos nós da árvore como ao contador de elementos desta. Esta versão, permite acesso em simultâneo por diversas tarefas (*threads*) sem que a integridade dos dados seja comprometida. Durante os testes, esta versão demonstrou uma alta eficiência em operações de leitura e evidenciou problemas de contenção relacionados com o aumento do número de tarefas, bem como a sobrecarga (*overhead*) dos mecanismos de bloqueio e criação das próprias tarefas. Na secção 4, serão abordadas com maior detalhe as métricas recolhidas. À semelhança da versão sequencial, detalha-se de seguida o algoritmo de inserção com controlo global de concorrência.

2.3 Versão com Mutex Local

Esta versão, utiliza um Mutex local a cada nó da árvore. Manteve-se um contador global de nós, que é protegido por um trinco RwLock.

```
Inicio;
Bloquear RwLock para escrita;
Se raiz vazia:
        Adicionar elemento na raiz;
        Desbloquear RwLock;
        Terminar;
Se raiz não vazia:
Ponteiro p = raiz;
Enquanto p não vazio:
Se p.valor == valor:
        Valor já inserido;
        Desbloquear RwLock;
        Terminar;
Se p.valor < valor:
        Se p.esquerda vazio:
                Adicionar elemento em p.esquerda;
                Desbloquear RwLock;
                Terminar;
        p = p.esquerda;
Se p.valor > valor:
        Se p. direita vazio:
                Adicionar elemento em p.esquerda;
                Desbloquear RwLock;
                Terminar;
        p = p.direita;
```

Pseudocódigo 2.2. Inserção com trinco RwLock global.

2.4 Estratégias de Teste

Com o objetivo de identificar cenários concretos onde cada versão da árvore binária melhor se adapta, foram definidas diversas estratégias de testes. Cada estratégia aglomera um ou mais tipo de operações realizadas num ciclo com um determinado número de iterações. Em cada iteração é escolhida, de forma aleatória, uma operação, recorrendo à função rand_r() [rand_r, 2024], onde a semente é única a cada tarefa. Isto permite que cada tarefa execute diferentes operações na árvore em simultâneo. Todas as estratégias recolhem a contagem de cada operação executada para cada cenário de teste.

Os dados utilizados nos testes são pré-gerados e misturados com recurso ao algoritmo moderno de Fisher–Yates [Eberl, 2024].

Tendencialmente, cada execução de um cenário de teste, resulta em contagens semelhantes para as operações incluídas na estratégia escolhida. Por fim, para simular uma maior carga na comparação das chaves nos nós, em vez de comparar os valores dos nós, são executadas 250 instruções de cópia para criar um atraso artificial. Inicialmente foi testada a utilização da função usleep() [usleep, 2024], disponível no cabeçalho

unistd.h, no entanto esta função suspende a tarefa atual e esta fica à mercê do escalonamento do sistema operativo, podendo o tempo de espera ser superior ao solicitado, o que iria invalidar as métricas recolhidas.

```
Inicio;
Bloquear RwLock global para escrita;
Se raiz vazia:
        Adicionar elemento na raiz;
        Desbloquear RwLock global;
        Terminar;
Se raiz não vazia:
        Desbloquear RwLock global;
        Bloquear RwLock global para leitura;
        Ponteiro p = raiz;
        Enquanto p não vazio:
                Bloquear RwLock local p para escrita;
                Se p.valor == valor:
                        Valor já inserido;
                        Desbloquear RwLock local p;
                        Desbloquear RwLock global;
                        Terminar;
                Se p.valor < valor:
                        Se p.esquerda vazio:
                                Adicionar elemento em p.esquerda;
                                Desbloquear RwLock local p;
                                Desbloquear RwLock global;
                                Terminar;
                        Desbloquear RwLock local p;
                        p = p.esquerda;
                Se p.valor > valor:
                        Se p. direita vazio:
                                Adicionar elemento em p.esquerda;
                                Desbloquear RwLock local p;
                                Desbloquear RwLock global;
                                Terminar;
                        Desbloquear RwLock local p;
                        p = p.direita;
```

Pseudocódigo 2.3. Inserção com Mutex local a cada nó da árvore.

2.4.1 Inserção

Esta estratégia permite aferir os ganhos ou perdas de cada versão da árvore para operações de inserção.

2.4.2 Escrita

Com esta estratégia pretende-se avaliar alterações na árvore, contemplando tanto a inserção como a remoção de elementos de forma aleatória, em paralelo.

2.4.3 Leitura

Esta estratégia foca-se apenas na leitura e aleatoriamente são executadas as operações de pesquisa, determinação do valor mínimo e máximo das chaves, bem como o cálculo da altura e largura da árvore.

2.4.4 Leitura e Escrita

A estratégia de leitura e escrita, combina todas as operações de escrita com as de leitura. Por defeito, existe uma probabilidade aproximada de 50% de a operação ser de escrita ou de leitura, podendo esta probabilidade ser configurada.

3 Utilização do Software

Para que os cenários de teste sejam replicáveis e de fácil execução, a biblioteca é acompanhada de um executável que permite escolher a estratégia de testes, a quantidade de operações pretendidas e o número de tarefas para executar as operações.

3.1 Opções

Através dos argumentos da linha de comandos é possível configurar e executar um ou mais cenários de teste.

Número de operações
-o <#op>: Define o número de operações a executar numa estratégia de teste. No caso da estratégia de inserção, são efetuadas <#op> inserções ou no caso da estratégia de leitura são executadas <#op> operações de leitura.

Número de tarefas
-t <#threads>: Define o número de tarefas a utilizar. Caso seja escolhida a árvore binária sequencial, esta opção é ignorada e apenas uma tarefa é utilizada.

Número de repetições
-r <#rep>: Define o número de repetições da estratégia configurada, com o objetivo de posteriormente obter uma média dos resultados.

Estratégia de testes
-s <strat>: Define uma estratégia de testes a utilizar. Podem ser especificadas múltiplas estratégias, sendo que as restantes opções se aplicam de igual modo a todas as estratégias. 1 – Inserção; 2 – Escrita; 3 – Leitura; 4 – Leitura e Escrita.

Árvore binária sequencial (ST)

-c: Define que a estratégia de testes deverá utilizar a árvore binária sequencial, ou seja, sem mecanismo de controlo de concorrência. Podem ser especificados múltiplos tipos de árvore, sendo que as restantes opções se aplicam de igual modo a todos os tipos de árvore, ou seja pode ser combinado com as opções -a, -g e -l.

Árvore binária com RwLock Global (*Coarse-Grained Lock,* CGL)
-g: Define que a estratégia de testes deverá utilizar a árvore com o mecanismo de controlo de concorrência RwLock global.

Árvore binária com Mutex local (Fine-*Grained Lock,* FGL)
-l: Define que a estratégia de testes deverá utilizar a árvore com o mecanismo de controlo de concorrência RwLock local.

Árvore binária sem trincos (Lock-Free, AT)
-a: Define que a estratégia de testes deverá utilizar a árvore com o mecanismo de controlo de concorrência baseado em operações atómicas.

Output
Para fácil importação para ferramentas de tratamento de dados, o output do programa de testes tem um formato CSV com as seguintes colunas:
<bst_type>,<strategy>,<#operations>,<#threads>,<#tree_node_count>,<tree_min>,<tree_max>,<tree_height>,<tree_width>,<time_taken>,<#inserts>,<#searches>,<#mins>,<#maxs>,<#heights>,<#widths>,<#deletes>,<#rebalances>

4. Métricas e Resultados

Com o objetivo de identificar os possíveis ganhos ou perdas, a bateria de testes para as várias versões da árvore de pesquisa binária inclui cenários distintos onde o número total de operações é de mil, dez mil e cem mil. No caso da estratégia de inserção, além dos já referidos, existem dois cenários adicionais, com um milhão e dez milhões de operações. Para reduzir possíveis perturbações nos testes, cada cenário foi executado dez vezes e as métricas apresentadas são a média dessas execuções. Os resultados foram medidos em termos de tempo de execução, S*peedup* e Eficiência.

Define-se o *Speedup S=Ts/Tp* (Ganho de Velocidade) como uma métrica para medir o desempenho de sistemas paralelos. É igual ao quociente do tempo *Ts* de execução da versão sequencial pelo tempo *Tp* da versão paralela. Tem como máximo teórico o número de processadores (núcleos) utilizados na execução do programa. A eficiência é dada por *E=S/p* onde *p* é o número de processadores. Pode ser vista como a fração do tempo que os *p* processadores do sistema paralelo estão a realizar trabalho útil.

4.1 Inserção

Na estratégia de inserção, onde apenas novos elementos são inseridos na árvore, a versão base, ou sequencial (ST), apresentou um tempo de execução de 0,156s para dez mil operações. A árvore com o mecanismo RwLock Global (CGL), evidencia, naturalmente uma alta contenção, uma vez que cada operação obtém o bloqueio de escrita não

permitindo que mais nenhuma tarefa progrida, não obtendo melhorias no desempenho. Por outro lado, a árvore com o mecanismo com Mutex local (FGL) demonstra que se obtêm ganhos de desempenho ao aumentar o número de tarefas, porém, a partir de quatro tarefas, observa-se uma degradação do desempenho. Nesta estratégia, a árvore sem trincos (AT) demonstrou a maior adaptabilidade, observando-se os maiores ganhos.

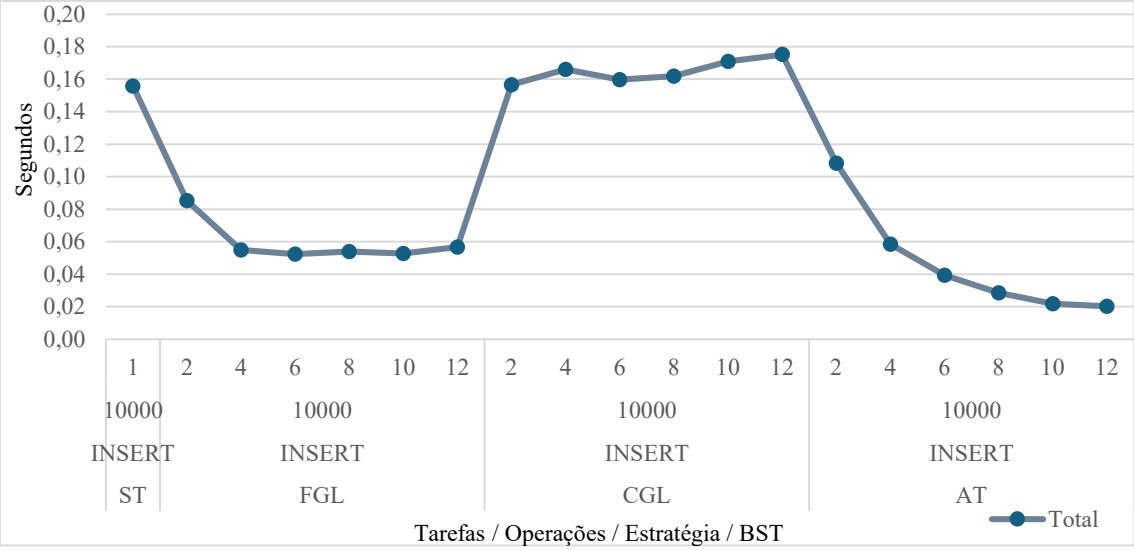

Figura 4.1. Tempos de execução, inserção, 10.000 operações

Obtidos os resultados dos tempos de execução, calculou-se tanto o *speedup* como a eficiência e, como esperado, a versão com recurso a operações atómicas é capaz de suster a eficiência com o aumento do número de tarefas. Por outro lado, as versões com mecanismos com recurso a bloqueios demonstram a contenção através da baixa eficiência e *speedup*.

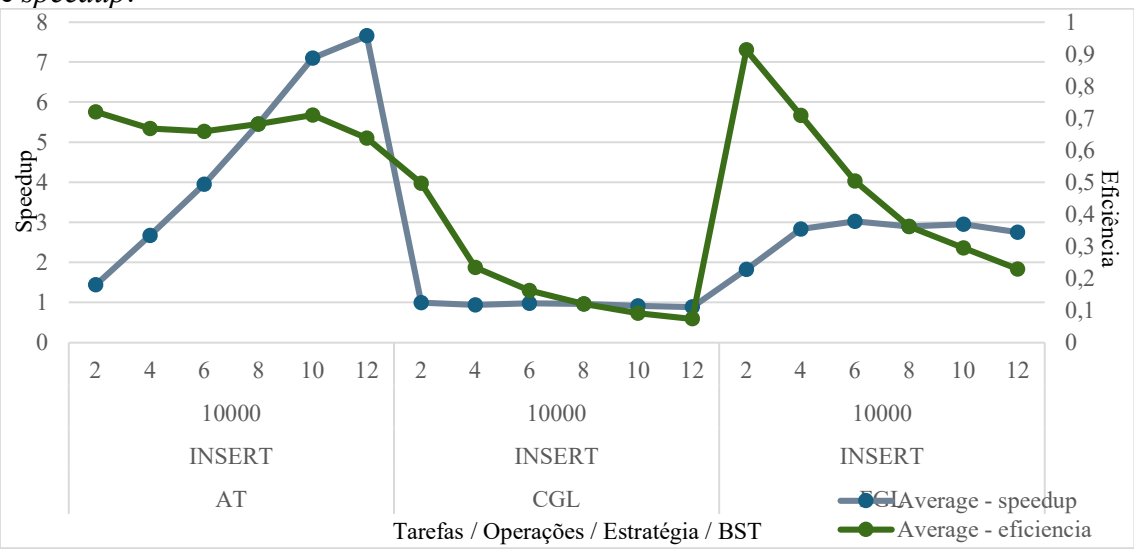

Figura 4.2. Speedup e eficiência, inserção, 10.000 operações

4.2 Escrita

Nesta estratégia, que envolve a inserção e remoção de elementos da árvore, os resultados evidenciam o mesmo comportamento em todas as versões da estratégia anterior. Assim, verifica-se que o mecanismo de controlo de concorrência que melhor se adapta a cenários com operações mistas, é o controlo de concorrência sem trincos (AT).

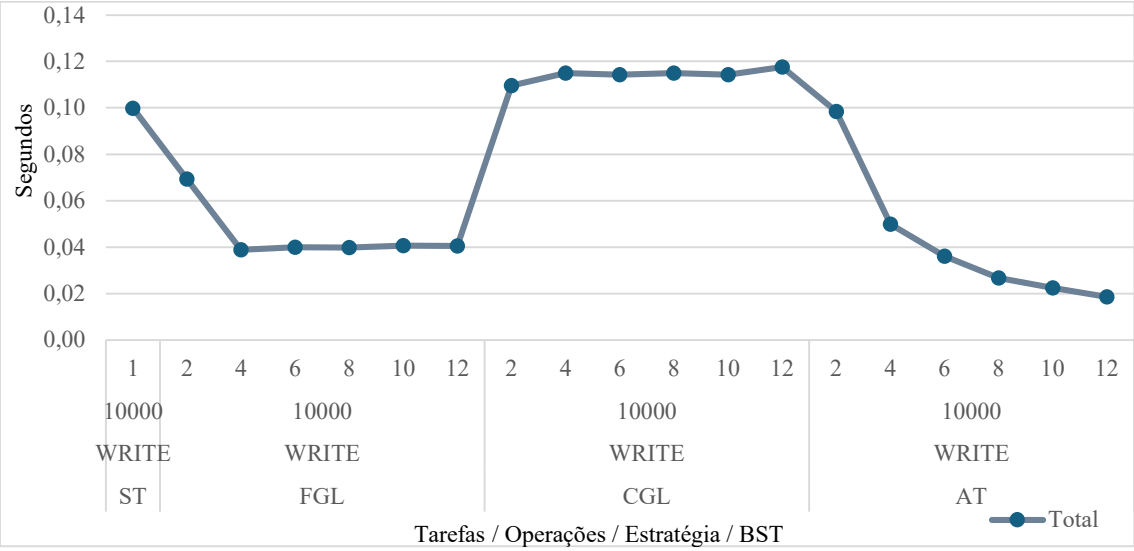

Figura 4.3. Tempos de execução, escrita, 10.000 operações

No entanto, foi observado que em cenários com um maior número de operações, as falhas das instruções CAS - *Compare-and-Set*, bem como o custo que o algoritmo incorre, tomam relevância, demonstrando que a versão com bloqueios locais se torna a opção mais viável neste cenário.

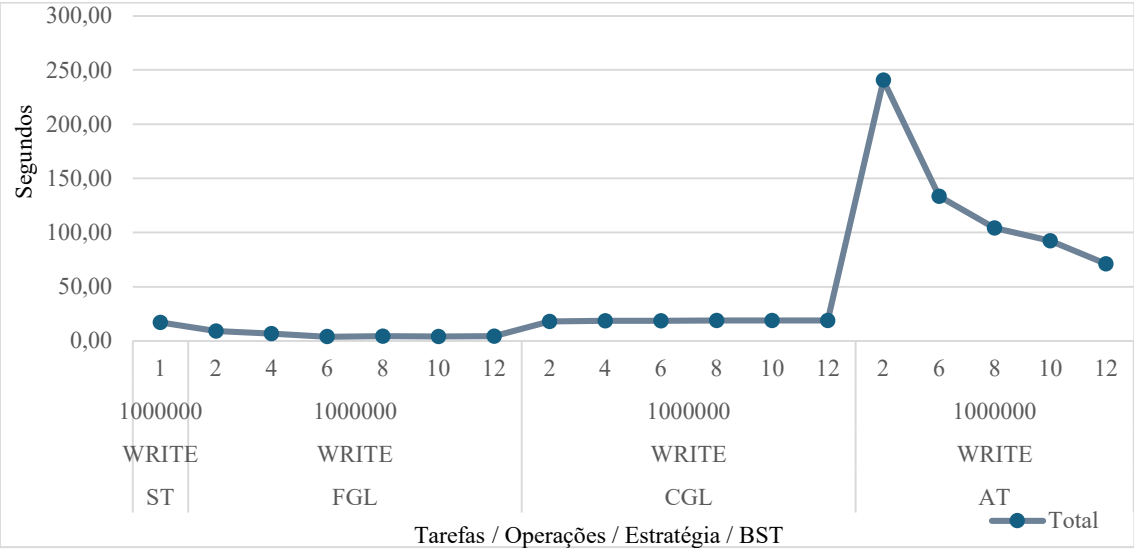

Figura 4.4. Tempos de execução, escrita, 1.000.000 operações

No entanto, e apesar da versão com bloqueios locais se adaptar melhor a cenários com um alto volume de operações, tanto o *speedup* como a eficiência são baixos.

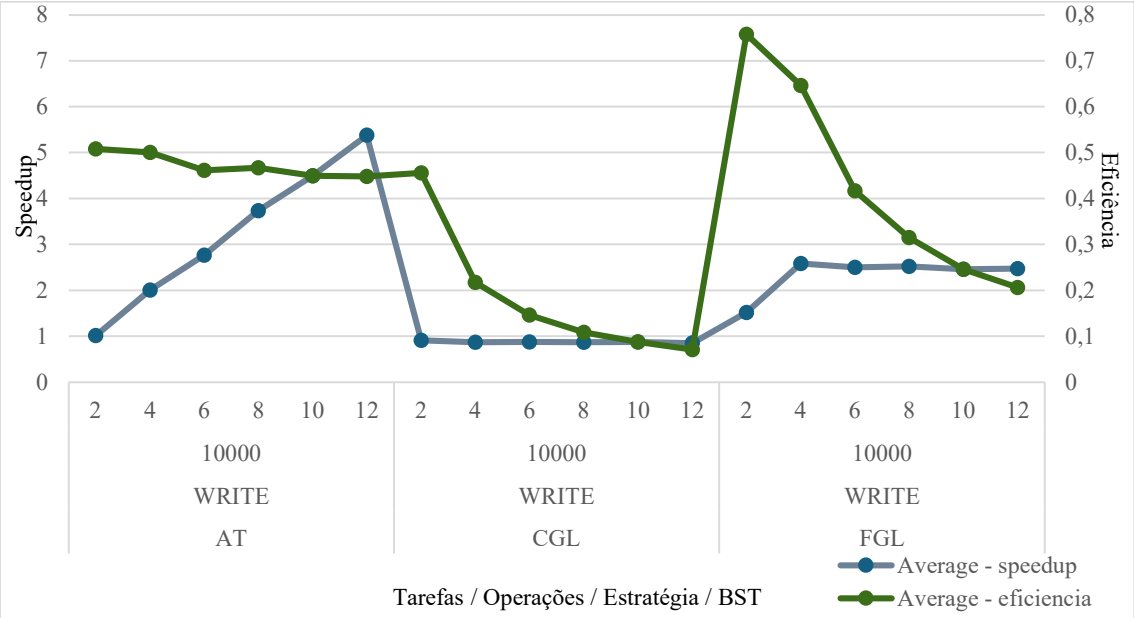

Figura 4.5. Speedup e eficiência, escrita, 10.000 operações

4.3 Leitura

Para a estratégia de leitura, a árvore com o mecanismo com Mutex local (FGL) foi a mais penalizada devido à sobrecarga de bloquear e desbloquear cada elemento no ramo e à própria contenção originada pelos diversos bloqueios. Este comportamento foi observado em todas as escalas de operações.

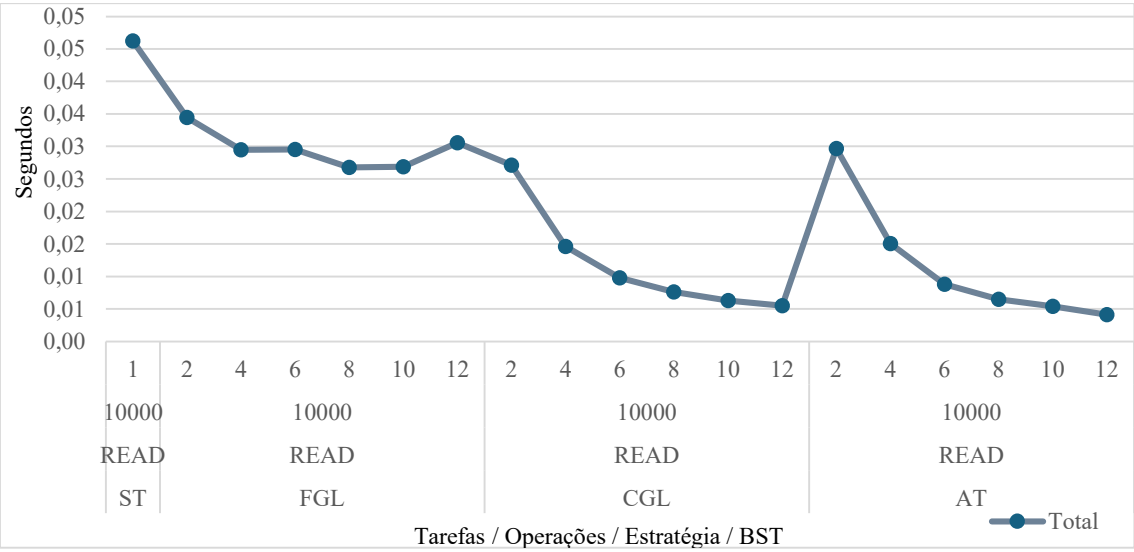

Figura 4.6. Tempos de execução, leitura, 10.000 operações

Assim, tanto o *speedup* como a eficiência foram penalizados para a versão com recurso a bloqueio locais.

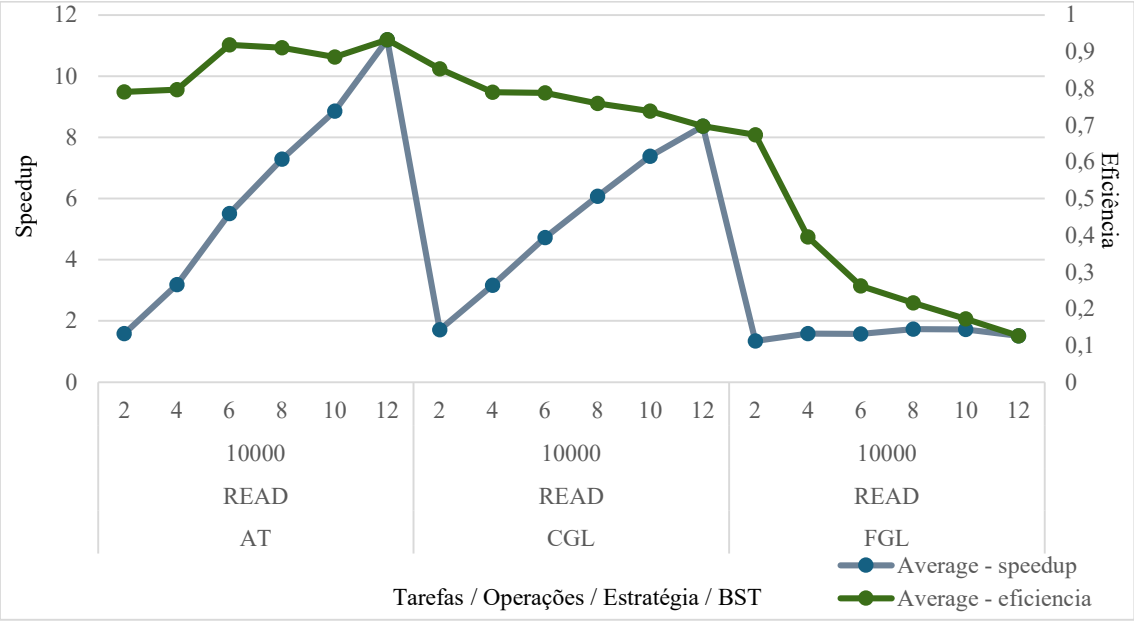

Figura 4.7. Speedup e eficiência, leitura, 10.000 operações

4.4 Leitura e Escrita

Na estratégia de leitura e escrita, verificou-se o mesmo comportamento que nas estratégias de inserção e escrita, onde a versão atómica aparenta ser a que melhor se adapta, no entanto, em cenários com um maior volume de operações, o custo das falhas da instrução *Compare-and-Set* tornam-se relevantes, degradando a eficiência.

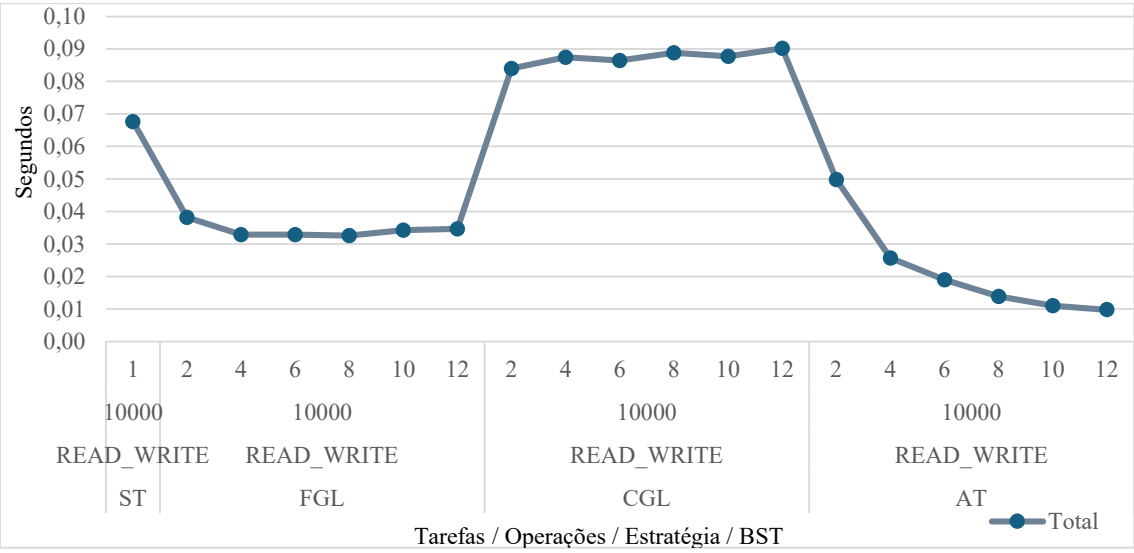

Figura 4.8. Tempos de execução, leitura e escrita, 10.000 operações

Nesta estratégia, este custo é ainda mais evidente, chegando a ter um desempenho inferior à versão com um bloqueio global.

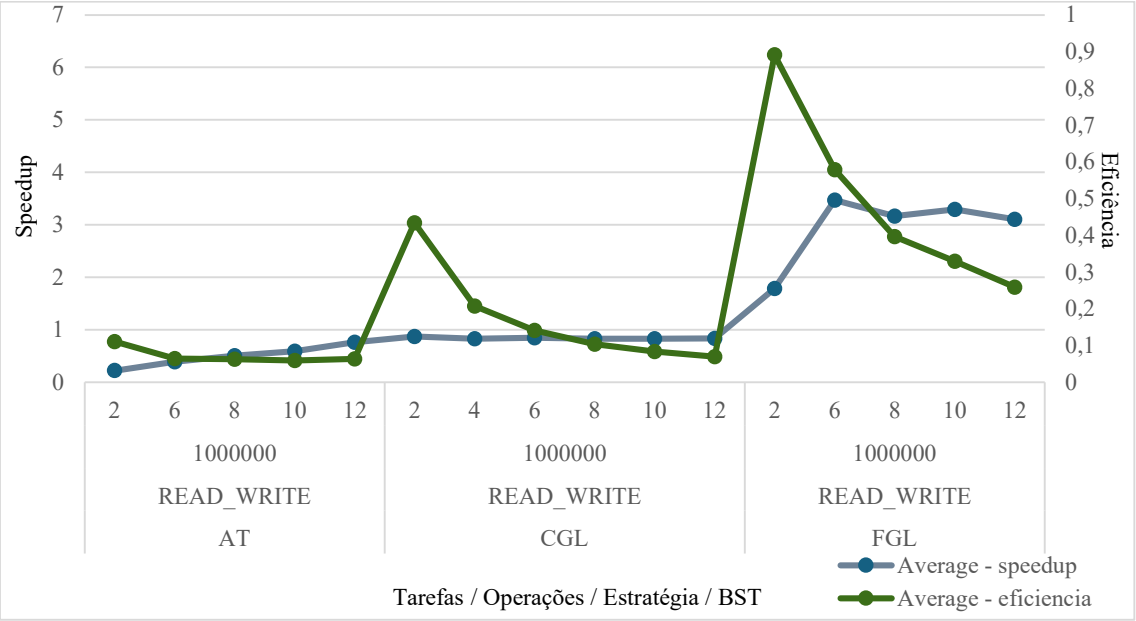

Figura 4.9. Speedup e eficiência, leitura e escrita, 1.000.000 operações

5. Conclusões

Ao longo deste estudo, foi possível implementar e avaliar diferentes abordagens de controlo de concorrência, tendo os resultados mostrado variações significativas no desempenho consoante o método utilizado e o tipo de estratégia de teste (ou utilização) da árvore.

Na estratégia de inserção, a árvore com a versão do mecanismo de controlo de concorrência com bloqueio global (C*oarse-Grained Lock)* apresentou um desempenho inferior às restantes, devido à utilização de um único bloqueio global, originando contenção logo a partir de duas tarefas. Este fenómeno é exacerbado pelo aumento do número de tarefas. Por outro lado, a versão com bloqueio local (*Fine-Grained Lock*) apresenta melhorias significativas de desempenho, no entanto é ultrapassada pela versão Atómica sem trincos, onde, a eficiência aumenta consoante o número de tarefas aumenta.

Na estratégia de escrita, à semelhança da estratégia de inserção, versão do mecanismo de controlo de concorrência com bloqueio global *(Coarse-Grained Lock)* apresentou o pior desempenho, pelas mesmas razões. No entanto, em contraste com a estratégia anterior, a versão com bloqueio local (*Fine-Grained Lock*) destaca-se em relação à versão Atómica sem trincos em todos os testes feitos, atribuindo-se este facto às falhas das operações *Compare-and-Set* (CAS), que se tornam mais relevantes quando existem remoções de valores da árvore, obrigando a percorrer novamente a árvore na inserção.

Na estratégia de leitura, em oposto às anteriores, a versão com bloqueio local (*Fine-Grained Lock*) apresenta o pior desempenho, devido à sobrecarga de bloqueio e

desbloqueio dos diversos nós da árvore. Operações de leitura como o cálculo do valor mínimo ou valor máximo, necessitam de bloquear e desbloquear todos os nós da árvore. Já a versão com bloqueio global (C*oarse-Grained Lock)*, ao utilizar apenas um RwLock Global, cada operação de leitura apenas necessita de efetuar um único bloqueio de leitura por operação. Por outro lado, a versão Atómica sem trincos destacou-se em relação às restantes, oferecendo tempos inferiores a um segundo nesta estratégia, para cem mil operações, por não utilizar bloqueios nem existir falhas de operações de CAS.

Por fim, a estratégia de leitura e escrita apresenta valores que revelam os pontos fortes e fracos de cada tipo de gestão de concorrência, com semelhanças de comportamento com a estratégia de escrita, no entanto, a partir de um milhão de operações, as falhas da instrução CAS penalizam a execução de tal forma que as restantes versões conseguem completar os cenários de teste em metade do tempo.

Apesar dos resultados promissores, este estudo apresenta certas limitações que devem ser consideradas, nomeadamente o ambiente de testes, tendo sido os mesmos realizados utilizando o mesmo CPU (AMD Ryzen 9 3950x), a otimização especifica na compilação (gcc -O1), podendo outras otimizações ou outras arquiteturas afetar os resultados apresentados. Por outro lado, a simulação da carga nas operações de comparação de chaves, através da utilização de instruções artificiais de cópia, pode não refletir com precisão os cenários reais de uma aplicação. Por fim, e embora o estudo tenha explorado diferentes ordens de grandeza de operações e tarefas, outros cenários distintos podem revelar outras limitações ou comportamentos não observados.

De uma forma geral e com base nas conclusões e limitações identificadas, é possível sugerir melhorias a serem estudadas e implementadas de futuro como otimização dos bloqueios, gestão de memória ou algoritmos mais eficientes de leitura. A otimização dos bloqueios prende-se com a possível utilização de algoritmos híbridos, onde tanto bloqueios locais como técnicas sem trincos (*Lock-free*) podem ser usadas para melhorar ainda mais a eficiência e reduzir a contenção. A otimização da gestão de memória é mais focada na versão atómica, onde estudar os ganhos ou perdas de diferentes esquemas de reclamação de memória, podem originar métricas mais concretas. Na versão atual foi utilizado o esquema de *Hazard Pointers* [Michael, 2004], no entanto existem outras técnicas como *Epoch-Based Reclamation* [Brown, 2017].

REFERÊNCIAS

Agrawal, V., Peraza, L., Lock-free Binary Search Trees. (2024). Github.io. https://vasuagrawal.github.io/418FinalProject/

AVL tree. (2021, June 4). Wikipedia. https://en.wikipedia.org/wiki/AVL_tree

Brown, T. (2017). Reclaiming memory for lock-free data structures: there has to be a better way. ArXiv (Cornell University). https://doi.org/10.48550/arxiv.1712.01044

CS 111 Spring 2005. (2024). Harvard.edu. https://read.seas.harvard.edu/~kohler/class/cs111-s05/notes/notes8.html

Durstenfeld, R. (1964). Algorithm 235: Random permutation. Communications of the ACM, 7(7), 420. https://doi.org/10.1145/364520.364540

Eberl, M. (2024). The Fisher-Yates shuffle. https://www.isa-afp.org/browser_info/current/AFP/Fisher_Yates/outline.pdf

Gonçalves, H., 2024, https://github.com/goncah/bst

Gupta, K., & Sharma, T. (2021). Changing Trends in Computer Architecture : A Comprehensive Analysis of ARM and x86 Processors. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 7(3), 619–631. https://doi.org/10.32628/cseit2173188

Knuth, Donald E. (1969). Seminumerical algorithms. The Art of Computer Programming. Vol. 2. Reading, MA: Addison–Wesley. OCLC 85975465.

Michael, M. M. (2004). Hazard pointers: safe memory reclamation for lock-free objects. IEEE Transactions on Parallel and Distributed Systems, 15(6), 491–504. https://doi.org/10.1109/tpds.2004.8

Natarajan, A., & Mittal, N. (2014). Fast concurrent lock-free binary search trees. https://doi.org/10.1145/2555243.2555256

Nguyen, C. (2023, November 20). The History & Evolution of CPUs | WinC Services. WinC Services. https://www.wincservices.com/blog/the-history-evolution-of-cpus/

Parallel Computer Architecture and Programming CMU 15-418/15-618, Spring 2018. (n.d.). Retrieved November 22, 2024, from https://www.cs.cmu.edu/afs/cs/academic/class/15418-f18/www/lectures/17_lockfree.pdf

Pimpale, S., Kudtarkar, R., Concurrent Lock-free Binary Search Tree. (2015). Github.io. https://swapnil-pimpale.github.io/lock-free-BST/proposal.html

pthreads(7): POSIX threads - Linux man page. (2024). Die.net. https://linux.die.net/man/7/pthreads

rand_r(3): pseudo-random number generator - Linux man page. (2024). Die.net. https://linux.die.net/man/3/rand_r

Schwarzkopf, M. (2022). CSCI 0300/1310: Fundamentals of Computer Systems. Brown.edu. https://cs.brown.edu/courses/csci0300/2022/notes/l21.html

Standard library header - cppreference.com. (2022). Cppreference.com. https://en.cppreference.com/w/cpp/header/stdatomic.h

Threads. (n.d.). Www.di.ubi.pt. https://www.di.ubi.pt/~operativos/praticos/html/9-threads.html

usleep(3) - Linux man page. (2024). Die.net. https://linux.die.net/man/3/usleep

Wikipedia Contributors. (2019, May 18). Red–black tree. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

Wikipedia Contributors. (2020, January 13). Binary search tree. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Binary_search_tree

Hugo Gonçalves, licenciado em Engenharia Informática pela Universidade Aberta em 2024, atualmente mestrando do curso de Mestrado em Engenharia Informática e Tecnologia Web da mesma instituição, em parceria com a Universidade de Trás-os-Montes e Alto Douro. Iniciou o seu percurso profissional no ano de 2012 no ramo das telecomunicações tendo vindo a especializar-se no desenvolvimento de software com arquiteturas distribuídas. Tem como áreas de interesse, a computação de alto desempenho e os sistemas distribuídos.

Paulo Shirley, Professor Auxiliar no Departamento de Ciências e Tecnologia (DCeT), Secção de Informática, Física e Tecnologia (SIFT). Licenciado em Engenharia Eletrotécnica e de Computadores em 1988 pelo IST-UTL. Obteve os graus de Mestre (perfil de Controlo e Robótica) e de Doutor em Eng. Eletrotécnica e de Computadores, pelo IST-UTL em 1993 e 2003 respetivamente. Tem como áreas de interesse, a intersecção da Informática (Computer Science) com a área do Controlo Automático, nomeadamente a área da "Computação de Alto Desempenho" (HPC) aplicada a problemas de otimização e cálculo científico.