Análise de Desempenho de *Hash Tables* Não-Bloqueantes na Linguagem C++

Douglas Pereira Luiz[1], **Odorico M. Mendizabal**[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brazil

douglas.pereira@grad.ufsc.br, odorico.mendizabal@ufsc.br

Resumo. *Hash Tables são estruturas de dados que associam chaves de busca a valores e são amplamente utilizadas no desenvolvimento de sistemas. Quando seu uso exigir compartilhamento entre threads podem ser implementadas com travas como método de sincronização ou de maneira não-bloqueante. Neste trabalho foram reunidas e comparadas implementações de hash tables bloqueantes e não-bloqueantes para a linguagem C++.*

1. Introdução

Em sistemas distribuídos de alta vazão, dos quais se deseja baixa latência, além de implementar algoritmos distribuídos eficientes, é importante garantir alto desempenho intranodo e aproveitar do paralelismo disponível localmente. A escolha das estruturas de dados tem impacto nesse desempenho, e construir essas estruturas com base em algoritmos não-bloqueantes ao invés de usar travas como mecanismo de sincronização pode evitar esperas durante as operações [Herlihy and Shavit 2012].

Uma estrutura de dados é bloqueante quando a sincronização é feita utilizando travas para prover exclusão mútua. Estruturas de dados compartilhadas construídas sem travas, com métodos *wait-free* ou *lock-free*, podem ser mais adequadas a um ambiente concorrente [Herlihy and Shavit 2012].

Muitas aplicações requerem conjuntos dinâmicos que suportem as operações de inserção, busca e remoção, como bancos de dados chave-valor, sistemas de cache distribuída e serviços para sincronização e coordenação distribuída. *Hash tables* implementam de forma eficaz essas operações [Cormen et al. 2001]. Em vista disso, esse trabalho tem como objetivo avaliar o desempenho de estruturas *hash table*, comparando soluções bloqueantes e não-bloqueantes presentes em bibliotecas de programação para C++.

2. Metodologia

Foram realizados testes com a reprodução de operações de busca, inserção e remoção sobre diferentes implementações de *hash table*. A biblioteca padrão *Unordered Map* foi usada como referência. Nesta implementação, que chamamos *LockUnordered*, há a sincronização entre operações com o uso de *mutex*. Para comparação, também foram utilizadas as bibliotecas: *Xenium*, que implementa uma *hash table lock-free* baseada na proposta de Michael [Michael 2002]; *LibCDS*, com *hash tables lock-free* construídas a partir das propostas de Feldman [Feldman et al. 2013] e Michael [Michael 2002]; *Wait-FreeCollections*, que contém uma implementação de *hash table wait-free* como descrita por Laborde, Feldman e Dechev [Laborde et al. 2017]; *TBB*, uma biblioteca mantida pela

Intel para auxiliar em paralelização de código que implementa um *hash map* bloqueante de granularidade fina com alto desempenho.

Os testes consistiram em obter o tempo para a execução de operações de busca, inserção e remoção sobre as *hash tables* em diferentes cenários. Em cada cenário foi contabilizado o tempo levado para a realização de 10^7 operações sobre cada implementação. Em cada cenário variamos: o número de *threads* realizando as operações (4, 8, 16 ou 32); o número de chaves distintas (1000 ou 10^6); o tamanho dos valores armazenados (4 bytes ou 4 kbytes); inserção prévia ou não de entradas para cada chave distinta; a distribuição das operações de busca, inserção e remoção. As chaves e valores de 4 bytes são do tipo *uint32_t* e os valores de 4k bytes são vetores de 1024 posições desse mesmo tipo.

O conjunto de cenários de teste foi composto por todas as combinações possíveis e foram realizadas 20 repetições de cada cenário. As possíveis distribuições de operações foram: 90% de buscas, 5% de inserções e 5% de remoções; 45% de buscas, 45% de inserções e 10% de remoções; 5% de buscas, 90% de inserções e 5% de remoções. A primeira distribuição é intensa em buscas, a segunda equilibra as buscas e inserções, e a terceira é intensa em inserções.

A implementação *MichaelHashMap* da LibCDS foi configurada com 4 elementos por *bucket* e estimativa do total de elementos igual ao total de chaves distintas. A implementação *harris_michael_hashmap* da Xenium foi configurada com 500 *buckets* para 1000 chaves distintas, e 500000 *buckets* para 10^6 chaves distintas. A implementação *FeldmanHashMap* da LibCDS foi configurada com 8 *head_bits* e 4 *array_bits*.

2.1. Ambiente de Teste

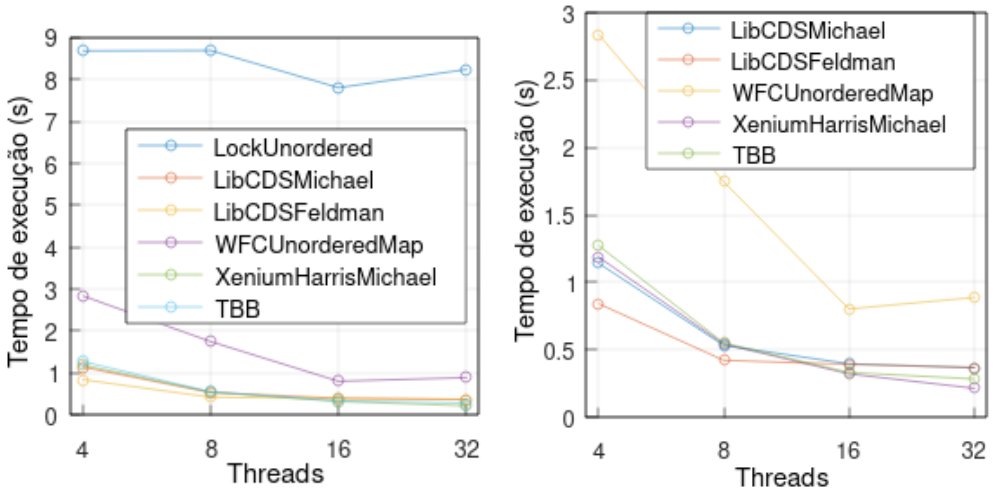
Os testes foram realizados em um computador com quatro processadores Intel Xeon E5-4620 2.2 GHz com 8 núcleos e 16 MB cache e 128 GB de memória RAM DDR3. O Sistema operacional usado foi o Ubuntu v20.04 de 64 bits. O programa de testes foi desenvolvido na linguagem C++17 e compilado com o *gcc v9.4.0* com parâmetro de compilação `-O3`. O parâmetro `-mcx16` foi usado para as bibliotecas que necessitam da instrução *CAS*.

3. Resultados

Os gráficos a seguir apresentam o tempo de execução em segundos, no eixo y, e o número de *threads*, no eixo x. São exibidos os resultados para cada implementação.

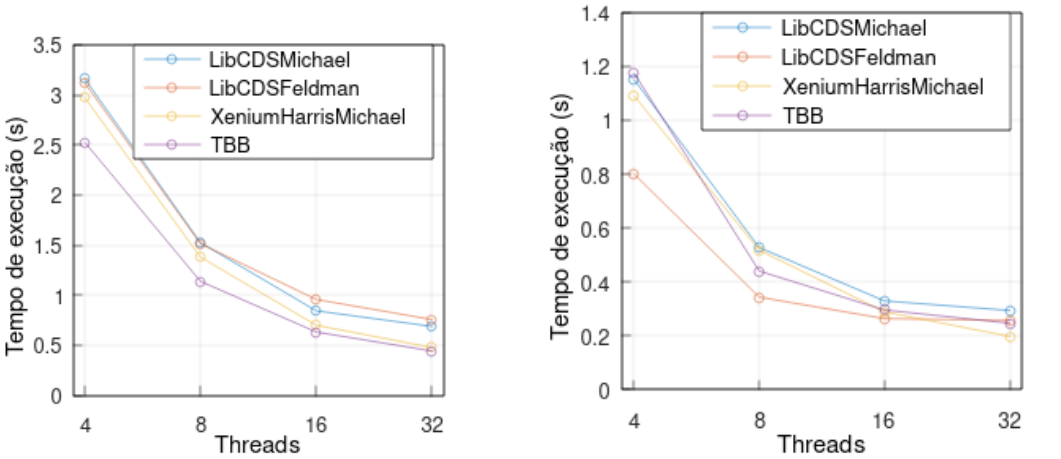
Na Figura 1(a), o custo ao empregar *mutex* com a implementação *LockUnordered* é visível. Este perfil de menor desempenho se mantém em todos os testes. A Figura 1(b) contém os mesmos dados da Figura 1(a), omitindo a implementação *LockUnordered*. É possível observar que, em média, para armazenamento de valores de 4 bytes, algumas das implementações não bloqueantes tiveram um desempenho maior. A implementação da WFC sofre com falhas de segmentação durante inserções quando a quantidade de operações é muito grande, por isso não aparece nas demais figuras.

A Figura 2(a) indica que o desempenho da implementação da TBB foi maior quando os valores armazenados eram de 4 kbytes. Além disso, em todas as implementações, o aumento de concorrência com o acréscimo de *threads* possibilitou uma redução no tempo total de execução.



(a) com *LockUnordered*. (b) sem *LockUnordered*.

Figure 1. Tempo médio de execução combinando os cenários com valores de 4 bytes.



(a) com valores de 4 kbytes. (b) com 90% buscas, 5% inserções e 5% remoções.

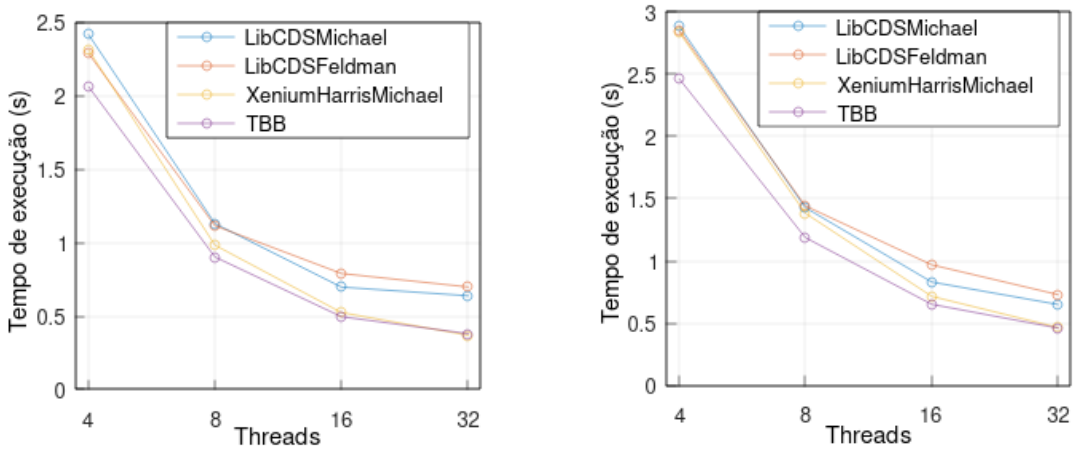
Figure 2. Tempo médio de execução combinando os cenários.

As Figuras 2(b), 3(a) e 3(b) dizem respeito aos resultados obtidos dadas diferentes distribuições das operações de busca, inserção e remoção. A Figura 2(b) mostra que o desempenho médio nos testes intensivos em busca foi maior para as implementações não-bloqueantes da LibCDS e Xenium. As Figuras 3(a) e 3(b) mostram que a implementação da TBB apresentou menores tempos de execução em testes mais intensos em escritas.

A Tabela 1 mostra que a implementação da Xenium apresentou fatores de redução de tempo de execução maiores em relação à implementação *baseline* nos cenários com 16 e 32 *threads*. A implementação baseada na proposta de Feldman da LibCDS obteve os melhores fatores nos cenários com 4 e 8 *threads*.

4. Conclusão

Os testes revelaram cenários que favoreceram implementações específicas. As implementações da LibCDS e da Xenium se saíram melhor quando os valores armazenados eram de poucos bytes ou as operações eram predominantemente buscas. A TBB teve



(a) com 45% buscas, 45% inserções e 10% remoções. (b) com 5% buscas, 90% inserções e 5% remoções.

Figure 3. Tempo médio de execução.

Table 1. Média dos tempos de execução de todos os testes com N *threads* da LockUnordered dividido pela média dos tempos de execução de todos os testes com N *threads* da implementação X.

X \ N threads	4 threads	8 threads	16 threads	32 threads
TBB	6,82374	15,8245	23,5821	29,0221
LibCDS Michael	7,61399	16,3424	19,7643	22,4641
LibCDS Feldman	10,3311	20,6593	19,9587	22,45
Xenium Harris Michael	7,31599	16,0414	24,3294	38,5973

os menores tempos de execução quando os valores armazenados eram de 4 kbytes ou grande parte das operações eram inserções.

As implementações da Xenium e da LibCDS baseadas na proposta de Michael [Michael 2002] conseguiram competir com a sofisticada implementação bloqueante da TBB, mas foi necessário ter uma estimativa da quantidade de elementos na *hash table*. Ainda assim os testes indicam que implementações não-bloqueantes oferecem melhora no desempenho de aplicações que utilizam *hash tables*.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. The MIT Press, 2nd edition.

Feldman, S., LaBorde, P., and Dechev, D. (2013). Concurrent multi-level arrays: Wait-free extensible hash maps.

Herlihy, M. and Shavit, N. (2012). *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., 1st edition.

Laborde, P., Feldman, S., and Dechev, D. (2017). A wait-free hash map. *Int. J. Parallel Program.*, 45(3):421–448.

Michael, M. M. (2002). High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 73–82.