UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Douglas Pereira Luiz

# Estudo de Estruturas de Dados Não Bloqueantes e Análise de Impacto do seu uso sobre uma Aplicação Paralela

Florianópolis
2023

Douglas Pereira Luiz

# Estudo de Estruturas de Dados Não Bloqueantes e Análise de Impacto do seu uso sobre uma Aplicação Paralela

Trabalho de Conclusão do Curso de Graduação em Ciências da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Ciências da Computação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.

Florianópolis
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Luiz, Douglas Pereira
Estudo de Estruturas de Dados Não Bloqueantes e Análise de Impacto do seu uso sobre uma Aplicação Paralela / Douglas Pereira Luiz ; orientador, Odorico Machado Mendizabal, 2023.
115 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2023.

Inclui referências.

1. Ciências da Computação. 2. Estruturas de dados. 3. Computação Paralela. 4. Sincronização não bloqueante. 5. Sistemas de alta vazão. I. Mendizabal, Odorico Machado. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

Douglas Pereira Luiz

**Estudo de Estruturas de Dados Não Bloqueantes e Análise de Impacto do seu uso sobre uma Aplicação Paralela**

O presente trabalho em nível de bacharelado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Alexandre Goncalves Silva, Dr.

Prof. Pedro Belin Castellucci, Dr.

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Bacharel em Ciências da Computação.

---

Profa. Lúcia Helena Martins Pacheco, Dr.
Coordenadora do Curso de Graduação

---

Prof. Odorico Machado Mendizabal, Dr.
Orientador

Florianópolis, 2023.

## RESUMO

Usufruir da capacidade de processar requisições simultaneamente ou paralelamente é uma forma comum de aumentar o desempenho computacional. A programação concorrente, entretanto, impõe desafios gerados pelo impacto de diferentes unidades de processamento trabalharem com dados sobre um mesmo espaço de memória. Para garantir o progresso de um sistema concorrente a estados consistentes é prático definir regiões críticas e utilizar de mecanismos de exclusão mútua. Estruturas de dados preparadas para acesso concorrente podem ser projetadas utilizando desse método, fazendo com que determinada área da estrutura não tenha acessos simultâneos, permitindo uma aquisição da área por um processo, e colocando os outros processos em espera. Essa forma de lidar com os acessos concorrentes pode inibir o paralelismo e colocar em espera processos por tempo indeterminado. Apesar disso, estruturas de dados podem ser construídas com garantias quanto ao não-bloqueio e progresso. Uma estrutura *lock-free* garante o progresso de ao menos um processo em um número finito de passos na realização de operações sobre a estrutura; e em uma estrutura *wait-free* o progresso em um número finito de passos é garantido para todos os processos. Essas garantias são altamente desejadas em sistemas distribuídos, que aproveitam do poder de processamento de nós distribuídos em uma rede para fornecer garantias como disponibilidade, escalabilidade e concorrência. Porém estruturas eficientes com essas propriedades são difíceis de construir. Neste trabalho, foram estudadas construções de estruturas que fornecem essas garantias e avaliou o impacto de seu uso em uma aplicação paralela. Para a análise de desempenho, foram conduzidas simulações e testes em uma aplicação modificada, adaptada para incorporar essas estruturas de dados. Antes desse processo, também foi verificada a disponibilidade de implementações em bibliotecas de programação existentes, as quais oferecem ferramentas para a construção de ambientes com tais características.



**Palavras-chave**: Estruturas de dados. Computação paralela. Sincronização não bloqueante. Sistemas de alta vazão.

## ABSTRACT

Benefiting from the capacity to process requests simultaneously or in parallel is a common way to increase computational performance. Concurrent programming, however, presents challenges generated by the impact of different processing units working with data in the same memory space. To ensure the progress of a concurrent system towards consistent states, it is practical to define critical sections and use mutual exclusion mechanisms. Data structures prepared for concurrent access can be designed using this method, so that a certain area of the structure does not have simultaneous accesses, allowing one process to acquire the area and putting the other processes on hold. This way of dealing with concurrent accesses can inhibit parallelism make processes wait indefinitely. Despite this, data structures can be built with non-blocking progress guarantees. A lock-free structure guarantees the progress of at least one process in a finite number of steps in performing operations on the structure; and in a wait-free structure, progress in a finite number of steps is guaranteed for all processes. These guarantees are highly desirable in distributed systems, which take advantage of the processing power of distributed nodes in a network to provide guarantees such as availability, scalability, and concurrency. However, efficient structures with these properties are difficult to construct. The aim of this work is to study and analyze the construction of structures that provide these guarantees and evaluate the impact on the system when used in distributed algorithms. For performance analysis, simulations and tests were conducted on a modified application, adapted to incorporate these data structures. Before this process, the availability of implementations in existing programming libraries, which provide tools for building environments with such characteristics, was also verified.



**Keywords**: Data-structures. Parallel computing. Non-blocking synchronization. High-throughput systems.

## LISTA DE FIGURAS

**SUMÁRIO**

# 1 INTRODUÇÃO

Estruturas de dados podem ter um *design* simples quando não se tem o intuito de envolvê-las em operações concorrentes, mas é comum que sistemas aproveitem do paralelismo disponível, que impõe a necessidade de cuidados com acesso à memória compartilhada. As modificações necessárias para tornar uma estrutura de dados segura para uso compartilhado tradicionalmente são feitas sem grande aumento de complexidade dos algoritmos, delimitando seções críticas e impedindo acesso concorrente com mecanismos de exclusão mútua (HERLIHY, M.; SHAVIT, 2012).

Estruturas de dados desempenham um importante papel na construção de sistemas paralelos e de alta vazão. Filas podem ser usadas para modelar filas de mensagens a serem processadas (COULOURIS *et al.*, 2013). Grafos podem ser utilizados para modelar relações de dependência de dados ou de tarefas, sendo empregados no projeto de escalonadores que objetivam minimizar o tempo de ociosidade das unidades de processamento (CANON *et al.*, 2008). Algoritmos de particionamento de grafos podem ser utilizados para predição de um bom particionamento para requisições futuras, a fim de maximizar o paralelismo no processamento de requisições (TROMBETA, J. G., 2021). Tabelas de dispersão podem ser usadas em mecanismos de *cache* em memória para armazenar dados frequentemente usados (FITZPATRICK, 2004), ou em mecanismos de deduplicação para identificar dados duplicados e ajudar a minimizar a quantidade de dados armazenados (MANDAGERE *et al.*, 2008; XIA *et al.*, 2016).

Entretanto projetar essas estruturas de dados usando travas em seções críticas força pontos de serialização, que causam contenção. Ainda, é pouco adequado para sistemas assíncronos tolerantes a falhas. Além da existência de um tempo de espera que varia de acordo com a capacidade de processamento ou estado dos outros componentes do sistema, se um processo falho para em meio a uma seção crítica todos os processos serão impedidos de progredir (HERLIHY, M., 1991).

Para evitar tempos de espera indeterminados ou a parada total do sistema, é possível projetar estruturas de dados com garantias mais fortes quanto ao progresso. Estruturas *lock-free* garantem que na chamada concorrente por múltiplas *threads*, ao menos uma progride em um número finito de passos, enquanto estruturas *wait-free* garantem o progresso de todas as *threads* em um número finito de passos (HERLIHY, M.; SHAVIT, 2012). Estruturas eficientes que oferecem essas garantias não são fáceis de construir, sendo muitas vezes preferível utilizar uma implementação eficiente e bloqueante que apresenta poucas garantias a uma lenta *wait-free*.

É possível encontrar propostas de implementações não bloqueantes de filas (MICHAEL; SCOTT, 1996; LADAN-MOZES; SHAVIT, 2004; KOGAN; PETRANK, 2011), listas (HARRIS, 2001; MICHAEL, 2002a; FOMITCHEV; RUPPERT, 2004; TIMNAT *et al.*, 2012), *hash tables* (MICHAEL, 2002a; SHALEV; SHAVIT, 2006; FELDMAN, S.;

LABORDE; DECHEV, 2013; LABORDE; FELDMAN, S.; DECHEV, 2017), *skip lists* (SUNDELL; TSIGAS, 2005; FOMITCHEV; RUPPERT, 2004) e grafos (CHATTERJEE *et al.*, 2019; PERI; REDDY; SA, 2019).

Em vista disso, é de grande interesse da área de sistemas paralelos e distribuídos encontrar estruturas não bloqueantes e eficientes que garantem o progresso e tenham bom proveito do paralelismo disponível em cada nó. O intuito deste trabalho é investigar as existentes propostas de implementações de diferentes estruturas não bloqueantes e analisar o impacto no desempenho em simulação e em um ambiente distribuído real da substituição de estruturas bloqueantes por equivalentes que satisfazem *wait-freedom* ou *lock-freedom*.

## 1.1 OBJETIVOS

### 1.1.1 Objetivo Geral

Este trabalho tem como objetivo principal verificar e analisar as alterações de desempenho que estruturas de dados *lock-free* ou *wait-free* trazem a uma aplicação paralela.

### 1.1.2 Objetivos Específicos

a) Reunir e analisar trabalhos existentes com propostas de algoritmos para construção de estruturas de dados *lock-free* e *wait-free*;

b) Reunir implementações de estruturas de dados *lock-free* e *wait-free* disponibilizadas em bibliotecas de programação nas linguagem C e C++, e analisá-las quanto ao desempenho na realização de operações por meio de *microbenchmarks* com execução local, comparando-as entre si e com implementações bloqueantes;

c) Analisar o impacto no desempenho devido à implantação de estruturas de dados não bloqueantes em uma aplicação existente.

# 2 TRABALHOS RELACIONADOS

Esse capítulo apresenta trabalhos relacionados divididos em dois grupos. Primeiro, são apresentadas trabalhos que propõem algoritmos para implementação de diferentes estruturas de dados. Depois, são discutidas pesquisas que avaliam e comparam estruturas de dados concorrentes.

## 2.1 PROPOSTAS DE ESTRUTURAS DE DADOS NÃO BLOQUEANTES

Existem diversas propostas não bloqueantes de diferentes estruturas de dados. Uma generalização para implementação de estruturas de dados não bloqueantes foi apresentada por Herlihy em (HERLIHY, M., 1991). No entanto, estruturas de dados baseadas nesse esquema generalizado podem sofrer com pouca performance. Michael e Scott introduziram uma fila baseada em lista *lock-free* com alocação dinâmica de nós (MICHAEL; SCOTT, 1996), sendo considerada uma implementação eficiente e adotada na biblioteca padrão da linguagem Java e na biblioteca Boost C++. Ladan-Mozes e Shavit apresentam um fila baseada em uma lista duplamente ligada (LADAN-MOZES; SHAVIT, 2004). Mais tarde, Kogan e Petrank apresentam uma fila *wait-free* com um mecanismo de *helping* (KOGAN; PETRANK, 2011) baseada na proposta *lock-free* de Michael e Scott (MICHAEL; SCOTT, 1996).

Estruturas de dados não bloqueantes não se limitam a filas, e é possível encontrar implementações de estruturas mais complexas, como listas, *hash-tables*, grafos, *skip-lists* e árvores. Uma lista ligada não bloqueante foi apresentadas por *Harris* em (HARRIS, 2001), e foi base para o trabalho de Michael (MICHAEL, 2002a) que apresenta uma lista com suporte a mecanismos de gerenciamento de memória não bloqueantes, e uma *hash table lock-free* que não suporta incremento no número de *buckets*. Mais tarde, Shalev e Shavit apresentaram uma *hash-table lock-free* expansível (SHALEV; SHAVIT, 2006) que se baseia no uso de listas *lock-free*. O trabalho (TIMNAT *et al.*, 2012) apresentou uma lista *wait-free* com performance competitiva com implementações *lock-free* e bloqueantes. Laborde, Feldman e Dechev apresentam uma *hash-table wait-free* baseada em um esquema de *hashing* expansível (FELDMAN, S.; LABORDE; DECHEV, 2013; LABORDE; FELDMAN, S.; DECHEV, 2017).

Uma implementação de grafo *lock-free* foi apresentada em (CHATTERJEE *et al.*, 2019), baseado em listas com operações *lock-free* e suporta consultas de atingibilidade também *lock-free*. Em (PERI; REDDY; SA, 2019) foi apresentada uma implementação *wait-free* de grafo. Uma *skip list* não bloqueante foi apresentada por Sundell e Tsigas (SUNDELL; TSIGAS, 2005), e pouco depois por Formitchev e Ruppert (FOMITCHEV; RUPPERT, 2004).

## 2.2 ANÁLISE DE DESEMPENHO DE ESTRUTURAS DE DADOS CONCORRENTES

Diferentes trabalhos relacionados comparam estruturas de dados concorrentes. Em (XAVIER; MENDIZABAL, O. M., 2019), os autores realizam um estudo comparativo de técnicas de sincronização popularmente utilizadas na linguagem Java, comparando-as com a utilização da estrutura de dados concorrente *CopyOnWriteArrayList*. São explorados diferentes cenários de experimentação, alternando o grau de concorrência e o volume de dados manipulado pelas aplicações. Ainda é comparado o consumo de memória dos diferentes mecanismos de sincronização da linguagem. Os resultados do trabalho apontam a melhor utilização da *CopyOnWriteArrayList* em casos de maior incidência de operações leitura. Também é observado que o efeito da contenção de trava imposta por *Semaphores*, em cenários de maior concorrência, representa uma degradação de desempenho menor à utilização do monitor *Synchronized*.

O artigo (HUNT; SANDHU; CEZE, 2011) apresenta um estudo sobre a relação entre o desempenho e o consumo de energia de várias estruturas de dados *lock-free* com algoritmos baseados na primitiva *compare-and-swap*. São comparadas implementações de três estruturas de dados diferentes por meio da análise da execução de um conjunto de cargas de trabalho com alto nível de contenção. Os autores comparam o tempo de execução, potência de pico e consumo total de energia de cada implementação e explicam esses resultados com a ajuda de contadores de desempenho de *hardware*. Os resultados mostram que sob as cargas de trabalho usadas, as variantes não bloqueantes geralmente funcionam melhor e usam menos energia do que as implementações tradicionais com bloqueio.

Em (DUARTE *et al.*, 2019), são apresentadas sete implementações de *hash-tables* na linguagem Haskell, que usam desde mecanismos de sincronização de baixo nível a mecanismos de mais alto nível, como memória transacional. As *hash-tables* são comparadas usando diferentes tamanhos iniciais, *load-factors*, tipos de dados e funções de dispersão. Em um dos ambientes de teste, a implementação não bloqueante, implementada com base na proposta do trabalho (SHALEV; SHAVIT, 2006), esteve dentre as implementações com o melhor desempenho.

O trabalho (FELDMAN, S. D. *et al.*, 2013) apresenta a avaliação de desempenho de estruturas não bloqueantes por meio da refatoração de uma aplicação de deduplicação que usa mecanismos de sincronização convencionais utilizando uma *hash-table wait-free* e um conjunto de filas *lock-free* com o objetivo de aumentar o grau de paralelismo da aplicação. Os experimentos exploram as diferenças de desempenho de implementações que dependem de técnicas avançadas de exclusão mútua e sincronização não bloqueante. Foi observado que o impacto de desempenho das implementações de *hash table* supera as pequenas diferenças de desempenho causadas pela variação das implementações de filas, e que o uso de estruturas de dados não bloqueantes provocou a maior redução dos tempos de execução.

# 3 FUNDAMENTAÇÃO TEÓRICA

O propósito deste capítulo é fornecer uma base teórica por meio da apresentação de conceitos relacionados a estruturas de dados não bloqueantes. A revisão bibliográfica permitiu a identificação e seleção de trabalhos e autores que tratam do assunto. Este capítulo tem como objetivo apresentar e explicar os principais fundamentos dos assuntos relacionados ao projeto de pesquisa, a fim de fornecer ao leitor a compreensão e motivação da pesquisa e de sua importância.

## 3.1 CONCORRÊNCIA

A maioria dos sistemas operacionais modernos são multitarefa, fornecendo suporte para a execução aparentemente simultânea de vários programas. Isso é possível mesmo em um sistema com um único processador, uma vez que um processo pode ser executado por um pequeno intervalo de tempo e depois, mesmo que não tenha terminado, o sistema operacional pode executar um processo diferente (PACHECO; MALENSEK, 2021). Em sistemas com múltiplos processadores, como em multiprocessadores de memória compartilhada, é possível efetivamente executar tarefas diferentes simultaneamente, e as comunicações inter-processadores necessárias podem ser realizadas por meio de variáveis compartilhadas em memória (PATTERSON, David A.; HENNESSY, John L., 2020).

Esses recursos podem tornar a computação mais efetiva, e para usa-los adequadamente é necessário compreender os problemas relacionados à gestão dos múltiplos processos concorrentes e conhecer mecanismos capazes de facilitar o raciocínio sobre o projeto desse tipo de sistema. É importante ainda garantir bom uso dos multiprocessadores, que podem tornar a computação mais eficiente ao explorar o paralelismo e aproveitar de múltiplos processadores para trabalhar em uma única tarefa (HERLIHY, M.; SHAVIT, 2012).

### 3.1.1 *Threads*

Em um ambiente multitarefa, se um processo precisa esperar por um recurso, que pode ser, por exemplo, uma leitura de um dispositivo externo, ele *bloqueia*. Entretanto, muitos programas podem continuar realizando algum trabalho útil mesmo que a parte do programa que está sendo executada no momento tenha que esperar por um recurso. O mecanismo de *threads* permite dividir um programa em tarefas diferentes, com a propriedade de que, quando uma *thread* está bloqueada, outra pode ser executada concorrentemente (PACHECO; MALENSEK, 2021).

Além de concorrência, multiprocessadores permitem que *threads* sejam executadas por diferentes processadores. Por exemplo podemos somar dois vetores mais

rapidamente distribuindo as várias operações de soma para diferentes *threads*. A soma dos vetores no programa 3.1 poderia ser feita por dois processadores diferentes, cada um somando 5 elementos dos vetores, sem prejudicar o resultado.

Código 3.1 – Soma de vetores

```
1 int x[10], y[10];
2 ...
3 for (int i = 0; i < 10; i++){
4     x[i] += y[i];
5 }
```

Fonte: Elaborado pelo autor (2023).

### 3.1.2 Condições de corrida

*Threads* ou processos podem tentar acessar simultaneamente um recurso compartilhado. Esses acessos podem resultar em um erro conhecido como condição de corrida (*race condition*) (PACHECO; MALENSEK, 2021), porque as *threads* estão em uma "corrida" para executar uma operação. O Código 3.2 pode apresentar esse comportamento se for executado por mais de uma *thread*. A variável *x* é compartilhada entre as *threads* e foi inicializada com o valor 0.

Código 3.2 – Incrementar *x* com condição de corrida

```
1 int x = 0;
2 void incrementar(){
3     x += 1;
4 }
```

Fonte: Elaborado pelo autor (2023).

No código 3.2, uma invocação do procedimento *incrementar* incrementará *x* em 1. Se *incrementar* for invocado por mais de uma *thread*, pode-se esperar que ao final da execução o valor de *x* tenha valor igual ao número de invocações. O problema surge uma vez que a operação de incremento é dividida em mais de uma operação. É necessário primeiro carregar o valor da memória, realizar a operação e depois gravá-lo na memória novamente. A Figura 1 apresenta possíveis ordens de computação com duas *threads* que resultam em valores diferentes gravados em *x*. A execução A finaliza com o valor de *x* igual à 1, enquanto a execução B termina com o valor de *x* esperado.

### 3.1.3 Travas

Uma forma de evitar as condições de corrida é sincronizando a operação de incremento de forma que só uma *thread* possa estar fazendo todos os passos dessa operação em um dado momento. Pode-se definir a operação de incremento como uma *seção crítica* do código, ou seja, um trecho que manipula um recurso compartilhado.

Figura 1 – Execuções com condição de corrida

Execução A

| T | Thread 1 | Thread 2 |
|---|---|---|
| 1 | carrega 0 de *x* | ... |
| 2 | soma *1* e *0* | carrega 0 de *x* |
| 3 | escreve 1 em x | soma *1* e *0* |
| 4 | ... | escreve 1 em x |

Execução B

| T | Thread 1 | Thread 2 |
|---|---|---|
| 1 | ... | carrega 0 de *x* |
| 2 | ... | soma *1* e *0* |
| 3 | ... | escreve 1 em x |
| 4 | carrega 1 de *x* | ... |
| 5 | soma *1* e *1* | ... |
| 6 | escreve 2 em x | ... |

Fonte: Elaborado pelo autor (2023).

O mecanismo de travas, também conhecido como *locks* ou *mutexes*, permite garantir que uma seção crítica seja executada em exclusão mútua, por meio das operações de *aquisição* da trava antes de iniciar a seção e da *liberação* da trava depois do término da seção.

O Código 3.3 considera o uso de um *mutex mtx*, compartilhado entre todas as *threads* e inicialmente liberado. Para que uma *thread* possa realizar o incremento é necessário que ela adquira a trava com a chamada de *mtx.lock()*. A aquisição da trava é sincronizada, sendo garantido que uma *trava* não será segurada por mais de uma *thread* ao mesmo tempo. Se uma *thread* tentar adquirir a *trava* enquanto outra *thread* a estiver segurando, a primeira será bloqueada. A *thread* bloqueada ficará em espera até que a trava seja liberada com a chamada de *mtx.unlock()*. Quando a trava for liberada, alguma outra *thread* que esperava para entrar na seção crítica irá adquirir a trava e poderá realizar a operação de incremento.

Código 3.3 – Incrementar *x* com travas

```
1 int x = 0;
2 mutex mtx;
3 void incrementar(){
4     mtx.lock();
5     x += 1;
6     mtx.unlock();
7 }
```

Fonte: Elaborado pelo autor (2023).

### 3.1.4 Primitivas Atômicas

Uma outra maneira de sincronizar computações concorrentes é usando primitivas atômicas de *hardware*. Essas primitivas atômicas podem ser usadas para construir bloqueios ou serem usadas por si só para projetar algoritmos concorrentes. Existem operações atômicas para somar, incrementar, trocar valores, ou realizar outras diversas operações.

Uma primitiva típica para construir operações de sincronização é a *Atomic-Exchange*, ou *Atomic-Swap*, que troca atomicamente um valor carregado por outro na memória (HENNESSY, John L; PATTERSON, David A, 2006). Com essa primitiva é possível construir uma trava, cujo funcionamento foi apresentado anteriormente. Uma trava pode ser uma variável na memória, para qual o valor 0 indica “livre” e o valor 1 “adquirida”. Um processador tenta adquirir a trava fazendo uma troca de 1, valor já carregado, com o valor no endereço de memória correspondente à trava. O valor retornado da instrução de troca é 1 se algum outro processador já tiver reivindicado o acesso e 0 caso contrário. Nesse último caso, o valor também é alterado para 1, impedindo que qualquer troca concorrente também recupere um 0. Como as trocas são indivisíveis, é impossível para processadores diferentes que tentem definir a variável de sincronização “pensarem” que definiram simultaneamente a variável.

*Compare-And-Swap* (CAS) é uma primitiva atômica que permite que o processador atualize atomicamente o valor de um local de memória se e somente se o valor atual do local de memória corresponder a um valor específico. A instrução *CAS* leva um local de memória, um valor esperado e um novo valor. A instrução compara o valor do local de memória com o valor esperado e, se eles corresponderem, atualiza o local de memória com o novo valor. Se a comparação falhar (ou seja, o valor atual do local de memória não corresponder ao valor esperado), a instrução não modifica o local de memória (HERLIHY, M.; SHAVIT, 2012).

Uma alternativa a essas instruções apresentadas é ter um par de instruções em que a segunda instrução retorna um valor a partir do qual pode ser deduzido se o par foi executado como se as instruções fossem atômicas. *Load-Linked* e *Store-Conditional* são instruções usadas em sequência. Se o conteúdo do local de memória especificado pelo *Load-Linked* for alterado antes que o *Store-Conditional* para o mesmo endereço ocorra, o *Store-Conditional* falha (HENNESSY, John L; PATTERSON, David A, 2006).

Outra instrução atômica é a *Fetch-And-Increment*, também conhecida como *Fetch-And-Add*. Essa instrução atomicamente carrega e incrementa um valor da memória. Anteriormente foi visto que a operação de incremente sofria com condições de corrida, e foi feita a correção do Código 3.2 com a utilização de travas. Contudo, é possível fazer essa correção utilizando a operação *Fetch-And-Increment*.

### 3.1.5 *Deadlocks*

Com o mecanismo de travas é possível oferecer a sincronização necessária para sistemas concorrentes e paralelos, sendo comum o uso de diversas travas em um mesmo sistema. Se um programa usa mais de uma trava, e as travas podem ser adquiridas em ordens diferentes, as *threads* podem esperar indefinidamente a liberação de uma trava. Ou seja, o sistema pode sofrer um *deadlock*. A sequência de operações da tabela 1 exemplifica o evento: no tempo 2 a *thread* 1 espera pela liberação da *trava_b* enquanto mantém a *trava_a*. A *trava_b* é mantida pela *thread* 2, que por sua vez, espera pela liberação da *trava_a*.

Tabela 1 – Deadlock

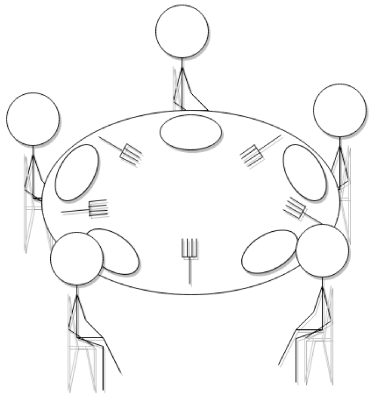
| T | Thread 1 | Thread 2 |
|---|---|---|
| 1 | trava_a.lock() | trava_b.lock() |
| 2 | trava_b.lock() | trava_a.lock() |

Fonte: Elaborado pelo autor (2023).

### 3.1.6 Inanição

Uma *thread* sofre com inanição quando continua em execução indefinidamente sem fazer qualquer progresso. O “Jantar dos Filósofos” (DIJKSTRA, 1965), um problema clássico de sincronização proposto por Dijkstra em 1965, exemplifica a necessidade de cuidado com o problema. No problema, cinco filósofos sentam em volta de uma mesa circular com um prato de comida cada. Entre cada par de pratos há um garfo. Quando um filósofo fica com fome, tenta pegar um garfo da sua esquerda e um da sua direita, um de cada vez. Se conseguir pegar os dois garfos, ele come, solta os garfos e volta a pensar. A Figura 2 mostra o cenário na mesa.

Figura 2 – Jantar dos Filósofos.



Fonte: Elaborado pelo autor (2023).

Para construir um programa eficiente que represente o cenário e não permita que os filósofos sofram de inanição é necessário tomar cuidados durante a aquisição

dos recursos compartilhados, os garfos. Por exemplo, se todos os filósofos primeiro pegarem o garfo à sua direita, nenhum deles pegará dois garfos. Uma solução pode envolver soltar o garfo já pego e esperar por um tempo aleatório até tentar novamente (TANENBAUM; BOS, 2014). Uma solução muito simples para o problema é a que explora a quebra de simetria na aquisição dos recursos compartilhados. Nesse método, alguns dos filósofos devem pegar o garfo à esquerda antes do garfo à direita, enquanto os outros pegam os garfos na ordem inversa. Dessa forma, dentre os filósofos que se sentam ao lado de um filósofo que pega os garfos na ordem inversa à sua, sempre haverá algum que conseguirá pegar os dois garfos.

## 3.2 OBJETOS CONCORRENTES

Uma estrutura de dados é um conjunto de algoritmos com o objetivo de organizar dados a fim de facilitar acessos e modificações (CORMEN *et al.*, 2009). Por exemplo, uma fila FIFO (do inglês: *First In, First Out*, “primeiro a entrar, primeiro a sair”) pode organizar os dados em um *buffer* circular e gerenciar acessos e modificações através das operações de inserção e remoção.

Em um modelo de computação concorrente com um conjunto de processos que compartilham memória, é necessário que o sistema ou a própria estrutura de dados gerencie acessos concorrentes. Uma maneira simples de fazer isso é utilizando travas. Uma fila FIFO pode ser implementada na linguagem C++, como no Código 3.4, baseado no código Java apresentado em (HERLIHY, M.; SHAVIT, 2012), com um *buffer* circular, índices para cabeça e cauda, e um *mutex*.

Código 3.4 – Fila com trava

```
1  template <typename T, int Capacity>
2  class FilaLock{
3   int head = 0, tail = 0;
4   T buf[Capacity];
5   std::mutex mtx;
6
7     T remover(){
8       mtx.lock();
9       if(tail == head){
10        mtx.unlock();
11        throw std::runtime_error("Fila Vazia");
12        }
13      T v = items[head % Capacity];
14      head++;
15      mtx.unlock();
16      return v;
17    }
18
```

```
19      void inserir(T valor){
20        mtx.lock();
21        if( ( tail - head ) == Capacity){
22          mtx.unlock();
23          throw std::runtime_error("Fila Cheia");
24          }
25        items[tail % Capacity] = valor;
26        tail++;
27        mtx.unlock();
28      }
29  };
```

Fonte: Elaborado pelo autor (2023).

Não é preciso empregar muito esforço para verificar que a implementação está correta porque as operações que acessam memória compartilhada são mutuamente exclusivas (HERLIHY, M.; SHAVIT, 2012). A trava garante que os métodos sejam chamados um por vez, oferecendo atomicidade aos métodos e permitindo especificá-los em pré e pós condições. Assim a propriedade FIFO da implementação fica evidente, já que os conjuntos de operações das seções críticas são sequenciadas, mesmo quando as chamadas dos métodos se sobrepõem no tempo.

É possível condicionar estruturas de dados a operações concorrentes de outras formas, não somente com travas. Uma Fila concorrente, na qual somente uma *thread* realiza operações de inserção e somente outra realiza remoções, pode ser facilmente construída 3.5. A implementação é a mesma de 3.4, mas sem travas.

Código 3.5 – Wait-free SPSC FIFO

```
1  template <typename T, int Capacity>
2  class FilaSPSCWaitFree{
3    int head = 0, tail = 0;
4    T buf[Capacity];
5  }
6  T remover(){
7    if(tail == head)
8        throw std::runtime_error("Fila Vazia");
9    T v = items[head % Capacity];
10   head++;
11   return v;
12 }
13 void inserir(T valor){
14   if( ( tail - head ) == Capacity)
15       throw std::runtime_error("Fila Cheia");
16   items[tail % Capacity] = valor;
17   tail++;
18 }
```

Fonte: Elaborado pelo autor (2023).

Como não há travas, uma inserção e uma remoção podem acontecer ao mesmo tempo. Uma *thread* pode modificar o índice para a cabeça enquanto a outra modifica o índice para a cauda. As operações não são sequenciadas, isso torna mais difícil verificar que a implementação está correta, já que não há uma ordenação óbvia para as operações e a propriedade FIFO não é tão evidente (HERLIHY, M.; SHAVIT, 2012).

Algoritmos e estruturas de dados que utilizam aquisição e liberação de regiões críticas como método de sincronização são chamados de bloqueantes. Enquanto a FilaLock (Código 3.4) é bloqueante, a FilaSPSCWaitFree (Código 3.5) é construída com uma condição de progresso não bloqueante. Para facilitar a compreensão do funcionamento dessas estruturas é possível permitir a especificação e avaliação das mesmas por meio de técnicas de verificação conhecidas do domínio sequencial. Uma forma de permitir isso é fazendo da estrutura linearizável (HERLIHY, M. P.; WING, 1990). É mais fácil raciocinar sobre as execuções de operações sobre objetos compartilhados quando eles atendem essa condição de corretude.

## 3.3 LINEARIZABILIDADE

Linearizabilidade é uma condição de corretude que pode ser informalmente definida como a garantia de que toda chamada de método deve aparentar surtir efeito instantaneamente em algum momento entre sua chamada e seu retorno. Ela garante que uma execução não termine de maneira inesperada, e facilita a compreensão do comportamento do sistema.

É possível descrever linearizabilidade em função de *históricos de computação*. Um sistema concorrente consiste de uma coleção de processos que se comunicam por meio de objetos concorrentes e executam conjunto de operações. Uma execução do sistema pode ser modelada por um histórico, que é um sequência finita de invocações de operações e eventos de resposta (WING; GONG, 1993). Escreveremos uma invocação como $\langle$*x op(args*) A*$\rangle$ onde *x* é o nome de um objeto, *op* é um nome de operação, *args** denota uma sequência de valores de argumento e *A* é o nome de um processo. A resposta a uma invocação é escrita como $\langle$*x term(res*) A*$\rangle$, onde *term* é uma condição de término e *res** é uma sequência de resultados. Usamos "*ok*" para término normal.

Considerando $<_H$ a ordem parcial do histórico *H*, *H* é linearizável se pode ser estendido à um histórico $H'$, adicionando zero ou mais eventos de resposta à invocação, tal qual:

- $H'$ é consistente com alguma execução sequencial S válida, isto é, os processos parecem intercalar somente em operações completas;
- $<_H \subseteq <_S$, ou seja, a aparente intercalação sequencial respeita a ordem de precedência em tempo real das operações.

Figura 3 – Históricos

| H1 | H2 |
|---|---|
| q inserir(x) A | q inserir(x) A |
| q ok() A | q ok() A |
| q inserir(y) B | q inserir(y) B |
| q ok() B | q remover() A |
| q remover() B | q ok() B |
| q ok(x) B | q ok(y) A |
| q remover() A | |
| q ok(y) A | |
| q inserir(z) A | |

Fonte: Elaborado pelo autor (2023).

O histórico H1, apresentado na Figura 3, é linearizável porque quando adicionado ⟨*x term(res*) A*⟩ é equivalente à um histórico sequencial. O histórico H2, também apresentado na Figura 3, não é linearizável porque a operação completa ( invocação e resposta) da inserção de *x* precedem a inserção de *y*, mas *y* é removido antes de *x*.

Objetos linearizáveis são aqueles para os quais os históricos de computação são linearizáveis de acordo com uma especificação sequencial (HERLIHY, M. P.; WING, 1990). Uma maneira comum de mostrar que uma implementação de objeto concorrente é linearizável é identificando para cada método um *ponto de linearização* (HERLIHY, M.; SHAVIT, 2012) no qual o método é efetivado. Para implementações baseadas em trava, o fim da seção crítica de cada método, o momento no qual uma trava é liberada, pode servir como seu ponto de linearização. Para implementações que não utilizam travas, o ponto de linearização é tipicamente um único passo onde os efeitos da chamada do método se tornam visíveis para outras chamadas de método.

Os pontos de linearização da FilaLock são facilmente identificados junto às seções críticas dos métodos de inserção e remoção. Já na FilaSPSCWaitFree, os pontos de linearização dependem da execução. Na remoção, o momento em que o índice para cabeça é atualizado é um ponto de linearização quando o método retorna um valor. Quando a fila está vazia, o ponto de linearização da remoção é o lançamento da exceção. O método de inserção tem pontos de linearização semelhantes.

## 3.4 CONDIÇÕES DE PROGRESSO

Para gerenciar o acesso a recursos compartilhados podem ser usados mecanismos de bloqueio tradicionais, como *mutexes* e semáforo, sendo possível identificar características quanto ao progresso da computação em algoritmos que usam esses mecanismos. Um algoritmo *deadlock-free* garante que alguma *thread* que tente adquirir a trava eventualmente consiga. *Starvation-free* é a condição de progresso que

indica que toda *thread* que tente adquirir uma trava, eventualmente conseguirá. A segunda, que garante o progresso de todas as *threads*, é uma condição mais forte que a primeira, que somente garante progresso de ao menos uma *thread*.

*Deadlock-freedom* e *starvation-freedom* normalmente são descritas em função dos mecanismos de travas e seções críticas e são adequadas para descrever algoritmos como os usados nas operações da implementação bloqueante descrita no Código 3.4. É possível que a preempção possa atrasar uma *thread* que mantém um *lock*, efetivamente bloqueando o progresso de outras *threads*. Para lidar com esses problemas, foram desenvolvidas condições de progresso não bloqueante (*non-blocking*) (HERLIHY, M.; SHAVIT, 2011) e que descrevem algoritmos do mesmo tipo que os apresentados na implementação *wait-free* descrita no Código 3.5.

### 3.4.1 *Non-blocking*

Condições de progresso não bloqueantes (HERLIHY, M.; SHAVIT, 2011) garantem que o atraso de qualquer *thread* não impeça outras *threads* de progredirem. Algoritmos não bloqueantes são *lock-free* se alguma *thread* que chama esse método eventualmente retorna, e são *wait-free* se todas as *threads* que chamam esse método eventualmente retornam.

As condições de progresso bloqueantes *deadlock-free* e *starvation-free* garantem o progresso somente se cada *thread* eventualmente sai da seção crítica. Enquanto isso, *wait-freedom* e *lock-freedom* são independentes do escalonador, o que garante o progresso do sistema desde que as *threads* sejam escalonadas, não importando como são escalonadas (HERLIHY, M.; SHAVIT, 2011). Algoritmos não bloqueantes podem ser desafiadores de projetar e implementar, pois exigem uma coordenação cuidadosa do acesso concorrente a recursos compartilhados.

#### 3.4.1.1 *Lock-freedom*

*Lock-freedom* (HERLIHY, M.; SHAVIT, 2012) é uma condição de progresso não bloqueante que garante que, durante a execução concorrente do algoritmo por múltiplas *threads*, pelo menos uma delas completará sua operação em um número finito de passos, mesmo que outras *threads* estejam atrasadas, paradas ou falhando. O progresso do sistema não depende do comportamento individual de nenhuma *thread*, e nenhuma *thread* pode impedir o sistema de avançar. Apesar de um algoritmo *lock-free* garantir o progresso do sistema como um todo, ele não previne inanição de uma ou mais *threads* (HERLIHY, M.; SHAVIT, 2011).

Algoritmos *lock-free* evitam completamente o uso de bloqueios e frequentemente empregam operações atômicas, como CAS (*Compare-And-Swap* ou LL/SC (*Load-Linked/Store-Conditional*), para realizar operações sobre um recurso compartilhado. Essa característica é o que normalmente garante resiliência às decisões de

escalonamento, uma vez que o algoritmo não requer a aquisição de travas que potencialmente colocam *threads* em espera.

#### 3.4.1.2 *Wait-freedom*

Enquanto *lock-freedom* garante que pelo menos uma *thread* fará progresso mesmo que outras estejam paradas ou atrasadas, uma implementação *wait-free* de um objeto concorrente é aquela que garante que qualquer *thread* possa concluir qualquer operação em um número finito de passo, independentemente das velocidades de execução nos outros processos. *Wait-freedom* ainda oferece tolerância a falhas. Já que nenhum processo pode ser impedido de concluir uma operação, falhas de parada não detectadas ou variações nas velocidades de algum processo não podem impedir o progresso de nenhum outro processo (HERLIHY, M., 1991).

Além de primitivas atômicas, estruturas de dados *wait-free* são projetadas com o mecanismos de *helping*, nos quais *threads* evitam o próprio atraso em decorrência da lentidão de outra *thread* mais lentas completando a tarefa da *thread* mais lenta em seu lugar (HERLIHY, M.; SHAVIT, 2011). Embora *wait-freedom* seja uma condição de progresso mais forte do que *lock-freedom*, pode ser difícil projetar uma estrutura de dados que apresente bom desempenho e também tenha essa propriedade. Por isso, muitas vezes opta-se por uma condição de progresso mais fraca.

# 4 IMPLEMENTAÇÕES

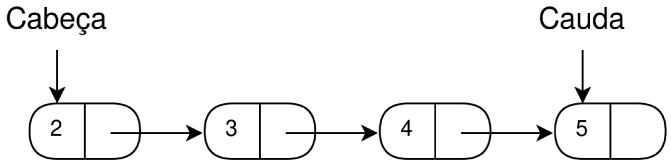
Algoritmos não bloqueantes são capazes de oferecer benefícios significativos para sistemas paralelos. Esses algoritmos podem usar primitivas atômicas de baixo nível como CAS (*compare-and-swap*) e LL/SC (*load-linked/store-conditional*). Com um projeto cuidadoso e evitando o uso de travas é possível construir sistemas que escalam em ambientes altamente paralelos e são resilientes às decisões de escalonamento (HARRIS, 2001). Este capítulo apresenta algumas implementações de estruturas de dados não bloqueantes.

Uma implementação de fila não bloqueante foi apresentada no Código 3.5. Estruturas de dados *wait-free* e *lock-free* não são limitadas a estruturas simples ou com restrição de um consumidor e um produtor. No entanto, o projeto de estruturas mais complexas e que atendam condições de corretude satisfatórias, como linearizabilidade, pode ser muito grande.

## 4.1 *MSQUEUE*

Proposta por Maged Michael e Michael Scott (MICHAEL; SCOTT, 1996), é uma implementação clássica de fila *lock-free* linearizável. A estrutura, denominada *MSQueue*, é baseada em uma lista ligada. Inserções e remoções são realizadas utilizando a operação CAS, de forma que os acessos à cabeça da estrutura não interferem com os acessos à cauda, oferecendo paralelismo entre as diferentes operações. A estrutura da fila pode ser vista na Figura 4.

Figura 4 – MSQueue.

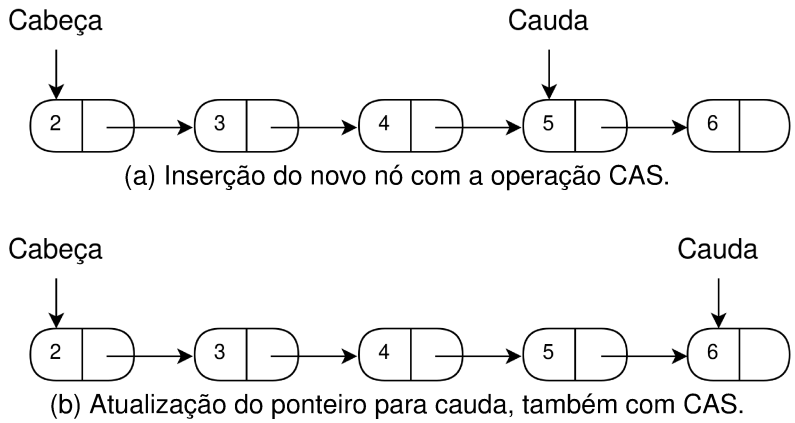



Fonte: Elaborado pelo autor (2023).

Para inserir na fila, cria-se um novo nó e obtém-se o ponteiro para a cauda da fila e o ponteiro para o elemento seguinte à cauda. Uma operação CAS tenta substituir o ponteiro para o elemento seguinte da cauda pelo ponteiro do novo elemento, na condição de que aquele deve ser nulo. Se o CAS falha, é indicativo de que uma outra *thread* já realizou uma inserção, então o ponteiro da cauda precisa ser avançado. A operação é realizada em um laço de repetição até que o CAS seja bem sucedido, e então outra operação CAS é utilizada para redefinir o ponteiro para a cauda. A Figura 5 ilustra a efetivação da operação em dois passos.

O algoritmo de inserção permite que o ponteiro para a cauda não aponte devidamente para o último elemento da lista, já que substituição do ponteiro seguinte ao

Figura 5 – *MSQueue*: Inserção.




Fonte: Elaborado pelo autor (2023).

último elemento e a substituição do ponteiro para a cauda acontecem em momentos diferentes. Todavia, um elemento sempre será inserido ao fim da fila, dado que a operação CAS somente será bem sucedida se o ponteiro para próximo do elemento em teste for nulo, e esse só será nulo se o elemento em teste for o último da fila. Quando detectado que o elemento em teste não é o último, tenta-se avançar o ponteiro para a cauda, corrigindo a referência.

A remoção também acontece em repetição. Obtém-se o primeiro elemento da fila e a referência para a cabeça, e tenta-se atualizar a referência com a operação CAS, na condição de que esta não foi alterada desde a obtenção do primeiro elemento da fila. Se detectada uma remoção em andamento, ou seja, a operação CAS falhou, tenta-se avançar a cabeça da fila. Em uma repetição do laço, não é garantido que a mesma *thread* que atualizou o ponteiro para a cabeça consiga efetivar a operação, entretanto, isso indica um avanço geral na computação, já que essa *thread* falhou porque outra realizou operação. Esse tipo de fluxo de execução é comum no *design* de estruturas de dados *lock-free*.

## 4.2 SCQ E LSCQ

O trabalho (NIKOLAEV, 2019) apresenta filas *lock-free* de alto desempenho. Nikolaev propõe que a operação atômica FAA (*Fetch-And-Add*) é mais adequada para construção de filas circulares do que a operação CAS, demonstrando experimentalmente que o incremento atômico escala melhor. O trabalho então propõe algoritmos de inserção e remoção de filas que faz o uso de ambas operações.

O trabalho toma como base um algoritmo de fila circular que utiliza a operação FAA sobre um vetor para garantir a linearizabilidade durante a chamada concorrente das operações de inserção e remoção por diferentes *threads*. O algoritmo inicial (Figura

6) é simples, sujeito a *livelocks* e assume um vetor de tamanho infinito. Esse algoritmo também exige reservar dois valores para controle (na Figura 6 são representados por $\perp$ e $\top$), que indicarão quando uma posição está livre ou está inválida para inserção. Nikolaev constrói uma fila circular *lock-free* real, denominada SCQ (*Scalable Circular Queue)*, corrigindo a condição de *livelock* e adequando os algoritmos de inserção e remoção para um vetor de tamanho finito.

Figura 6 – Fila *lock-free* com *buffer* de tamanho infinito.

```
   // data:  an array of pointers
   // aq is initialized empty
   // fq is initialized full
 1 bool enqueue_ptr(void * ptr)
 2     int index = fq.dequeue();
 3     if ( index = ∅ ) return False;        // Full
 4     data[index] = ptr;
 5     aq.enqueue(index);
 6     return True;                          // Success

 7 void * dequeue_ptr()
 8     int index = aq.dequeue();
 9     if ( index = ∅ ) return nullptr;      // Empty
10     ptr = data[index];
11     fq.enqueue(index);
12     return ptr;                           // Success
```

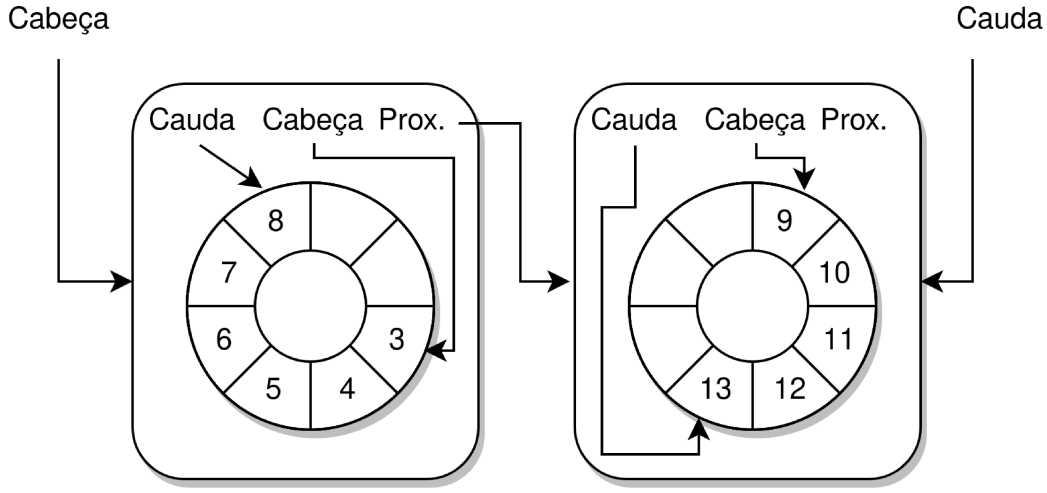
Fonte: NIKOLAEV, Ruslan (2019).

O trabalho ainda apresenta um fila *lock-free* de tamanho variável, denominada LSCQ, que, de forma simplificada, é uma fila *lock-free* de filas SCQ. A Figura 7 representa graficamente a estrutura. Uma inserção na LSCQ provoca a inserção na SCQ apontada pela cauda da LSCQ, enquanto uma remoção provoca a remoção de um elemento da SCQ apontada pela cabeça da LSCQ. Quando uma remoção provoca o esvaziar de uma SCQ, essa SCQ é liberada da memória. Quando uma inserção verificar a falta de espaço na SCQ apontada pela cauda, uma nova SCQ é criada e apontada como a cabeça.

## 4.3 LISTA *LOCK-FREE* DE HARRIS

Uma lista ligada não bloqueante linearizável foi proposta por Harris (HARRIS, 2001). No seu trabalho, Harris apresenta algoritmos para um caso específico de lista: uma lista ordenada em ordem crescente com base nas chaves. Listas ligadas são umas das mais básicas estruturas de dados usadas no projeto de sistemas. Os algoritmos dessa implementação são capazes de exemplificar a diferença de complexidade entre projetos bloqueantes e projetos *lock-free*.

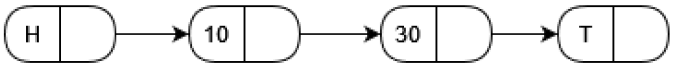
Figura 7 – LSCQ.




Fonte: Elaborado pelo autor (2023).

Essa estrutura de dados é composta por nós com valores da lista e dois nós sentinelas, uma para cabeça (*H*) da lista e outro para a cauda (*T*). Os nós possuem os campos *key* e *next* que armazenam o valor e o nó seguinte, respectivamente. É guardada à referência para cabeça, e para encontrar os elementos da lista é necessário acessar sucessivamente o campo *next* até encontrar um valor desejado. A Figura 8 mostra uma representação gráfica da lista.

Figura 8 – Lista Ligada.

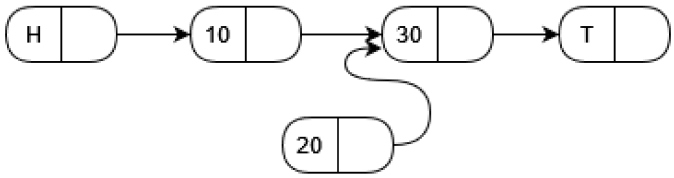



Fonte: Elaborado pelo autor (2023).

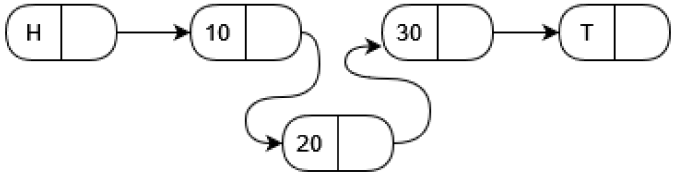
Para realizar uma operação de inserção é criado um nó com *key* igual ao valor a inserir e *next* igual ao nó seguinte. Apesar de que o novo nó contém uma referência para o nó seguinte, ele ainda não seria visível durante a travessia da lista e, portanto, a operação ainda não surtiu efeito, como mostra a Figura 9 (a). O novo nó só será visível após a realização de uma operação para trocar atomicamente o campo *next* do nó anterior pelo novo nó. Além disso, essa operação só se concretizará se o campo *next* do nó anterior ainda conter a referência ao nó seguinte.

A técnica chave para a garantir a corretude dessa estrutura *lock-free* surge no projeto do algoritmo de remoção. Trocar atomicamente o campo *next* do nó anterior para o nó seguinte ao nó deletado não poderia por si só garantir concordância dessa implementação com a especificação sequencial de uma lista-ligada, uma vez que uma inserção concorrente pode ser perdida, como mostra a Figura 10, que mostra a operação de remoção do elemento 10 concorrendo com a inserção do elemento 20, resultando na perda da inserção.

Figura 9 – Harris *Lock-Free List*: Inserção de 20.




(a) Criação do novo nó.

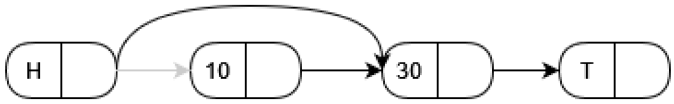



(b) Efetivação da operação com CAS.

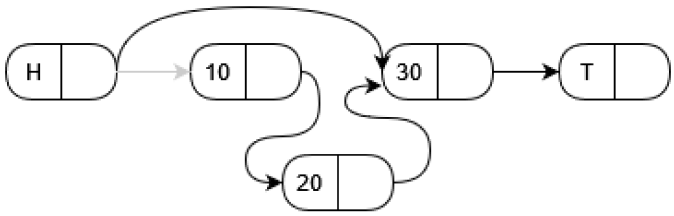
Fonte: Elaborado pelo autor (2023).

A inserção de 20 inicia a travessia e encontra que a posição de inserção deve ser entre o nó 10 e 30. Cria-se um nó com o valor 20 apontando para 30, e obtém-se a referência de 10, do qual será modificada a referência para o nó seguinte. Enquanto isso, a remoção de 10 ocorre, e o ponteiro da cabeça é alterado para apontar para 30. A execução da inserção de 20, então realiza a troca da referência do nó seguinte à 10, o que causa a situação da Figura 10.

Figura 10 – Harris *Lock-Free List*: Remoção incorreta.




(a) CAS do campo *next* do nó anterior para o nó posterior.




(b) Inserção concorrente que havia guardado a referência ao nó removido.

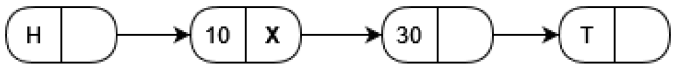
Fonte: Elaborado pelo autor (2023).

Com uma única operação CAS não seria possível detectar alterações entre os nós 10 e 30 ao mesmo tempo que é detectado se o nó 10 ainda é o nó seguinte à cabeça da lista, como apresentado na Figura 10. Então, ao invés de utilizar uma única operação de CAS, as remoções são realizadas em duas etapas, como na Figura 11. A primeira etapa é a *remoção lógica* do nó. Um nó é logicamente removido por meio

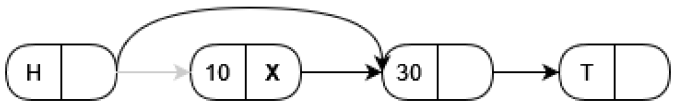
de uma marcação no seu campo *next*, feita com um CAS, que indicará para outras operações que esse nó deve ser desconsiderado. A segunda etapa é a *remoção física* do nó, efetivamente desvinculando ele da estrutura usando outra operação CAS.

Na primeira etapa de remoção, a marcação é feita no ponteiro para o próximo elemento durante a chamada da função de remoção pelo usuário. A segunda etapa só é feita na execução de buscas, inserções e novas remoções, que removem fisicamente nós marcados encontrados durante a travessia. A marcação garante que a tentativa de trocar o elemento seguinte a um que será removido fisicamente sempre falhe, já que essa troca é feita com uma operação CAS sobre o ponteiro para o próximo elemento. A situação da Figura 10 não ocorrerá mais, porque, por meio da marcação, uma inserção pode detectar um nó que está prestes a ser removido e removê-lo antes de realizar a inserção.

Figura 11 – Harris *Lock-Free List*: Remoção do valor 10.




(a) Remoção Lógica.




(b) Remoção Física.

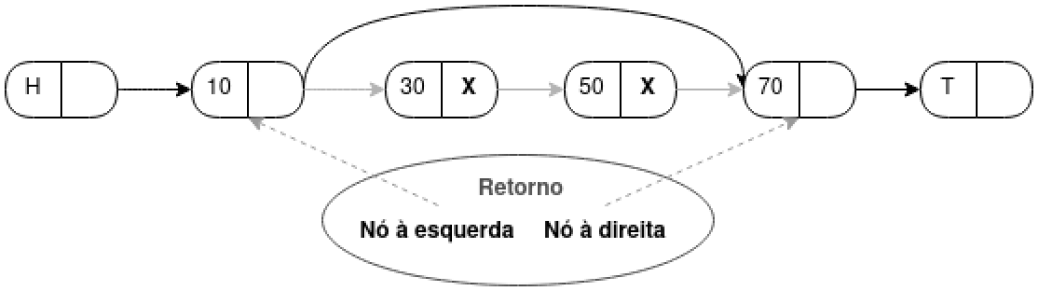
Fonte: Elaborado pelo autor (2023).

Toda travessia é realizada por uma função auxiliar de busca. Dado uma chave de busca, essa função deve retornar dois nós adjacentes e não marcados. O nó “à esquerda” deve ter valor menor que a chave de busca e o nó “à direita” deve ter valor maior ou igual ao da chave de busca. O algoritmo armazena um nó temporário não marcado com valor menor que o valor da chave de busca e continua a travessia até encontrar um nó não marcado seguinte. Se o novo nó não marcado encontrado tiver valor menor que o valor da chave de busca, esse será o novo nó temporário, e uma nova busca por um nó não marcado seguinte será realizada. Esses nós encontrados podem não ser adjacentes, sendo possível existir um ou mais nós marcados entre eles, e nesse caso os nós intermediários são removidos antes do retorno, como mostra a Figura 12.

## 4.4 *HASH TABLE* E LISTA *LOCK-FREE* DE MICHAEL

Uma *hash table lock-free* linearizável foi proposta por Maged M. Michael (MICHAEL, 2002a). Uma *hash table*, ou tabela de dispersão, é uma estrutura de dados

Figura 12 – Harris *Lock-Free List*: Resultado de busca por 40.

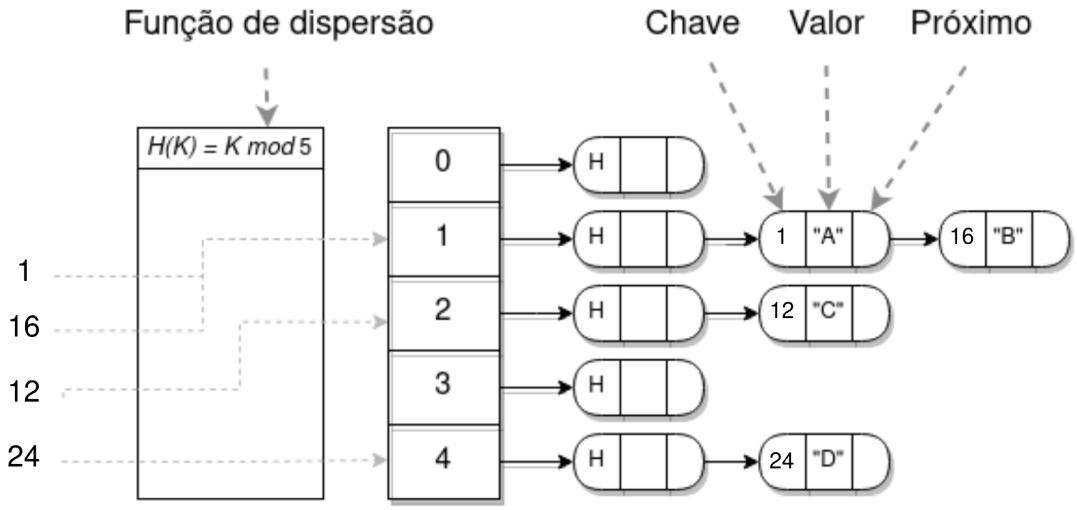



Fonte: Elaborado pelo autor (2023).

que armazena um conjunto de pares de *chave-valor*, fornecendo métodos de acesso como busca, inserção e remoção, e coordena o local de armazenamento por meio de uma função de dispersão. Para acessar a estrutura é necessário passar uma chave para o método de acesso. A estrutura deve ser implementada de forma que a função de dispersão permita encontrar o local em que o valor associado à chave fornecida deve ser encontrado.

Uma função de dispersão pode fornecer o mesmo resultado para duas chaves distintas. Um método de resolução de colisões é encadear pares para os quais a chave resulta em um mesmo *hash* em uma lista ligada, chamada de *bucket*. A Figura 13 é uma representação gráfica da proposta de Michael, que resolve as colisões dessa maneira.

Figura 13 – Michael *Lock-Free Hash Table*: Exemplo de tabela de dispersão com 5 *buckets* e função de mapeamento *h(k) = k mod 5*.




Fonte: Elaborado pelo autor (2023).

Essa implementação propõe um vetor de *buckets* de tamanho fixo, com cada posição do vetor apontando para uma lista ligada *lock-free*. Para suportar a associação de chaves e valores, os nós das listas ligadas possuem um campo que armazena um

valor. A implementação de lista ligada é semelhante à proposta de Harris (HARRIS, 2001), e a propriedade *lock-freedom* dessa *hash-table* surge do uso das listas *lock-free* como *buckets*. O principal diferencial da proposta de Michael é a compatibilidade com mecanismos de gerenciamento de memória *lock-free*, que permitem liberar a memória de nós já removidos da estrutura de dados.

As operações de inserção e remoção podem ser descritas da mesma maneira que a proposta de Harris, e graficamente representadas como nas Figuras 9 e 11. A remoção usa o recurso de marcação para remoção em duas etapas, lógica e física, e a inserção usa a operação CAS para mover o ponteiro de próximo do nó anterior para um novo. Da mesma maneira que na lista de Harris, a remoção física e a busca dos nós adjacentes para inserção se dão em uma função auxiliar de busca.

Diferentemente da função de busca de Harris, as buscas feitas pela lista de Michael fazem uso de variáveis da instância ao invés de retornar os nós adjacentes. Uma busca irá realizar a travessia da lista até que encontre um chave maior ou igual a chave de busca, ou até que chegue ao fim sem encontrar uma chave que satisfaça a condição. Se nós adjacentes não marcados que satisfazem as condição de que o nó “à direita” deve ter a chave maior ou igual a chave de busca, eles são gravados nas variáveis do objeto.

Uma operação de inserção na lista recorre à uma busca, que grava os nós adjacentes em variáveis da instância, e por meio de uma operação CAS insere o novo nó entre. Uma falha no CAS indica que um ou mais dentre três eventos ocorreram desde a gravação dos nós adjacentes pela busca. O nó “à esquerda” foi marcado, o nó “à direita” foi removido, ou um novo nó já foi inserido entre esses nós adjacentes. Nesses casos, haverá uma nova busca por nós adjacentes para inserção.

As operações de inserção, remoção e busca na *hash-table* de Michael, em resumo, recebem uma chave (e um valor, no caso de uma inserção), obtém o índice do *bucket* por meio da aplicação da função de dispersão sobre a chave de entrada, e delega à operação à lista ligada obtida. A garantia de *lock-freedom* das operações se baseia na propriedade *lock-free* das listas ligadas.

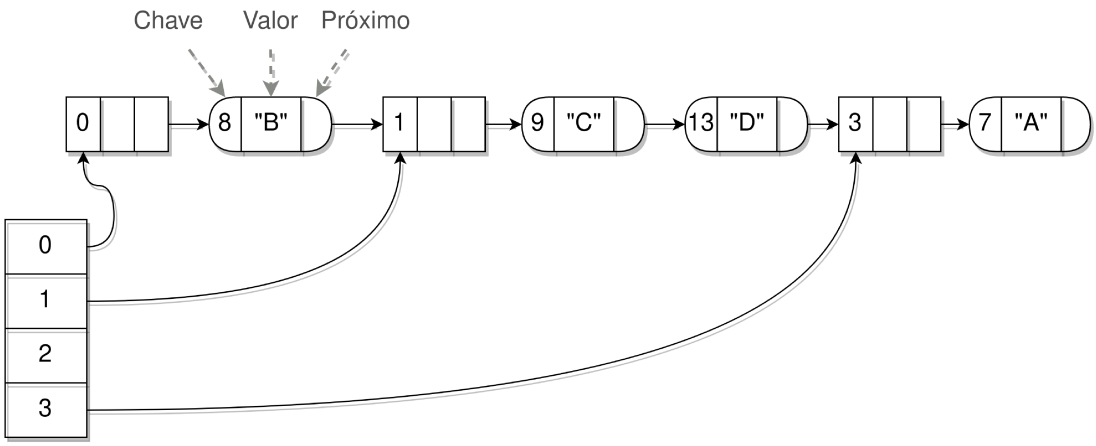
## 4.5 *HASH TABLE* EXTENSÍVEL *LOCK-FREE* DE SHALEV E SHAVIT

Shalev e Shavit (SHALEV; SHAVIT, 2006) apresentaram uma *hash-table lock-free* que possibilita o redimensionamento gradual da estrutura. O algoritmo se baseia em manter todos os pares de chave-valor em uma única lista ligada *lock-free* e considerar que porções dessa lista, ou sub-listas, são os *buckets* da *hash table*.

Essa *hash-table* pode ser vista como duas subestruturas interconectadas. Uma lista ligada de chaves e valores, e uma vetor expansível de ponteiros para nós da lista. As entradas do vetor são os *buckets lógicos*. Qualquer par pode ser encontrado fazendo a travessia do início da lista, enquanto os *buckets* oferecem atalhos para

minimizar o custo de acesso. O vetor tem tamanho 2 e dobra seu tamanho quando o número de elementos na lista excede $size * L$, com *size* sendo o tamanho atual do vetor e *L* um parâmetro configurável da estrutura chamado *Load Factor*. Uma representação gráfica da estrutura pode ser vista na Figura 14.

Figura 14 – *Split-ordered Hash Table*




Fonte: Elaborado pelo autor (2023).

Um *bucket* inicia com um nó sentinela. O *bucket* de índice 0 é inicializado durante construção da *hash table*, com a criação do nó sentinela como a cabeça e único elemento da lista ligada. Os demais são inicializados no primeiro acesso, e nesse momento é criado o nó sentinela que indica o início desse novo *bucket*. Para criação de um novo *bucket*, um *bucket* já existente é "dividido" quando um novo nó sentinela é introduzido entre seus elementos.

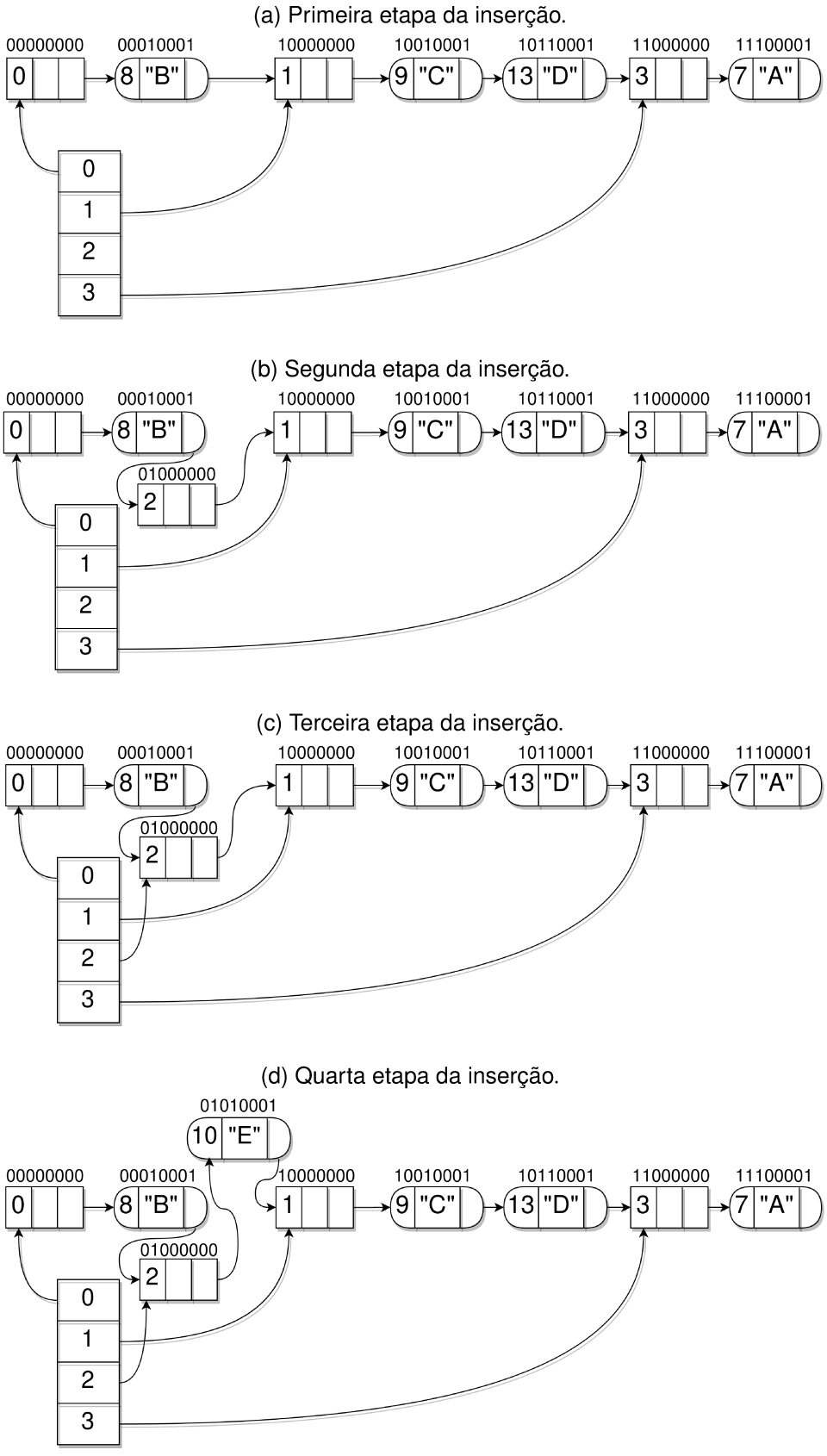
Deve ser possível reproduzir essa divisão recursivamente, já que um *bucket* pode ser dividido repetidamente. O método usado para garantir essa propriedade à lista é denominado *recursive split-ordering*, que é uma ordenação aplicada as chaves que mantém os elementos em um determinado *bucket* adjacentes ao longo dos processos de divisão de *buckets*.

A ordenação é garantida invertendo a representação binária dos valores de *hash* das chaves. Inicialmente o tamanho do vetor de *buckets* é $2^i$, e o *bucket b* para uma chave *k* é $k \mod 2^i$. Quando o vetor passa a ter tamanho $2^{i+1}$, depois de um redimensionamento, os elementos serão divididos em dois grupos. Alguns elementos permanecerão no *bucket b*, enquanto outros migrarão para o *bucket k* $\mod 2^{i+1} = b + 2^i$. Com isso é possível saber onde colocar o novo nó sentinela.

Os elementos desses dois grupos podem ser diferenciados pelo *bit i* menos significativo. Os que elementos no primeiro grupo tem esse *bit* em 0, os que passarão a ser do segundo grupos terão esse *bit* em 1. Por exemplo, os elementos com chave $9(1001_b)$ e $13(1101_b)$ são do mesmo *bucket* quando o vetor tem tamanho $2^2$ porque possuem os mesmos 2 *bits* menos significativos. Esses mesmos elementos passarão à

ser de *buckets* diferentes quando o vetor for expandido para acomodar $2^3$ posições, já que o terceiro *bit* menos significativo é diferentes nessas duas chaves. Esse processo induz a ordenação em divisão recursiva. Um exemplo de inserção pode ser conferido na Figura 15.

Figura 15 – *Split-ordered Hash Table*: Inserção da chave 10 na tabela.




Fonte: Elaborado pelo autor (2023).

A Figura 15 ilustra a inserção de um novo valor. A representação binária inversa,

junto com a marcação nos nós de dados, aparece acima de cada nó. Na primeira etapa (Figura 15 (a)) os *buckets* 0, 1 e 3 já estão inicializados. Na segunda etapa (Figura 15 (a)) um item com valor de *hash* 10 é inserido, fazendo com que um novo sentinela seja inserido. Na terceira etapa (Figura 15 (a)) o *bucket* 2 é atribuído ao novo sentinela. Por fim, na Etapa quarta e última etapa (Figura 15 (a)) a chave 10 é inserida no *bucket* 2.

A proposta inicialmente apresenta a estrutura considerando somente um vetor de *buckets* de tamanho fixo, utilizando uma variável de tamanho que limita o uso do vetor e que tem valor dobrado à cada expansão necessária. Posteriormente os autores propõem que essa subestrutura contenha dois níveis. O primeiro nível deve ser composto por um vetor denominado *tabela de segmentos* e o segundo nível deve ser composto por vetores denominados *segmentos*. Uma posição na tabela de segmentos aponta para um segmento, e uma posição no segmento aponta para o nó sentinela de um *bucket*.

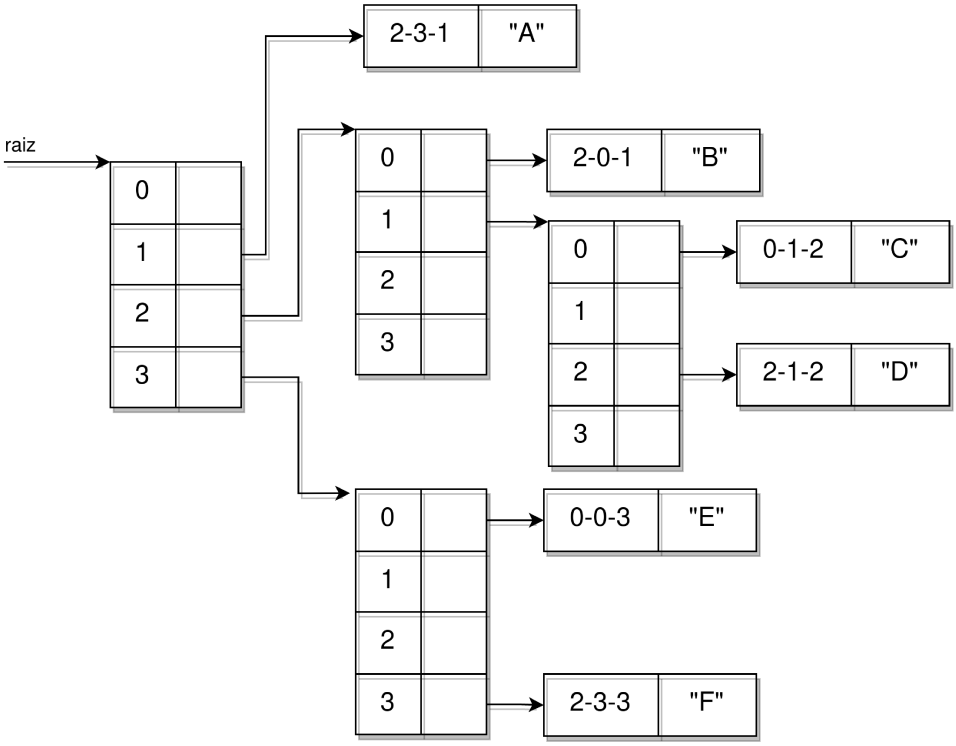
Os autores também ressaltam que essa implementação é modular no sentido de que é possível utilizar qualquer implementação de lista ligada para armazenar os pares de chave-valor. Para garantir a propriedade *lock-free* da tabela é necessária que a lista ligada escolhida também seja *lock-free*.

## 4.6 *HASH TABLE* EXTENSÍVEL *WAIT-FREE* DE FELDMAN

Essa é uma implementação *wait-free* de uma *hash-table* com uma estrutura similar à uma árvore apresentada por Feldman, Laborde e Dechev (FELDMAN, S.; LABORDE; DECHEV, 2013), sendo cada posição da árvore um vetor de nós ou um nó de dados. Essa proposta usa um esquema de *hashing* expansível: o índice gerado pela função de dispersão é tratado como uma sequência de *bits* e é *re-dispersa* incrementalmente. Essa sequencia de bits é dividida de modo que cada sub-sequência represente uma posição em um vetor de nós na árvore. O esquema usado possibilita a expansão da estrutura de forma que seja possível acomodar tantos pares de chave-valor quanto o número de diferentes valores possíveis de serem representados com o número de *bits* da chave. A Figura 16 ilustra a estrutura da *hash-table*. A letra *A* é a marcação que indica que o campo aponta para um vetor de nós, e a letra *X* é a marcação que indica a necessidade de expansão.

O vetor de nós tem comprimento fixo determinado na construção da estrutura. Uma posição do vetor aponta para elementos da árvore que podem ser ou um outro vetor de nós ou um nó de dados. É possível diferenciar um ponteiro para um vetor ou para um dado por meio de uma marcação no próprio ponteiro que diferencia o tipo de elemento apontado. Um nó de dados contém um valor e o resultado da função de dispersão sobre a chave associada à esse valor. O nó de dados pode ser marcado, indicando que deve haver uma expansão nesse nó, e qualquer *thread* o substituirá por um vetor de nós.

Figura 16 – Feldman *Wait-Free Extensible Hash Table*: estrutura em forma de árvore com múltiplos níveis de vetores.

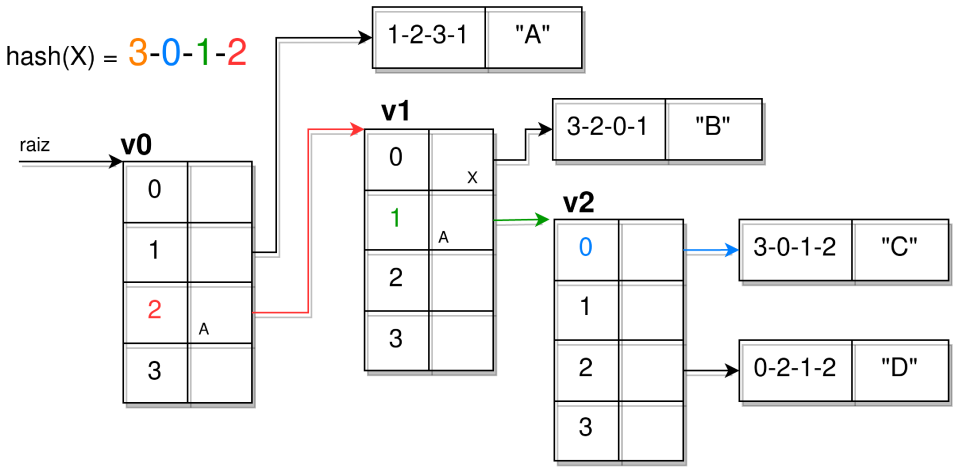



Fonte: Elaborado pelo autor (2023).

Para realizar uma busca, o resultado da função de dispersão de uma chave será tratada como uma sequência de *bits*. Com isso é possível extrair as subsequências que indicam as posições nos vetores que contém ponteiros para o próximo nível. A posição no vetor apontado pela cabeça é indicada pela *bits* menos significativos. As posições seguintes são obtidas sucessivamente pelas sequências menos significativas seguintes. A Figura 17 exemplifica a travessia até o nó correspondente à uma chave X, cuja *hash* é a um valor com subsequências de *bits* correspondentes à *3*, *0*, *1* e *2* na representação decimal. 2 será a posição no vetor apontado pela cabeça da árvore que contém o endereço para o próximo vetor *v1*. *1* é a posição no vetor *v1* que aponta para o vetor seguinte, *v2*. *0* é a posição que deve ser considerada em *v2*, que contém o nó de dados. Como o nó de dados foi encontrado em *v2*, o valor de *A* não é usado nessa busca.

Uma busca na *hash-table* é realizada como descrito no parágrafo anterior. Se um nó de dados é encontrado, como no exemplo, o valor armazenado é retornado, se não um valor nulo é retornado. O número total de vetores visitados é limitado pelo número de *bits* usado na representação binária da chave dividido pelo número de *bits* necessários para representar o comprimento de cada de cada vetor. Por exemplo, com chaves de 32 *bits* e vetores de comprimento igual à 64 a profundidade da estrutura é de no máximo 6 porque $\lceil 32/log_2 64 \rceil = 6$.

Figura 17 – Feldman *Wait-Free Extensible Hash Table*: travessia em busca do local para o par associado à chave que produz o *hash* 3-0-1-2




Fonte: Elaborado pelo autor (2023).

Uma inserção cria um novo nó com a chave e o valor de entrada e atravessa o mapa como em uma busca. Se for encontrado um nó de dados não marcado, o novo nó é inserido com uma operação CAS, e o nó que ocupava a posição é liberado. Se for encontrada uma posição vazia em um vetor de nós, o novo nó é somente inserido usando uma operação CAS. Uma falha nessa operação de CAS indica que uma outra *thread* pode ter marcado, estendido, ou realizado a inserção concorrentemente. Caso a posição tenha sido marcada será feita a extensão e continuado a travessia. Caso tenha ocorrido uma extensão, a travessia é continuada adentro do vetor. Caso uma inserção tenha ocorrido a inserção atual é descartada, liberando o novo nó da memória. Se for encontrado um ponteiro marcado para extensão, é realizado uma extensão que troca o ponteiro marcado para outro que aponte para um novo vetor de nós, que deve conter referência para o nó de dados antes apontado pelo ponteiro marcado.

A remoção também realiza a travessia como em uma busca. Se não for encontrado um nó de dados com a chave de entrada é retornado falso. Se for encontrado um nó marcado para expansão é realizada uma expansão e a busca pode continuar adentro do novo vetor. Se for encontrado o nó de dados não marcado e que contenha o valor associado à chave de entrada é realizada uma tentativa de substituição da referência ao nó por uma referência nula. Caso a operação CAS suceda, o nó removido é liberado da memória. Caso falhe, é possível que outra *thread* tenha marcado o nó para extensão, tenha realizado uma extensão, ou já tenha removido o nó. Se o nó foi marcado, é realizado uma extensão, e a operação de remoção pode continuar adentro o novo vetor de nós. Se outra *thread* realizou uma extensão, um novo vetor de nós tomou o lugar do nó, e a remoção pode continuar adentro desse vetor. Se a remoção já ocorreu, é retornado que a operação de remoção foi bem sucedida.

Os autores apontam que para garantir progresso *wait-free* das operações sobre a *hash-table* é necessário que as alocações de novos nós e as liberações de nós existentes sejam realizadas por meio de um mecanismo de liberação de memória também *wait-free*.

## 4.7 TRABALHOS DISCUTIDOS

Nesse capítulo foi discutido o funcionamento de algumas implementações de filas, listas e tabelas de dispersão não bloqueantes, demonstrando como é possível utilizar a operações atômicas para realizar a sincronização de recursos compartilhados e construir algoritmos concorrentes eficientes. Os trabalhos discutidos estão apresentados na Tabela 2, que mostra o autor, a estrutura de dados proposta, a condição de progresso e o ano em que a proposta foi publicada.

Tabela 2 – Trabalhos discutidos.

| **Autores** | **Estrutura de Dados** | **Progresso** | **Ano** |
|---|---|---|---|
| Michael, Maged M.; Michael, Scott L. | Fila | Lock-free | 1996 |
| Timothy L. Harris | Lista | Lock-free | 2001 |
| Michael, Maged M. | Lista | Lock-free | 2002 |
| Michael, Maged M. | Tabela de Dispersão | Lock-free | 2002 |
| Shalev, Ori; Shavit, Nir. | Tabela de Dispersão | Lock-free | 2006 |
| Feldman, Steven; Laborde, Pierre; Dechev, Damian. | Tabela de Dispersão | Wait-free | 2013 |
| Nikolaev, Ruslan; | Fila | Lock-free | 2019 |

Fonte: Elaborado pelo autor (2023).

Os trabalhos discutidos apresentam o potencial da sincronização não bloqueante e a complexidade do projeto desse tipo de algoritmo. A apresentação dos algoritmos também demonstrou técnicas recorrentes no projeto de estruturas de dados que atendem essa condição de progresso, como a marcação de ponteiros e os laços de repetição com a operação CAS atômica. Todos os algoritmos deste capítulo podem ser implementados em arquiteturas atuais, o que torna possível a substituição de estruturas de dados bloqueantes por versões não bloqueantes em *software* existente.

# 5 ANÁLISE DE DESEMPENHO DE ESTRUTURAS DE DADOS NÃO BLOQUEANTES

Esse capítulo apresenta uma comparação de desempenho realizada entre diferentes implementações de estruturas de dados utilizando os resultados obtidos de *microbenchmarks*. O capítulo é dividido em duas partes, uma dedicada à filas, e outra dedicada à tabelas de dispersão. Parte dos resultados discutidos foram apresentados em (LUIZ; MENDIZABAL, O., 2023).

As implementações comparadas foram desenvolvidas na linguagem C++ e obtidas de bibliotecas de programação disponíveis em repositórios de código público. Parte das implementações não bloqueantes testadas são baseadas em trabalhos apresentados na seção Implementações, e foram obtidas das bibliotecas de programação *Xenium*[1], *LibCDS*[2] e *WFC*[3], que fornecem coleções de estruturas de dados concorrentes e algoritmos de gerenciamento de memória. As estruturas de dados são parametrizáveis, sendo possível escolher dentre diferentes mecanismos de gerenciamento de memória que permitem usufruir plenamente das propriedades das estruturas de dados não bloqueantes.

As bibliotecas TBB[4] e *Boost*[5] foram usadas como fonte de implementações robustas de estruturas de dados. TBB (*Thread Building Blocks*) é uma biblioteca de programação mantida pela Intel usada para simplificação da paralelização de código e fornece estruturas de dados preparadas para acesso concorrente. A biblioteca Boost promete melhoria de produtividade proporcionando código revisado por especialistas, sendo fonte de diversas adições à biblioteca padrão do C++. Foram usadas as bibliotecas TBB e Boost nas versões *2021.5.0-7* e *1.74*, respectivamente.

Além dessas implementações, foram desenvolvidas estruturas de dados concorrentes baseadas nas implementações disponíveis na biblioteca padrão utilizando um único *mutex* que trava à cada operação. Essas implementações não tomam cuidados com desempenho, e foram usadas para possibilitar uma comparação com uma implementação que emprega a solução mais básica para sincronização de acessos à memória compartilhada.

Os testes foram realizados em um computador com quatro processadores Intel Xeon E5-4620, cada um com 2.2 GHz, 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*. O parâmetro `-mcx16` foi usado para garantir disponibilidade da instrução *CAS* às bibliotecas.

[1] `https://mpoeter.github.io/xenium/`
[2] `https://libcds.sourceforge.net/doc/cds-api/index.html`
[3] `https://github.com/Exerg/WaitFreeCollections`
[4] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html`
[5] `https://www.boost.org/`

O código dos programas de teste para avaliar o desempenho das filas e tabelas de dispersão está disponível no GitHub, que pode ser acessado pelo endereço `https://github.com/douglaspereira04/non-blocking-cpp`, estando disponível na *branch main*.

## 5.1 AVALIAÇÃO EXPERIMENTAL COM FILAS

Os experimentos com filas compararam cinco implementações. Uma implementação baseada na implementação *queue* da STL do C++, que trava a cada operação, representa uma implementação bloqueante básica, enquanto uma implementação da biblioteca *Boost* representa uma implementação bloqueante mais otimizada. Da biblioteca TBB, foi utilizada uma implementação de fila não-bloqueante. A documentação da TBB não oferece uma descrição dos algoritmos usados e também não indica se a implementação é *lock-free* ou *wait-free*. Uma implementação de fila *lock-free* baseada na proposta de Nikolaev (NIKOLAEV, 2019), a LSCQ, foi obtida da biblioteca Xenium para os testes. Foram usadas também duas implementações da proposta de Michael e Scott (MICHAEL; SCOTT, 1996), uma da biblioteca LibCDS e outra da Xenium.

### 5.1.1 Protótipo

Foi produzido um programa de testes para avaliar o desempenho e escalabilidade das diferentes implementações de filas. As comparações são feitas com base no tempo levado para realizar um número pré-definido de operações.

O programa é baseado em um modelo de testes do trabalho (LADAN-MOZES; SHAVIT, 2004). O programa de experimentação realiza $k$ operações sobre a fila. Cada uma das $n$ *threads* realiza $k/n$ iterações em um laço, que a cada iteração realiza uma operação sobre a fila e uma outra tarefa com custo computacional. A operação realizada é decidida a cada iteração, utilizando valores obtidos de um gerador de números pseudoaleatórios durante a execução, e segue uma distribuição uniforme, com 50% de chances de sortear uma inserção e outros 50% de sortear uma remoção.

A outra tarefa é a execução de um laço de repetição que simula uma trabalho local da *thread*. Ele serve para tornar o experimento mais realista, evitando longas execuções consecutivas de operações sobre a fila pela mesma *thread* (MICHAEL; SCOTT, 1996). O número de repetições no laço é sorteado durante a execução, sendo distribuído uniformemente entre 0 e 1000, conforme o trabalho (LADAN-MOZES; SHAVIT, 2004). Dentro do laço somente é realizado o incremento de um contador local.

O programa compara as estruturas considerando o número de iterações $k$ igual a $10^7$, e são realizadas 10 repetições do experimento para cada cenário de teste. O conjunto de cenários é a combinação dos seguintes parâmetros de teste:

- número de *threads* (1, 2, 4, 8, 16, 32);

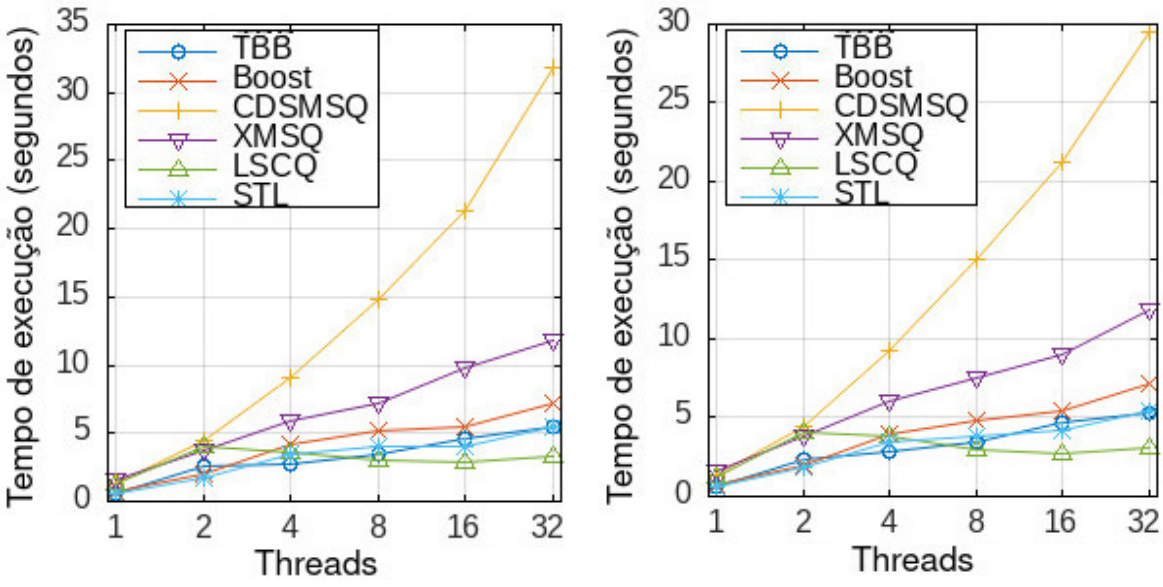
- pré-inserção ou não de $10^7$ elementos na fila;
- tamanho dos valores (4 bytes ou 4 kbytes).

Nos cenários com elementos pré inseridos, a estrutura sempre terá elementos a remover. Isso ajuda avaliar cenários nos quais dificilmente a fila estará vazia e evita que as remoções tomem um caminho rápido de execução, no qual, por exemplo, só é feita uma verificação da indisponibilidade de elementos na fila.

### 5.1.2 Resultados

As saídas textuais do programa de testes foram usadas para produzir gráficos que facilitam a comparação dos resultados das diferentes implementações. Os gráficos apresentam tempo de execução em segundos, no eixo *y*, e o número de *threads*, no eixo *x*. Foram produzidos quatro gráficos, correspondentes à combinação dos possíveis valores para pré-inserções e tamanhos dos valores armazenados. A Figura 18 apresenta os resultados de testes com valores de 4 bytes, enquanto a Figura 19 contém os resultados para valores de 4 kbytes.

Figura 18 – *Resultados dos testes com filas de valores de 4 bytes.*



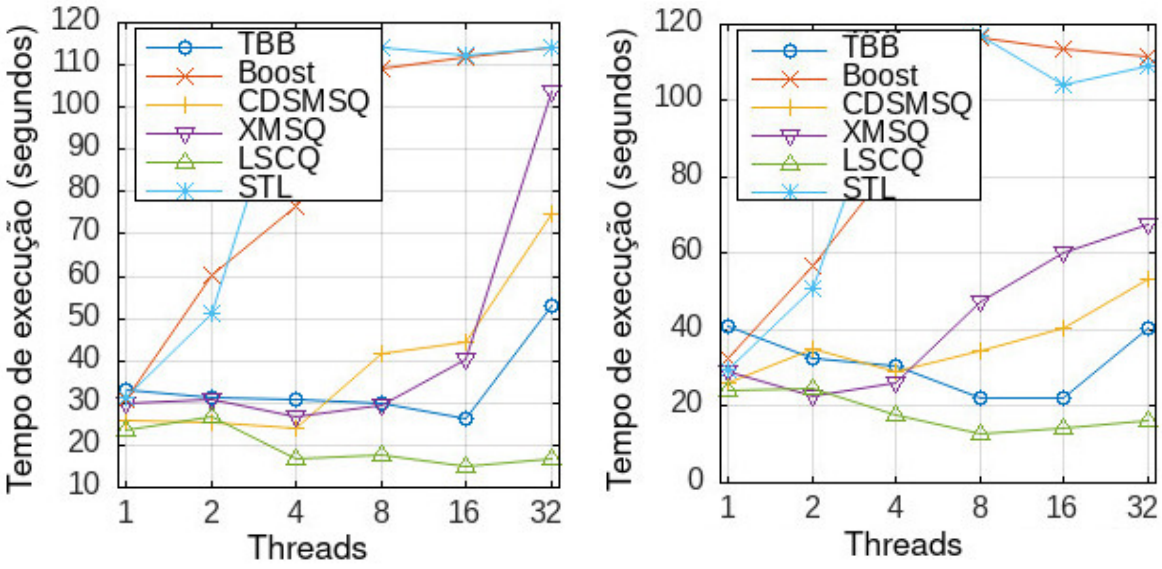

(a) Filas inicialmente vazias. (b) Filas com $10^7$ elementos pré inseridos.

Fonte: Elaborado pelo autor (2023).

#### 5.1.2.1 Valores pequenos

Nos resultados apresentados na Figura 18, com valores de 4 bytes, é possível observar que todas as implementações apresentaram seus melhores resultados com a execução em uma só *thread*. Nenhuma das implementações escalou bem, enquanto a existência de elementos previamente inseridos na fila não pareceu afetar signifi-

Figura 19 – *Resultados dos testes com filas de valores de 4 kbytes.*




(a) Filas inicialmente vazias. (b) Filas com $10^7$ elementos.

Fonte: Elaborado pelo autor (2023).

cativamente os resultados, sendo possível observar que o desempenho relativo das implementações é visualmente muito semelhante nos dois gráficos da figura.

As implementações *lock-free* baseadas na proposta de Michael e Scott (MICHAEL; SCOTT, 1996), a MSQueue, apresentaram as maiores quedas de desempenho. Com 1 *thread*, ambas as implementações apresentaram média dos tempos de execução em torno de 1,4 segundos, semelhante às demais implementações, entretanto, sofreram com um crescimento próximo de uma linear, e com 32 *threads* essas implementação das bibliotecas Xenium e LibCDS obtiveram médias próximas dos 11 e 30 segundos, respectivamente.

Diferentemente da MSQueue, a LSCQ tem seus algoritmos projetados com o uso de FAA, que escala melhor do que a instrução CAS (NIKOLAEV, 2019). Essa implementação *lock-free* apresentou os melhores resultados com 8, 16 e 32 *threads*, obtendo médias de menos de 3 segundos com 16 *threads*, enquanto todas as outras implementações apresentaram médias somente acima dos 4 segundos a partir desse nível de paralelização. Desconsiderando à execução não paralelizada dos experimentos, foi a única implementação que obteve melhorias de desempenho com o aumento do número de *threads*, com reduções das médias acima de 15%. Considerando essas características, essa implementação aparentou ser a mais resiliente ao aumento do número de *threads*.

Os resultados da implementação bloqueante, que trava um *mutex* a cada operação, são surpreendentemente bons, visto que são semelhantes aos da implementação não bloqueante da TBB e, em geral, melhores que os resultados da biblioteca Boost. Com 32 *threads*, os resultados da TBB e da STL são próximos de 5,4 segundos, com

diferença inferior a 0,1 segundo. Com 2, 4, 8 e 16 *threads*, a diferença pode representar até 28% a mais de tempo médio de execução. Dentre essas duas implementações é difícil definir qual é a melhor.

A implementação bloqueante da biblioteca Boost apresentou resultados medianos, e não obteve a melhor média dos tempos de execução para nenhum número de *threads*. Com 2 *threads* seus resultados são muito próximos aos da implementação baseada na STL, mas pareceu sofrer maior perda de desempenho com o aumento do número de *threads*, resultando em médias melhores somente que as das implementações baseadas na MSQueue.

#### 5.1.2.2 Valores grandes

Os experimentos com valores de 4 kbytes possibilitaram verificar o grande custo de gerenciar valores maiores. Nos testes base, com uma só *thread*, os resultados, que eram próximos de 1 segundo quando os valores eram pequenos, passaram ter médias de mais de 20 segundos.

Assim como nos experimentos com valores pequenos, a LSCQ ainda aparenta ser a mais adequada para o aumento do paralelismo da execução, porém, diferentemente dos cenários anteriores, o melhor resultado não é obtido com uma só *thread*. Com valores de 4 kbytes os melhores resultados são obtidos por essa implementação, que, em relação à execução sequencial, apresentou redução de até 38% da média do tempo de execução em experimentos com mais que 4 *threads*.

Ao contrário dos testes com valores de 4 bytes, a implementação da TBB apresentou melhorias no desempenho em relação ao teste base nos experimentos com valores de 4 kbytes. Com as filas inicialmente vazias, a melhoria foi de até 20%, enquanto com elementos previamente inseridos os ganhos foram de até 46%. O desempenho sofre uma grande queda com a execução em 32 *threads*, havendo uma queda de 60% do valor da média do tempo de execução em relação a execução sequencial no cenário com filas inicialmente vazias. No cenário com inserções prévias, a média dos tempos de execução da TBB com 32 *threads* se aproximou da média do cenário com 1 *thread*, como mostra a Figura 19(b).

As duas implementações bloqueantes testadas, STL e Boost, apresentaram os piores resultados. Com duas *threads*, a média do tempo de execução ultrapassa os 50 segundos para ambas as implementações, tempo ao menos 60% maior que teste sequencial e que os resultados das outras implemetações com esse mesmo número de *threads*. O desempenho permanece com o aumento do número de *threads*.

A implementação da Xenium baseada na MSQueue, a XMSQ nas figuras, apresentou melhorias de desempenho em relação à execução sequencial. Foram observadas melhorias de 10% nos cenários com filas vazias e de 20% nos cenários com inserções prévias, atingindo o melhor desempenho com 2 *threads* dentre todas as

implementações nesse ultimo caso. O desempenho sofre grande queda, e no cenário da Figura 19(a), a média dos tempos de execução aumenta bruscamente em mais de 200% com o uso de 32 *threads*, enquanto no cenário da Figura 19(b) o aumento é menor, sendo aproximadamente 130%.

## 5.2 AVALIAÇÃO EXPERIMENTAL COM TABELAS DE DISPERSÃO

Foram utilizadas implementação de tabelas de dispersão baseadas nos trabalhos de Feldman, Laborde e Dechev (FELDMAN, S.; LABORDE; DECHEV, 2013), de Michael (MICHAEL, 2002a), e de Shalev e Shavit (SHALEV; SHAVIT, 2006). A biblioteca *WFC* contém uma implementação baseada na proposta de Laborde, Feldman e Dechev (LABORDE; FELDMAN, S.; DECHEV, 2017). No conjunto de implementações comparadas foi incluída a implementação da biblioteca TBB, que é construída com uso de travas de granularidade fina, sendo usada como referência de uma implementação bloqueante robusta. Uma implementação bloqueante baseada no *unordered_map* da biblioteca padrão associada a um *mutex* que é bloqueado a cada operação foi usada como referência para uma implementação básica.

### 5.2.1 Protótipo

Para comparar as implementações de *hash tables* foi desenvolvido um programa para realização de um conjunto de operações sobre cada uma das estruturas na linguagem C++17. O programa inicializa a estrutura necessária para o uso de uma implementação em teste e registra o tempo de execução em um cenário de teste. Dentre os parâmetros variados nos diferentes cenários, estão:

- número de *threads* (1, 4, 8, 16, 32);
- número de chaves distintas ($2^{10}$ ou $2^{20}$);
- pré-inserção ou não de todas as chaves distintas;
- tamanho dos valores (4 bytes ou 4 kbytes).

A variação do número de *threads* permite avaliar a escalabilidade das implementações perante paralelização da carga de trabalho. A variação das poucas $2^{10}$ chaves distintas para a grande quantidade de $2^{20}$ chaves distintas permite observar o comportamento das implementações quando o cenário torna mais frequente operações em uma mesma área da estrutura, com poucas chaves distintas, e quando o cenário potencializa o crescimento da estrutura de dados, com muitas chaves distintas. A comparação de cenários sem e com pré-inserção de todas as chaves distintas permite avaliar o custo do crescimento da estrutura. E a variação de valores pequenos de 4 bytes para valores grandes de 4 kbytes permite observar o comportamento da estrutura em avaliação quando o tamanho dos valores armazenados cresce.

Os cenários também variam na distribuição dos tipos de operação. São três diferentes distribuições:

- Distribuição A, com 90% de buscas, 5% de inserções e 5% de remoções;
- Distribuição B, com 45% de buscas, 45% de inserções e 10% de remoções;
- Distribuição C, com 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.

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. Todos os cenários exigem 10.000.000 operações e foram realizadas 10 repetições de cada cenário.

As implementações baseadas na *hash table* de *Michael* (MICHAEL, 2002a) são configuradas com o número de *buckets* igual ao número de chaves distintas do cenário. Essas configurações assumem uma estimativa ótima para o número de chaves na estrutura de dados. Durante os comentários dos resultado a implementação da biblioteca Xenium baseada na proposta de Michael (MICHAEL, 2002a) será tratada pelo nome *XeniumMichael*, e a implementação da biblioteca LibCDS baseada também na proposta de Michael será chamada de *LibCDSMichael*.

As implementações baseadas nos trabalhos de Laborde, Feldman e Dechev (LABORDE; FELDMAN, S.; DECHEV, 2017; FELDMAN, S.; LABORDE; DECHEV, 2013) usam vetores de 16 ou 256 posições. O valor de dispersão das chaves tem 32 bits, então a profundidade máxima da estrutura usando vetores de 16 posições é de $\lceil 32/(log_2(16)) \rceil = 8$, e com vetores de $2^8$ posições é de $\lceil 32/(log_2(256)) \rceil = 4$. São usadas essas duas configurações diferentes para possibilitar uma observação do impacto da variação do tamanho dos vetores e número de níveis no desempenho. Na seção de resultados, a implementação baseada na proposta de Feldman configurada com vetores de $2^4$ posições será chamada de *Feldman4*, e a mesma implementação configurada com vetores de $2^8$ posições será chamada de *Feldman8*.

A implementação baseada na solução apresentada por Shalev e Shavit (SHALEV; SHAVIT, 2006) são configuradas com *Max Load Factor* $L = 1$. No trabalho que propõe essa solução é indicado que o impacto da mudança de *load factor* não é muito significativo, mas ainda é possível observar nos resultados de testes, que variam esse valor em 1, 4, 8 e 12, que execuções com *load factor* igual à 1 resultaram em um maior número de operações por milissegundo. O valor 1 é padrão na biblioteca usada para testar a solução *lock-free*, e também é o valor padrão na implementação de *hash table* da biblioteca padrão do C++. A implementação de lista *lock-free* usada para armazenar as chaves e valores é uma implementação da lista de Michael (MICHAEL, 2002a). Essa implementação será chamada de *SplitOrdered* na seção de resultados.

As implementações presentes nas bibliotecas LibCDS e Xenium foram configuradas para uso de *Hazard Pointers* (MICHAEL, 2002b) como mecanismo de ge-

renciamento de memória. Foram realizados testes preliminares indicando que esse mecanismo apresentava o melhor desempenho.

Um teste começa por inicializar a estrutura necessária para usar a implementação em experimentação. A biblioteca *LibCDS* impõe a necessidade de vincular as *threads* usadas à um gerenciador de *threads* próprio antes de instanciar e usar uma estrutura de dados da mesma biblioteca. Para as outras bibliotecas basta instanciar a estrutura de dados. Os testes consideram chaves inteiras de 0 até um valor igual ao número de chaves distintas. Quando o cenário exige a pré-inserção de todas as chaves distintas, o teste insere na *hash table* um valor associado à cada uma das chaves distintas. Depois disso o teste começa a contabilizar o tempo, e para cada *thread* é inicializado um gerador de números pseudo-aleatórios *Mersenne Twister*, com período igual à $2^{19937}$-1, usado para obter valores em uma distribuição uniforme com valores reais entre 0 e 1 e em uma distribuição uniforme com valores inteiros entre 0 e o número de chaves distintas. Os valores são usados para definir qual operação será feita e qual chave será usada na operação. Cada *thread* entrará em um laço de repetição e realizará uma porção de operações igual ao número de operações dividido pelo número de *threads*, ambos definidos pelo cenário de teste. A cada iteração é sorteada uma chave e qual tipo de operação será realizada, busca, inserção ou remoção. Quando todas as *threads* terminam suas operações o programa grava o tempo decorrido.

### 5.2.2 Resultados

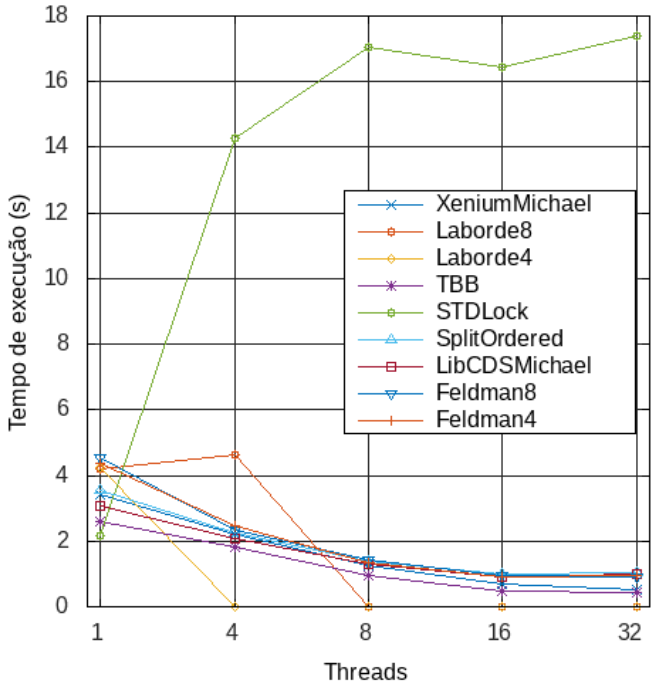
Nessa seção, serão analisados os desempenhos das diferentes implementações e o impacto da variação dos parâmetros dos cenários de teste. Essa seção foi dividida em subseções, cada uma dedicada a comentar os resultados obtidos com a variação de um dos parâmetros de teste. A partir das saídas textuais do programa de testes foram produzidos gráficos para facilitar a visualização dos resultados. Os gráficos apresentam tempo de execução em segundos, no eixo *y*, e o número de *threads*, no eixo *x*.

A implementação da biblioteca padrão que trava um *mutex* durante cada operação é apresentada na legenda dos gráficos como *STDLock*. Os resultados dos testes permitiram observar que o custo de travar o acesso à qualquer parte da estrutura à cada operação foi muito grande e não proporcionou ganhos de desempenho significativos com o aumento do número de *threads*. As outras implementações, em geral, apresentaram uma redução do tempo de execução com o aumento do número de *threads* e tempos de execução muito menores que os obtidos implementação *STDLock*.

A Figura 20 exemplifica o comportamento da implementação *STDLock*. Nesse cenário, com 1 *thread* o resultado da implementação *STDLock* é competitivo com o de outras implementações. O aumento do número de *threads* provoca o aumento da

diferença dos resultados dessa implementação em relação aos resultados das demais implementações. O mal desempenho da *STDLock* pode ser observado nos gráficos de todos os cenários, que podem ser conferidos no Apêndice C.

Figura 20 – *Resultados dos testes com hash tables inicialmente vazias, poucas chaves distintas, valores grandes e distribuição B.*




Fonte: Elaborado pelo autor (2023).

A implementação baseada na proposta de Laborde da biblioteca *WFC* apresentou erros durante as execuções. Por meio de uma ferramenta de depuração de código foi verificado a existência de erros durante as inserções resultantes da tentativa de modificação de atributos de um nó de dados que não está alocado na memória, o que sugere a existência de erros na implementação da biblioteca *WFC*. A Figura 20 mostra um cenário no qual só foi possível obter resultados em testes com 1 e 4 *threads*.

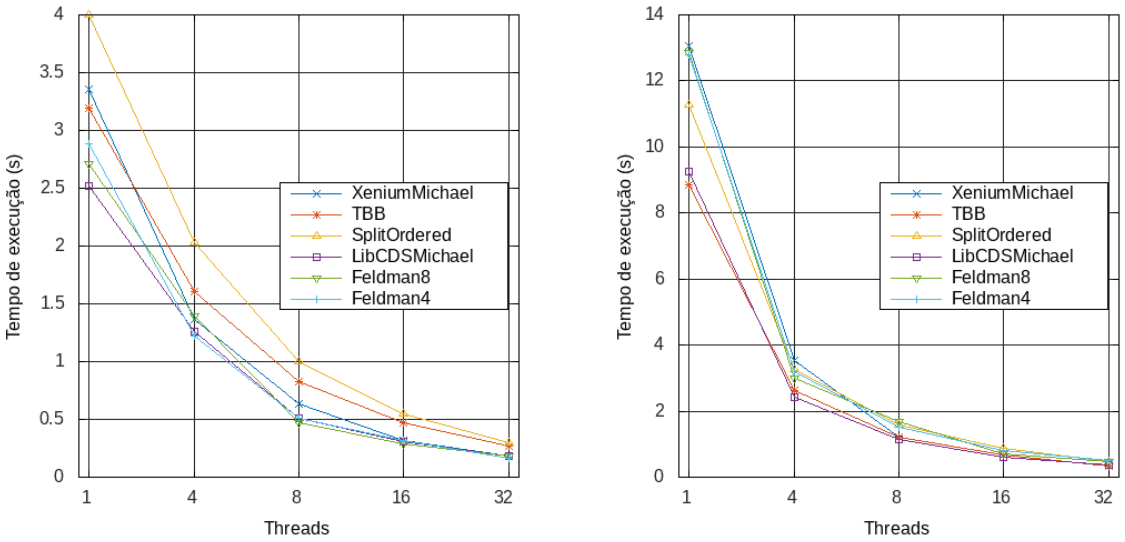
Devido ao baixo desempenho da implementação *STDLock* e ao grande número de falhas apresentados nas execução de testes da implementação de *hash table* presente na biblioteca *WFC*, os resultados dessas implementações não serão incluídos nos demais gráficos dessa seção, facilitando a leitura dos resultados das outras implementações. Os gráficos com todas as implementações testadas estão disponíveis no Apêndice C.

#### 5.2.2.1 Tamanho dos valores armazenados

Com a referência visual dos gráficos produzidos foi possível concluir sobre o comportamento das diferentes implementações quando variado o tamanho dos valores

armazenados. Os resultados indicaram que as implementações não bloqueantes tiveram o desempenho mais afetado pelo aumento do tamanho dos valores. Os cenários com pré-inserções das chaves, muitas chaves distintas e distribuição B, que é balanceada em buscas e inserções podem exemplificar o comportamento. As Figuras 21(a) e 21(b) permitem comparar os resultados com os diferentes tamanhos de valores.

Figura 21 – *Resultados dos testes com pré-inserções, muitas chaves distintas e distribuição B.*




(a) com valores pequenos. (b) com valores grandes.

Fonte: Elaborado pelo autor (2023).

Com valores pequenos (Figura 21(a)) implementações não bloqueantes, em especial as baseadas nas propostas de Michael e de Feldman foram capazes de apresentar resultados superiores aos obtidos pela implementação bloqueante da *TBB*. O melhor resultado foi de 0,168 segundos, obtido pela *Feldman4*, sendo 38% menor que o resultado de 0,270 segundos obtido pela *TBB*. Quando os valores armazenados passaram a ser grandes, na Figura 21(b) o melhor resultado passou a ser da implementação *TBB*, que obteve uma média de tempos de execução de 0,347 segundos, somente 28% maior que o tempo obtido com valores pequenos. A implementação *XeniumMichael* passou a apresentar o resultado de 0,348 segundos, valor 88% maior que o obtido no cenário anterior, mas ficando somente atrás do obtido resultado obtido pela *TBB*. Os resultados da *Feldman4* e *Feldman8* passaram ser acima de 0,45 segundos, sofrendo um aumento dos tempo de execução de ao menos 150%. A implementação *SplitOrdered* e a implementação *LibCDSMichael* sofreram com aumentos de 56% e 97% respectivamente.

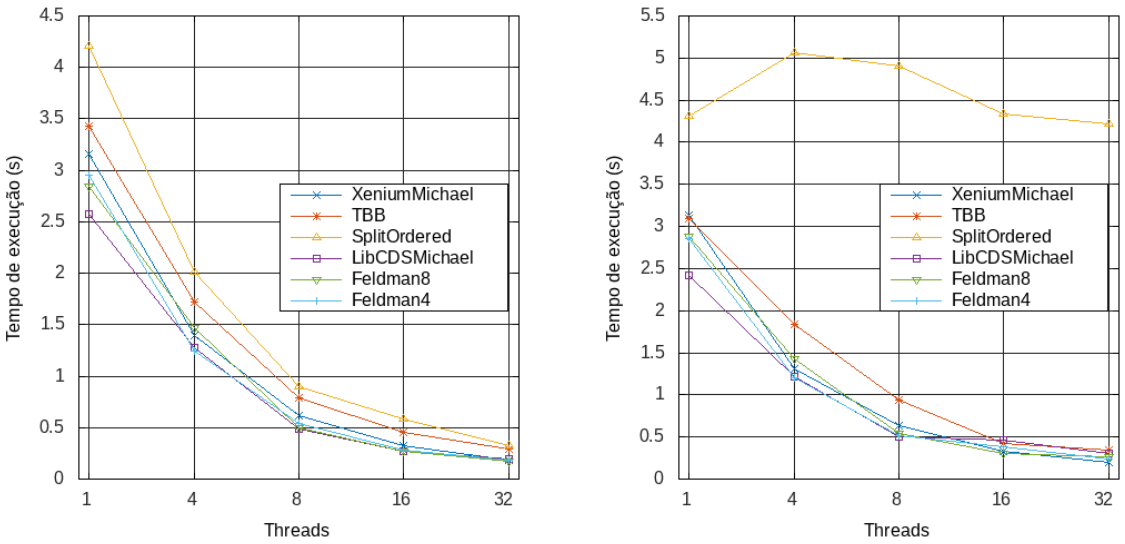
Os resultados indicam que, dentre as implementações testadas, a *TBB* sofre o menor impacto negativo devido ao aumento do tamanho dos valores armazenados. As figuras produzidas a partir dos resultados de outros cenários permitem observar

esse comportamento graficamente, bastando observar os cenários em pares com os mesmos parâmetros de teste, exceto pelo tamanho dos valores. O restante das figuras estão disponíveis no Apêndice C.

#### 5.2.2.2 Inserção prévia das chaves distintas

A implementação *SplitOrdered* aparentou ser pouco eficiente em cenários com tabelas inicialmente vazias e que tenham uma grande proporção de inserções e um grande número de chaves distintas. Esses cenários causam um número elevado de inserções de novas chaves, e a comparação com os resultados de cenários com poucas chaves distintas, com uma distribuição de operações mais intensa em buscas, ou com as tabelas já preenchidas indica que a inserção de um grande número de novas é muito custosa nessa implementação.

Figura 22 – *Resultados dos testes com muitas chaves distintas, valores pequenos e distribuição C.*




(a) com pré-inserções das chaves distintas (b) com tabelas inicialmente vazias

Fonte: Elaborado pelo autor (2023).

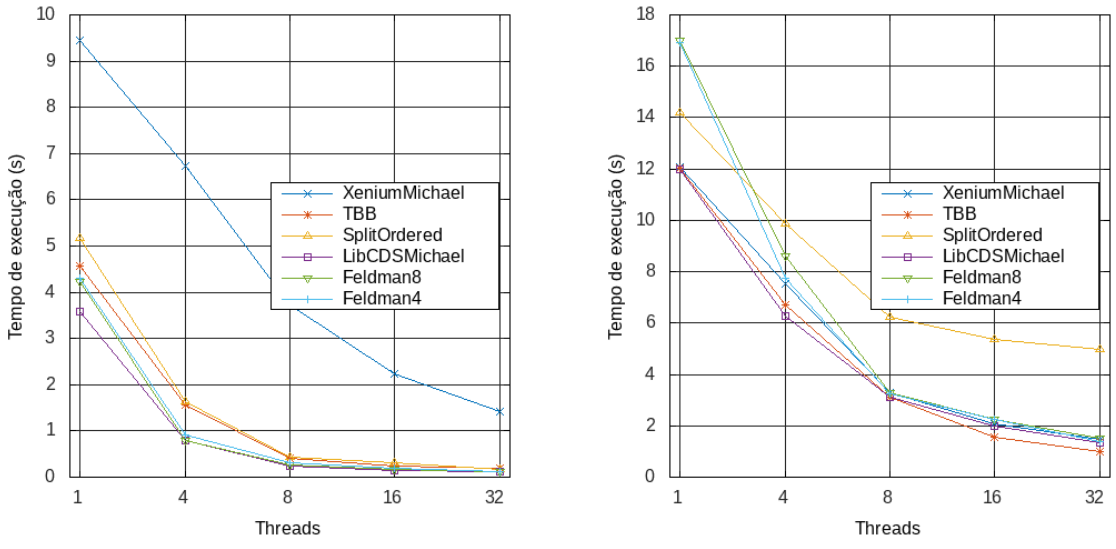
Comparando a Figura 22(a) com a Figura 22(b) é possível observar o comportamento. Com tabelas inicialmente vazias, muitas chaves distintas e distribuição de operações intensa em inserções (Figura 22(b)), a implementação *SplitOrdered* apresenta tempos de execução de mais de 4 segundos e não apresenta melhorias de desempenho com o aumento do número de *threads*, enquanto as outras implementações escalam bem e mostram resultados com menos de 0,5 segundos. Quando as tabelas são preenchidas previamente (Figura 22(a)) os resultados da *SplitOrdered* passam a ser próximos dos resultados obtidos pelas outras implementações, que não apresentaram diferenças tão significativas nas médias de tempos de execução. É possível observar esse mesmo comportamento ao considerar cenários com muitas chaves

distintas e distribuição de operações B ou C, e comparar dentre eles os cenários com tabelas inicialmente vazias com os cenários com chaves previamente inseridas.

#### 5.2.2.3 Proporção de operações

Os resultados indicam que o aumento da proporção de inserções tende a aumentar a média dos tempos de execução de qualquer implementação. Inserções podem exigir alocação de novas partes da estrutura de dados e agregarem o custo da realização de uma travessia até o local de inserção. Nos resultados dos testes intensos em inserções, com tabelas vazias, muitas chaves distintas, valores grandes e distribuição de operações com maior proporção de buscas (Figura 23(a)), o melhor resultado foi de 0,111 segundos, obtido com 32 *threads* pela implementação *Feldman8*. O resultado da *TBB* foi o segundo pior, com uma média de tempo de execução igual à 0,188 segundos, 69% maior que o melhor resultado. Os resultados da implementação *XeniumMichael* foram muito distantes dos obtidos pelas outras implementações, com 1,410 segundos. A Figura 23(b) mostra os resultados quando a distribuição de operações passa a ser intensa em inserções. O comportamento da implementação *SplitOrdered* já foi explicado. Com 32 *threads*, a implementação *XeniumMichael* mantém um resultado semelhante ao obtido anteriormente (1,451 segundos), e as outras implementações aproximam seus resultados desse tempo de execução. A média dos tempos de execução da implementação da *TBB* foi de 0,981 segundos, que é ao menos 27,55% menor que as médias obtidas pelas outras implementações, que apresentam resultados maiores que 1,354 segundos.

Figura 23 – *Resultados dos testes com tabelas inicialmente vazias, muitas chaves distintas e valores grandes*

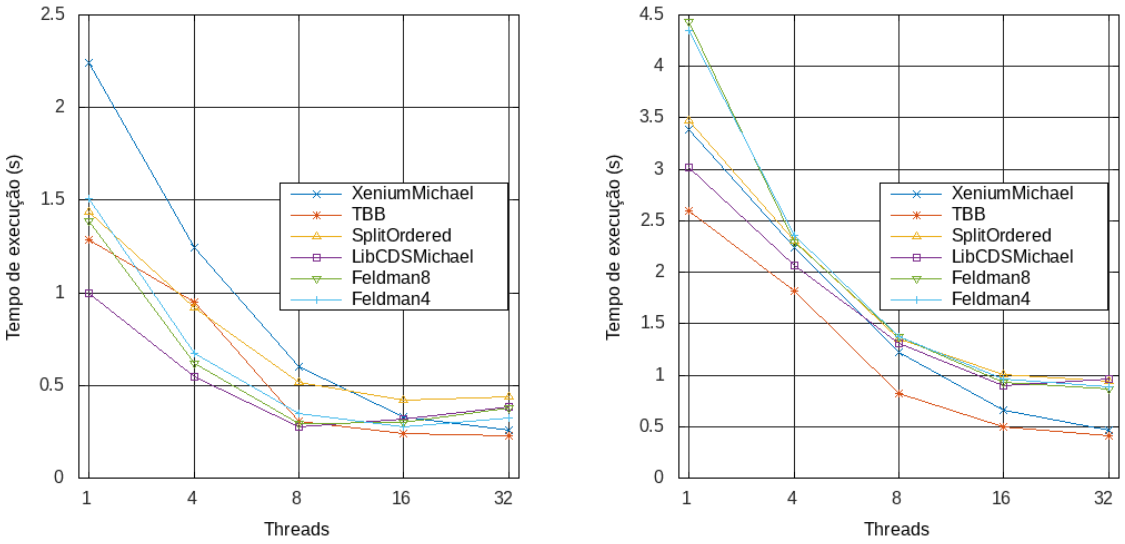



(a) com distribuição A. (b) com distribuição C.

Fonte: Elaborado pelo autor (2023).

Outros cenários também tornam evidente o impacto que a mudança na distribuição de operações tem nos resultados. Comparando a Figura 24(a) com a Figura 24(b) é possível observar as curvas das implementações *TBB* e *XeniumMichael* reduzindo em relação às curvas das outras implementações quando a distribuição de operações passa de A para B. Todas as implementações sofreram com perda de desempenho devido à mudança do cenário, e o efeito é causado por uma maior queda de desempenho das implementações da biblioteca *LibCDS* em relação as outras implementações.

Figura 24 – *Resultados dos testes com pré inserções, poucas chaves distintas e valores grandes*




(a) com distribuição A. (b) com distribuição B.

Fonte: Elaborado pelo autor (2023).

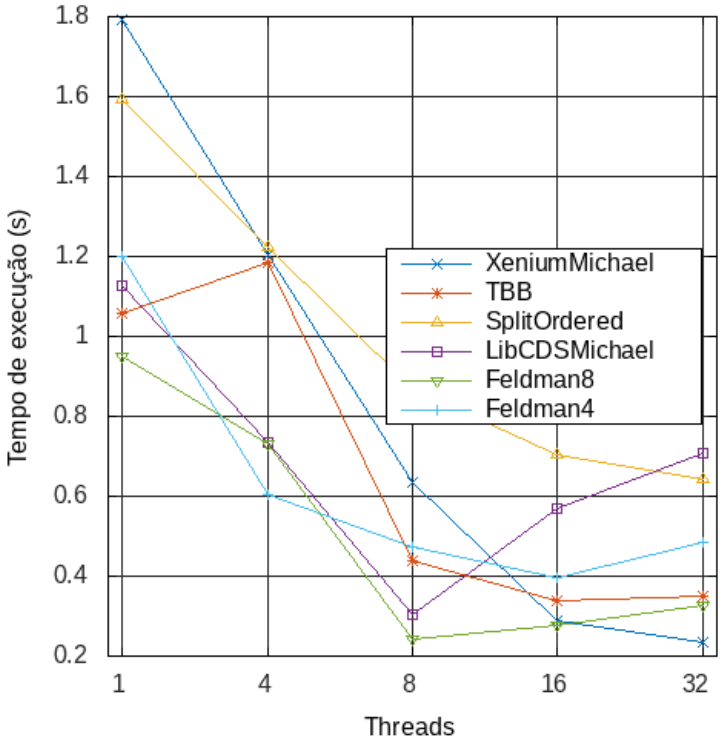
#### 5.2.2.4 Número de chaves distintas

Cenários com poucas chaves distintas tornam mais comum acessos à uma mesma região da estrutura por diferentes *threads*, que pode ser causada por colisões na função de dispersão ou acessos à um mesmo nó durante uma busca, sendo uma situação com maior contenção. Em uma implementação bloqueante isso pode aumentar o tempo que as *threads* passam em espera, e em implementações não bloqueante isso pode causar um aumento no número de passos necessários para completar as operações. Esses cenários também evitam a necessidade de crescimento das estruturas, já que são poucas chaves distintas parar armazenar.

Comparando a Figura 21(b) com a Figura 24(b), é visível que a passagem do cenário com muitas para o cenário com poucas chaves distintas resultou na redução da melhoria obtida pelas implementações da *LibCDS* com o aumento do número de *threads* a partir de 16, provocando um afastamento dos resultados das implementações *XeniumMichael* e *TBB*. O efeito da redução do número de chaves é mais perceptível

na comparação do cenário da Figura 21(a) com o da Figura 25. A implementação da *TBB* passa a sofrer redução de desempenho com as passagens de 1 para 4 e de 16 para 32 *threads*, e outras implementações sofrem com redução de desempenho já com a passagem de 4 para 8 *threads*.

Figura 25 – *Resultados dos testes com pré inserções, poucas chaves distintas, valores pequenos e distribuição B.*

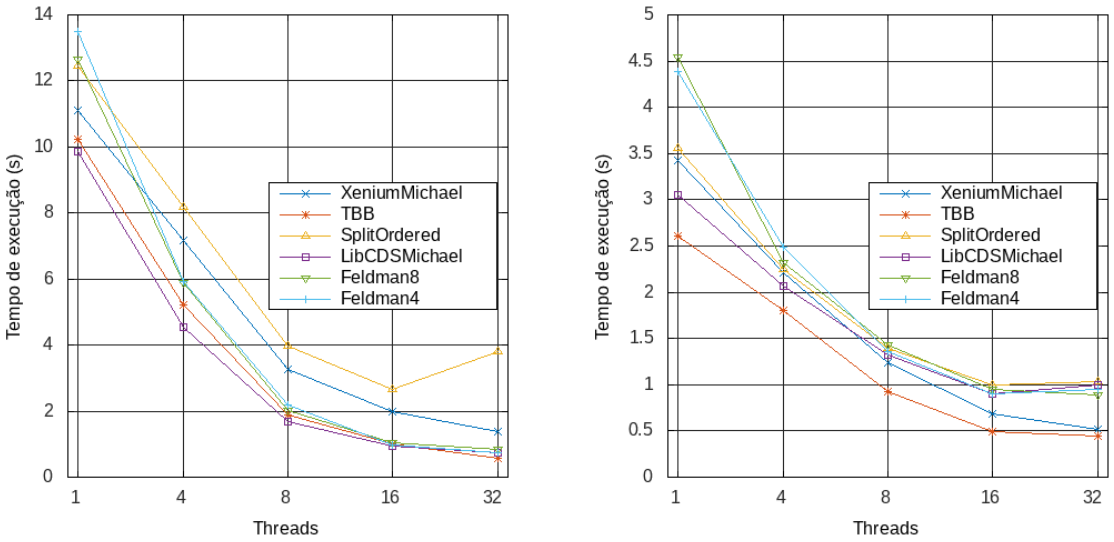



Fonte: Elaborado pelo autor (2023).

Os resultados dos testes em cenários com tabelas vazias, valores grandes e distribuição de operações B são apresentados nas Figuras 26(a) e 26(b). Essa distribuição apresenta uma quantidade considerável de inserções, fazendo com que os resultados do cenários com muitas chaves distintas (Figura 26(a)) apresente tempos de execução maiores que o cenário com poucas chaves distintas (Figura 26(b)) quando o número de *threads* não é tão grande.

Observando as alterações decorrentes do aumento do número de chaves distintas, os resultados de mais de 10 segundos com uma *thread* para ser distribuídos entre 2,5 e 5 segundos. Os melhores resultados da implementação *LibCDSMichael* e das implementações baseadas na proposta de Feldman passaram de próximos dos 0,75 segundos para próximos dos 0,9 segundos. A implementação *XeniumMichael* passou de 1,385 segundos para 0,523 segundos, a implementação da *TBB* passou de 0,576 segundos para 0,441 segundos, e a implementação *SplitOrdered* passou de aproximadamente 2,5 segundos para 0,994 segundos.

Figura 26 – *Resultados dos testes com tabelas inicialmente vazias, valores grandes e distribuição B.*




(a) com muitas chaves distintas.

(b) com poucas chaves distintas.

Fonte: Elaborado pelo autor (2023).

Nesse cenários, as implementações *XeniumMichael* e *TBB* aparentaram se beneficiar da redução do número de chaves distintas, enquanto as implementações da *LibCDS*, com exceção da implementação *SplitOrdered*, foram prejudicadas, apresentando inclusive redução de desempenho com a passagem de 16 para 32 *threads*.

# 6 ESTUDO DE CASO

Para avaliar o impacto no desempenho que a implantação de estruturas de dados não bloqueantes causam em um programa concorrente foi realizado um estudo de caso considerando uma aplicação de armazenamento de pares chaves e valores em memória. Em (TROMBETA, J.; MENDIZABAL, O., 2020) foi apresentado um simulador para comparar diferentes políticas de escalonamento em replicação máquina de estados paralela, que foi capaz de verificar a eficiência de técnicas de corte em grafos como parte da estratégia de escalonamento. Os resultados obtidos motivaram a proposta de técnica de escalonamento discutida em (GABRIEL TROMBETA; MACHADO MENDIZABAL, 2020), cuja implementação foi posteriormente apresentada e avaliada no trabalho (TROMBETA, J. G., 2021). Esse estudo de caso objetiva reduzir a contenção do escalonador apresentado nesse trabalho. Para isso, um novo algoritmo é proposto e comparado com o método original.

Considerando o programa modificado, são comparadas duas versões: uma com as mesmas implementações de estruturas de dados originalmente utilizadas; e outra utilizando implementações não bloqueantes com funções equivalentes. As estruturas de dados que necessitam de sincronização no escalonamento são filas, enquanto durante os acessos a chaves e valores é necessário garantir sincronização em uma tabela de dispersão.

## 6.1 PROTÓTIPO DE *KEY-VALUE STORE*

O trabalho (TROMBETA, J. G., 2021) apresentou uma avaliação comparativa de diferentes algoritmos de particionamento em grafos para um protótipo de serviço de armazenamento de pares chave-valor. O estudo conduziu experimentos com o objetivo de analisar como a reconfiguração do esquema de particionamento em meio a execução afeta a vazão do sistema. O protótipo é configurável, sendo possível escolher o número de partições, o intervalo de particionamento, o número de chaves inicialmente no sistema e o algoritmos de particionamento, dentre quatro algoritmos disponíveis. Foram apresentadas duas versões do protótipo, uma versão com execução local, e uma versão distribuída.

O protótipo local conta com os seguintes componentes:

- *Worker threads* associadas a partições. Essas *threads* recebem requisições em uma fila e as executam. Requisições que dependem de chaves em uma única partição são executadas imediatamente, enquanto requisições que dependem de chaves em outras partições são coordenadas junto às demais *worker threads*;
- O *pattern tracker*, uma *thread* responsável pelo registro do padrão de aces-

sos as chaves. Esse componente conta com uma fila, da qual retira requisições usadas e atualiza o grafo que representa a carga de trabalho;

- Um escalonador, que recebe requisições e repassa, em ordem, tanto para as *worker threads* envolvidas, coordenando a sincronização necessária, quanto para a *thread* que gerencia o grafo da carga de trabalho. A partição encarregada do gerenciamento de determinada chave é definida por um mapa de partições gerenciado pelo escalonador. Em um intervalo $\delta p$, o escalonador sincroniza com a *thread* responsável pela atualização do grafo, realiza um reparticionamento e atualiza o mapa de partições. Antes de voltar a escalonar seguindo o novo mapa de partições, o escalonador coordena a sincronização de todas as *worker threads*, garantindo a execução segura das requisições repassadas às *worker threads* seguindo o esquema de partições antigo.

As requisições recebidas pelo escalonador podem ser de leitura, escrita ou escaneamento, contendo necessariamente uma chave. Adicionalmente, requisições de escrita contém um valor, e requisições de escaneamento contém um tamanho.

A *worker thread* encarregada do tratamento de uma requisição é decidida pelo escalonador, que considera um mapa de chaves à identificadores de *Threads/Partições* nessa tomada de decisão. O procedimento de reconfiguração do particionamento, ou reparticionamento, faz à modificação desse mapeamento como base nas informações contidas no grafo construído pelo *pattern tracker*.

Considerando a requisição *request*, *type*(*request*) = *READ* indica a solicitação do valor associado à chave *key*(*request*). Quando *type*(*request*) = *WRITE*, a requisição solicita a inserção do par (*key*(*request*), *value*(*request*)) no banco. Quando *type*(*request*) = *SCAN*, a requisição solicita todos os valores associados à sequência de chaves [*key*(*request*), ...,*key*(*request*) + *size*(*request*) – 1].

Para realizar sincronizações com a *thread* de registro de acessos e coordenar a sincronização das *worker threads* no atendimento de requisições *SCAN*, o escalonador faz uso de requisições de sincronização. Esse tipo de requisição não é solicitada pelo cliente, sendo criada somente pelo sistema para gerenciamento das sincronizações. Quando uma *worker thread* retira de sua fila uma requisição *request* com *type*(*request*) = *SYNC*, ela espera na barreira *barrier*(*request*). Para garantir consistência no atendimento de requisições *SCAN* é necessário:

1. inserir uma requisição de sincronização na fila de cada *worker thread* que gerencia uma das chaves no intervalo da requisição;
2. inserir a requisição de *SCAN* na fila de somente uma das *worker threads* envolvidas;
3. inserir uma segunda requisição de sincronização na fila de cada *worker thread* envolvida.

A primeira sincronização garante que todas as partições estão em um estado consistente com o atendimento das requisições em ordem de chegada. A segunda sincronização previne que as *worker threads* deixem esse estado consistente antes do atendimento da requisição de *SCAN*.

O Algoritmo 1 apresenta a inicialização de variáveis comuns aos componentes do protótipo local, que inclui a criação o *PatternTracker* e das *worker threads*, enquanto o Algoritmo 2 apresenta o funcionamento do do escalonador. O escalonador considera o conjunto *Workers* de *n worker threads*, conjunto *KeyToPartition*, que associa chaves à partições, e o *PatternTracker*, que é responsável por atualizar o *WorkloadGraph*.

**Algoritmo 1** Init

```
    variables:
 1: n                                                  ▷ Número de partições.
 2: δp                                                 ▷ Intervalo entre reparticionamentos.
 3: Workers[0,...,n−1]                                 ▷ n worker threads.
 4: patternTracker                                     ▷ Pattern tracker.
 5: Graph ← ∅                                          ▷ Grafo que caracteriza a carga de trabalho.
 6: TrackerQ ← ∅                                       ▷ Fila bloqueante de requisições do Pattern Tracker
 7: KeyToPartition ← ∅                                 ▷ Conjunto de tuplas que associam chaves à partições.
 8: rrCounter ← 0                                      ▷ Contador usado para associar uma partição à uma nova chave.
 9: dispatched ← 0                                     ▷ Número de requisições despachadas às worker threads.
    initialization:
10: ∀p ∈ {0,...,n−1}, Workers[p] ← new WorkerThread    ▷ Inicializa n worker threads
11: patternTracker ← new PatternTracker                ▷ Inicializa o Pattern Tracker
```

Fonte: Elaborado pelo autor (2023).

**Algoritmo 2** Scheduler

```
 1: procedure SCHEDULE(request)
 2:     DISPATCH(request)                  ▷ Envia requisição as partições e ao PatternTracker
 3:     dispatched ← dispatched + 1                                  ▷ Incrementa contador
 4:     if dispatched mod δp = 0 then      ▷ Foram processadas δp requisições
 5:         syncRequest ← new sync request for 2 threads
 6:         trackerQ.enqueue(syncRequest)                            ▷ Solicita
 7:         barrier(syncRequest).wait()    ▷ Espera a parada do PatternTracker
 8:         KeyToPartition ← PARTITIONING(workloadGraph, n)          ▷ Realiza o
    particionamento para n partições com a s informações de workloadGraph
 9:         SYNC(Workers)                  ▷ Sincroniza todas as WorkerThreads
10:     end if
11: end procedure
```

Fonte: Elaborado pelo autor (2023).

**Algoritmo 3** Procedimentos auxiliares para o escalonamento

```
1:  procedure SYNC(Involved)
2:      syncRequest ← new sync request for |Involved| threads        ▷ Requisição de sincronização.
3:      ∀worker ∈ Involved, worker.enqueue(syncRequest)    ▷ Entrega da requisição de sincronização.
4:  end procedure
5:  procedure DISPATCH(request)
6:      if type(request) = WRITE ∧ ∄(key(request), ∗) ∈ KeyToPartition then▷ A chave é do tipo WRITE e não está associada à nenhuma partição
7:          KeyToPartition ← KeyToPartition ∪ {(key(request), rrCounter)}   ▷ Associa a chave à uma partição
8:          rrCounter ← (rrCounter + 1)  mod n                    ▷ Atualiza o contador
9:      end if
10:     if type(request) = SCAN then                              ▷ A requisição é do tipo SCAN
11:         Involved ← {Workers[p] | (key, p) ∈ KeyToPartition ∧ key ∈ {key(request), ..., key(request) + size(request) − 1}}
12:         SYNC(Involved)                                        ▷ Sincroniza as workers envolvidas
13:         (key(request), p) ∈ KeyToPartition, Workers[p].enqueue(request)   ▷ Despacha a requisição
14:         SYNC(Involved)                                        ▷ Sincroniza novamente
15:     else
16:         (key(request), p) ∈ KeyToPartition, Workers[p].enqueue(request)   ▷ Despacha a requisição
17:     end if
18:     trackerQ.enqueue(request)                                 ▷ Entrega ao PatternTracker
19: end procedure
```

Fonte: Elaborado pelo autor (2023).

O procedimento *Schedule* (linha 1) atende uma requisição, enviando-as para as *worker threads* e para o *PatternTracker*. A cada $\delta p$ requisições processadas pelo o escalonador e pelo *PatternTracker* é feita uma reconfiguração do particionamento.

Para que o escalonador possa realizar o particionamento com um grafo que considera todas as requisições recebidas é necessário garantir que o *pattern tracker* tenha também processado todas as requisições. Para isso o escalonador envia uma requisição de sincronização ao *pattern tracker* e espera na barreira (linhas 6 7). A função *Partitioning* faz uso de um método de particionamento de grafos sobre *Graph* para criar um conjunto que associa as chaves às *n* partições de forma à balancear a carga de trabalho. A chamada de *Partitioning*, substituindo *KeyToPartition* pelo novo mapeamento (linha 8), é sucedida por uma sincronização de todas *worker threads* para evitar que requisições sejam processadas com diferentes mapeamentos ao mesmo tempo, o que poderia causar acessos simultâneos a uma mesma chave.

**Algoritmo 4** PatternTracker

```
    on thread run:
1:  while true do
2:      request ← trackerQ.waitAndPop() ▷ Espera pela próxima requisição e retira da
    fila
3:      if type(request) = SYNC then                    ▷ Requisição de sincronização
4:          barrier(request).wait()                      ▷ Sincroniza com Scheduler
5:      else                                  ▷ Requisições WRITE, READ ou SCAN
6:          UPDATEGRAPH(Graph, request)                  ▷ Atualiza o grafo
7:      end if
8:  end while
```

Fonte: Elaborado pelo autor (2023).

O Algoritmo 3 descreve procedimentos auxiliares ao escalonamento. O procedimento *Dispatch*, invocado no procedimento *Schedule*, tem as funções de:

- Associar novas chaves a partições em regime *round robin* (linhas [6,10]);
- Escalonar as requisições, coordenando a sincronização nas requisições do tipo *SCAN* com auxílio do método *Sync* (linhas [10,18]);
- Enviar a requisição para o *PatternTracker* (linha 18).

O Algoritmo 4 apresenta o funcionamento do *pattern tracker*, que é responsável por atualizar o grafo. A chamada *waitAndPop* (linha 2) sobre a fila *trackerQ* espera que a fila não esteja vazia, então retira uma requisição. Quando a requisição é de sincronização, é esperado na barreira, quando é de qualquer outro tipo, atualiza-se o grafo com a invocação de *UpdateGraph*. As modificações que o grafo sofre são descritas no trabalho (TROMBETA, J. G., 2021).

## 6.2 ESCALONAMENTO DE BAIXA CONTENÇÃO

O algoritmo de escalonamento do protótipo sofre com a necessidade de sincronização em diferentes momentos. Foi adotado o uso de *mutexes* para sincronizar os acessos às filas das *worker threads* e à fila do *pattern tracker*. A estrutura usada para armazenar os pares chave-valor é uma tabela de dispersão da biblioteca TBB, que usa um mecanismo bloqueante de granularidade fina. Além dos acessos às estruturas de dados, antes de realizar um particionamento, o escalonamento sincroniza com o *pattern tracker* em uma barreira.

Outro ponto do algoritmo de escalonamento que pode reduzir o desempenho é a execução do algoritmo que define o esquema de partições. Durante o particionamento, que pode durar bastante tempo, novas requisições não são escalonadas, o que pode causar redução da vazão devido à falta de requisições despachadas às *worker threads*.

Neste trabalho, para reduzir os custos de sincronização no escalonamento são introduzidas estruturas de dados não bloqueantes na implementação do protótipo local, além de modificados os algoritmos dos componentes do protótipo para que as operações bloqueantes usadas se limitem à incrementos e tentativas de decremento de semáforos.

A técnica se baseia em realizar o particionamento em uma linha de execução diferente da de escalonamento, usando uma cópia do grafo de carga de trabalho tirada antes do início da execução do algoritmo de corte. Isso retira a necessidade de esperas do procedimento de escalonamento, delegando a cópia do grafo ao *pattern tracker* e o particionamento a uma nova *thread*, o *particionador*. Ao término da construção de um novo mapeamento, o particionador sinaliza o escalonador, que somente troca o mapa utilizado nas decisões de escalonamento.

A técnica mantém o escalonamento vivo durante o particionamento em troca da adição de uma nova linha de processamento. A reconfiguração é dada em função da carga de trabalho no início do particionamento, e como requisições continuarão a ser recebidas e atendidas, o escalonador fará a troca do mapeamento para um mapa baseado em uma carga de trabalho que reflete um padrão de acessos antigo.

**Algoritmo 5** New Init

```
    variables:                          ▷ Mesmas vaiáveis de Init (Algoritmo 1), com adição de:
1:  repartSem ← new semaphore with 0 permits
2:  updateSem ← new semaphore with 0 permits
3:  updatedSem ← new semaphore with 0 permits
4:  copyGraphMutex ← new mutex
5:  partitioner                         ▷ Thread dedicada ao reparticionamento
6:  UpdatedKeyToPartition ← ∅           ▷ Mapeamento atualizado
7:  GraphCopy ← ∅                       ▷ Copia de Graph
    initialization:                     ▷ Mesma de Init (Algoritmo 1), com adição de:
8:  partitioner ← newpartitioner
```

Fonte: Elaborado pelo autor (2023).

Os Algoritmos 6, 7 e 8 apresentam o funcionamento do protótipo modificado. O novo escalonador conta com as mesmas variáveis do escalonador antigo, além de algumas adições. Os procedimentos auxiliares de *Dispatch* e *Sync* continuam os mesmos do algoritmo anterior.

No Algoritmo 6, o procedimento *Schedule* não é mais responsável pela realização do particionamento, e somente envia uma requisição do tipo *REPART* ao novo *pattern tracker*, na linha 9.

O novo *pattern tracker*, no Algoritmo 7, agora trata de requisições do tipo *REPART*. No procedimento *Track*, quando uma requisição desse tipo é removida da fila, é

**Algoritmo 6** New Scheduler

```
 1: procedure SCHEDULE(request)
 2:     DISPATCH(request)          ▷ Envia requisição as partições e ao PatternTracker
 3:     if updateSem.tryWait() then          ▷ Detectado o fim de um reparticionamento
 4:         KeyToPartition ← UpdatedKeyToPartition          ▷ Atualiza o mapeamento
 5:         updatedSem.post()  ▷ Sinaliza o término da atualização ao reparticionador
 6:     end if
 7:     if dispatched  mod δp = 0 then          ▷ Foram processadas δp requisições
 8:         repartRequest ← new repart request
 9:         trackerQ.enqueue(repartRequest)          ▷ Envia reparticionamento ao
    PatternTracker
10:     end if
11: end procedure
```

Fonte: Elaborado pelo autor (2023).

**Algoritmo 7** New PatternTracker

```
    on thread run:
 1: while true do
 2:     request ← trackerQ.waitAndPop() ▷ Espera pela próxima requisição e retira da
    fila
 3:     if type(request) = REPART then          ▷ Requisição de reparticionamento
 4:         copyGraphMutex.lock()          ▷ Bloqueia o mutex da cópia do grafo
 5:         GraphCopy ← Graph          ▷ Copia o grafo
 6:         copyGraphMutex.unlock()          ▷ Libera o mutex da cópia do grafo
 7:         repartSem.post()          ▷ Libera início do reparticionamento
 8:     else          ▷ Requisições WRITE, READ ou SCAN
 9:         UPDATEGRAPH(Graph, request)          ▷ Atualiza o grafo
10:     end if
11: end while
```

Fonte: Elaborado pelo autor (2023).

feita uma cópia de *Graph* e então sinalizado o *Partitioner*, com a liberação do semáforo *repartSem*, nas linhas 5 e 7. A cópia é feita com a trava de *copyGraphMutex* para evitar que o particionamento seja feito com uma cópia em estado inválido. Não haverá sincronizações entre escalonador e *PatternTracker*.

O Algoritmo 8 mostra o funcionamento do particionador. Na linha 4, após ser liberado pelo *pattern tracker*, um novo mapeamento construído com as informações da cópia do grafo é armazenado em *UpdatedKeyToPartition*. Como explicado no parágrafo anterior, o particionamento é feito com a trava de *copyGraphMutex* para evitar o uso de uma cópia em estado inválido. O escalonador é sinalizado por meio do se-

**Algoritmo 8** Partitioner

```
    on thread run:
1:  while true do
2:      repartSem.wait()                                          ▷ Espera comando
3:      copyGraphMutex.lock()                  ▷ Bloqueia o mutex da cópia do grafo
4:      UpdatedKeyToPartition ← PARTITIONING(GraphCopy, n)        ▷ Cria um
    mapeamento atualizado usando a cópia do grafo
5:      copyGraphMutex.unlock()                ▷ Libera o mutex da cópia do grafo
6:      updateSem.post()                       ▷ Indica termino do reparticionamento
7:      updatedSem.wait()      ▷ Espera pelo término da atualização do mapeamento
8:  end while
```

Fonte: Elaborado pelo autor (2023).

máforo *updateSem*, e o particionador espera que o escalonador sinalize o término da troca do mapa de partições em uso, com o objetivo de impedir acesso concorrente à *UpdatedKeyToPartition*.

Quando o escalonamento detectar o término de um particionamento, na linha 3 do Algoritmo 6, *KeyToPartition* será substituído por *UpdatedKeyToPartition*. Depois, o escalonador sinaliza o termino da atualização ao particionador.

## 6.3 AVALIAÇÃO EXPERIMENTAL

Foram realizados experimentos com as mesmas cargas de trabalho sobre três implementações diferentes do protótipo local. A primeira implementação é a versão base, apresentada no trabalho (TROMBETA, J. G., 2021), que utiliza os algoritmos descritos na seção Protótipo de *Key-Value Store*. Na segunda, o particionamento e escalonamento são desacoplados utilizando os algoritmos da seção Escalonamento de Baixa Contenção, mantendo as implementações bloqueantes das estruturas de dados. A terceira utiliza os algoritmos da seção Escalonamento de Baixa Contenção em conjunto com estruturas de dados não bloqueantes.

Qualquer versão do protótipo utiliza uma tabela de dispersão para armazenar os pares chave-valor em memória. A versão bloqueante utilizada é a disponível na biblioteca TBB, e já fazia parte da implementação base do protótipo. A versão não bloqueante é a implementação baseada na proposta de Feldman da biblioteca LibCDS, escolhida por apresentar desempenho competitivo com a TBB em diferentes cenários dos *microbenchmarks* e ainda ser extensível. A implementação de Feldman foi configurada com um vetor de $2^{16}$ posições no primeiro nível, em vista da grande quantidade de elementos esperada, e $2^8$ posições nos níveis subsequentes.

O procedimento de escalonamento interage com as filas de requisições das *worker threads* e com a fila do *Pattern Tracker*. Cada uma dessas filas é acessada

por somente duas *threads*, uma *thread* produtora, o escalonador, e uma consumidora, correspondente a um dos outros componentes do sistema. Os elementos das filas são estruturas de 31 bytes que armazenam informações sobre a requisição escalonada. A implementação de fila bloqueante utilizada é baseada na fila disponível na STL com um *mutex* bloqueando a cada operação. Nos cenários com duas *threads* da experimentação com *microbenchmarks*, essa implementação apresentou os melhores resultados. A implementação não bloqueante utilizada foi a da biblioteca TBB, que apresentou os segundo melhor resultado nesses cenários.

Nos experimentos utilizados para comparar as versões do protótipo, METIS (KARYPIS; KUMAR, 1998) foi o algoritmo de particionamento utilizado em todos os experimentos, o número de pares chave-valor inicialmente no sistema foi de $10^6$, o número de *worker threads* foi 8, o tamanho das chaves armazenadas é de 4 bytes, e o tamanho dos valores armazenados é de 4 kbytes. As cargas de trabalho utilizadas nos testes reproduzem perfis de acesso baseados nas cargas D e E do Yahoo! Cloud Serving Bechmark (YCSB) (COOPER *et al.*, 2010). Na carga D, 5% das requisições são escritas de novos pares e 95% são leituras. A distribuição das chaves acessadas é denominada *Latest*, na qual as leituras são principalmente de pares recentemente inseridos no sistema. Na carga E, 95% das requisições são escaneamentos, com variação do tamanho uniformemente entre 2 e 8, e 5% de atualizações de chaves já presentes no sistema. A carga E é adequada para avaliar a capacidade do particionamento de reduzir o número de sincronizações entre *worker threads*, enquanto a carga D pode avaliar o desempenho perante o crescimento do número de pares armazenados. Foram realizados experimentos variando os seguintes parâmetros:

- carga de trabalho (carga D com 50.000.000 requisições ou carga E com 5.000.000 requisições)
- intervalo de particionamento $\delta p$ (10.000, 500.000, 1.000.000, 5.000.000 ou 10.000.000)

Nos experimentos com a carga E foram realizados somente 5.000.000 devido a grande quantidade de requisições de escaneamento, que geram muita contenção e tornam o tempo de execução muito maior. Por conseguinte, os intervalos de particionamento não passaram de 1.000.000 para essa carga.

Os testes foram realizados em um computador com dois processadores Intel Xeon E5-2630, cada um com 2.4 GHz, 8 núcleos e 20 MB cache, e 64 GB de memória RAM DDR4. O Sistema operacional usado foi o Ubuntu v22.04 de 64 bits. O programa de testes foi desenvolvido na linguagem C++17 e compilado com o *gcc v9.4.0*. O parâmetro `-mcx16` foi usado para garantir disponibilidade da instrução *CAS* às bibliotecas.

Os protótipos usados nos experimentos estão armazenados de maneira pública na plataforma GitHub, e podem ser acessados pelo endereço `https://github.com/`

`douglaspereira04/kvpaxos`. As versões que utilizam as implementações de estruturas de dados bloqueantes estão no *branch simplified-replica*, enquanto a versão que utiliza as variantes não bloqueantes das estruturas de dados está disponível no *branch lock-free-q*.

## 6.4 RESULTADOS

Dos experimentos foram obtidos informações sobre a vazão do sistema ao longo da execução, coletando o número de requisições atendidas à cada segundo. Com os dados foram produzidos gráficos que apresentam vazão em número de operações, no eixo $y$, e o tempo decorrido em segundos, no eixo $x$. Nos gráficos, a versão base é denominada *KVStore*, a versão modificada com estruturas de dados bloqueantes é denominada *KVStore* ||, e a versão modificada com estruturas de dados não bloqueantes é chamada *KVStore* || *NB*.

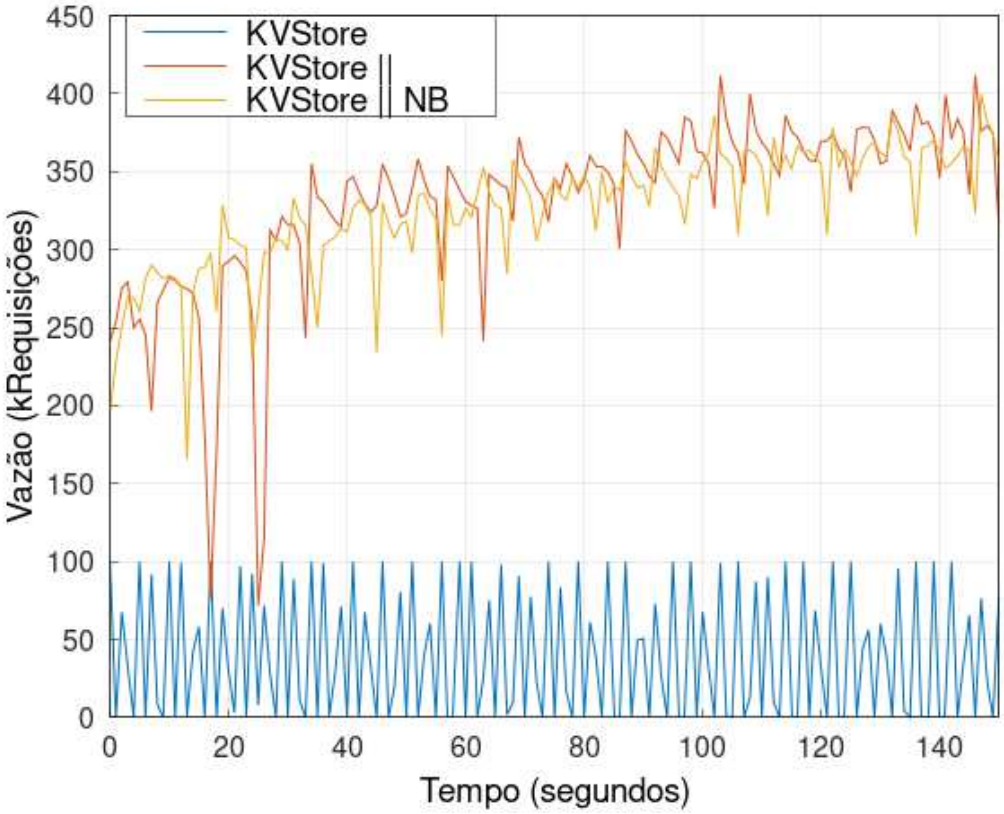
### 6.4.1 Execução da carga D

A Figura 27 apresenta resultados para experimentos com carga D e $\delta p$ = 100.000. A versão base não se adapta bem à intervalos de particionamento pequenos, e fica com vazão aparentemente limitada à $\delta p$, como mostra a curva de *KVStore*. A aparente limitação ocorre devido ao longo tempo tomado pelos procedimentos de particionamento, que chegaram à atingir duração de mais de 5 segundos, o que tornou a vazão do sistema zero por um longo período. Em função disso, a versão base ultrapassou os 45 minutos de execução.

Ambas as implementações que utilizam os novos algoritmos apresentaram um comportamento bem diferente da primeira versão. O tempo de execução para o mesmo número de operações com a carga D foi próximo de 2 minutos e 30 segundos. Diferentemente do que acontece com a versão base, a vazão do sistema não parece limitada a $\delta p$ já que não há espera pelo particionamento.

Nas versões novas do protótipo, o escalonamento das 50 milhões de requisições finalizou em menos de 100 segundos, enquanto na versão base, o escalonamento termina próximo do fim da execução. Com $\delta p$ = 100.000, a implementação que utiliza estruturas de dados não bloqueante levou cerca de 100 segundos para finalizar o escalonamento, enquanto a versão com a nova técnica que utiliza estruturas de dados bloqueantes levou somente 80 segundos. Com um valor de $\delta p$ de 10.000.000, os tempos foram de 130 segundos para a implementação com estruturas não bloqueantes e 116 segundos para a implementação que usa bloqueios nas estruturas de dados. Apesar de que ambas as implementações que utilizam o novo método de escalonamento apresentaram desempenho superior à implementação base, a implementação do protótipo que adota estruturas de dados não bloqueantes registrou tempos de exe-

Figura 27 – Resultados dos experimentos com carga D e $\delta p$ = 100.000.



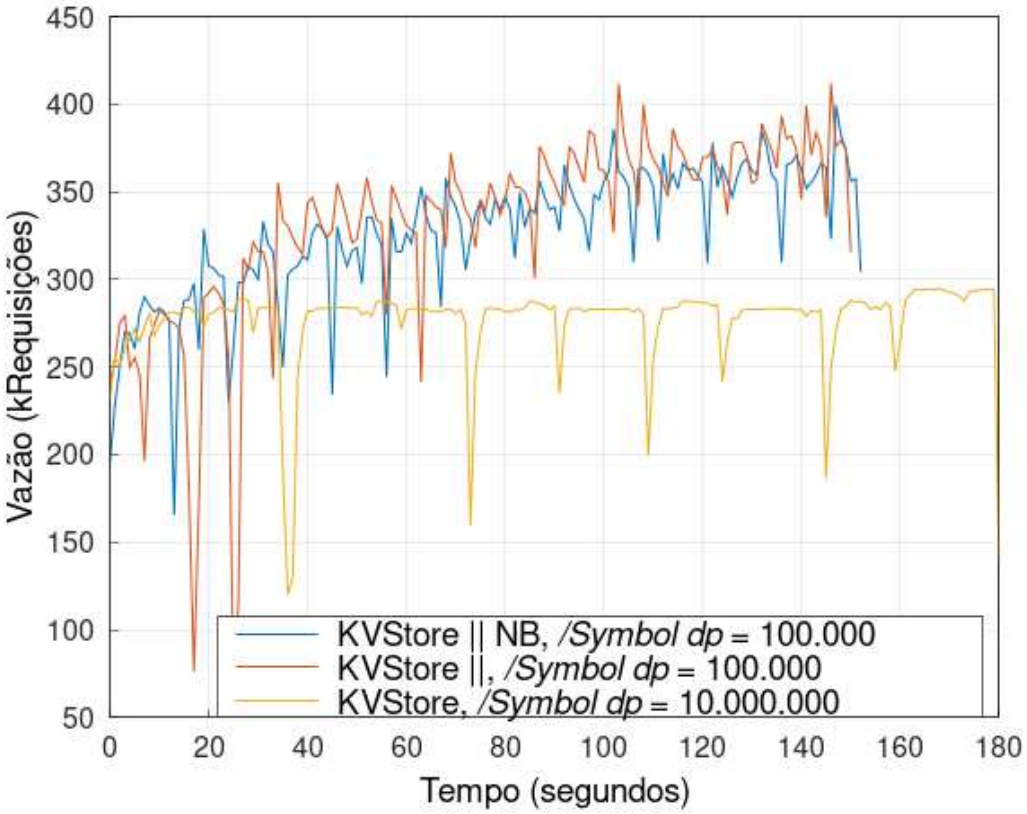

Fonte: Elaborado pelo autor (2023).

cução maior que a que utiliza implementações bloqueantes, sendo o maior tempo para realização do escalonamento decorrente do uso da implementação não bloqueante de fila um fator contribuinte.

A versão base apresentou seu melhores resultados com valores de $\delta p$ maiores devido ao menor número de paradas do escalonador. A Figura 29 mostra a comparação dos resultados sob a carga D e com $\delta p$ = 10.000.000, que possibilita no máximo 5 particionamentos. Em relação aos resultados com $\delta p$ = 100.000, o método antigo apresentou grande redução do tempo de execução, atendendo as requisições em 180 segundos. Com durações de aproximadamente 175 e 180 segundos, as versões do protótipo que utilizam o novo método sofreram aumento no tempo de execução.

A Figura 28 apresenta os melhores resultados de cada implementação sob a carga D. Essa figura evidencia os ganhos obtidos com a nova versão do protótipo, que se beneficiou de valores de $\delta p$ menores e demonstrou desempenho superior quando submetido a carga D. O tempo de execução de aproximadamente 150 segundos foi cerca de 16% menor que o melhor resultado obtido pela versão base, de 180 segundos. A vazão que era de menos de 300.000 operações por segundos na versão base, chegou à ultrapassar 400.000 de operações por segundo.

Figura 28 – Melhores resultados de cada implementação em experimentos com carga D



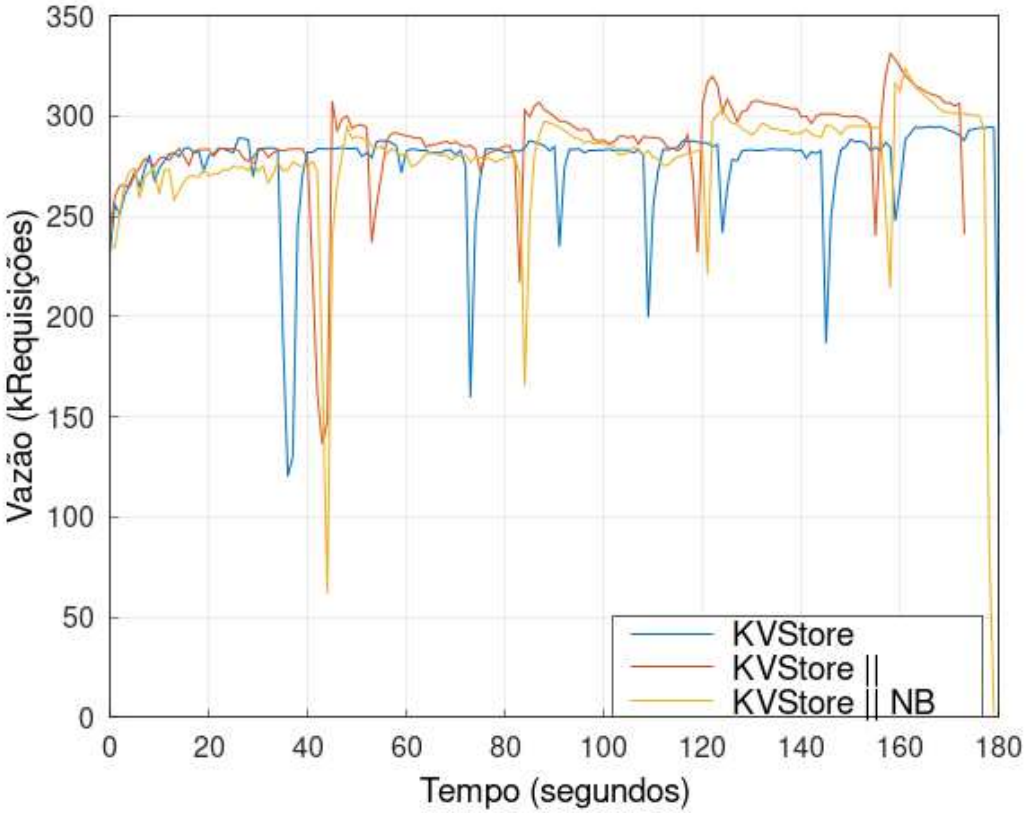

Fonte: Elaborado pelo autor (2023).

### 6.4.2 Execução da carga E

A nova versão do escalonador apresentou os melhores tempos de execução quando submetido à carga D, mas o oposto ocorreu durante a execução com a carga E, como mostra as Figuras 30 e 31. No cenário com carga E e $\delta p$ = 100.000, a implementação base rapidamente apresentou um aumento de desempenho aos 40 segundos, devido ao término do particionamento pouco após ter atendido 100.000 requisições. A nova técnica somente apresentou melhoria de desempenho ao aproximar de 10 minutos de execução, para ambas as implementações.

A maior demora para a melhoria se dá devido ao longo tempo de particionamento, combinado à alta taxa de chegada e baixa taxa de saída das filas das *worker threads*. A carga E consiste em 95% de escaneamento, o que exige um grande número de sincronizações entre as *worker threads*, e por consequência, uma baixa taxa de saída das filas. Durante o particionamento, o escalonador continua a despachar requisições usando um mapeamento de antigo. Ao fim do particionamento, há um acúmulo de requisições enviadas seguindo um esquema de particionamento desatualizado, e somente depois dessas requisições serem atendidas que o novo esquema será usado.

Figura 29 – Resultados dos experimentos com carga D e $\delta p$ = 10.000.000.

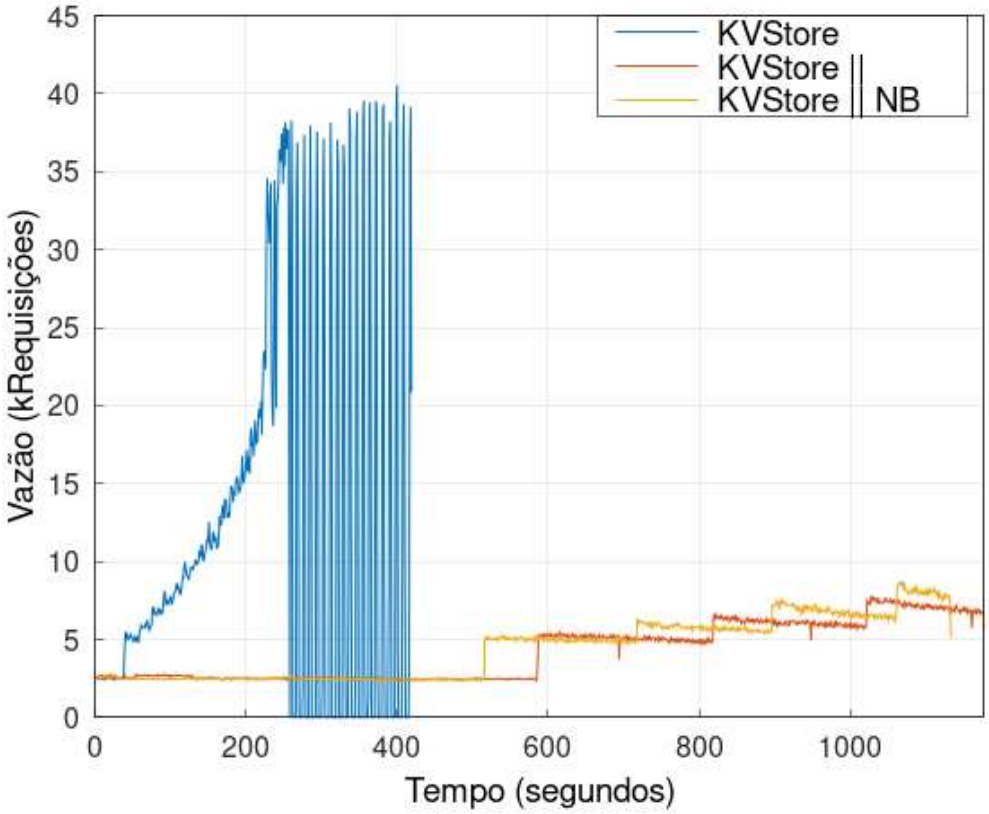



Fonte: Elaborado pelo autor (2023).

O problema não ocorre com a versão base porque o escalonador para de despachar enquanto o particionamento está sendo realizado. Diferentemente de execuções com a carga D, com a carga E o particionamento não necessariamente causou quedas de vazão, já que após $\delta p$ requisições despachadas as filas das *worker threads* podem ter acumulado requisições o suficiente para que o tempo de particionamento seja menor que o tempo de consumo dessas requisições. Na Figura 30 pode se observar que as grandes quedas de vazão só começaram a ocorrer após os 200 segundos de execução e cerca de 30 reparticionamentos, ou seja, somente após esse momento o esquema de particionamento proveu independência de execução suficiente às *worker threads* para que elas pudessem consumir toda fila de requisições durante um particionamento.

Enquanto método base oferecia a garantia de que o particionamento seria iniciado, finalizado e aplicado logo após $\delta p$ requisições despachadas, o novo protótipo não oferece nenhuma dessas garantias. Sob a carga E, com a finalização do escalonamento em cerca de 30 segundos, ambas as implementações do novo protótipo foram capazes de aplicar menos que 4 reconfigurações do esquema de partições, para qualquer $\delta p$ experimentado. A quantidade de reconfigurações aplicadas pela versão

Figura 30 – Resultados dos experimentos com carga E e $\delta p$ = 100.000.



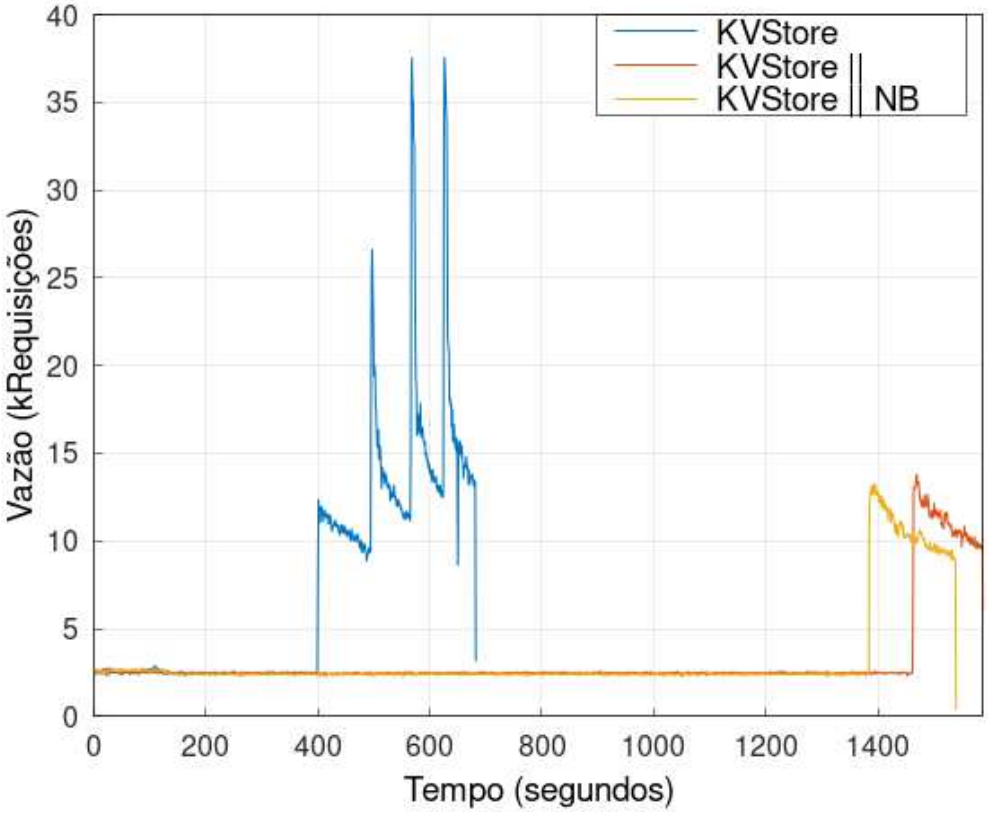

Fonte: Elaborado pelo autor (2023).

base foi igual ao número de requisições dividido por $\delta p$, já que o escalonador só volta a executar quando o novo esquema de partições estiver pronto.

Sob a carga E, ao contrário do que aconteceu nos experimentos com a carga D, a versão base do protótipo apresentou tempos de execução maiores com o aumento de *deltap*, como visto na Figura 31. O tempo de execução de aproximadamente 7 minutos com $\delta p$ = 100.000 foi consideravelmente menor que o tempo de mais de 11 minutos com $\delta p$ = 1.000.000.

Dentre as implementações do novo escalonador, a que utiliza estruturas de dados não bloqueantes apresentou um tempo de execução menor que o da implementação que utiliza estruturas de dados bloqueantes. A utilização das estruturas de dados não bloqueantes provocou uma redução de aproximadamente 5% no tempo de execução, como mostram as Figuras 30 e 31. Entretanto, o ganho não se deu por uma maior velocidade proveniente do uso das implementações não bloqueantes. Pelo contrário, o menor desempenho que a utilização das filas não bloqueantes causaram ao escalonador, resultou em um menor número de requisições seguindo uma configuração de particionamento não otimizada sendo inserido na fila das *worker threads* durante o particionamento, resultando em um aumento de desempenho mais rapidamente.

Figura 31 – Resultados dos experimentos com carga E e $\delta p$ = 1.000.000.




Fonte: Elaborado pelo autor (2023).

# 7 CONCLUSÃO

Para esse trabalho, foram reunidas da literatura propostas de implementações não bloqueantes de diferentes estruturas de dados e foi apresentado o funcionamento de algumas das propostas encontradas. Foram identificadas desde filas com operações simples até grafos com diferentes operações de consulta não bloqueantes. Em vista desses trabalhos, foi possível verificar a existência de esforço para produzir algoritmos com as fortes garantias de progresso *lock-free* e *wait-free* e que proporcionassem alto desempenho.

Conduziram-se experimentos comparativos de dois tipos estruturas de dados. Foram testadas filas e tabelas de dispersão não bloqueantes disponibilizadas por bibliotecas de programação de código aberto na linguagem C++. Os experimentos consistiram de *microbenchmarks* locais comparando implementações bloqueantes e não bloqueantes sofrendo acessos e modificações por múltiplas *threads*. Nos experimentos com filas, foram variados os tamanhos dos elementos armazenados, enquanto nos experimentos com tabelas de dispersão foram realizados testes com diferentes cargas de trabalho, tamanhos de valores armazenados, carga inicial das estruturas e número de chaves distintas. Os resultados demonstraram a disponibilidade de implementações de estruturas de dados não bloqueantes em bibliotecas de programação que puderam apresentar desempenho superior em relação a implementações bloqueantes de bibliotecas de uso comercial, como Boost e TBB, em diferentes cenários de teste. Além disso, a experimentação com filas permitiu a observação de resultados inesperadamente bons da implementação que utiliza um *mutex* para sincronizar a implementação de fila da biblioteca padrão.

Foi ainda conduzido um estudo de caso considerando uma aplicação de armazenamento de chaves e valores que utiliza uma técnica de particionamento para paralelizar o atendimento de requisições. O estudo objetivou reduzir a contenção no escalonamento e verificar o impacto do uso de estruturas de dados não bloqueantes no desempenho geral do sistema. Foram apresentados novos algoritmos para a implementação do sistema, com a criação de duas versões adicionais. Uma delas utiliza as mesmas implementações de estruturas de dados da versão base, enquanto a outra substitui algumas implementações por versões não bloqueantes. Por meio de uma avaliação experimental foi possível verificar que, em determinados casos de teste, os novos algoritmos proporcionaram o aumento de desempenho em relação à técnica implementada anteriormente pelo protótipo. Entretanto, o uso das estruturas de dados não bloqueantes não ofereceram melhorias de desempenho em relação ao uso de bloqueios.

## 7.1 TRABALHOS FUTUROS

Como trabalhos futuros, serão estudadas outras implementações de estruturas de dados não bloqueantes. As aplicação estudada utiliza as filas em um cenário específico de uma *thread* produtora e *outra* consumidora, sendo possível desenvolver uma implementação que se limite à essa restrição a fim de oferecer maior desempenho.

Os experimentos sugeriram que sob a carga E a versão original da aplicação oferece maior desempenho. Entretanto, os resultados são limitados a cargas de 5.000.000 a 50.000.000 de requisições, e com mais tempo seria possível verificar o comportamento do sistema em mais cenários e com maior número de requisições a serem atendidas.

Em vista da separação do particionamento e escalonamento em duas linhas de processamento diferentes, é estudada a possibilidade de realizar o particionamento continuamente, desconsiderando $\delta p$ na decisão de início do particionamento e exigindo que o particionador sempre esteja processando uma nova configuração do esquema de partições.

São ainda estudados meios de evitar a redução de desempenho sofrida pela nova implementação do protótipo quando submetido à carga E, como por exemplo, limitar a velocidade de escalonamento, adaptando a taxa de chegada das filas das *worker threads* à suas taxas de saída.

Ainda sobre a aplicação estudada, para a realização dos particionamentos, o programa utiliza bibliotecas de programação que fornecem funções as quais as entradas não são compatíveis com a forma como o grafo é construído pelo *pattern tracker*, o que exige efetivamente uma cópia dos grafo de carga de trabalho. É então estudada a possibilidade de modificar a forma como o *pattern tracker* armazena o grafo ou modificar os programas de particionamento com o objetivo de eliminar a necessidade de cópia.

## REFERÊNCIAS

CANON, Louis-Claude; JEANNOT, Emmanuel; SAKELLARIOU, Rizos; ZHENG, Wei. Comparative Evaluation Of The Robustness Of DAG Scheduling Heuristics. *In*: **Grid Computing: Achievements and Prospects**. Edição: Sergei Gorlatch, Paraskevi Fragopoulou e Thierry Priol. Boston, MA: Springer US, 2008. P. 73–84. ISBN 978-0-387-09457-1.

CHATTERJEE, Bapi; PERI, Sathya; SA, Muktikanta; SINGHAL, Nandini. A Simple and Practical Concurrent Non-Blocking Unbounded Graph with Linearizable Reachability Queries. *In*: PROCEEDINGS of the 20th International Conference on Distributed Computing and Networking. Bangalore, India: Association for Computing Machinery, 2019. (ICDCN '19), p. 168–177.

COOPER, Brian F.; SILBERSTEIN, Adam; TAM, Erwin; RAMAKRISHNAN, Raghu; SEARS, Russell. Benchmarking Cloud Serving Systems with YCSB. *In*: PROCEEDINGS of the 1st ACM Symposium on Cloud Computing. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010. (SoCC '10), p. 143–154.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. **Introduction to Algorithms, 3rd Edition**. [*S.l.*]: MIT Press, 2009. ISBN 978-0-262-03384-8.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Sistemas Distribuıdos - 5ed: Conceitos e Projeto**. [*S.l.*]: Bookman Editora, 2013. ISBN 9788582600542.

DIJKSTRA, E. W. Solution of a Problem in Concurrent Programming Control. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 8, n. 9, p. 569, set. 1965. ISSN 0001-0782.

DUARTE, Rodrigo Medeiros; DU BOIS, André Rauber; PILLA, Maurício Lima; CAVALHEIRO, Gerson Geraldo H.; REISER, Renata Hax Sander. Comparing the performance of concurrent hash tables implemented in Haskell. **Science of Computer Programming**, v. 173, p. 56–70, 2019. Brazilian Symposium on Programming Languages (SBLP '15+16). ISSN 0167-6423.

FELDMAN, Steven; LABORDE, Pierre; DECHEV, Damian. Concurrent multi-level arrays: Wait-free extensible hash maps. *In*: 2013 International Conference on

Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). [*S.l.*: *s.n.*], 2013. P. 155–163.

FELDMAN, Steven D.; BHAT, Akshatha; LABORDE, Pierre; YI, Qing; DECHEV, Damain. Effective Use of Non-Blocking Data Structures in a Deduplication Application. *In*: PROCEEDINGS of the 2013 Companion Publication for Conference on Systems, Programming, And Applications: Software for Humanity. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013. (SPLASH '13), p. 133–142.

FITZPATRICK, Brad. Distributed Caching with Memcached. **Linux J.**, Belltown Media, Houston, TX, v. 2004, n. 124, p. 5, ago. 2004. ISSN 1075-3583.

FOMITCHEV, Mikhail; RUPPERT, Eric. Lock-Free Linked Lists and Skip Lists. *In*: PROCEEDINGS of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing. St. John's, Newfoundland, Canada: Association for Computing Machinery, 2004. (PODC '04), p. 50–59.

GABRIEL TROMBETA, João; MACHADO MENDIZABAL, Odorico. **Proposta para Reparticionamento de Estado em Replicação Máquina de Estado Paralela**. [*S.l.*]: Universidade do Vale do Itajaí, 2020. Disponível em: `http://dx.doi.org/10.14210/cotb.v11n1.p071-073`.

HARRIS, Timothy L. A Pragmatic Implementation of Non-Blocking Linked-Lists. *In*: PROCEEDINGS of the 15th International Conference on Distributed Computing. Berlin, Heidelberg: Springer-Verlag, 2001. (DISC '01), p. 300–314.

HENNESSY, John L; PATTERSON, David A. **Computer Architecture**. 4. ed. Oxford, England: Morgan Kaufmann, nov. 2006. (The Morgan Kaufmann Series in Computer Architecture and Design).

HERLIHY, Maurice. Wait-Free Synchronization. **ACM Trans. Program. Lang. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 13, n. 1, p. 124–149, jan. 1991. ISSN 0164-0925.

HERLIHY, Maurice; SHAVIT, Nir. On the Nature of Progress. *In*: PROCEEDINGS of the 15th International Conference on Principles of Distributed Systems. Toulouse, France: Springer-Verlag, 2011. (OPODIS'11), p. 313–328.

HERLIHY, Maurice; SHAVIT, Nir. **The Art of Multiprocessor Programming, Revised Reprint**. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN 9780123973375.

HERLIHY, Maurice P.; WING, Jeannette M. Linearizability: A Correctness Condition for Concurrent Objects. **ACM Trans. Program. Lang. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 12, n. 3, p. 463–492, jul. 1990. ISSN 0164-0925.

HUNT, Nicholas; SANDHU, Paramjit Singh; CEZE, Luis. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures. *In*: 2011 15th Workshop on Interaction between Compilers and Computer Architectures. [*S.l.*: *s.n.*], 2011. P. 63–70.

KARYPIS, George; KUMAR, Vipin. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. **SIAM Journal on Scientific Computing**, v. 20, n. 1, p. 359–392, 1998. eprint: `https://doi.org/10.1137/S1064827595287997`.

KOGAN, Alex; PETRANK, Erez. Wait-Free Queues with Multiple Enqueuers and Dequeuers. *In*: PROCEEDINGS of the 16th ACM Symposium on Principles and Practice of Parallel Programming. San Antonio, TX, USA: Association for Computing Machinery, 2011. (PPoPP '11), p. 223–234.

LABORDE, Pierre; FELDMAN, Steven; DECHEV, Damian. A Wait-Free Hash Map. **Int. J. Parallel Program.**, Kluwer Academic Publishers, USA, v. 45, n. 3, p. 421–448, jun. 2017. ISSN 0885-7458.

LADAN-MOZES, Edya; SHAVIT, Nir. An Optimistic Approach to Lock-Free FIFO Queues. *In*: GUERRAOUI, Rachid (Ed.). **Distributed Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 117–131.

LUIZ, Douglas; MENDIZABAL, Odorico. Análise de Desempenho de Hash Tables Não-Bloqueantes na Linguagem C++. *In*: ANAIS da XXIII Escola Regional de Alto Desempenho da Região Sul. Porto Alegre/RS: SBC, 2023. P. 73–76.

MANDAGERE, Nagapramod; ZHOU, Pin; SMITH, Mark A; UTTAMCHANDANI, Sandeep. Demystifying Data Deduplication. *In*: PROCEEDINGS of the ACM/IFIP/USENIX Middleware '08 Conference Companion. Leuven, Belgium: Association for Computing Machinery, 2008. (Companion '08), p. 12–17.

MICHAEL, Maged M. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. *In*: PROCEEDINGS of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures. Winnipeg, Manitoba, Canada: Association for Computing Machinery, 2002. (SPAA '02), p. 73–82.

MICHAEL, Maged M. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes. *In*: PROCEEDINGS of the Twenty-First Annual Symposium on Principles of Distributed Computing. Monterey, California: Association for Computing Machinery, 2002. (PODC '02), p. 21–30.

MICHAEL, Maged M.; SCOTT, Michael L. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. *In*: PROCEEDINGS of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996. (PODC '96), p. 267–275.

NIKOLAEV, Ruslan. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. en. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2019.

PACHECO, P.; MALENSEK, M. **An Introduction to Parallel Programming**. 2nd. [*S.l.*]: Morgan Kaufmann, 2021. ISBN 9780128046180.

PATTERSON, David A.; HENNESSY, John L. **Computer Organization and Design RISC-V Edition: The Hardware Software Interface**. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2020. ISBN 9780128203316.

PERI, Sathya; REDDY, Chandra Kiran; SA, Muktikanta. An Efficient Practical Concurrent Wait-Free Unbounded Graph. *In*: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). [*S.l.*: *s.n.*], 2019. P. 2487–2494.

SHALEV, Ori; SHAVIT, Nir. Split-Ordered Lists: Lock-Free Extensible Hash Tables. **J. ACM**, Association for Computing Machinery, New York, NY, USA, v. 53, n. 3, p. 379–405, mai. 2006. ISSN 0004-5411.

SUNDELL, Håkan; TSIGAS, Philippas. Fast and lock-free concurrent priority queues for multi-thread systems. **Journal of Parallel and Distributed Computing**, v. 65, n. 5, p. 609–627, 2005. ISSN 0743-7315.

TANENBAUM, Andrew S; BOS, Herbert. **Modern Operating Systems**. 4. ed. Upper Saddle River, NJ: Pearson, mar. 2014.

TIMNAT, Shahar; BRAGINSKY, Anastasia; KOGAN, Alex; PETRANK, Erez. Wait-Free Linked-Lists. Association for Computing Machinery, New Orleans, Louisiana, USA, p. 309–310, 2012.

TROMBETA, João; MENDIZABAL, Odorico. Simulador para medição de paralelismo em algoritmos de escalonamento para Replicação Máquina de Estados Paralela. *In*: ANAIS da XX Escola Regional de Alto Desempenho da Região Sul. Santa Maria: SBC, 2020. P. 49–52.

TROMBETA, João Gabriel. **Análise do uso de particionamento balanceado de grafos para explorar paralelismo em Replicação Máquina de Estados Paralela**. 2021. F. 83. Monografia (Graduação) – Ciências da Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, Brasil.

WING, Jeannette M.; GONG, Chun. Testing and Verifying Concurrent Objects. **J. Parallel Distrib. Comput.**, Academic Press, Inc., USA, v. 17, 1–2, p. 164–182, jan. 1993. ISSN 0743-7315.

XAVIER, Luiz Gustavo Coutinho; MENDIZABAL, Odorico M. Análise de Desempenho de Estruturas de Dados Concorrentes Implementadas na Linguagem Java. *In*: ANAIS da XIX Escola Regional de Alto Desempenho da Região Sul. Três de Maio: SBC, 2019.

XIA, Wen; JIANG, Hong; FENG, Dan; DOUGLIS, Fred; SHILANE, Philip; HUA, Yu; FU, Min; ZHANG, Yucheng; ZHOU, Yukun. A Comprehensive Study of the Past, Present, and Future of Data Deduplication. **Proceedings of the IEEE**, v. 104, n. 9, p. 1681–1710, 2016.

# APÊNDICE A – CÓDIGO FONTE DO PROGRAMA DE AVALIAÇÃO DE DESEMPENHO DE ESTRUTURAS DE DADOS NÃO BLOQUEANTES

O código dos programas desenvolvidos para avaliar o desempenho das filas e tabelas de dispersão está disponível no GitHub, que pode ser acessado pelo endereço `https://github.com/douglaspereira04/non-blocking-cpp`, no *branch main* e *commit* 607380b3d5c0aff430b5581026da6d08f8a879aa.

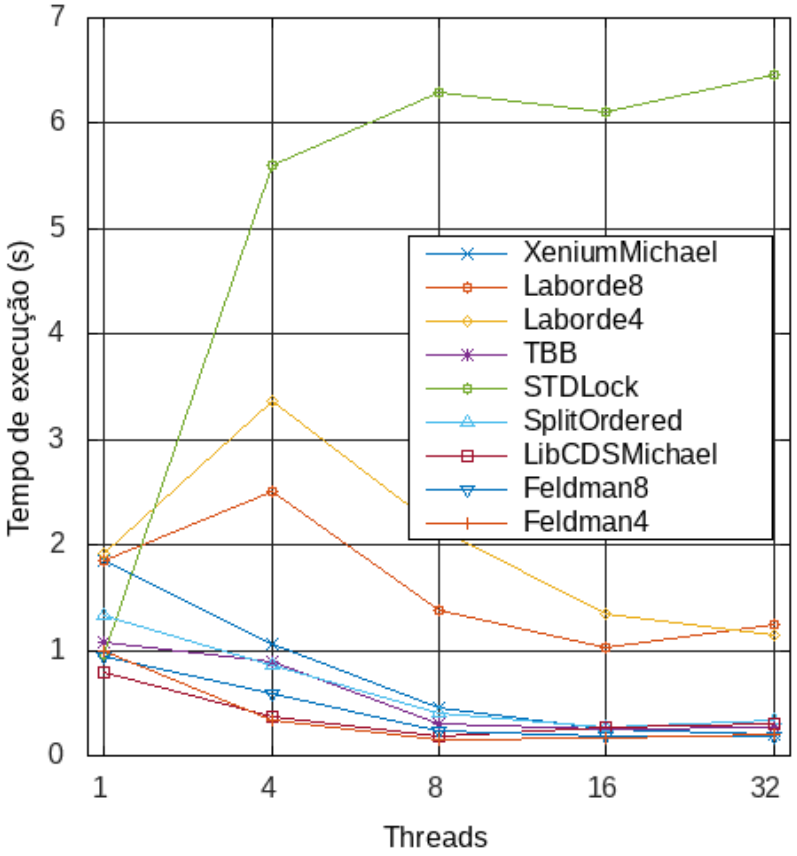
# APÊNDICE B – CÓDIGO FONTE DOS PROTÓTIPOS DE *KEY-VALUE STORE*

O código dos protótipos usados no estudo de caso está disponível em `https://github.com/douglaspereira04/kvpaxos`. As versões que utilizam as implementações de estruturas de dados bloqueantes estão no *branch simplified-replica*, no *commit* 78804e604cf6bbd85cf1d2fde6681cc72f4b00b6, enquanto a versão que utiliza as avriantes não bloqueantes das estruturas de dados está na *branch lock-free-q*, no *commit* a75686e3496b545b826b4c5ce9b22d8e3703b21b.

# APÊNDICE C – GRÁFICOS PRODUZIDOS PARA A AVALIAÇÃO DE DESEMPENHO DE TABELAS DE DISPERSÃO

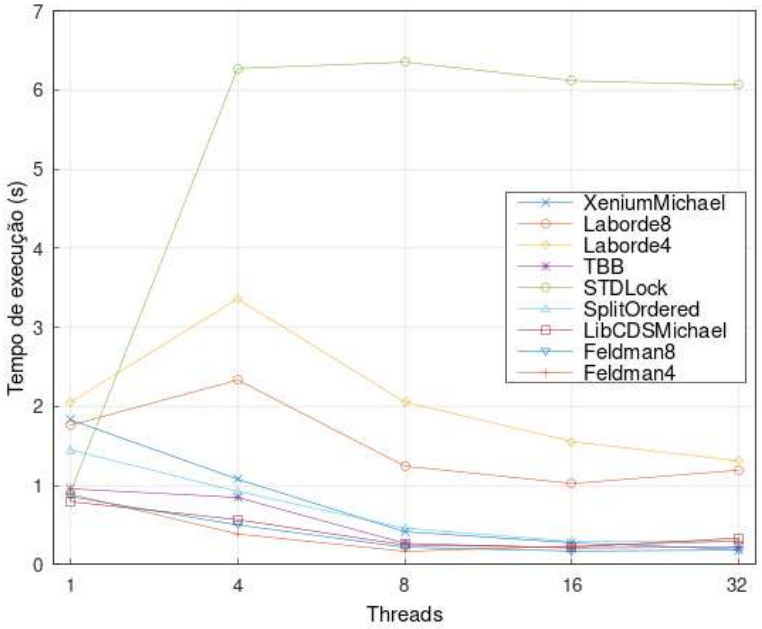
Os gráficos produzidos para auxiliar na avaliação de desempenho das tabelas de dispersão são apresentados nesse apêndice.

Figura 32 – *Resultados dos testes com tabelas inicialmente vazias, poucas chaves distintas, valores pequenos e distribuição A.*



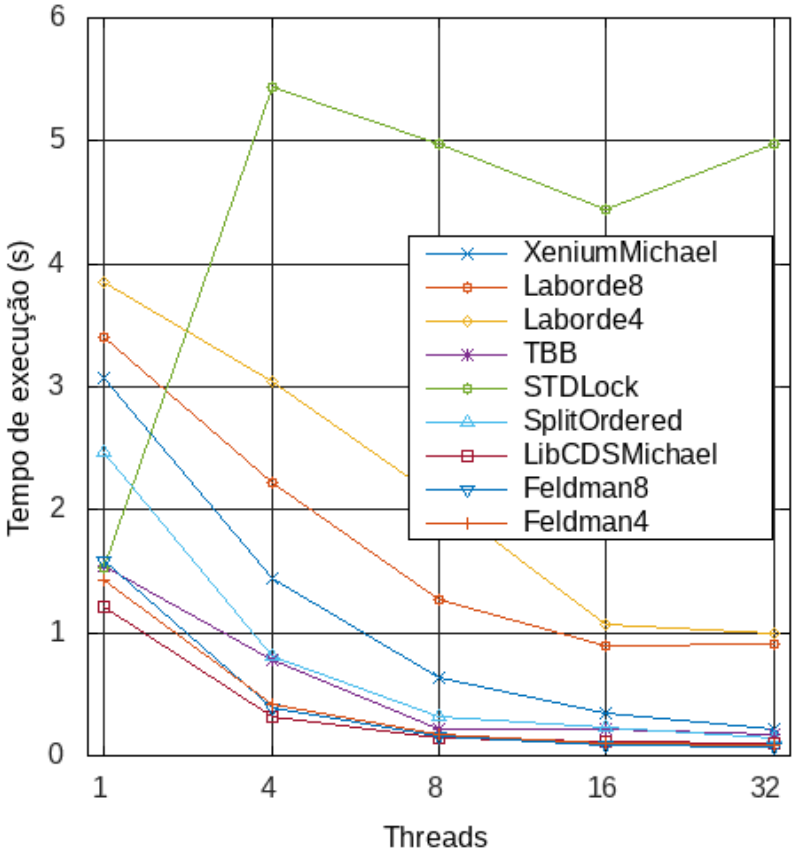

Fonte: O autor (2023)

Figura 33 – *Resultados dos testes com chaves previamente inseridas, poucas chaves distintas, valores pequenos e distribuição A.*



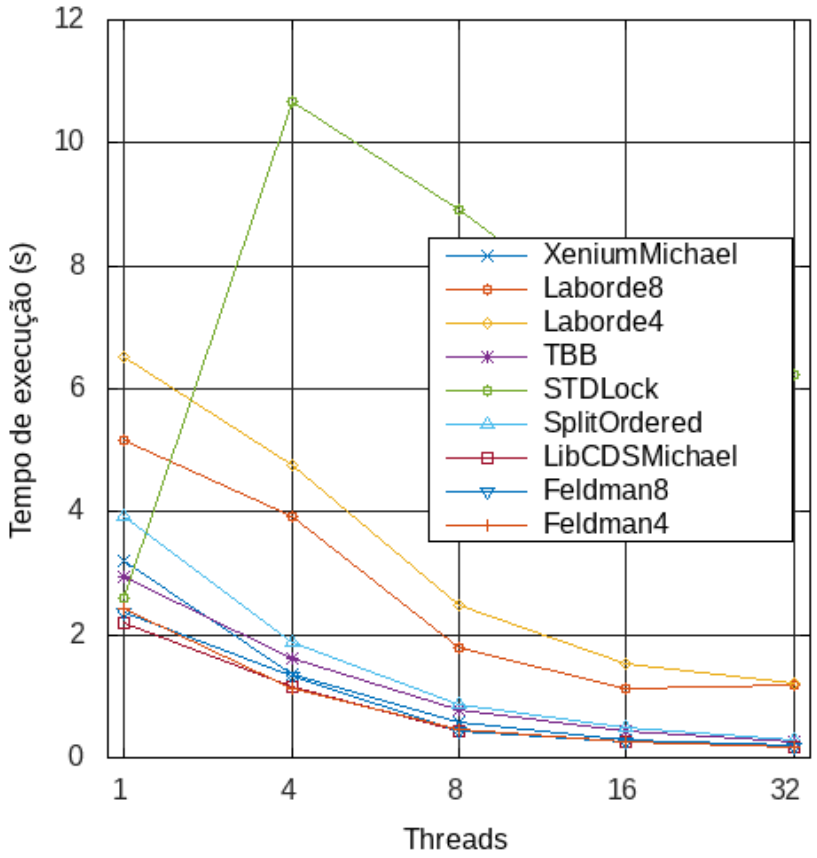

Fonte: O autor (2023)

Figura 34 – *Resultados dos testes com tabelas inicialmente vazias, muitas chaves distintas, valores pequenos e distribuição A.*



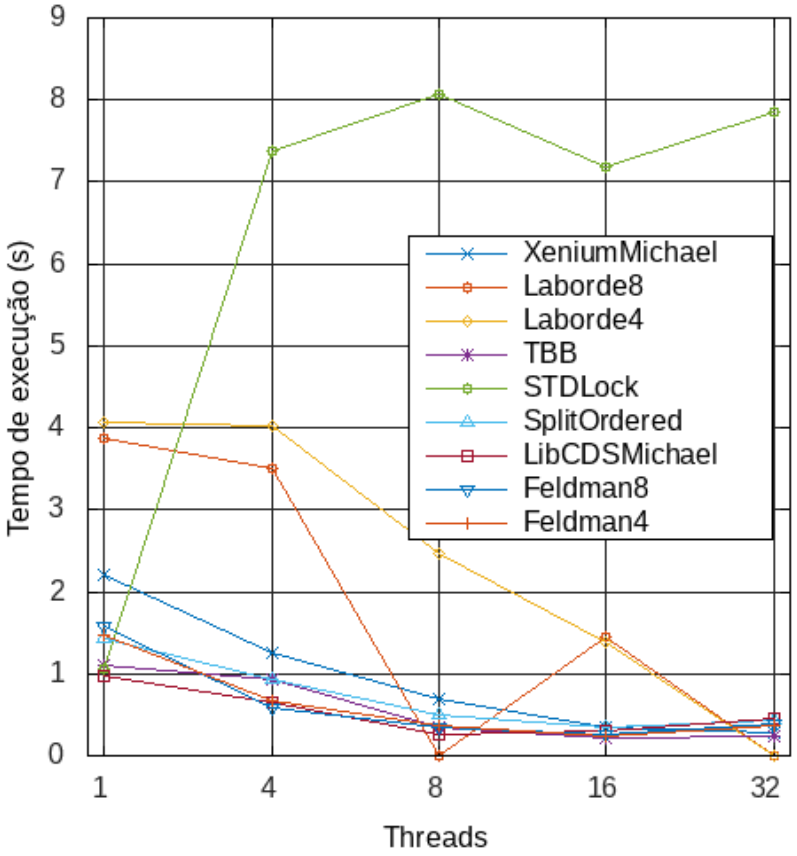

Fonte: O autor (2023)

Figura 35 – *Resultados dos testes com chaves previamente inseridas, muitas chaves distintas, valores pequenos e distribuição A.*



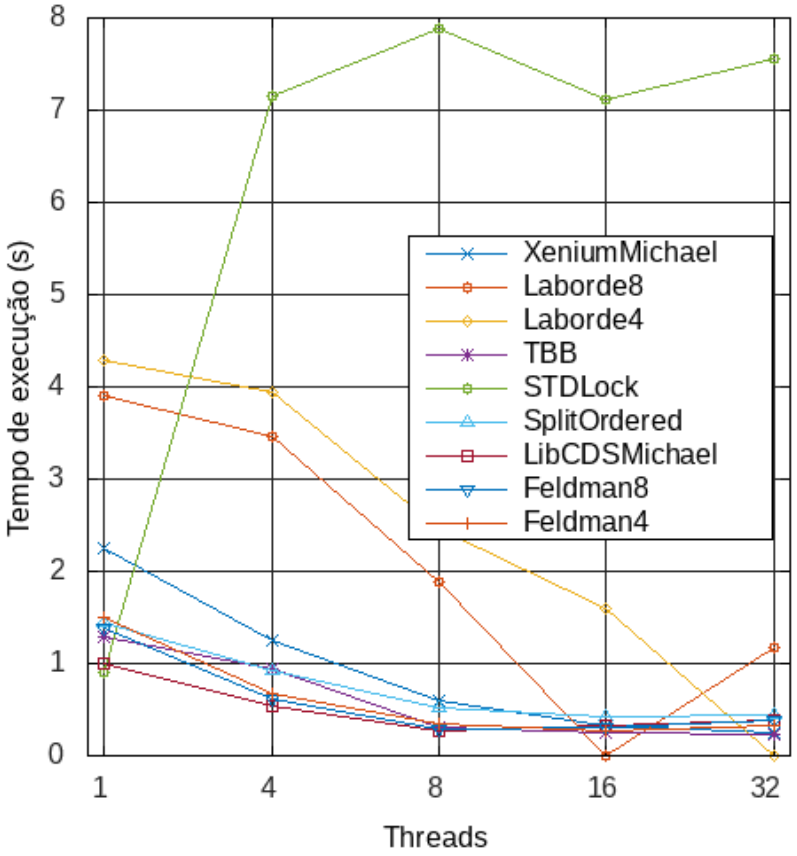

Fonte: O autor (2023)

Figura 36 – *Resultados dos testes com tabelas inicialmente vazias, poucas chaves distintas, valores grandes e distribuição A.*



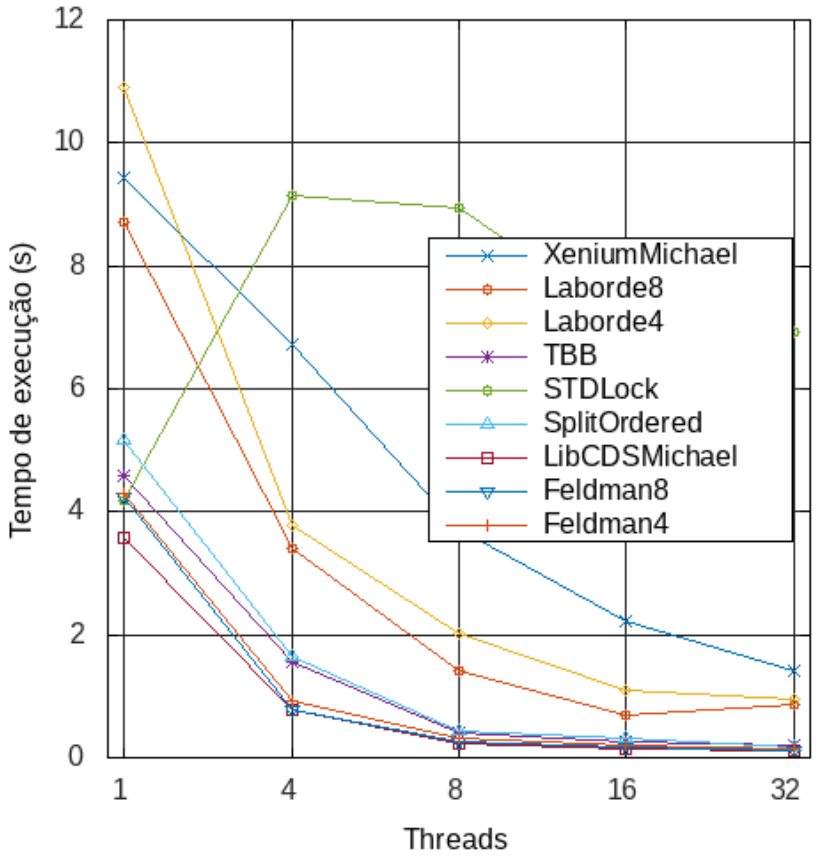

Fonte: O autor (2023)

Figura 37 – *Resultados dos testes com chaves previamente inseridas, poucas chaves distintas, valores grandes e distribuição A.*



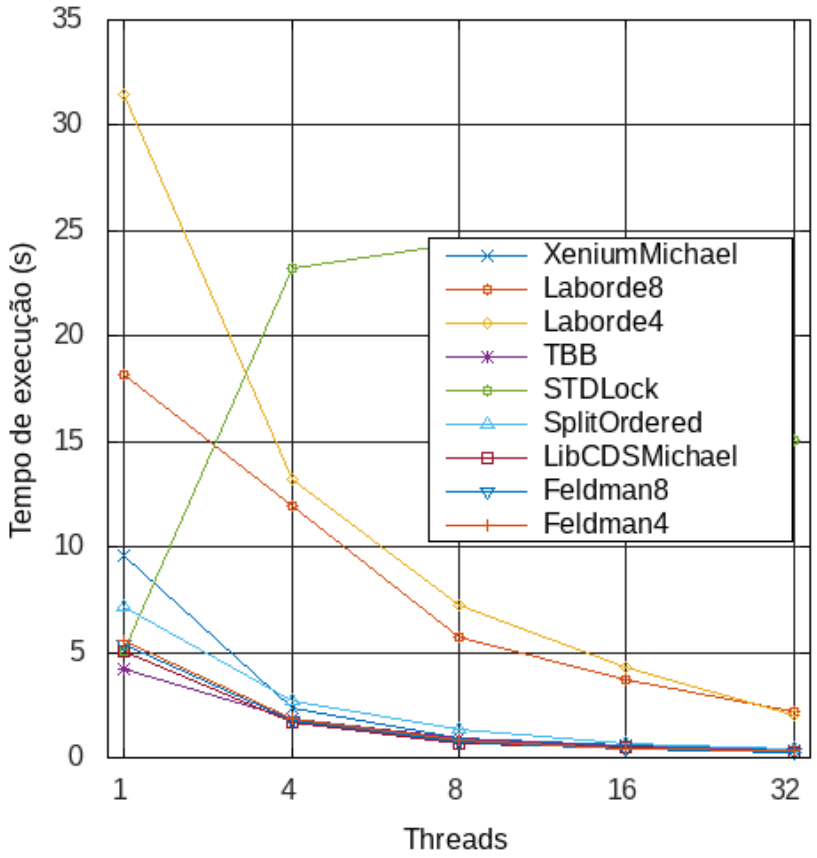

Fonte: O autor (2023)

Figura 38 – *Resultados dos testes com tabelas inicialmente vazias, muitas chaves distintas, valores grandes e distribuição A.*



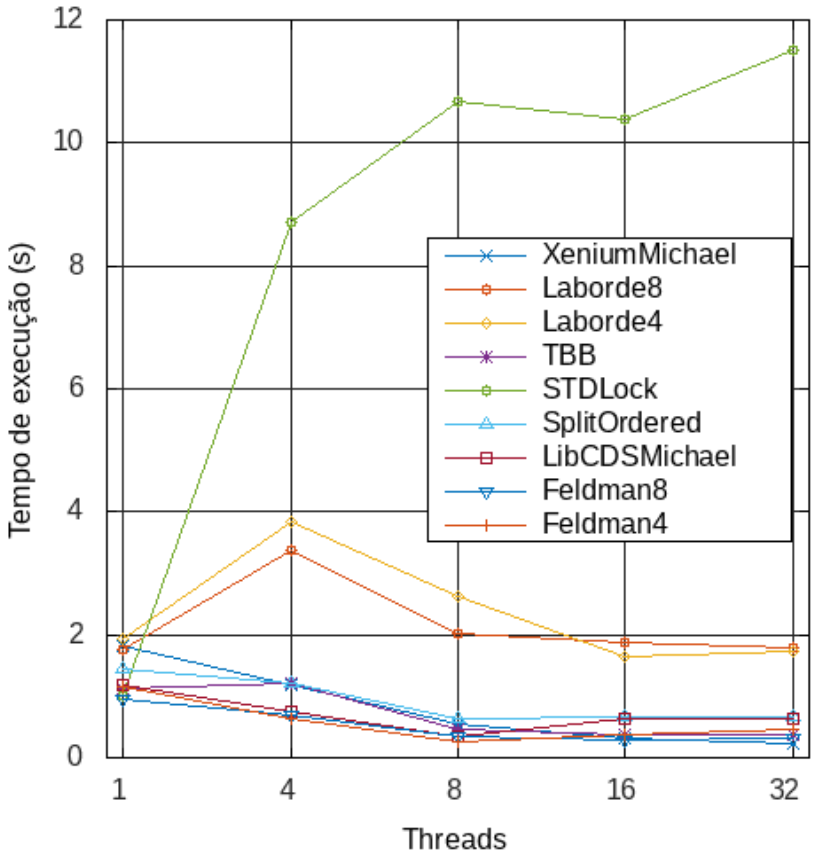

Fonte: O autor (2023)

Figura 39 – *Resultados dos testes com chaves previamente inseridas, muitas chaves distintas, valores grandes e distribuição A.*



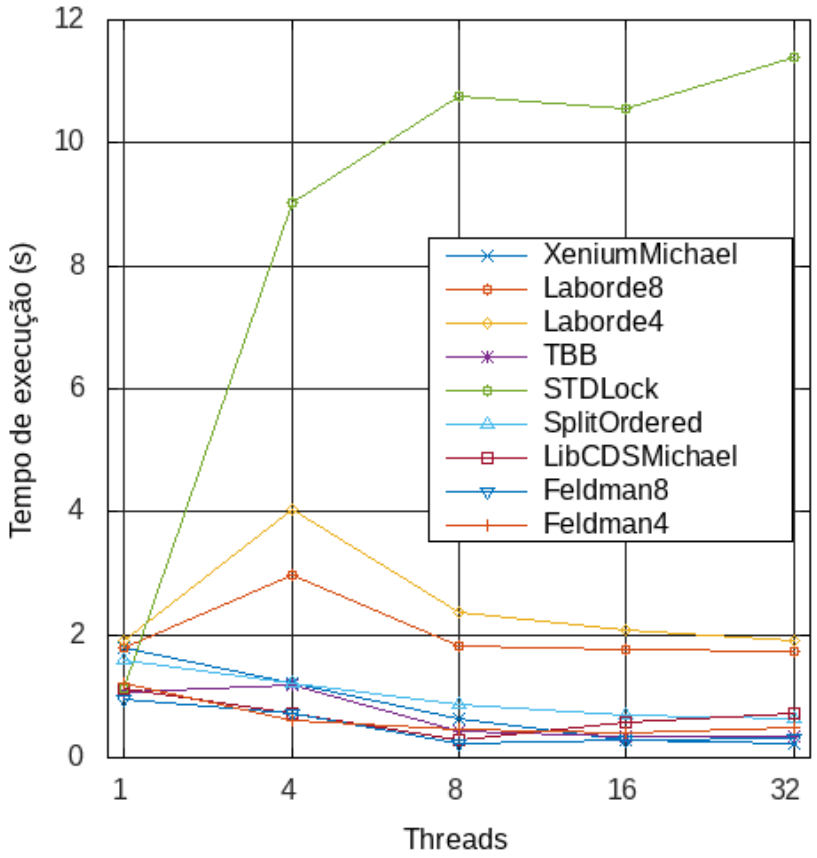

Fonte: O autor (2023)

Figura 40 – *Resultados dos testes com tabelas inicialmente vazias, poucas chaves distintas, valores pequenos e distribuição B*



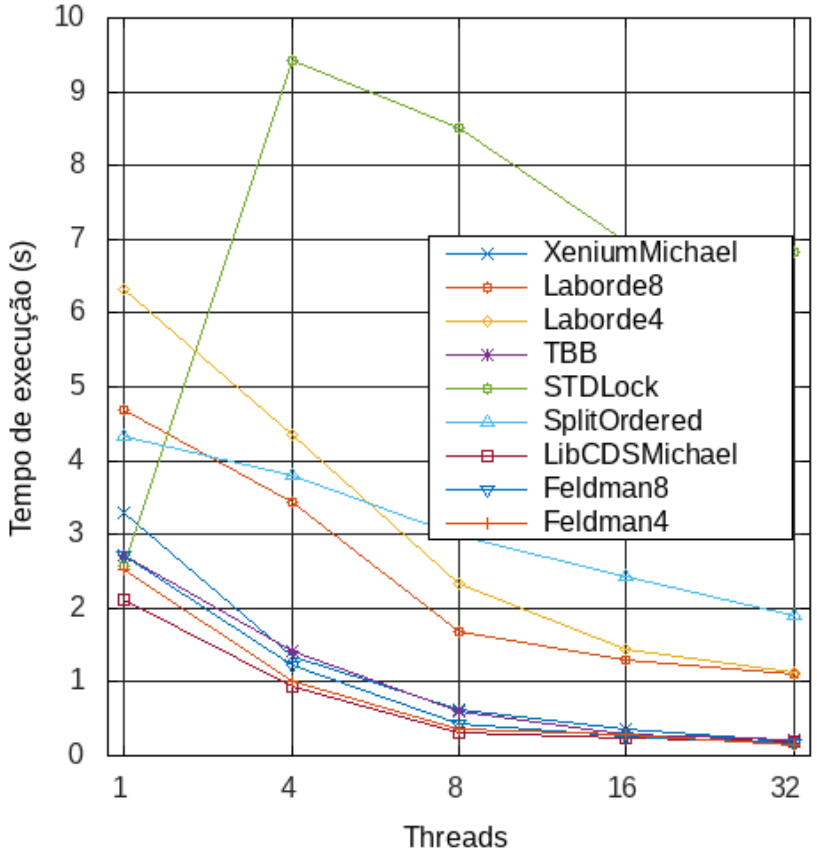

Fonte: O autor (2023)

Figura 41 – *Resultados dos testes com chaves previamente inseridas, poucas chaves distintas, valores pequenos e distribuição B*



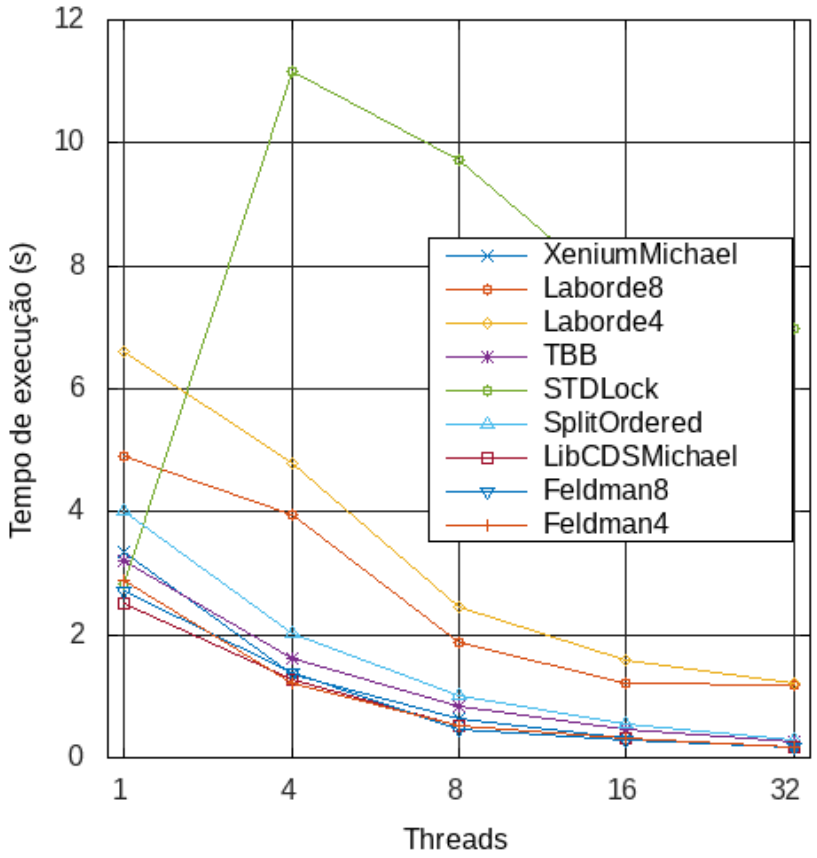

Fonte: O autor (2023)

Figura 42 – *Resultados dos testes com tabelas inicialmente vazias, muitas chaves distintas, valores pequenos e distribuição B*



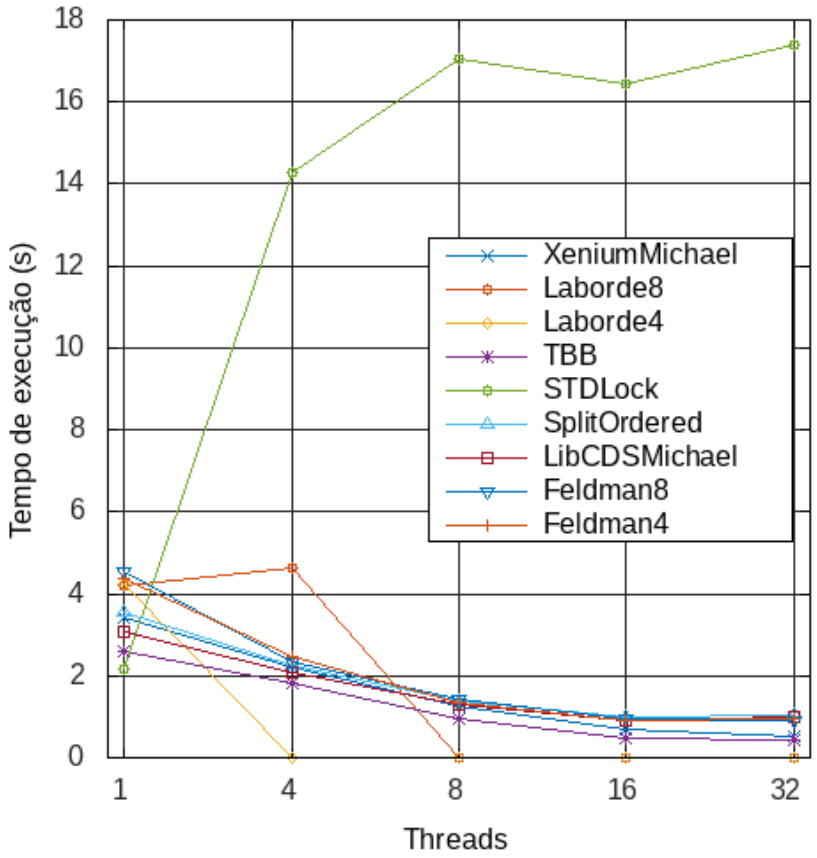

Fonte: O autor (2023)

Figura 43 – *Resultados dos testes com chaves previamente inseridas, muitas chaves distintas, valores pequenos e distribuição B*



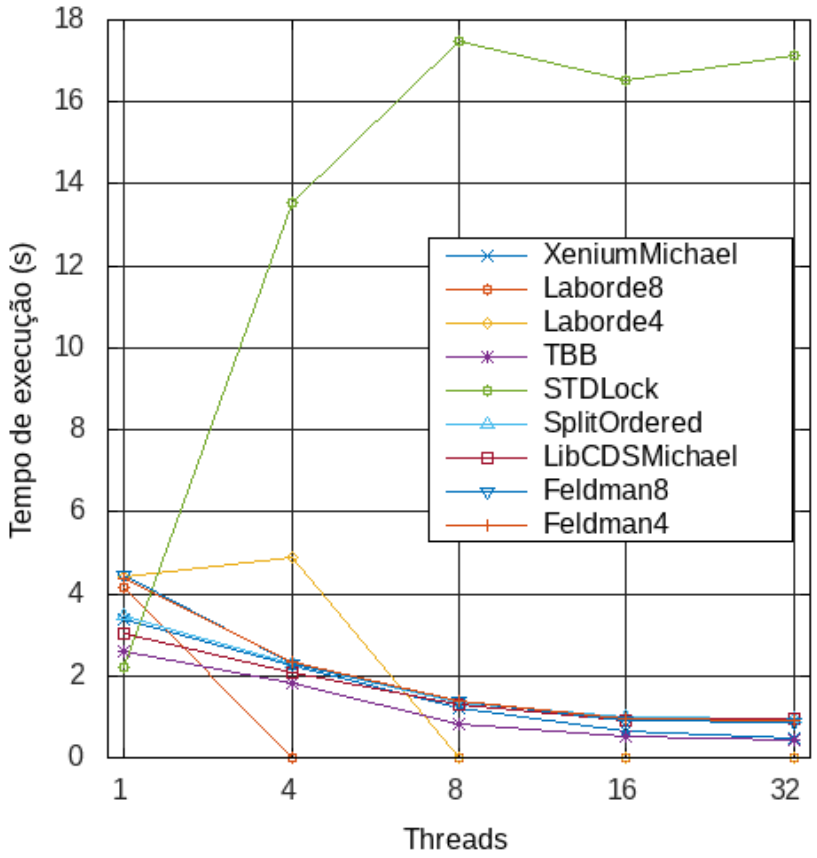

Fonte: O autor (2023)

Figura 44 – *Resultados dos testes com tabelas inicialmente vazias, poucas chaves distintas, valores grandes e distribuição B*



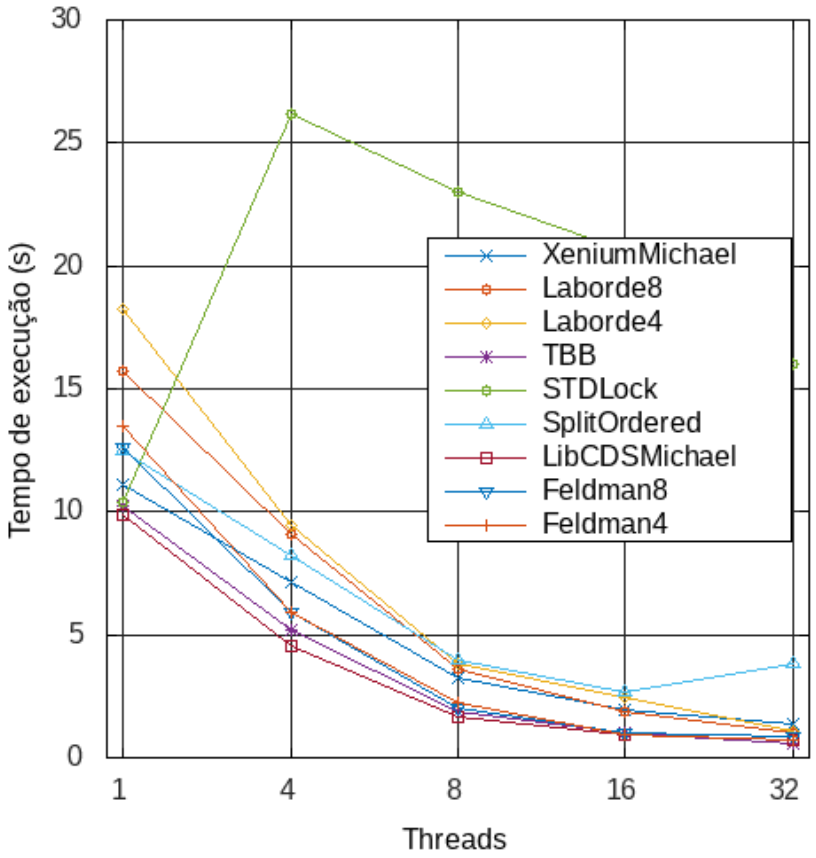

Fonte: O autor (2023)

Figura 45 – *Resultados dos testes com chaves previamente inseridas, poucas chaves distintas, valores grandes e distribuição B*



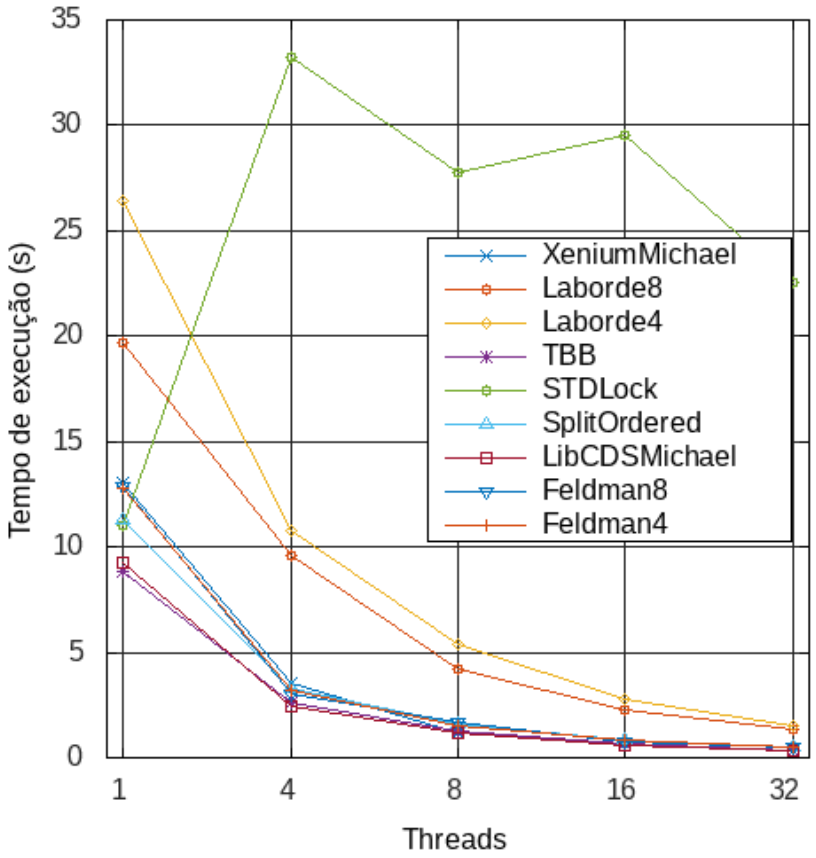

Fonte: O autor (2023)

Figura 46 – *Resultados dos testes com tabelas inicialmente vazias, muitas chaves distintas, valores grandes e distribuição B*



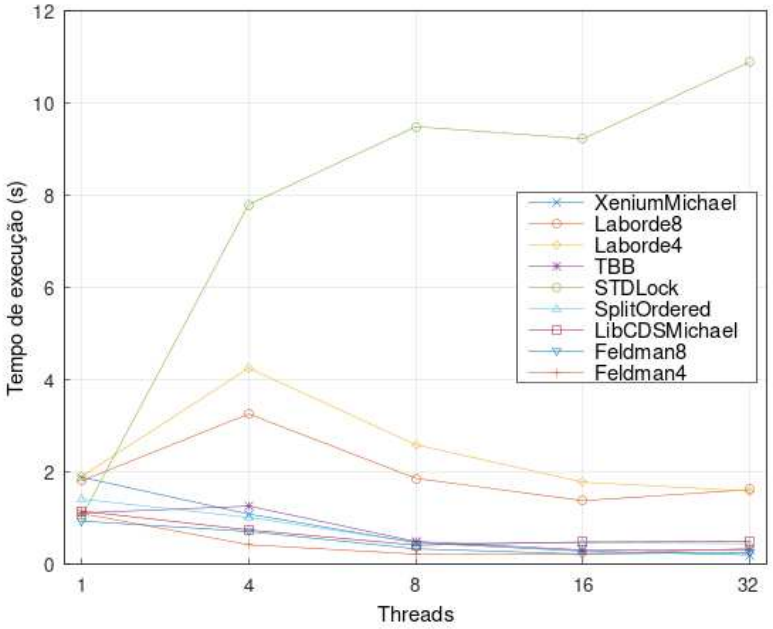

Fonte: O autor (2023)

Figura 47 – *Resultados dos testes com chaves previamente inseridas, muitas chaves distintas, valores grandes e distribuição B*



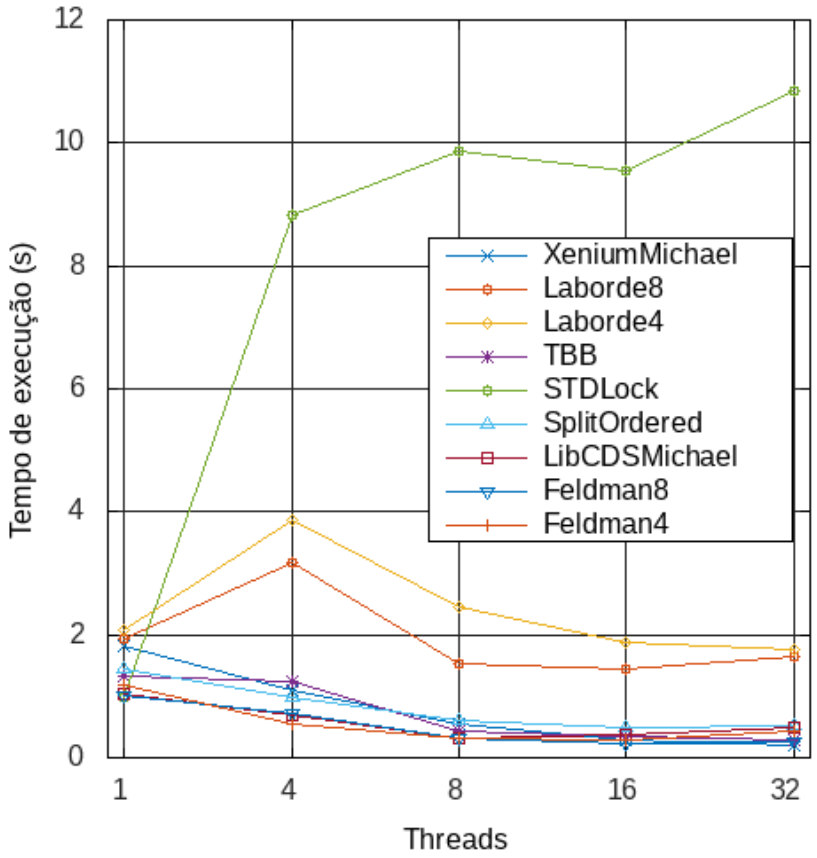

Fonte: O autor (2023)

Figura 48 – *Resultados dos testes com tabelas inicialmente vazias, poucas chaves distintas, valores pequenos e distribuição C.*



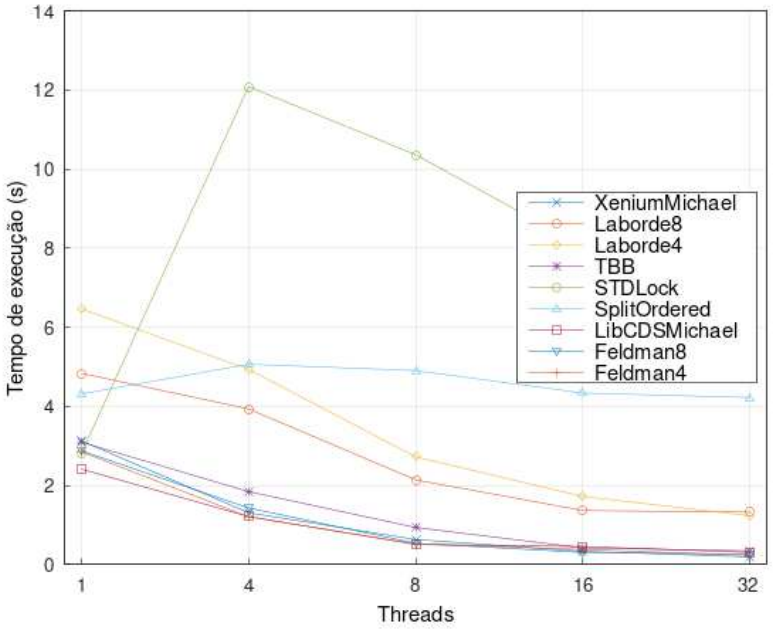

Fonte: O autor (2023)

Figura 49 – *Resultados dos testes com chaves previamente inseridas, poucas chaves distintas, valores pequenos e distribuição C.*



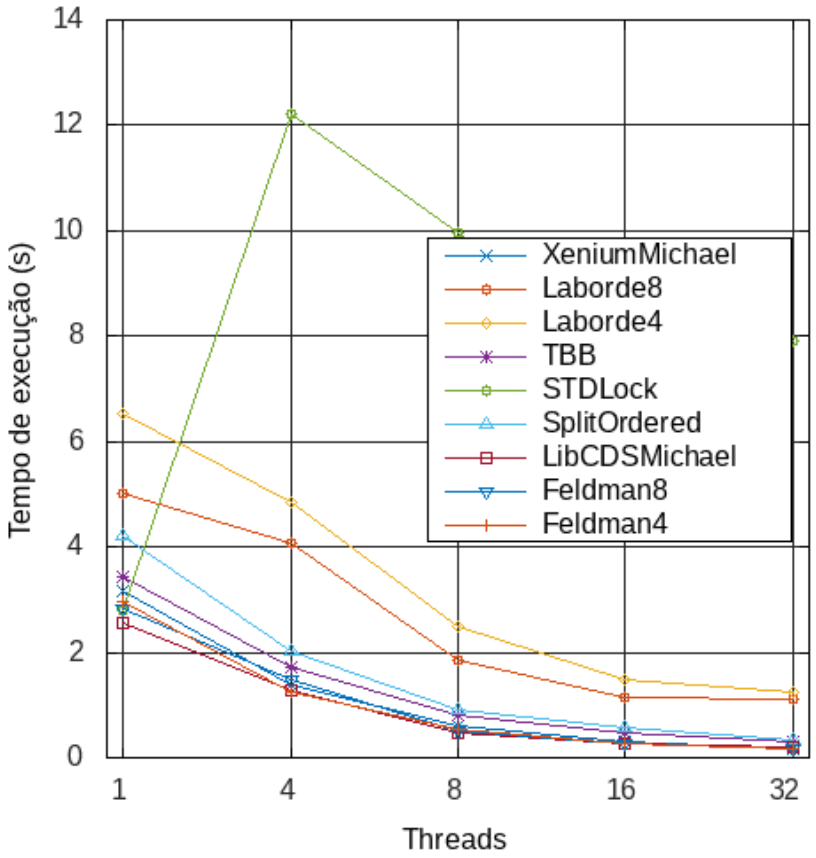

Fonte: O autor (2023)

Figura 50 – *Resultados dos testes com tabelas inicialmente vazias, muitas chaves distintas, valores pequenos e distribuição C.*



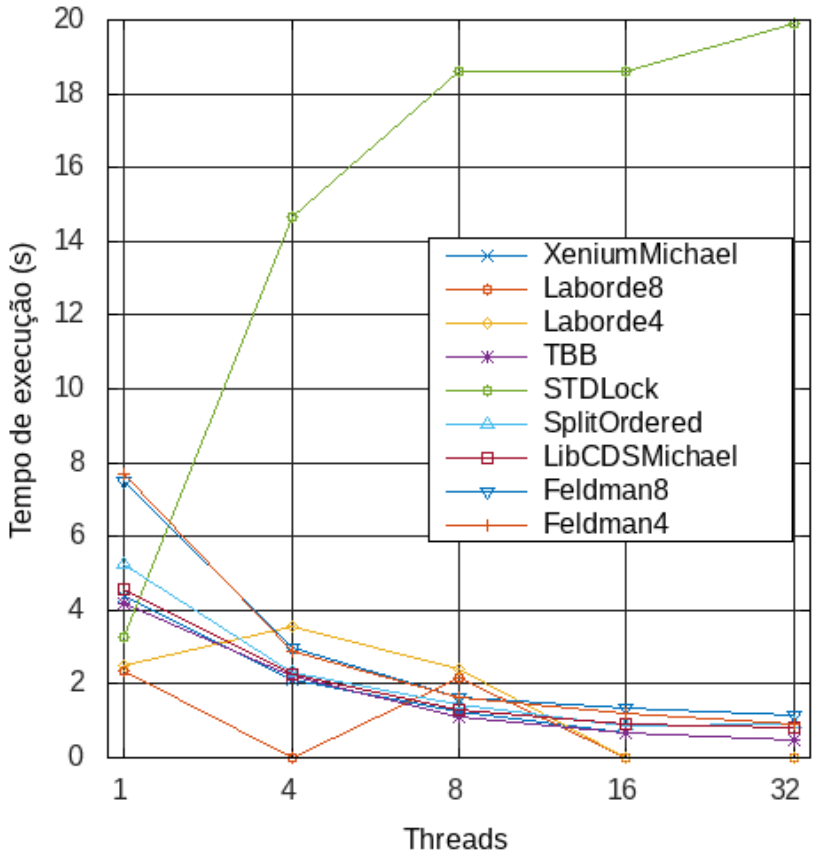

Fonte: O autor (2023)

Figura 51 – *Resultados dos testes com chaves previamente inseridas, muitas chaves distintas, valores pequenos e distribuição C.*



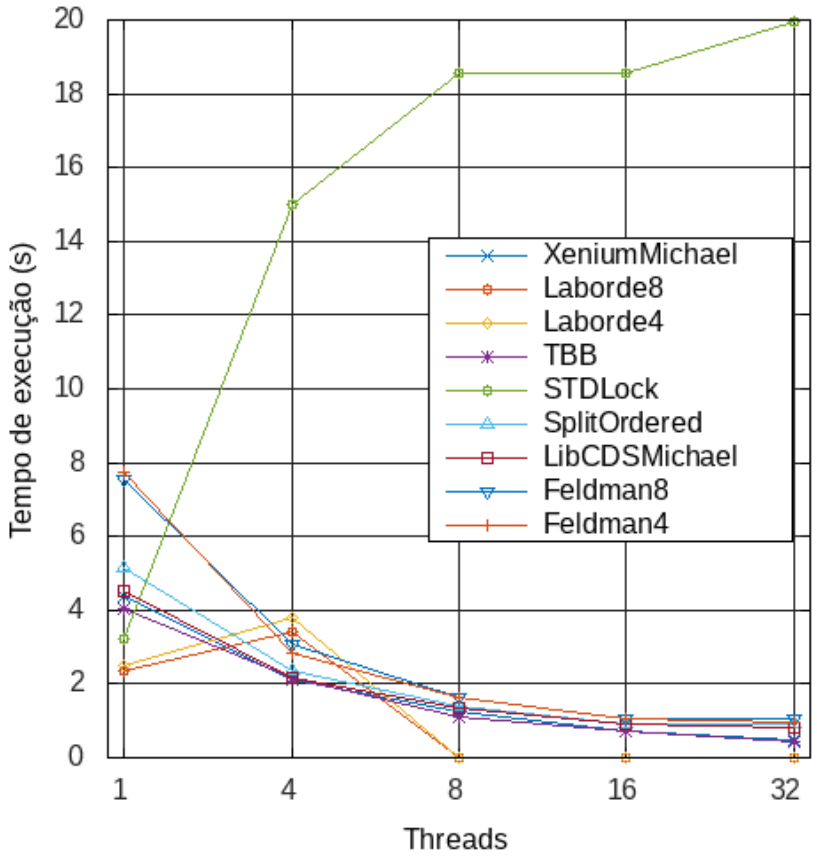

Fonte: O autor (2023)

Figura 52 – *Resultados dos testes com tabelas inicialmente vazias, poucas chaves distintas, valores grandes e distribuição C.*



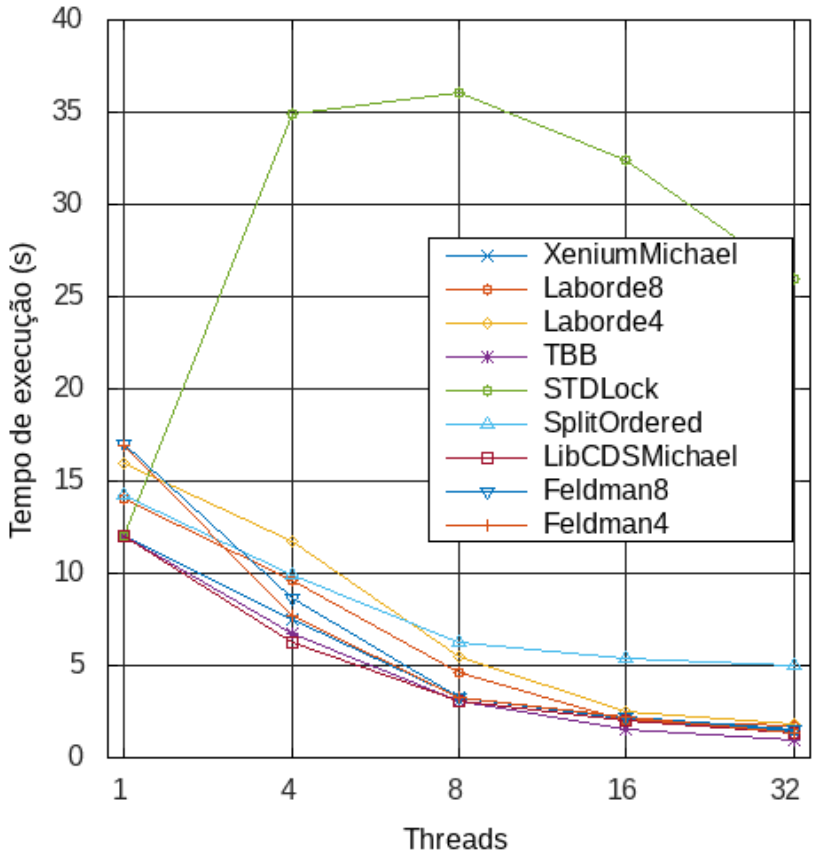

Fonte: O autor (2023)

Figura 53 – *Resultados dos testes com chaves previamente inseridas, poucas chaves distintas, valores grandes e distribuição C.*



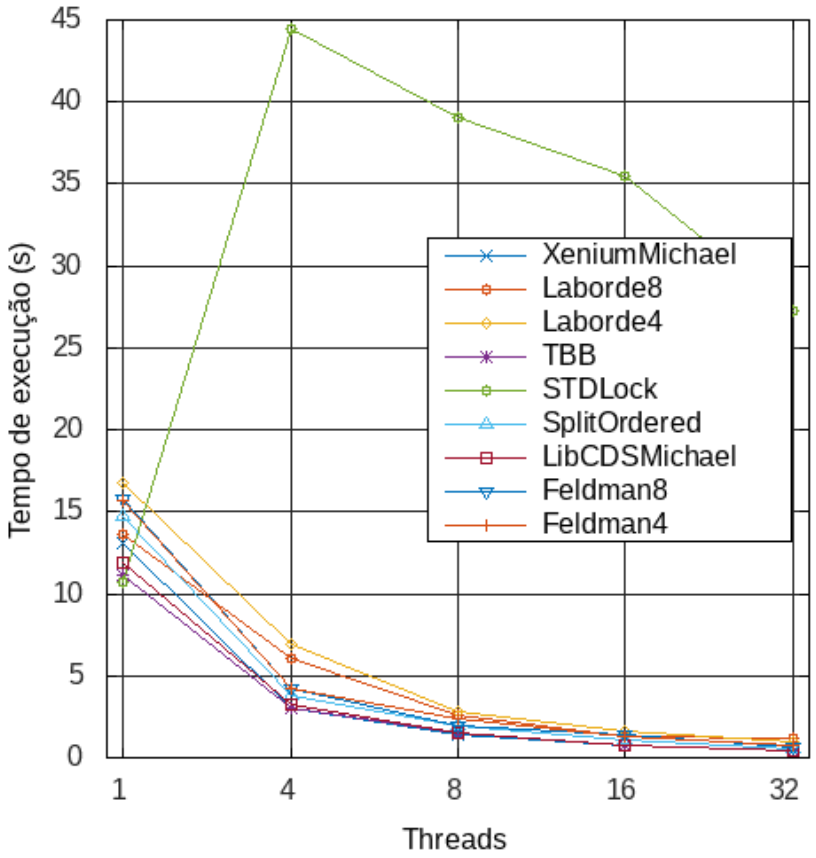

Fonte: O autor (2023)

Figura 54 – *Resultados dos testes com tabelas inicialmente vazias, muitas chaves distintas, valores grandes e distribuição C.*




Fonte: O autor (2023)

Figura 55 – *Resultados dos testes com chaves previamente inseridas, muitas chaves distintas, valores grandes e distribuição C.*

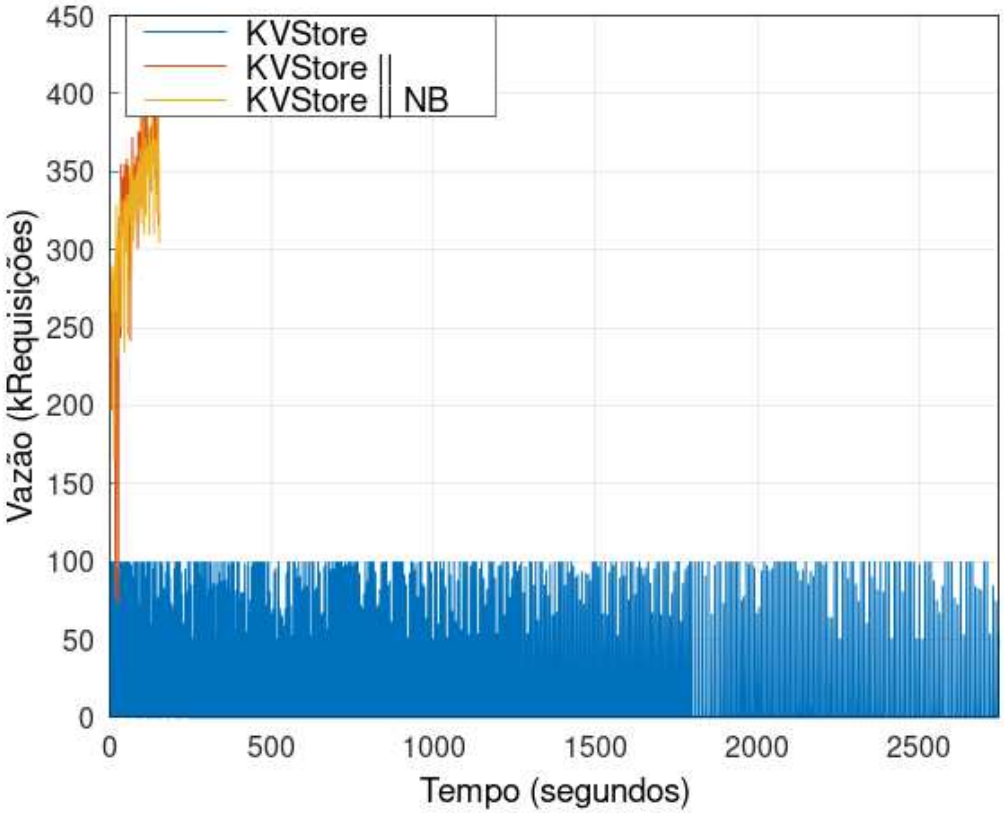



Fonte: O autor (2023)

# APÊNDICE D – GRÁFICOS PRODUZIDOS PARA A AVALIAÇÃO DE DESEMPENHO DOS PROTÓTIPOS DE *KEY-VALUE STORE*

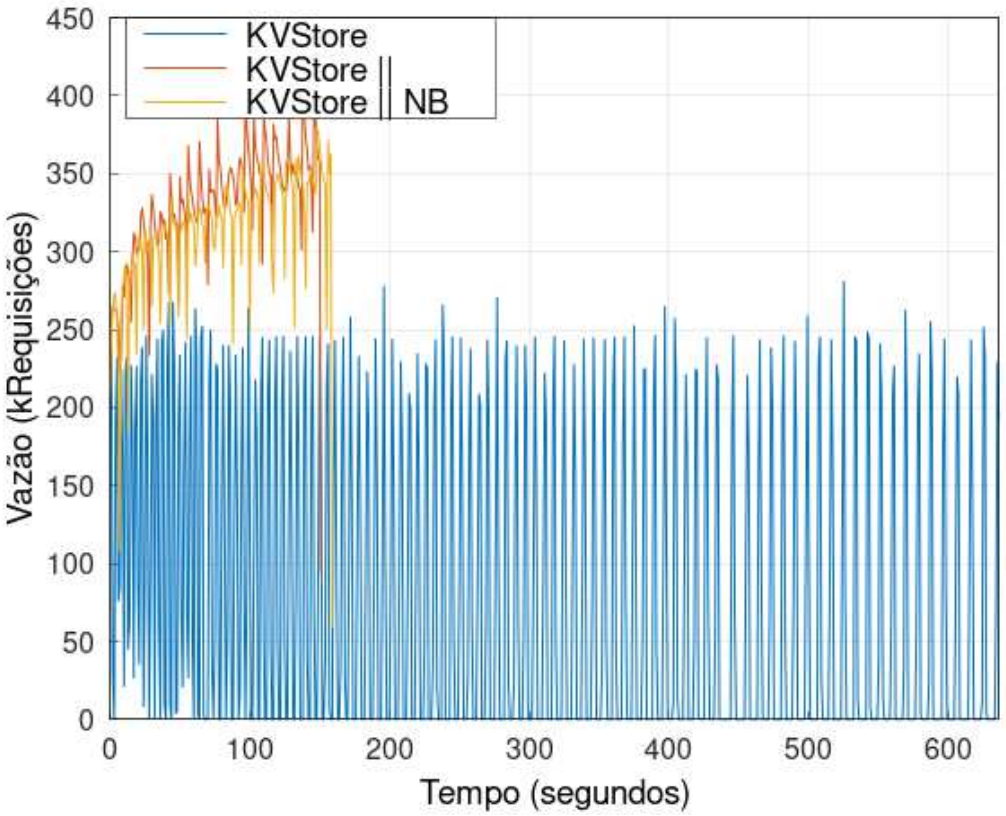
Os gráficos produzidos para auxiliar na avaliação de desempenho dos protótipos de *Key-Value Store* são apresentados nesse apêndice.

Figura 56 – Resultados dos experimentos com carga D e $\delta p$ = 100.000.



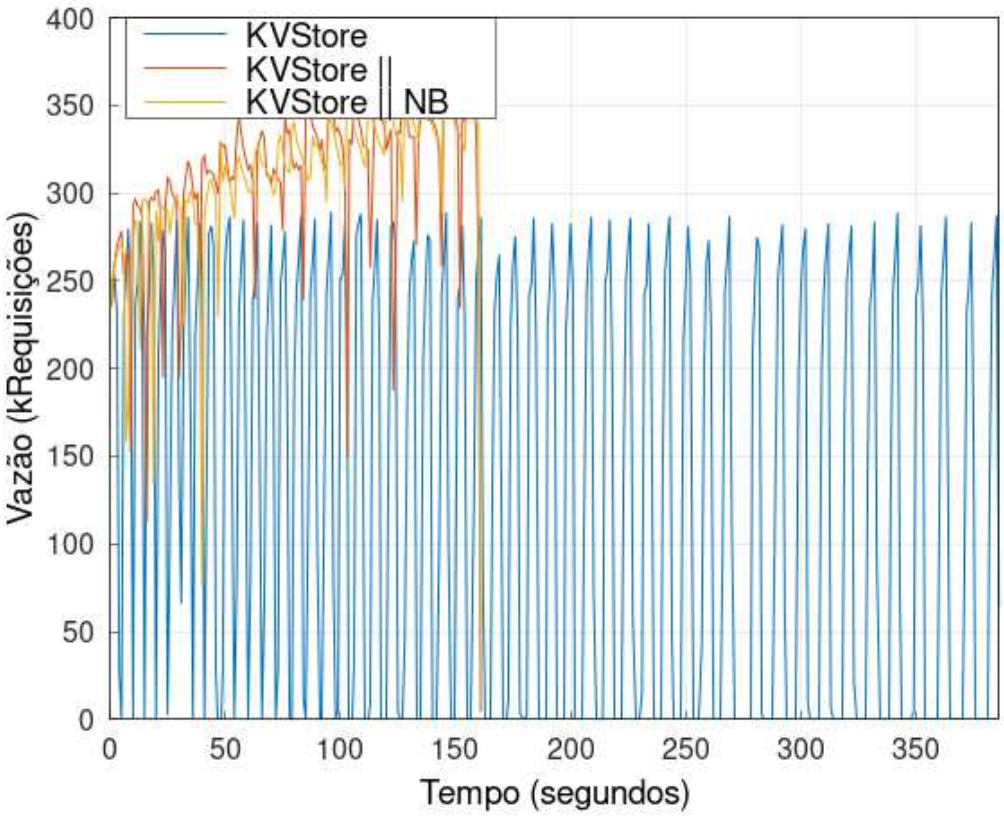

Fonte: O autor (2023)

Figura 57 – Resultados dos experimentos com carga D e $\delta p$ = 500.000.



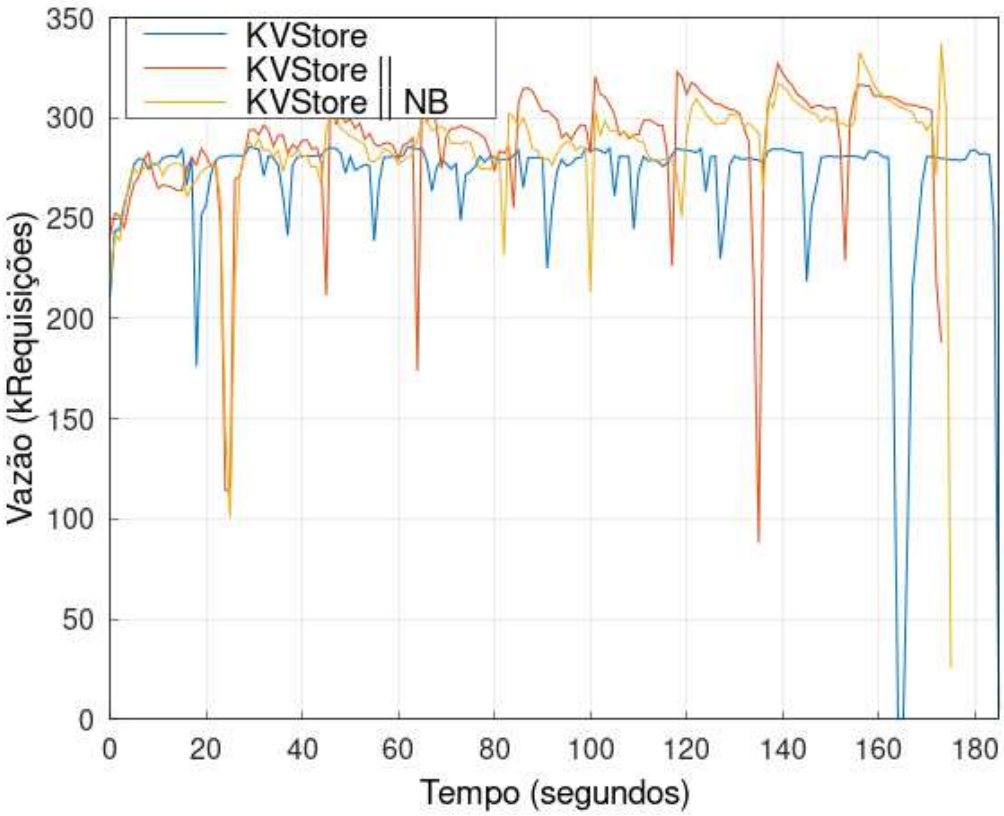

Fonte: O autor (2023)

Figura 58 – Resultados dos experimentos com carga D e $\delta p$ = 1.000.000.



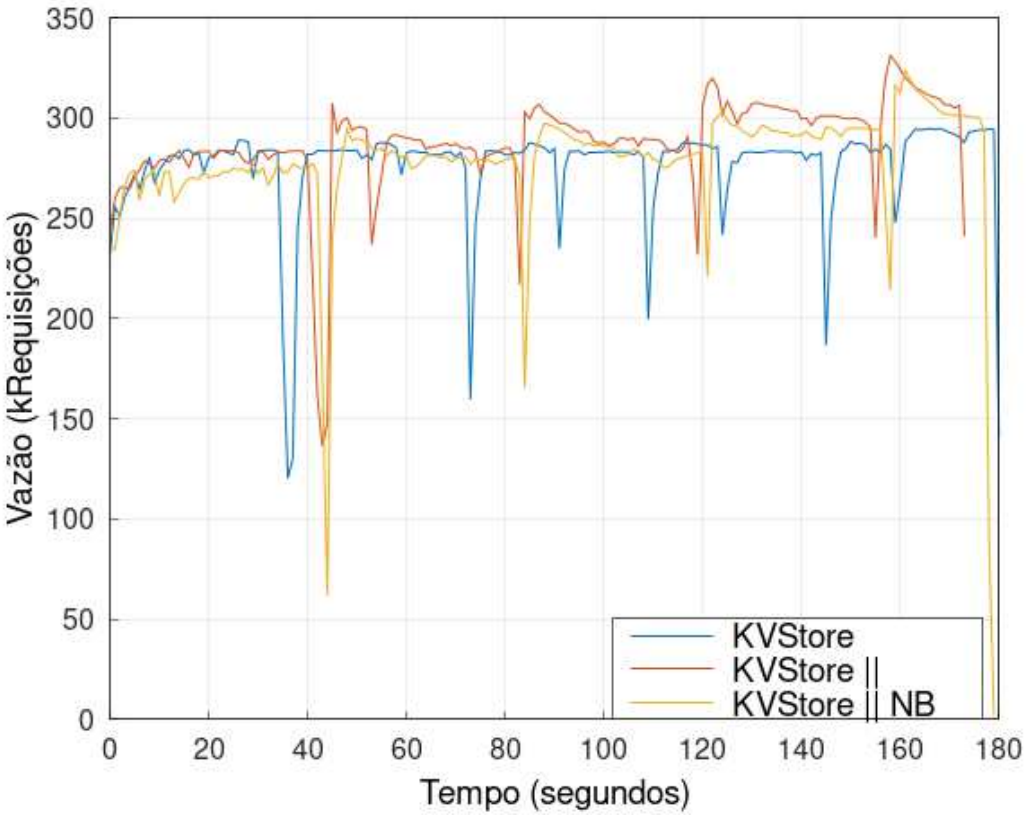

Fonte: O autor (2023)

Figura 59 – Resultados dos experimentos com carga D e $\delta p$ = 5.000.000.



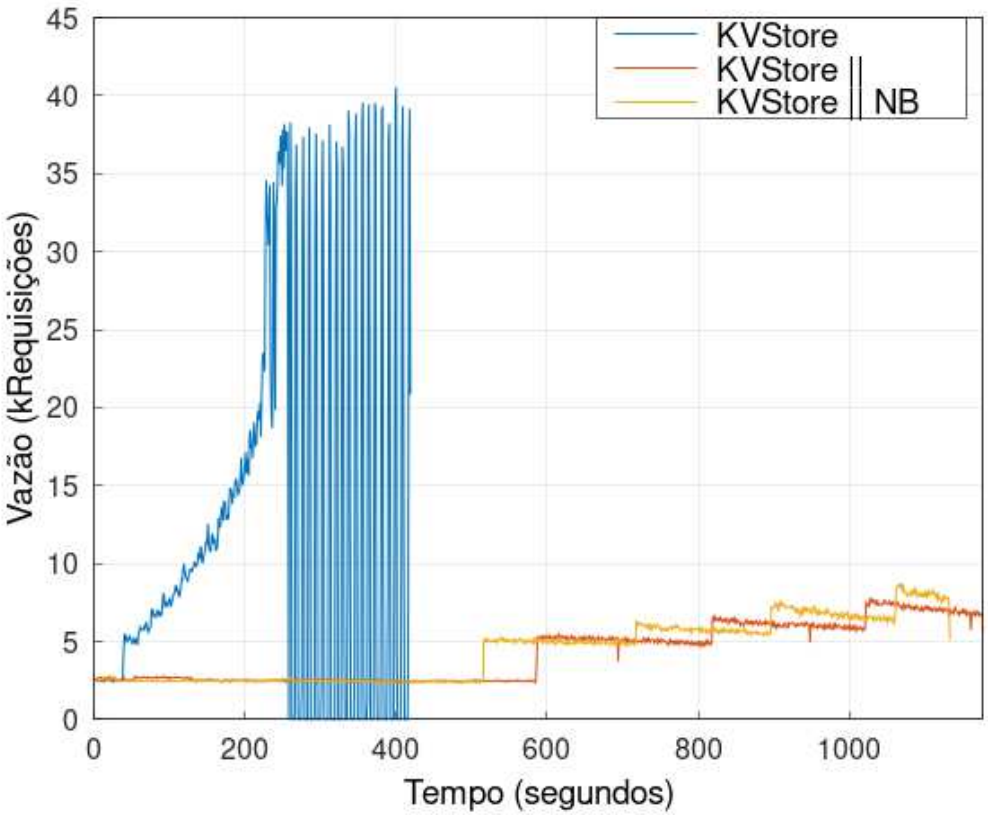

Fonte: O autor (2023)

Figura 60 – Resultados dos experimentos com carga D e $\delta p$ = 10.000.000.



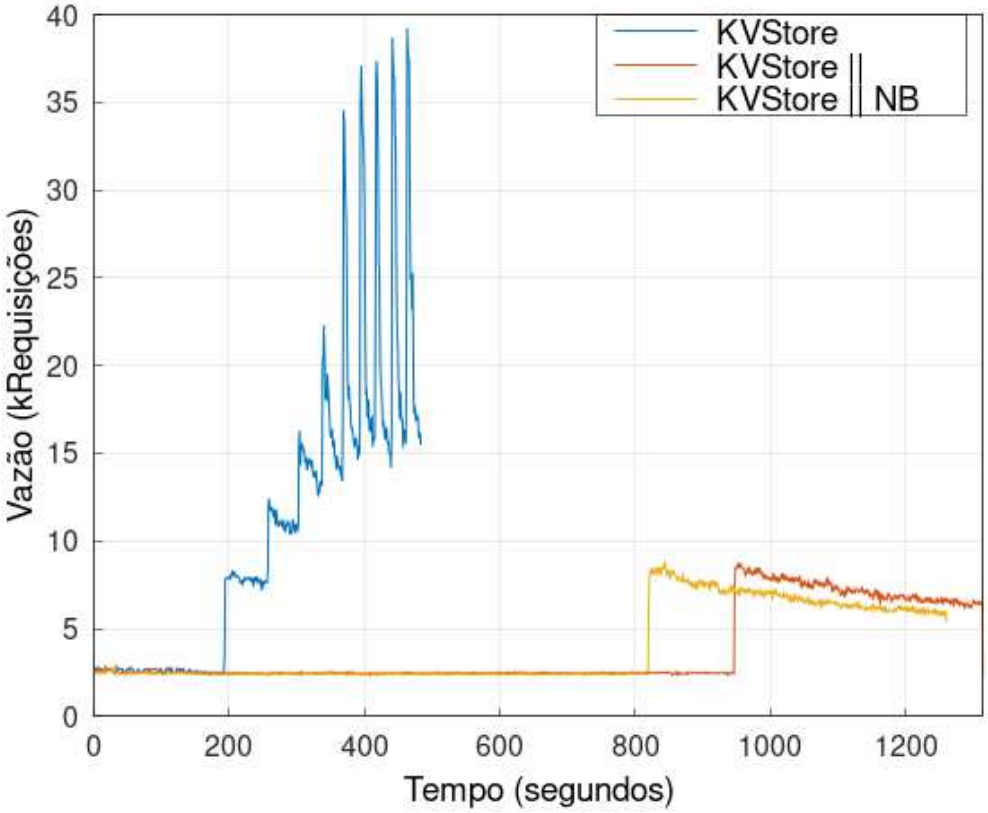

Fonte: O autor (2023)

Figura 61 – Resultados dos experimentos com carga E e $\delta p$ = 100.000.



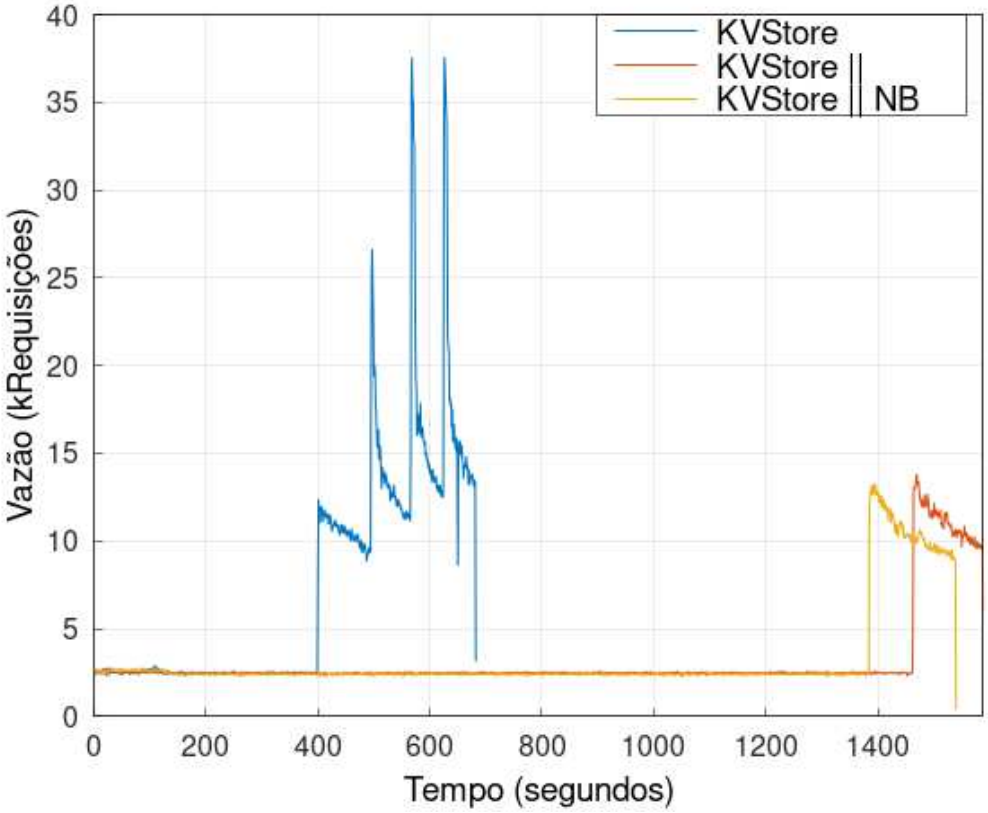

Fonte: O autor (2023)

Figura 62 – Resultados dos experimentos com carga E e $\delta p$ = 500.000.




Fonte: O autor (2023)

Figura 63 – Resultados dos experimentos com carga E e $\delta p$ = 1.000.000.




Fonte: O autor (2023)

# APÊNDICE E – ARTIGO

Os resultados obtidos durante o desenvolvimento do trabalho possibilitaram a produção do artigo Análise de Desempenho de Hash Tables Não-Bloqueantes na Linguagem C++ (LUIZ; MENDIZABAL, O., 2023), que foi submetido ao fórum de iniciação científica do evento XIII Escola Regional de Alto Desempenho da Região Sul (ERAD/RS), aceito e posteriormente publicado.

# 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

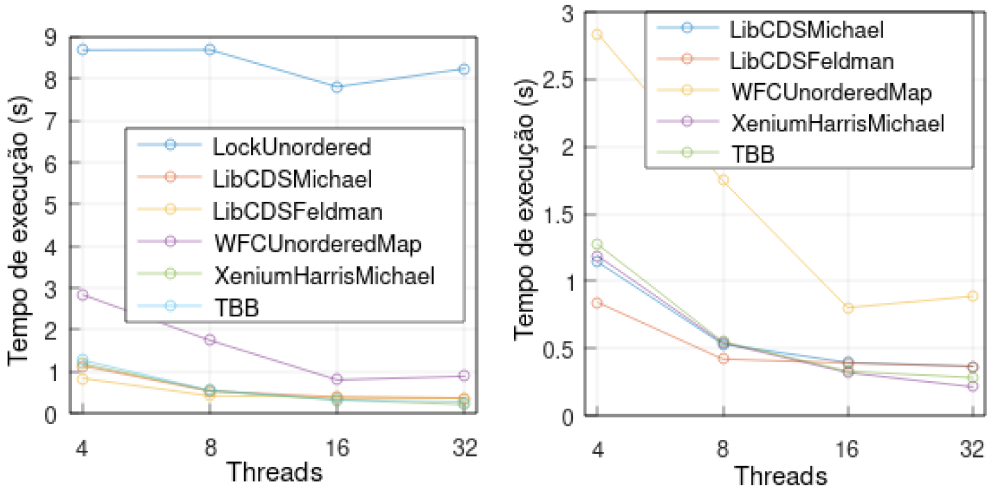
Os testes foram realizados em um computador com quatro processadores Intel Xeon E5-4620 2.2 GHz com 8 núcleos e 16 MB cache e 128 GB de memória RAM DDR3. O Sistema operacional usado foi o Ubuntu v20.04 de 64 bits. O programa de testes foi desenvolvido na linguagem C++17 e compilado com o *gcc v9.4.0*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

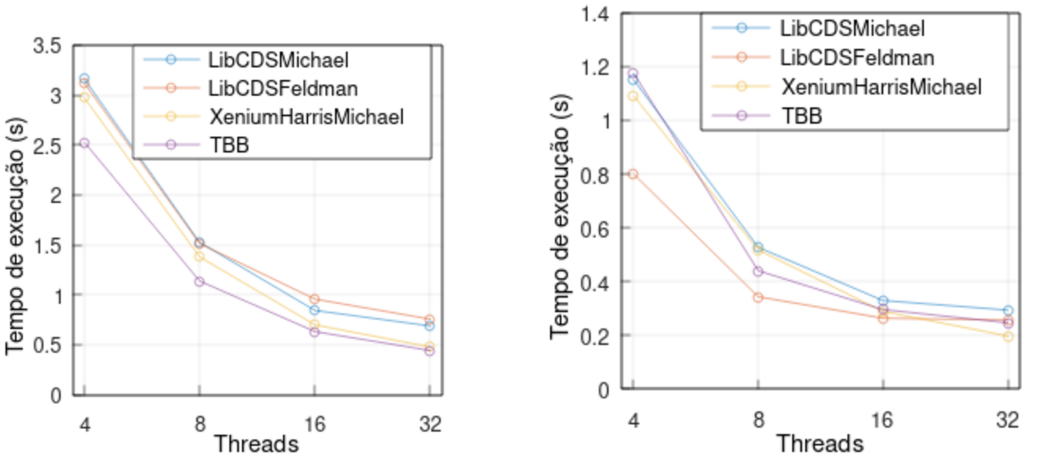
Na Figura 1(a), o custo ao empregar *mutex* com a implementação *LockUnordered* é visível. Este perfil de menor desempenho se mantém em todos os testes. A Figura 1(b) contém os mesmos dados da Figura 1(a), omitindo a implementação *LockUnordered*. É possível observar que, em média, para armazenamento 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.

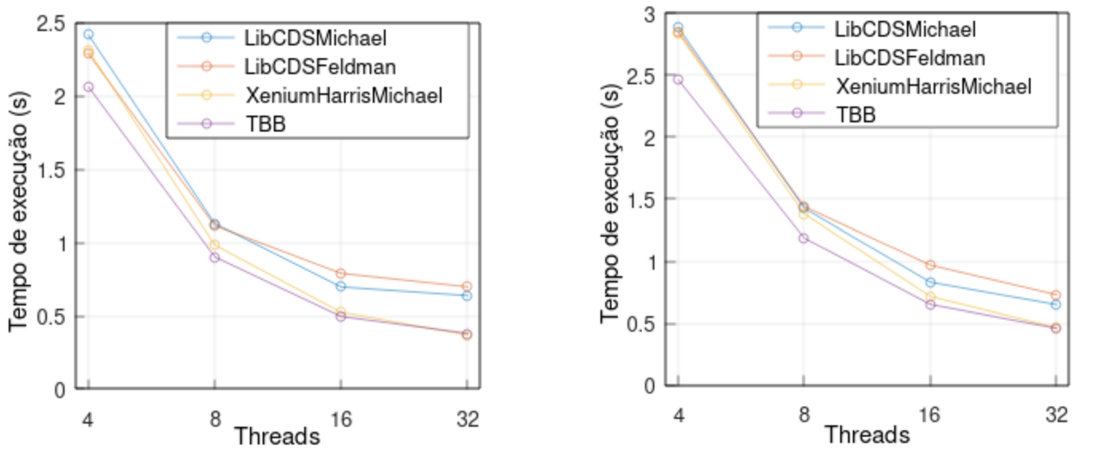
**Figure 2. Tempo médio de execução combinando os cenários.**

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.