Um Estudo Inicial de Desempenho de Protocolos de Exclusão Mútua Escaláveis

Pedro de Matos Fedricci[1], **Calebe de Paula Bianchini**[1,2]

[1]Departamento de Ciência de Computação – Centro Universitário da FEI
São Bernardo do Campo – SP – Brasil

[2]Faculdade de Computação e Informática – FCI
Universidade Presbiteriana Mackenzie – São Paulo, SP – Brasil

pedromfedricci@gmail.com

calebe@fei.edu.br[1], calebe.bianchini@mackenize.br[2]

Resumo. *Protocolos de exclusão mútua escaláveis são mecanismos otimizados para gerenciar o acesso exclusivo e concorrente a recursos compartilhados em contexto de alta contenção, onde múltiplas threads competem intensamente pelo mesmo recurso. Este estudo compara o desempenho de alguns desses protocolos contra os não escaláveis, em contexto de alta contenção, agrupados por políticas de espera ativa e passiva.*

1. Introdução

A sincronização de acesso e modificação exclusiva é frequentemente implementada por *locks*, mecanismos que protegem as seções críticas por meio de protocolos diversos, com o propósito de linearizar as operações. O uso de algoritmos de exclusão mútua não otimizados para o *hardware* e carga de trabalho da aplicação pode aumentar desnecessariamente o tempo sequencial, reduzindo o desempenho [Guiroux 2018]. Por conta disso, o estudo de protocolos de exclusão mútua é um fator importante para melhorar a escalabilidade de programas *multithread*.

O objetivo desse trabalho é analisar o desempenho de um grupo de protocolos, implementados na linguagem de programação Rust, agrupados por políticas de espera ativa e passiva, e em contexto de alta contenção no acesso a recursos compartilhados. Os protocolos escaláveis apresentados nesse artigo foram implementados em Rust, enquanto os demais foram selecionados no ecossistema e biblioteca padrão dessa linguagem.

2. Coerência de Cache, Alta Contenção e Políticas de Espera

O protocolo de coerência de *cache* é um conjunto de regras que garante a consistência das cópias de um mesmo dado armazenadas nos *caches* de diferentes núcleos em sistemas *multithread*. Isso é realizado por meio de mecanismos como os protocolos de invalidação, que propagam rapidamente qualquer modificação, assegurando que exista no máximo uma cópia gravável do dado por linha de *cache* a qualquer momento. Essa abordagem evita que *threads* acessem versões desatualizadas dos dados [Scott and Brown 2013].

Alta contenção no acesso a recursos compartilhados ocorre quando um grande número de *threads* ou processos competem simultaneamente pelo mesmo recurso com a intenção de modificá-lo. Essa concorrência por acesso gravável gera conflitos frequentes,

introduz atrasos e impõe sobrecarga aos mecanismos de sincronização. Os impactos negativos no desempenho são devido à pressão sobre os mecanismos de coerência de *cache* por conta da constante invalidação da cópia do dado gravado [Herlihy et al. 2020].

As políticas de espera ativa e passiva definem estratégias distintas para gerenciar a contenção de acesso a recursos compartilhados [Silberschatz et al. 2013]. Na espera ativa, a *thread* mantém-se em um laço constante, verificando a disponibilidade do *lock*, o que pode reduzir a latência, mas aumenta o consumo de ciclos de CPU. Na espera passiva, a *thread* é bloqueada pelo sistema operacional até que o recurso seja liberado, otimizando o uso da CPU, mas introduz um *overhead* relacionado à suspensão e retomada da execução.

3. Procolos de Exclusão Mútua Escaláveis

O primeiro protocolo de exclusão mútua escalável estudado é o MCS *lock* [Mellor-Crummey and Scott 1991]. O mecanismo desse *lock* é baseado em fila explícita, onde as *threads* são enfileiradas e servidas em ordem de chegada (FIFO). Cada *thread* mantém um nó de fila, e anexa esse nó atomicamente à cauda da fila. Ao liberar o *lock*, o detentor atual sinaliza a próxima *thread* na fila, evitando assim a contenção a um único estado global do *lock*. A implementação desse *lock* é providenciada pela biblioteca *mcslock*[1].

CLH [Craig 1993] [Magnusson et al. 1994], o segundo protocolo estudado, é produto de trabalhos independentes que definem um mesmo mecanismo de *lock* baseado em fila implícita. Como o MCS *lock*, as *threads* são enfileiradas e servidas em ordem de chegada (FIFO), e cada *thread* aloca um nó para vincular-se à fila de forma atômica. Novamente, cada *thread* gira em seu próprio nó, reduzindo o tráfego no barramento. A implementação desse *lock* é providenciada pela biblioteca *clhlock*[1].

O terceiro protocolo é o Malthusian *lock* [Dice 2017]. Esse *lock* também é baseado em gerenciamento de filas. Porém, diferentemente dos anteriores, o protocolo não garante que as *threads* são servidas em ordem de chegada, somente que a política de seleção é eventualmente justa. Esse protocolo introduz um mecanismo adaptativo para regular a taxa de chegada de *threads*, restringindo intencionalmente a concorrência. A implementação desse *lock* é providenciada pela biblioteca *malthlock*[1].

Hemlock [Dice and Kogan 2021], o último protocolo escalável estudado, é um *lock* baseado em fila implícita, onde as *threads* são servidas em ordem de chegada (FIFO). O Hemlock é extremamente compacto e oferece *spin* local para as *threads* enfileiradas na maioria das situações, reduzindo a contenção no acesso do estado do *lock*. Esse protocolo não suporta políticas de espera passivas, pois seu modelo é incompatível com as interfaces de sistema operacional necessárias para essas políticas. A implementação desse *lock* é providenciada pela biblioteca *hemlock*[1].

4. Experimento

Para a realização dos experimentos, foi utilizado um sistema com o processador AMD Ryzen™ 3 7330U, com 4 núcleos e 8 *hyperthreads*, operando em uma frequência base

[1] Veja o repositório de mesmo nome da biblioteca citada em `https://github.com/pedromfedricci?tab=repositories`

de 2.3 GHz; com 16 GB de memória RAM DDR4 de 3200 MHz; e sistema operacional Linux (GNU/Linux), *kernel* na versão 6.13.7, com suporte a SMP (*Symmetric Multi-Processing*) e Preempção Dinâmica (`PREEMPT_DYNAMIC`) ativadas. A ferramenta de análise de desempenho utilizada no experimento foi o Criterion.rs[2], que executa os testes de forma iterativa, coletando múltiplas medições de tempo de execução para cada função avaliada, aplicando técnicas estatísticas para eliminar valores atípicos e calcular métricas relevantes, como a média, o desvio padrão e os intervalos de confiança.

Neste experimento, um *array* de 100 números de ponto flutuante de 64 *bits* é utilizado como dado compartilhado, com *padding* de cache para evitar o *false sharing*. As *threads* são criadas sequencialmente em um laço, e cada *thread* inicializa suas variáveis locais necessárias para as operações aritméticas e de bloqueio (nós de filas reutilizáveis). Após essa fase de inicialização, todas as *threads* se sincronizam por meio de uma barreira para garantir que estejam prontas para a execução da seção crítica, que é medida com precisão por um cronômetro monotônico não decrescente de alta resolução. Cada *thread* executa a seção crítica uma única vez, sem operações intermediárias, na qual um laço de 10.000 iterações realiza multiplicações: cada elemento do *array* compartilhado é multiplicado por um valor de ponto flutuante que é incrementado localmente, simulando uma carga computacional intensa e aumentando a contenção do *lock*. Essa seção crítica apresenta, em média, uma duração de 1,4 milissegundos. Ao final, o tempo decorrido por cada *thread* é retornado e coletado para o cálculo da média dos tempos de execução.

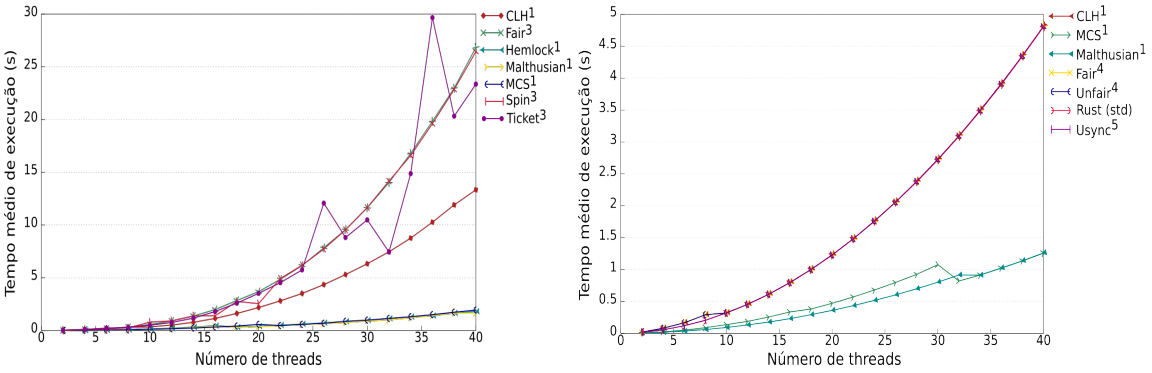
Os resultados experimentais são apresentados em dois gráficos que agrupam os dados de desempenho conforme a política de espera utilizada, sendo uma para espera ativa e outra para espera passiva, conforme mostra a Figura 1. Em ambos, o eixo horizontal mostra o número de *threads* concorrentes, variando de 2 até 40 *threads* (5 vezes o número de CPUs lógicas), em incrementos de 2. O eixo vertical indica o tempo médio que cada protocolo leva para completar um número de seções críticas igual ao número de *threads* (valores menores representam melhor desempenho), onde cada *thread* executa uma única seção crítica.

O gráfico da Figura 1a ilustra a política de espera ativa, onde as *threads* executam um laço de *busy waiting* com a instrução `PAUSE` enquanto aguardam a liberação do *lock*, e nele são apresentados os protocolos não escaláveis do projeto *spin-rs*[3]. O gráfico da Figura 1b exibe a política de espera passiva, na qual é empregada uma abordagem *spin-then-park* para a espera. Nesse gráfico, são mostrados os protocolos escaláveis (exceto o Hemlock) e, para comparação, também são incluídos o *mutex* da biblioteca padrão do Rust e os *mutexes* dos projetos *parking_lot*[4] e *usync*[5].

5. Conclusões

Observa-se que os protocolos de exclusão mútua escaláveis apresentam desempenho superior ou significativamente superior aos protocolos não escaláveis, em contexto de alta contenção, quando empregados em conjunto com políticas de espera ativa, nas quais não ocorre intervenção do sistema operacional para suspender as *threads*. Por outro lado, ao

[2] `https://github.com/bheisler/criterion.rs`
[3] `https://github.com/zesterer/spin-rs`
[4] `https://github.com/Amanieu/parking_lot`
[5] `https://github.com/kprotty/usync`



(a) Políticas de espera ativas **(b) Políticas de espera passivas**

Figura 1. Resultados experimentais de desempenho dos *locks* selecionados

utilizar políticas de espera passiva, onde o sistema operacional suspende as *threads* que não possuem o *lock*, os protocolos escaláveis demonstram desempenho igual ou superior aos outros protocolos. Isso ocorre, em grande parte, por conta da redução substancial da pressão sobre o protocolo de coerência de *cache* no acesso ao estado do *lock* por essas políticas. A suspensão contínua das *threads* em espera reduz o acesso ao estado compartilhado do *lock* e, por consequência, mitiga as sincronizações custosas entre *caches*.

Referências

Craig, T. (1993). Building fifo and priority-queuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, Department of Computer Science, University of Washington.

Dice, D. (2017). Malthusian locks. In Alonso, G., Bianchini, R., and Vukolic, M., editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 314–327. ACM.

Dice, D. and Kogan, A. (2021). Hemlock: Compact and scalable mutual exclusion. In Agrawal, K. and Azar, Y., editors, *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, pages 173–183. ACM.

Guiroux, H. (2018). *Understanding the performance of mutual exclusion algorithms on modern multicore machines.* PhD thesis, Université Grenoble Alpes.

Herlihy, M., Shavit, N., Luchangco, V., and Spear, M. (2020). *The art of multiprocessor programming*. Newnes.

Magnusson, P. S., Landin, A., and Hagersten, E. (1994). Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*, pages 165–171.

Mellor-Crummey, J. M. and Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65.

Scott, M. L. and Brown, T. (2013). *Shared-memory synchronization*. Springer.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2013). *Operating system concepts essentials*. Wiley Publishing.