Execução Paralela de Transações Baseada em Análise Dinâmica e Auto-Verificável de Conflitos *

Jefferson P. Silva[1], **Eduardo Alchieri**[1], **Fernando Dotti**[2]

[1]Departamento de Ciência da Computação
Universidade de Brasília – Brasília – DF – Brasil

[2]Programa de Pós-Graduação em Ciência da Computação – Escola Politécnica
Pontifícia Universidade Católica do Rio Grande do Sul – Porto Alegre – RS – Brasil

Abstract. *The solutions for blockchains usually sequentially execute transactions by miners, allowing validators to reproduce this execution to validate its result. However, this approach does not allow to exploit modern multi-core resources efficiently, thus limiting performance and increasing application latency. Existing solutions that allow parallel execution of some transactions usually use static analysis (before execution) or a directed aciclic graph (DAG) to deal with conflic/dependencies among transactions. We propose a new solution to allow parallel execution of transactions through a dynamic conflict analysis using a DAG in a way that conflicts are self-verifiable at validators. In order to evaluate the benefits of our proposal over traditional sequential execution, we created for smart contract applications that simulate the execution of a real blockchain. Experiments show that our proposal outperforms the sequential execution by up to* $5\times$.

Resumo. *As soluções para blockchains geralmente executam transações sequencialmente pelos mineradores, permitindo que validadores reproduzam esta execução para validar o seu resultado. Porém, tal abordagem é incapaz de explorar os recursos multi-core modernos de forma eficiente, limitando assim o desempenho e aumentando a latência das aplicações. Soluções existentes que permitem execução paralela de uma parte das transações geralmente fazem uso de análise estática (antes da execução) ou utilizam um grafo acíclico dirigido (DAG) para lidar com conflitos/dependências entre transações. Neste contexto, propomos uma nova solução para permitir execuções paralelas em uma blockchain utilizando análise dinâmica de conflitos através da utilização de um DAG onde os conflitos de uma transação podem ser auto-verificados pelos validadores. A fim de avaliar os benefícios da nossa proposta sobre a execução sequencial tradicional, criamos quatro aplicações de contratos inteligentes que simulam a execução de uma blockchain real. Experimentos mostram que nossa proposta atinge uma aceleração que supera em até* $5\times$ *a execução sequencial.*

1. Introdução

A blockchain surgiu como um paradigma para construção de aplicações seguras e descentralizadas com potencial para revolucionar vários aspectos de nossa vida digital, sendo introduzido por Satoshi Nakamoto em 2009 com o Bitcoin [Nakamoto 2008] para realizar

*Este trabalho recebeu apoio do CNPq através do projeto Universal 420092/2018-8.

transações eletrônicas sem interferência de terceiros. Nestes sistemas, clientes encaminham transações para serem executadas por mineradores, os quais executam e empacotam tais transações em blocos. Mineradores propõem novos blocos para serem adicionados a blockchain e para isso seguem um protocolo de consenso global em que cada minerador concorda sobre quais blocos serão adicionados na blockchain. Cada bloco contém um *hash* criptográfico do bloco anterior, dificultando a adulteração da blockchain. Um bloco somente é aceito para ser incluído na blockchain caso seja válido, i.e., validadores verificam se as informações no bloco estão corretas bem como o resultado da execução pelo minerador. Um mesmo nó pode executar as duas tarefas e funcionar como minerador e validador (como por exemplo os *full nodes* da rede Ethereum).

Para permitir a validação, um caminho natural é a execução sequencial das transações de forma que os validadores consigam facilmente reproduzir a execução realizada pelo minerador. Como a execução sequencial não consegue tirar proveito de sistemas *multi-cores* modernos, soluções recentes permitem a execução paralela de transações que não conflitam (e.g.; [Dickerson et al. 2017, Anjana et al. 2019, Saraph and Herlihy 2019]). Duas transações conflitam ou são dependentes, caso acessem um mesmo objeto compartilhado e pelo menos uma delas altera o seu valor. Caso contrário, são não conflitantes ou independentes.

Em geral, estas soluções (i) fazem uma análise estática antes da execução para separar transações conflitantes das não conflitantes [Bartoletti et al. 2020], criando um conjunto de transações que pode ser executado de forma paralela, ou ainda (ii) utilizam mecanismos como *locks* [Saraph and Herlihy 2019] ou memória transacional (STM) [Anjana et al. 2019] para executar as transações de forma especulativa e armazenar os conflitos encontrados, geralmente em um grafo direcionado acíclico (DAG). No caso (i), um processamento adicional é necessário para separar os conjuntos, além de sempre supor o pior caso para as transações conflitantes pois descarta a possibilidade de transações conflitantes não executarem ao mesmo tempo físico e, consequentemente, não acessarem os mesmos objetos ao mesmo tempo. Já no caso (ii), o DAG (ou a informação sobre quais transações conflitam) é enviado para os validadores, os quais devem seguir as dependências reportadas no DAG para validação da execução realizada pelo minerador. No entanto, um minerador malicioso pode incluir intencionalmente dependências inexistentes no DAG de forma a atrasar a verificação pelos validadores. Estes trabalhos apenas indicam que um minerador poderia ser recompensado por apresentar um DAG com elevado grau de paralelismo, porém a segurança destas redes é baseada no poder computacional dos nós [Garay et al. 2015] e, consequentemente, um minerador malicioso poderia atrasar outros nós da rede (mineradores que também são validadores) e controlar a rede. Além disso, outro problema relacionado com a execução especulativa é que é necessário fazer o *rollback* de transações quando conflitos são encontrados, o que não é possível para qualquer aplicação.

Visando contornar estas limitações, este trabalho propõe a utilização de análise dinâmica de conflitos durante a execução e da utilização de uma DAG para armazenar as informações sobre os conflitos de uma forma auto-verificável. De forma resumida, um *escalonador* obtém as transações que pertencerão a um bloco e as insere, uma a uma, no DAG adicionando também os conflitos conforme uma função de conflito fornecida ao sistema. Esta função deve analisar os dados de duas transações e retornar verdadeiro

caso conflitam ou falso caso contrário. Paralelamente, um *conjunto de executores* obtém do DAG as transações com os conflitos já resolvidos e as executam, resolvendo novos conflitos e liberando novas transações para execução. Note que, diferente dos outros trabalhos, a execução não é especulativa e também os conflitos são auto-verificáveis uma vez que durante a validação é possível verificar se os mesmos realmente existem bastando para isso utilizar a função de conflito.

De forma resumida, este trabalho apresenta as seguintes contribuições:

- Proposta de uma nova forma de executar e/ou validar transações em paralelo, utilizando análise dinâmica durante a execução e um DAG para armazenar conflitos auto-verificáveis.
- Apresenta uma análise de desempenho da solução proposta através da implementação de quatro aplicações de contratos inteligentes que simulam a execução de uma blockchain real. Experimentos mostram que a solução proposta melhora em até $5\times$ o desempenho quando comparado com o modelo sequencial.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados. A Seção 3 discute a proposta deste trabalho, enquanto que a Seção 4 apresenta sua análise experimental. Finalmente, a Seção 5 conclui o trabalho.

2. Trabalhos Relacionados

Esta seção apresenta alguns trabalhos relacionados sobre execução concorrente em blockchains de acordo com a abordagem proposta.

Software Transactional Memory - STM. Bibliotecas STM tem como objetivo instrumentar os acessos simultâneos à regiões de memória associados a diferentes transações, detectando e lidando com conflitos de forma a fornecer um resultado final equivalente à execução sequencial.

Dickerson *et al.* [Dickerson et al. 2017] apresentam uma nova maneira de permitir que mineradores e validadores executem contratos inteligentes em paralelo, com base em técnicas adaptadas de STM. Os mineradores executam transações especulativamente em paralelo, de forma que transações não conflitantes prossigam paralelamente. Os conflitos são descobertos durante o acesso aos objetos compartilhados, sendo que, caso duas transações conflitem, uma delas é revertida (*rollback*) ou atrasada até a outra ser concluída. Esta solução ainda utiliza *locks* abstratos relacionados com cada objeto compartilhado e *logs* de operações para possibilitar o *rollback*. O minerador fornece aos validadores perfis de *locks* que indicam quais transações obtiveram quais *locks* e em qual ordem. Desta forma, os validadores conseguem reconstruir o mesmo escalonamento utilizado no minerador.

Anjana *et al.* [Anjana et al. 2019] propuseram a utilização de STM de leitura-escrita otimista usando protocolos baseados em BTO (*Basic timestamp order*) e MVTO (*Multi-Version Timestamp Order*). Estes protocolos baseados em *timestamps* são usados para identificar os conflitos entre AUs (*Atomic Units*), que são os contratos inteligentes. O minerador executa as AUs usando o RWSTM e constrói um grafo de blocos (BG - *Block Graph*) para armazenar os conflitos dinamicamente em tempo de execução, usando

os *timestamps*. Os validadores concorrentemente executam as AUs no bloco utilizando o BG para lidar com os conflitos.

Devido à quantidade de conflitos e mecanismos necessários para manter o estado consistente, bibliotecas STM de uso geral geralmente sofrem limitações de desempenho em comparação com soluções personalizadas e raramente são implantadas em produção [Cascaval et al. 2008].

Estruturas de dados multi-versão. Estruturas de dados multi-versão são projetadas para evitar conflitos de escrita. Essas estruturas mapeiam locais de memória e valores que são indexados com base em versões que são atribuídas a transações usando *timestamps*.

Saraph e Herlihy [Saraph and Herlihy 2019] propuseram uma abordagem especulativa de duas fases para executar contratos concorrentes na blockchain Ethereum [Buterin 2013]. Na primeira fase, o minerador usa bloqueios (*locks*) e executa as transações em um bloco concorrente, revertendo (*rollback*) e abortando transações que apresentaram conflito, os quais são descobertos através dos *locks*. Todas as transações abortadas são, então, mantidos em um armazém sequencial e executadas na segunda fase sequencialmente. Posteriormente, o validador executa os contratos do armazenamento concorrente em paralelo e os contratos do armazenamento sequencial sequencialmente.

Comitê de *peers*. As soluções vistas até aqui utilizam paralelismo dentro dos *peers* através da utilização de arquiteturas *multi-cores*. Seguindo outra abordagem, Baheti *et al.* [Baheti et al. 2022] propuseram uma solução baseada em *líder–seguidores* que formam um comitê de *peers*. O líder analisa estaticamente as transações, criando diferentes grupos (*shards*) de transações independentes e as distribui aos seguidores para execução concorrente. Quando um bloco é criado com sucesso, o líder envia o bloco proposto para outros pares na rede para validação. Ao receber um bloco, os validadores reexecutam as transações do bloco e aceitam o bloco se atingirem o mesmo estado compartilhado pelo minerador. A validação também pode ser feita em paralelo, seguindo a mesma abordagem líder-seguidor da mineração. No entanto, definir a quantidade ideal de *shards* para um determinado conjunto de transações não é trivial, além de ser necessário mover o estado da aplicação, que pode ser grande, entre o líder e os seguidores.

3. Execução Paralela Baseada em DAG com Conflitos Auto-Verificáveis

Esta seção discute nossa proposta para execução paralela de transações baseada em um DAG, onde os conflitos são analisados e inseridos dinamicamente durante a execução. Além disso, as relações de conflito são auto-verificáveis, o que impede que um minerador malicioso atrase os outros nós da rede para obter vantagens.

3.1. Visão Geral

A Figura 1 apresenta a visão geral da solução empregada para execução paralela de transações. As transações enviadas pelos clientes são armazenadas ① para posterior execução. O escalonador (*thread* escalonadora) obtém uma lista de transações para formar um bloco ②. O tamanho desejado para o bloco define o tamanho desta lista. Então, o escalonador adiciona estas transações uma a uma em um grafo juntamente com os conflitos, i.e., cada vértice representa uma transação e uma aresta entre dois vértices representa

que estas transações conflitam. Neste exemplo, considere que as transações são inseridas na ordem $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8$, e ainda que as transações t_1, t_6 e t_8 conflitam entre si ③. Paralelamente, as transações que não possuem conflitos para serem resolvidos (considerando o exemplo, $t_1, t_2, t_3, t_4, t_5, t_7$) podem ser executadas em qualquer ordem pelos executores (*threads* executoras) disponíveis ④. Após a execução de uma transação, o executor correspondente marca como resolvidos os conflitos associados com a transação. Note que durante uma inserção, o escalonador precisa verificar todas as transações existentes no grafo para definir os conflitos com a transação sendo inserida. Adicionalmente, os executores acessam o grafo concorrentemente para obter transações. Desta forma, os acessos ao grafo precisam ser *thread-safe* [Escobar et al. 2019].

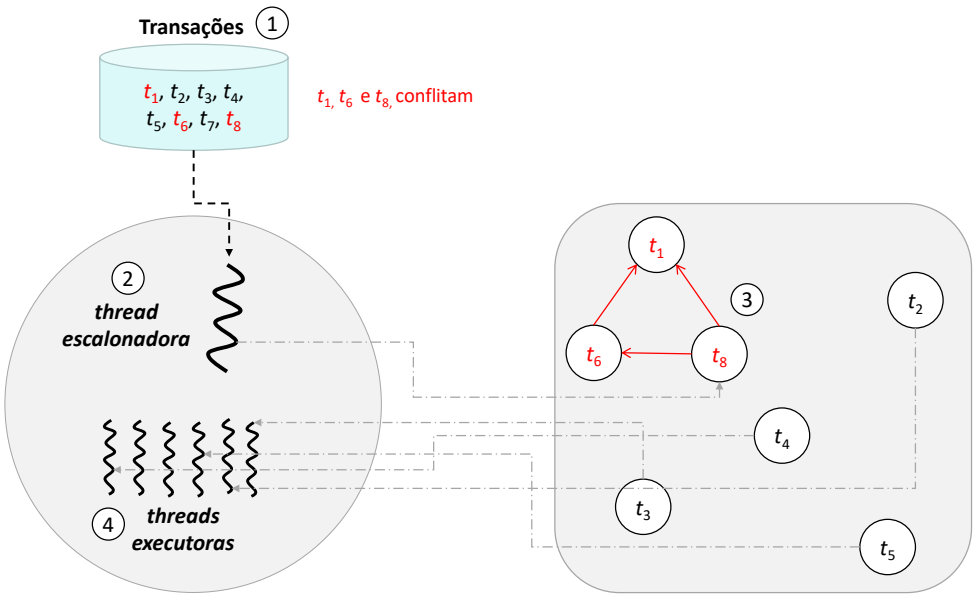

Figura 1. Visão geral.

3.2. Mineração

Abstraímos o DAG utilizado para rastrear dependências entre transações em um Conjunto Ordenado por Conflito (COS) [Escobar et al. 2019], que rastreia a ordem entre transações conflitantes. O COS foi definido por três primitivas:

- *insert (t)*: insere a transação t na estrutura de dados.
- *t: get()*: retorna *t*, se e somente se: *t* está na estrutura de dados; nenhuma operação *get()* já retornou *t*; e não existe *t'* na estrutura de dados inserida antes de *t* e que conflite com *t*.
- *remove(t)*: remove a transação *t* da estrutura de dados.

Durante o processo de mineração não é possível remover informações do COS, pois os validadores precisarão deste conjunto completo para poder refazer a execução do minerador, respeitando todos os conflitos. Desta forma, incluímos uma nova primitiva chamada *resolve(t)* que apenas marca os conflitos relacionados com a transação *t* como resolvidos. Adicionalmente, durante a inserção de uma transação que conflita com outra transação já executada, um conflito já resolvido é adicionado no grafo. Posteriormente, o validador usará estas informações para refazer a execução do minerador.

Neste trabalho, utilizamos uma implementação do COS através de uma DAG livre de bloqueios [Escobar et al. 2019], com pequenas adaptações para suportar a primitiva *resolve*.

O Algoritmo 1 detalha o comportamento da *thread* escalonadora e das *threads* executoras com base no COS. A *thread* escalonadora busca um conjunto de transações que estão armazenadas para execução, o tamanho deste conjunto pode ser configurado (linha 9). Cada transação de um bloco é inserida na estrutura de dados (linha 10). Para isso, é utilizado uma função de conflito fornecida para o sistema, a qual é implementada sobre os tipos de transações e seus parâmetros e deve indicar se duas transações conflitam.

Adicionalmente, um número variável de *threads* executoras obtém transações com conflitos já resolvidos para executar (linha 13). Quando as transações são executadas (linha 14), os conflitos associados são marcados como resolvidos (linha 15), possivelmente liberando novas transações para execução. Depois que todas as transações são executadas, a *thread* escalonadora forma um bloco e propõe tal bloco para validação.

Algoritmo 1 Escalonador e *threads* executoras

```
 1: constantes e estruturas de dados
 2:     T : número de threads executoras
 3:     COS : o conjunto ordenado por conflito
 4: procedure Init()
 5:     for all id ∈ 1..T do                          {para cada executora...}
 6:         start workingThread t_id
 7:     start scheduler
 8: scheduler works as follows:
 9: for all t ∈ next_block do                          {para cada transação do bloco}
10:     COS.insert(t)
11: workingThread t_id executes as follows:
12: loop
13:     t ← COS.get()                                 {pega uma transação t sem dependências}
14:     execute(t)                                    {executa t}
15:     COS.resolve(t)                                {resolve as dependências de t}
```

3.3. Validação

Cada validador deve refazer a execução do minerador, ou uma execução equivalente, para verificar se o resultado das execuções das transações estão corretos. Nossa abordagem permite duas formas de validação (Figura 2): o minerador envia o grafo com os conflitos já computados, o qual é utilizado pelos validadores; ou o minerador envia a ordem em que as transações foram inseridas no grafo, e os validadores utilizam uma arquitetura semelhante ao minerador para inserir as transações no grafo.

Minerador envia o grafo. Caso o minerador enviar o grafo, basta que cada validador utilize as *threads* executoras para executar as transações respeitando as restrições de conflitos contidas no grafo. Para isso, o minerador deve manter e enviar o grafo completo para os validadores, i.e., os conflitos são apenas marcados como resolvidos na linha 15 do Algoritmo 1. Todas estas marcações são removidas nos validadores antes do início da execução, para que os conflitos voltem a ficar todos ativos e sejam novamente removidos conforme as transações forem sendo executadas. Adicionalmente, os validadores devem

utilizar a função de conflito para verificar se cada um dos conflitos reportados no grafo realmente existe (é válido). Caso um conflito inválido seja encontrado ou o resultado da execução de alguma transação não seja o mesmo daquele reportado pelo minerador, o bloco é considerado inválido.

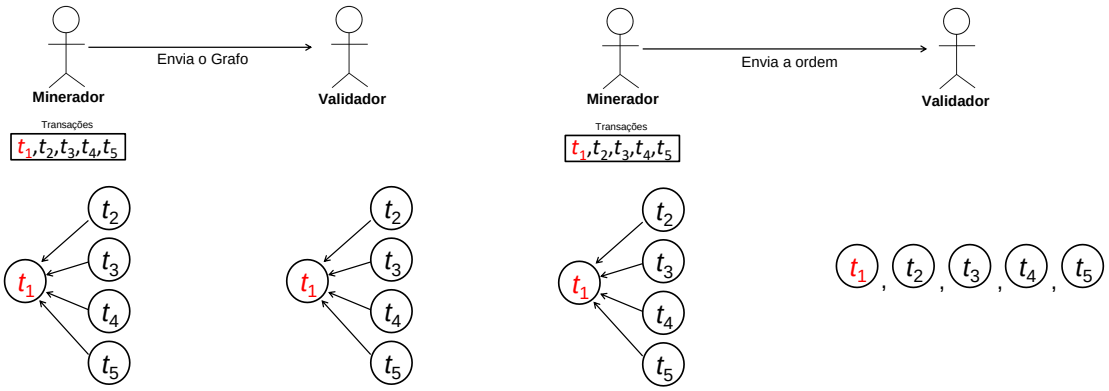

Figura 2. Minerador envia o grafo (esquerda) ou a ordem (direita).

Minerador envia a ordem. Ao invés de enviar todo o grafo, o minerador pode enviar apenas a ordem em que as transações foram inseridas no grafo. Neste caso, o validador deve utilizar a mesma estrutura do minerador para refazer o mesmo (ou equivalente) processamento (Algoritmo 1). Nesta abordagem, o grafo não precisa ser preservado pelo minerador (ou validadores) e a operação *remove* pode ser utilizada na linha 15 do Algoritmo 1. Note que neste caso o próprio validador vai computar os conflitos durante as inserções no grafo. Como as inserções ocorrem na mesma ordem do minerador, a execução é equivalente e o resultado da execução de cada transação deve ser o mesmo. Caso algum resultado seja diferente, significa que o minerador enviou uma ordem diferente daquela por ele usada e o bloco é considerado inválido.

3.4. Lidando com Mineradores Desonestos

Um minerador desonesto pode tentar fazer com que a execução nos validadores seja mais demorada, com o intuito de ganhar vantagens na mineração do próximo bloco. Conforme já comentado, geralmente os mineradores também exercem o papel de validadores. Na abordagem praticada pela maioria das propostas existentes (*e.g.*, [Amiri et al. 2019, Anjana et al. 2019, Dickerson et al. 2017, Saraph and Herlihy 2019]), o validador apenas utiliza as informações sobre conflitos enviadas pelo minerador, sem verificar se os conflitos realmente existem ou se a configuração enviada é aquela que fornece o mesmo grau de paralelismo da execução realizada pelo minerador. Alguns trabalhos também sugerem utilizar incentivos aos mineradores que enviarem seus escalonamentos com maior grau de paralelismo [Dickerson et al. 2017] ou adotar uma estratégia de mineração que dê maior prioridade a transações paralelizáveis [Bartoletti et al. 2020].

No modelo de execução proposto, um minerador desonesto pode: (1) reorganizar as transações e conflitos do grafo, fornecendo um grafo que permite um menor paralelismo; (2) enviar uma ordem diferente daquela utilizada na geração do grafo, com o objetivo de fazer com que os validadores gerem um grafo com menor paralelismo. A Figura 3 ilustra um exemplo do que foi discutido acima. Suponha um bloco com as transações t_1, t_2, t_3, t_4, t_5, e ainda que apenas t_1 conflita com todas as outras transações

pois executa um contrato que escreve em um objeto compartilhado O que é lido pelas demais transações. Neste caso, o minerador pode incluir no grafo as transações na ordem t_1, t_2, t_3, t_4, t_5, gerando o grafo da esquerda, e enviar para os validadores o grafo da direita (ou a ordem t_2, t_3, t_1, t_4, t_5). Claramente, o grafo da direita fornece um menor grau de paralelismo, resultando em um maior tempo para execução. Além disso, todos os conflitos incluídos no grafo são válidos.

Felizmente, em nosso modelo este comportamento malicioso fará com que o resultado final da execução não seja o mesmo para todas as transações do bloco, que será considerado inválido. De fato, a execução de t_2, t_3, t_4, t_5 no minerador é realizada sobre o valor atualizado de O por t_1, enquanto que nos validadores a execução de t_2, t_3 utilizará este valor antes da atualização por t_1. Pelos mesmos argumentos empregados no contexto de replicação máquina de estados [Schneider 1990, Escobar et al. 2019], apenas a ordem de execução entre transações não conflitantes pode ser alterada para produzir uma execução equivalente, e felizmente neste caso o grau de paralelismo fornecido pelo grafo permanece o mesmo.

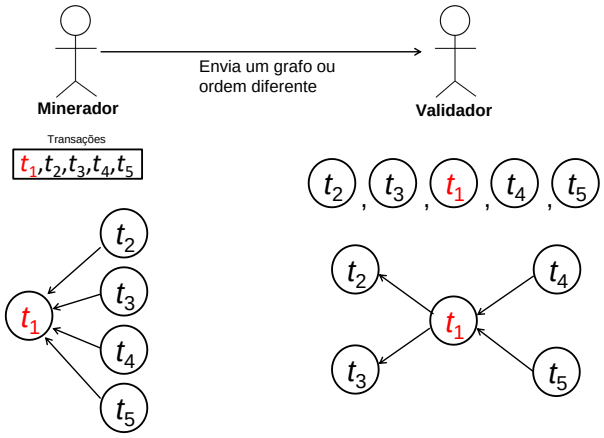

Figura 3. Minerador desonesto

4. Experimentos

Para avaliar o desempenho da solução proposta, implementamos e analisamos quatro aplicações, bem como os contratos inteligentes necessários para acessar tais aplicações. Na blockchain Ethereum [Buterin 2013], as aplicações são escritas na linguagem de programação Solidity [Dannen 2017] e executados na *Ethereum Virtual Machine*. No entanto, esta máquina virtual não suporta *multi-threading*. Portanto, para explorar a utilização eficiente de recursos *multi-core* e melhorar o desempenho, convertemos aplicações escritas na linguagem Solidity em aplicações escritas na linguagem Java e as executamos usando a JVM (*Java Virtual Machine*).

4.1. Aplicações

As quatro aplicações utilizadas neste estudo são descritas a seguir.

Transferência: O contrato inteligente para esta aplicação é uma simples transferência de valores entre carteiras. Todos os usuários da blockchain podem participar deste contrato, sendo enviando (caso tenha saldo na sua carteira) ou recebendo valores. Este contrato contém a carteira de origem, o valor a ser transferido e o destino da transferência.

Caso a carteira que está realizando a transferência tenha saldo suficiente, o valor é subtraído de seu saldo atual e o valor do saldo do destinatário é somado ao valor enviado pelo remetente. Duas transações de transferência conflitam quando compartilharem a mesma origem ou destino, ou ainda quando a origem de uma transação for igual ao destino da outra transação.

Leilão: Este contrato inicia-se quando um usuário cria um item de leilão e adiciona um valor mínimo de lance. Assim que o leilão é iniciado, os usuários da blockchain podem fazer lances através de uma transação que verifica se o saldo da carteira de quem está registrando o lance é maior ou igual ao valor ofertado. Em caso positivo, também verifica se o leilão está aberto, para então criar um novo lance e adicionar aos lances recebidos para o objeto do leilão. Caso o leilão esteja fechado, este lance não é registrado. Quando este contrato finaliza, um método de devolução de dinheiro é executado para os lances que não obtiveram êxito. Sendo assim, os licitantes podem então retirar seu dinheiro. Para fins destes experimentos, esse contrato é inicializado com 100.000 licitantes. O conflito ocorre sempre que dois ou mais licitantes ofertam um lance superior ao maior já registrado, pois neste caso acessam o mesmo item de dado compartilhado.

Eleição: Um contrato de eleição é bastante complexo, principalmente em aspectos relacionados em como atribuir os direitos e evitar fraudes, como voto duplo ou votos com pesos distintos. Na inicialização desta aplicação, todos os eleitores e as propostas são inicializadas (nestes experimentos, 100.000 eleitores e 100.000 propostas foram criadas) e um período de tempo é definido para o término da eleição. Em seguida, o criador do contrato atua como presidente dando direito de voto a todos os eleitores. Assim, os eleitores podem lançar seus votos aos candidatos/propostas ou delegar seu direito de voto para outro eleitor. Quando a eleição terminar, um método retornará a proposta que obteve o maior número de votos. Duas transações conflitam quando dois eleitores votarem na mesma proposta, i.e., quando as duas transações são para votos na mesma proposta.

Mix: Esta aplicação é uma combinação das anteriores. As transações são criadas e distribuídas aleatoriamente, mas na mesma proporção, entre as aplicações acima descritas.

4.2. Configuração do experimento

Os experimentos foram executados em um Intel Core i9-10900K de 10 núcleos de 3.70 GHz e 16 GB de memória e o ambiente de software foi o Ubuntu 22.04.1 64 bits LTS e a máquina virtual Java 64 bits versão 11.0.10. Variamos o número de *threads* executoras e o tamanho dos blocos de cada aplicação, para avaliar a latência (tempo para executar as transações de um bloco) e a aceleração quando comparado com o modelo sequencial. Variamos o número de transações em cada bloco de 50 até 20.000 transações (tamanho superior ao praticado atualmente [Blockchain.com 2022]), e utilizamos de 1 até 8 *threads* executoras. Para avaliar o impacto dos conflitos, executamos experimentos sem conflitos e também com uma taxa de 15% de conflitos para as aplicações. Vale destacar que todos os parâmetros das transações foram definidos de forma aleatória seguindo uma distribuição uniforme. Os tempos reportados referem-se apenas a execução das transações, i.e., o tempo necessário para resolver a prova-de-trabalho empregada nestas blockchains [Nakamoto 2008] não foi considerado pois estamos particularmente interessados na execução das transações. Para estes experimentos, consideramos a abordagem que envia a ordem para os mineradores, assim tanto mineradores quanto validadores

executam o mesmo processamento e os tempos reportados referem-se tanto a mineração quanto a validação. A Seção 4.5 compara as duas abordagens propostas.

4.3. Análise de desempenho em cargas de trabalho sem conflito

No primeiro conjunto de experimentos, medimos o desempenho das aplicações sem conflitos. A execução sequencial foi usada como caso base para avaliarmos a aceleração da nossa proposta. Em todos os experimentos, a execução com apenas uma *thread* executora se assemelha a execução sequencial, ficando um pouco inferior pois existe o custo de criar e gerenciar o grafo (COS).

Transferência: Os resultados para esta aplicação são apresentados na Figura 4. Como podemos ver, o modelo paralelo apresentou uma pequena desaceleração quando comparado com a execução sequencial, visto que essa aplicação é muito simples e seu custo computacional é muito baixo. Sendo assim, o custo de criar o grafo e adicionar as transações supera o custo de executá-las diretamente.

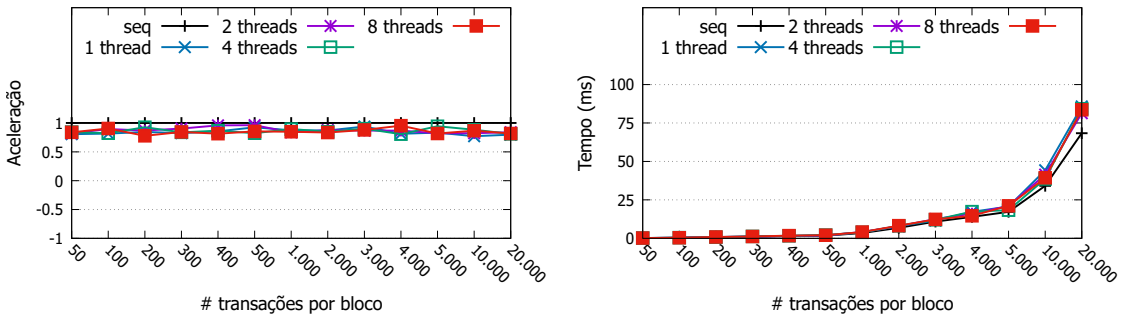

Figura 4. Transferência – sem conflitos

Eleição: A Figura 5 apresenta os resultados para esta aplicação. Podemos observar que a aceleração e o desempenho foram superior a sequencial. Com apenas duas *threads* executoras, a aceleração é duas vezes maior que a execução sequencial, e com 4 *threads* a aceleração é próxima a quatro vezes a sequencial. Com 8 *threads*, o resultado fica em torno de $5.5\times$ o medido na execução sequencial. Além disso, é possível observar uma grande melhora na latência, i.e., com 8 *threads* foram executadas 20.000 transações em menos de 500*ms*, diferentemente da execução sequencial que demorou mais de 2500*ms*.

Leilão: A aplicação de leilão (Figura 6) também teve o desempenho bastante melhorado através da execução paralela. Uma observação importante é em relação ao pequeno decremento que é observado quando analisamos a execução com uma *thread*, pois neste caso existe o custo de criação do grafo e adição das transações, sendo que apenas uma *thread* é responsável pela execução, que também será sequencial.

Mix: O resultado da execução dessa aplicação, representado na Figura 7, apresenta valores que se aproximam da média dos resultados anteriores, pois todo o conjunto de transações que os clientes enviam são aleatoriamente distribuídos entre estas aplicações (mantendo a proporção de 1/3 por aplicação). Sendo assim, em geral o desempenho é $5\times$ superior a execução sequencial.

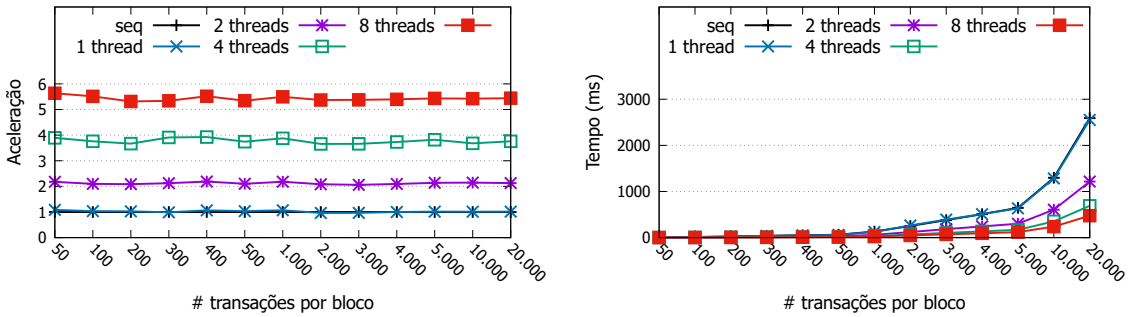

Figura 5. Eleição – sem conflitos

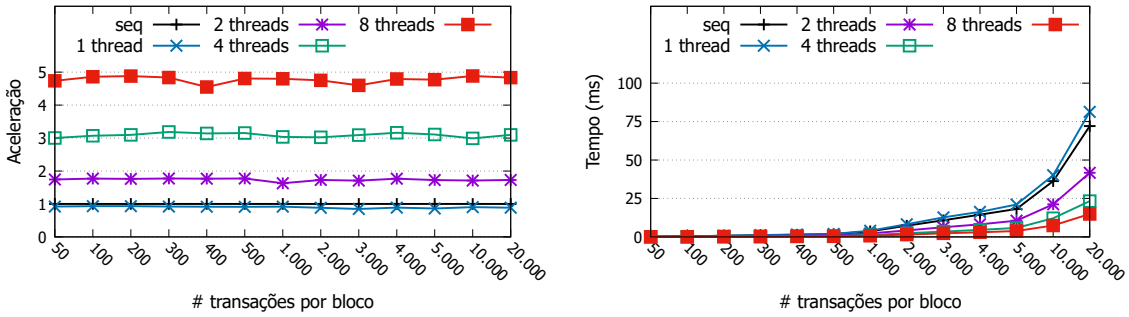

Figura 6. Leilão – sem conflitos

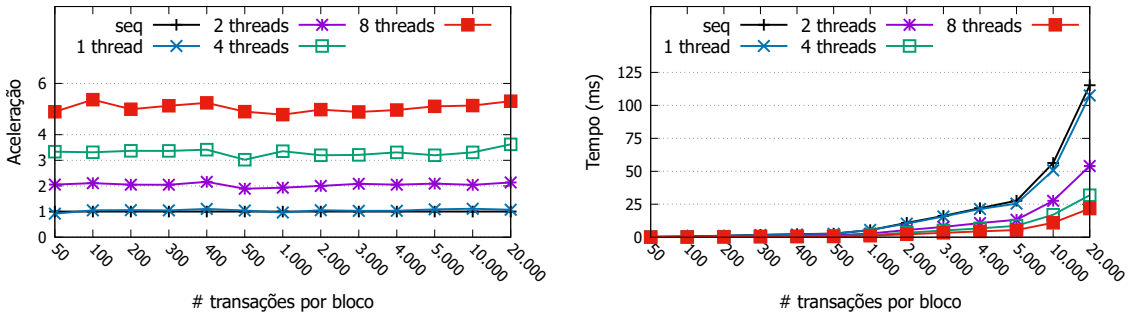

Figura 7. Mix – sem conflitos

4.4. Análise de desempenho em cargas de trabalho com conflito

No próximo conjunto de experimentos, medimos o desempenho das aplicações com 15% de conflito. Os conflitos ocorrem entre as transações da mesma aplicação que acessam dados compartilhados, conforme já explicado. Além disso, as transações com conflitos são adicionadas no grafo de forma aleatória seguindo uma distribuição uniforme.

Transferência: Os resultados reportados na Figura 8 mostram que a aplicação de transferência, por ser uma aplicação de custo muito baixo, teve um desempenho semelhante ao apresentado no caso sem conflitos. De fato, como a execução é muito rápida, o fato de conflitos diminuírem o grau de paralelismo não acarreta em prejuízos no desempenho desta aplicação.

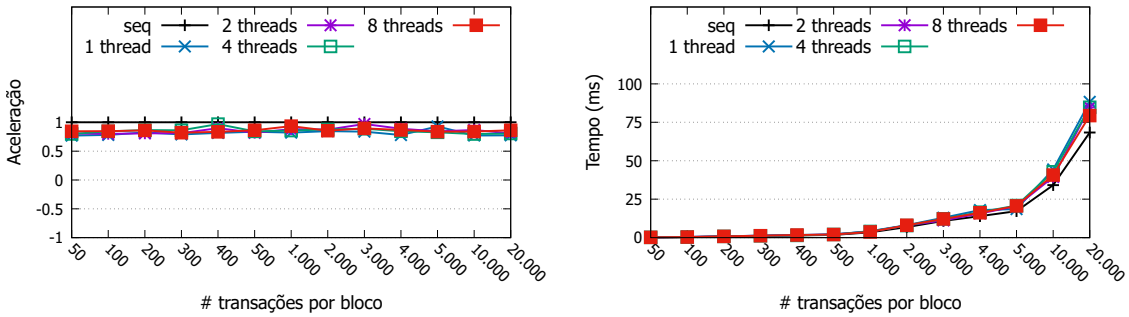

Figura 8. Transferência – 15% de conflitos

Eleição: Para esta aplicação podemos observar um decremento no desempenho quando comparado a execução sem conflitos (Figura 9). Esta aplicação usa listas para eleitores e para propostas, que são percorridas durante a execução das transações para encontrar o eleitor e para registrar o voto em alguma proposta. Desta forma, por ser uma transação com um custo de processamento maior, os impactos dos conflitos são mais notados. No entanto, ainda apresenta um desempenho superior a execução sequencial, passando de $2\times$ na configuração com 8 *threads* executoras.

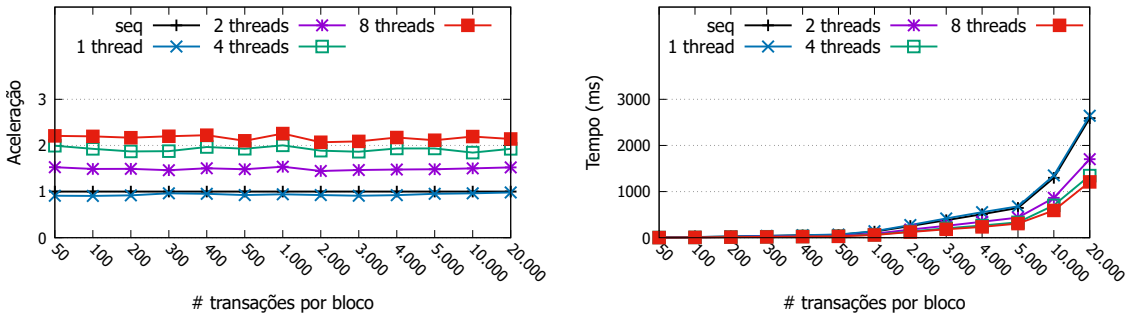

Figura 9. Eleição – 15% de conflitos

Leilão: Essa aplicação apresentou uma pequena queda de desempenho quando executada sobre uma carga com 15% de conflitos, conforme visto na Figura 10. Essa aplicação também utiliza listas que armazenam os itens de leilão e os licitantes, as quais precisam ser percorridas para execução de uma transação de lance. No entanto, por ser uma transação com custo menor do que as de eleição (o custo maior está em localizar o licitante), o impacto causado pelos conflitos é menor quando comparado com a aplicação de eleição.

Mix: O resultado da execução dessa aplicação, visualizado na Figura 11, novamente se aproxima da média dos resultados anteriores. Podemos também observar que os resultados se mantiveram similares a execução desse mesmo conjunto de aplicações para a carga de trabalho sem conflitos, com uma leve queda no desempenho.

4.5. Comparando as abordagens

A Figura 12 apresenta a comparação entre as abordagens propostas, considerando um sistema com 8 *threads* e blocos com 500 transações. A abordagem de manter e enviar o grafo apresenta pior desempenho, uma vez que o minerador precisa computar mais conflitos pois as transações executadas permanecem no grafo. O processo de validação também

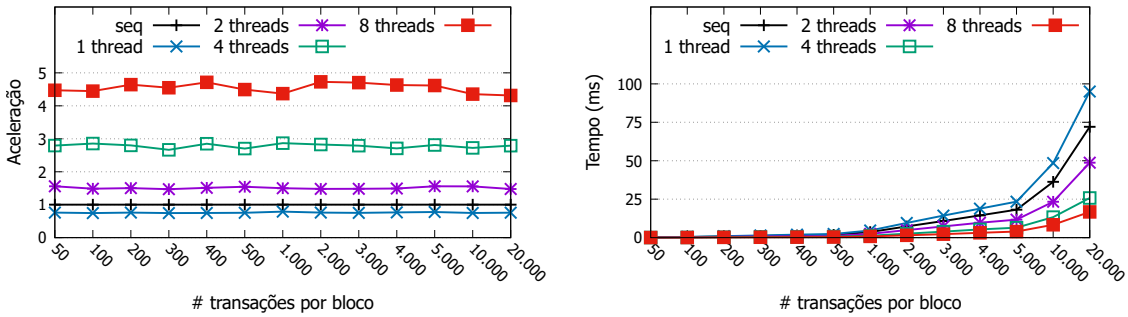

Figura 10. Leilão – 15% de conflitos

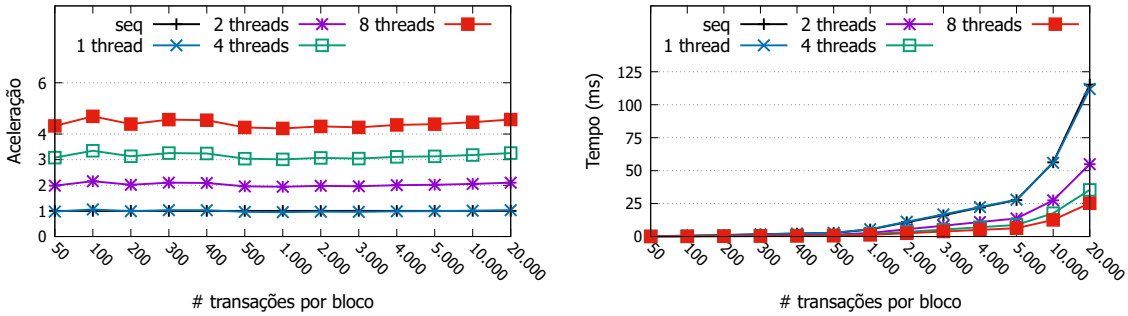

Figura 11. Mix – 15% de conflitos

é mais demorado pois na implementação *lock-free* utilizada [Escobar et al. 2019], as transações (nós do grafo) apenas são marcadas como removidas mas a remoção física ocorre em uma posterior inserção. Como não temos inserções nesta abordagem de validação, o grafo permanece inteiro e a lista completa de nós precisa ser percorrida para buscar uma transação disponível para execução. Note que este comportamento fica exacerbado na aplicação de transferência por possuir transações mais rápidas.

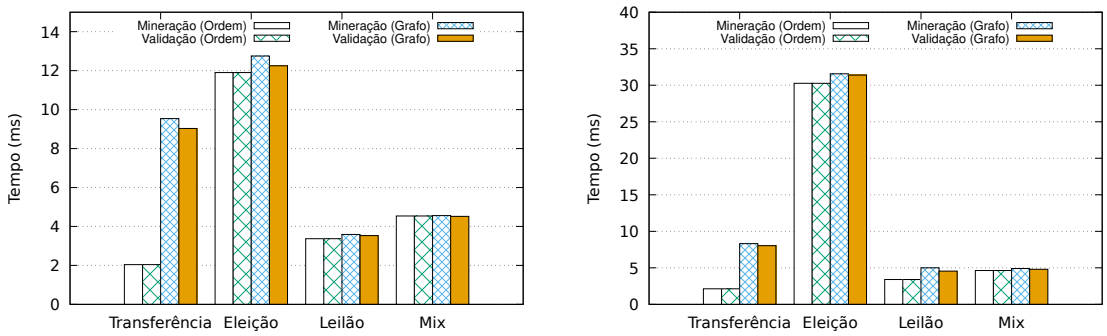

Figura 12. Sistema configurado com 8 *threads* e blocos de 500 transações. Sem conflitos (esquerda) e 15% de conflitos (direita).

5. Conclusões

Neste trabalho mostramos que explorar arquiteturas *multi-cores* e soluções que permitem a execução paralela de uma parte das transações aumenta o desempenho de aplicações em ambientes blockchain. A principal vantagem da solução proposta é que a mesma impede um minerador desonesto de atrasar o processo de validação para obter vantagens

na mineração do bloco seguinte. De fato, um validador pode refazer exatamente os mesmos processamentos do minerador ou apenas verificar se os conflitos reportados no grafo são válidos. Em ambos os casos, o grau de paralelismo fornecido pela carga de trabalho é a mesma do minerador. Adicionalmente, o custo computacional é menor caso o validador receba o grafo do minerador, visto que apenas os conflitos reportados no grafo precisam ser verificados. Caso o validador receba apenas a ordem de inserção no grafo, é necessário analisar novamente se uma transação conflita com todas as outras presentes no grafo durante uma inserção. Por fim, testamos nossa arquitetura em aplicações comuns de contratos e, em geral, obtivemos ganhos de desempenho proporcionais ao número de *threads* executoras, exceto na aplicação de transferência que é muito leve e não justifica os custos necessários para gerenciamento do grafo de conflitos.

Referências

Amiri, M. J., Agrawal, D., and El Abbadi, A. (2019). Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1337–1347. IEEE.

Anjana, P. S., Kumari, S., Peri, S., Rathor, S., and Somani, A. (2019). An efficient framework for optimistic concurrent execution of smart contracts. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*.

Baheti, S., Anjana, P. S., Peri, S., and Simmhan, Y. (2022). Dipetrans: A framework for distributed parallel execution of transactions of blocks in blockchains. *Concurrency and Computation: Practice and Experience*, 34(10):e6804.

Bartoletti, M., Galletta, L., and Murgia, M. (2020). A true concurrent model of smart contracts executions. In *International Conference on Coordination Languages and Models*, pages 243–260. Springer.

Blockchain.com (2022). Bitcoin block size. urlhttps://www.blockchain.com/explorer/blocks/btc.

Buterin, V. (2013). A next-generation smart contract and decentralized application platform https://github.com/ethereum/wiki/wiki.

Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., and Chatterjee, S. (2008). Software transactional memory: Why is it only a research toy? *Communications of the ACM*, 51(11):40–46.

Dannen, C. (2017). Solidity documentation. In *Introducing Ethereum and solidity*, pages 69–88. Springer.

Dickerson, T., Gazzillo, P., Herlihy, M., and Koskinen, E. (2017). Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 303–312.

Escobar, I. A., Alchieri, E., Dotti, F. L., and Pedone, F. (2019). Boosting concurrency in parallel state machine replication. In *Proceedings of the 20th International Middleware Conference*, pages 228–240.

Garay, J., Kiayias, A., and Leonardos, N. (2015). The bitcoin backbone protocol: Analysis and applications. In Oswald, E. and Fischlin, M., editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 281–310, Berlin, Heidelberg. Springer Berlin Heidelberg.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260.

Saraph, V. and Herlihy, M. (2019). An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376*.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319.