Simulador para medição de paralelismo em algoritmos de escalonamento para Replicação Máquina de Estados Paralela

João Gabriel Trombeta[1], **Odorico Machado Mendizabal**[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina
Caixa Postal 476 – 88040-900 – Florianópolis – SC – Brasil

joao.gabriel.trombeta@grad.ufsc.br, odorico.mendizabal@ufsc.br

Resumo. *Ao desenvolver um novo algoritmo de escalonamento de requisições para Replicação Máquina de Estados Paralela, é difícil mensurar seu grau de paralelismo sob diferentes cargas de trabalho e configurações, ou compará-lo com técnicas existentes. Neste trabalho é proposto um simulador que abstrai custos de uma implementação real, para que sejam analisados possíveis ganhos de desempenho decorrentes exclusivamente das estratégias de escalonamento.*

1. Introdução

Replicação Máquina de Estado (RME) [Lamport 1978, Schneider 1990] é uma técnica utilizada para garantir tolerância a falhas em sistemas distribuídos, ao mesmo tempo que garante consistência forte. Em RME, todas as réplicas partem do mesmo estado inicial e executam as mesmas requisições na mesma ordem, garantindo assim que todas atravessem os mesmos estados. A execução sequencial representa uma limitação no desempenho do sistema, abrindo espaço para implementações de RME Paralela [Kotla and Dahlin 2004], que visam extrair paralelismo na execução de requisições, sem comprometer a consistência. Com o surgimento dessa técnica, estratégias para balancear a execução de requisições entregues às réplicas ganham importância.

Diversas técnicas foram propostas para o escalonamento de requisições em RMEP. CBase [Kotla and Dahlin 2004] utiliza um grafo para manter controle das dependências entre as requisições, em [Mendizabal et al. 2017] mapas de bits anotam informação de dependência entre comandos, enquanto lotes de requisições são utilizados para aumentar o taxa de ocupação nas *threads* executoras. Outras técnicas separam as requisições em classes de conflito e definem o relacionamento entre as classes [Alchieri et al. 2018]. Uma das dificuldades ao desenvolver uma nova técnica, porém, é medir o potencial ganho em paralelismo que a técnica pode extrair e como compará-la com técnicas já existentes, sem que haja a necessidade de implementá-la em um sistema completo.

Nesse trabalho é descrito um simulador de execução de requisições em RMEP, para analisar o paralelismo obtido por algoritmos de escalonamento, abstraindo demais custos de uma implementação real. Durante a simulação, requisições são geradas e enviadas para o algoritmo de escalonamento, que registra dados da simulação para posterior análise. As etapas da execução são modularizadas de forma que é possível editar aspectos pontuais da simulação para representar diferentes ambientes e comportamentos. Através dos dados levantados, é possível medir o potencial paralelismo obtido por políticas de escalonamento, observar como diferentes parâmetros afetam o desempenho, e comparar técnicas, antes de implementá-las em um sistema real.

2. Funcionamento do simulador

O uso do simulador passa pelas configurações iniciais, geração de requisições, execução e saída de dados. A Figura 1 mostra a arquitetura do simulador.

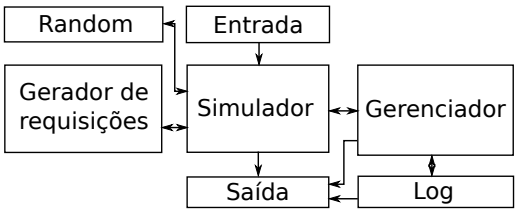


Figura 1. Arquitetura do simulador

As configurações do simulador, como número de chaves no sistema, quantidade e distribuição de requisições, são especificadas em um arquivo de entrada. É no arquivo de configuração que são definidos comportamentos que podem variar entre diferentes simulações, visando a análise da execução sob diferentes parâmetros.

Requisições podem ser geradas de maneira aleatória ou importadas de um arquivo externo. Existem dois tipos de requisições, requisições que acessam uma única chave, e requisições que acessam múltiplas chaves. Caso sejam importadas de um arquivo externo, o caminho de um arquivo definindo uma lista de requisições é esperado. Se for necessário gerar requisições aleatórias, o simulador chama o módulo `Random`, como demonstrado na Figura 1, para buscar geradores de números pseudo-aleatórios que satisfaçam a distribuição desejada.

Ao optar por gerar requisições aleatórias, em requisições de chave única é possível especificar a quantidade de requisições e a distribuição a ser utilizada durante a geração. Em requisições que acessam múltiplas chaves, é preciso definir a quantidade de requisições, o número mínimo e máximo de chaves envolvidas, e a distribuição utilizada para escolher o tamanho e as chaves envolvidas nas requisições. Atualmente estão implementadas as distribuições fixa, uniforme e binomial.

Durante a execução da simulação é assumido que todas as requisições tenham o mesmo tempo de execução, definido como uma unidade de tempo. Seguindo a arquitetura demonstrada na Figura 1, o Simulador interage com o módulo `Gerenciador`, que deve ser implementado estendendo uma classe de mesmo nome. É preciso que a nova classe implemente o método `execute_requests`, que deve realizar o escalonamento das execuções. Durante o escalonamento, é possível interagir com o `Log` para administrar o estado de cada $thread$ simulada.

Após a execução, o simulador escreve em um arquivo informações sobre as $threads$ simuladas. São apresentados o instante de tempo em que as *threads* terminaram de executar a última requisição, a quantidade de tempo em que estiveram ociosas e a porcentagem total que esse tempo representa. Essas informações são obtidas através do `Log`, utilizado durante o escalonamento. Além das informações coletadas pelo `Log`, o `Gerenciador` pode, através do método `export_data`, escrever dados adicionais. As requisições utilizadas durante a execução podem ser exportadas e reutilizadas em simulações futuras.

3. Análise de uso: comparando técnicas

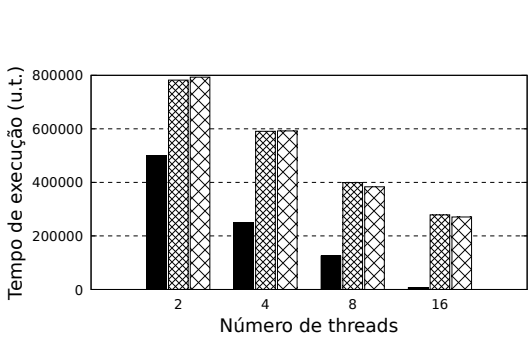
Para mostrar o simulador em funcionamento, foram implementadas variações do `Gerenciador` para duas técnicas de escalonamento: CBase e corte em grafo. Dada a estrutura modular da ferramenta, outras estratégias de escalonamento podem ser facilmente incorporadas ao simulador futuramente.

O CBase [Kotla and Dahlin 2004] cria um grafo de dependências, em que vértices são requisições e existe uma aresta (x, y) se, e somente se, a execução de x antecede y. $Threads$ acessam o grafo e executam as requisições representadas por vértices fonte, e depois disso atualizam o grafo, removendo o vértice. Esse algoritmo é considerado ótimo em exploração de paralelismo, porém implementações reais apresentam um grande custo devido ao acesso concorrente ao grafo e à detecção de conflitos [Mendizabal et al. 2017].

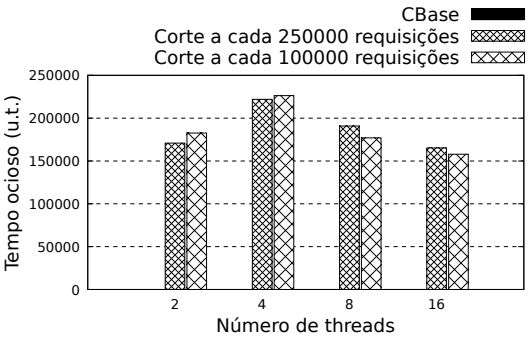
O corte mínimo em grafos é uma técnica de balanceamento de carga (e.g. [Hendrickson and Kolda 2000]). O grafo aqui é modelado de forma que os vértices são chaves, seu peso é a quantidade de vezes que foram acessadas, e uma aresta (a, b, x) diz que as chaves a e b foram acessados por uma mesma requisição x vezes, onde $x > 0$. O corte resulta em conjuntos disjuntos de chaves, chamados de partições, cada uma atribuída a uma $thread$. A execução de requisições é feita em paralelo por todas as $threads$, cada uma executando as requisições nas chaves em suas partições. Caso uma requisição acesse chaves que estão em partições diferentes, as $threads$ que possuem as chaves acessadas sincronizam a execução do comando, onde apenas uma executa o comando, e as demais ficam bloqueadas. O número de partições é definido como o mesmo número de *threads*, e inicialmente as chaves são distribuídas pelas partições utilizando *Round-Robin*. A biblioteca METIS [Karypis and Kumar 1998] foi utilizada para realizar o corte mínimo.

As simulações foram configuradas com 1.000 chaves e 1.000.000 de requisições, sendo que 75% das requisições acessam uma única chave e as demais acessam múltiplas chaves. Para gerar requisições de um único valor, foi utilizada a distribuição binomial na escolha da chave; para requisições que acessam múltiplas chaves, o número de chaves pode variar de 2 a 8, de acordo com uma distribuição binomial, e a escolha das chaves segue uma distribuição uniforme. Duas situações do corte em grafos foram exploradas, uma com reparticionamento a cada 250.000 requisições, e outra com reparticionamento a cada 100.000 requisições. Foram simulados cenários com 2, 4, 8 e 16 *threads*.

A Figura 2(a) mostra o gráfico do tempo necessário para a execução das requisições, enquanto a Figura 2(b) exibe o tempo médio em que as *threads* ficam ociosas, indicados em unidades de tempo (u.t.). A ociosidade é computada pela *thread* a cada instante em que aguarda pela sincronização com outra(s) *thread(s)*. Apesar do alto custo de implementação, o CBase é ideal em termos de exploração do paralelismo, o que causa um baixo tempo de execução e nenhuma ociosidade. Nos escalonamentos utilizando corte mínimo, o tempo em que as *threads* permanecem ociosas é maior, devido à necessidade de sincronizações. Isso reflete no tempo total de execução, maior do que o obtido com o CBase. É possível perceber como a frequência de reparticionamento pouco afetou o tempo de execução ou a ociosidade, e que uma frequência maior pode, por vezes, deteriorar o desempenho. O simulador tem como objetivo levantar informações sobre número de sincronizações, ociosidade das *threads* e tempo de execução total de maneira simples, para que sejam considerados durante a concepção de novos algoritmos ou em comparações entre diferentes técnicas e configurações.



(a) Tempo total de execução.



(b) Média do tempo ocioso entre as *threads*.

Figura 2. Gráficos obtidos a partir das informações de saída do simulador.

4. Considerações finais

Com a introdução de Replicação Máquina de Estados Paralela, é necessário que sejam exploradas técnicas de escalonamento de requisições que consigam extrair melhor paralelismo do modelo. É apresentando, então, um simulador que abstrai custos de uma implementação real, para que durante o concebimento de um novo algoritmo seja possível analisar o potencial paralelismo da técnica sob diferentes configurações. Pode-se, também, comparar algoritmos existentes, como um instrumento de análise.

Agradecimentos

O presente trabalho foi realizado com o apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq - Brasil.

Referências

Alchieri, E., Dotti, F., Marandi, P., Mendizabal, O., and Pedone, F. (2018). Boosting state machine replication with concurrent execution. In *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*.

Hendrickson, B. and Kolda, T. G. (2000). Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534.

Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392.

Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*.

Mendizabal, O. M., De Moura, R. S., Dotti, F. L., and Pedone, F. (2017). Efficient and deterministic scheduling for parallel state machine replication. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

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