UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

TIAGO DE ALMEIDA

Sincronização *lock-free* escalável para uma *Crit-bit Tree*, com elementos de memória transacional

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Sérgio Luis Cechin

Porto Alegre
2014

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do Curso de Engenharia de Computação: Prof. Marcelo Götz
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço ao meu professor orientador Dr. Sérgio Luis Cechin pelos produtivos debates e fundamentais conselhos ao longo do desenvolvimento deste trabalho, assim como ao seu compromisso em exercer com excelência sua tarefa de docência. Ainda, agradeço aos amigos Ricardo Sanchez e Vinicios Barros pelas inspiradoras discussões técnicas e valiosas contribuições para este trabalho.

Agradeço pela paciência e suporte diários da minha namorada, Ariella Yung Soares, durante a fase de desenvolvimento deste trabalho. Ainda, agradeço a sua parceria e companheirismo presentes em cada dia da minha vida.

Finalmente, agradeço pelo incessante apoio recebido da minha mãe, Rosane Schaefer, meu pai Ubirajara de Almeida, e meu irmão, Diogo de Almeida. Obrigado por proverem os sólidos fundamentos nos quais pude me apoiar e sempre acreditarem nos meus objetivos.

RESUMO

Arquiteturas de processamento paralelo têm se apresentado como a principal proposta de atendimento da crescente demanda por poder de processamento. Ao multiplicar o número de unidades de processamento, tais arquiteturas prometem, em teoria, multiplicar a taxa de execução de instruções. Na prática, contudo, este comportamento é difícil de ser alcançado. Múltiplos desafios impedem que desenvolvedores consigam extrair o máximo potencial das arquiteturas paralelas. A fim de auxiliar no melhor aproveitamento deste potencial este trabalho propõe, desenvolve e analisa uma nova estratégia de sincronização elaborada sobre uma estrutura de dados em formato de árvore binária. Tal estratégia permite a execução paralela das operações de inserção, busca e remoção na estrutura. Além disso, a estratégia foi desenvolvida de maneira a apresentar baixa concorrência entre as operações e, consequentemente, alcançar um alto grau de paralelismo. Os testes desenvolvidos sobre a implementação da estrutura apresentam medidas de concorrência, desempenho e escalabilidade. Os dados obtidos permitem avaliar o benefício do uso da estratégia de sincronização proposta em um ambiente multiprocessado.

Palavras-chave: Multiprocessamento. *Multithread*. Sincronização. Estrutura de Dados.

Transactional memory inspired *lock-free* synchronization for a *Crit-bit Tree*

ABSTRACT

Parallel processing architectures have been shown as the major proposed solution for the increasing demand for processing power. By multiplying the number of processing units such architectures promise, in theory, to multiply the rate of instruction execution. In practice, however, this behavior is not reached easily. Multiple challenges prevent developers from extracting the full potential of parallel architectures. In order to help reaching this potential this paper proposes, develop and analyses a new synchronization strategy developed over a binary tree data structure. This strategy allows parallel insertion, search and removing from the structure. Furthermore, the strategy was developed in a way to avoid concurrency among operations and, consequently, reach a high level of parallelism. The tests executed over the data structure present measures on concurrency, performance and scalability. The data obtained allow us to evaluate the advantage of using the proposed synchronization strategy over a multiprocessed environment.

Keywords: Multiprocessing. Multithread. Synchronization. Data Structure.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A execução concorrente de tarefas se mostrou uma necessidade desde o desenvolvimento dos primeiros sistemas operacionais. Mesmo rodando em apenas uma unidade de execução, o sistema deveria gerenciar diversas tarefas e, quando possível, passar ao usuário a impressão de que estas eram executadas ao mesmo tempo. Tanto tarefas do usuário quanto tarefas do próprio sistema, entre operações de cálculos e chamadas de sistema, deveriam executar de maneira minimamente intrusiva e garantir que a execução do conjunto de tarefas, como um todo, progredisse.

Para permitir a execução concorrente de várias tarefas em apenas um processador soluções de escalonamento de tarefas foram implementadas. Dessa maneira, cada tarefa executaria por um determinado tempo e então liberaria o processador para as próximas tarefas, até que recebesse sua próxima fatia de tempo para execução. Estas trocas aconteceriam tão rapidamente que passariam a impressão de que as tarefas estariam executando ao mesmo tempo.

Estratégias de escalonamento preocupam-se, entre outras, com questões de corretude e equidade. Corretude procura garantir que os dados de uma tarefa não sejam corrompidos durante o período em que esta não esteja executando. Estes dados podem ser corrompidos, por exemplo, por outra tarefa que também os usa em sua execução. Equidade, por sua vez, preocupa-se com o tempo de execução oferecido para cada tarefa, em relação ao conjunto total de tarefas. Tarefas relacionadas a questões de tempo real, como a tarefa que controla o movimento do cursor do *mouse* na tela, por exemplo, devem obter tempo de execução suficiente para satisfazer tais requisitos. Por outro lado, o problema de *starvation*, quando uma tarefa nunca obtém a sua fatia de tempo de execução, deve ser evitado.

Ao longo da evolução dos sistemas computacionais, soluções com mais de uma unidade de execução foram propostas e implementadas. Em busca de maior desempenho, supercomputadores com diversos processadores foram construídos e, posteriormente, diversos computadores foram agregados para a construção de *clusters*. Mais recentemente, quando a indústria observou a dificuldade em dar continuidade ao aumento da frequência dos processadores, outras soluções para o aumento de desempenho foram buscadas. Entre elas, passou-se a inserir diversos núcleos dentro de um mesmo *chip*, oferecendo então diversas unidades de execução em um mesmo processador. Em níveis diferentes mas semelhantes, os supercomputadores, os *clusters* e os processadores com vários núcleos oferecem um ambiente

de processamento paralelo, onde as tarefas não apenas concorrem por uma unidade de execução, mas executam de fato simultaneamente em diferentes processadores ou núcleos.

A computação paralela trouxe consigo diversos desafios: se por um lado desenvolvedores têm a sua disposição um poder de processamento somado muito maior, por outro não conseguem facilmente tirar proveito deste cenário. Dependência de dados e sincronização de tarefas evitam que programas consigam manter as diversas unidades de execução ocupadas ao mesmo tempo. Como consequência, o poder de processamento somado de todas as unidades é frequentemente subutilizado.

Diversas abordagens são utilizadas a fim de gerenciar o uso de múltiplas unidades de processamento. Em sistemas distribuídos em diversos computadores a troca de mensagens pela rede procura sincronizar as execuções de maneira a garantir a corretude do sistema como um todo. Em computadores com múltiplos processadores ou processadores com múltiplos núcleos, a troca de mensagens entre processos também é utilizada, porém menos comum. Neste cenário, onde frequentemente a memória é compartilhada entre tarefas, APIs de sincronização gerenciam a execução de tarefas através de semáforos, *locks*, barreiras de memória e instruções atômicas.

O campo de estudo deste trabalho restringe-se a ambientes multiprocessados de memória compartilhada. Estratégias de sincronização para tais ambientes costumam oferecer uma grande dificuldade de programação, além de tendência a erros difíceis de serem encontrados. Algoritmos e estruturas de dados devem ser bem projetados a fim de escalarem a medida que novas unidades de execução sejam inseridas. Um algoritmo projetado sem esta ideia pode encontrar na sincronização um gargalo de execução e, consequentemente, não oferecer aumento de desempenho em ambientes multiprocessados.

O presente trabalho traz a proposta de uma nova estratégia de sincronização elaborada pelo autor para uma estrutura de dados. Esta estratégia não apenas garante a corretude das operações executadas simultaneamente na estrutura, mas objetiva alcançar alto grau de paralelismo e desempenho. Experimentos foram executados sobre a implementação da proposta a fim de obter medidas sobre o seu comportamento. A estratégia de sincronização é detalhada nos próximos capítulos, assim como os experimentos executados e as medidas obtidas.

2 MOTIVAÇÃO E OBJETIVO

Ambientes de execução paralela oferecem um enorme potencial para computação de alto desempenho. Fazer uma aplicação tirar proveito de todo este potencial, por outro lado, não é simples. A tradicional abordagem de execução sequencial e determinística de código oferece ao desenvolvedor maior controle sobre o comportamento de sua aplicação. Esta abordagem, contudo, não consegue tirar proveito de ambientes multiprocessados. Para tal, o desenvolvedor precisa projetar a sua aplicação de forma que esta seja um conjunto de diversas tarefas que executam em paralelo. Como consequência, maior complexidade e comportamento não determinístico são adicionados à sua aplicação.

Com o objetivo de aumentar o controle e garantir a corretude e coerência da execução, soluções de sincronização são utilizadas. Estas estabelecem ordem onde tarefas não podem executar de forma paralela. Estratégias de sincronização não são um benefício, e sim uma necessidade. O procedimento de organizar a execução naturalmente paralela e não determinística de diversas tarefas torna o progresso da execução como um todo mais lenta. Com estratégias de sincronização mal elaboradas, um programa paralelo pode não encontrar benefícios na execução em ambientes multiprocessados.

Tendo em vista o cenário desafiador de desenvolvimento de programas paralelos, este trabalho tem por objetivo a implementação de uma biblioteca de auxílio ao desenvolvedor. Esta biblioteca contém uma estrutura de dados no formato de árvore binária preparada para ambientes de execução paralela. A estrutura foi desenvolvida com o objetivo de se comportar de forma escalável, baseando-se em um alto grau de paralelismo em suas operações. Para tal, uma estratégia de sincronização foi elaborada e implementada pelo autor.

O trabalho apresentado neste documento é uma expansão sobre um projeto desenvolvido pelo autor na empresa Taghos Tecnologia. A empresa desenvolve uma solução de *cache* de conteúdo *web* para provedores de internet, solução esta instalada em um servidor multiprocessado. Este servidor, rodando o sistema de *cache* desenvolvido pela empresa, é instalado em meio a rede do provedor e oferece os benefícios do conceito de *cache*: acesso mais rápido aos dados e economia no tráfego de busca destes dados.

O sistema de *cache* em questão monitora o tráfego de milhares de clientes. Deste tráfego, milhões de objetos (imagens, vídeos, conteúdo estático em geral…) são inseridos no *cache* para posterior consulta. A arquitetura do *software* tira proveito das múltiplas unidades de processamento disponíveis em *hardware* em diversos níveis da aplicação, inclusive na inserção, busca e remoção de objetos do *cache*. Para um sistema que objetiva aumentar a

eficiência de entrega de conteúdo, o grande volume de objetos facilmente se torna um gargalo e prejudica o seu funcionamento. Analisando este cenário uma versão preliminar deste projeto foi desenvolvida pelo aluno na empresa a fim de fornecer uma estrutura de dados que permitisse o armazenamento e consulta de milhões de objetos de forma paralela, eficiente e escalável. A solução foi implantada em 2011 e continua em uso até hoje.

A estrutura de dados em questão foi desenvolvida na linguagem C, em ambiente Linux, e faz uso da biblioteca de *threads* Pthreads. Esta biblioteca permite a criação das *threads*, assim como disponibiliza ferramentas de sincronização utilizadas na solução proposta. Ainda, os testes de avaliação foram executados no sistema operacional Linux Mint, Kernel versão 3.11, em um computador com um processador Intel Core i7 com 4 núcleos, *hyperthreading*, e 2,3 GHz de frequência, além de 8GB de memória RAM DDR3 1600 MHz.

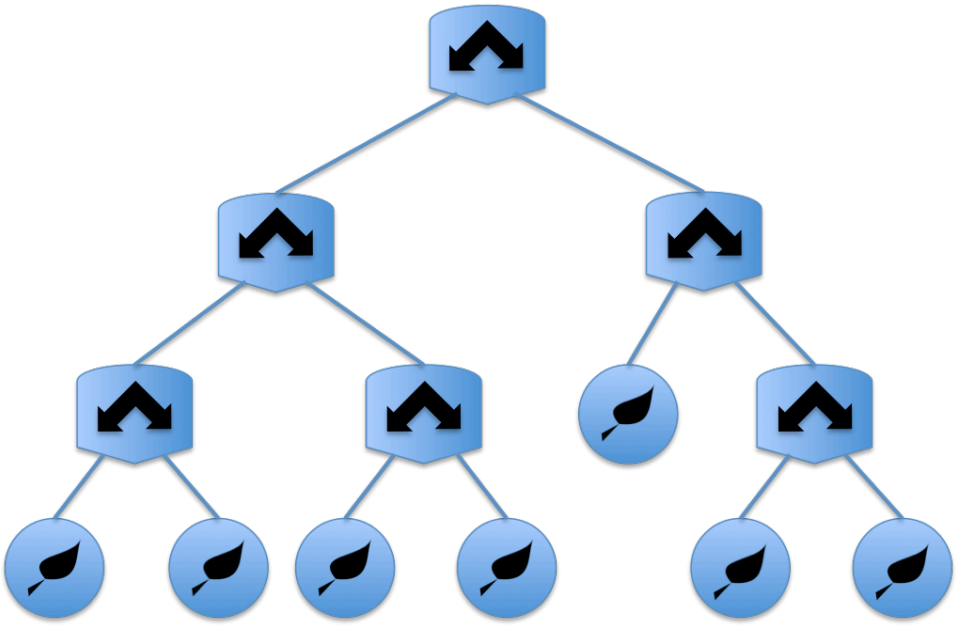
3 ESTRUTURA DE DADOS

Estruturas de dados organizam dados na memória de forma que estes sejam facilmente recuperados. Em ambientes paralelos de memória compartilhada a maioria destas estruturas podem ser lidas, mas não modificadas paralelamente por diferentes *threads*. Para tornar a modificação paralela possível e eficiente, um projeto adequado da estrutura em questão e de uma estratégia de sincronização é fundamental.

A natureza distribuída da maioria das estruturas de dados oferece alto potencial de aumento de desempenho para processamento *multithread*, permitindo que *threads* interajam paralelamente com partes diferentes da estrutura e, consequentemente, em áreas não conflitantes de memória. Por outro lado, *threads* podem também concorrer na interação com a mesma área de memória. Estratégias de sincronização são, portanto, fundamentais para garantir a corretude do funcionamento de estruturas de dados *thread-safe* em ambientes de memória compartilhada.

Para este trabalho a estrutura de dados utilizada foi a *Crit-bit Tree*, também chamada de Radix Tree ou Patricia Tree (Knizhnik, 2008). Esta estrutura, em formato de árvore binária, armazena os dados apenas nos seus nodos folha. Os nodos intermediários, por sua vez, guardam as informações de roteamento necessárias para encontrar o nodo folha procurado. Esta organização torna a árvore uma estrutura dispersa e permite a sua manipulação paralela por diversas *threads*, desde que em regiões diferentes da árvore (Figura 3.1).

Figura 3.1: *Crit-bit Tree*



A *Crit-bit Tree* oferece uma abordagem econômica em termos de memória ocupada, onde cada nodo com apenas um filho é unido ao seu nodo pai. Buscas acontecem sem modificações na estrutura da árvore. Inserções adicionam, além do nodo folha que armazena os dados de interesse, também um nodo auxiliar. Remoções, por sua vez, removem um nodo folha e também um nodo auxiliar. Além disso, a chave de cada objeto inserido não é avaliada como um todo a cada pulo na árvore, e sim pedaço a pedaço. As informações de roteamento definem a direção avaliando cada *bit* da chave. Dessa forma, cada *byte* é avaliado de cada vez até encontrar o primeiro *bit* que diferencia uma chave da outra. Esta estratégia torna o caminhamento pela árvore menos custoso.

A árvore se expande e contrai de acordo com inserções e remoções, e não necessita de balanceamento. Esta característica, juntamente com o seu perfil de armazenamento disperso dos objetos, é o que permite a implementação de um esquema de sincronização de granularidade fina e alto grau de paralelismo.

4 ESTRATÉGIA DE SINCRONIZAÇÃO

O desempenho e, principalmente, a escalabilidade de uma estrutura de dados é fortemente dependente da estratégia de sincronização utilizada. Um *lock* global em uma árvore binária, por exemplo, pode se tornar um gargalo, evitando que *threads* que modificam áreas de memória não conflitantes executem em paralelo. Uma abordagem de maior granularidade, dividindo a estrutura em regiões ou nodos com *locks* independentes pode oferecer um maior nível de escalabilidade. Por outro lado, esta abordagem de granularidade fina pode oferecer maior complexidade de controle, assim como maior consumo de memória.

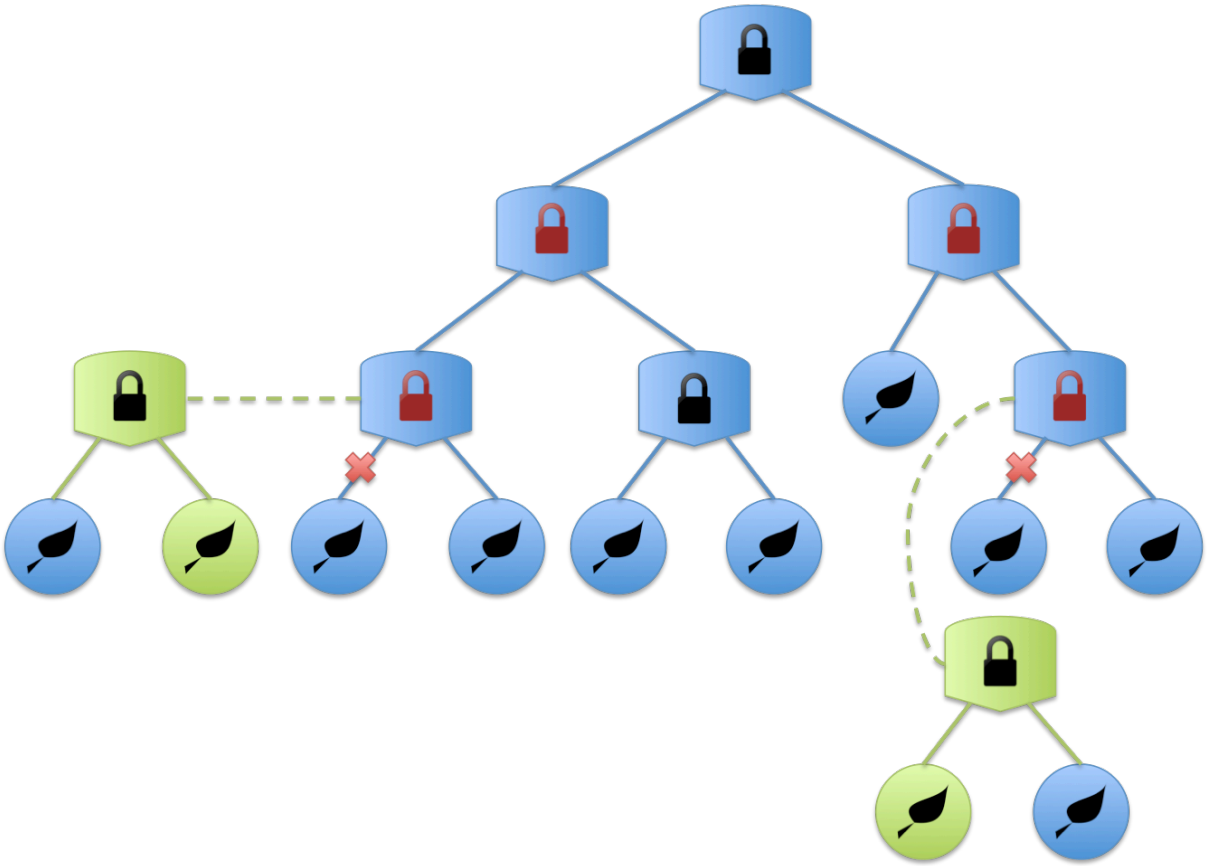
A abordagem de sincronização desenvolvida para este trabalho tem por objetivo alcançar baixos níveis de concorrência e, por consequência, altos níveis de paralelismo. Para tal, três principais estratégias foram adotadas e são explicadas nas seções 4.1, 4.2 e 4.3.

4.1 Granularidade de *lock*

As operações de inserção e remoção necessitam de exclusão mútua. Ou seja, duas ou mais operações que modificam a árvore não podem ocorrer paralelamente em regiões conflitantes. Para garantir este comportamento cada nodo intermediário possui uma variável de *lock* que é utilizada quando alguma modificação envolvendo este nodo em questão acontece.

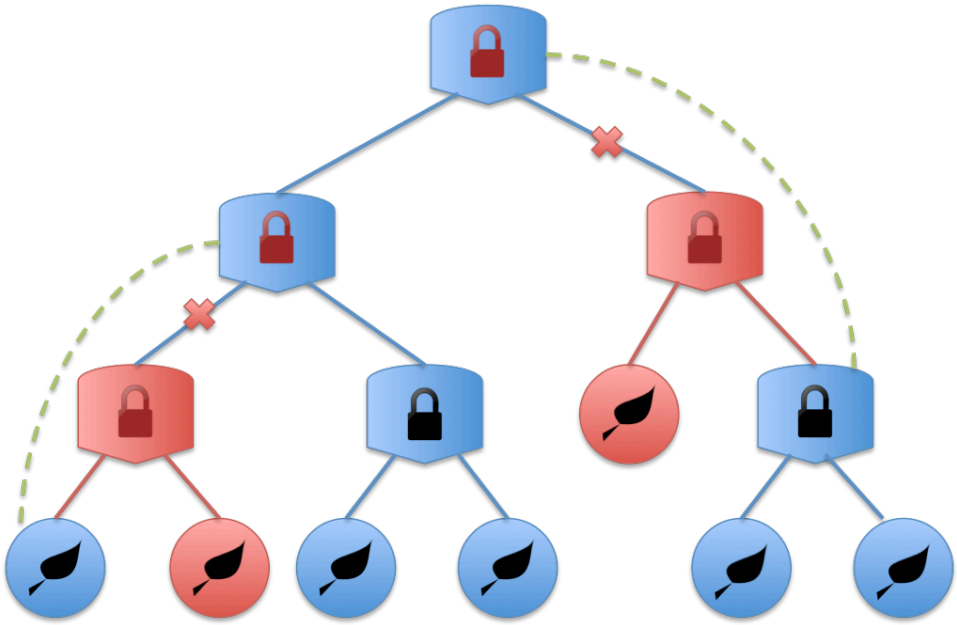
A figura 4.1 demonstra duas operações de inserção ocorrendo simultaneamente na árvore. Neste cenário, após a localização da posição de inserção, os *locks* dos dois nodos intermediários anteriores a posição são obtidos (cadeados vermelhos). A obtenção destes *locks* garante que apenas a *thread* em questão tenha permissão de modificar os *links* destes nodos com seus nodos filhos. Esta, portanto, é a região “trancada” pela exclusão mútua, onde operações de modificação da árvore não ocorrem em paralelo. Após a obtenção dos *locks* o *link* indicado por um “X” vermelho é substituído pelo novo *link*, indicado pela linha verde pontilhada. Este novo *link* aponta para o novo nodo auxiliar alocado. Este novo nodo, por sua vez, aponta para o novo nodo folha inserido e para o nodo folha que já ocupava aquele posicionamento da árvore anteriormente.

Figura 4.1: Inserções simultâneas na *Crit-bit Tree*



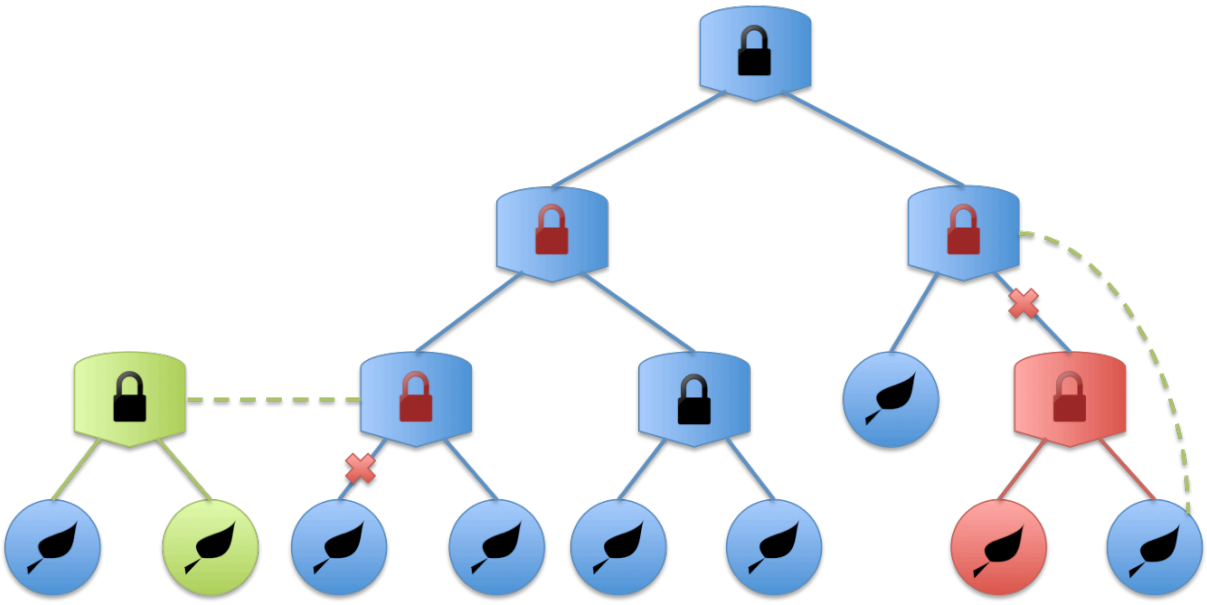
Analisando ainda mais a figura 4.1 é possível observar que os *links* do nodo mais superior não são modificados na operação de inserção. O *lock* deste nodo ainda assim é obtido, porém, para garantir a exclusão mútua das operações de inserção com as operações de remoção. A figura 4.2 apresenta duas operações de remoção acontecendo simultaneamente. Nestas, da mesma maneira, os *locks* dos dois nodos intermediários anteriores à posição de interesse são obtidos. Como é possível observar na figura apenas uma substituição de *link* é suficiente para excluir da árvore o nodo folha desejado, assim como o nodo intermediário imediatamente anterior a este. O nodo intermediário azul com cadeado vermelho apresentado na figura então passa a apontar diretamente para o nodo folha remanescente.

Figura 4.2: Remoções simultâneas na *Crit-bit Tree*



A figura 4.3, ainda, apresenta uma inserção e uma remoção ocorrendo simultaneamente. É possível observar que uma operação de modificação da árvore impede que outra operação similar aconteça paralelamente em, no máximo, 4 nodos folha. Quaisquer duas ou mais operações que aconteçam em regiões não conflitantes da árvore não necessitam de exclusão mútua e, portanto, podem acontecer simultaneamente. Esta análise demonstra que a estratégia de sincronização elaborada é de granularidade bastante fina e, portanto, permite um alto grau de paralelismo nas operações executadas na árvore.

Figura 4.3: Inserção e remoção simultâneas na *Crit-bit Tree*



4.2 *Wait-free* e *Lock-free*

Qualquer operação na árvore possui inerentemente uma busca pelo nodo folha da posição procurada: inserções conhecem a chave do novo objeto a ser inserido, mas precisam buscar a posição correta para tal. Buscas também precisam caminhar pela árvore para encontrar o objeto a ser retornado, enquanto remoções fazem o mesmo caminhamento, executando a modificação desejada na árvore quando o objeto é encontrado.

Para garantir o baixo nível de concorrência e a maior eficiência destas operações, todo e qualquer caminhamento na árvore é feito sem a obtenção de *locks*. Por consequência, a operação de busca, que nunca modifica a estrutura da árvore, nunca obtém *locks*. Esta estratégia resulta num comportamento *wait-free* das operações de busca (definição de sincronização *wait-free*: anexo A, seção 2.1) (Vyukov).

Operações de inserção e remoção também executam o caminhamento pela estrutura sem a interferência de *locks*. Contudo, quando a posição alvo de modificação é encontrada, os respectivos *locks* são tentativamente adquiridos. Neste ponto, a execução da *thread* pode ser interrompida apenas por outra *thread* que esteja realizando modificações naquela região e que, portanto, tenha adquirido pelo menos um dos *locks* de interesse da primeira *thread*. Esta estratégia resulta num comportamento *lock-free* das operações de inserção de remoção (definição de sincronização *lock-free*: anexo A, seção 2.1) (Vyukov).

4.3 Retentativas

A estratégia de caminhamento livre pela árvore até o encontro da posição de interesse garante maior velocidade de execução e menor concorrência das operações. Por outro lado, o seguinte cenário de incoerência pode ocorrer: Duas *threads* tem interesse em realizar operações de inserção ou remoção na mesma região da árvore. Ambas encontram a posição de interesse antes da obtenção de qualquer *lock*. Ao tentar obter os *locks* necessários, apenas uma delas tem sucesso e realiza a modificação na árvore. Assim que esta libera os *locks*, a próxima *thread* encontra a região da árvore diferente do que havia encontrado antes de obter os *locks*. A posição onde esta *thread* pretendia inserir ou remover um objeto pode não estar mais coerente.

Uma solução para tal cenário foi buscada nas estratégias de sincronização por memória transacional (anexo A, seção 3.4) (Herlihy ,1993). Nestas arquiteturas duas operações possivelmente concorrentes, chamadas de transações, executam livremente até o

final. Neste ponto, o sistema verifica se houve de fato concorrência na execução. Em caso positivo, apenas uma das transações é efetivada, enquanto a outra retorna para uma nova tentativa. Em caso negativo, as duas transações são efetivadas.

No nosso cenário a segunda *thread* encontra uma modificação na região de interesse: algum objeto foi inserido ou removido. A *thread* então verifica se os nodos de seu interesse continuam coerentes: *links* entre nodos não modificados. Em caso negativo, a *thread* retorna para o início da árvore e refaz a busca *wait-free* pela posição atualizada para a sua operação. Em caso positivo, a *thread* executa a sua operação sem retentativa.

Esta solução tira proveito da eficiência do caminhamento livre pela árvore para efetuar retentativas quando estas forem necessárias. Ainda, o taxa de retentativas realizadas por cada *thread* pode ser monitorada como um parâmetro de nível de concorrência na estrutura.

5 FERRAMENTAS DE MEDIÇÃO

A fim de avaliar o desempenho da estrutura de dados e da estratégia de sincronização proposta um *framework* de coleta e análise de dados chamado *Performance Counters for Linux* (PCL) foi utilizado. Comumente chamado de *perf*, *perf tools*, ou *Perf Events*, esta ferramenta está presente no Kernel do Linux desde a sua versão 2.6.31.

Por ser implementado no Kernel do Linux, e por acessar contadores e monitores implementados em *hardware*, *perf* tem um comportamento não intrusivo e, consequentemente, permite a obtenção de medidas reais da execução do programa. (“PERF: Linux profiling with performance counters”) (“PERFORMANCE Counters for Linux (PCL) Tools and perf”)

A ferramenta nos deu acesso à medidas de tempo de execução, ciclos de CPU, instruções executadas, trocas de contexto e migrações de CPU. Além disso, a ferramenta facilitou a execução repetida de testes e cálculo de médias das medidas citadas.

6 TESTES E RESULTADOS

Dois principais cenários de testes foram implementados. O primeiro é um cenário com carga controlada que tem por objetivo forçar a execução paralela da mesma operação em diferentes números de *threads*. Neste cenário uma fase pesada de inserções é executada, seguida por uma fase de buscas e posteriormente de remoções. Para algumas medidas duas implementações foram testadas: uma utilizando *Spinlock* como ferramenta de *lock*, e outra utilizando *Mutex*. Quando este for o caso os resultados para ambas as implementações são apresentados. Nos demais testes apenas a implementação com *Spinlock* foi utilizada, e os resultados são demonstrados com variação em outros parâmetros.

O segundo, por sua vez, é um cenário com carga aleatória e tem por objetivo observar o comportamento da estratégia de sincronização em um ambiente mais perto de uma aplicação real, onde operações de inserção, busca e remoção acontecem paralelamente. Neste cenário, cada *thread* executa as três operações, escolhidas de forma aleatória.

Ainda, um terceiro cenário extra foi elaborado. Este, chamado de cenário de comparação *monothread*, tem por objetivo evidenciar o benefício do uso da estrutura de dados e da estratégia de sincronização em um ambiente *multithread*, frente ao uso da estrutura em um cenário *monothread*.

A fim de evitar uma possível influência nos resultados dos testes e aproximar os experimentos de aplicações reais, a geração das chaves dos objetos é feito de forma aleatória. Além disso, todos os experimentos foram repetidos no mínimo 20 vezes. Portanto, todos os valores indicados ao longo deste capítulo são médias tomadas destas medições. Ainda, sempre que possível, as medidas em questão são relativizadas em relação ao tempo para facilitar a comparação direta entre diferentes métricas.

6.1 Cenário com Carga Controlada

Neste cenário, 16 milhões de objetos são primeiramente inseridos, depois buscados, e então removidos. O número de *threads* responsáveis pelas operações varia de 1 a 32. Quando mais de uma *thread* é criada o total de objetos é dividido igualmente entre elas. Dessa maneira é possível observar a variação da concorrência e eficiência da execução da mesma tarefa por diferentes conjuntos de *threads*.

6.1.1 Concorrência

Três métricas foram utilizadas com o objetivo de analisar o nível de concorrência do cenário com carga controlada em relação do número de *threads* utilizadas. São elas:

- Número de retentativas: Na estratégia de sincronização implementada retentativas de inserção ou remoção ocorrem quando duas ou mais *threads* tentam modificar a mesma região da árvore. Ou seja, quando duas ou mais *threads* concorrem pela operação. Portanto, o número de retentativas é encarada como uma métrica de concorrência.

- Número de troca de contextos: A taxa com que ocorrem trocas de contexto é relacionada ao número de *threads* competindo pelo processador. Enquanto houver a relação de pelo menos um núcleo por *thread*, espera-se que o número de troca de contextos se mantenha baixo. Quando esta relação fica abaixo de um núcleo por *thread*, espera-se que esta medida aumente consideravelmente. Este comportamento é analisado nas próximos parágrafos.

- Número de migrações de CPU: Por questões de escalonamento, *threads* são migradas de um núcleo para outro. Esta medida apresentou grande variação relacionada ao número de *threads* utilizadas e, portanto, é analisada nas próximas seções. Com comportamento similar ao número de troca de contextos, é esperado que essa medida aumente consideravelmente quando houverem mais *threads* executando do que núcleos disponíveis.

6.1.1.1 Retentativas

As figuras 6.1, 6.2 e 6.3 apresentam as medições para números de retentativas por segundo no cenário com carga controlada. As figuras 6.1 e 6.2 apresentam as medidas realizadas na inserção de 16 milhões de objetos, com variação no número de *threads*. Para esta métrica foram utilizadas uma implementação com *Spinlock* e outra com *Mutex*.

Figura 6.1: Retentativas por segundo na inserção - até 16 *threads*

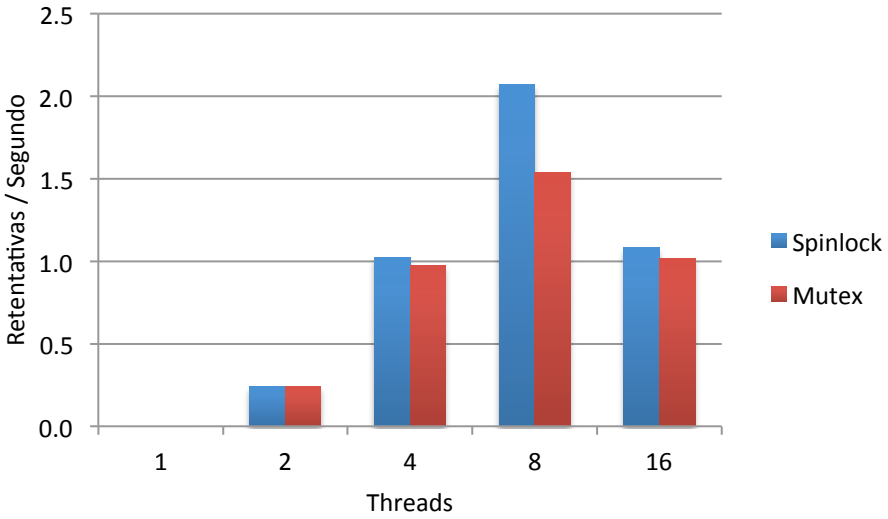


Figura 6.2: Retentativas por segundo na inserção - até 32 *threads*

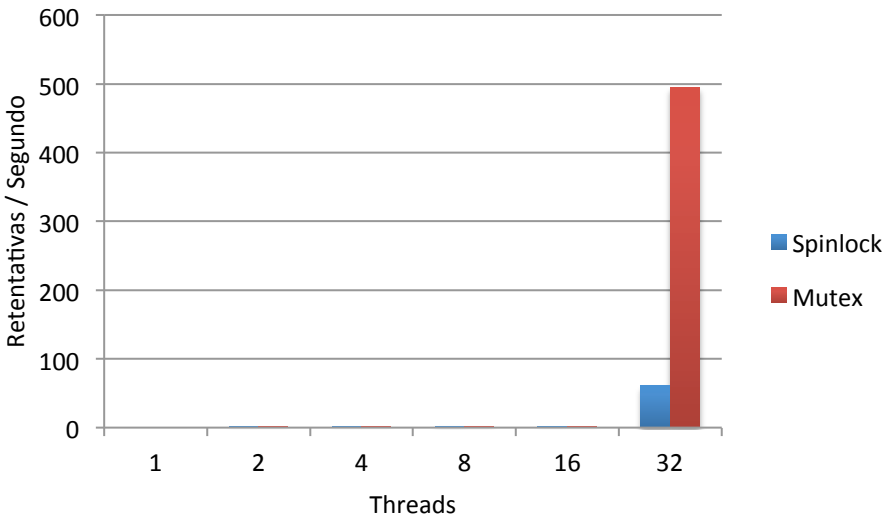
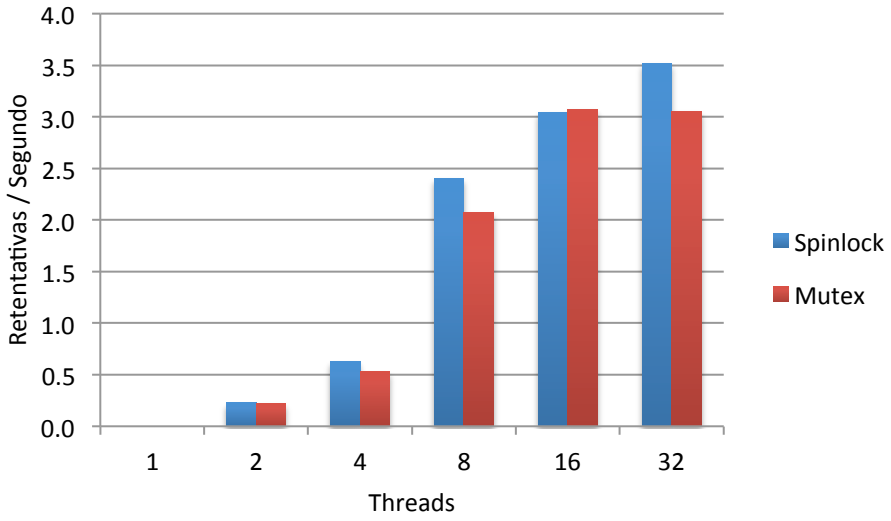


Figura 6.3: Retentativas por segundo na remoção



Na figura 6.1 é possível observar o crescimento do número de retentativas executadas por segundo a medida que se aumenta o número de *threads* executando a operação. Quanto mais *threads* inserindo, maior a quantidade de conflitos de inserção e, portanto, maior a concorrência. O aumento desta métrica é moderado até 8 *threads*, chegando a 2 retentativas por segundo na versão com *Spinlock* e 1,5 retentativas por segundo na versão com *Mutex*. Há ainda uma queda nesta taxa para a execução com 16 *threads*.

A figura 6.2, por sua vez, mostra a diferença deste comportamento quando o número de *threads* inserindo paralelamente passa para 32. O tipo de *lock* utilizado também passa a oferecer uma variação maior nos resultados. Na versão com *Spinlock* o número de retentativas executadas por segundo sobe para cerca de 60. A versão com *Mutex*, por sua vez, tem um aumento ainda maior, apresentando cerca de 500 retentativas por segundo no mesmo cenário.

A figura 6.3, ainda, apresenta o número de retentativas no cenário de exclusão. 16 milhões de objetos são excluídos por variados números de *threads*. Os dois tipos de *locks* também foram testados. Neste caso, o aumento da concorrência é moderado mesmo até 32 *threads*, chegando a taxas de 3 retentativas por segundo para a implementação com *Mutex* e 3.5 retentativas por segundo para a implementação com *Spinlock*.

6.1.1.2 Trocas de Contexto

As figuras 6.4 e 6.5 apresentam as medições de trocas de contexto realizadas no cenário com carga controlada. Em cada figura diferentes números de *threads* são apresentados para facilitar a visualização das medidas. Ainda, cada figura apresenta as medições para as três operações.

Figura 6.4: Trocas de contexto por segundo - até 4 *threads*

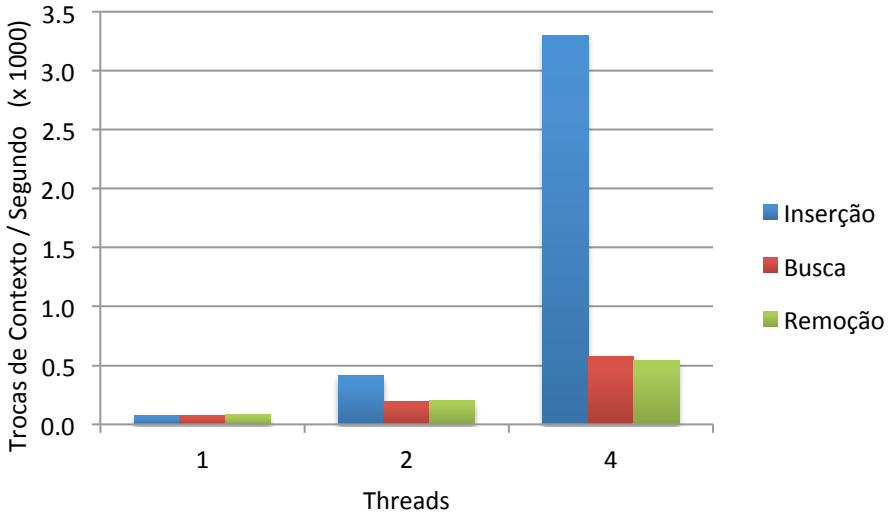
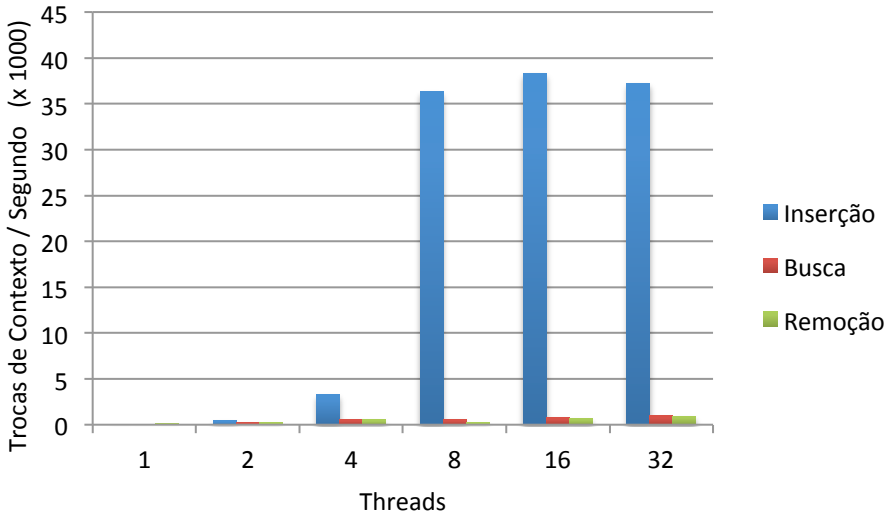


Figura 6.5: Trocas de contexto por segundo - até 32 *threads*



O número de trocas de contexto cresce bastante no cenário de inserção, enquanto não demonstra muita variação nos cenários de busca e exclusão. Nas inserções, cerca de 3,3 mil trocas de contexto acontecem por segundo quando 4 *threads* inserem em paralelo na estrutura. Quando o número de *threads* aumenta, contudo, a taxa cresce para valores em torno de 37 mil trocas de contexto por segundo.

6.1.1.3 Migrações de CPU

Foi medida ainda a taxa de migrações de CPU que ocorre durante a inserção, busca e remoção, para diferentes números de *threads*.

Figura 6.6: Migrações de CPU por segundo - até 4 *threads*

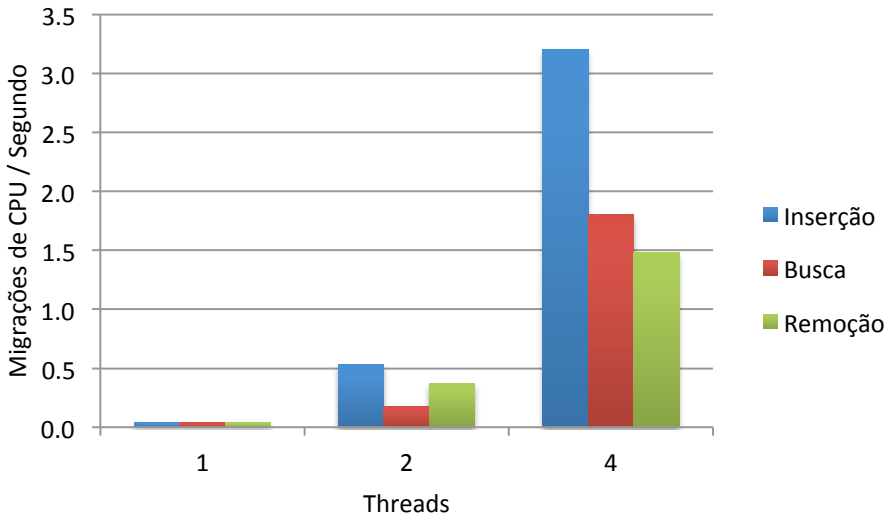


Figura 6.7: Migrações de CPU por segundo - até 8 *threads*

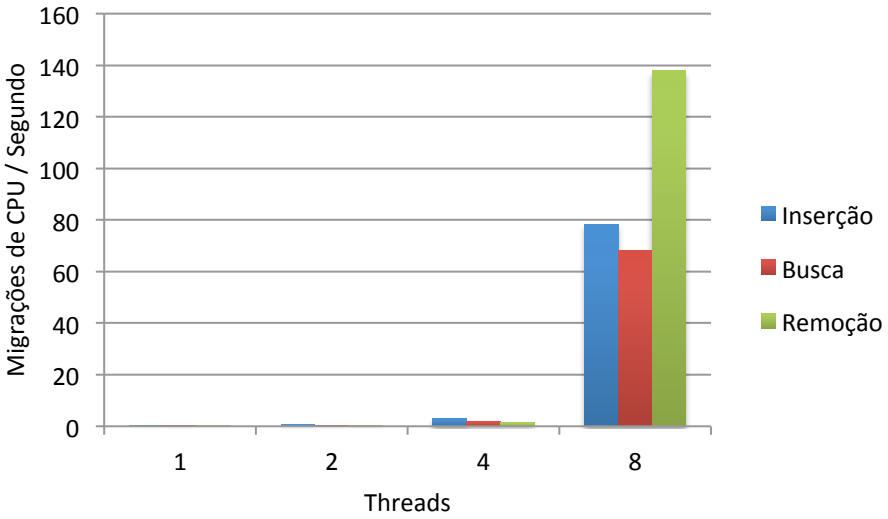
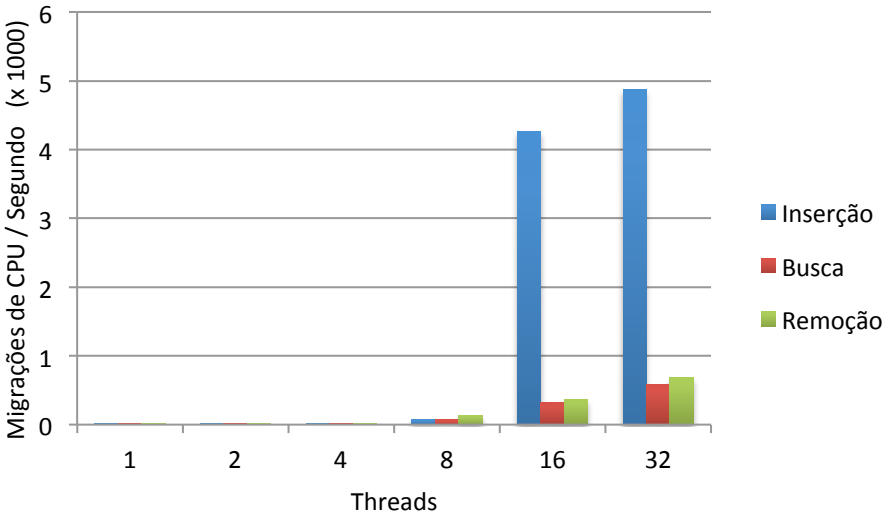


Figura 6.8: Migrações de CPU por segundo - até 32 *threads*



As figuras 6.6 e 6.7 apresentam as medidas obtidas para esta métrica até 4 e 8 *threads*. O aumento das medidas para 4 *threads* é moderado em relação ao uso de 2 *threads*. Este aumento é mais significativo quando 8 *threads* são utilizadas, variando entre 70 e 140 migrações de CPU por segundo, dependendo da operação.

Os número crescem demasiadamente para a operação de inserção quando 16 ou 32 *threads* são utilizadas. A figura 6.8 mostra que a taxa alcança valores entre 4 e 5 mil migrações de CPU por segundo quando um número maior de *threads* é utilizado. O aumento desta medida para as operações de busca e remoção, contudo, é bem menos significativo.

6.1.1.4 Análise de Concorrência

As medidas apresentadas demonstram dois principais perfis de concorrência nas operações da estrutura. Buscas e remoções observam menores consequências quando do aumento do número de *threads*. Mesmo com mais *threads* do que núcleos disponíveis as operações de busca e remoção não demonstram um aumento exagerado nas medições apresentadas. Nas próximas seções serão apresentadas as medidas de desempenho destas operações e a sua relação com a baixa concorrência observada.

A operação de inserção, por outro lado, observa maiores impactos na concorrência. Este impacto aparece em momentos diferentes mas próximos em cada uma das medidas. Para a taxa de retentativas por segundo observadas na inserção os resultados apresentam um crescimento bastante grande quando do uso de 32 *threads*. Neste cenário as inserções simultâneas em regiões conflitantes da árvore se tornam muito mais frequentes. Quando medindo a taxa de trocas de contexto observamos o maior aumento a partir de 8 *threads*. Esse comportamento provavelmente está relacionado com o fato de que o cenário de testes possui 4 núcleos de execução. Quando a quantidade de *threads* ultrapassa o número de núcleos disponíveis para execução a taxa de troca de contextos tende a aumentar a fim de garantir o progresso na execução de todas as *threads*.

Ainda, nas medidas da taxa de migrações de CPU, observa-se que o salto nas medidas acontece a partir de 16 *threads*. Esta métrica pode estar bastante relacionada a estratégias de escalonamento do sistema operacional utilizado. Acredita-se que esta métrica possa ser utilizada como um indicativo complementar mas não como uma métrica isolada do nível de concorrência observado no sistema.

A estratégia de sincronização elaborada garante a execução *wait-free* das operações de busca. As operações de remoção, por sua vez, têm garantia *lock-free*. A principal diferença entre a implementação destas duas operações é a obtenção e liberação de dois *locks* por parte da operação de remoção. A partir das medidas apresentadas é possível concluir que esta diferença na execução é pouco significativa em termos de concorrência. Mesmo obtendo *locks* e, portanto, evitando a operação de outras *threads* naquela região por algum curto período, a execução dentro do *lock* é tão rápida e a distribuição dos *locks* é tão dispersa que conflitos de fato raramente ocorrem.

Inserções também possuem garantia de execução *lock-free*, da mesma maneira que exclusões. Ao comparar a implementação das duas operações é possível identificar algumas linhas de configuração de informações de roteamento dos novos nodos a serem inseridos.

Estas, porém, não justificam o aumento nas medidas das métricas de concorrência. O diferente perfil de execução da operação de inserção se deve à alocação de memória inerente ao seu objetivo. Um nodo auxiliar e um nodo folha precisam ser alocados a cada inserção. A cada requisição de memória a *thread* interrompe a sua execução até que a chamada de sistema retorne a memória alocada. Esta espera faz com que o sistema provavelmente escalone a *thread* para dar espaço a outra que esteja pronta para executar, causando uma troca de contexto. Quando muitas *threads* concorrem pelo processador, o número de trocas de contexto e migrações de CPU cresce demasiadamente. Alocação de memória é, portanto, o principal custo da operação de inserção. Este prejudica a estratégia de sincronização e diminui o desempenho e escalabilidade da operação.

6.1.2 Desempenho

Três medidas foram utilizadas para avaliar o desempenho da estrutura e da estratégia de sincronização no cenário com carga controlada. São elas:

- Tempo de execução: Para cada número de *threads* foi medido o tempo necessário para inserir, buscar e remover 16 milhões de objetos na estrutura. Idealmente, quanto mais *threads* executando uma parte da tarefa, menos tempo seria necessário para completar a tarefa. Na prática este resultado é diferente, como se pode observar nos próximos parágrafos.

- Operações por segundo: Para cada variação no número de *threads* foi medida a taxa de inserção, busca e remoção alcançada. Esta métrica é uma das mais diretas e objetivas. Entender quantos objetos se pode inserir, buscar e remover da estrutura a cada segundo é uma avaliação suficiente para muitos casos de aplicação.

- Instruções por segundo: Ainda como uma terceira métrica de desempenho, mas bastante relacionada à métrica de operações por segundo, medimos a quantidade de instruções executadas pelo processador por segundo. A ideia é observar com que número de *threads* se alcança a maior taxa de instruções executadas por segundo.

6.1.2.1 Tempo de Execução

As figuras 6.9, 6.10 e 6.11 apresentam as medidas de tempo de execução para a inserção, busca e remoção de 16 milhões de objetos. O eixo X varia o número de *threads* responsáveis pela tarefa e as colunas azul e vermelha apresentam as medidas para a versão com *Spinlock* e a versão com *Mutex*, respectivamente.

Figura 6.9: Tempo de execução da inserção

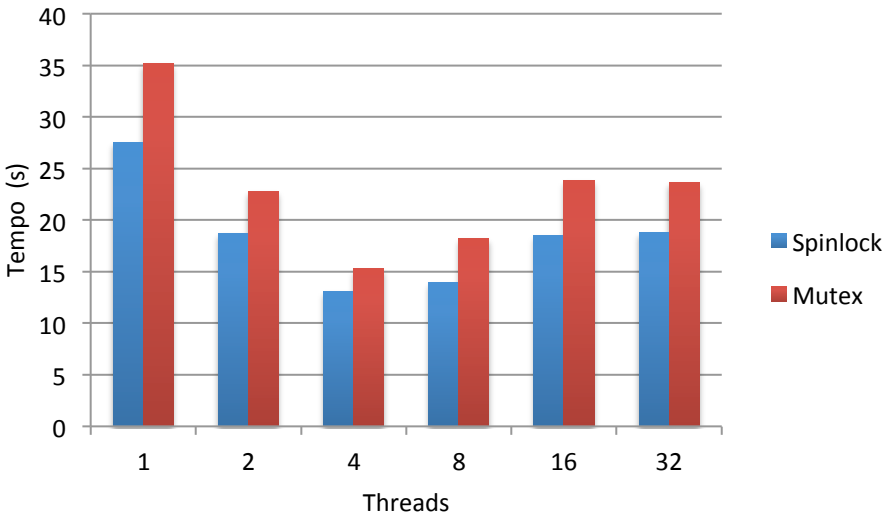


Figura 6.10: Tempo de execução da busca

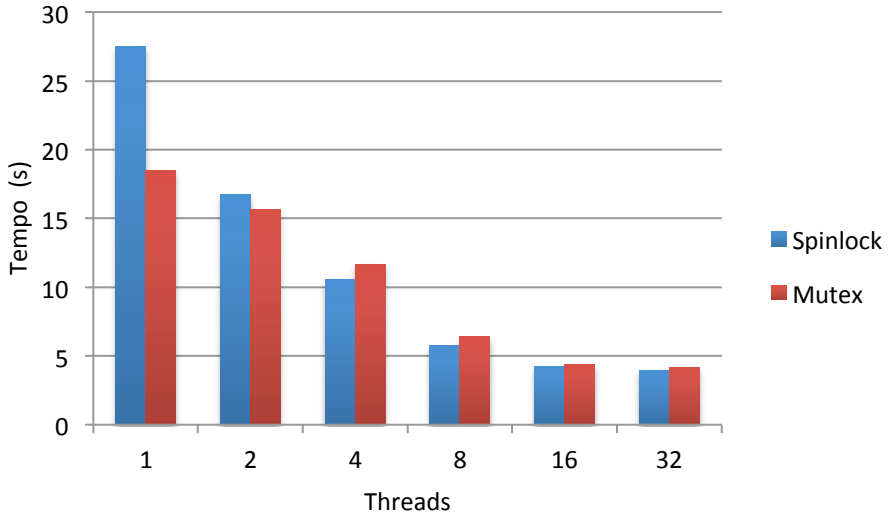
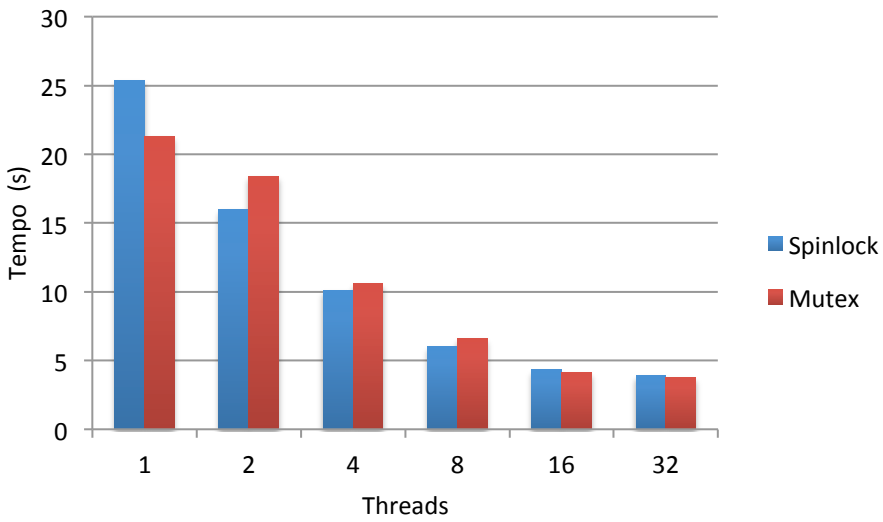


Figura 6.11: Tempo de execução da remoção



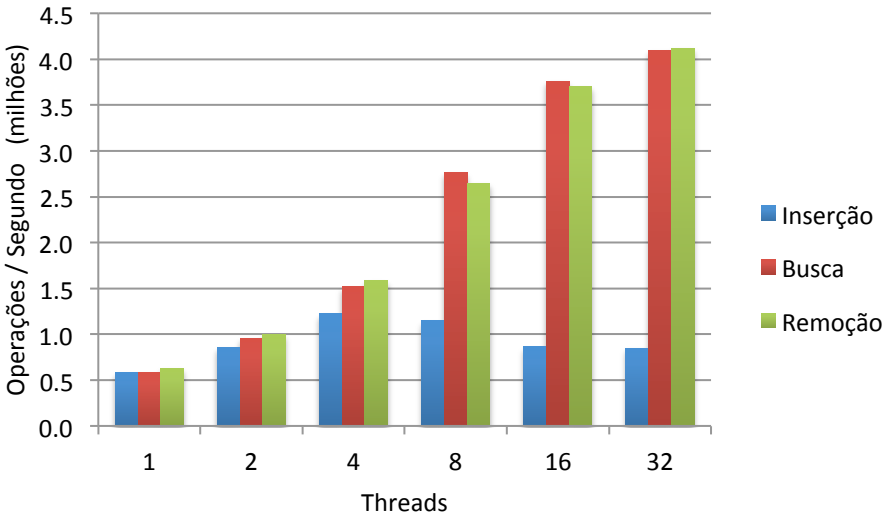
A figura 6.9 apresenta o benefício que a tarefa de inserção dos 16 milhões de objetos obtém da execução *multithread* em termos de tempo de execução. Este benefício é observado claramente até 4 *threads*. A execução com 8 *threads* apresenta um desempenho não muito diferente da execução com 4 *threads*, mas demonstra uma tendência de aumento no tempo de execução. Esta tendência é confirmada quando do uso de 16 e 32 *threads*.

As figuras 6.10 e 6.11 apresentam as medidas de tempo de execução para a busca e remoção, respectivamente, dos 16 milhões de objetos. Estas operações continuam obtendo benefícios na execução *multithread* mesmo após o número de *threads* ultrapassar o número de núcleos disponíveis para execução. A diferença no tempo de execução deixa de ser significativa a partir de 16 *threads*, mas diminui consideravelmente até este número.

6.1.2.2 Operações

A figura 6.12 apresenta as medidas obtidas para a taxa de operações executadas por segundo. Cada linha apresenta uma das operações.

Figura 6.12: Operações por segundo



A taxa de inserções realizadas na estrutura aumenta até 4 *threads*, onde atinge pouco mais de 1,2 milhões de inserções por segundo. O desempenho quando utilizando 8 *threads* novamente não muda consideravelmente, mas demonstra uma tendência de queda. A taxa de inserção cai mais significativamente a partir de 16 *threads*.

As taxas de busca e remoção, por sua vez, não demonstram queda no desempenho. Os valores aumentam consideravelmente até 16 *threads*, mas ainda apresentam um ganho visível com 32 *threads*, chegando a cerca de 4,1 milhões de buscas e remoções por segundo.

6.1.2.3 Instruções e Ciclos

A taxa de instruções executadas pelo processador a cada segundo é apresentada, para cada operação, na figura 6.13. Ainda, a taxa de ciclos por segundo executados também é apresentada, esta na figura 6.14.

Figura 6.13: Instruções por segundo

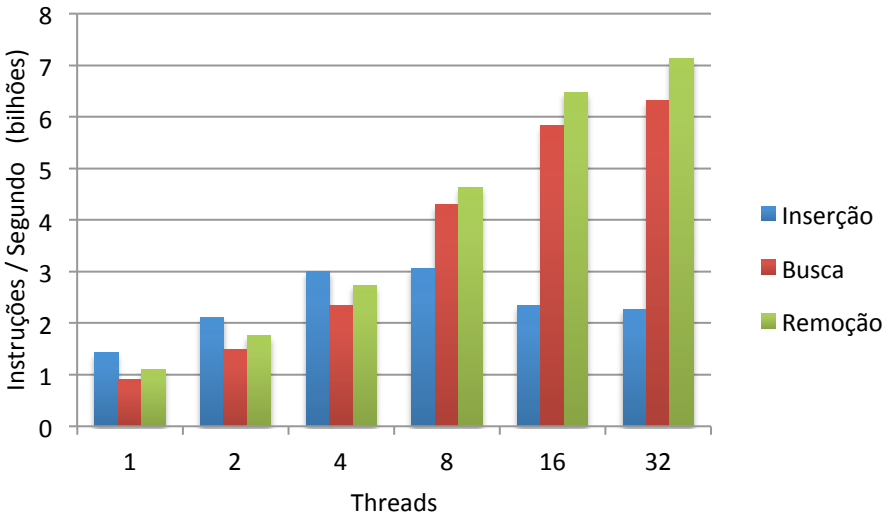
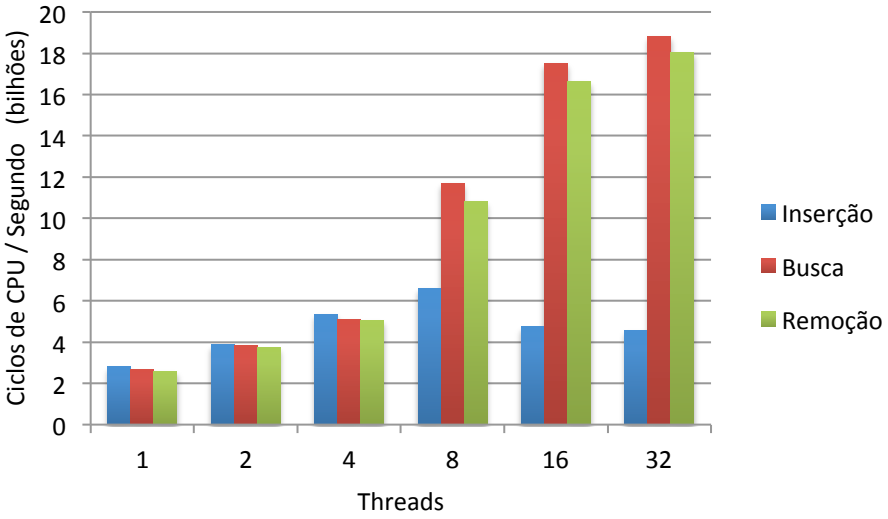


Figure 6.14: Ciclos de CPU por segundo



Os gráficos apresentam comportamento similar ao gráfico da figura 6.12. Novamente é possível observar que a operação de inserção encontra seu pico de desempenho entre 4 e 8 *threads*, caindo a partir de então. As operações de busca e remoção, por sua vez, apresentam comportamento crescente até o número de *threads* utilizadas nos testes.

6.1.2.3 Análise de Desempenho

Das figuras 6.9, 6.10 e 6.11, que apresentam o tempo de execução para cada operação, é possível observar a relação do desempenho com o nível de concorrência apresentado anteriormente. A operação de inserção observa benefício da execução paralela enquanto os níveis de concorrência se mantém baixos. A partir de 16 *threads*, quando a concorrência aumenta demasiadamente, o desempenho da operação é prejudicado. O peso da concorrência aumenta o tempo de execução da tarefa.

As operações de busca e remoção, por sua vez, não encontram aumento no seu tempo de execução. Como analisado anteriormente, estas operações não sofrem consequências significativas com o aumento do número de *threads*, observando ainda assim baixos níveis de concorrência. Este comportamento se reflete no desempenho destas operações. O aumento de desempenho, porém, é reduzido a medida que se acrescenta mais *threads*. A baixa concorrência evita a perda de desempenho. Por outro lado, a limitação no número de núcleos de execução reduz o aumento de desempenho para pequenas vantagens entre 16 e 32 *threads* na busca, e atinge a relativa estabilização do tempo entre 16 e 32 *threads* na remoção.

As medidas de tempo de execução da inserção demonstram um melhor desempenho da implementação que utiliza *Spinlock* como seu mecanismo de *lock*. Este desempenho demonstra o perfil de baixa concorrência alcançado pela estratégia de sincronização: como são raros os momentos em que uma *thread* precisa retentar a obtenção do *lock*, o menor *overhead* de obtenção e liberação de *lock* do *Spinlock* oferece melhores tempos de execução. Este comportamento se repete nas operações de busca e remoção quando da utilização de várias *threads*. As medidas obtidas, contudo, invertem este comportamento quando do uso de apenas uma *thread*. Este comportamento não condiz com o esperado, principalmente quando há a certeza de que o *Spinlock* não efetuará retentativas de obtenção do *lock*.

A figura 6.12 apresenta a taxa de operações por segundo alcançada para cada conjunto de *threads*. Os resultados correspondem ao tempo de execução analisado anteriormente. Na inserção, os picos ficam entre 4 e 8 *threads* e caem a partir daí. As operações de busca e remoção, por sua vez, continuam a escalar a medida que mais *threads* são adicionadas à execução.

As figuras 6.13 e 6.14 seguem o mesmo padrão de comportamento, apresentando picos de 3 bilhões de instruções por segundo e 6,6 bilhões de ciclos de CPU por segundo para a inserção. As operações de busca e remoção, que mantém a escalabilidade até o número de

threads testadas, chegam a cerca de 7 bilhões de instruções por segundo e 17 bilhões de ciclos de CPU por segundo.

É interessante observar que, até 4 *threads*, a relação entre os valores de cada operação, em cada gráfico, permanece estável. Igualmente, relacionando os gráficos e observando seu comportamento até 4 *threads* é possível observar que, apesar de executar menos operações por segundo, as inserções executam mais instruções por segundo do que as demais operações. Isso acontece porque para cada inserção na árvore esta operação de fato executa mais instruções enquanto define valores de roteamento da estrutura e aloca memória para os novos nodos. Este comportamento também explica o posicionamento levemente superior das colunas da operação de remoção em relação a busca, ao longo do gráfico da figura 6.13. Apesar de executarem taxas muito próximas de operações por segundo, a operação de remoção de fato executa mais instruções a cada remoção por necessitar obter e liberar *locks*, além de manter informações de roteamento do caminho percorrido a fim de efetuar a remoção.

Os gráficos das três figuras são profundamente relacionados e, portanto, apresentam comportamento muito similar. É visível a exemplar escalabilidade alcançada nas operações de busca e remoção. Estas tendem a aumentar o seu desempenho mesmo quando o número de *threads* ultrapassa o número de unidades de execução disponíveis. A operação de inserção, por sua vez, apresenta escalabilidade limitada. Esta limitação está relacionada ao grande aumento dos níveis de concorrência observados quando o número de *threads* ultrapassa o número de unidades de execução, como explicado na seção 6.1.1.4.

6.2 Cenário com Carga Aleatória

No cenário com carga aleatória cada iteração do programa de teste roda, com número variado de *threads*, por 30 segundos. Além disso, cada *thread* tem a possibilidade de executar qualquer uma das três operações, o que é definido aleatoriamente durante a execução. Assim como no cenário com carga controlada, as chaves de cada objeto são geradas de forma aleatória. Neste caso, porém, a variação das chaves é limitada no intervalo de zero a um milhão.

A definição aleatória de cada operação executada por cada *thread*, juntamente com a limitação de intervalo da chave, resulta em um comportamento interessante e benéfico para o cenário de teste. Como a chave a ser inserida, buscada ou removida é definida de maneira aleatória, a quantidade de objetos presentes na estrutura tende a aumentar enquanto este

número for menor do que metade da variação da chave, ou seja, até 500 mil objetos. Isso acontece porque, inicialmente, a maior parte das inserções é de objetos que ainda não existem na estrutura e, portanto, acontecem com sucesso. As remoções, por sua vez, na maior parte das vezes são de chaves que ainda não se encontram na estrutura e, portanto, falham. Quando o número de objetos presente na árvore se aproxima da metade possível, cerca de metade das inserções acontece com sucesso, enquanto cerca de metade das remoções também acontece com sucesso. A velocidade com que o número de objetos converge para este valor depende da quantidade de *threads* ativas no teste. Porém, independente do número de *threads*, um equilíbrio em torno de 500 mil objetos sempre é alcançado. Este comportamento é benéfico para o cenário de teste por garantir que haverá sucesso e falha, na mesma proporção, para qualquer uma das três operações.

6.2.1 Concorrência

A fim de analisar a concorrência no cenário com carga aleatória, medidas de trocas de contexto e migrações de CPU foram obtidas. A seção 6.1.1 explica as razões pelas quais tais medidas são utilizadas para a avaliação da concorrência no cenário com carga controlada. As mesmas razões se aplicam ao cenário com carga aleatória.

6.2.1.1 Trocas de Contexto

As figuras 6.15 e 6.16 apresentam a taxa de trocas de contexto por segundo na execução do teste do cenário com carga aleatória durante 30 segundos para diferentes números de *threads*. Para fins de comparação, o mesmo dado para as inserções do cenário com carga controlada também é apresentado em cada gráfico.

Figura 6.15: Trocas de contexto por segundo - até 4 *threads*

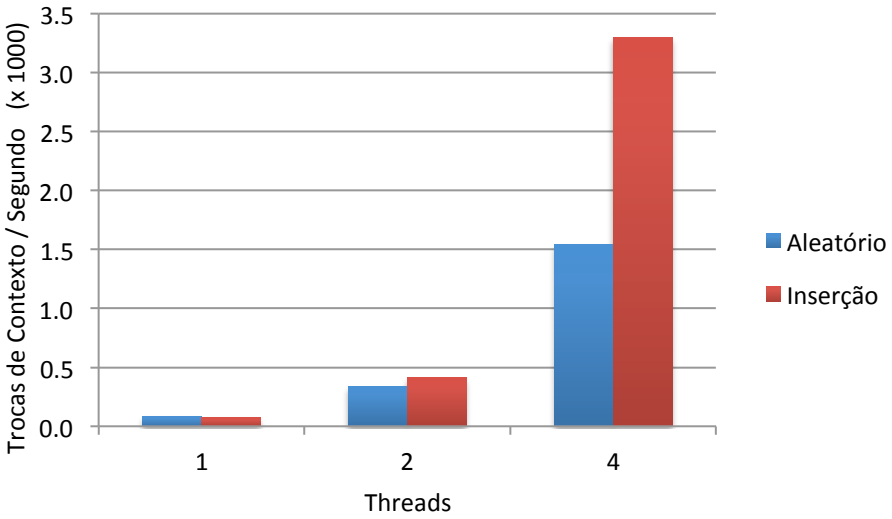
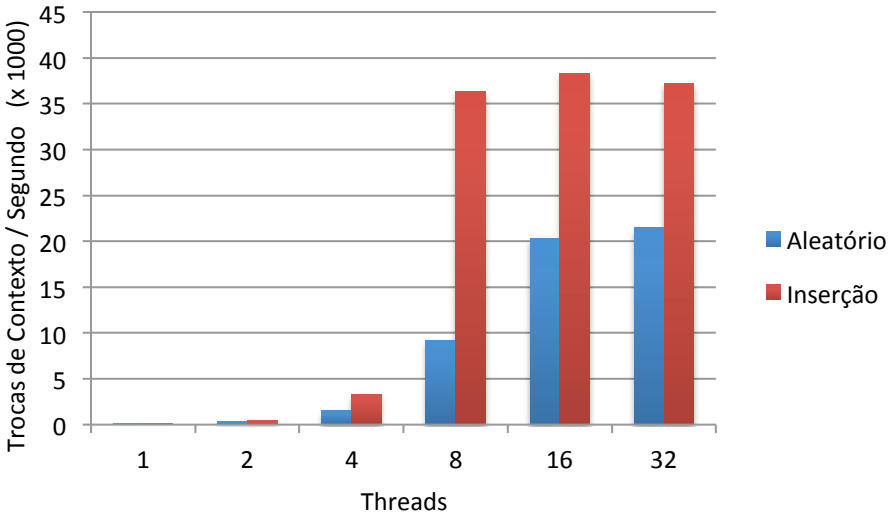


Figura 6.16: Trocas de contexto por segundo - até 32 *threads*



A taxa de trocas de contexto sobre para cerca de 1,5 mil quando da execução do cenário com carga aleatória com 4 *threads*. Este número cresce com o aumento do número de *threads* executando paralelamente, principalmente quando este número ultrapassa o número de núcleos disponíveis para execução, chegando a taxa de 21 mil trocas de contexto por segundo quando do uso de 32 *threads* no cenário com carga aleatória.

6.2.1.2 Migrações de CPU

As figuras 6.17, 6.18 e 6.19 apresentam a taxa de migrações de CPU por segundo na execução do teste do cenário com carga aleatória durante 30 segundos, para diferentes números de *threads*. Novamente, para fins de comparação, os mesmos dados para as inserções do cenário com carga controlada também são apresentados em cada gráfico.

Figura 6.17: Migrações de CPU por segundo - até 4 *threads*

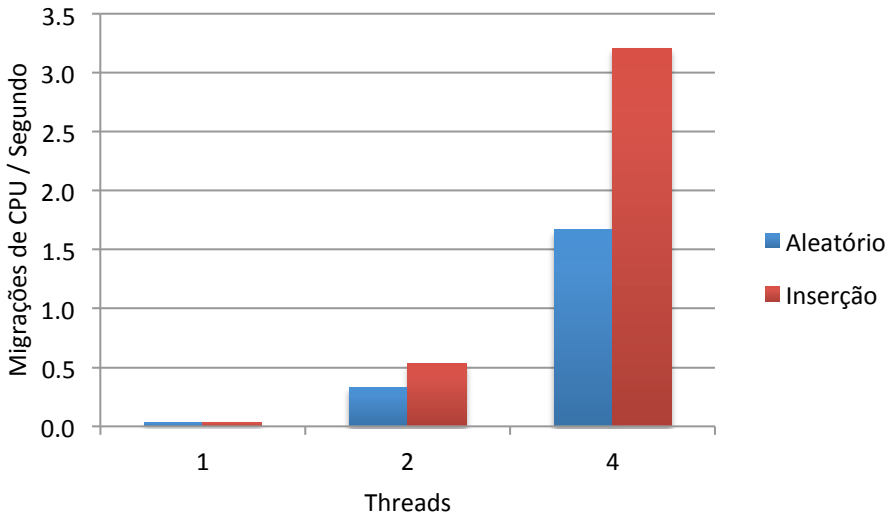


Figura 6.18: Migrações de CPU por segundo - até 8 *threads*

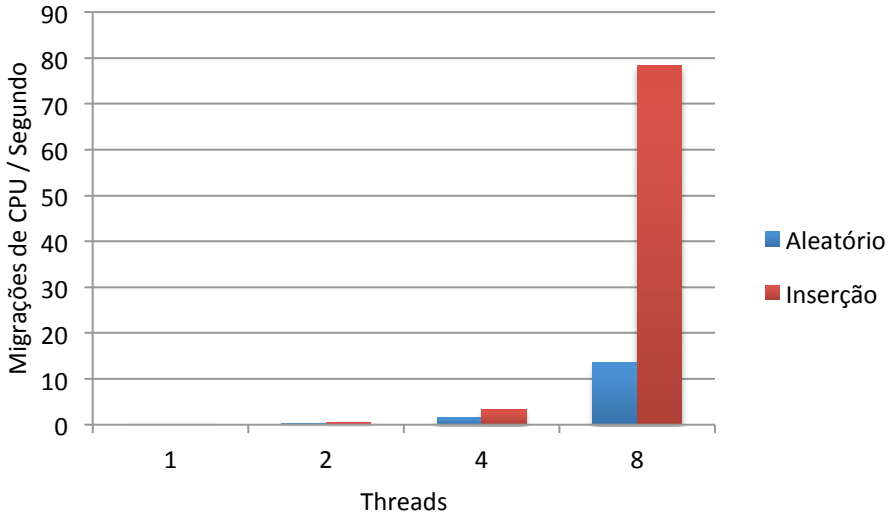
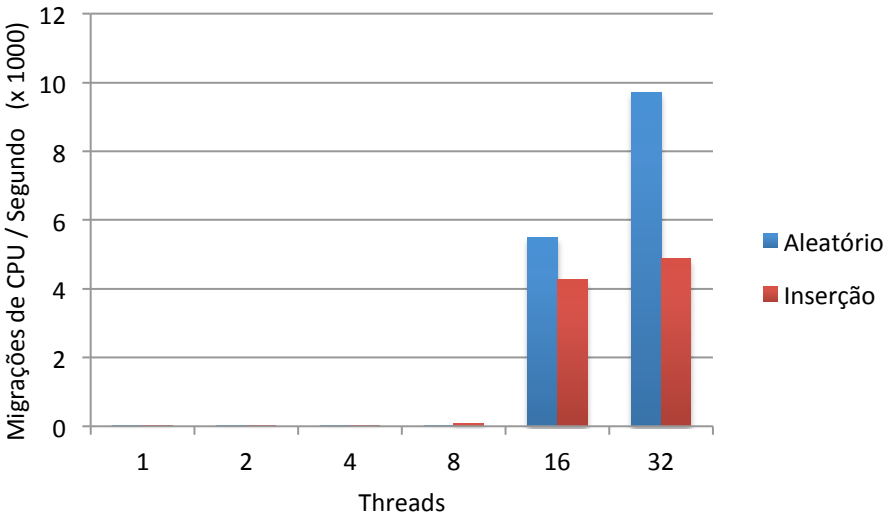


Figura 6.19: Migrações de CPU por segundo - até 32 *threads*



A taxa de migrações de CPU cresce moderadamente até 8 *threads*. Este valor sobe demasiadamente a partir de 16 *threads*, chegando a quase de 10 mil migrações de CPU por segundo quando da execução do teste do cenário com carga aleatória com 32 *threads*.

6.2.1.3 Análise de Concorrência

As colunas que apresentam as medidas de trocas de contexto e migrações de CPU para as inserções do cenário com carga controlada foram adicionadas aos gráficos das figuras 6.15 a 6.19 para facilitar a comparação entre estas medidas e as medidas do cenário com carga aleatória.

As medidas de trocas de contexto apresentadas nas figuras 6.15 e 6.16 apresentam valores altos, mas abaixo dos valores das inserções do cenário com carga controlada. A operação de inserção se mostrou como a operação com maior carga desta medida no cenário com carga controlada. O cenário com carga aleatória executa, juntamente com as inserções, também buscas e remoções. Estas, no cenário com carga controlada, apresentaram taxas extremamente menores de trocas de contexto. Não é possível afirmar que as taxas da execução conjunta destas operações resultaria em uma média das medidas isoladas previamente apresentadas. Ainda assim, é razoável esperar que a execução do cenário com carga aleatória não apresente taxas tão altas quanto a execução isolada das operações de inserção.

As figuras 6.17, 6.18 e 6.19, que apresentam as taxas de migração de CPU para o cenário com carga aleatória, demonstram o mesmo comportamento discutido no parágrafo anterior até o uso de 8 *threads*. A partir da execução do cenário com carga aleatória com 16 *threads*, contudo, as taxas para este cenário ultrapassam em grande número as taxas alcançadas nas inserções do cenário com carga controlada. Este comportamento não segue o raciocínio explicitado para o comportamento das taxas de trocas de contexto. A execução concorrente das 3 operações por um número grande de *threads* resulta em um aumento ainda maior destas medidas. Como citado anteriormente, esta métrica é tomada como indicativo adicional para a análise da concorrência e, neste cenário, apresenta um comportamento curioso do qual não foi possível obter conclusões.

As taxas apresentadas nesta seção, ainda, estão acima das taxas observadas para as buscas e remoções do cenário com carga controlada. As altas taxas se devem à presença de operações de inserção no cenário com carga aleatória. Esta operação, com um perfil de concorrência muito mais pesado do que as demais, eleva as medidas das métricas de concorrência neste cenário. Além de novamente demonstrar a singularidade do comportamento das inserções frente às buscas e remoções, estes dados demonstram que, em termos de concorrência, o estudo do comportamento das inserções paralelas pode levar a informações sobre o comportamento de cenários diferentes, nos quais as demais operações também aconteçam em similar proporção.

É interessante ainda citar que, apesar do número elevado de trocas de contexto e migrações de CPU, nenhuma retentativa foi executada durante as execuções do cenário com carga aleatória, para qualquer número de *threads*. Isto demonstra que, para cenários de inserção ou remoção pesada, a concorrência é maior, resultando em retentativas como visto nos gráficos das figuras 6.1, 6.2 e 6.3. Para cenários de diferentes operações paralelas como o cenário apresentado nesta seção, contudo, a concorrência diminui, e casos de retentativas caem a zero.

6.2.2 Desempenho

A fim de analisar o desempenho no cenário com carga aleatória foram obtidas medidas de taxas de ciclos de CPU por segundo e instruções executadas por segundo.

6.2.2.1 Instruções e Ciclos

A figura 6.20 apresenta as medidas de instruções executadas por segundo no cenário com carga aleatória e nas inserções do cenário com carga controlada em relação ao número de *threads* executando paralelamente. A figura 6.21, por sua vez, apresenta as medidas de ciclos de CPU por segundo no cenário com carga aleatória, assim como nas inserções do cenário com carga controlada.

Figura 6.20: Instruções por segundo

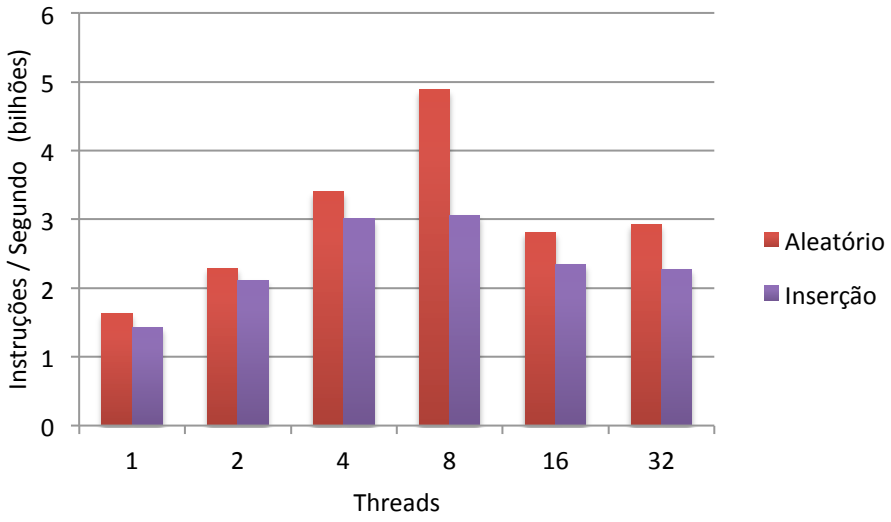
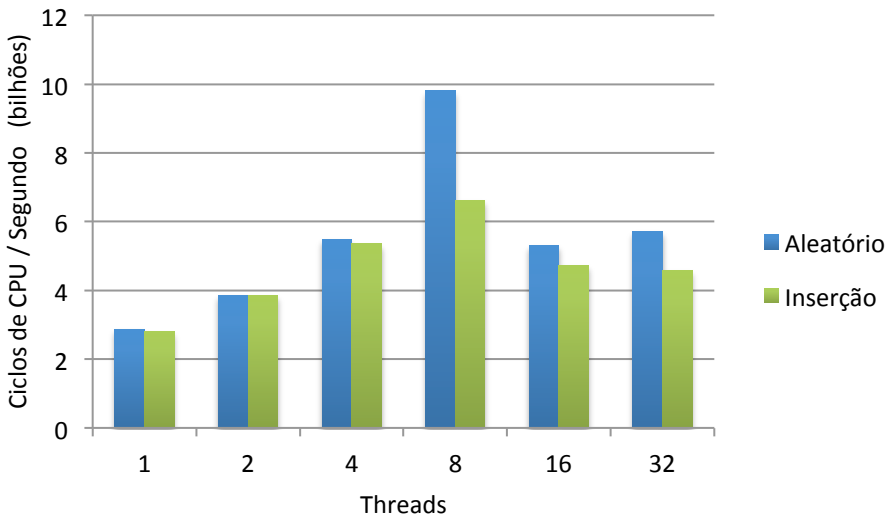


Figura 6.21: Ciclos de CPU por segundo



6.2.2.2 Análise de Desempenho

A figura 6.20 demonstra que o cenário com carga aleatória executa mais instruções por segundo do que as instruções do cenário com carga controlada. A execução paralela das três operações alcança taxas um pouco maiores de instruções por segundo, chegando a um pico de 4,9 bilhões com 8 *threads*. A figura 6.21 demonstra que a taxa de ciclos de CPU por segundo apresenta um comportamento similar, chegando a quase 10 bilhões de ciclos de CPU por segundo, quando executando em 8 *threads*.

É interessante observar que, apesar de alcançar taxas um pouco maiores, o comportamento do gráfico segue o perfil de escalabilidade das operações de inserção do cenário com carga controlada. Ou seja, a exemplar escalabilidade das operações de busca e remoção é prejudicada pelo perfil limitado da operação de inserção. As altas medidas das métricas de concorrência apresentadas em 6.2.1 justificam a queda de desempenho do cenário com carga aleatória para 16 ou mais *threads*. Além disso, a análise apresentada em 6.2.1.3 demonstra a influência do perfil de concorrência e, consequentemente, de desempenho e escalabilidade da operação de inserção em um cenário de operações aleatórias.

6.2 Cenário de Comparação *Monothread*

Com o objetivo de evidenciar os benefícios do uso da estrutura e da estratégia de sincronização em um ambiente *multithread*, frente a um ambiente *monothread*, outro teste foi elaborado. Neste, foram executadas inserções, buscas e remoções de 4, 8, 12 e 16 milhões de objetos na estrutura com uma, quatro e oito *threads*.

As figuras 6.22, 6.23 e 6.24 apresentam as medidas de tempo de execução para cada caso.

Figura 6.22: Tempo de execução da inserção

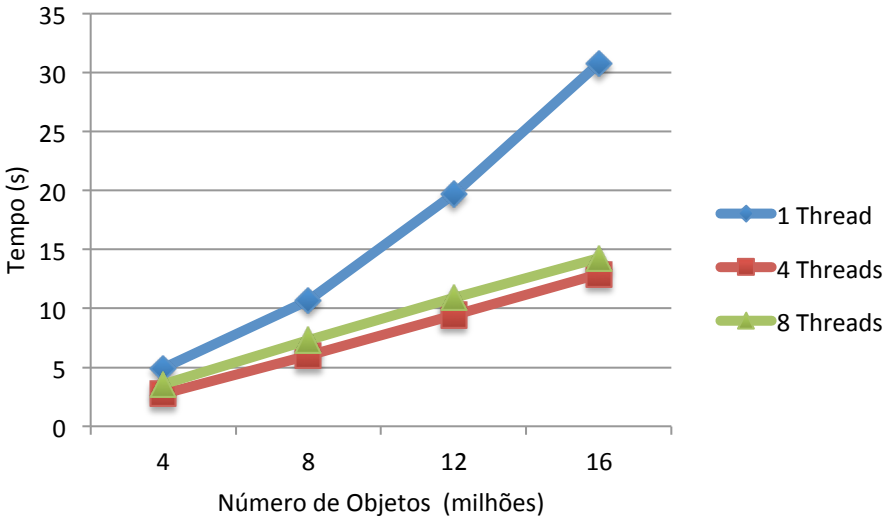


Figura 6.23: Tempo de execução da busca

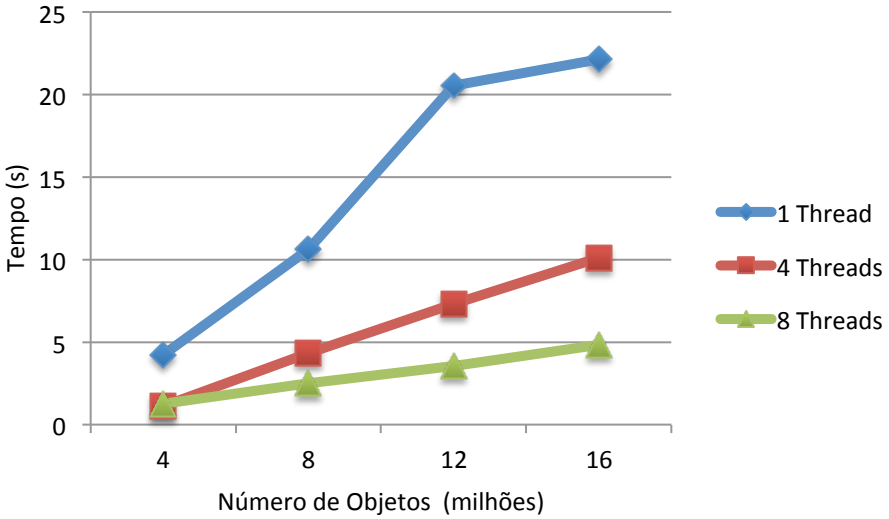
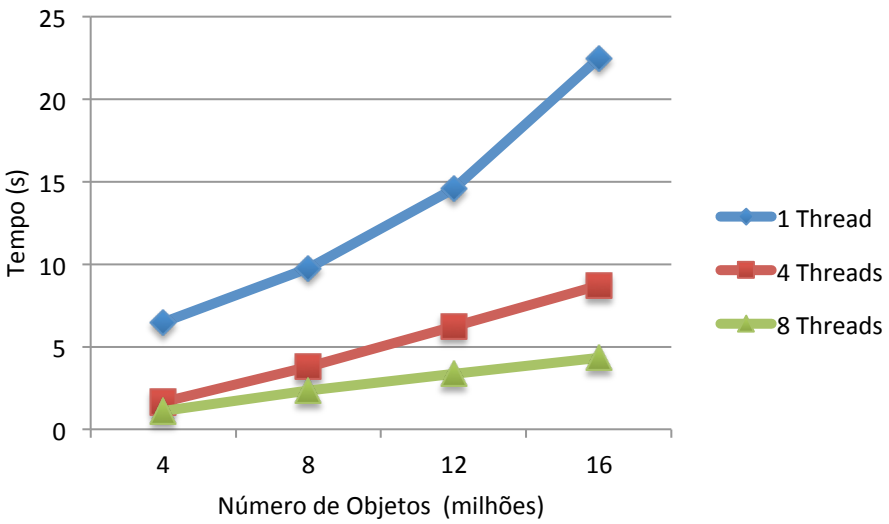


Figura 6.24: Tempo de execução da remoção



As curvas das figuras 6.22, 6.23 e 6.24 evidenciam a vantagem das operações quando executadas por múltiplas *threads*. Mais do que isso, as curvas demonstram a escalabilidade da solução proposta. As curvas das execuções com 4 e 8 *threads* avançam de maneira previsível e confiável, permitindo a antecipação do seu comportamento.

O gráfico de figura 6.22 demonstra um desempenho muito similar para as operações quando executadas por 4 ou 8 *threads*. Esse comportamento da inserção já havia sido observado na seção 6.1.2. Os gráficos das figuras 6.23 e 6.24, por sua vez, demonstram a escalabilidade das operações de busca e remoção, que aumentam seu desempenho de forma confiável a medida que mais *threads* são utilizadas nas operações.

7 CONCLUSÃO

Estruturas de dados são fundamentais na implementação de variados sistemas. A escolha da estrutura com as características certas para cada problema é fundamental para garantir o desempenho e escalabilidade da solução. Enquanto isso, o desempenho de sistemas computacionais cada vez mais depende de multiprocessamento e soluções que escalam nestes ambientes. Estruturas de dados clássicas, contudo, não são preparadas para ambientes *multithread*. Este trabalho oferece a proposta de uma estrutura de dados escalável e de alto desempenho preparada para ambientes de multiprocessamento.

O trabalho toma por base uma estrutura de dados em formato de árvore binária. Este tipo de estrutura oferece eficiência nas suas operações mesmo na presença de milhões de objetos. A partir daí desenvolveu-se uma estratégia de sincronização fundamental para alcançar o desempenho e escalabilidade propostos e tirar proveito de ambientes multiprocessados.

As análises apresentadas confirmam o funcionamento da estrutura, assim como o alto desempenho alcançado com a solução implementada. Foram realizados testes em diferentes cenários e obtidas diversas medidas que nos permitiram analisar as melhores configurações para o uso da solução. As operações de busca e remoção da estrutura, por exemplo, apresentam escalabilidade exemplar, oferencendo aumento de desempenho mesmo quando o número de *threads* ultrapassa o número de unidades de processamento disponíveis. A operação de inserção, por sua vez, também apresenta ótimo desempenho, mas sofre maiores consequências da concorrência, diminuindo o seu desempenho quando esta aumenta demasiadamente. A principal causa deste comportamento também foi indicada no trabalho.

O desenvolvimento de sistemas preparados para o multiprocessamento é desafiador. Tais desafios, contudo, uma vez superados, oferecem resultados eficientes e preparados para o futuro. Estruturas de dados *thread-safe* bem sincronizadas, por exemplo, tendem a escalar quando implantadas em ambientes com mais unidades de execução.

O presente trabalho, assim como os códigos fonte, ficará disponível para uso público. Desta maneira, espera-se contribuir para o estudo e desenvolvimento de sistemas *multithread*, assim como oferecer para a comunidade soluções de maior desempenho e escalabilidade.

8 TRABALHOS FUTUROS

Durante o desenvolvimento deste trabalho foram observados pontos interessantes para desenvolvimento em trabalhos futuros. Estes são indicados nas seções seguintes.

8.1 Pré-alocação de Memória

As análises apresentadas neste trabalho tornam evidente que as operações de inserção são as que apresentam a maior concorrência, principalmente quando o número de *threads* ultrapassa o número de unidades de execução disponíveis. Como consequência, o desempenho dessa operação e principalmente a sua escalabilidade são comprometidos.

Em busca de maiores explicações para este comportamento, análises sobre o código fonte foram realizadas. Uma única evidência que explica a diferença de comportamento entre operações de inserção e remoção foi encontrada, e esta é a alocação de memória. Cada operação de inserção faz duas requisições de alocação de memória ao sistema, e a consequência disso é visível nos resultados obtidos dos testes.

Para trabalhos futuros, portanto, é sugerida a implementação de uma estratégia de pré-alocação de memória. Retirando-se o custo destas chamadas de sistema da operação de inserção, espera-se que esta apresente desempenho e escalabilidade similares à operação de remoção. Dessa maneira, todas as operações apresentariam exemplar nível de escalabilidade.

8.2 *Garbage Collection*

A fim de alcançar os baixos níveis de concorrência e alto grau de paralelismo na estrutura de dados desenvolvida algumas decisões de projeto foram tomadas. Entre elas, percebeu-se a necessidade de postergar a liberação da memória na operação de remoção.

O caminhamento livre na árvore, citado ao longo do trabalho, oferece vantagens para todas as operações. Inserções e remoções não precisam de nenhum *lock* até encontrarem a posição onde desejam inserir ou remover um nodo. Buscas, por sua vez, nunca obtém nenhum *lock* e, consequentemente, apresentam perfil *wait-free* de execução.

Esta estratégia impede que a memória dos nodos removidos da árvore pela operação de remoção seja imediatamente liberada. No instante da remoção, outra *thread* pode estar acessando aqueles nodos durante o caminhamento livre.

Portanto, sugere-se como desenvolvimento para trabalhos futuros a implementação de um coletor de lixo (*garbage collector*) que gerencie a liberação desta memória. Observa-se,

ainda, a impossibilidade deste ser baseado em contagem de referência, pois esta estratégia exigiria mudanças significativas e comprometedoras na implementação da estratégia de sincronização. O coletor de lixo deverá provavelmente ser baseado em questões temporais, quando puder se garantir temporalmente que a memória não possa mais ser acessada.

REFERÊNCIAS

HERLIHY, M.; MOSS, J. E. B. **Transactional memory**: Architectural support for *lock-free* data structures. 1993. 20th ISCA.

KNIZHNIK, K. **Patricia tries**: A better index for prefix searches. 2008. Dr. Dobb's Journal.

PERF: Linux profiling with performance counters. Disponível em: <https://perf.wiki.kernel.org/index.php>. Acesso em: 01 julho 2014.

PERFORMANCE Counters for Linux (PCL) Tools and perf. Disponível em: < https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Developer_Guide/perf.html>. Acesso em: 01 julho 2014.

VYUKOV, D. **1024 Cores**. Disponível em: <http://www.1024cores.net>. Acesso em: 01 julho 2014.

ANEXO A
TRABALHO DE GRADUAÇÃO I

Escalabilidade de sincronização em ambientes multiprocessados de memória compartilhada

Tiago de Almeida[1], **Sérgio Luis Cechin**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

tiago@almeida.inf.br, cechin@inf.ufrgs.br

Abstract. *Parallel processing architectures have been the major proposed solution to the increasing demand for processing power. By multiplying the number of processing units, such architectures promise, in theory, to multiply the rate of instruction execution. In practice, however, this result is not observed. Multiple challenges prevent programmers to extract the full potential of parallel architectures. This paper examines some of these problems and proposes a blended solution, based on already known strategies. Additionally, we work on a case study for the implementation and testing of this solution.*

Resumo. *Arquiteturas de processamento paralelo têm sido a principal proposta de solução para a crescente demanda por poder de processamento. Multiplicando o número de unidades de processamento, tais arquiteturas prometem, na teoria, multiplicar a taxa de execução de instruções. Na prática, contudo, este resultado não é observado. Múltiplos desafios impedem que programadores consigam extrair o máximo potencial de arquiteturas paralelas. Este trabalho estuda alguns destes problemas e propõe uma solução mista, baseada em estratégias já conhecidas. Adicionalmente, é feito um estudo de caso para a aplicação e experimentação desta solução.*

1. Introdução

A execução concorrente de tarefas se mostrou uma necessidade desde o desenvolvimento dos primeiros sistemas operacionais. Mesmo rodando em apenas uma unidade de execução, o sistema deveria gerenciar diversas tarefas e, quando possível, passar ao usuário a impressão de que estas eram executadas ao mesmo tempo. Tanto tarefas do usuário quanto tarefas do próprio sistema deveriam executar de maneira minimamente intrusiva e garantir que a execução do conjunto de tarefas, como um todo, progredisse.

Para permitir a execução concorrente de várias tarefas em apenas um processador, soluções de escalonamento de tarefas foram implementadas. Dessa maneira, cada tarefa

executaria por um determinado tempo e então liberaria o processador para as próximas tarefas, até que recebesse sua próxima fatia de tempo para execução. Estas trocas aconteceriam tão rapidamente que passariam a impressão de que as tarefas estariam executando ao mesmo tempo.

Estratégias de escalonamento preocupam-se, entre outras, com questões de corretude e equidade. Corretude procura garantir que os dados de uma tarefa não sejam corrompidos durante o período em que esta não esteja executando. Estes dados podem ser corrompidos, por exemplo, por outra tarefa que também os usa em sua execução. Equidade, por sua vez, preocupa-se com o tempo de execução oferecido para cada tarefa, em relação ao conjunto total de tarefas. Tarefas relacionadas à questões de tempo-real, como a tarefa que controla o movimento do cursor do mouse na tela, por exemplo, devem obter tempo de execução suficiente para satisfazer tais requisitos. Por outro lado, o problema de *starvation*, quando uma tarefa nunca obtém a sua fatia de tempo de execução, também deve ser evitado.

Ao longo da evolução dos sistemas computacionais, soluções com mais de uma unidade de execução foram propostas e implementadas. Em busca de maior desempenho, super-computadores com diversos processadores foram construídos e, posteriormente, diversos computadores foram agregados para a construção de *clusters*. Mais recentemente, quando a indústria passou a enfrentar dificuldades em dar continuidade ao aumento da frequência dos processadores, outras soluções para o aumento de desempenho foram buscadas. Entre elas, passou-se a inserir diversos núcleos dentro de um mesmo chip, oferecendo então diversas unidades de execução em um mesmo processador. Em níveis diferentes mas semelhantes, os super-computadores, os *clusters* e os processadores com vários núcleos oferecem um ambiente de processamento paralelo, onde as tarefas não apenas concorrem por uma unidade de execução, mas executam de fato paralelamente.

A computação paralela trouxe consigo diversos desafios. Se por um lado desenvolvedores têm à sua disposição um poder de processamento somado muito maior, por outro lado não conseguem facilmente tirar proveito deste cenário. Dependência de dados e sincronização de tarefas evitam que programas consigam manter as diversas unidades de execução ocupadas ao mesmo tempo. Como consequência, o poder de processamento somado de todas as unidades é frequentemente subutilizado.

Diversas abordagens são utilizadas a fim de gerenciar o uso de múltiplas unidades de processamento. Em sistemas distribuídos em diversos computadores, a troca de mensagens pela rede procura sincronizar as execuções de maneira a garantir a corretude do sistema com o um todo. Em computadores com múltiplos processadores, ou processadores com múltiplos núcleos, a troca de mensagens entre processos também é utilizada, porém menos comum. Neste cenário, onde frequentemente a memória é compartilhada entre tarefas, APIs de sincronização gerenciam a execução de tarefas através de semáforos, *locks*, barreiras de memória e instruções atômicas.

O campo de estudo deste trabalho restringe-se a ambientes multiprocessados de memória compartilhada. Estratégias de sincronização para tais ambientes costumam oferecer uma grande dificuldade de programação, além de tendência a erros difíceis de serem encontrados. Algoritmos e estruturas de dados devem ser bem projetados a fim de escalarem a medida que novas unidades de execução sejam inseridas. Um algoritmo projetado

sem esta ideia pode encontrar na sincronização um gargalo de execução e, consequentemente, não oferecer aumento de desempenho em ambientes multiprocessados.

O presente trabalho apresenta os fundamentos e as atuais estratégias de sincronização para ambientes multiprocessados de memória compartilhada. As soluções disponíveis serão avaliadas a fim de identificar suas vantagens e desvantagens, assim como encontrar o seu melhor cenário de aplicação. Adicionalmente, será proposta a implementação e avaliação de uma estrutura de dados de alto desempenho, altamente paralela e escalável, para armazenamento e recuperação de dados. Tal estrutura deve adotar uma abordagem mista no uso das estratégias de sincronização, utilizando-se das melhores alternativas avaliadas no estudo prévio. Métricas de avaliação serão propostas para tal estrutura e seus métodos, a fim de comparar o seu desempenho a outras estruturas disponíveis na literatura.

2. Contextualização Científica

O processamento de dados tem sido abordado de maneira sequencial desde os fundamentos da computação. Estes foram estabelecidos na década de 1930 por Alan Turing e Alonzo Church que, de maneira independente, formularam o que posteriormente foi nomeado de Tese de Church-Turing. Esta tese define o que pode e o que não pode ser computável, através do uso de um computador teórico chamado Máquina de Turing (ou o equivalente Cálculo Lambda de Church). [Herlihy and Shavit 2012]

Este trabalho se desenvolve no ambiente de computação paralela em memória compartilhada. Tal ambiente consiste em múltiplas tarefas executando em paralelo, cada uma lendo e escrevendo em uma memória compartilhada com as demais. Estas tarefas comportam-se de maneira assíncrona, de maneira que cada uma pode executar em velocidade diferente das demais, e ainda permanecer parada (sem progresso em sua execução) por tempo indeterminado. Este ambiente de execução assíncrona reflete a realidade de arquiteturas multiprocessadas modernas, onde uma tarefa pode parar por microsegundos (*cache misses*), milisegundos (*page faults*) ou ainda segundos (interrupções de escalonamento). [Herlihy and Shavit 2012]

Este capítulo apresentará conceitos de multiprocessamento, sincronização e memória compartilhada, assim como as implementações práticas relevantes a este trabalho encontradas na indústria e na academia. O texto será desenvolvido de forma a apresentar uma análise crítica sobre as atuais soluções.

2.1. Sincronização

Estratégias de sincronização têm por objetivo primário a garantia de corretude e coerência do sistema. Ou seja, seções críticas devem ser respeitadas para que dados não sejam corrompidos. A partir da garantia desta premissa, diferentes estratégias procuram garantir o progresso da execução das tarefas da melhor maneira possível. Tais estratégias dividem-se fundamentalmente nas seguintes três categorias:

- Sincronização *wait-free*: Cada tarefa progride em sua execução independente de qualquer fator externo. *Locks* ou barreiras de outras tarefas que impediriam a sua execução não estão presentes, e as operações são executadas em um número pré-determinado de passos. Sincronização *wait-free* é a garantia mais forte de progresso de execução de uma tarefa

- Sincronização *lock-free*: O sistema como um todo sempre progride em sua execução. A execução de cada tarefa, porém, não é garantida. Ou seja, tarefas podem ser impedidas temporariamente de executar, mas apenas em benefício da execução de outra tarefa do sistema. É uma garantia de progresso de execução mais fraca que a sincronização *wait-free*.

- Sincronização obstruction-free: Uma tarefa progride em sua execução apenas se não encontrar *lock* ou barreiras de outras tarefas. Ou seja, uma tarefa pode bloquear a execução de outra e, inclusive, ambas podem bloquear a execução uma da outras, impedindo que o sistema como um todo progrida. É uma garantia de progresso de execução mais fraca que a sincronização *lock-free*. [Vyukov 2014]

2.2. Processos e *Threads*

Em Sistemas Operacionais, tarefas são implementadas através de processos e *threads*. Um processo é uma abstração de um programa rodando no sistema operacional. É, na verdade, uma das abstrações mais antigas e fundamentais oferecidas por um sistema operacional. Através deles, o sistema consegue executar concorrentemente diversas tarefas, mesmo em um único processador.

Cada processo, por sua vez, possui seu espaço de endereçamento de memória e uma *thread* de controle que representa a sua linha de execução. Há situações, contudo, em que é desejável que um processo tenha mais de uma linha de execução no mesmo espaço de endereçamento de memória. A divisão da resolução de um problema na resolução de diversos problemas menores que, colaborativamente, constrõem a solução final, é um exemplo. *Threads* são utilizadas, portanto, para oferecer a um processo diversas linhas de execução com acesso à mesma região de memória. Esta habilidade de compartilhar a mesma memória e os respectivos dados é essencial para certas soluções, e é o maior diferencial entre o uso de *threads* e processos para processamento paralelo. Além disso, *threads* são mais leves e rápidas de serem criadas e excluídas, beneficiando assim processos com a natureza de variação periódica no número de subtarefas [Tanenbaum 2009]. Este trabalho foca-se em processamento paralelo em memória compartilhada através do uso de *threads*, e explora os benefícios e desafios do desenvolvimento de *software* para este cenário.

2.3. Memória Compatilhada

O compartilhamento do mesmo espaço de endereçamento de memória entre várias *threads* do mesmo processo oferece benefícios, como uma maior facilidade na cooperação das *threads* no processamento conjunto do dados, ou a leitura sempre atualizada de um valor possivelmente modificado por outra *thread*, por exemplo. Por outro lado, a manipulação concorrente de memória pode facilmente gerar dados inconsistentes. Uma simples função que lê uma variável, incrementa seu valor e grava-o de volta na mesma variável não é segura de ser executada paralelamente por duas *threads*. Caso ambas as *threads* leiam o valor ao mesmo tempo, incrementem-no e gravem-no de volta, teremos um resultado incorreto, incrementado apenas uma vez.

No exemplo acima, o trecho de código que corresponde à leitura e atualização do valor da variável é chamado de sessão crítica, e não pode ser executado por mais de uma *thread* ao mesmo tempo, do início do seu processamento até o final. Esta execução

indivisível da sessão crítica é chamada de execução atômica. O modelo de memória dos computadores atuais, porém, não garante a atomicidade destas execuções. Por isso, o processador oferece instruções atômicas nas quais são baseadas todas as implementações de primitivas de sincronização em ambientes de memória compartilhada.

2.4. Operações Primitivas de Sincronização

"A novice was trying to fix a broken Lisp machine by turning the power off and on. Knight, seeing what the student was doing spoke sternly: "You cannot fix a machine just by power-cycling it with no understanding of what is going wrong."
Knight turned the machine off and on.
The machine worked."
[*AI Koans*, uma coleção de piadas popular no MIT na década de 1980]

No desenvolvimento de programas sequenciais para processadores de apenas uma unidade de processamento, é normalmente seguro ignorar os detalhes de funcionamento da camada arquitetural de *hardware*. Ou seja, o desenvolvedor pode fazer um ótimo trabalho mesmo sem entender muito da arquitetura da plataforma para a qual ele está desenvolvendo. Desenvolvimento para multiprocessadores ainda não atingiu este estágio e, por enquanto, exige do desenvolvedor um entendimento detalhado da arquitetura de *hardware* da plataforma em questão. [Herlihy and Shavit 2012]

Para prover estratégias de sincronização de execuções paralelas e, principalmente, de interações paralelas com o modelo de memória compartilhada, multiprocessadores modernos precisam de estruturas em *hardware*, em nível arquitetural, que garantam a corretude destes acessos. Tais estruturas são chamadas de primitivas de sincronização, e servem da base para a implementação de todos os demais mecanismos de sincronização.

A principal primitiva de sincronizacão, oferecida por uma ampla variedade de arquiteturas, é a chamada *Compare-And-Swap*, normalmente acessível através de uma instrução assembly e respectivas funções em APIs de alto nível. Abreviada por CAS, esta instrução recebe 3 argumentos: um endereço de memória *a*, um valor esperado *e*, e um valor atualizado *v*. O suporte em *hardware* para tal primitiva garante a execução atômica dos seguintes passos:

- Se a memória do endereço *a* contém o valor esperado *e*,
- Escreva o valor atualizado *v* no endereço *a*, retorne *true*,
- Caso contrário, não altere a memória no endereço *a*, retorne *false*.

Em pseudo código:

```
function CAS(a, e, v)
    if content(a) = e then
        content(a) ← v
        return true
    else
        return false
    end if
end function
```

Através desta primitiva básica é possível ler o valor de uma variável, calcular seu valor atualizado baseado no valor lido, e então escrever o novo valor na variável apenas se esta não tiver sido modificada por outra *thread*. Esta primitiva é normalmente utilizada dentro de um *loop* onde, caso a instrução falhe (a variável foi modificada), seu valor é novamente lido e uma nova tentativa de atualização é feita. Estratégias mais complexas de sincronização utilizam desta primitiva para abstrair um *lock* em uma variável, por exemplo. Neste caso, uma *thread* obtém o *lock* escrevendo 1 na variável apenas se outra *thread* já não o tiver feito. A liberação do *lock* é feita escrevendo 0 na variável. Com o uso de CAS, apenas uma *thread* de cada vez poderá escrever o valor 1 na variável e, consequentemente, obter o *lock*.

Não raramente, outras primitivas de sincronização com suporte em *hardware* são oferecidas pela arquitetura, como *loads* e *stores* atômicos, *Test-And-Set* (escrita condicional) e *Fetch-And-Add* (adição), todas com garantia de execução atômica.

Primitivas de sincronização oferecem estas poderosas garantias em troca de um preço. Sua execução é mais dispendiosa ao sistema do que *loads*, *stores* e *if*s convencionais. O modelo de memória utilizado nos computadores modernos é fortemente baseado na execução sequencial de código. Para otimizar a execução deste tipo de código, diversas estratégias foram implementadas e, atualmente, dificultam a execução de código paralelo. Escritas à memória, por exemplo, são acumuladas para serem efetuadas em grupos. Como consequência, *threads* podem ler valores desatualizados que foram modificados por outras *threads*, mas ainda não escritos à memória. Compiladores também evoluíram e, através de análise de código, re-ordenam a execução de certas instruções afim de obter maior desempenho. Esta reordenação também pode causar confusão na sincronização de execuções paralelas sobre a mesma memória. [Herlihy and Shavit 2012]

Primitivas de sincronização, portanto, precisam utilizar estratégias que garantam a coerência da memória acessada apesar do ambiente hostil criado por outras otimizações. O custo preciso destas estratégias depende de muitos fatores, e varia não apenas entre arquiteturas, mas entre aplicações das instruções dentro da mesma arquitetura. É seguro dizer que estas instruções atômicas podem ser uma ordem de magnitude mais lentas que *loads* e *stores* convencionais [Herlihy and Shavit 2012]. Apesar de necessárias, seu uso deve ser racional para evitar que se tornem um gargalo para o sistema.

2.5. Exclusão Mútua

A análise de computação paralela está fortemente ligada a análise de tempo. Na maior parte das vezes queremos mais de uma *thread* executando ao mesmo tempo, afim de alcançarmos o melhor desempenho. Em outros momentos, porém, queremos que duas ou mais *threads* não executem o mesmo trecho de código ao mesmo tempo, ou ainda que executem em uma ordem esperada. Frequentemente temos que analisar como diferentes linhas de execução, ou intervalos de tempo, podem se sobrepor.

Exclusão mútua é um dos problemas mais fundamentais da área, e talvez uma das formas mais comuns de sincronização em programação de multiprocessadores. Sua ideia fundamental é impedir que duas *threads* executem uma sessão crítica ao mesmo tempo.

2.6. *Locks*

Locks são utilizados para garantir a exclusão mútua entre duas ou mais *threads* em um determinado trecho de código. A biblioteca POSIX *Threads* oferece APIs de *Locks* que trabalham com esta garantia de três maneiras diferentes. As três estratégias, contudo, controlam o *lock* em uma variável previamente inicializada, além de proverem funções para adquirir e liberar o *lock*.

- *Spin lock*: Utiliza a técnica de *spinning* para verificar se um *lock* está livre. Ou seja, em um laço *busy-wait*, a técnica continua fazendo solicitações de *lock* até que este esteja livre e seja adquirido por ela. É a técnica mais simples, e mantém a *thread* utilizando tempo de processamento para executar as tentativas. Portanto, é aconselhada para seções críticas extremamente curtas e cenários de pouca concorrência. Por outro lado, esta técnica exige menos variáveis de controle e, consequentemente, obtém e libera o *lock* de maneira leve e rápida.

- *Mutex lock*: Garante a exclusão mútua através de funções que obtém e liberam o *lock*. Porém, não mantém a *thread* em *busy-wait*. Quando uma *thread* tenta obter um *lock* que não está livre, esta para a sua execução na chamada da função de *lock*, e entra em modo *sleep*, podendo ser preemptada. Assim que o *lock* é liberado, as *threads* que estão em modo *sleep* esperando a sua liberação são avisadas e uma delas obtém o *lock*. Mutex *lock* evita que a *thread* consuma tempo de processamento enquanto espera o acesso à sessão crítica. Por outro lado, tem um custo de chamada de função maior que *Spin lock*, caracterizando a sua preferência para sessões críticas maiores.

- *Read Write lock*: Muitos problemas de concorrência permitem o acesso paralelo à sessão crítica por *threads* de leitura. Ou seja, *threads* que consultem mas não modifiquem valores podem acessar a memória compartilhada paralelamente. *Threads* de escrita, por outro lado, continuam exigindo exclusividade no acesso à memória, tanto em relação a outras *threads* de escrita, quanto a outras *threads* de leitura. *Read Write lock* é uma API que oferece exatamente estas garantias, através de funções específicas para *lock* de leitura e *lock* de escrita. Esta técnica é eficaz em cenários com grande concorrência de leitura, mas raras escritas, permitindo alcançar grandes níveis de paralelismo. Os eventos de escrita, por outro lado, apresentam maior *overhead*, devido ao controle que deve ser exercido sobre a sessão crítica, garantindo a exclusão inclusive das *threads* de leitura que possivelmente estão acessando-a naquele momento.

Locks e primitivas de sincronização são uma necessidade, e não um benefício. Seu objetivo é garantir a corretude na execução de um programa paralelo, e não o seu desempenho. De fato, um programa sem *locks* tem a liberdade de executar suas *threads* sem pausas, no ponto de vista de sincronização. *Locks*, quando mal utilizados, criam gargalos e reduzem o desempenho de um programa. O uso correto destas estratégias de sincronização, na verdade, envolve torná-las o menos perceptíveis possíveis na execução do programa, permitindo que as *threads*, sempre que possível, executem livremente.

Locks e primitivas de sincronização são ferramentas utilizadas em diferentes cenários de problemas e arquiteturas de *software*. Seu uso consciente, ou ainda eficiente, exige um entendimento profundo tanto do seu funcionamento, quanto da natureza do problema abordado. Nas próximas sessões, discutiremos os principais problemas no uso de *threads* e suas estratégias de sincronização, assim como algumas abordagens alternativas

para processamento paralelo.

3. Abordagens Alternativas

3.1. O Problema com *Threads*

"A folk definition of insanity is to do the same thing over and over again and expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs." [Lee 2006]

Segundo Edward A. Lee, *threads* são um modelo computacional fundamentalmente falho, pois seu comportamento é extremamente não determinístico. O trabalho do programador é reduzir este não determinismo através de ferramentas como *locks* e primitivas de sincronização.

Este ponto de vista defende a necessidade de construirmos modelos de computação paralela fundamentalmente determinísticos que, por sua vez, permitam a adição explícita, cuidadosa e criteriosa de elementos não determinísticos onde estes se fazem necessários. *Threads* seguem uma abordagem contrária, oferecendo um ambiente de execução altamente não determinístico, e confiando em estratégias de programação para limitar este comportamento e alcançar objetivos determinísticos.

Defensores deste ponto de vista especulam que a maioria dos programas concorrentes não triviais baseados em *threads* possuem *bugs* de sincronização. Segundo esta abordagem, tais *bugs* ainda não se apresentaram como graves falhas pois as arquiteturas e sistemas operacionais atuais oferecem níveis modestos de paralelismo. Nestes ambientes, apenas uma pequena parte das possíveis intercalações e execuções paralelas de instruções concorrentes realmente acontece na prática. Seguindo este raciocínio, à medida que novas arquiteturas oferecerem maiores níveis de paralelismo, estes *bugs* aparecerão. Como resultado, programas desenvolvidos utilizando o modelo não determinístico de *threads* irão falhar constantemente. [Lee 2006]

3.2. O Problema com *Locks*

Além de resultar em uma complexidade enorme de controle em programas não triviais, sincronização por *locks* acaba por apresentar uma série de armadilhas difíceis de serem detectadas até mesmo por programadores experientes. Entre as mais comuns estão:

- Inversão de prioridade: Ocorre quando uma *thread* de menor prioridade é preemptada enquanto possui um *lock* necessário por uma *thread* de maior prioridade. Neste caso, a *thread* de maior prioridade fica impedida de prosseguir a sua execução, pois necessita que o *lock* seja liberado.

- *Convoying*: Apresenta-se como um problema similar a inversão de prioridade, mas ocorre com *threads* de mesma prioridade. A principal consequência, neste caso, é a grande quantidade de trocas de contexto executadas pelas *threads* que tentam adquirir o *lock* em posse da *thread* inativa. Estas *threads* são empilhadas e, mesmo após a liberação do *lock*, pode levar algum tempo para desempilhá-las e atender as requisições de todas elas.

- *Deadlock*: Pode ocorrer quando *threads* tentam adquirir os mesmos *locks* em ordem diferente. Neste caso, cada *thread* espera pela liberação do *lock* em posse da outra

e, consequentemente, ficam trancadas neste ponto. Em ambientes com muitos *locks* e sem uma ordem pré-determinada para adquirí-los, evitar este problema pode ser difícil. [Herlihy and Moss 1993]

Organização, controle e manutenção de grandes sistemas baseados em *locks* é atualmente um grande problema. As associações entre dados e *locks* são majoritariamente estabelecidas apenas por convenções, e documentadas apenas por comentários no código fonte. [Herlihy and Shavit 2012]

As duas próximas sessões abordarão tecnologias que utilizam abordagens alternativas de sincronização, afim de evitar os problemas comuns à ambientes de execução paralela sincronizados por *locks*. Suas idéias fundamentais servirão de inspiração para o trabalho proposto neste documento, e seu posterior desenvolvimento.

3.3. *Read-Copy Update*

Read-Copy Update (RCU) é uma estratégia de sincronização que alcança altos níveis de escalabilidade em cenários de intensa leitura e moderada escrita. Fundamentalmente, RCU permite a execução concorrente de leituras, que nunca encontram ponto de contenção ou *lock*, juntamente com a presença de escritas ou atualizações. Este comportamente, que não exige a exclusão mútua das *threads* na sessão crítica, é alcançado através da seguinte estratégia: Sempre que uma atualização é realizada, o algoritmo mantém, enquanto necessárias, a cópia antiga e a cópia atualizada dos dados. Novas leituras, então, são direcionadas à cópia atualizada, enquanto leituras que ainda estejam em andamento acessam a cópia antiga. Assim que estas leituras forem finalizadas, a escrita é totalmente estabelecidade, descartando a cópia desatualizada. RCU utiliza mecanismos eficientes e escaláveis para controlar a publicação e leitura de múltiplas novas versões dos dados, enquanto libera gradativamente apenas a memória dos dados não mais acessados.

A ideia fundamental de RCU é simples, e evita gargalos criados pela exclusão mútua de múltiplos leitores, ou mesmo de leitores com escritores, comum em sessões críticas controladas por *locks*. Por outro lado, RCU possui um sistema de controle complexo que deve ser dominado e analisado de acordo com o cenário de aplicação. [McKeenney and Slingwine 1998]

3.4. Memória Transacional

Memória transacional também se distancia da abordagem de exclusão mútua utilizada pela sincronização baseada em *locks*. Com o objetivo de atingir melhores níveis de desempenho e maior facilidade de programação, este mecanismo se inspira na sincronização por transações, já comum em banco de dados.

Em memória transacional, ao invés de obter um *lock* antes de acessar uma sessão crítica, e liberá-lo após, *threads* utilizam uma abordagem diferente. Cada *thread* começa uma transação antes de qualquer modificação à memória, realiza estas modificações e, ao final, confirma a transação. Durante este processo, o sistema de memória transacional mantém registros de toda a memória que a *thread* leu e escreveu. Quando a transação é confirmada, o sistema certifica-se de que nenhuma outra *thread* fez qualquer modificação à memória utilizada pela transação. Neste caso, as modificações realizadas pela transação são aplicadas, e a *thread* continua sua execução. Caso outra *thread* tenha modificado a memória durante a transação, esta é abortada e suas modificações não

são aplicadas. Neste caso, a *thread* pode tentar a transação novamente, tentar uma abordagem diferente (por exemplo *locks*), ou desistir da operação. [Herlihy and Moss 1993] [Herlihy and Shavit 2012]

Bibliotecas que implementam memória transacional em *software* existem para diversas plataformas. Estas oferecem a facilidade de programação inerente a esta estratégia, porém causam *overhead* associado à camada de gerenciamento de memória acima do *hardware*. Soluções em *hardware*, mais eficientes, vem se tornando mais comum, e recentemente alcançaram o mercado de processadores comerciais.

4. Descrição do Trabalho

4.1. Implementação e Desenvolvimento

Os seguintes componentes serão implementados durante o desenvolvimento do Trabalho de Graduação

- Estrutura de dados em formato árvore: A estrutura será baseada em árvores binárias *crit-bit* (Radix Tree, Patricia Tree), que armazenam os objetos apenas nas folhas e, assim, oferecem uma distribuição bastante dispersa [Knizhnik 2008]. Esta distribuição abre espaço para a implementação de uma estratégia de sincronização de granularidade bastante fina e, portanto, alta escalabilidade.

- Mecanismo de sincronização: Afim de permitir a execução paralela e sem interrupções (*lock-free*) de operações de inserção, busca e exclusão de objetos, será implementado um mecanismo de sincronização. Este mecanismo, implementado diretamente no código-fonte da estrutura de dados, utilizará de uma composição de mecanismos de *lock* e primitivas de sincronização *Compare-And-Swap*.

- Programa de teste: Para testar o funcionamento da estrutura implementada, será desenvolvido um programa de teste que explora as suas operações. Tal programa testará a execução das operações em paralelo através da criação de várias threads que inserem, buscam e excluem objetos da árvore. Medições necessárias serão acrescentadas ao código-fonte deste programa, quando necessárias, para analisar o desempenho da estrutura segundo as métricas estabelecidas em 4.4.

4.2. Estruturas de Dados

Um projeto adequado das estruturas de dados é fundamental para se obter um processamento paralelo eficiente. A natureza distribuída da maioria destas estruturas oferece alto potêncial de aumento de desempenho para processamento multithread, permitindo que *threads* interajam paralelamente com partes diferentes da estrutura e, consequentemente, em áreas não conflitantes de memória. Por outro lado, *threads* podem também concorrer na interação com a mesma área de memória. Estratégias de sincronização, portanto, são fundamentais para a implementação de estruturas de dados *thread-safe* em ambientes de memória compartilhada.

O desempenho e, principalmente, e escalabilidade de uma estrutura de dados é fortemente dependente da estratégia de sincronização utilizada. Um *lock* global em uma árvore binária, por exemplo, pode se tornar um gargá-lo, evitando que *threads* que modifiquem áreas de memória não conflitantes executem em paralelo. Uma abordagem de maior granularidade, dividindo a estrutura em regiões ou nodos com *locks* independentes,

por exemplo, pode oferecer um maior nível de escalabilidade. Por outro lado, esta abordagem de granularidade fina pode oferecer maior complexidade e *overhead* de controle, assim como maior consumo de memória.

4.3. Abordagem de Sincronização

A estratégia de sincronização utilizará de uma composição de mecanismos de *lock* para exclusão mútua, assim como primitivas de sincronização como CAS, para operações atômicas. O comportamento da sincronização será inspirado em RCU, no que diz respeito a execução ininterrupta de leituras e escritas, utilizando mais de uma cópia dos objetos quando necessário. Os fundamentos de Memória Transacional também serão inspiração para a estratégia de sincronização desta implementação, no que diz respeito à execução prévia das operações e posterior aplicação destas na estrutura.

Através desta composição de mecanismos e algoritmos de sincronização, será desenvolvida uma estrutura de dados *thread-safe* escalável de alto desempenho. Baseando-se em tecnologias do estado da arte e foco em escalabilidade, espera-se observar um desempenho compatível com a demanda atual em ambientes multiprocessados. Para analisar o desempenho alcançado, utilizaremos das técnicas citadas na próxima sessão.

4.4. Avaliação de Desempenho

Para avaliar o desempenho e escalabilidade da implementação serão utilizadas métricas pertinentes a arquiteturas de processamento paralelo e estruturas de dados. Serão então feitos experimentos que testam as operações da estrutura comparativamente com outras estruturas de objetivos semelhantes presentes na literatura, com o objetivo de analisar os pontos fracos e fortes, e os cenários de melhor aplicação da estrutura proposta neste trabalho. As seguintes métricas de avaliação serão medidas e analisadas:

- Operações por segundo: Uma das métricas de desempenho mais simples para uma estrutura de dados é a taxa com que se consegue inserir, buscar e excluir objetos. Através da implementação de um simples programa que força a execução de milhares de operações na estrutura, mediremos a taxa de inserções, buscas e remoções alcançada.

- IPS (*Instructions per Second*): Também com o objetivo de medir o desempenho da estrutura em um ambiente de execução paralela, mediremos a taxa de instruções executadas por segundo em um cenário forçado de inserções, buscas e exclusões paralelas. Para analisar o efetivo uso das unidades de processamento, realizaremos tal experimento com diferentes números de *threads*. Dessa maneira, observaremos a variação da taxa de instruções executadas afim de avaliar o real aumento do uso das unidades de processamento.

- Instruções por Operação: Afim de medir a eficiência de uma única operação de inserção, busca ou exclusão, mediremos a quantidade de instruções do processador necessárias para completar cada operação.

- Taxa de falhas do *Compare-and-Swap*: A instrução atômica CAS apenas falha caso outra *thread* tenha modificado a mesma área de memória no mesmo período. Dessa maneira, a taxa de falhas das intruções CAS no código representa uma maneira de medir o nível de concorrência presente em determinadas regiões do código.

- Taxa de *busy-time* do processador: Também com o objetivo de medir o quanto a estratégias de sincronização da estrutura é capaz de manter as unidades de execução

ocupadas, utilizaremos ferramentas de monitoramento do processador para medirmos a relação de *busy-time* e *idle-time* do processador.

- *SpeedUp*: Esta medida relaciona o ganho de velocidade de execução de um algoritmo paralelo à sua versão sequencial. Variando o número de *threads* em execuções de simples testes de inserção, busca e exclusão de objetos na estrutura, poderemos medir o *SpeedUp* da estrutura de dados. A Lei de Amdahl especifica que a curva de *SpeedUp* de um algoritmo, à medida que aumenta-se o número de *threads*, sempre demonstra um comportamento logarítmico. Ou seja, a partir de certo ponto, o aumento do número de *threads* não oferecerá aumento considerável de desempenho. Faremos as medidas de *SpeedUp* com diferentes número de *threads* a fim de encontrar esta curva de comportamente para as operações da estrutura proposta. [Amdahl 1967]

4.5. Ferramenta de Medição

Para a obtenção das métricas citadas em 4.4, será utilizado um *framework* de coleta e análise de dados de performance chamado *Performance Counters for Linux* (PCL). Comumente chamado de *perf*, *perf tools* ou *Perf Events*, esta ferramenta é suportada pelo *kernel* do Linux desde a sua versão 2.6.31.

Processadores modernos possuem contadores em *hardware* que monitoram diversos eventos. *perf* acessa tais contadores e analisa seus dados. Entre outros, a ferramenta é capaz de disponibilizar dados como ciclos do processador, instruções executadas, *page faults*, trocas de contexto, *cache loads*, *cache misses*, migrações de threads entre CPUs, *branches* e *branque misses*. Tais medidas podem ser relacionadas à execução do sistema, de um processo ou ainda de uma *thread* em específico.

Por ser implementada no *kernel* do Linux, e por acessar contadores e monitores implementados em *hardware*, *perf* tem um comportamento não intrusivo e, consequentemente, permite a obtenção de medidas reais da execução do programa. [RedHat 2014], [PerfWiki 2014]

As medidas serão obtidas pelo *perf* na execução do programa de teste, citado em 4.1, e seu uso será fundamental para o desenvolvimento deste trabalho.

5. Cronograma de Atividades

O seguinte cronograma procura organizar o desenvolvimento do trabalho proposto neste artigo. Ao longo dos próximos meses, o trabalho será desenvolvimento de acordo com as seguintes etapas:

Abril de 2014 - Implementação da estrutura de dados proposta em 4.2.

Maio de 2014 - Experimentação e avaliação de desempenho baseada nas métricas propostas em 4.3.

Junho de 2014 - Desenvolvimento da monografia final de Trabalho de Graduação.

Referências

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference*.

Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: Architectural support for lock-free data structures. *20th ISCA*.

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

Knizhnik, K. (2008). Patricia tries: A better index for prefix searches. *Dr. Dobb's Journal*.

Lee, E. A. (2006). The problem with threads. *IEEE Computer Society*.

McKeenney, P. E. and Slingwine, J. D. (1998). Read-copy update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*.

PerfWiki, W. (2014). *perf: Linux profiling with performance counters*. https://perf.wiki.kernel.org/index.php.

RedHat, L. (2014). *Performance Counters for Linux (PCL) Tools and perf*. https://access.redhat.com/site/documentation.

Tanenbaum, A. S. (2009). *Modern Operating Systems*. Pearson Prentice Hall, 3rd edition.

Vyukov, D. (2014). *1024 Cores*. http://www.1024cores.net.