ANALISE DO COMPORTAMENTO DE MECANISMOS DE ACESSO A DADOS COMPARTILHADOS.

Michael Alexandre Costa[1*]; Gerson Geraldo H. Cavalheiro[1]; André Rauber Du Bois[1]; Maurício Lima Pilla[1]

[1]Universidade Federal de Pelotas – {macosta, gerson.cavalheiro, dubois, pilla}@inf.ufpel.edu.br

1. INTRODUÇÃO

As arquiteturas paralelas são praticamente onipresentes nas plataformas computacionais modernas. Processadores convencionais com múltiplos *cores* são usados para construção de computadores domésticos e super computadores. O aumento do paralelismo contido nestes processadores tendem a aumentar com passar dos anos, visto que, o aumento de desempenho das arquiteturas atuais baseiam-se no desenvolvimento de arquiteturas paralelas.

Para extrair o máximo de desempenho das arquiteturas paralelas, o código deve explorar todo o poder computacional, porém desenvolver programas para estas arquiteturas acarretam em um esforço maior do programador, para isto, existem ferramentas que auxiliam no controle do fluxo de execução e o controle dos dados na memória.

Programas paralelos podem apresentar áreas de memórias compartilhadas, isto é, várias *threads* ou processos podem acessar e manipular os mesmos dados concorrentemente com leituras e escritas (COSTA, 2015), estas áreas são denominadas seções críticas. Para manipular dados em uma seção crítica pode ser utilizado métodos de sincronização, estes métodos utilizam regime de exclusão mútua, onde apenas uma *thread* pode acessar o dado por vez, assim, mantendo a coerência dos dados na memória.

Em programas concorrentes, uma das formas de garantir a sincronização do acesso à memória compartilhada por meio de exclusão mútua é a utilização de um *mutex*, este é um objeto usado para representar o direito exclusivo de acessar algum recurso (STROUSTRUP, 2013). Para acessar o recurso, deve-se adquirir o *mutex*, utilizar o dado, e em seguida liberar o *mutex*. O principal problema relacionado ao uso deste método está na ocorrência de *deadlock*, onde a região bloqueada não é liberada impedindo o termino do programa.

Para evitar o regime de exclusão mútua, tem-se como alternativa algoritmos não bloqueantes (*Lock Free)*, estes algoritmos garantem que se uma *thread* realizar uma operação sobre um dado compartilhado esta operação será completada em um número finito de passos independentes das ações das outras *threads*.

Uma instrução atômica é uma sequência de uma ou mais instruções de máquina que executam sem interrupção. Primitivas atômicas podem ser utilizadas para construção de algoritmos *lock-free*. Quando uma *thread* realiza uma operação atômica, esta operação aparenta ocorrer instantaneamente para outras *threads*.

Memórias transacionais (TM) são um novo método de sincronização que vem sendo estudado. TMs apresentam como vantagem um alto nível de

* Bolsista de Iniciação Cientifica FAPERGS

programação onde o desenvolvedor não preocupa-se com aquisição e liberação de locks para garantir a consistência dos dados na memória.

Este trabalho apresenta o comportamento dos métodos descritos acima com operações realizadas sobre grafos em arquiteturas paralelas. Para isto foram desenvolvidas quatro bibliotecas que dão suporte paralelo as operações de um grafo, estas operações utilizam *mutex com lock de grão fino e com lock de grão grosso*, operações atômicas e memórias transacionais.

2. METODOLOGIA

O trabalho foi desenvolvido em C++ e apresenta quatro bibliotecas distintas para manipulação de um grafo concorrente, este grafo é apresentado como uma matriz de adjacência, na qual possui vetor de inteiros para indicar e existência de nós. É possível manipular as arestas assim como os nós do grafo, para isto foi implementado os métodos: *insertNode; deletNode; testNode; decreaseEdge; increaseEdge; changeEdge; getVal*.

Os métodos *increseNode* e *deletNode* manipulam o vetor de nós que é representado por 1 se o nó existir no grafo e 0 se este não possuir nenhuma aresta, *testNode* verifica a existência de um nó no grafo.

Para manipular os valores das arestas apresentadas na matriz de adjacência é utilizado *increaseEdge* e *decreaseEdge* que incrementam e decrementam o valor 1 da posição V*ij* da matriz, se a posição *Vij* possuir valor 0 esta aresta não existe, assim, os nós não estão conectados, se um destes nós não possuem mais nenhuma aresta, este nó pode ser removido por meio do *deletNode*.

O método *changeEdge* cria a aresta e os nós caso estes ainda não estiverem em uso, o grafo apresenta valores positivos para suas arestas, não permitindo valores negativos.

Para a manipulação do grafo concorrente, foram implementadas as bibliotecas utilizando *mutex* com lock de grão fino e com lock de grão grosso, também foi implementada uma biblioteca que utiliza o método *atomic* do C++ 2011 e uma biblioteca utilizando memórias transacionais em software (STM) com a biblioteca de STM TinySTM.

A implementação de *mutex* foi realizada de duas maneiras, para analisar o comportamento que um lock de grão fino apresenta comparando-a com um lock de grão grosso.

Para o lock de grão fino, foi utilizado uma matriz de *mutex*, onde é bloqueado o acesso da posição que será modificada no momento do incremento ou decremento de uma aresta. Para realizar a remoção de um nó, adquirisse o bloqueio de todas as arestas que o nó está utilizando para garantir a integridade do grafo.

Para o lock de grão grosso, foi utilizado um *mutex* para todo grafo, onde é parado todo grafo para realizar a remoção de um nó, e são bloqueados os métodos que operam sobre as arestas para incremento ou decremento de uma aresta.

Os testes consistem em incrementar e decrementar arestas a partir de um arquivo de entrada que indica os nós a serem criados, após a inserção dos dados no grafo é realizado buscas no grafo. São realizadas 30.000.000 operações sendo divididas em 15.000.000 incrementos 7.500.000 decrementos e 7.500.000 buscas em uma matriz de adjacência de tamanho 50x50.

Os testes foram realizados em uma máquina com processador Intel Core i7 2600 com 4 *cores* físicos e 8 *cores* lógicos, com 8 Gb de memória RAM. Foram efetuadas 30 execuções para cada biblioteca em cenários com 2, 4, 6, e 8 threads. No melhor caso a variação dos resultados definida pelo desvio padrão dividido pela média dos resultados foi praticamente nula (0%), que ocorreu no teste Mutex lock de grão fino com 8 threads, e no pior caso, foi de aproximadamente 9%, que ocorreu no teste Atomic com 2 threads.

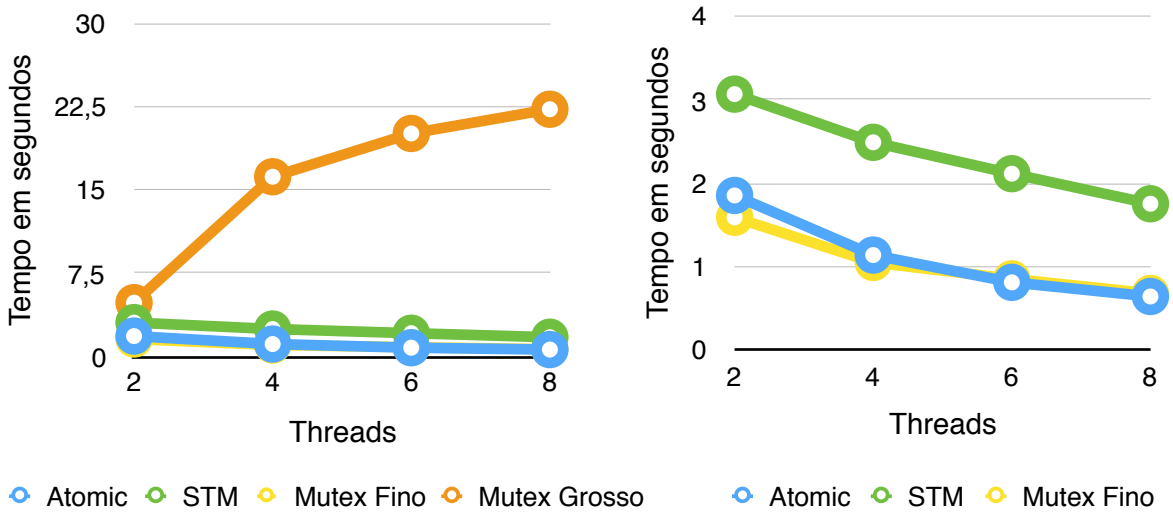
3. RESULTADOS E DISCUSSÃO

Os resultados apresentaram valores distintos para cada teste, como pode ser visto na Figura 1 (b), os testes Atomic, STM e Mutex com lock de grão fino (Mutex Fino) apresentam escalabilidade nos resultados. Já o teste Mutex com lock de grão grosso (Mutex Grosso), Figura 1 (a), apresenta uma piora no desempenho conforme aumenta o número de threads, isto ocorre devido ao aumento da concorrência pelo número de threads.

O Mutex Grosso, Figura 1 (a), mostra este resultado visto que, o mutex com lock de grão grosso bloqueia todo o grafo para inserir ou remover arestas gerando a serialização do código, assim, o desempenho da execução mostra-se pior com o aumento da contenção.

O teste Atomic com 8 threads possui o melhor desempenho de tempo de execução sendo, 63% menor que o teste STM com 8 threads e 5% menor que o teste Mutex Fino, quando comparado ao teste Mutex Grosso o teste Atomic apresenta uma melhora de 97% no tempo de execução.

Como pode ser visto na Figura 1 (b), o teste Atomic apresenta pior desempenho comparado ao teste Mutex Fino para 2 e 4 threads, para 6 e 8 threads o Atomic apresenta um desempenho de tempo de execução melhor. Este desempenho é apresentado visto que, com o aumento do paralelismo o mutex com lock de grão fino apresenta overhead ao adquirir e liberar os locks.



(a) Média de tempo com mutex grosso

(b) Média de tempo das execuções

FIGURA 1: Gráficos das médias de tempo do grafo de tamanho 50x50

4. CONCLUSÕES

Neste trabalho foi implementada quatro bibliotecas que operam sobre grafos em arquiteturas paralelas com as principais soluções para o controle de memória compartilhada, assim pode-se analisar e comparar o desempenho das atuais ferramentas de sincronização e seu comportamento em diferentes situações.

Os resultados mostram que os testes Atomic, STM e Mutex Fino apresentam melhor desempenho com o aumento do número de threads, sendo que entre estes testes STM apresentou para todos os cenários de threads um pior desempenho no tempo de execução.

O teste Atomic com 8 threads mostrou o melhor desempenho de tempo, sendo aproximadamente 63% melhor que o teste STM com 8 threads. A solução implementada com memórias transacionais possui uma interface de programação de alto nível, e apresenta melhor desempenho que o teste Mutex Grosso.

A biblioteca com Mutex Grosso possui o pior desempenho comparado as outras implementações e apresenta perda de desempenho com o aumento do número de threads.

A solução com Mutex Fino apresenta desempenho comparável com a implementação que utiliza operações atômicas, porem esta apresenta uma interface complexa de programação, deixando a cargo do desenvolvedor o controle das sincronizações.

Para trabalhos futuros pretende-se analisar operações de busca do melhor caminho no grafo realizado pelo algoritmo de Dijkstra, também estender os testes para aumentar o tamanho do grafo e número de entrada de dados. Outro trabalho futuro que pretende-se realizar é a implementação de uma biblioteca utilizando STM disponibilizada no C++, assim comparando-a com a biblioteca implementada neste trabalho que utiliza a TinySTM.

5. REFERÊNCIAS BIBLIOGRÁFICAS

COSTA, M.A. Avaliação de Tempo do Benchmark Lee-TM Adaptado a TinySTM. **XXIV Congresso de Iniciação Científica da Universidade Federal de Pelotas**, Pelotas, 2015.

STROUSTRUP, B. **The C++ Programing Language**. Boston: Addison-Wesley, 2013. 4v.