Utilização de Estruturas de Dados não-bloqueantes em programação multiprocessada*

Guilherme P. B. Cousin†, Alan S. Araujo‡,
Cícero Augusto de S. Camargo§, Gerson Geraldo H. Cavalheiro

Centro de Desenvolvimento Tecnológico
Bacharelado em Ciência da Computação
Universidade Federal de Pelotas (UFPEL) Pelotas – RS – Brasil

{gpbcousin,asdaraujo,cadscamargo,gerson.cavalheiro}@inf.ufpel.edu.br

Resumo. *Este trabalho aborda o problema de reduzir o de uso de seções críticas para manipulação de listas encadeadas. A expectativa é que, implementando um algoritmo* lock-free*, seja reduzida a contenção de execução, eliminando este limitador de paralelismo.*

1. Introdução

Processadores *multicore* popularizaram o uso de arquiteturas paralelas em praticamente todos sistemas computacionais. Como consequência, a programação paralela (ou concorrente) tornou-se uma necessidade no desenvolvimento de programas para exploração eficiente deste tipo de arquitetura. Oferecer recursos de programação de alto nível que permitam explorar estas arquiteturas passou a ser uma necessidade nos ambientes de desenvolvimento atuais. Neste trabalho o enfoque é o projeto e a implementação de uma estrutura de dados não bloqueante para Anahy [Cavalheiro et al. 2006].

Programas paralelos se beneficiam do hardware paralelo quando são capazes de manter os processadores desta arquitetura em operação a maior parte do tempo. A ociosidade de um processador, enquanto o programa não terminou, representa perda potencial de desempenho e desperdício do investimento realizado no hardware adquirido. Sincronização para controlar a entrada em seções críticas para acesso a dados compartilhados podem, potencialmente, serializar a execução de um programa. Muitas seções críticas, portanto, podem reduzir consideravelmente a capacidade de execução paralela do programa e, consequentemente, reduzir a capacidade de aproveitamento do hardware disponível.

Este trabalho aborda o problema de reduzir o de uso de seções críticas para manipulação de listas encadeadas. O objetivo é conceber uma estrutura de lista para o ambiente de execução Anahy com vistas a reduzir o *overhead* de operação do escalonador na manipulação das *threads* geradas pelo programa aplicativo. Embora existam diversos trabalhos na área, como [Fomitchev and Ruppert 2004] e [D. Hendler and Yerushalmi 2004], a presente pesquisa se diferencia por buscar uma

*FAPERGS/PqG (11/1065-1), PRONEX/FAPERGS/CNPq (10/0042-8)
†Bolsista PIBIC/CNPq
‡Bolsista PIBIC/CNPq
§Bolsista CAPES

solução eficiente especifica para as necessidades de Anahy. A expectativa é que, implementando um algoritmo *lock-free*, seja reduzida a contenção de execução, eliminando este limitador de paralelismo.

2. Conceito de estrutura de dados não bloqueantes

Programas são feitos para manipular dados. Os dados em um programa devem ser organizados de forma a permitir sua manipulação nos diversos trechos de código. É comum que grandes coleções de dados estejam armazenadas em conteiners de dados, como vetores ou listas. A otimização da manipulação destas estruturas tem impacto direto no desempenho de um programa. Considerando a programação paralela, um dos aspectos a serem otimizados é a contenção dos *threads* no acesso a listas compartilhadas para inserir ou retirar elementos.

Os algoritmos não-bloqueantes baseiam-se no princípio de que conflitos de sincronização são raros e devem ser tratados como exceção. Quando um conflito é detectado em uma estrutura, a operação que detectou o conflito é abortada e sua execução é refeita. Algoritmos não-bloqueantes de sincronização evitam *deadlocks* e inversões de prioridade. Estes algoritmos devem ser, quando implementados, tolerantes a falhas de *threads* e robustos o suficiente para tratar *page faults* e *cache misses*. Como consequência o programa em execução não sofrerá degradação de desempenho por preempção, forçada pela chamada de uma operação de sincronização executada em nível sistema.

Algoritmos não-bloqueantes são pouco utilizados na prática, pois são de implementação complexa que inclui diversas limitações de execução sobre o *hardware* [Kunz 2010]. Entretanto, estes algoritmos podem ser adequados para aplicações paralelas de larga escala devido ao seu potencial de execução de alto desempenho.

Na fase atual do trabalho, estão sendo estudados algoritmos não-bloqueantes para manipulação de listas encadeadas. Lista é uma estrutura onde as operações inserir, retirar e localizar são definidas com uma política especifica. Listas dependendo do tipo da implementação podem ser estruturas muito flexíveis, porque podem crescer ou diminuir de tamanho durante a execução de um determinado programa, Em uma implementação paralela, itens podem ser acessados, inseridos ou retirados deste tipo de estrutura por diferentes *threads* de forma concorrente.

Listas são adequadas para aplicações onde não é possível prever a demanda por memória, permitindo a manipulação de quantidades imprevisíveis de dados, de formato também imprevisível. Estas são úteis em aplicações tais como manipulação simbólica, gerência de memória, simulação e compiladores. Na manipulação simbólica os termos de uma fórmula podem crescer sem limites. Em simulação dirigida por *clock* pode ser criado um número imprevisível de processos, os quais têm que ser escalonados para execução de acordo com alguma ordem predefinida.

A Figura 1 apresenta as estruturas de lista simplesmente encadeadas utilizadas em implementações *multithread*. A estrutura da esquerda representa uma implementação *thread-safe* clássica, onde um *mutex* permite o acesso à toda a lista em exclusividade para os diferentes *threads* que a manipulam. A estrutura da direita é uma versão da lista para algoritmos não bloqueantes. Neste caso, cada nó da lista possui um campo adicional contendo dados de controle utilizados pelo algoritmo de manipulação. O problema

Figura 1. A esquerda uma lista com bloqueio e na direita uma lista com estrutura não-bloqueante

básico na programação de uma lista *lock-free* é quando um *thread* tenta eliminar um nó, com uso de uma instrução atômica CAS*(Compare-and-Swap)* no nó, tem que existir uma garantia na qual o ponteiro da direita deste nó não seja alterado por uma execução paralela, pois isto causaria uma inconsistência nesta lista. Este problema é denominado *ABA probelm* [Valois 1995].

Para ilustrar este problema, o seguinte cenário pode ser montado: um processo ao tentar retirar um item de uma fila, implementada como uma estrutura não bloqueante, lê as informações na cabeça desta estrutura e determina a área de memória que contém o segundo elemento. Então usa CAS para atualizar o ponteiro da cabeça para o segundo elemento. Caso a cabeça tenha sido alterada por um segundo processo, após o primeiro ter lido as informações, a instrução CAS falhará e corromperá a estrutura. Este corrompimento da estrutura durante operações de leituras inconsistentes define o problema ABA.

Uma solução convencional deste problema é usar uma variante da instrução CAS, onde são mantidas duas *flag* ao mesmo tempo, uma é usada para manter o ponteiro para outro nodo adjacente na lista e a outra é usada para manter uma *tag* que é incrementada toda vez que o ponteiro é alterado. Desta forma, se um processo alterar o ponteiro e em seguida alterar seu valor pra o original, a *tag* será capaz de indicar esta mudança e, então, a operação CAS não será concluída para não corromper a estrutura [Valois 1995]. Na figura 2 apresenta um exemplo de ocorrência do ABA *problem*

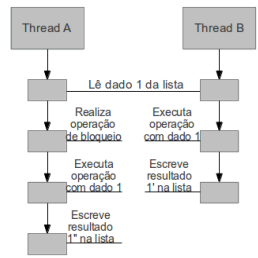

Figura 2. Exemplo de ocorrência do ABA *problem*

3. Recursos para implementação

As arquiteturas modernas disponibilizam diversas instruções que permitem a manipular atomicamente os dados de uma estrutura. Na programação concorrente a utilização de instruções atômicas é bastante importante pois uma instrução deste tipo, garante que a lista de execução da determinada instrução não vai ser interrompida. Desta forma impede-se a preempção na instrução, pois a implementação destas instruções são feitas em *hardware*.

Tabela 1. Instruções atômicas da família de processadores *Itanium* da Intel

Instrução	Chamada	Função
Fetch add	*type _sync_fetch_and_add (type *ptr, type value, ...)*	Faz uma soma
Fetch sub	*type _sync_fetch_and_sub (type *ptr, type value, ...)*	Faz uma subtração
Fetch or	*type _sync_fetch_and_or (type *ptr, type value, ...)*	Faz a operação *or* logica
Fetch and	*type _sync_fetch_and_and (type *ptr, type value, ...)*	Faz a operação *and* logica
Fetch xor	*type _sync_fetch_and_xor (type *ptr, type value, ...)*	Faz a operação *xor* logica
Fetch nand	*type _sync_fetch_and_nand (type *ptr, type value, ...)*	Faz a operação *nand* logica
Compare and Swap	*type _sync_val_compare_and_swap (type *ptr, type oldval, type newval, ...)*	Comparação atômica e troca
Test and set	*type _sync_lock_test_and_set (type *ptr, type value, ...)*	Testa, bloqueia e troca
Lock release	*void _sync_lock_release(type * ptr, ...)*	Libera bloqueio do *test and set*

A Tabela 1 apresenta algumas das instruções atômica da família de processadores *Itanium* da Intel no compilador GCC, destas instruções serão utilizadas nesta implementação serão o *Compare-and-Swap* e o *fetch-and-add*.

4. Conclusão

A estrutura a ser implementada no ambiente de execução Anahy, é uma variação de uma estrutura de lista, sendo manipulada como características de uma *heap* onde os elementos só podem ser inseridos no final da lista, e a remoção pode ser feita tanto no início, quanto no final. O objetivo com o uso desta estrutura é minimizar os custos de gestão dos *threads* criados durante a execução do programa aplicativo.

A implementação desta estrutura está em curso e será avaliada uma aplicação sintética onde nodos serão inseridos e removidos da lista por um número variado de *threads* e comparar as implementações da lista tradicional com esta implementação realizada, tendo como objetivo criar uma biblioteca e a incorporação desta, no núcleo de escalonamento de Anahy, um ambiente de execução para o processamento de alto desempenho.

Referências

Cavalheiro, G. G. H., Gaspary, L. P., Cardozo, M. A., and Cordeiro, O. C. (2006). Anahy: A programming environment for cluster computing. In *VECPAR*, pages 198–211.

D. Hendler, N. S. and Yerushalmi, L. (2004). A scalable lock-free stack algorithm. In *Proc. of the 16th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 206–215.

Fomitchev, M. and Ruppert, E. (2004). Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 50–59, New York, NY, USA. ACM.

Kunz, L. (2010). Memória transacional em hardware para sistemas embarcados multiprocessados conectados por redes-em-chip. Master's thesis, URGS, Porto Alegre.

Valois, J. D. (1995). Lock-free linked list using compare-and-swap. In *In Porceding of the 14th Annual ACM Symp. on Principles of Distributed Computind*, pages 214 – 222.