MEMÓRIAS TRANSACIONAIS EM OPENMP

ANDRÉ D. JARDIM[1]; ANDRÉ DU BOIS[2]; GERSON GERALDO H. CAVALHEIRO[3]

[1]*Universidade Federal de Pelotas – andre.jardim@inf.ufpel.edu.br*
[2]*Universidade Federal de Pelotas – dubois@inf.ufpel.edu.br*
[3]*Universidade Federal de Pelotas – gerson.cavalheiro@inf.ufpel.edu.br*

1. INTRODUÇÃO

Na programação concorrente, a sincronização é um dos aspectos mais desafiadores, referindo-se a relacionamentos entre eventos de forma que estes tenham conhecimento mútuos sobre seus estados de execução. As restrições de sincronização, como a serialização (o evento B deve seguir o evento A) e a exclusão mútua (os eventos A e B não podem acontecer ao mesmo tempo), tradicionalmente são expressas utilizando mecanismos baseados em *semáforos binários*. Porém, a programação empregando mecanismos explícitos de sincronização é propensa a erros, complexa e contra intuitiva (WILLIAMS, 2012).

Sob o ponto de vista dos processadores multicore, semáforos binários, usualmente chamados *mutexes*, limitam o potencial de exploração de paralelismo. O uso de mutexes implica em obtenção do direito de avançar sobre uma região de código protegida, região esta denominada de seção crítica. A execução sobre esta região é crítica por manipular recursos compartilhados. Obter um mutex implica em ter direito ao uso, em regime de exclusão mútua, do recurso compartilhado, introduzindo um gargalo serial quando dois ou mais threads competem pelo mesmo recurso. Qualquer tentativa de acesso, ou seja, de obtenção do mutex, à região crítica enquanto ela estiver ocupada implica em bloqueio do thread solicitante, mesmo que o acesso a ser realizado não cause qualquer inconsistência ao resultado computado pelo thread detentor do mutex (RIGO et al, 2007)

À medida que os programas multithread aumentam em tamanho e complexidade, abstrações mais avançadas são necessárias para abrandar a complexidade da programação que naturalmente decorre do uso frequente da sincronização em sistemas de software em larga escala (WONG et al., 2014)

Nesse cenário, as Memórias Transacionais surgiram como uma alternativa promissora aos mecanismos de sincronização baseados em mecanismos de exclusão mútua. Elas fornecem uma nova construção de controle de sincronização que evita problemas comuns de bloqueios e simplifica significativamente o esforço de programação para produzir o software correto. O esforço de pesquisa se concentra em introduzir esta nova abstração em ferramentas tradicionais para programação multithread, como OpenMP.

Neste trabalho é apresentado um estudo, tendo como foco Memória Transacional para o OpenMP, os conceitos principais das respectivas áreas de estudo, a fim de a fim de verificar a viabilidade do uso de memória transacional em OpenM, buscando a melhora de desempenho na programação multithread.

2. METODOLOGIA

A exploração de concorrência em sistemas de computação se dá com a execução simultânea de diversos fluxos de execução sobre os recursos de uma mesma arquitetura. Quando relacionado ao desenvolvimento de aplicações, o termo concorrência é associado a aspectos ligados à descrição de atividades concorrentes (CAVALHEIRO; DU BOIS, 2014). O controle de concorrência visa

gerenciar o acesso a recursos compartilhados. O objetivo é controlar como múltiplos acessos podem utilizar um recurso sem conflitos, pois o acesso concorrente simultâneo pode gerar inconsistência dos dados. Tal controle é importante, pois o acesso concorrente a dados e recursos partilhados pode criar uma situação de inconsistência desses mesmos recursos. Para que uma rotina ou programa seja consistente são necessários mecanismos que assegurem a execução ordenada e correta dos processos cooperantes.

A especificação de OpenMP (*Open Multi-Processing*) (DAGUM; MENON, 1998) define uma interface de programação, com o objetivo de explorar a concorrência em programas C e Fortran, voltada para ambientes de memória compartilhada. Uma vez que possui variáveis de ambiente, biblioteca de funções e um conjunto de diretivas de compilação, a ferramenta possibilita que o programador expresse a concorrência e controle o paralelismo de sua aplicação (CHAPMAN et al, 2007).

Em OpenMP, regiões paralelas definem seções do código em que a concorrência é apresentada de forma explícita. A concorrência descrita nestas regiões paralelas permite a criação de tarefas, que são manipuladas pelo núcleo de escalonamento OpenMP e após escalonadas sobre os threads disponíveis. Cada thread executa o mesmo código, mas sobre um conjunto diferente de dados. Utiliza o modelo *fork/join*: novos threads são criados com *fork* para dividir o trabalho, criando regiões paralelas; quando o processamento finaliza, os threads são sincronizados com *join*, e somente o thread mestre segue o fluxo de execução (CHANDRA et al, 2001).

Diretivas possuem a forma: `#pragma omp <diretiva> [cláusulas]`. Nesta sintaxe, `#pragma omp` instruí o pré-processador que uma diretiva OpenMP será expandida, para que seu código correspondente seja gerado. Uma diretiva é formada por um ou mais comandos, podendo ser seguida de cláusulas opcionais. Uma variável pode ser do tipo compartilhada (`shared`) ou privada (`private`). Variáveis privadas são de uso específico de cada thread, não sendo iniciadas. Variáveis compartilhadas são visíveis por todos os threads que executam o código, por isso o acesso às mesmas deve ocorrer de maneira organizada e sincronizada a fim de evitar condições de corrida. Quando uma condição de corrida acontece, o resultado depende da ordem de execução dos threads, não havendo garantia que a variável seja atualiza com o valor correto. portanto, diretivas de garantem que o acesso ou atualização de uma determinada variável compartilhada aconteça no momento certo (CHAPMAN et al, 2007).

Em um programa OpenMP threads podem se comunicar através de operações regulares de leitura/escrita em variáveis no espaço de endereçamento compartilhado. Embora a comunicação em um programa OpenMP seja implícita, geralmente é necessário coordenar o acesso a variáveis compartilhadas por múltiplos threads, a fim de garantir a execução correta.

Memória Transacional é um mecanismo de sincronização alternativo que tem como base, para garantir sincronismo entre threads concorrentes, o conceito de transação. Uma transação consiste em uma sequência de instruções com garantia de atomicidade e isolamento. Durante sua execução, uma transação armazena localmente os acessos de leitura e escrita feitos aos dados compartilhados. Caso não ocorra nenhum conflito, torna visível suas alterações locais para o restante do sistema. Caso contrário, a transação é cancelada, suas alterações locais são descartadas e sua execução reiniciada. O uso de Memória Transacional facilita a programação multithread, pois o programador não precisa se preocupar em garantir a sincronização. Todo o controle de acesso à memória compartilhada é realizado automaticamente (RIGO et al., 2007)

A implementação do mecanismo de Memória Transacional pode ser realizada exclusivamente em software (Memória Transacional de Software), ou contar com suporte em hardware (Memória Transacional de Hardware). Ambas abordagens devem fornecer (HARRIS et al, 2010): (i) armazenamento especulativo dos dados; (ii) detecção e resolução de conflitos entre transações; e (iii) efetivação e cancelamento atômicos. As abordagens diferem no modo como esses requisitos são implementados: Memórias Transacionais de Software empregam bibliotecas, compiladores e/ou sistemas de execução, Memórias Transacionais de Hardware modificam componentes da arquitetura (barramento, cache, CPU, protocolo de coerência). Abordagens híbridas também existem.

3. RESULTADOS E DISCUSSÃO

O advento dos microprocessadores multicore de memória compartilhada criou uma abertura para explorar o paralelismo em nível de thread. Na maioria dos aplicativos, a execução de threads paralelos requer mecanismos de sincronização ou pedido para acessar dados compartilhados. Os modelos tradicionais de programação multithreaded geralmente oferecem um conjunto de primitivas de baixo nível, como mutexes, para garantir a exclusão mútua. No entanto, os mutexes são complexos de usar e propensos a erros, especialmente quando um programador está tentando evitar situações de *deadlock* ou para obter uma melhor escalabilidade em hardware altamente paralelo usando bloqueio de granularidade fina.

OpenMP é um modelo de programação tradicional em termos de mecanismos oferecidos para garantir a exclusão mútua. Um número significativo de aplicativos foram paralelizados para sistemas de multiprocessador de memória compartilhada usando o OpenMP. Oferece um conjunto de primitivas de baixo nível em torno de mecanismos de exclusão mútua.

A memória transacional pode ser utilizada para este problema, abstraindo as complexidades associadas ao acesso simultâneo aos dados compartilhados, onde vários segmentos precisam acessar simultaneamente posições de memória compartilhadas atomicamente. Memória transacional torna relativamente simples desenvolver programas concorrentes. As propriedades das transações fornecem uma abstração conveniente para coordenar leituras e escritas simultâneas de dados compartilhados em um sistema concorrente. Transações fornecem uma abordagem alternativa para coordenar threads concorrentes. Um programa pode envolver uma computação em uma transação. A atomicidade garante que a computação complete com sucesso o resultado em sua totalidade (*commit*) ou aborte (*abort*). Além disso, o isolamento garante que a transação produz o mesmo resultado independente que quantas outras transação estão sendo executadas simultaneamente (HARRIS et al, 2010).

4. CONCLUSÕES

As sincronizações suportadas pelo OpenMP são seções críticas, barreiras, mutexes ou *locks* e *atomics* do OpenMP. *Locks* e *atomics* são abstrações básicas usadas para acessar e modificar o estado compartilhado. A partir desta visão, essas abstrações visam evitar condições de corrida e sincronizar objetos na memória compartilhada. O bloqueio de granularidade fina – onde todos os dados do programa são protegidos usando um ou poucos *locks* – ou o uso de *atomics* resultam numa associação complexa entre dados e sincronização que protege o acesso a esses dados. A falta de manutenção correta dessas

associações ao longo do programa leva a erros de concorrência, como condições de corrida e deadlocks. Além disso, as estratégias de sincronização projetadas para funcionarem bem em uma plataforma, muitas vezes funcionam mal em uma plataforma com um número diferente de threads de hardware ou um custo diferente para as primitivas de sincronização.

As operações atômicas não se destinam a compor em operações atômicas maiores. A programação concorrente implica inerentemente uma maior complexidade no desenvolvimento de programas, no raciocínio sobre programas e na reprodução de bugs. Ao usar uma transação – em vez de um ou mais *locks* – para sincronizar uma seção de código, o programador não precisa especificar quais metadados (ou seja, qual variável chamada) são usados para sincronizar dados. Além de simplificar o software resultante, esta abordagem alivia a necessidade de uma ordem de execução fixa, que é exigida por *lock* para evitar deadlock. Isso resulta em projetos mais simples, mais fáceis de escrever, e de mais fácil manutenção. Além disso, permite suporte de sincronização especializado para diferentes plataformas, que podem ser melhoradas ao longo do tempo, sem exigir alterações no código do aplicativo.

5. REFERÊNCIAS BIBLIOGRÁFICAS

CAVALHEIRO, G. G. H.; DU BOIS, A. R. **Ferramentas modernas para programação multithread**. In SALGADO, A. C.; LÓSCIO, B. F.; ALCHIERI, E.; BARRETO, P. S., editores, JAI, páginas 41–83. Sociedade Brasileira de Computação, Porto Alegre, 2014.

CHANDRA, R.; DAGUM, L.; KOHR, D.; MAYDAN, D.; McDONALD, J.; MENON, R. **Parallel Programming in OpenMP**. Morgan Kaufmann, San Francisco, 2001.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP: Portable Shared Memory Parallel Programming** (Scientific and Engineering Computation). The MIT Press, 2007.

DAGUM, L.; MENON, R. **Openmp: An industry-standard api for shared-memory programming**. IEEE Comput. Sci. Eng., 5(1):46–55, 1998.

HARRIS, T.; LARUS, J.; RAJWAR, R. **Transactional memory**, 2nd edition. Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.

RIGO, S.; CENTODUCATTE, P.. BALDASSIN, A. **Memórias transacionais: Uma nova alternativa para programação concorrente**. Laboratório de Sistemas de Computação – Instituto de Computação – Unicamp, 2007

WILLIAMS, A. **C++ Concurrency in Action: Practical Multithreading**. Manning Pubs Co Series. Manning, 2012.

WONG, M.; AYGUADÉ, E.; GOTTSCHLICH, J.; LUCHANGCO, V.; DE SUPINSKI, B. R.; BIHARI, B. **Towards Transactional Memory for OpenMP**, pages 130–145. Springer International Publishing, Cham, 2014.