EXTENSÃO PARA MEMÓRIAS TRANSACIONAIS NAS MODERNAS FERRAMENTAS PARA PROGRAMAÇÃO MULTITHREAD

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

A sincronização na programação concorrente é um dos aspectos mais desafiadores, pois refere-se a relacionamentos entre eventos. As restrições de sincronização permitem serializar eventos, no qual um evento B deve seguir a execução de um evento A, e ou executar eventos em regime de exclusão mútua, no qual os eventos A e B não podem ocorrer ao mesmo tempo. Para a sincronização de threads para execução em regime de exclusão mútua o mecanismo tradicional é baseado em semáforos binários, normalmente referenciados como mutexes (WILLIAMS, 2012).

Este trabalho é desenvolvido no contexto da extensão da interface de programação de ferramentas de programação multithread, para contemplar o uso de memórias transacionais. À medida que os programas multithread aumentam em tamanho e complexidade, abstrações mais avançadas são necessárias para abrandar a complexidade da programação que decorre do uso frequente da sincronização em sistemas de software em larga escala (WONG et al., 2014)

OpenMP é um modelo de programação tradicional em termos de mecanismos oferecidos para garantir a exclusão mútua. Oferece um acesso baseado em mutexes para os dados compartilhados. Nesse cenário, as Memórias Transacionais surgiram como uma alternativa promissora aos mecanismos de sincronização baseados em mutex. 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 objetivo deste trabalho é estender o estado da arte em interfaces para programação concorrente multithread, pela introdução de recursos para manipulação de memórias transacionais em ferramentas de programação consolidadas, como o OpenMP. O diferencial buscado é o de garantir que a ferramenta original não terá suas características de programação alteradas pela introdução dos novos recursos. A contribuição é apresentar uma proposta com modelo de memória transacional consistente com o modelo de memória oferecido pela interface de programação de uma ferramenta de programação multithread.

2. METODOLOGIA

Em sistemas de computação, a exploração de concorrência 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 para 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, manipuladas pelo núcleo de escalonamento OpenMP e após escalonadas sobre os threads disponíveis. Cada thread executa o mesmo código, 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 (CAVALHEIRO; DU BOIS, 2014).

Em um programa OpenMP threads podem se comunicar por 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).

Na memória transacional, o aninhamento (MOSS; HOSKING, 2006) é usado para facilitar a composabilidade do código. Um conjunto de transações menores podem ser combinadas, pelo aninhamento, para formar uma transação maior. Transações aninhadas ocorrem quando uma região atômica é definida internamente a outra. A execução de subtransações aninhadas pode ser representada conceitualmente por uma árvore dinâmica de subtransações ativas, denominada árvore de transações, na qual as transações são relacionadas pelo relacionamento pai-filho. (TURCU; RAVINDRAN, 2012).

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. A propriedade de um ou mais mutexes protege o acesso a dados compartilhados. 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 implementa, em uma interface de programação de alto nível, um modelo de programação tradicional em termos de mecanismos oferecidos para garantir a exclusão mútua. Embora os recursos de exclusão mútua de OpenMP possuam alto grau de abstração, eles ainda possuem a mesma complexidade de utilização que a requerida pelas implementações diretas do modelo de mutex.

O modelo de memória transacional pode ser utilizado para abstrair as complexidades associadas ao acesso simultâneo aos dados compartilhados, onde vários segmentos precisam acessar simultaneamente posições de memória compartilhadas atomicamente. 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 transações estão sendo executadas simultaneamente (HARRIS et al, 2010). As memórias transacionais fornecem uma abstração de programação que permite obter programas mais robustos e com a propriedade de composabilidade: transações suportam naturalmente a composição de código. Para criar uma nova operação com base em outras já existentes, basta invocá-las dentro de uma nova transação (aninhamento).

Este trabalho propõe uma abordagem considerando a natureza das aplicações para as quais o OpenMP é voltado. Assim, será analisado o impacto do uso de memórias transacionais com duas das principais diretivas de paralelização de OpenMP: `parallel for` e `task`. O aninhamento permite que uma transação seja composta de tarefas paralelas. Isso faz da memória transacional um paradigma mais poderoso em relação à divisão de programas em tarefas paralelas. O desafio está em fornecer tal benefício de maneira eficiente, de tal forma que não impeça ganhos potenciais do paralelismo explorado.

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.

A principal contribuição do trabalho é estender a API (*Application Programming Interface*) de ferramentas de programação multithread, para suportar memória transacional, tendo como caso de estudo, OpenMP. A diferença na abordagem está em considerar o impacto do uso de memória transacional com os principais construtores de paralelização destas ferramentas (for para fazer uma iteração, *task* para uma recursão), tendo ênfase nas questões de aninhamento. Como resultados esperados, destaca-se o desenvolvimento mais robusto de código, e com possibilidade de composabilidade.

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.

MOSS, J. E. B.; HOSKING, A. L. **Nested Transactional Memory: Model and Architecture Sketches**. Sci. Comput. Program., Amsterdam, The Netherlands, The Netherlands, v.63, n.2, p.186–201, Dec. 2006.

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

TURCU, A.; RAVINDRAN, B. **On Open Nesting in Distributed Transactional Memory**. In: Annual Iinternational Systems and Storage Conference, 5., 2012, New York, NY, USA. Proceedings. . . ACM, p.4:1–4:12. 2012.

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. inger International Publishing, Cham, 2014.