# Extensão para Memórias Transacionais nas Modernas Ferramentas para Programação Multithread

**André D. Jardim[1], André Du Bois[1], Gerson Geraldo H. Cavalheiro[1]**

[1]Centro de Desenvolvimento Tecnológico (CDTec) – Universidade Federal de Pelotas (UFPel)
Caixa Postal 354 – 96010-610 – Pelotas – RS – Brazil

{andre.jardim,dubois,gerson.cavalheiro}@inf.ufpel.edu.br

***Resumo.*** *Este trabalho tem como objetivo incorporar suporte a memórias transacionais em ferramentas de programação multithread. O diferencial desta proposta é contemplar as relações de dependências de dados previstas em tais ferramentas de programação e dar tratamento aos diferentes aninhamentos de transações observados. O aninhamento permite que uma transação seja composta de tarefas paralelas.*

## 1. Introdução

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].

O sincronismo entre threads é usualmente explorado com uso de mutexes ou de alguma estrutura similar provida pela ferramenta de programação utilizada. Porém, a programação com controle de sincronização por mutex é propensa a erros, complexa e contra intuitiva e alguns problemas podem decorrer do seu uso. A robustez do código gerado depende do algoritmo implementado e da atenção do programador no seu desenvolvimento. Outro problema com a utilização de mutexes diz respeito à composição de software.

OpenMP implementa um modelo de programação tradicional, baseado em seções críticas e implementado sobre mecanismos de mutex, para garantir acesso a dados em exclusão mútua. Nesse cenário, o modelo de 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. OpenMP e Memórias Transacionais

Em sistemas de computação, quando relacionado ao desenvolvimento de aplicações, o termo concorrência é associado a aspectos ligados à descrição de atividades concorrentes [Cavalheiro e 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. 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 e Menon 1998] define uma interface de programação para explorar a concorrência em programas C e Fortran, voltada para ambientes de memória compartilhada. Em OpenMP, *regiões paralelas* definem seções do código em que a concorrência é apresentada de forma explícita, sob o modelo *Fork/Join* aninhado. A concorrência descrita nestas regiões paralelas permite a criação de *tarefas*, manipuladas pelo núcleo de escalonamento OpenMP e após escalonadas sobre os recursos de processamento, denominados *threads do pool de execução*, disponíveis.

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. OpenMP prevê, para tanto, mecanismos que restringem o acesso aos dados via parametrização das tarefas, como mecanismos de alto nível provendo um regime de execução em exclusão mútua.

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 [Harris et al. 2010].

Em memórias transacionais, o aninhamento [Moss e 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 e Ravindran 2012]. A organização das transações e seu ordenamento tem impactos significatívos no desempenho dos programas.

## 3. Considerações e Discussão

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.

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 Padrão OpenMP define semântica de acesso a dados compartilhados por cláusulas associadas às diretivas de paralelização. Assim, neste trabalho entende-se que a semântica de acesso a dados manipulados por transações também deva ser definida da mesma forma, ou seja, sendo definida por cláusulas. 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

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 – ainda em sua fase inicial – 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, em relação aos trabalhos relacionados estudados, 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), com ênfase nas questões de aninhamento. Tendo em vista tal abordagem, como resultados esperados, destaca-se o desenvolvimento mais robusto de código, e com possibilidade de composabilidade.

## 5. Referências


Cavalheiro, G. G. H.; Du Bois, A. R. (2014) "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.

Chapman, B.; Jost, G.; Pas, R. v. d. (2007) "Using OpenMP: Portable Shared Memory Parallel Programming", (Scientific and Engineering Computation). The MIT Press.

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

Harrris, T.; Larus, J.; Rajwar, R. (2010) "Transactional memory", 2nd edition. Synthesis Lectures on Computer Architecture, 5(1):1–263.

Moss, J. E. B.; Hosking, A. L. (2006) "Nested Transactional Memory: Model and Architecture Sketches", Sci. Comput. Program., Amsterdam, The Netherlands, The Netherlands, v.63, n.2, p.186–201.

Turcu, A.; Ravidran, B. (2012) "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.

Williams, A. (2012) "C++ Concurrency in Action: Practical Multithreading", Manning Pubs Co Series. Manning.

Wong, M.; Ayguadé, E.; Gottschlich, J.; Luchangco, V.; De Supinski, B. R.; Bihari, B. (2014) "Towards Transactional Memory for OpenMP", pages 130–145. inger International Publishing, Cham.