Estudo da Viabilidade de uma Interface para Memórias Transacionais em OpenMP*

Henry S. Pereira, Kevin O. de Oliveira, André D. Jardim
André R. Du Bois, Gerson Geraldo H. Cavalheiro[1]

[1]Programa de Pós-Graduação em Computação
Centro de Desenvolvimento Tecnológico
Universidade Federal de Pelotas
Pelotas – RS – Brasil

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

Abstract. *Although modern programming language libraries and tools for parallel programming offer efficient hardware utilization, the support for data synchronization still reflects classic models based on critical sections. Transactional Memory (TM) is a high level concurrency control model that is usually not available in parallel programming tools. This paper presents an OpenMP extension for TM and analyses its support using two different implementations of TM. The results point to the viability of the proposed extension and show that design options used to manage the TM system directly influence the performance of programs.*

Resumo. *Embora as modernas ferramentas para programação multithread ofereçam recursos para exploração eficiente do hardware, os suportes à sincronização a dados compartilhados ainda refletem modelos baseados em seção crítica clássicos. Memória Transacional (TM) propõe um modelo de controle de concorrência de mais alto nível, mas não se encontra disponível em tais ferramentas. Neste artigo é apresentada uma extensão à OpenMP para suporte à TM e uma avaliação de desempenho de sua prototipação sobre duas ferramentas que suportam TM. Os resultados apontam a viabilidade da extensão proposta e a análise das execuções permitiu concluir que as políticas aplicadas para gestão da TM são decisivas para o bom desempenho do programa.*

1. Introdução

A programação concorrente inicialmente proposta para desenvolvimento de sistemas operacionais confiáveis, foi rapidamente compreendida como aplicável "a qualquer forma de computação paralela", nos anos 1960 [Hansen et al. 2002]. Modelos e ferramentas de programação surgiram, desde então, oferecendo diferentes níveis de abstração em relação à descrição da concorrência da aplicação e a exploração do hardware paralelo

*GREEN-CLOUD: Computação em Cloud com Computação Sustentável (16/2551-0000 488-9), Programa PRONEX – FAPERGS/CNPq.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001

[Skillicorn and Talia 1998]. A despeito de numerosos avanços nas tecnologias de hardware para processamento paralelo, tanto em diversidade, quanto em volume de paralelismo oferecido, os avanços em recursos de programação podem ser considerados modestos em oferecer novas abstrações que facilitem o desenvolvimento de software paralelo.

Situando a discussão no contexto da programação em multiprocessadores, é esperada a formação de programadores no uso de ferramentas para programação multithread. Esta formação, atualmente remete ao domínio de recursos clássicos de programação, como mutex e semáforos, cuja proposta data também dos anos 1960, e monitores, datado do início dos anos 1970. Ferramentas de mercado para programação multithread, como C++, Java e OpenMP oferecem tais recursos de programação. Mesmo TBB, representante de uma nova geração de ferramentas de programação multithread, rende-se à prática de seu público alvo e oferece mutex para controle de acesso a dados compartilhados.

Embora possam ser eficientes em cumprir seus propósitos, tais recursos clássicos para sincronização a dados compartilhados podem não representar as necessidades do mercado de produção de software em massa para sistemas computacionais. Atualmente, a realidade das arquiteturas paralelas apresenta uma escala de paralelismo superior aquela disponível nas décadas de 1960 e 1970. Além disto, requisitos de robustez e composabilidade não são atributos respondidos por tais mecanismos de sincronização. Uma alternativa, antecipando a aurora dos processadores multicore como commodity, foi, no início da década de 1990, a proposição do uso de Memórias Transacionais.

Memórias Transacionais representam um modelo de sincronização a dados em memória. Podendo ser implementada em hardware ou software, Memórias Transacionais oferecem uma alternativa com mais alto nível de abstração aos mecanismos clássicos de sincronização. A literatura registra várias implementações de Memória Transacional em Software (STM), contexto em que se insere este trabalho. No entanto, a não adoção efetiva deste modelo na interface de programação de ferramentas populares indica que existe a possibilidade de pesquisas e propostas na área.

Neste trabalho, propõe-se a extensão da interface de programação OpenMP para incorporar sincronização por Memória Transacional, sendo apresentado um estudo sobre seu uso em casos de estudo. O diferencial desta proposta é adequar o uso do modelo de Memória Transacional aos hábitos de programação nesta ferramenta. A validação dos resultados é apresentada na forma de uma avaliação do desempenho da prototipação desta extensão com duas ferramentas que oferecem suporte à Memória Transacional, comparando os resultados obtidos.

O restante deste artigo está organizado como segue. A Seção 2 apresenta o estado da arte na área e trabalhos relacionados ao proposto. As seção 3 e 4 apresentam, respectivamente, conceitos do modelo de Memória Transacional e de OpenMP relevantes ao trabalho. A interface proposta é apresentada na Seção 5. A Seção 6 apresenta a avaliação de desempenho realizada e, então, tecidos comentários finais na Seção 7 como conclusão.

2. Estado da Arte e Trabalhos Relacionados

Um estudo bibliográfico, documentado em [Jardim 2020], identificou que, após o termo *Transaction Memory* atingir um pico de ocorrências em artigos científicos no ano de 2007, a frequência de citações a este termo diminuiu na mesma ordem em que a frequência dos

termos multicore e OpenMP cresceu. Os autores correlacionam estes dados ao lançamento ao mercado dos primeiros processadores dual-core em 2005 e à introdução de novas funcionalidades em OpenMP em 2008. Por um lado, a chegada ao grande público de arquiteturas paralelas aumentou o interesse pela programação na área, influenciando pesquisas em interfaces de programação. Por outro, o sucesso de OpenMP como alternativa com alto poder de expressão e bom desempenho, associado a uma curva de aprendizado curta, centralizou o interesse de boa parte da comunidade científica.

Como consequência, ferramentas para programação paralela, em particular multithread como o próprio OpenMP e o moderno TBB [Voss et al. 2019], ainda são oferecidas com recursos *clássicos* de sincronização, tais os baseados em controle de acesso a região crítica por mutex ou mecanismos equivalentes de barreira. Embora este tipo de recurso de programação seja amplamente dominado pela comunidade de programadores, o uso de tais mecanismos tem efeitos negativos, tanto no desempenho de programas como em seu processo de desenvolvimento.

A contenção pela potencial perda de execução paralela é um dos aspectos negativos em termos de desempenho [Boyd-Wickizer et al. 2010], assim como perda da escalabilidade do programa em termos de incremento no número de CPUs [Voss et al. 2019]. O desempenho também é afetado pelos custos operacionais associados a gestão da sincronização, como chamadas de sistema e troca de contexto de threads [Voss et al. 2019]. Já a perda da composabilidade [Sutter 2007, Wong et al. 2014], necessária em um modelo de produção de software em escala, bem como o incremento da complexidade da utilização de mecanismos de sincronização em função do tamanho do programa, são aspectos negativos relacionados ao processo de desenvolvimento.

Do período de auge nas pesquisas com TM, as ferramentas OpenTM [Baek et al. 2007] e Nebelung [Milovanović et al. 2007] podem ser consideradas relacionadas a presente proposta. OpenTM propõe diretivas específicas para manipulação de TM tendo como base as diretivas nativas de OpenMP. Nebelung, por sua vez, se apresenta como um framework a ser utilizado em conjunto à OpenMP oferecendo uma diretiva própria para manipulação de TM. A estas duas ferramentas também soma-se [Wong et al. 2014], o qual também propõe diretivas para manipulação de TM, além de outros mecanismos de sincronização.

A Figura 1 ilustra as interfaces propostas pelas ferramentas apresentadas. Tanto OpenTM como [Wong et al. 2014] propõem o uso de diretivas que permitem instanciar uma coleção de tarefas que manipulam dados compartilhados segundo o modelo de TM. Já Nebelung propõe uma diretiva parametrizada por cláusulas que permitem identificar as variáveis que devem ser monitoradas na transação e aquela em que o monitoramento não é necessário.

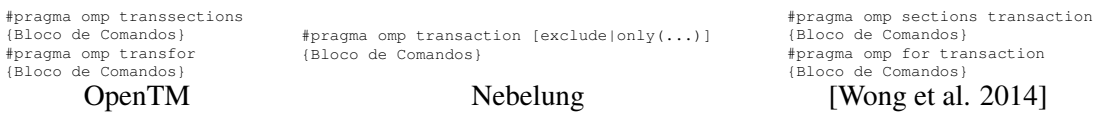

Figura 1. Interfaces propostas por trabalhos relacionados.

Uma das razões que podem ter influenciado a não adoção de TM para OpenMP é o fato de que as propostas apresentadas na literatura oferecem soluções que não são

consonantes com os principais mecanismos de controle ao acesso a dados. Na verdade, as propostas apresentadas se caracterizam mais fortemente como uma nova abordagem para a implementação dos mecanismos de exclusão mútua já existentes em OpenMP do que de uma nova proposta abstração para TM.

Ainda que seções críticas estejam disponíveis em OpenMP, o uso deste recurso é preterido, em boa parte das aplicações, pela parametrização da criação de tarefas com as semânticas de acesso a dados compartilhados. Esta parametrização, via cláusulas, é característica de OpenMP e oferece um nível de abstração mais alto. A presente proposta difere das apresentadas por introduzir uma nova cláusula em OpenMP para manipulação de TM, entendendo que, assim, a extensão proposta estará mais próxima a realidade de desenvolvimento nesta linguagem, oferecendo uma nova semântica de acesso a dados compartilhados.

3. Memórias Transacionais

É possível implementar o modelo de Memórias Transacionais (TM) tanto em hardware como em software, havendo ainda a possibilidade de uma implementação mista. A despeito de esforços de grandes empresas de tecnologias, na primeira década do século XXI, para introduzir suporte à memória transacional nas arquiteturas de processadores, uma solução estável não foi apresentada comercialmente. Por outro lado, ferramentas de programação com suporte à Memória Transacional em Software (STM) se popularizaram. Em paralelo a esta popularização, diversos estudos quanto às políticas para implementação dos mecanismo envolvidos neste modelo de sincronização, como versionamento e controle de concorrência, caracterizaram comportamentos esperados no seu uso em função das características das aplicações e propriedades paralelas da arquitetura.

O princípio das operações da TM é garantir a execução de uma transação, representada por uma sequência de instruções, sobre dados compartilhados, 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 [Rigo et al. 2007].

Considerando o modelo de programação paralela tradicional, pode-se dizer que uma transação é equivalente a uma seção crítica, cujo objetivo é garantir que as leituras e escritas na memória realizadas pelos diversos fluxos de execução não resultem em um estado inconsistente. Todavia, esta nova abordagem utiliza uma sintaxe mais simples e fácil de compreender, além de evitar as serializações por contenção [Guerraoui and Kapałka 2010]. Uma possível interface é apresentada em [Harris et al. 2010] com os seguintes serviços:

- `void StartTx()`: inicia uma transação no thread atual;
- `bool CommitTx()`: tenta concluir a transação, retornando `true` se for bem sucedida e `false` se falhar;
- `T ReadTx(T *address)`: realiza leitura no endereço `address` de um valor do tipo `T`;
- `void WriteTx(T *address, T v)`: escreve no endereço `address` um novo valor `v` do tipo `T`.

O suporte a TM garante a consistência do dado compartilhado na execução da operação `CommitTx` após monitorar os acessos promovidos por `ReadTx` e `WriteTx` no contexto das transações em execução simultânea. Ocorrendo conflito, a transação falha (`CommitTx==false`) e a transação é reiniciada. A programação com TM, portanto, exige que o programador informe quais dados devem ser monitorados (`ReadTx` e `WriteTx`) e quais são os limites de monitoramento (`StartTx`/`CommitTx`), liberando-o, no entanto, de introduzir mecanismos de barreira.

Memórias transacionais são apresentadas como alternativa aos métodos baseados em mutex, oferecendo as seguintes vantagens [Rigo et al. 2007]:

- **Facilidade de programação**: a programação torna-se mais fácil, pois o programador não precisa se preocupar em como garantir a sincronização, e sim em especificar o que deve ser executado atomicamente. Basta especificar o trecho de código que deve ser executado atomicamente e o sistema de execução garante a sincronização. Ou seja, a responsabilidade pela sincronização passa do programador para a entidade que implementa as transações.
- **Escalabilidade**: transações que acessem um mesmo dado para leitura podem ser executadas concorrentemente. Também podem ser executadas em paralelo as transações que modifiquem partes distintas de uma mesma estrutura de dados. Essa característica permite que mais desempenho seja obtido com o aumento do número de processadores, pois o nível de paralelismo exposto é maior.
- **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. Novamente, o sistema de execução garante que as operações sejam executadas de forma atômica.

Devido às características de atomicidade e isolamento, sistemas transacionais podem explorar mais paralelismo, aumentando sua escalabilidade e seu desempenho. O uso de memória transacional também facilita a programação multithreaded, pois o programador não precisa se preocupar em garantir a sincronização. Todo o controle de acesso à memória compartilhada é realizado automaticamente pelo sistema que implementa memória transacional.

4. Aspectos de OpenMP

OpenMP é uma ferramenta de programação multithread no modelo de $n \times m$. A concorrência da aplicação, representada por n tarefas, é descrita independente dos m recursos de execução disponíveis. Esta independência entre a descrição da concorrência e a execução paralela é garantida por um *runtime* implementando um mecanismo de escalonamento em nível usuário. Tal recurso em OpenMP, quando proposto, foi um diferencial, por permitir a exploração eficiente de multiprocessadores sem a necessidade de introdução de código específico às ações de escalonamento no código do programa. Desde sua primeira versão, datada de 1998, OpenMP manteve-se atualizada, incorporando novas abstrações para exploração do hardware disponível. Em particular, incorporação de suporte às instruções atômicas e SIMD e suporte a aceleradores (OpenMP 4.0). No entanto, seu modelo de memória mantém-se inalterado desde sua primeira versão.

Sendo um programa multithread, todas as tarefas podem, virtualmente, acessar todo espaço de endereçamento de um processo. A prática em OpenMP é limitar o acesso

aos dados em memória impondo uma semântica à manipulação dos identificadores visíveis ao escopo das tarefas, estando os dados armazenados na memória do processo ou na memória específica do thread. Para tanto, a operação de criação de tarefas é *parametrizada* de forma que a semântica de manipulação dos dados compartilhados siga uma regra definida. Como exemplo, tanto `reduction(+:a)` como `shared(b)` correspondem a anotação de parâmetros de entrada e saída (`a` e `b`) para as tarefas. Enquanto a semântica associada à manipulação `a` garante consistência no acesso ao dado em uma política de leituras e escritas concorrentes, o mesmo não é garantido na manipulação de `b`.

O programador possui alternativas para manipular dados compartilhados em regime de exclusão mútua: a diretiva `critical` e uso de mutex, com apoio de um tipo de dado específico e serviços da biblioteca. Apresentando uma interface de mais alto nível, `critical` opera em regime de exclusão mútua aplicada a blocos. Assim, a região crítica está contida no contexto de uma tarefa. Ocorre uma situação de deadlock, no entanto, caso o mesmo thread tente entrar, de forma aninhada, típica em algoritmos recursivos, uma segunda vez em uma barreira `critical`. O uso do mutex, não impõe uma estrutura de bloco, existindo a possibilidade de manipulação de forma aninhada (*nested lock*). No entanto, o programador deve ter atenção quando inserir a aquisição e a liberação de um mutex em diferentes partes de uma tarefa: pode ocorrer desta tarefa ser preemptada e uma outra tarefa, manipulando o mesmo mutex, ser lançada sobre o thread então liberado. Caso seja um mutex reentrante, pode ocorrer uma condição de corrida; sendo um mutex normal, uma situação de deadlock. Também é comum, desencorajar o programador a utilizar mecanismos de exclusão mútua em tarefas não amarradas (*untied*), pois estas tarefas estão sujeitas a executar sobre qualquer thread, potencializando a ocorrência de situações de deadlock e, em função de sua menor prioridade de execução, podem ocasionar contenção do paralelismo [Süß and Leopold 2008].

Outro aspecto relevante é a diferença conceitual entre o uso de seções críticas e o uso de cláusulas para parametrização da semântica de acesso a dados pelas tarefas. No primeiro caso, o controle é realizado sobre um trecho de código; no segundo, sobre os endereços de memória. Soma-se o fato de que o próprio OpenMP garante a consistência na manipulação destes endereços de memória.

Considerando a questão do compartilhamento de dados entre tarefas, observa-se uma lacuna em OpenMP que pode ser coberta pela adoção de suporte à Memoria Transacional. Entende-se que este suporte deve respeitar as práticas de programação nesta ferramenta, não introduzindo um recurso totalmente estranho a ela, nem sobrepondo recursos já existentes. Outrossim, reduzindo a necessidade de introdução em seus algoritmos dos mecanismos de controle de execução em regime de exclusão mútua.

5. Proposta da Interface

A extensão à OpenMP proposta para suporte à memória transacional introduz uma nova cláusula, denominada `transaction`, às diretivas construtoras de tarefas. A cláusula `transaction` recebe uma lista de tuplas, identificando as variáveis a serem monitoradas na transação e a operação, leitura, escrita ou leitura/escrita, que cada uma destas variáveis irá sofrer. A gramática (simplificada) desta extensão é apresentada na Figura 2.

A transação referente a uma cláusula `transaction` corresponde ao código do `Bloco de Comandos` que a sucede. Caso outras tarefas sejam criadas neste bloco, a

```
#pragma omp <diretiva> transaction(<op>:<id>,...) [<cláusulas>]
{ Bloco de Comandos }
```

Onde:

```
<diretiva>  :: sections | for | task
      <op>  :: R | W | RW
      <id>  :: identificador para a variável
<cláusulas> :: outras cláusulas OpenMP
```

Figura 2. Interface proposta.

semântica de acesso às variáveis monitoradas pela transação se dará no modo default à respectiva diretiva ou no modo em que for explicitado, como `private`, `reduction` ou mesmo novamente `transaction`.

Ao ser aplicado a uma diretiva `sections`, as transações são delimitadas pelo escopo das tarefas criadas nas diferentes `section`. A aplicação na diretiva `for` implica que cada tarefa criada, para cada *chunk* gerado, será uma transação. No uso com a diretiva `task`, a transação corresponde ao código associado à tarefa criada.

5.1. Tratamento do Aninhamento

O modelo de execução *fork*/*join* aninhado de OpenMP conduz as decisões sobre o modelo de aninhamento adotado: aninhamento fechado. Neste modelo de aninhamento, quando um transação aninhada a outra falha, somente esta transação é reexecutada, uma vez que a probabilidade de conflito está associada as demais tarefas instanciadas no mesmo *fork* e, concluindo com sucesso, as demais transações aninhadas no mesmo nível só terão acesso ao dado atualizado quando a transação mais externa concluir. Como a transação está associada ao bloco no qual é aplica, abortá-la, mesmo quando o dado manipulado também esteja sendo monitorado pela tarefa de seu escopo envolvente, não é produzido efeito em cascata. A transação em uma tarefa está relacionada com as transações das demais tarefas instanciadas na mesma operação de *fork*.

As situações de aninhamento ocorrem quando a cláusula `transaction` é utilizada em conjunto com a cláusula `nowait` e quando aplicada à diretiva `task`. A cláusula `nowait` relaxa a condição de barreira no `join`, permitindo que a execução da tarefa no escopo envolvente siga sua execução mesmo que todas as tarefas criadas não tenham sido concluída. Por sua vez, a diretiva `task` impõe uma natureza recursiva ao algoritmo. O uso de um aninhamento fechado implica que uma transação interna ao concluir com sucesso não atualizará os dados da transação caso uma transação mais externa já tiver ela própria concluído com sucesso. Os dados são, portanto, descartados. A vantagem deste mecanismo é seu baixo custo operacional, além de apresentar a mesma semântica de uma execução sequencial.

5.2. Prototipação

A prototipação da extensão proposta explora a biblioteca *Vanilla*-TM, concebida para permitir a compatibilidade das implementações realizadas sobre diferentes bibliotecas oferecendo serviços de memória transacional. *Vanilla*-TM oferece o seguinte conjunto de serviços:

- `vtm_tm_t* vtm_start( vtm_dataset_t* dataSet );`
  Determina o início de uma transação e identifica o conjunto de dados monitorados. Retorna um descritor para a transação criada.
- `void vtm_commit( vtm_tm_t* tm );`
  Caso a transação conclua com sucesso, prossegue a execução. Caso contrário, realiza nova entrada.
- `void* vtm_read( vtm_tm_t* tm, void* data);`
  Realiza acesso em leitura de `data`.
- `void  vtm_write( vtm_tm_t* tm, void* data, void* value);`
  Realiza acesso em `data` para escrita de `value`.

A opção por manipular o *dataset* com o tipo `void*` oferece a possibilidade do uso de *Vanilla*-TM em programas C. Os tipos de dados `vtm_tm_t` e `vtm_dataset_t` são tipos de dados opacos cujas primitivas de manipulação não são apresentadas neste texto. De forma resumida, `vtm_tm_t` consiste no descritor de uma transação e `vtm_dataset_t` no descritor do conjunto de dados manipulados na transação.

Dentre as ferramentas com suporte ao modelo de TM em Software, este artigo aponta TinySTM e GCC-TM como alternativas à implementação de *Vanilla*-TM. O primeiro representa a implementação de TM sob a forma de uma biblioteca, compatível com programas C/C++. O segundo, o suporte no compilador Gnu para a linguagem C/C++ para a especificação de memórias transacionais desta linguagem.

TinySTM [Felber et al. 2008] implementa uma variação do algoritmo TL2 (Transactional Locking 2) [Dice et al. 2006]. Esta biblioteca disponibiliza três estratégias de versionamento, duas com versionamento atrasado (write-back) e uma com versionamento adiantado (write-through). A estratégia de gerenciamento de contenção padrão interrompe a execução da transação imediatamente a detecção de um conflito. Outras três estratégias para este gerenciamento estão disponíveis.

O GCC-TM é uma biblioteca de memórias transacionais para o GCC cuja interface é baseada na especificação de primitivas transacionais para o C++ [Ali-Reza Adl-Tabatabai and Gottschlich 2012]. O sistema permite que várias bibliotecas de STM sejam usadas junto ao GCC, porém a instalação padrão utiliza a biblioteca libitm [Free Software Foundation, Inc 2020]. Por padrão, essa biblioteca utiliza versionamento adiantado e uma abordagem mista de detecção de conflitos, sendo adiantada para conflitos de escrita e atrasada para conflitos de leitura [Paznikov et al. 2019].

5.3. Implementação dos Problemas

A validação da interface proposta se deu pela implementação dos seis programas do benchmark Cowichan [Wilson 1994] em que foi identificada a aplicabilidade do modelo de memória transacional: hull, norm, outer, sor, thresh e vecdiff. Este benchmark não foi concebido para análise de desempenho, mas sim para caracterizar as ferramentas de programação paralela em termos da capacidade de representação de seus recursos de programação. Na Figura 3.a é apresentado o código implementado na interface proposta, onde o uso da diretiva `transaction` é aplicada ao dado compartilhado `dMax`. O código na Figura 3.b apresenta a implementação intermediária, obtida a partir do código em Figura 3.a, empregando *Vanilla*-TM. A linguagem intermediária *Vanilla*-TM é utilizada para representar a implementação em baixo nível do programa. O suporte operacional é realizado por TinySTM e GCC-TM.

```
void outer( Pontos* v, int n, double** m )
 double dMax = 0.0;
 #pragma omp for
 for( r = 0 ; r < n ; ++r )
  #pragma omp for transaction( RW:dMax ) firstprivate(d)
   for( c = 0 ; c < r ; ++c ) {
    d = distancia(Pontos[r],Pontos[c]);
    if( d > dMax )
     dMax = d;
    m[r][c] = m[c][r] = d;
  }
}
```

(a) Código na interface proposta: cláusula `transaction`.

```
void outer( Pontos* v, int n, double** m )
 double dMax = 0.0;
 #pragma omp for
 for( r = 0 ; r < n ; ++r )
  #pragma omp for shared(dMax) firstprivate(d)
   for( c = 0 ; c < r ; ++c ) {
    double __dMax__;
    vtm_tm_t* tm;
    vtm_dataset_t ds;
    vtm_dataset_init(&ds);
    vtm_dataset_pack(&ds,&dMax,VTM_READWRITE);
    tm = vtm_start(&ds);
    d = distancia(Pontos[r],Pontos[c]);
    __dMax__ = *(double*) vtm_read(tm,&dMax,&__dMax__);
    if( d > __dMax__ ) {
     __dMax__ = d;
     vtm_read(tm,&dMax,&dMax);
    m[r][c] = m[c][r] = d;
  }
}
```

(b) Código intermediário com *Vanilla*-TM.

Figura 3. Trecho de código do programa outer na interface proposta.

6. Experimentação

Foram realizadas três implementações para os programas Cowichan selecionados. Uma em OpenMP e as outras duas com OpenMP utilizando os suportes de Memória Transacional em TinySTM e GCC-TM. Os experimentos foram realizados em dois diferentes ambientes, ambos arquiteturas NUMA: hydra (arquitetura Opteron com 64 *cores*, 4 nós, 120 GB RAM) e tekoha (arquitetura Xeon com 192 *cores*, 8 nós, 120 GB RAM). Os experimentos consideraram três tamanhos de entrada do problema, definidos conforme a natureza de cada aplicação: *pequeno*, *médio* e *grande*. Foram coletados tempos de execução para 2, 4, 8, 16, 32 e 64 threads no time de execução de OpenMP.

Os experimentos foram executados com o uso dedicado das máquinas. Os fontes, bem como os scripts para execução e geração de gráficos, estão disponíveis em `https://github.com/GersonCavalheiro/OpenMP_TM`. Observa-se que a implementação nas versões com TM possui o mesmo algoritmo implementado na versão com OpenMP pura, para uniformizar a análise.

A metodologia do experimento envolveu a amostragem dos tempos de execução de cada uma das combinações das variantes para o estudo de caso: 3 ferramentas de programação, 2 arquiteturas, 6 problemas, 3 tamanhos de entrada, 6 configurações para o time de execução. O total de instâncias analisadas foi de 648, sendo que cada uma executada 30 vezes para obtenção do tempo médio de execução. O conjunto de amostras de cada instância foi aferido, pelo teste de Kolmogorov-Smirnov, para verificar adesão a uma distribuição normal, visando identificar em quais casos a média é representativa. Em seguida, tomando as médias aceitas, aplicando o teste t de Student (significância de 95%) como teste de hipótese, foram comparados os desempenhos das implementações com OpenMP, OpenMP/TinySTM, OpenMP/GCC-TM.

A Tabela 1 exemplifica os resultados obtidos. Dentre as variáveis do caso de estudo, foram fixados o tamanho da entrada do problema em *grande* e o número de threads no time de execução em 32. As células marcadas em vermelho correspondem aos experimentos cujas amostras não aderiram a uma curva normal. Neste caso, os tempos médios são apresentados de forma ilustrativa, mas não foram utilizados na comparação entre as ferramentas. A discussão na sequência considera todo o conjunto das amostras, não apenas a ilustração dos resultados apresentada na Tabela 1.

Tabela 1. Desempenho do caso (tempo em segundos): Entrada *grande*, com 32 threads no time de execução.

	Hydra			Tekoha		
	OpenMP	*TinySTM*	*GCC-TM*	*OpenMP*	*TinySTM*	*GCC-TM*
hull	2,42 s	9,13 s	150,29 s	2,44 s	13,39 s	266,60 s
norm	0,61 s	1,29 s	47,81 s	0,15 s	0,63 s	80,67 s
outer	0,82 s	4,56 s	133,04 s	1,92 s	2,26 s	241,26 s
sor	4,05 s	4,20 s	4,37 s	0,82 s	0,90 s	1,08 s
thresh	1,59 s	265,56 s	503,10 s	0,94 s	489,86 s	777,62
vecdiff	0,85 s	2,50 s	223,36 s	0,19 s	1,43 s	399,56

Análise das amostras. O teste de normalidade das amostras permitiu compreender o impacto do tamanho do problema na representatividade dos tempos de execução coletados. Para problemas cujos tempos de execução são curtos, a variabilidade dos tempos associados a gestão de todo o processo tem maior influência mais que o tempo efetivo da computação útil no tempo total. Por outro lado, execuções longas nas implementações com memória transacional também mostraram, em determinadas situações, variabilidade muito alta. Neste caso, é possível atribuir este comportamento à operação do mecanismo de gerencia da TM promovido pelas diferentes ferramentas – este aspecto deve ser confirmado em outro tipo de análise. No entanto, a quantidade de experimentos que retornaram médias representativas permitiu embasar as conclusões apresentadas sobre o desempenho.

Teste de Hipóteses. Foram testadas duas hipóteses, a primeira que a as execuções com OpenMP puro fossem mais eficientes, comparando as médias de OpenMP com as oferecidas por TinySTM e GCC-TM. A segunda que as ferramentas com suporte a TM apresentassem médias nos tempos de execução distintas, podendo ser identificada aquela com melhor desempenho, comparando as médias de TinySTM e GCC-TM. O resultado de que as execuções com OpenMP puro teriam melhor desempenho, em razão do sobrecusto da adição dos mecanismos de gestão de TM, ficou comprovado na maior parte dos casos. No entanto, a aplicação sor ilustra uma situação onde não é possível afirmar a superioridade de OpenMP, uma vez que o teste T não foi capaz de confirmar, com 95% de confiança, que as médias obtidas para OpenMP e TinySTM proveem de amostras distintas. A comparação entre TinySTM e GCC-TM também aponta clara vantagem, em termos de desempenho, para a primeira, exceto também na aplicação sor onde o teste T também não foi conclusivo.

No presente trabalho não foram exploradas as diferentes políticas de gerência de memória transacional oferecidas por TinySTM e GCC-TM, sendo utilizados os mecanismos apresentados por padrão. Isso pode justificar a diferença de tempos obtidas em diferentes casos de estudo. Também, de forma intencional, procurou-se realizar a implementação utilizando os recursos mais simples das ferramentas, tendo como perspectiva a tradução do código da linguagem intermediária *Vanilla*-TM a ser implementada em uma etapa futura deste projeto. A adequação das ferramentas de TM aos problemas Cowichan deve ser objeto de uma análise posterior.

A análise global dos resultados permite inferir que OpenMP puro produz naturalmente melhor desempenho por utilizar recursos de mais baixo nível. Também é observado

que para diferente casos, sor e norm em destaque neste ponto, o uso de TM (em particular com TinySTM) pode ser considerado. Para outros, como thresh e hull, ferramentas de TM não são apropriadas, pelo menos não utilizando o mesmo algoritmo implementado em OpenMP.

7. Conclusão

Em reação ao surgimento de novas tecnologias de hardware, OpenMP tem evoluído pela incorporação de novos recursos de programação em sua API para manipulá-los. Exemplos são a introdução de diretivas para manipulação de instruções atômicas e SIMD e, mais recentemente, ao suporte de hardware heterogêneo. Mesmo em relação ao incremento de *cores*, típicos em arquiteturas NUMA, novos mecanismos para gestão de grupos de threads em diferentes times de execução foram introduzidos. No entanto, OpenMP continua oferecendo os mesmos recursos de sincronização a dados compartilhados de quando lançada sua primeira versão, em 1998.

Programadores OpenMP exploram, como principal recurso de coordenação aos dados, a parametrização das tarefas pelas cláusulas (`private`, `shared`, `reduction` ...), recorrendo ao uso de mecanismos de exclusão mútua sobre dados `shared`, para os quais não há nenhuma semântica de coerência implícita. Neste artigo foi proposto o uso da diretiva `transaction` como uma extensão à OpenMP consonante com o estilo de programação OpenMP e oferecendo todas a propriedades de composibilidade e robustez promovido pelo modelo de memória transacional.

Além da extensão proposta, foi apresentada sua linguagem intermediária *Vanilla*-TM e uma análise de desempenho utilizando a prototipação dos programas OpenMP utilizando a interface proposta prototipada sobre TinySTM e GCC-TM, duas ferramentas oferecendo suporte à TM. O objetivo deste trabalho foi verificar a viabilidade da implementação da interface proposta. Como discutido na Seção 6, a interface é viável, cabendo um estudo adicional que caracterize às aplicações mais adequadas ao uso do recurso proposto.

Os trabalhos futuros envolvem, além da efetiva introdução da extensão em OpenMP, uma avaliação de desempenho considerando benchmark específico para avaliar ferramentas oferecendo o modelo de Memória Transacional.

Referências

Ali-Reza Adl-Tabatabai, T. S. and Gottschlich, J. (2012). Draft Specification of Transactional Language Constructsfor C++. https://sites.google.com/site/tmforcplusplus/.

Baek, W., Minh, C. C., Trautmann, M., Kozyrakis, C., and Olukotun, K. (2007). The OpenTM transactional application programming interface. In *Proc of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 376–387, Washington, DC. IEEE Computer Society.

Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R. T., Zeldovich, N., et al. (2010). An analysis of Linux scalability to many cores. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI'10*.

Dice, D., Shalev, O., and Shavit, N. (2006). Transactional Locking II. In *Proc. of the 20th Int. Conf. on Distributed Computing*, DISC'06, Berlin. Springer-Verlag.

Felber, P., Fetzer, C., and Riegel, T. (2008). Dynamic performance tuning of word-based software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246.

Free Software Foundation, Inc (2020). The GNU Transactional Memory Library. https://gcc.gnu.org/onlinedocs/libitm.pdf.

Guerraoui, R. and Kapałka, M. (2010). Principles of transactional memory. *Synthesis Lectures on Distributed Computing Theory*, 1(1):1–193.

Hansen, P. B., Dijkstra, E. W., and Hoare, C. A. R. (2002). *The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag, Berlin.

Harris, T., Larus, J., and Rajwar, R. (2010). Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263.

Jardim, A. D. (2020). Extensão para memórias transacionais nas modernas ferramentas para programação multithread. Proposta de Tese de Doutorado.

Milovanović, M., Ferrer, R., Gajinov, V., Unsal, O. S., Cristal, A., Ayguadé, E., and Valero, M. (2007). Multithreaded software transactional memory and OpenMP. In *Proc of the 2007 Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, MEDEA '07, pages 81–88, New York. ACM.

Paznikov, A. A., Smirnov, V. A., and Omelnichenko, A. R. (2019). Towards efficient implementation of concurrent hash tables and search trees based on software transactional memory. In *2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, pages 1–5.

Rigo, S., Centoducatte, P., and Baldassin, A. (2007). Memórias Transacionais: Uma nova alternativa para programaçao concorrente. *Laboratório de Sistemas de Computação–Instituto de Computação–Unicamp*.

Skillicorn, D. B. and Talia, D. (1998). Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169.

Süß, M. and Leopold, C. (2008). Common mistakes in Openmp and how to avoid them. In Mueller, M. S., Chapman, B. M., de Supinski, B. R., Malony, A. D., and Voss, M., editors, *OpenMP Shared Memory Parallel Programming*, Berlin. Springer.

Sutter, H. (2007). The Pillars of Concurrency. *Dr. Dobbs*.

Voss, M., Asenjo, R., and Reinders, J. (2019). *SynchronizationSynchronization: Why and How to Avoid It*, pages 137–178. Apress, Berkeley, CA.

Wilson, G. V. (1994). Assessing the usability of parallel programming systems: The Cowichan problems. In Decker, K. M. and Rehmann, R. M., editors, *Programming Environments for Massively Parallel Distributed Systems*, Basel. Birkhäuser Basel.

Wong, M., Ayguadé, E., Gottschlich, J., Luchangco, V., de Supinski, B. R., and Bihari, B. (2014). Towards transactional memory for OpenMP. In DeRose, L., de Supinski, B. R., Olivier, S. L., Chapman, B. M., and Müller, M. S., editors, *Using and Improving OpenMP for Devices, Tasks, and More*, pages 130–145, Cham. Springer International.